PRACTICAL FILE FOR CORE PAPER XIII: COMPUTER GRAPHICS

Name: Aryan Singh

Course: B.Sc.(H) Computer Science

College Roll No: CSC/21/32

University Roll No: 21059570002

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

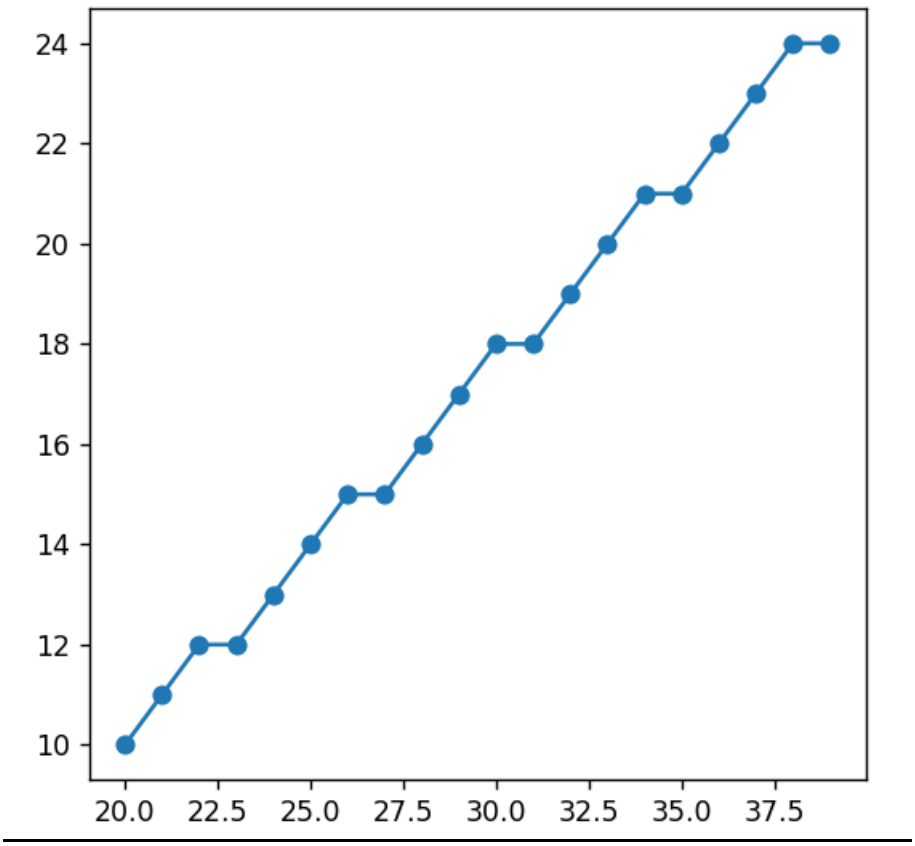
```
import matplotlib.pyplot as plt

def round(a):
    return int(a + 0.5)

def dda(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    steps = abs(dx) if abs(dx) > abs(dy) else abs(dy)
    Xinc = dx / steps
    Yinc = dy / steps
    x, y = x1, y1
    points = []
    for _ in range(steps):
        points.append((round(x), round(y)))
        x += Xinc
        y += Yinc
    return points

def bresenham(x1, y1, x2, y2):
    dx = abs(x2 - x1)
    dy = abs(y2 - y1)
    slope = dy / dx if dx != 0 else float('inf')
    x, y = x1, y1
    points = []
    if slope <= 1:
        p = 2 * dy - dx
        for _ in range(dx):
            points.append((x, y))
            x += 1
            if p < 0:
                p = p + 2 * dy
            else:
                y += 1
                p = p + 2 * dy - 2 * dx
    else:
        p = 2 * dx - dy
        for _ in range(dy):
            points.append((x, y))
            y += 1
            if p < 0:
                p = p + 2 * dx
```

```
            else:
                x += 1
                p = p + 2 * dx - 2 * dy
    return points

def plot_line(points):
    plt.figure(figsize=(5,5))
    plt.plot(*zip(*points), marker='o')
    plt.show()

if __name__ == "__main__":
    x1 = int(input("Enter x1: "))
    y1 = int(input("Enter y1: "))
    x2 = int(input("Enter x2: "))
    y2 = int(input("Enter y2: "))
    print("DDA:")
    points = dda(x1, y1, x2, y2)
    plot_line(points)
    print("Bresenham:")
    points = bresenham(x1, y1, x2, y2)
    plot_line(points)
```

