**Problem 1: Write a Python program to implement DDA(Digital Differential Analyser) line generation algorithm.**

**Code:**

# Python program for DDA line generation

from matplotlib import pyplot as plt

# DDA Function for line generation

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

print("CODE OUTPUT: ")

# 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

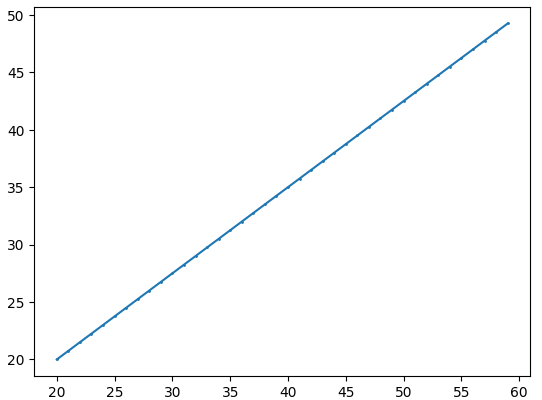
# coordinates of 2nd point

x1, y1 = 60, 50

# Function call

DDA(x0, y0, x1, y1)

**Output:**

****

**Problem 2- Write a Python program to implement Bresenham’s line generation algorithm.**

**Code:**

import matplotlib.pyplot as plt

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

"""

Bresenham's Line Generation Algorithm

Args:

x1 (int): x-coordinate of the starting point

y1 (int): y-coordinate of the starting point

x2 (int): x-coordinate of the ending point

y2 (int): y-coordinate of the ending point

Returns:

list: A list of points (x, y) representing the line

"""

points = []

dx = abs(x2 - x1)

dy = abs(y2 - y1)

x, y = x1, y1

sx = 1 if x1 < x2 else -1

sy = 1 if y1 < y2 else -1

err = dx - dy

while True:

points.append((x, y))

if x == x2 and y == y2:

break

e2 = 2 \* err

if e2 > -dy:

err -= dy

x += sx

if e2 < dx:

err += dx

y += sy

return points

# Example usage

x1, y1 = 15, 15

x2, y2 = 25, 25

points = bresenham(x1, y1, x2, y2)

import matplotlib.pyplot as plt

# ... (your bresenham function remains the same) ...

# Plotting the line with reduced thickness and anti-aliasing

x\_coords, y\_coords = zip(\*points)

plt.plot(x\_coords, y\_coords, marker='o', linestyle='-', linewidth=0.5, antialiased=True) # Modified

plt.xlabel("X-axis")

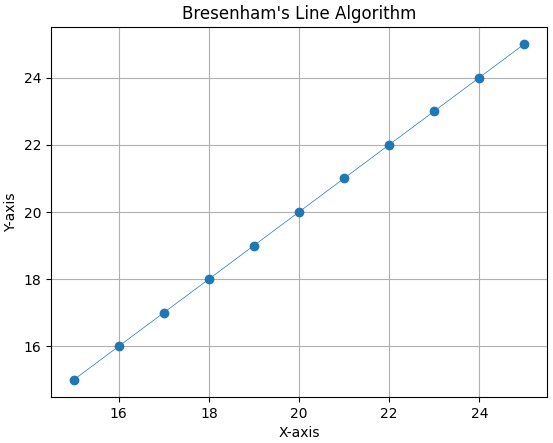
plt.ylabel("Y-axis")

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

plt.grid(True)

print("CODE OUTPUT: ")

plt.show()

**Output:**

**Problem 3: Write a Python program to implement Bresenham’s Circle generation algorithm.**

**Code:**

import matplotlib.pyplot as plt

# Function to put pixels at subsequence points

def drawCircle(xc, yc, x, y, ax):

ax.plot(xc + x, yc + y, 'ro')

ax.plot(xc - x, yc + y, 'ro')

ax.plot(xc + x, yc - y, 'ro')

ax.plot(xc - x, yc - y, 'ro')

ax.plot(xc + y, yc + x, 'ro')

ax.plot(xc - y, yc + x, 'ro')

ax.plot(xc + y, yc - x, 'ro')

ax.plot(xc - y, yc - x, 'ro')

