

B.TECH(CS)

COMPUTER GRAPHICS RCS-603

UNIT-1

Scan Conversion Definition

It is a process of representing graphics objects a collection of pixels. The graphics objects are continuous. The pixels used are discrete. Each pixel can have either on or off state.

The circuitry of the video display device of the computer is capable of converting binary values (0, 1) into a pixel on and pixel off information. 0 is represented by pixel off. 1 is represented using pixel on. Using this ability graphics computer represent picture having discrete dots.

Any model of graphics can be reproduced with a dense matrix of dots or points. Most human beings think graphics objects as points, lines, circles, ellipses. For generating graphical object, many algorithms have been developed.

Advantage of developing algorithms for scan conversion

1. Algorithms can generate graphics objects at a faster rate.
2. Using algorithms memory can be used efficiently.
3. Algorithms can develop a higher level of graphical objects.

Examples of objects which can be scan converted

1. Point
2. Line
3. Sector
4. Arc
5. Ellipse
6. Rectangle
7. Polygon
8. Characters
9. Filled Regions

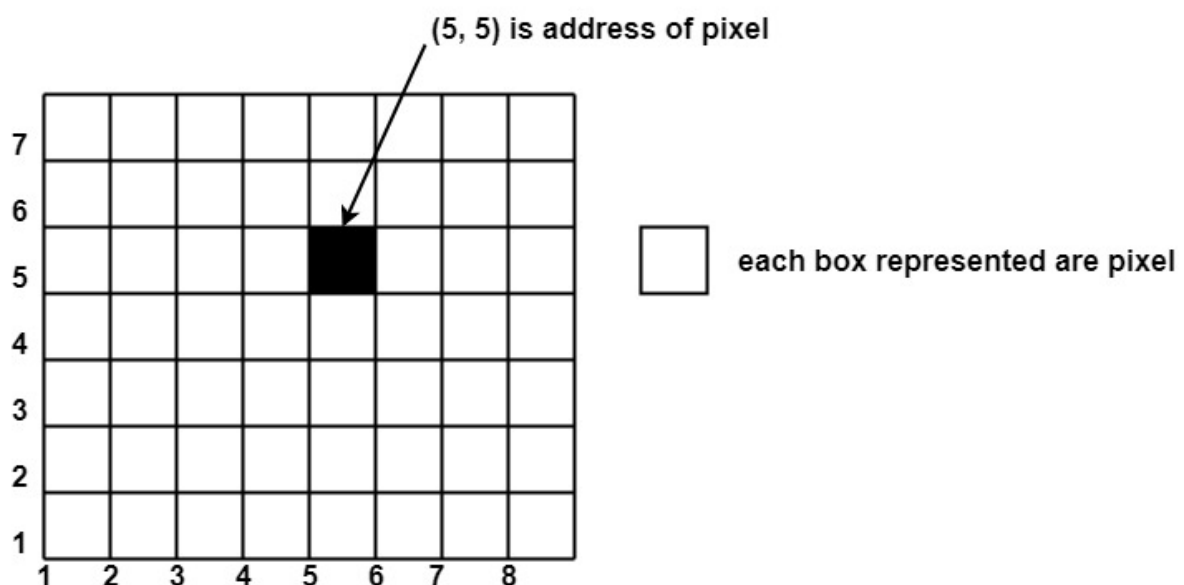
The process of converting is also called as rasterization. The algorithms implementation varies from one computer system to another computer system. Some algorithms are implemented using the software. Some are performed using hardware or firmware. Some are performed using various combinations of hardware, firmware, and software.

Pixel or Pel:

The term pixel is a short form of the picture element. It is also called a point or dot. It is the smallest picture unit accepted by display devices. A picture is constructed from hundreds of such pixels. Pixels are generated using commands. Lines, circle, arcs, characters; curves are drawn with closely spaced pixels. To display the digit or letter matrix of pixels is used.

The closer the dots or pixels are, the better will be the quality of picture. Closer the dots are, crisper will be the picture. Picture will not appear jagged and unclear if pixels are closely spaced. So the quality of the picture is directly proportional to the density of pixels on the screen.

Pixels are also defined as the smallest addressable unit or element of the screen. Each pixel can be assigned an address as shown in fig:



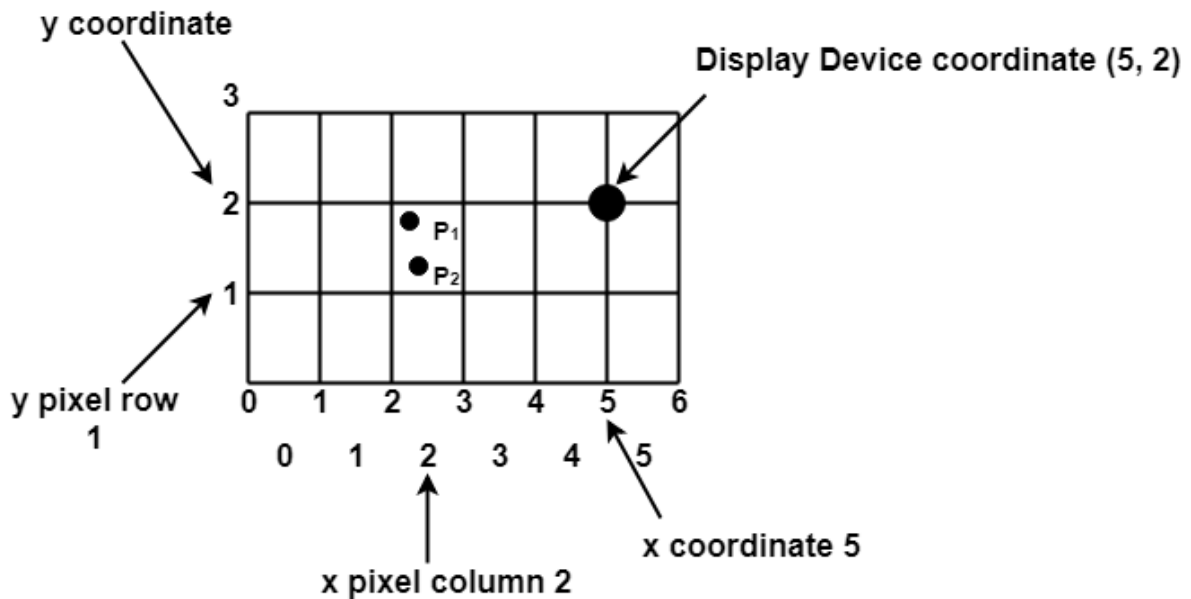
Different graphics objects can be generated by setting the different intensity of pixels and different colors of pixels. Each pixel has some co-ordinate value. The coordinate is represented using row and column.

P (5, 5) used to represent a pixel in the 5th row and the 5th column. Each pixel has some intensity value which is represented in memory of computer called a **frame buffer**. Frame Buffer is also called a refresh buffer. This memory is a storage area for storing pixels values using which pictures are displayed. It is also called as digital memory. Inside the buffer, image is stored as a pattern of binary digits either 0 or 1. So there is an array of 0 or 1 used to represent the picture. In black and white monitors, black pixels are represented using 1's and white pixels are represented using 0's. In case of systems having one bit per pixel frame buffer is called a bitmap. In systems with multiple bits per pixel it is called a pixmap.

Scan Converting a Point

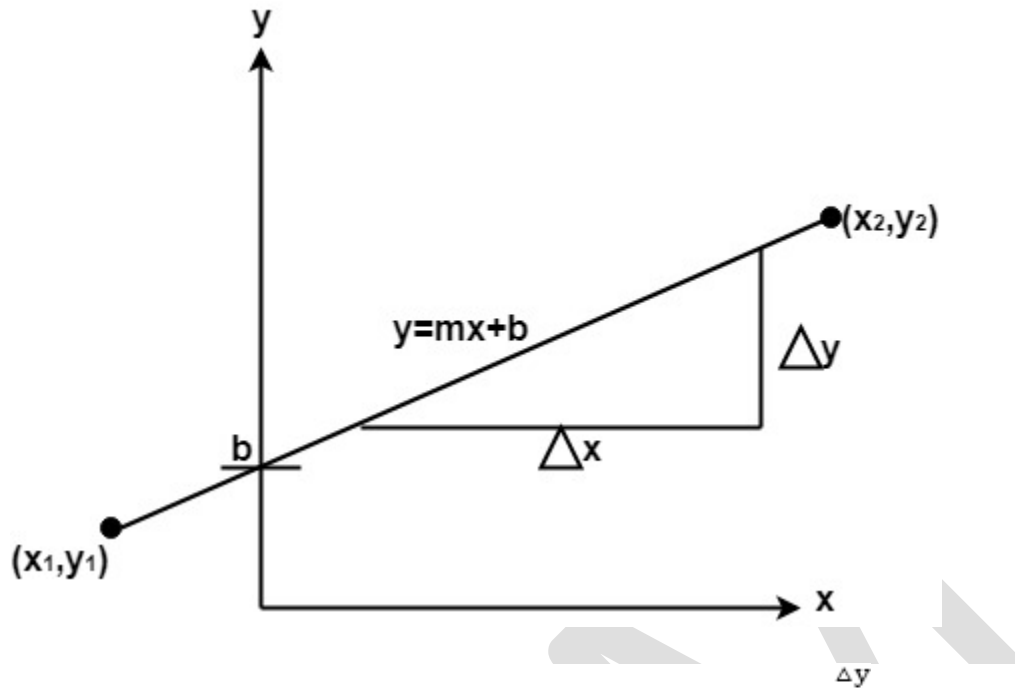
Each pixel on the graphics display does not represent a mathematical point. Instead, it means a region which theoretically can contain an infinite number of points. Scan-Converting a point involves illuminating the pixel that contains the point.