INPUT

```
llege stuff/cg graphics/P01.py"
Enter x1: 20
Enter y1: 10
Enter x2: 40
Enter y2: 25
DDA:
```

OUTPUT



2. Write a program to implement mid-point circle drawing algorithm.

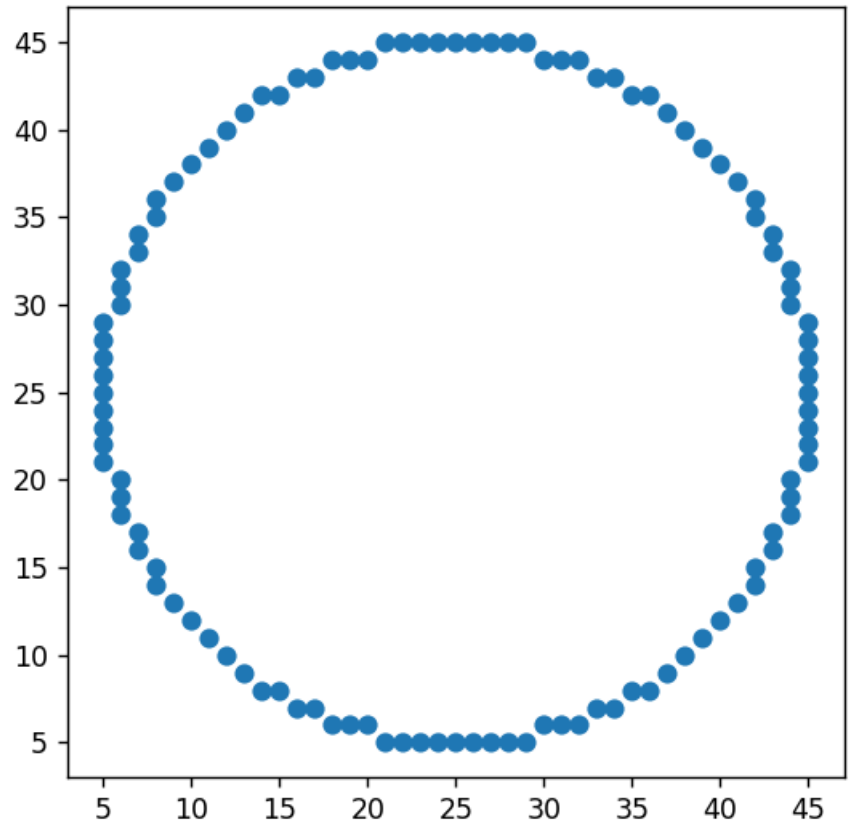
```
import matplotlib.pyplot as plt
import numpy as np

def mid_point_circle_draw(x_centre, y_centre, r):
    x = r
    y = 0
    points = []
    # Printing the initial point on the axes after translation
    points.append((x + x_centre, y + y_centre))
    # When radius is zero, only a single point is printed
    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))
    # Initialising the value of P
    P = 1 - r
    while x > y:
        y += 1
        # Mid-point inside or on the perimeter
        if P <= 0:
            P = P + 2 * y + 1
        # Mid-point outside the perimeter
        else:
            x -= 1
            P = P + 2 * y - 2 * x + 1
        # All the perimeter points have already been printed
        if x < y:
            break
        # Printing the generated point its reflection in the other octants
after translation
        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 the generated point is on the line x = y then the perimeter
points have already been printed
        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
```

```
def plot_circle(points):
    plt.figure(figsize=(5,5))
    plt.scatter(*zip(*points), marker='o')
    plt.show()

if __name__ == "__main__":
    x_centre = int(input("Enter x_centre: "))
    y_centre = int(input("Enter y_centre: "))
    r = int(input("Enter radius: "))
    points = mid_point_circle_draw(x_centre, y_centre, r)
    plot_circle(points)
```