# Function for circle generation using Bresenham's algorithm

def circleBres(xc, yc, r, ax):

x = 0

y = r

d = 3 - 2 \* r

drawCircle(xc, yc, x, y, ax)

while y >= x:

if d > 0:

y -= 1

d = d + 4 \* (x - y) + 10

else:

d = d + 4 \* x + 6

x += 1

drawCircle(xc, yc, x, y, ax)

# Main code to run the circle drawing

def main():

xc, yc = 200, 200 # Center of the circle

r = 100 # Radius of the circle

# Create a plot

fig, ax = plt.subplots(figsize=(9, 9))

ax.set\_aspect('equal')

# Set the limits of the plot

ax.set\_xlim(0, 400)

ax.set\_ylim(0, 400)

# Run the circle drawing function

circleBres(xc, yc, r, ax)

# Show the plot

plt.gca().invert\_yaxis() # To match the coordinate system

plt.grid()

print("CODE OUTPUT: ")

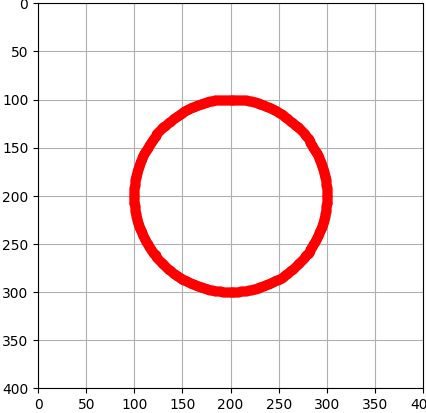
plt.show()

# Call the main function

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

main()

**Output:**

****

**Problem 4: Write a Python program to implement Midpoint Line generation algorithm.**

**Code:**

**import matplotlib.pyplot as plt**

# midPoint function for line generation

def midPoint(X1, Y1, X2, Y2):

# calculate dx & dy

dx = X2 - X1

dy = Y2 - Y1

# initial value of decision parameter d

d = 2 \* dy - dx

x = X1

y = Y1 # Changed Y to y for consistency

# Store points for plotting

x\_coords = [x]

y\_coords = [y]

# iterate through value of X

while (x < X2):

x = x + 1

# E or East is chosen

if (d < 0):

d = d + 2 \* dy

# NE or North East is chosen

else:

d = d + (dy - dx)

y = y + 1

# Store intermediate points

x\_coords.append(x)

y\_coords.append(y)

# Plot the line using matplotlib

plt.plot(x\_coords, y\_coords)

plt.xlabel("X-axis")

plt.ylabel("Y-axis")

plt.title("Midpoint Line Algorithm")

plt.grid(True)

print("CODE OUTPUT: ")

plt.show()

# Driver program

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

X1 = 3

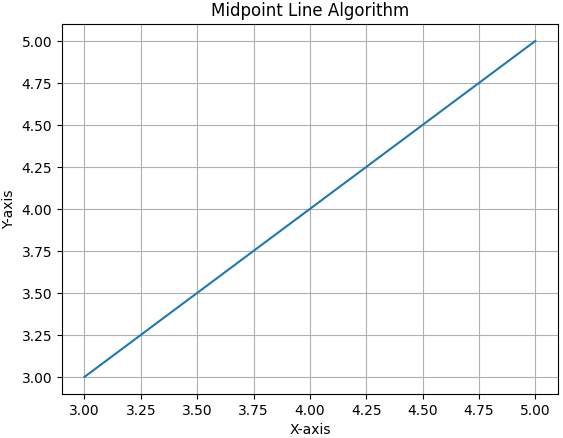
Y1 = 3

X2 = 5

Y2 = 5

midPoint(X1, Y1, X2, Y2)

**Output:**

****

**Problem 5: Write a Python program to implement Midpoint Circle generation algorithm.**

**Code:**

**import matplotlib.pyplot as plt**

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

x = r

y = 0

# Create lists to store the circle points

points = []