Example: Display coordinates points $P_1(2\frac{1}{4}, 1\frac{3}{4})$ & $P_2(2\frac{2}{3}, 1\frac{1}{4})$ as shown in fig would both be represented by pixel (2, 1). In general, a point $p(x, y)$ is represented by the integer part of x & the integer part of y that is pixels $[(\text{INT}(x), \text{INT}(y))]$.



Scan Converting a Straight Line

A straight line may be defined by two endpoints & an equation. In fig the two endpoints are described by (x_1, y_1) and (x_2, y_2) . The equation of the line is used to determine the x, y coordinates of all the points that lie between these two endpoints.



Using the equation of a straight line, $y = mx + b$ where $m = \frac{\Delta y}{\Delta x}$ & b = the y intercept, we can find values of y by incrementing x from $x = x_1$, to $x = x_2$. By scan-converting these calculated x, y values, we represent the line as a sequence of pixels.

Properties of Good Line Drawing Algorithm:

1. Line should appear Straight: We must appropriate the line by choosing addressable points close to it. If we choose well, the line will appear straight, if not, we shall produce crossed lines.

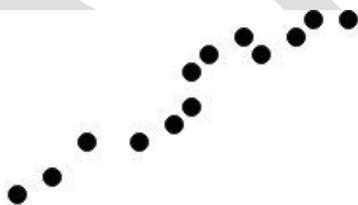


Fig: O/P from a poor line generating algorithm

The lines must be generated parallel or at 45° to the x and y -axes. Other lines cause a problem: a line segment through it starts and finishes at addressable points, may happen to pass through no another addressable points in between.

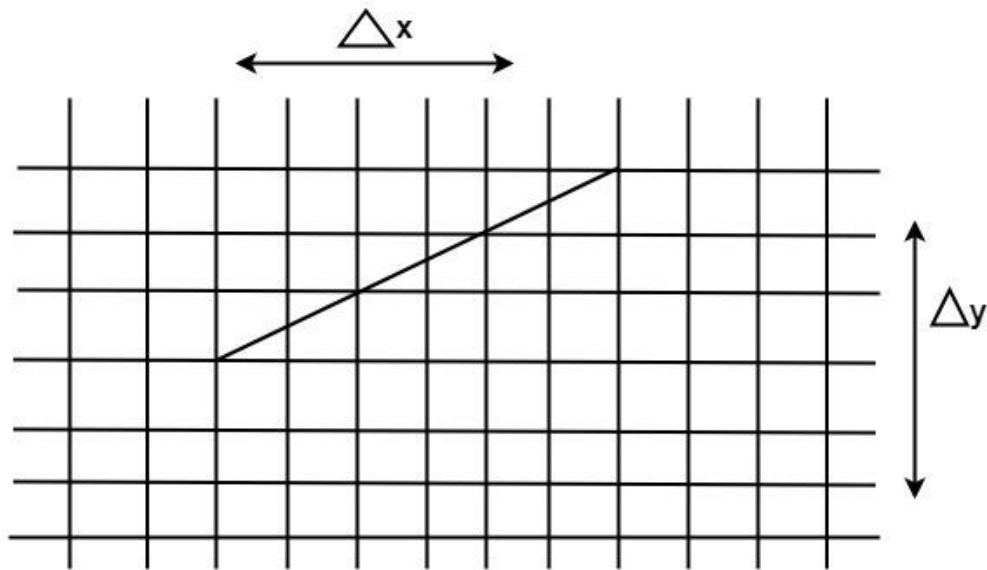


Fig: A straight line segment connecting 2 grid intersection may fail to pass through any other grid intersections.

2. Lines should terminate accurately: Unless lines are plotted accurately, they may terminate at the wrong place.

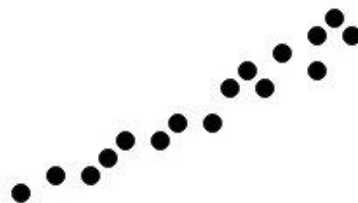


Fig: Uneven line density caused by bunching of dots.

3. Lines should have constant density: Line density is proportional to the no. of dots displayed divided by the length of the line.

To maintain constant density, dots should be equally spaced.

4. Line density should be independent of line length and angle: This can be done by computing an approximating line-length estimate and to use a line-generation algorithm that keeps line density constant to within the accuracy of this estimate.

5. Line should be drawn rapidly: This computation should be performed by special-purpose hardware.

Algorithm for line Drawing:

1. Direct use of line equation
2. DDA (Digital Differential Analyzer)
3. Bresenham's Algorithm

Direct use of line equation:

It is the simplest form of conversion. First of all scan P_1 and P_2 points. P_1 has co-ordinates (x_1', y_1') and (x_2', y_2') .

Then $m = (y_2', y_1') / (x_2', x_1')$ and $b = y_1' - mx_1'$

If value of $|m| \leq 1$ for each integer value of x . But do not consider x_1^1 and x_2^2

If value of $|m| > 1$ for each integer value of y . But do not consider y_1^1 and y_2^2

Example: A line with starting point as $(0, 0)$ and ending point $(6, 18)$ is given. Calculate value of intermediate points and slope of line.

Solution: $P_1 (0,0)$ $P_7 (6,18)$

$$x_1=0$$

$$y_1=0$$

$$x_2=6$$

$$y_2=18$$

$$M = \frac{\Delta y}{\Delta x} = \frac{y_2 - y_1}{x_2 - x_1} = \frac{18 - 0}{6 - 0} = \frac{18}{6} = 3$$

We know equation of line is

$$y = m x + b$$

$$y = 3x + b \dots \dots \dots \text{equation (1)}$$

put value of x from initial point in equation (1), i.e., $(0, 0)$ $x=0, y=0$

$$0 = 3 \times 0 + b$$

$$0 = b \Rightarrow b=0$$

put $b = 0$ in equation (1)

$$y = 3x + 0$$

$$y = 3x$$

Now calculate intermediate points

$$\text{Let } x = 1 \Rightarrow y = 3 \times 1 \Rightarrow y = 3$$

$$\text{Let } x = 2 \Rightarrow y = 3 \times 2 \Rightarrow y = 6$$

$$\text{Let } x = 3 \Rightarrow y = 3 \times 3 \Rightarrow y = 9$$

$$\text{Let } x = 4 \Rightarrow y = 3 \times 4 \Rightarrow y = 12$$

$$\text{Let } x = 5 \Rightarrow y = 3 \times 5 \Rightarrow y = 15$$

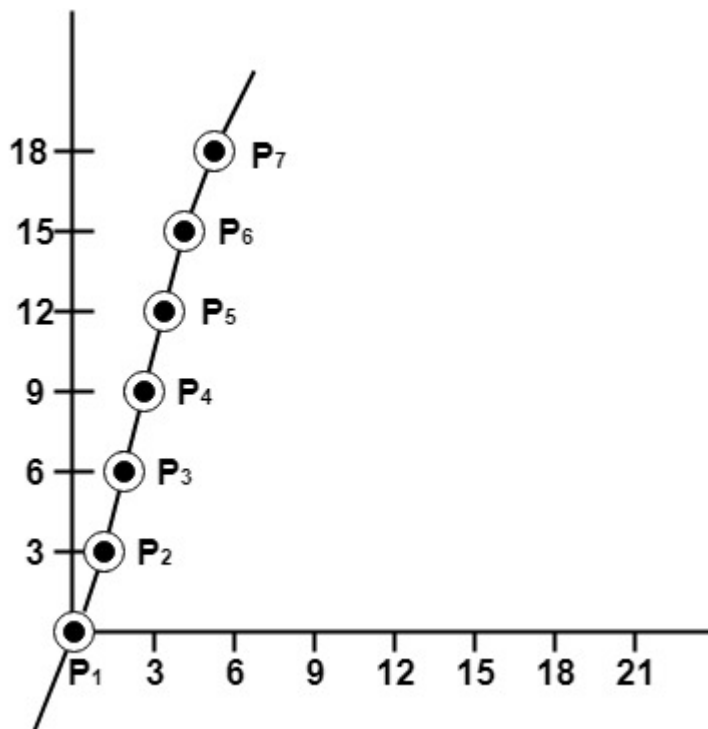
$$\text{Let } x = 6 \Rightarrow y = 3 \times 6 \Rightarrow y = 18$$

