**LAB-2**

**Midpoint Circle Drawing Algorithm**

**Objective: -**

* Efficiently determine the pixel positions required to draw a circle by avoiding floating-point calculations and using integer arithmetic.
* To exploit the symmetry of a circle, reducing the computation required to draw an entire circle to calculating points for only one-eighth of it.

**Theory: -**

The Midpoint Circle Drawing Algorithm efficiently plots a circle using integer arithmetic and symmetry principles. By calculating only one-eighth of the circle, it reduces redundancy and computational overhead. The algorithm determines pixel positions incrementally, ensuring smooth curves and uniformity while avoiding complex mathematical operations like trigonometry or floating-point calculations.

**Algorithm: -**

1. Initialize Variables:

a. Set the initial coordinates (x, y) as (0, radius).

b. Define the initial decision parameter (d) as 1 - radius.

c. Prepare a list or storage structure to store the calculated points.

2. Plot Initial Points:

a. Use the symmetry of the circle to plot the initial points in all eight octants.

b. For a circle centered at (0, 0), the octant points are:

(x, y), (y, x), (-x, y), (-y, x), (-x, -y), (-y, -x), (x, -y), (y, -x).

3. Iterate While x < y:

a. Evaluate the decision parameter (d):

i. If d < 0, the midpoint is inside the circle, so update d as d + 2 \* x + 3.

ii. If d ≥ 0, the midpoint is outside or on the circle, so:

- Update d as d + 2 \* (x - y) + 5.

- Decrement y by 1.

b. Increment x by 1.

4. Plot New Points:

a. Using the updated (x, y) values, calculate and plot the points for all eight octants using symmetry.

5. Repeat Steps 3-4 Until x >= y:

a. Continue updating (x, y) and plotting points until x equals or surpasses y.

6. Complete the Circle:

* a. Ensure all calculated points are plotted, forming a complete circle.

**Code: -**

import matplotlib.pyplot as plt

def MCA():

    def SP(x,y):

        xes.append(x)

        yes.append(y)

        xes.extend([x, -x, -x, x, y, -y, -y, y])

        yes.extend([y, y, -y, -y, x, x, -x, -x])

    r=int(input('Enter the radius: '))

    xc=int(input('Enter the x coord of center: '))

    yc=int(input('Enter the y coord of center: '))

    x,y=0,r

    p=1-r

    xes=[]

    yes=[]

    SP(x,y)

    while(x<y):

        x=x+1

        if(p<0):

            p=p+2\*x+1

        else:

            y=y-1

            p=p+2\*(x-y)+1

        SP(x,y)

    xes = [xc + xi for xi in xes]

    yes = [yc + yi for yi in yes]

    plt.scatter(xes, yes, color='blue', s=10)

    plt.title("Midpoint Circle Algorithm")

    plt.xlabel("X")

    plt.ylabel("Y")

    plt.grid()

    plt.show()

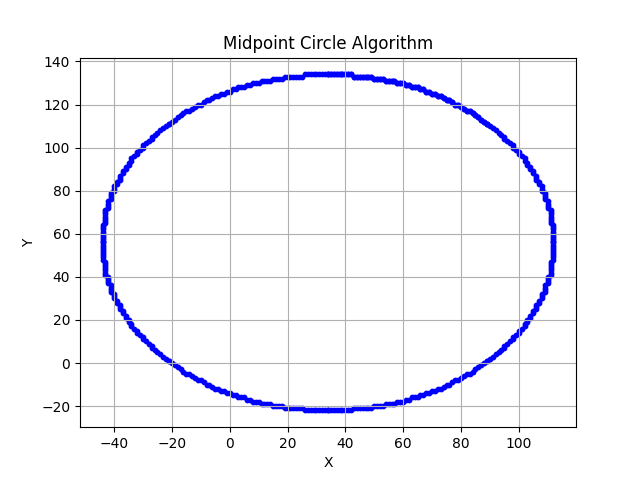
MCA()

**Output: -**

Enter the radius: 78

Enter the x coord of center: 34

Enter the y coord of center: 56



**Discussion**

The Midpoint Circle Drawing Algorithm is highly efficient due to its reliance on integer calculations and symmetry. By calculating only one-eighth of the circle's points and reflecting them, the algorithm minimizes redundant computations. This makes it an ideal choice for applications requiring real-time rendering of geometric shapes. The incremental nature of the decision parameter eliminates the need for complex mathematical operations, such as trigonometry, ensuring speed and precision.

Despite its advantages, the algorithm is best suited for drawing perfect circles. Adapting it to other shapes, such as ellipses, requires modifications, which can increase complexity. Moreover, the reliance on symmetry means it is limited to symmetrical objects, making it less versatile for non-standard curves or irregular shapes.

**Conclusion**

The Midpoint Circle Drawing Algorithm remains a cornerstone in computer graphics for rendering circles efficiently. Its use of symmetry and integer-based calculations ensures precise and smooth rendering on raster displays. While it has some limitations, its simplicity and computational efficiency make it indispensable for applications in graphics and visual computing.