INPUT

```
llege stuff/cg graphics/P02.py"
Enter x_centre: 25
Enter y_centre: 25
Enter radius: 20
```

OUTPUT



3. Write a program to clip a line using Cohen and Sutherland line clipping algorithm.

```
import matplotlib.pyplot as plt
# Defining region codes
INSIDE = 0  # 0000
LEFT = 1    # 0001
RIGHT = 2   # 0010
BOTTOM = 4  # 0100
TOP = 8     # 1000

# Function to compute region code for a point(x, y)
def compute_code(x, y, x_min, y_min, x_max, y_max):
    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

# Implementing Cohen-Sutherland algorithm
# Clipping a line from P1 = (x1, y1) to P2 = (x2, y2)
def cohen_sutherland(x1, y1, x2, y2, x_min, y_min, x_max, y_max):
    # Compute region codes for P1, P2
    code1 = compute_code(x1, y1, x_min, y_min, x_max, y_max)
    code2 = compute_code(x2, y2, x_min, y_min, x_max, y_max)
    accept = False
    while True:
        # If both endpoints lie within rectangle
        if code1 == 0 and code2 == 0:
            accept = True
            break
        # If both endpoints are outside rectangle
        elif (code1 & code2) != 0:
            break
        else:
            # Some segment of line lies within the rectangle
            # At least one endpoint is outside the rectangle, pick it.
            code_out = code1 if code1 != 0 else code2
            # Find intersection point
            if code_out & TOP:
```

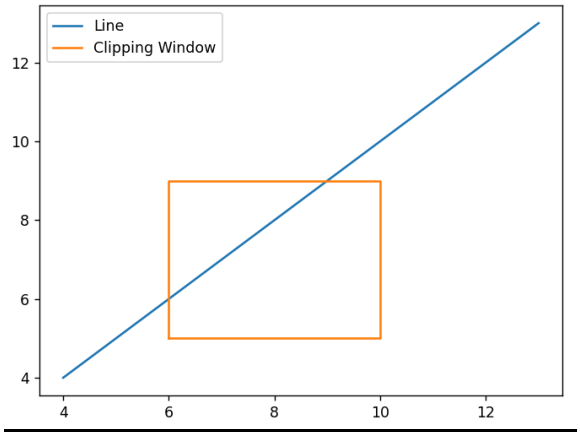
```
                # point is above the clip rectangle
                x = x1 + (x2 - x1) * (y_max - y1) / (y2 - y1)
                y = y_max
            elif code_out & BOTTOM:
                # point is below the rectangle
                x = x1 + (x2 - x1) * (y_min - y1) / (y2 - y1)
                y = y_min
            elif code_out & RIGHT:
                # point is to the right of rectangle
                y = y1 + (y2 - y1) * (x_max - x1) / (x2 - x1)
                x = x_max
            elif code_out & LEFT:
                # point is to the left of rectangle
                y = y1 + (y2 - y1) * (x_min - x1) / (x2 - x1)
                x = x_min
            # Now intersection point x, y is found
            # We replace point outside rectangle by intersection point
            if code_out == code1:
                x1, y1 = x, y
                code1 = compute_code(x1, y1, x_min, y_min, x_max, y_max)
            else:
                x2, y2 = x, y
                code2 = compute_code(x2, y2, x_min, y_min, x_max, y_max)
    if accept:
        return ((x1, y1), (x2, y2))
    else:
        return None


def plot_line(x1, y1, x2, y2, x_min, y_min, x_max, y_max):
    plt.figure()
    plt.plot([x1, x2], [y1, y2], label='Line')
    plt.plot([x_min, x_max, x_max, x_min, x_min], [y_min, y_min, y_max, y_max,
y_min], label='Clipping Window')
    plt.legend()
    plt.show()

if __name__ == "__main__":
    x1 = int(input("Enter x1: "))
    y1 = int(input("Enter y1: "))
    x2 = int(input("Enter x2: "))
    y2 = int(input("Enter y2: "))
    x_min = int(input("Enter x_min: "))
    y_min = int(input("Enter y_min: "))
    x_max = int(input("Enter x_max: "))
    y_max = int(input("Enter y_max: "))
    result = cohen_sutherland(x1, y1, x2, y2, x_min, y_min, x_max, y_max)
    if result is not None:
```