So points are $P_1 (0,0)$

$$P_2 (1,3)$$

$$P_3 (2,6)$$

$P_4 (3,9)$
 $P_5 (4,12)$
 $P_6 (5,15)$
 $P_7 (6,18)$



Algorithm for drawing line using equation:

Step1: Start Algorithm

Step2: Declare variables $x_1, x_2, y_1, y_2, dx, dy, m, b,$

Step3: Enter values of $x_1, x_2, y_1, y_2.$

The (x_1, y_1) are co-ordinates of a starting point of the line.

The (x_2, y_2) are co-ordinates of a ending point of the line.

Step4: Calculate $dx = x_2 - x_1$

Step5: Calculate $dy = y_2 - y_1$

Step6: Calculate $m = \frac{dy}{dx}$

Step7: Calculate $b = y_1 - m * x_1$

Step8: Set (x, y) equal to starting point, i.e., lowest point and x_{end} equal to largest value of x .

If $dx < 0$
 then $x = x_2$
 $y = y_2$
 $x_{end} = x_1$

```

    If dx > 0
        then x = x1
            y = y1
                Xend = X2

```

Step9: Check whether the complete line has been drawn if $x=x_{end}$, stop

Step10: Plot a point at current (x, y) coordinates

Step11: Increment value of x, i.e., $x = x+1$

Step12: Compute next value of y from equation $y = mx + b$

Step13: Go to Step9.

Program to draw a line using LineSlope Method

```

1. #include <graphics.h>
2. #include <stdlib.h>
3. #include <math.h>
4. #include <stdio.h>
5. #include <conio.h>
6. #include <iostream.h>
7.
8. class bresen
9. {
10.     float x, y, x1, y1, x2, y2, dx, dy, m, c, xend;
11.     public:
12.     void get ();
13.     void cal ();
14. };
15. void main ()
16. {
17.     bresen b;
18.     b.get ();
19.     b.cal ();
20.     getch ();
21. }
22. Void bresen :: get ()
23. {
24.     print ("Enter start & end points");
25.     print ("enter x1, y1, x2, y2");
26.     scanf ("%f%f%f%f",sx1, sx2, sx3, sx4)
27. }
28. void bresen ::cal ()
29. {

```



```

30.  /* request auto detection */
31.  int gdriver = DETECT,gmode, errorcode;
32.  /* initialize graphics and local variables */
33.  initgraph (&gdriver, &gmode, " ");
34.  /* read result of initialization */
35.  errorcode = graphresult ();
36.  if (errorcode != grOK)  /*an error occurred */
37.  {
38.      printf("Graphics error: %s \n", grapherrormsg (errorcode));
39.      printf ("Press any key to halt:");
40.      getch ();
41.      exit (1); /* terminate with an error code */
42.  }
43.  dx = x2-x1;
44.  dy=y2-y1;
45.  m = dy/dx;
46.  c = y1 - (m * x1);
47.  if (dx<0)
48.  {
49.      x=x2;
50.      y=y2;
51.      xend=x1;
52.  }
53.  else
54.  {
55.      x=x1;
56.      y=y1;
57.      xend=x2;
58.  }
59.  while (x<=xend)
60.  {
61.      putpixel (x, y, RED);
62.      y++;
63.      y=(x*x) +c;
64.  }
65. }

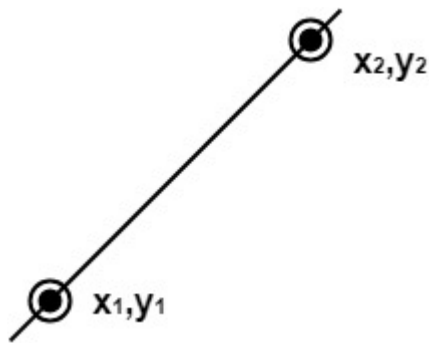
```

OUTPUT:

```

Enter Starting and End Points
Enter (X1, Y1, X2, Y2) 200 100 300 200

```



DDA Algorithm

DDA stands for Digital Differential Analyzer. It is an incremental method of scan conversion of line. In this method calculation is performed at each step but by using results of previous steps.

Suppose at step i , the pixels is (x_i, y_i)

The line of equation for step i

$$y_i = mx_i + b \dots \dots \dots \text{equation 1}$$

Next value will be

$$y_{i+1} = mx_{i+1} + b \dots \dots \dots \text{equation 2}$$

$$m = \frac{\Delta y}{\Delta x}$$

$$y_{i+1} - y_i = \Delta y \dots \dots \dots \text{equation 3}$$

$$y_{i+1} - x_i = \Delta x \dots \dots \dots \text{equation 4}$$

$$y_{i+1} = y_i + \Delta y$$

$$\Delta y = m \Delta x$$

$$y_{i+1} = y_i + m \Delta x$$

$$\Delta x = \Delta y / m$$

$$x_{i+1} = x_i + \Delta x$$

$$x_{i+1} = x_i + \Delta y / m$$

Case1: When $|M| < 1$ then (assume that $x_1 < x_2$)

$$x = x_1, y = y_1 \text{ set } \Delta x = 1$$

$$y_{i+1} = y_1 + m, \quad x = x + 1$$

Until $x = x_2$

Case2: When $|M| < 1$ then (assume that $y_1 < y_2$)

$$x = x_1, y = y_1 \text{ set } \Delta y = 1$$

$$\Delta x = \frac{1}{m}$$

$$x_{i+1} = x_i + \frac{1}{m}, \quad y = y + 1$$

Until $y = y_2$

Advantage:

1. It is a faster method than method of using direct use of line equation.
2. This method does not use multiplication theorem.
3. It allows us to detect the change in the value of x and y , so plotting of same point twice is not possible.
4. This method gives overflow indication when a point is repositioned.

5. It is an easy method because each step involves just two additions.

Disadvantage:

1. It involves floating point additions rounding off is done. Accumulations of round off error cause accumulation of error.
2. Rounding off operations and floating point operations consumes a lot of time.
3. It is more suitable for generating line using the software. But it is less suited for hardware implementation.

DDA Algorithm:

Step1: Start Algorithm

Step2: Declare x1,y1,x2,y2,dx,dy,x,y as integer variables.

Step3: Enter value of x1,y1,x2,y2.

Step4: Calculate $dx = x2 - x1$

Step5: Calculate $dy = y2 - y1$

Step6: If $ABS(dx) > ABS(dy)$
Then $step = abs(dx)$
Else

Step7: $xinc = dx / step$
 $yinc = dy / step$
assign $x = x1$
assign $y = y1$

Step8: Set pixel (x, y)

Step9: $x = x + xinc$
 $y = y + yinc$
Set pixels (Round (x), Round (y))