# Adding the initial points (the points on the axes after translation)

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

if r > 0:

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

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

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

# Initialize the value of P

P = 1 - r

# Plot points while x > y

while x > y:

y += 1

if P <= 0:

P = P + 2 \* y + 1

else:

x -= 1

P = P + 2 \* y - 2 \* x + 1

# If the point is generated, plot the points in all octants

if x < y:

break

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

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

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

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

if x != y:

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

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

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

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

return points

# Driver code

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

# To draw a circle of radius 3 centered at (0, 0)

points = midPointCircleDraw(0, 0, 3)

# Extract x and y coordinates for plotting

x\_vals = [point[0] for point in points]

y\_vals = [point[1] for point in points]

# Plotting the circle using matplotlib

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

plt.plot(x\_vals, y\_vals, 'ro', markersize=4)

plt.title("Mid-Point Circle Algorithm Visualization")

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

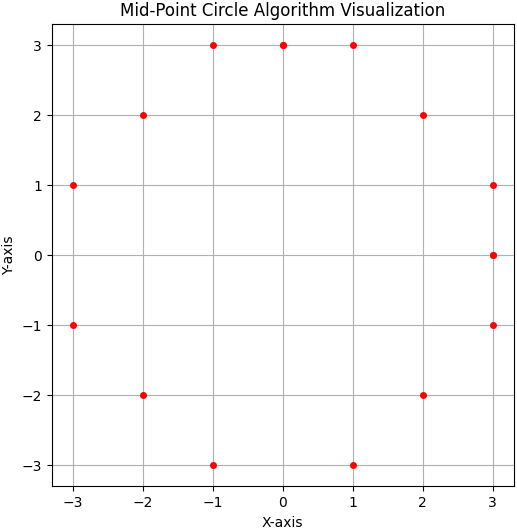
plt.xlabel("X-axis")

plt.ylabel("Y-axis")

plt.grid(True)

print("CODE OUTPUT: ")

plt.show()

**Output:**

**Problem 6: Write a Python program to implement Cohen-Sutherland Line Clipping algorithm.**

**Code:**

import matplotlib.pyplot as plt

# Define clipping window boundaries

xmin, xmax = 50, 80

ymin, ymax = 10, 40

INSIDE = 0

LEFT = 1

RIGHT = 2

BOTTOM = 4

TOP = 8

def compute\_outcode(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

def cohen\_sutherland\_line\_clip(x0, y0, x1, y1):

    outcode0 = compute\_outcode(x0, y0)

    outcode1 = compute\_outcode(x1, y1)

    accept = False

    while True:

        if not (outcode0 | outcode1):

            accept = True

            break

        elif outcode0 & outcode1:

            break

        else:

            outcode\_out = outcode1 if outcode1 > outcode0 else outcode0

            if outcode\_out & TOP:

                x = x0 + (x1 - x0) \* (ymax - y0) / (y1 - y0)

                y = ymax

            elif outcode\_out & BOTTOM:

                x = x0 + (x1 - x0) \* (ymin - y0) / (y1 - y0)

                y = ymin

            elif outcode\_out & RIGHT:

                y = y0 + (y1 - y0) \* (xmax - x0) / (x1 - x0)

                x = xmax

            elif outcode\_out & LEFT:

                y = y0 + (y1 - y0) \* (xmin - x0) / (x1 - x0)

                x = xmin

            if outcode\_out == outcode0:

                x0, y0 = x, y

                outcode0 = compute\_outcode(x0, y0)

            else:

                x1, y1 = x, y

                outcode1 = compute\_outcode(x1, y1)

    return accept, x0, y0, x1, y1

# Define line segment endpoints

x0, y0 = 70, 20

x1, y1 = 100, 10

# Clip the line segment

accept, x0\_clip, y0\_clip, x1\_clip, y1\_clip = cohen\_sutherland\_line\_clip(x0, y0, x1, y1)

# Plot the clipping window

plt.plot([xmin, xmax, xmax, xmin, xmin], [ymin, ymin, ymax, ymax, ymin], color='black')