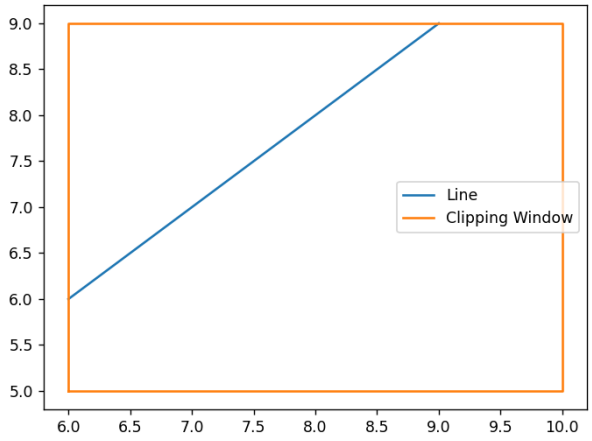
```
        print(f"The line from ({x1}, {y1}) to ({x2}, {y2}) clips to
{result}.")
        plot_line(x1, y1, x2, y2, x_min, y_min, x_max, y_max)
        plot_line(result[0][0], result[0][1], result[1][0], result[1][1],
x_min, y_min, x_max, y_max)

    else:
        print("The line doesn't lie within the rectangle.")
```

INPUT

```
Enter x1: 4
Enter y1: 4
Enter x2: 13
Enter y2: 13
Enter x_min: 6
Enter y_min: 5
Enter x_max: 10
Enter y_max: 9
The line from (4, 4) to (13, 13) clips to ((6, 6.0), (9.0, 9)).
```