Step10: Repeat step 9 until $x = x2$

Step11: End Algorithm

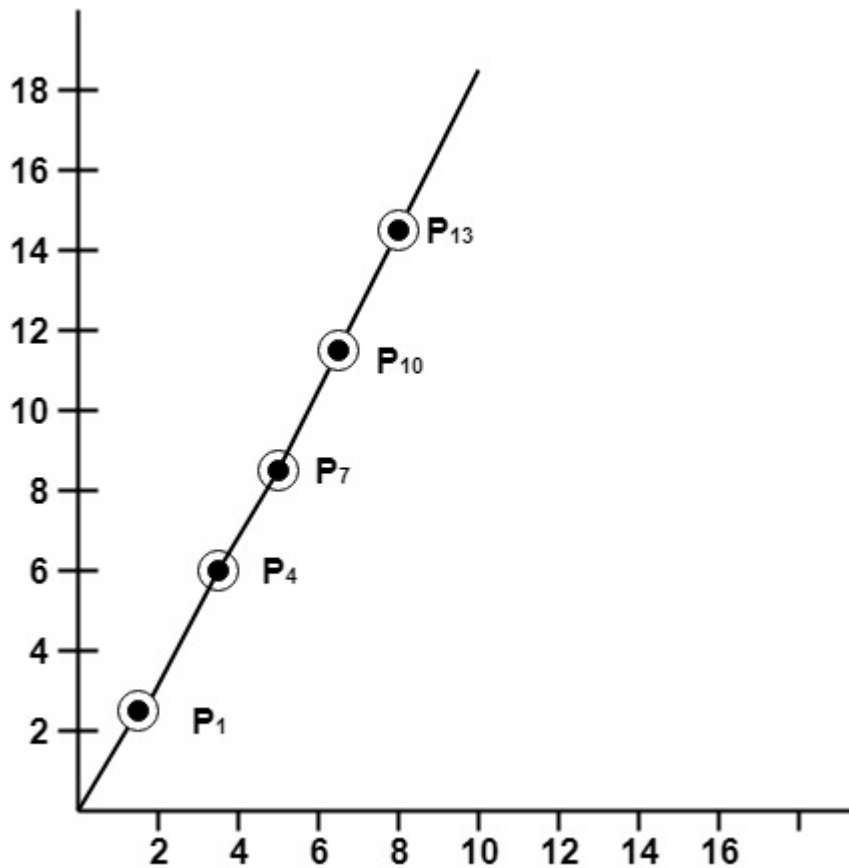
Example: If a line is drawn from (2, 3) to (6, 15) with use of DDA. How many points will needed to generate such line?

Solution: P1 (2,3) P11 (6,15)

$$\begin{aligned}x1 &= 2 \\y1 &= 3 \\x2 &= 6 \\y2 &= 15 \\dx &= 6 - 2 = 4 \\dy &= 15 - 3 = 12 \\m &= \frac{dy}{dx} = \frac{12}{4}\end{aligned}$$

For calculating next value of x takes $x = x + \frac{1}{m}$

$P_1(2, 3)$	point plotted
$P_2(2\frac{1}{3}, 4)$	point plotted
$P_3(2\frac{2}{3}, 5)$	point not plotted
$P_4(3, 6)$	point plotted
$P_5(3\frac{1}{3}, 7)$	point not plotted
$P_6(3\frac{2}{3}, 8)$	point not plotted
$P_7(4, 9)$	point plotted
$P_8(4\frac{1}{3}, 10)$	point not plotted
$P_9(4\frac{2}{3}, 11)$	point not plotted
$P_{10}(5, 12)$	point plotted
$P_{11}(5\frac{1}{3}, 13)$	point not plotted
$P_{12}(5\frac{2}{3}, 14)$	point not plotted
$P_{13}(6, 15)$	point plotted



Program to implement DDA Line Drawing Algorithm:

```

1. #include<graphics.h>
2. #include<conio.h>
3. #include<stdio.h>
4. void main()
5. {
6.     intgd = DETECT ,gm, i;
7.     float x, y,dx,dy,steps;
8.     int x0, x1, y0, y1;
9.     initgraph(&gd, &gm, "C:\\TC\\BGI");
10.    setbkcolor(WHITE);
11.    x0 = 100 , y0 = 200, x1 = 500, y1 = 300;
12.    dx = (float)(x1 - x0);
13.    dy = (float)(y1 - y0);
14.    if(dx>=dy)
15.    {
16.        steps = dx;
17.    }
18.    else
19.    {
20.        steps = dy;
21.    }

```

```

22. dx = dx/steps;
23. dy = dy/steps;
24. x = x0;
25. y = y0;
26. i = 1;
27. while(i <= steps)
28. {
29.     putpixel(x, y, RED);
30.     x += dx;
31.     y += dy;
32.     i=i+1;
33. }
34. getch();
35. closegraph();
36. }

```

Output:



Bresenham's Line Algorithm

This algorithm is used for scan converting a line. It was developed by Bresenham. It is an efficient method because it involves only integer addition, subtractions, and multiplication operations. These operations can be performed very rapidly so lines can be generated quickly.

In this method, next pixel selected is that one who has the least distance from true line.

The method works as follows:

Assume a pixel $P_1'(x_1', y_1')$, then select subsequent pixels as we work our way to the right, one pixel position at a time in the horizontal direction toward $P_2'(x_2', y_2')$.

Once a pixel is chosen at any step

The next pixel is

1. Either the one to its right (lower-bound for the line)
2. One top its right and up (upper-bound for the line)

The line is best approximated by those pixels that fall the least distance from the path between P_1' , P_2' .

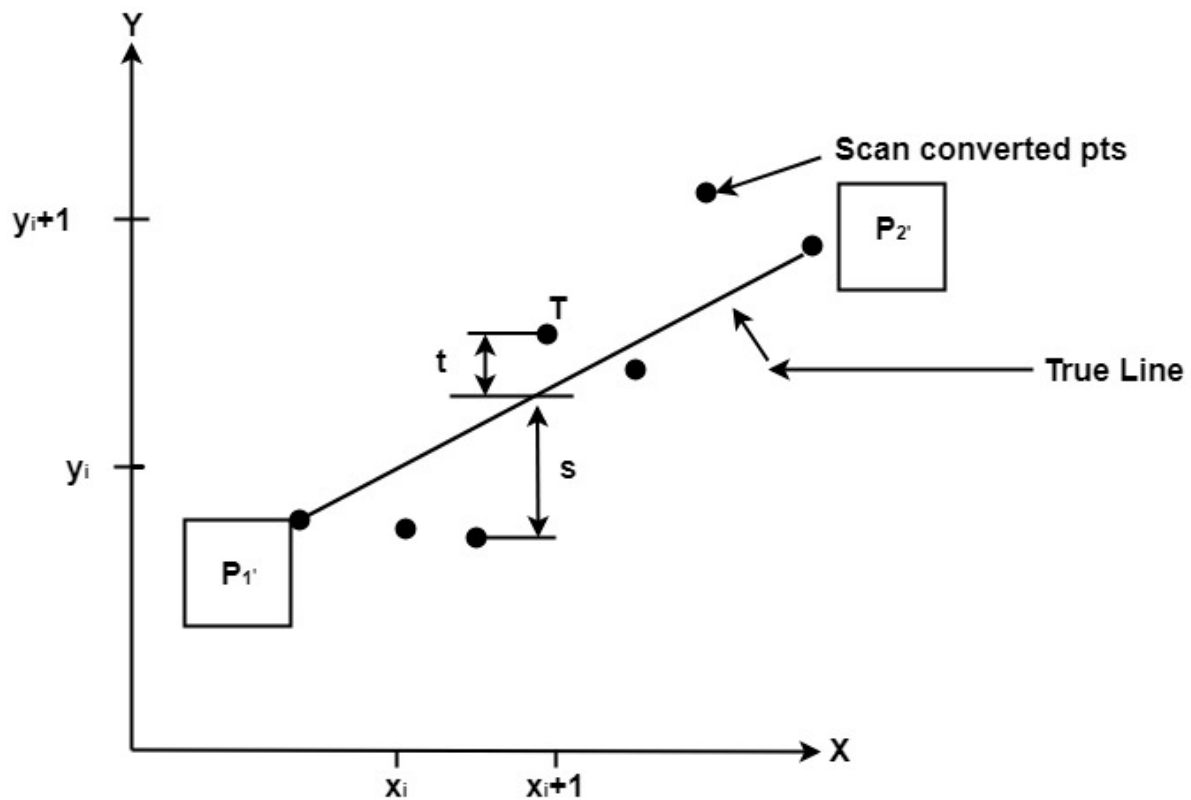


Fig: Scan Converting a line.

To choose the next one between the bottom pixel S and top pixel T.