# Plot the original line segment

plt.plot([x0, x1], [y0, y1], color='blue', label='Original Line')

# Plot the clipped line segment

if accept:

    plt.plot([x0\_clip, x1\_clip], [y0\_clip, y1\_clip], color='green', label='Clipped Line')

else:

    plt.text((x0 + x1) / 2, (y0 + y1) / 2, 'Line Rejected', color='red', fontsize=12)

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

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

plt.xlabel('X')

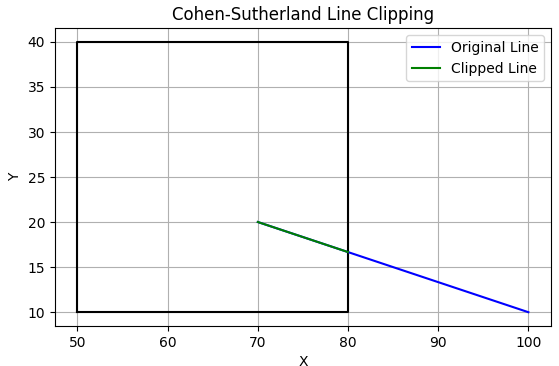
plt.ylabel('Y')

plt.legend()

plt.grid(True)

print("CODE OUTPUT: ")

plt.show()

**Output:**

**Problem 7: Write a Python program to implement Sutherland-Hodgeman Polygon Clipping Algorithm.**

**Code:**

import numpy as np

import matplotlib.pyplot as plt

# Defining maximum number of points in polygon

MAX\_POINTS = 20

# Function to return x-value of point of intersection 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 y-value of point of intersection 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 the edges w.r.t one clip edge of clipping area

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

    new\_points = np.zeros((MAX\_POINTS, 2), dtype=float)  # Changed to float for intersection points

    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]

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

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

        if i\_pos < 0 and k\_pos < 0:

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

            new\_poly\_size += 1

        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

        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

        else:

            pass

    clipped\_poly\_points = np.zeros((new\_poly\_size, 2), dtype=float) # Changed to float

    for i in range(new\_poly\_size):

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

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

# Function to implement 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 # Return the clipped polygon points and size

# Driver code

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

    # Defining polygon vertices in clockwise order

    poly\_size = 3

    poly\_points = np.array([[100,150], [200,250], [300,200]], dtype=float) # Changed to float

    # Defining clipper polygon vertices in clockwise order

    # 1st Example with square clipper

    clipper\_size = 4

    clipper\_points = np.array([[150,150], [150,200], [200,200], [200,150]], dtype=float) # Changed to float

    # 2nd Example with triangle clipper

    # clipper\_size = 3

    # clipper\_points = np.array([[100,300], [300,300], [200,100]], dtype=float) # Changed to float

    # Calling the clipping function

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

    # Plotting

    fig, ax = plt.subplots()

    # Plot original polygon

    x = np.append(poly\_points[:, 0], poly\_points[0, 0])  # Close the polygon

    y = np.append(poly\_points[:, 1], poly\_points[0, 1])

    ax.plot(x, y, 'b-', label='Original Polygon')

    # Plot clipper polygon

    x = np.append(clipper\_points[:, 0], clipper\_points[0, 0])

    y = np.append(clipper\_points[:, 1], clipper\_points[0, 1])

    ax.plot(x, y, 'r-', label='Clipper Polygon')

    # Plot clipped polygon

    x = np.append(clipped\_poly\_points[:clipped\_poly\_size, 0], clipped\_poly\_points[0, 0]) # Close the clipped polygon

    y = np.append(clipped\_poly\_points[:clipped\_poly\_size, 1], clipped\_poly\_points[0, 1])

    ax.plot(x, y, 'g-', label='Clipped Polygon')

    ax.set\_xlim(0, 400) # Set x-axis limits for better visualization

    ax.set\_ylim(0, 400) # Set y-axis limits

    ax.set\_xlabel('X-axis')

    ax.set\_ylabel('Y-axis')