OUTPUT





4. Write a program to clip a polygon using Sutherland Hodgeman algorithm.

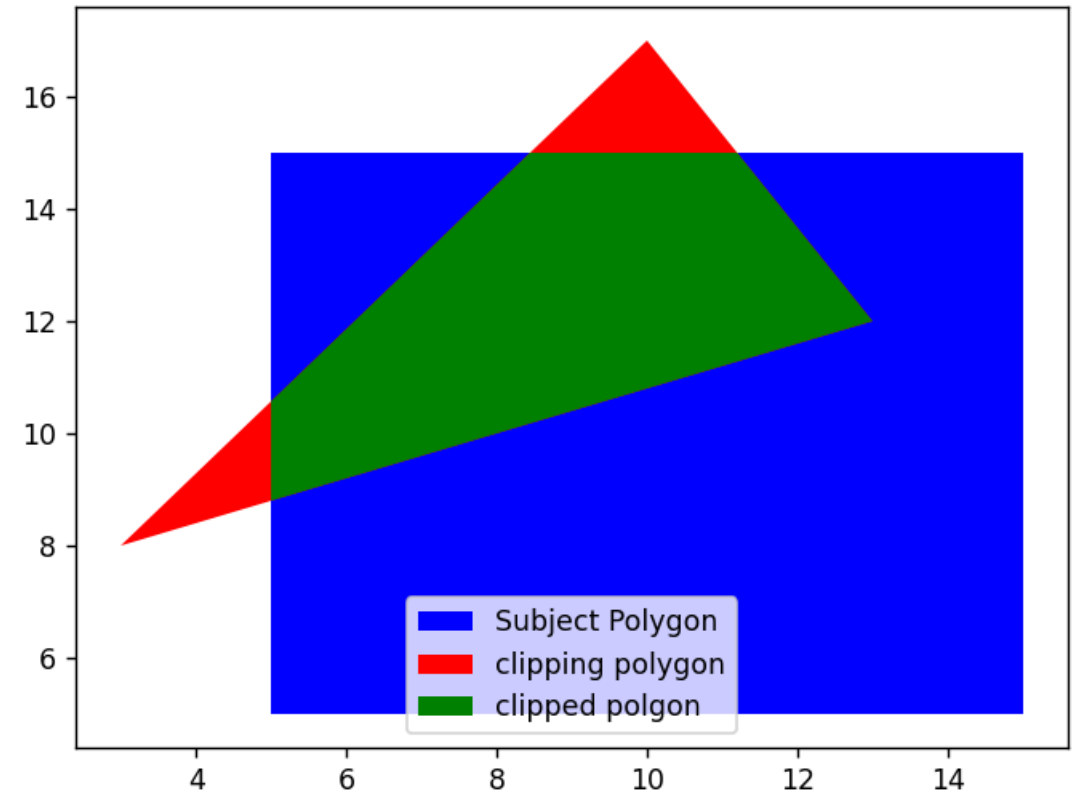
```
import matplotlib.pyplot as plt
import matplotlib.patches as patches
import numpy as np
import warnings

# POINTS NEED TO BE PRESENTED CLOCKWISE OR ELSE THIS WONT WORK

class PolygonClipper:

    def __init__(self,warn_if_empty=True):
        self.warn_if_empty = warn_if_empty

    def is_inside(self,p1,p2,q):
        R = (p2[0] - p1[0]) * (q[1] - p1[1]) - (p2[1] - p1[1]) * (q[0] -
p1[0])
        if R <= 0:
            return True
        else:
            return False

    def compute_intersection(self,p1,p2,p3,p4):
        # if first line is vertical
        if p2[0] - p1[0] == 0:
            x = p1[0]

            # slope and intercept of second line
            m2 = (p4[1] - p3[1]) / (p4[0] - p3[0])
            b2 = p3[1] - m2 * p3[0]

            # y-coordinate of intersection
            y = m2 * x + b2

        # if second line is vertical
        elif p4[0] - p3[0] == 0:
            x = p3[0]

            # slope and intercept of first line
            m1 = (p2[1] - p1[1]) / (p2[0] - p1[0])
            b1 = p1[1] - m1 * p1[0]

            # y-coordinate of intersection
            y = m1 * x + b1
```

```
        # if neither line is vertical
        else:
            m1 = (p2[1] - p1[1]) / (p2[0] - p1[0])
            b1 = p1[1] - m1 * p1[0]

            # slope and intercept of second line
            m2 = (p4[1] - p3[1]) / (p4[0] - p3[0])
            b2 = p3[1] - m2 * p3[0]

            # x-coordinate of intersection
            x = (b2 - b1) / (m1 - m2)

            # y-coordinate of intersection
            y = m1 * x + b1

        intersection = (x,y)

        return intersection

    def clip(self,subject_polygon,clipping_polygon):

        final_polygon = subject_polygon.copy()

        for i in range(len(clipping_polygon)):

            # stores the vertices of the next iteration of the clipping
procedure
            next_polygon = final_polygon.copy()

            # stores the vertices of the final clipped polygon
            final_polygon = []

            # these two vertices define a line segment (edge) in the clipping
            # polygon. It is assumed that indices wrap around, such that if
            # i = 1, then i - 1 = K.
            c_edge_start = clipping_polygon[i - 1]