If S is chosen

We have $x_{i+1} = x_i + 1$ and $y_{i+1} = y_i$

If T is chosen

We have $x_{i+1} = x_i + 1$ and $y_{i+1} = y_i + 1$

The actual y coordinates of the line at $x = x_{i+1}$ is

$$y = mx_{i+1} + b$$

$$y = m(x_i + 1) + b$$

The distance from S to the actual line in y direction

$$s = y - y_i$$

The distance from T to the actual line in y direction

$$t = (y_i + 1) - y$$

Now consider the difference between these 2 distance values

$$s - t$$

When $(s-t) < 0 \Rightarrow s < t$

The closest pixel is S

When $(s-t) \geq 0 \Rightarrow s \geq t$

The closest pixel is T

This difference is

$$\begin{aligned} s-t &= (y-y_i) - [(y_i+1)-y] \\ &= 2y - 2y_i - 1 \end{aligned}$$

$$s - t = 2m(x_i + 1) + 2b - 2y_i - 1$$

[Putting the value of (1)]

Substituting m by $\frac{\Delta y}{\Delta x}$ and introducing decision variable

$$d_i = \Delta x (s-t)$$

$$\begin{aligned} d_i &= \Delta x (2 \frac{\Delta y}{\Delta x} (x_i + 1) + 2b - 2y_i - 1) \\ &= 2\Delta x y_i - 2\Delta y - 1\Delta x + 2b - 2y_i \Delta x - \Delta x \\ d_i &= 2\Delta y \cdot x_i - 2\Delta x \cdot y_i + c \end{aligned}$$

Where $c = 2\Delta y + \Delta x (2b - 1)$

We can write the decision variable d_{i+1} for the next slip on

$$\begin{aligned} d_{i+1} &= 2\Delta y \cdot x_{i+1} - 2\Delta x \cdot y_{i+1} + c \\ d_{i+1} - d_i &= 2\Delta y \cdot (x_{i+1} - x_i) - 2\Delta x (y_{i+1} - y_i) \end{aligned}$$

Since $x_{i+1} = x_i + 1$, we have

$$d_{i+1} - d_i = 2\Delta y \cdot (x_i + 1 - x_i) - 2\Delta x (y_{i+1} - y_i)$$

Special Cases

If chosen pixel is at the top pixel T (i.e., $d_i \geq 0$) $\Rightarrow y_{i+1} = y_i + 1$

$$d_{i+1} = d_i + 2\Delta y - 2\Delta x$$

If chosen pixel is at the bottom pixel T (i.e., $d_i < 0$) $\Rightarrow y_{i+1} = y_i$

$$d_{i+1} = d_i + 2\Delta y$$

Finally, we calculate d_1

$$\begin{aligned} d_1 &= \Delta x [2m(x_1 + 1) + 2b - 2y_1 - 1] \\ d_1 &= \Delta x [2(mx_1 + b - y_1) + 2m - 1] \end{aligned}$$

Since $mx_1 + b - y_i = 0$ and $m = \frac{\Delta y}{\Delta x}$, we have
 $d_1 = 2\Delta y - \Delta x$

Advantage:

1. It involves only integer arithmetic, so it is simple.
2. It avoids the generation of duplicate points.
3. It can be implemented using hardware because it does not use multiplication and division.
4. It is faster as compared to DDA (Digital Differential Analyzer) because it does not involve floating point calculations like DDA Algorithm.

Disadvantage:

1. This algorithm is meant for basic line drawing only. Initializing is not a part of Bresenham's line algorithm. So to draw smooth lines, you should want to look into a different algorithm.

Bresenham's Line Algorithm:

Step1: Start Algorithm

Step2: Declare variable $x_1, x_2, y_1, y_2, d, i_1, i_2, dx, dy$

Step3: Enter value of x_1, y_1, x_2, y_2
 Where x_1, y_1 are coordinates of starting point
 And x_2, y_2 are coordinates of Ending point

Step4: Calculate $dx = x_2 - x_1$
 Calculate $dy = y_2 - y_1$
 Calculate $i_1 = 2 * dy$
 Calculate $i_2 = 2 * (dy - dx)$
 Calculate $d = i_1 - dx$

Step5: Consider (x, y) as starting point and x_{end} as maximum possible value of x .
 If $dx < 0$
 Then $x = x_2$
 $y = y_2$
 $x_{end} = x_1$
 If $dx > 0$
 Then $x = x_1$
 $y = y_1$
 $x_{end} = x_2$

Step6: Generate point at (x, y) coordinates.

Step7: Check if whole line is generated.
 If $x > x_{end}$
 Stop.

Step8: Calculate co-ordinates of the next pixel

If $d < 0$

Then $d = d + I_1$

If $d \geq 0$

Then $d = d + I_2$

Increment $y = y + 1$

Step9: Increment $x = x + 1$

Step10: Draw a point of latest (x, y) coordinates

Step11: Go to step 7

Step12: End of Algorithm

Example: Starting and Ending position of the line are $(1, 1)$ and $(8, 5)$. Find intermediate points.

Solution: $x_1 = 1$

$y_1 = 1$

$x_2 = 8$

$y_2 = 5$

$dx = x_2 - x_1 = 8 - 1 = 7$

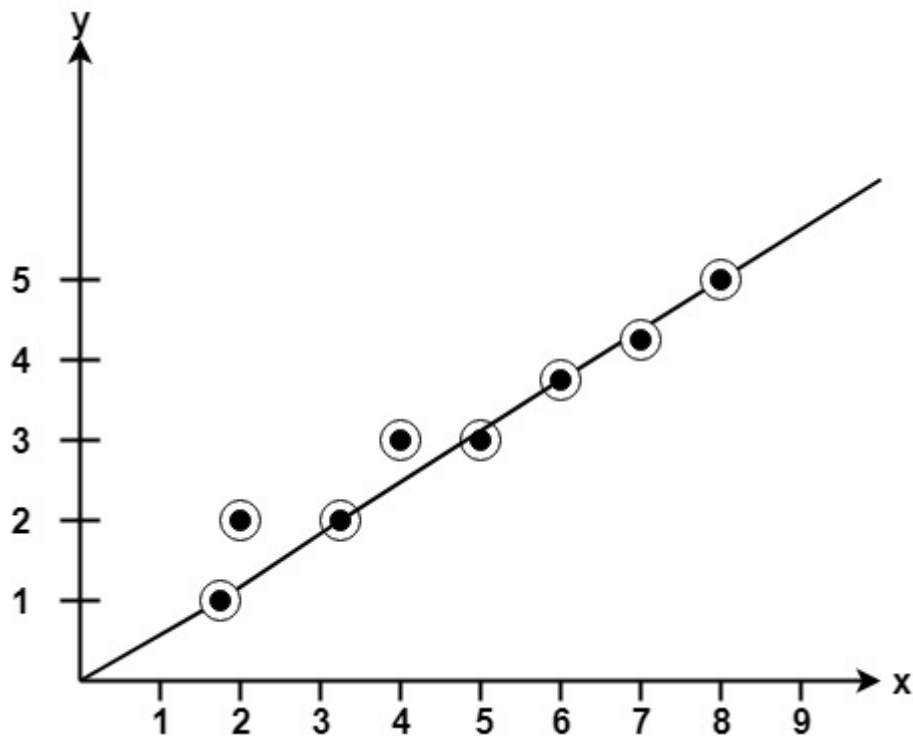
$dy = y_2 - y_1 = 5 - 1 = 4$

$I_1 = 2 * \Delta y = 2 * 4 = 8$

$I_2 = 2 * (\Delta y - \Delta x) = 2 * (4 - 7) = -6$

$d = I_1 - \Delta x = 8 - 7 = 1$

x	y	d=d+I ₁ or I ₂
1	1	$d+I_2=1+(-6)=-5$
2	2	$d+I_1=-5+8=3$
3	2	$d+I_2=3+(-6)=-3$
4	3	$d+I_1=-3+8=5$
5	3	$d+I_2=5+(-6)=-1$
6	4	$d+I_1=-1+8=7$
7	4	$d+I_2=7+(-6)=1$



Program to implement Bresenham's Line Drawing Algorithm:

```

1. #include<stdio.h>
2. #include<graphics.h>
3. void drawline(int x0, int y0, int x1, int y1)
4. {
5.     int dx, dy, p, x, y;
6.     dx=x1-x0;
7.     dy=y1-y0;
8.     x=x0;
9.     y=y0;
10.    p=2*dy-dx;
11.    while(x<x1)
12.    {
13.        if(p>=0)
14.        {
15.            putpixel(x,y,7);
16.            y=y+1;
17.            p=p+2*dy-2*dx;
18.        }
19.        else
20.        {
21.            putpixel(x,y,7);

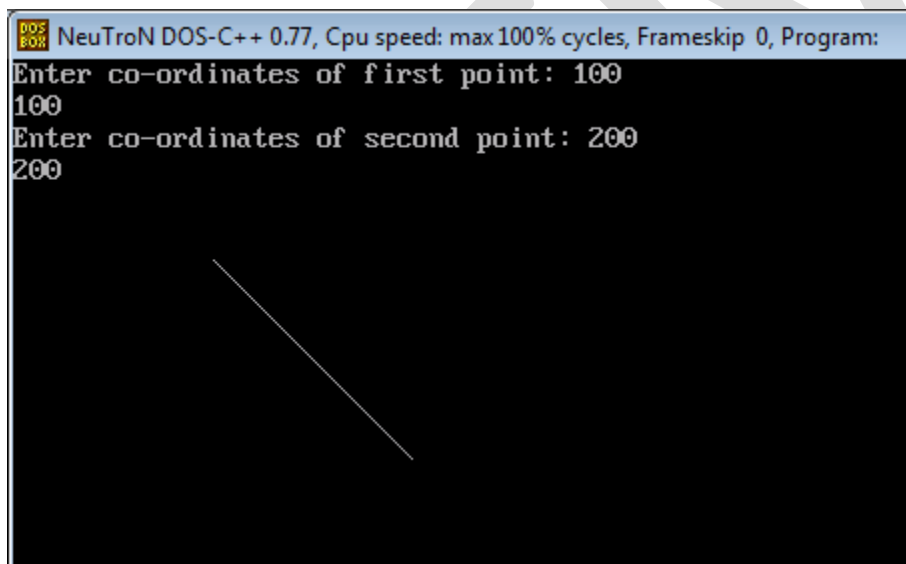
```

```

22.         p=p+2*dy;}
23.         x=x+1;
24.     }
25. }
26. int main()
27. {
28.     int gdriver=DETECT, gmode, error, x0, y0, x1, y1;
29.     initgraph(&gdriver, &gmode, "c:\\turbo3\\bgi");
30.     printf("Enter co-ordinates of first point: ");
31.     scanf("%d%d", &x0, &y0);
32.     printf("Enter co-ordinates of second point: ");
33.     scanf("%d%d", &x1, &y1);
34.     drawline(x0, y0, x1, y1);
35.     return 0;
36. }