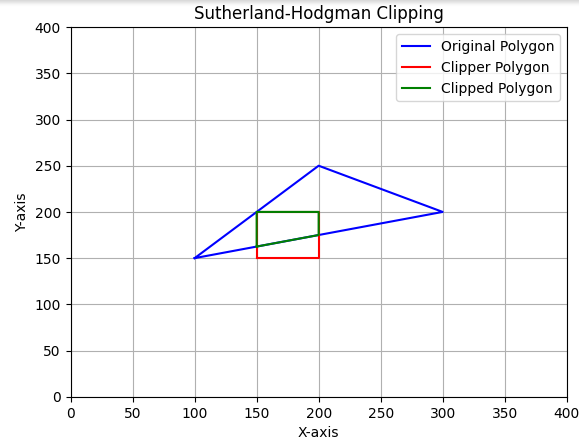
    ax.set\_title('Sutherland-Hodgman Clipping')

    ax.legend()

    plt.grid(True)

    print("CODE OUTPUT: ")

    plt.show()

**Output:**

**Problem 8: Write a Python Program to construct a 2D object (e.g. a Triangle) and apply basic geometric transformation, viz, Translation, Rotation and Scaling.**

**Code:**

import numpy as np

import matplotlib.pyplot as plt

def create\_triangle():

    return np.array([[0, 0], [1, 0], [0.5, 1], [0, 0]])  # Triangle vertices (closed loop)

def translation(shape, tx, ty):

    translation\_matrix = np.array([tx, ty])

    return shape + translation\_matrix

def rotation(shape, angle\_deg):

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

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

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

    return np.dot(shape, rotation\_matrix.T)

def scaling(shape, sx, sy):

    scaling\_matrix = np.array([[sx, 0], [0, sy]])

    return np.dot(shape, scaling\_matrix.T)

def plot\_shapes(original, translated, rotated, scaled):

    plt.figure(figsize=(6, 6))

    plt.plot(original[:, 0], original[:, 1], 'b-', label='Original')

    plt.plot(translated[:, 0], translated[:, 1], 'r-', label='Translated')

    plt.plot(rotated[:, 0], rotated[:, 1], 'g-', label='Rotated')

    plt.plot(scaled[:, 0], scaled[:, 1], 'm-', label='Scaled')

    plt.axhline(0, color='black',linewidth=0.5)

    plt.axvline(0, color='black',linewidth=0.5)

    plt.grid(color = 'gray', linestyle = '--', linewidth = 0.5)

    plt.legend()

    plt.title('Geometric Transformations')

    plt.axis('equal')

    plt.show()

def main():

    triangle = create\_triangle()

    # Apply transformations

    translated\_triangle = translation(triangle, 1, 1)  # Translate by (2, 3)

    rotated\_triangle = rotation(triangle, 45)          # Rotate by 45 degrees

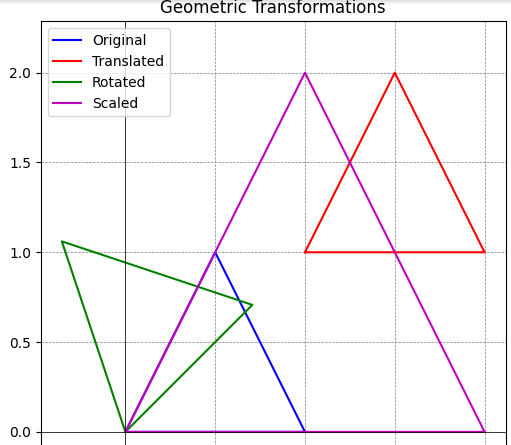
    scaled\_triangle = scaling(triangle, 2, 2)        # Scale by (2x, 0.5y)

    # Plot all shapes

    plot\_shapes(triangle, translated\_triangle, rotated\_triangle, scaled\_triangle)

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

    main()

**Output:**

**Problem 9: Write a Python program to construct a 3D object (e.g.) and apply basic geometric transformation viz. Translation, Rotation, Scaling, Sheering and Reflection.**