            c_edge_end = clipping_polygon[i]

            for j in range(len(next_polygon)):

                # these two vertices define a line segment (edge) in the
subject
                # polygon
                s_edge_start = next_polygon[j - 1]
                s_edge_end = next_polygon[j]

                if self.is_inside(c_edge_start,c_edge_end,s_edge_end):
```

```
                    if not
self.is_inside(c_edge_start,c_edge_end,s_edge_start):
                        intersection =
self.compute_intersection(s_edge_start,s_edge_end,c_edge_start,c_edge_end)
                        final_polygon.append(intersection)
                    final_polygon.append(tuple(s_edge_end))
                elif self.is_inside(c_edge_start,c_edge_end,s_edge_start):
                    intersection =
self.compute_intersection(s_edge_start,s_edge_end,c_edge_start,c_edge_end)
                    final_polygon.append(intersection)

        return np.asarray(final_polygon)

    def __call__(self,A,B):
        clipped_polygon = self.clip(A,B)
        if len(clipped_polygon) == 0 and self.warn_if_empty:
            warnings.warn("No intersections found. Are you sure your \
                           polygon coordinates are in clockwise order?")

        return clipped_polygon

if __name__ == '__main__':

    # some test polygons

    clip = PolygonClipper()

#     subject_polygon = [(0,3),(0.5,0.5),(3,0),(0.5,-0.5),(0,-3),(-0.5,-
0.5),(-3,0),(-0.5,0.5)]
#     clipping_polygon = [(-2,-2),(-2,2),(2,2),(2,-2)]
    subject_polygon = [(5,5),(15,5),(15,15),(5,15)]
    clipping_polygon = [(10,17),(13,12),(3,8)]

    # star and triangle
    # subject_polygon = [(0,3),(0.5,0.5),(3,0),(0.5,-0.5),(0,-3),(-0.5,-
0.5),(-3,0),(-0.5,0.5)]
    # clipping_polygon = [(0,2),(2,-2),(-2,-2)]

    subject_polygon = np.array(subject_polygon)
    clipping_polygon = np.array(clipping_polygon)
    clipped_polygon = clip(subject_polygon,clipping_polygon)
    fig, ax = plt.subplots()
    xs, ys = zip(*subject_polygon)# create lists of x and y values
    plt.fill(xs, ys, 'b',label='Subject Polygon')

    xs, ys = zip(*clipping_polygon)# create lists of x and y values
    plt.fill(xs, ys, 'r',label='clipping polygon')
```

```
    xs, ys = zip(*clipped_polygon) # create lists of x and y values
    plt.fill(xs, ys, 'g',label='clipped polgon')
    plt.legend()
    plt.show()
    print(clipped_polygon)
```