```

Output:



```

NeuTroN DOS-C++ 0.77, Cpu speed: max 100% cycles, Frameskip 0, Program:
Enter co-ordinates of first point: 100
100
Enter co-ordinates of second point: 200
200

```

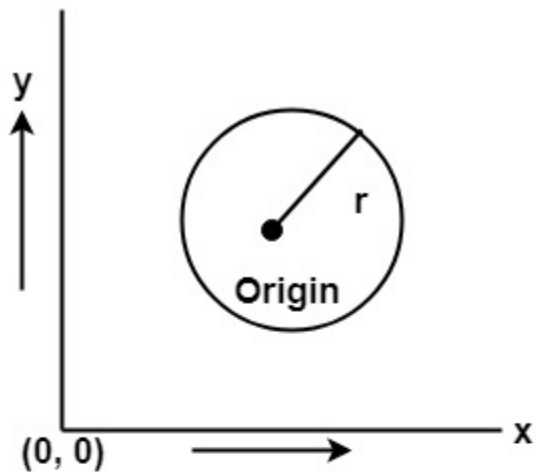
Differentiate between DDA Algorithm and Bresenham's Line Algorithm:

DDA Algorithm	Bresenham's Line Algorithm
1. DDA Algorithm use floating point, i.e., Real Arithmetic.	1. Bresenham's Line Algorithm use fixed point, i.e., Integer Arithmetic
2. DDA Algorithms uses multiplication & division its operation	2. Bresenham's Line Algorithm uses only subtraction and addition its operation
3. DDA Algorithm is slowly than Bresenham's Line Algorithm in line drawing because it uses real arithmetic (Floating Point operation)	3. Bresenham's Algorithm is faster than DDA Algorithm in line because it involves only addition & subtraction in its calculation and uses only integer arithmetic.
4. DDA Algorithm is not accurate and efficient as Bresenham's Line Algorithm.	4. Bresenham's Line Algorithm is more accurate and efficient at DDA Algorithm.
5. DDA Algorithm can draw circle and curves but are not accurate as Bresenham's Line Algorithm	5. Bresenham's Line Algorithm can draw circle and curves with more accurate than DDA Algorithm.

Defining a Circle:

Circle is an eight-way symmetric figure. The shape of circle is the same in all quadrants. In each quadrant, there are two octants. If the calculation of the point of one octant is done, then the other seven points can be calculated easily by using the concept of eight-way symmetry.

For drawing, circle considers it at the origin. If a point is $P_1(x, y)$, then the other seven points will be



P2 (x, -y)

P3 (-x, -y)

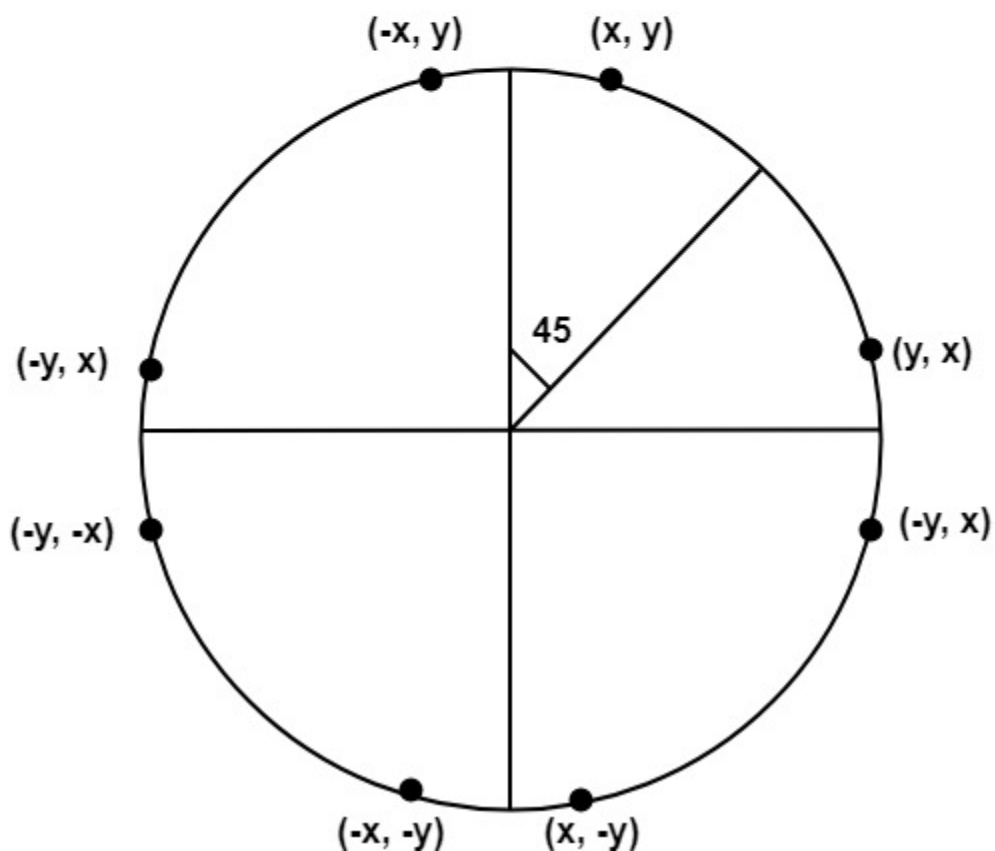
P4 (-x, y)

P5 (y, x)

P6 (y, -x)

P7 (-y, -x)

P8 (-y, x)



So we will calculate only 45° arc. From which the whole circle can be determined easily.

If we want to display circle on screen then the putpixel function is used for eight points as shown below:

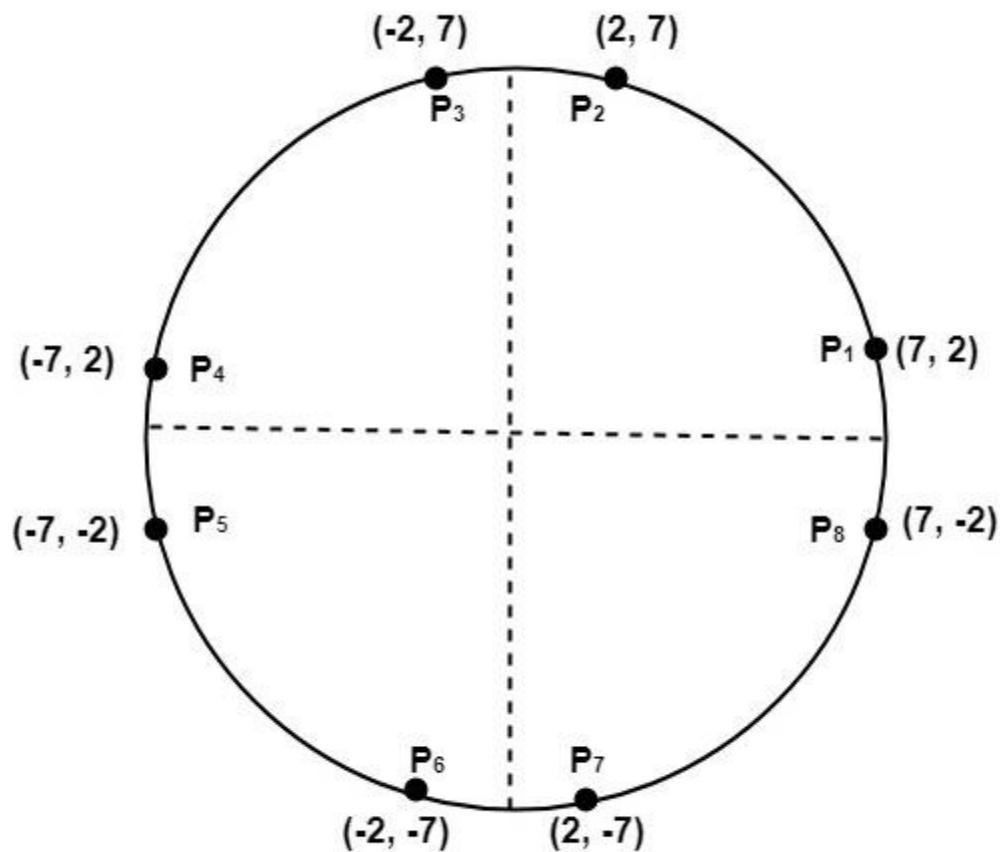
```
putpixel (x, y, color)
putpixel (x, -y, color)
putpixel (-x, y, color)
putpixel (-x, -y, color)
putpixel (y, x, color)
putpixel (y, -x, color)
```

```
putpixel (-y, x, color)
putpixel (-y, -x, color)
```