**Code:**

import numpy as np

import matplotlib.pyplot as plt

from mpl\_toolkits.mplot3d.art3d import Poly3DCollection

def create\_cube():

    return np.array([[0, 0, 0], [1, 0, 0], [1, 1, 0], [0, 1, 0],

                     [0, 0, 1], [1, 0, 1], [1, 1, 1], [0, 1, 1]])

def translation(shape, tx, ty, tz):

    return shape + np.array([tx, ty, tz])

def rotation\_x(shape, angle\_deg):

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

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

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

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

    return np.dot(shape, rotation\_matrix.T)

def scaling(shape, sx, sy, sz):

    scaling\_matrix = np.array([[sx, 0, 0], [0, sy, 0], [0, 0, sz]])

    return np.dot(shape, scaling\_matrix.T)

def shearing(shape, shxy, shxz, shyx, shyz, shzx, shzy):

    shear\_matrix = np.array([[1, shxy, shxz],

                             [shyx, 1, shyz],

                             [shzx, shzy, 1]])

    return np.dot(shape, shear\_matrix.T)

def reflection(shape, axis):

    reflection\_matrices = {

        'xy': np.array([[1, 0, 0], [0, 1, 0], [0, 0, -1]]),

        'xz': np.array([[1, 0, 0], [0, -1, 0], [0, 0, 1]]),

        'yz': np.array([[-1, 0, 0], [0, 1, 0], [0, 0, 1]])

    }

    return np.dot(shape, reflection\_matrices[axis].T)

def plot\_shapes(original, translated, rotated, scaled, sheared, reflected):

    fig = plt.figure(figsize=(5, 6))

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

    def draw\_cube(ax, shape, color, label):

        vertices = [[shape[0], shape[1], shape[2], shape[3]],

                    [shape[4], shape[5], shape[6], shape[7]],

                    [shape[0], shape[1], shape[5], shape[4]],

                    [shape[2], shape[3], shape[7], shape[6]],

                    [shape[1], shape[2], shape[6], shape[5]],

                    [shape[4], shape[7], shape[3], shape[0]]]

        ax.add\_collection3d(Poly3DCollection(vertices, facecolors=color, linewidths=1, edgecolors='k', alpha=0.6, label=label))

    draw\_cube(ax, original, 'blue', 'Original')

    draw\_cube(ax, translated, 'red', 'Translated')

    draw\_cube(ax, rotated, 'green', 'Rotated')

    draw\_cube(ax, scaled, 'purple', 'Scaled')

    draw\_cube(ax, sheared, 'orange', 'Sheared')

    draw\_cube(ax, reflected, 'cyan', 'Reflected')

    ax.set\_xlabel('X')

    ax.set\_ylabel('Y')

    ax.set\_zlabel('Z')

    ax.legend()

    plt.title('3D Geometric Transformations')

    plt.show()

def main():

    cube = create\_cube()

    # Apply transformations

    translated\_cube = translation(cube, 2, 1, 2)  # Translate by (2, 2, 2)

    rotated\_cube = rotation\_x(cube, 180)           # Rotate around X-axis by 45 degrees

    scaled\_cube = scaling(cube, 1,2,0.5)        # Scale by (1, 2, 0.5)

    sheared\_cube = shearing(cube, 1, 0, 0, 1, 0, 0)  # Shear along XY and YZ

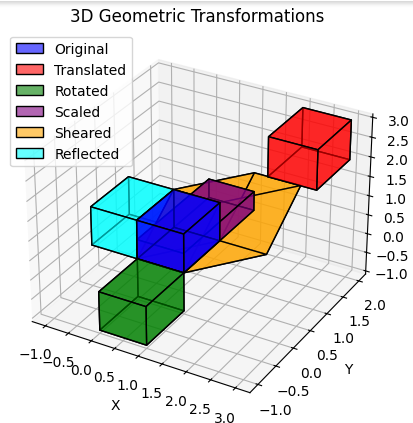
    reflected\_cube = reflection(cube, 'yz')       # Reflect across XY plane

    # Plot all shapes

    plot\_shapes(cube, translated\_cube, rotated\_cube, scaled\_cube, sheared\_cube, reflected\_cube)

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

    main()

**Output**