OUTPUT



5. Write a program to fill a polygon using Scan line fill algorithm.

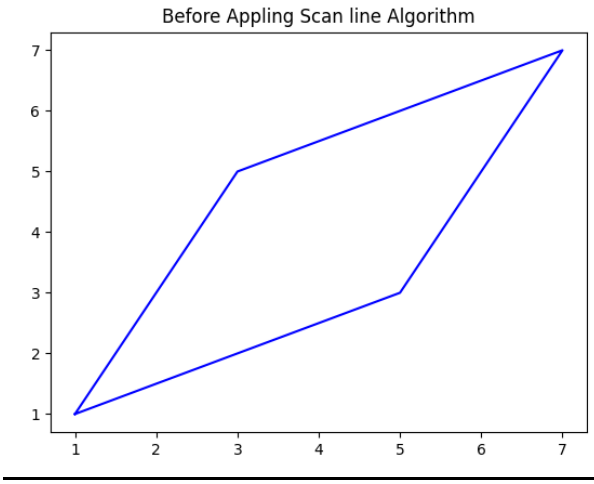
```
import matplotlib.pyplot as plt
from graphics import *
def draw_graph(V):
    x=[]
    y=[]
    for i in V:
        x.append(i[0])
        y.append(i[1])
    plt.title("Before Appling Scan line Algorithm")
    plt.plot(x,y,c="blue",label="Polygon")
    plt.show()
    plt.plot(x,y,c="blue",label="Polygon")
    return max(y),min(y),max(x),min(x)
def get_range(V):
    y2,y1,x2,x1=draw_graph(V)
    x=[]
    y=[]
    for i in range(len(V)-1):
        x1=V[i][0]
        y1=V[i][1]
        x2=V[i+1][0]
        y2=V[i+1][1]
        x3,y3=calpoints(x1,y1,x2,y2,x,y)
    x=x3
    y=y3
    for i in range(int(len(y)/2)):
        if i!= int(len(y)/2) and i!= 0:
            plt.plot([x[i],x[y.index(y[i],i+1)]],[y[i],y[i]],c="red")
    plt.title("After Appling Scan Line algorithm")
    plt.show()
def calpoints(x1,y1,x2,y2,xlist,ylist):
    m=(y2-y1)/(x2-x1)
    c=y1-(m*x1)
    for i in range(abs(y2-y1)*10):
        if x1<=x2:
            y=(i/10)+y1
            x=(y-c)/m
            xlist.append(x)
            ylist.append(y)
        else:
            y=y1-(i/10)
            x=(y-c)/m
```

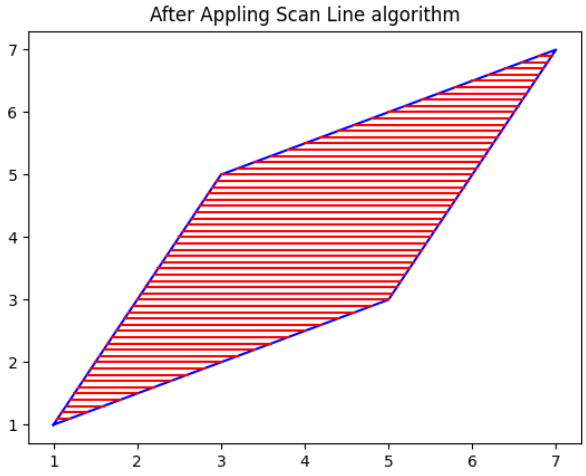
```
            xlist.append(x)
            ylist.append(y)
#     print(xlist,ylist)
    return xlist,ylist

count=int(input("Enter Number of vertices in a Polygon :- "))
V=[]
print("Start entering Points :- ")
for i in range(count):
    print("Enter Cordinates of ",i+1," Vertex :- ")
    x,y=input().split()
    V.append((int(x),int(y)))
V.append((V[0][0],V[0][1]))
get_range(V)
```

INPUT

```
Enter Number of vertices in a Polygon :- 4
Start entering Points :-
Enter Cordinates of  1  Vertex :-
1 1
Enter Cordinates of  2  Vertex :-
3 5
Enter Cordinates of  3  Vertex :-
7 7
Enter Cordinates of  4  Vertex :-
5 3
```

OUTPUT





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

```
import numpy as np

def translate(point, tx, ty):
    transformation_matrix = np.array([[1, 0, tx],
                                      [0, 1, ty],
                                      [0, 0, 1]])
    return np.dot(transformation_matrix, point)

def rotate(point, theta):
    theta = np.radians(theta)
    transformation_matrix = np.array([[np.cos(theta), -np.sin(theta), 0],
                                      [np.sin(theta),  np.cos(theta), 0],
                                      [0, 0, 1]])
    return np.dot(transformation_matrix, point)

def scale(point, sx, sy):
    transformation_matrix = np.array([[sx, 0, 0],
                                      [0, sy, 0],
                                      [0, 0, 1]])
    return np.dot(transformation_matrix, point)

# Example usage:
pointT = np.array([0, 0, 1])  # Homogeneous coordinates
pointR = np.array([4,3,1])
pointS = np.array([2,2,1])

point = translate(pointT, 2, 2)
print("After translation: ", point)

point = rotate(pointR, 45)
print("After rotation: ", point)

point = scale(pointS, 1.5, 0.5)
print("After scaling: ", point)
```

OUTPUT

```
After translation:  [2 2 1]
After rotation:  [0.70710678 4.94974747 1.        ]
After scaling:  [3. 1. 1.]
```

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

```
import numpy as np

def translate(point, tx, ty, tz):
    transformation_matrix = np.array([[1, 0, 0, tx],
                                      [0, 1, 0, ty],
                                      [0, 0, 1, tz],
                                      [0, 0, 0, 1]])
    return np.dot(transformation_matrix, point)

def rotate(point, ax, ay, az):
    ax, ay, az = np.radians(ax), np.radians(ay), np.radians(az)
    Rx = np.array([[1, 0, 0, 0],
                   [0, np.cos(ax), -np.sin(ax), 0],
                   [0, np.sin(ax), np.cos(ax), 0],
                   [0, 0, 0, 1]])
    Ry = np.array([[np.cos(ay), 0, np.sin(ay), 0],
                   [0, 1, 0, 0],
                   [-np.sin(ay), 0, np.cos(ay), 0],
                   [0, 0, 0, 1]])
    Rz = np.array([[np.cos(az), -np.sin(az), 0, 0],
                   [np.sin(az), np.cos(az), 0, 0],
                   [0, 0, 1, 0],
                   [0, 0, 0, 1]])
    R = np.dot(Rz, np.dot(Ry, Rx))
    return np.dot(R, point)

def scale(point, sx, sy, sz):
    transformation_matrix = np.array([[sx, 0, 0, 0],
                                      [0, sy, 0, 0],
                                      [0, 0, sz, 0],
                                      [0, 0, 0, 1]])
    return np.dot(transformation_matrix, point)

def parallel_project(point):
    projection_matrix = np.array([[1, 0, 0, 0],
                                  [0, 1, 0, 0],
                                  [0, 0, 0, 0],
                                  [0, 0, 0, 1]])
    return np.dot(projection_matrix, point)

def perspective_project(point, d):
```

```
    projection_matrix = np.array([[1, 0, 0, 0],
                                  [0, 1, 0, 0],
                                  [0, 0, 1, -1/d],
                                  [0, 0, 0, 1]])
    return np.dot(projection_matrix, point)

# Example usage:
point = np.array([1, 2, 3, 1])  # Homogeneous coordinates

point = translate(point, 2, 3, 4)
print("After translation: ", point)

point = rotate(point, 45, 45, 45)
print("After rotation: ", point)

point = scale(point, 2, 2, 2)
print("After scaling: ", point)

point = parallel_project(point)
print("After parallel projection: ", point)

point = perspective_project(point, 1)
print("After perspective projection: ", point)
```

OUTPUT

```
After translation:  [3 5 7 1]
After rotation:  [6.74264069 4.74264069 3.87867966 1.        ]
After scaling:  [13.48528137  9.48528137  7.75735931  1.        ]
After parallel projection:  [13.48528137  9.48528137  0.          1.        ]
After perspective projection:  [13.48528137  9.48528137 -1.          1.        ]
```

8. Write a program to draw Hermite /Bezier curve.

```
import numpy as np
import matplotlib.pyplot as plt
import scipy as scipy

def bezier_curve(points, t):
    n = len(points)
    B = np.zeros(2)
    for i in range(n):
        B += scipy.special.comb(n-1, i) * ((1-t)**(n-1-i)) * (t**i) *
points[i]
    return B

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

t = np.linspace(0, 1, num=1000)
curve = np.array([bezier_curve(points, i) for i in t])

plt.plot(curve[:, 0], curve[:, 1])
plt.show()
```

OUTPUT