Example: Let we determine a point (2, 7) of the circle then other points will be (2, -7), (-2, -7), (-2, 7), (7, 2), (-7, 2), (-7, -2), (7, -2)

These seven points are calculated by using the property of reflection. The reflection is accomplished in the following way:

The reflection is accomplished by reversing x, y co-ordinates.



Eight way symmetry of a Circle

There are two standards methods of mathematically defining a circle centered at the origin.

1. Defining a circle using Polynomial Method
2. Defining a circle using Polar Co-ordinates

Defining a circle using Polynomial Method:

The first method defines a circle with the second-order polynomial equation as shown in fig:

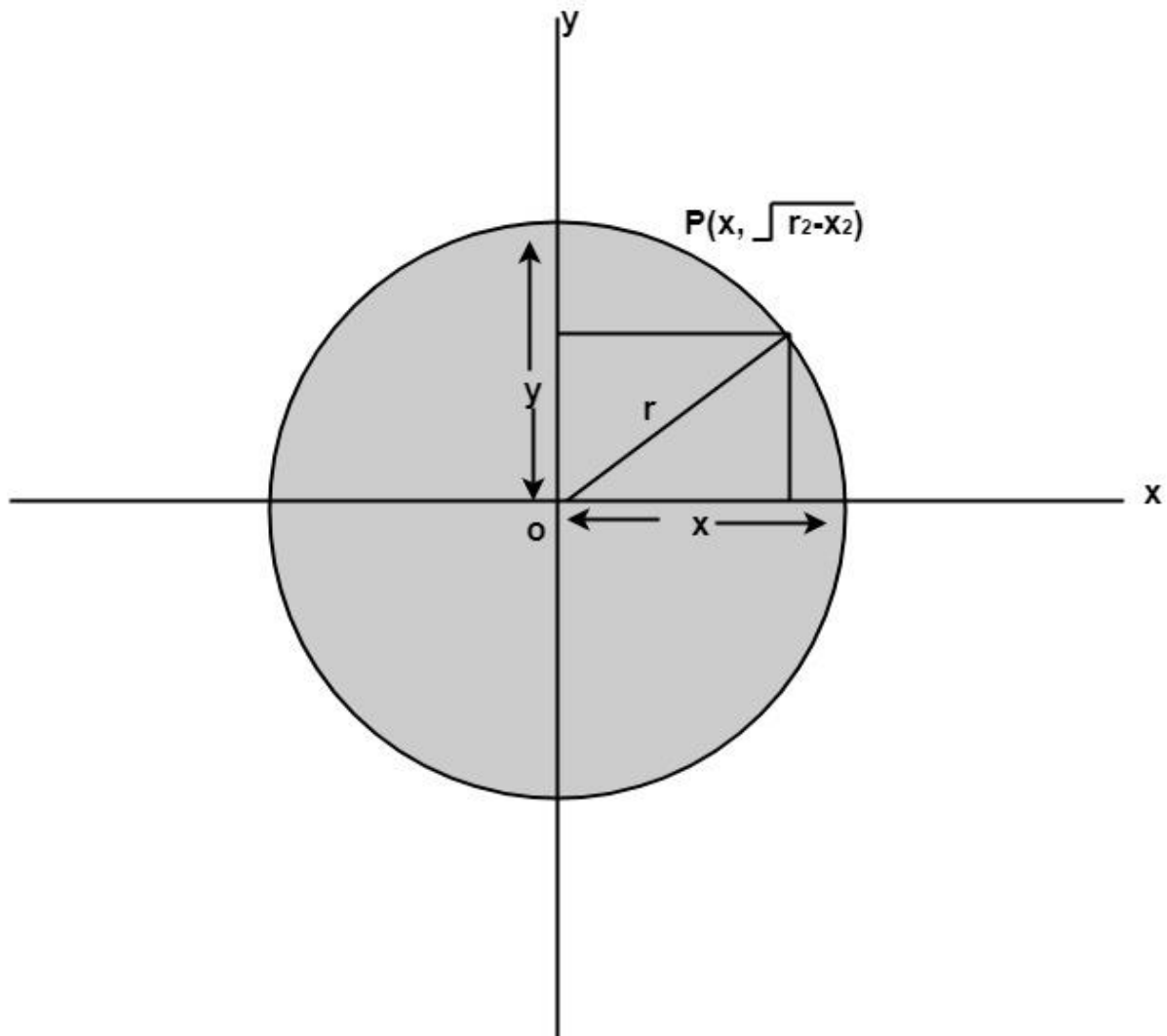
$$y^2 = r^2 - x^2$$

Where x = the x coordinate

y = the y coordinate

r = the circle radius

With the method, each x coordinate in the sector, from 90° to 45° , is found by stepping x from 0 to $\frac{r}{\sqrt{2}}$ & each y coordinate is found by evaluating $\sqrt{r^2 - x^2}$ for each step of x .



Algorithm:

Step1: Set the initial variables

r = circle radius

(h, k) = coordinates of circle center

$x=0$

$$I = \text{step size}$$

$$x_{\text{end}} = \frac{r}{\sqrt{2}}$$

Step2: Test to determine whether the entire circle has been scan-converted.

If $x > x_{\text{end}}$ then stop.

Step3: Compute $y = \sqrt{r^2 - x^2}$

Step4: Plot the eight points found by symmetry concerning the center (h, k) at the current (x, y) coordinates.

Plot (x + h, y + k)	Plot (-x + h, -y + k)
Plot (y + h, x + k)	Plot (-y + h, -x + k)
Plot (-y + h, x + k)	Plot (y + h, -x + k)
Plot (-x + h, y + k)	Plot (x + h, -y + k)

Step5: Increment $x = x + i$

Step6: Go to step (ii).

Program to draw a circle using Polynomial Method:

```

1. #include<graphics.h>
2. #include<conio.h>
3. #include<math.h>
4. voidsetPixel(int x, int y, int h, int k)
5. {
6.     putpixel(x+h, y+k, RED);
7.     putpixel(x+h, -y+k, RED);
8.     putpixel(-x+h, -y+k, RED);
9.     putpixel(-x+h, y+k, RED);
10.    putpixel(y+h, x+k, RED);
11.    putpixel(y+h, -x+k, RED);
12.    putpixel(-y+h, -x+k, RED);
13.    putpixel(-y+h, x+k, RED);
14. }
15. main()
16. {
17.     intgd=0, gm,h,k,r;
18.     double x,y,x2;
19.     h=200, k=200, r=100;
20.     initgraph(&gd, &gm, "C:\\TC\\BGI ");
21.     setbkcolor(WHITE);
22.     x=0,y=r;
23.     x2 = r/sqrt(2);
24.     while(x<=x2)

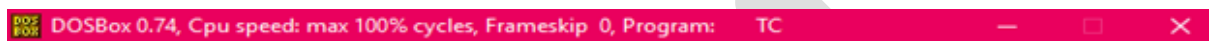
```

```

25. {
26.     y = sqrt(r*r - x*x);
27.     setPixel(floor(x), floor(y), h,k);
28.     x += 1;
29. }
30. getch();
31. closegraph();
32. return 0;
33.}

```

Output:

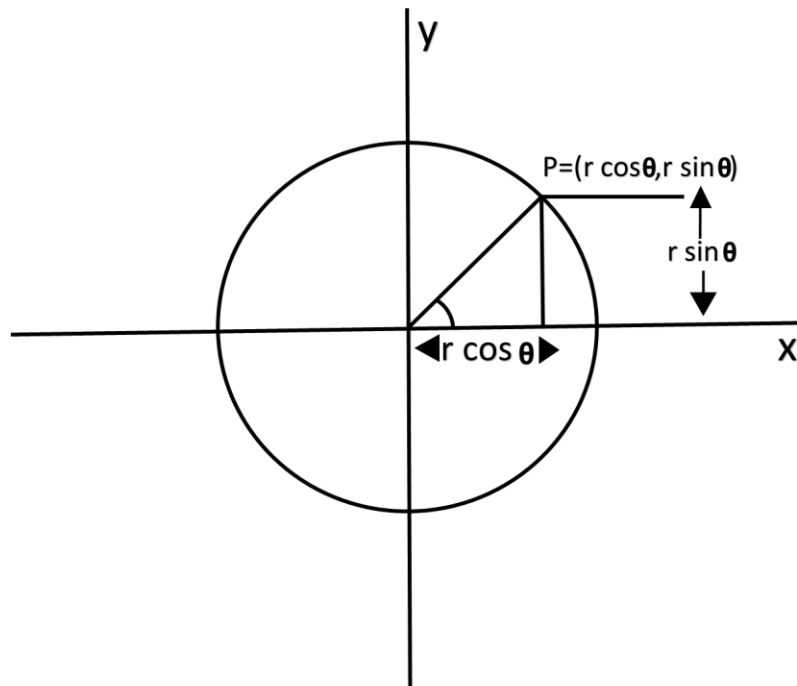


Defining a circle using Polar Co-ordinates :

The second method of defining a circle makes use of polar coordinates as shown in fig:

$x = r \cos \theta$ $y = r \sin \theta$
 Where θ = current angle
 r = circle radius
 x = x coordinate
 y = y coordinate

By this method, θ is stepped from 0 to $\frac{\pi}{4}$ & each value of x & y is calculated.



Algorithm:

Step1: Set the initial variables:

r = circle radius
 (h, k) = coordinates of the circle center
 i = step size
 $\theta_{end} = \left(\frac{22}{7}\right)/4$
 $\theta = 0$

Step2: If $\theta > \theta_{end}$ then stop.

Step3: Compute

$$x = r * \cos \theta \quad y = r * \sin \theta$$

Step4: Plot the eight points, found by symmetry i.e., the center (h, k) , at the current (x, y) coordinates.

Plot $(x + h, y + k)$	Plot $(-x + h, -y + k)$
Plot $(y + h, x + k)$	Plot $(-y + h, -x + k)$
Plot $(-y + h, x + k)$	Plot $(y + h, -x + k)$
Plot $(-x + h, y + k)$	Plot $(x + h, -y + k)$

Step5: Increment $\theta = \theta + i$

Step6: Go to step (ii).

Program to draw a circle using Polar Coordinates:

```
1. #include <graphics.h>
2. #include <stdlib.h>
3. #define color 10
4. void eightWaySymmetricPlot(int xc,int yc,int x,int y)
5. {
6.     putpixel(x+xc,y+yc,color);
7.     putpixel(x+xc,-y+yc,color);
8.     putpixel(-x+xc,-y+yc,color);
9.     putpixel(-x+xc,y+yc,color);
10.    putpixel(y+xc,x+yc,color);
11.    putpixel(y+xc,-x+yc,color);
12.    putpixel(-y+xc,-x+yc,color);
13.    putpixel(-y+xc,x+yc,color);
14. }
15. void PolarCircle(int xc,int yc,int r)
16. {
17.     int x,y,d;
18.     x=0;
19.     y=r;
20.     d=3-2*r;
21.     eightWaySymmetricPlot(xc,yc,x,y);
22.     while(x<=y)
23.     {
24.         if(d<=0)
25.         {
26.             d=d+4*x+6;
27.         }
28.         else
29.         {
30.             d=d+4*x-4*y+10;
31.             y=y-1;
32.         }
33.         x=x+1;
34.         eightWaySymmetricPlot(xc,yc,x,y);
35.     }
36. }
37. int main(void)
38. {
39.     int gdriver = DETECT, gmode, errorcode;
```

```

40.  int xc,yc,r;
41.  initgraph(&gdriver, &gmode, "c:\\turboc3\\bgi");
42. errorcode = graphresult();
43. if (errorcode != grOk)
44.  {
45.      printf("Graphics error: %s\n", grapherrormsg(errorcode));
46.      printf("Press any key to halt:");
47.      getch();
48.      exit(1);
49.  }
50. printf("Enter the values of xc and yc ,that is center points of circle : ");
51.  scanf("%d%d",&xc,&yc);
52.  printf("Enter the radius of circle : ");
53.  scanf("%d",&r);
54.  PolarCircle(xc,yc,r);
55.  getch();
56.  closegraph();
57.  return 0;
58. }

```

Output:

```

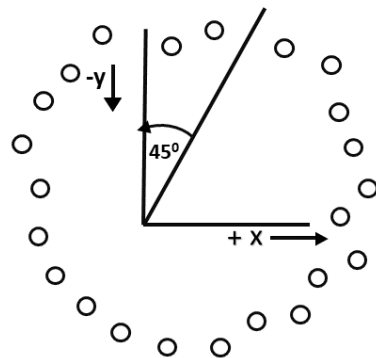
Enter the values of xc and yc ,that is center points of circle : 250 300
Enter the radius of circle : 80

```

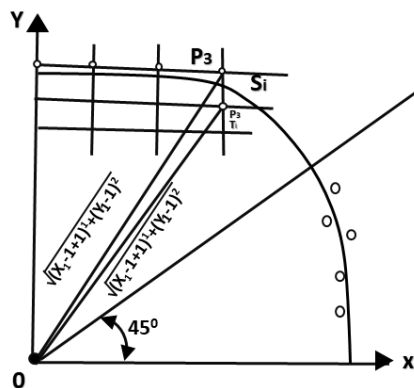


Bresenham's Circle Algorithm:

tion of the true circle will be described by those pixels whose distance from the true circle. We want to generate the



The best approximation of the true circle will be described by those pixels in the raster that falls the least distance from the true circle. We want to generate the points from



1. Move in the x-direction one unit or

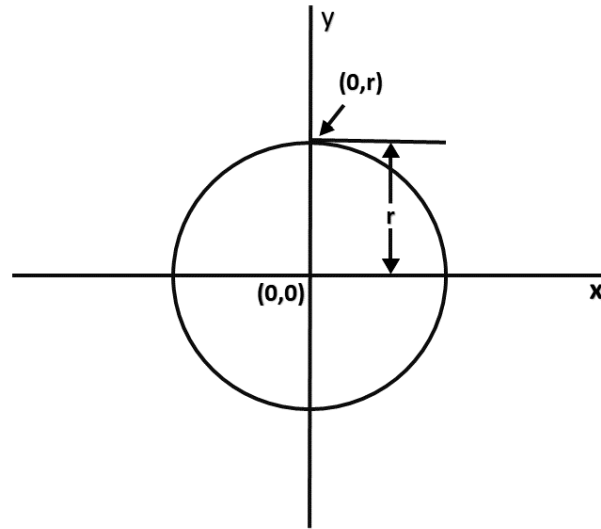
2. Move in the x- direction one unit & move in the negative y-direction one unit.

Let $D(S_i)$ is the distance from the origin to the true circle squared minus the distance to point P_3 squared. $D(T_i)$ is the distance from the origin to the true circle squared minus the distance to point P_2 squared. Therefore, the following expressions arise.

$$D(S_i) = (x_{i-1} + 1)^2 + y_{i-1}^2 - r^2$$

$$D(T_i) = (x_{i-1} + 1)^2 + (y_{i-1} - 1)^2 - r^2$$

Since $D(S_i)$ will always be +ve & $D(T_i)$ will always be -ve, a decision variable d may be defined as follows:



$$d_i = D(S_i) + D(T_i)$$

Therefore,

$$d_i = (x_{i-1} + 1)^2 + y_{i-1}^2 - r^2 + (x_{i-1} + 1)^2 + (y_{i-1} - 1)^2 - r^2$$

From this equation, we can drive initial values of d_i as

If it is assumed that the circle is centered at the origin, then at the first step $x = 0$ & $y = r$.

Therefore,

$$d_i = (0 + 1)^2 + r^2 - r^2 + (0 + 1)^2 + (r - 1)^2 - r^2$$

$$= 1 + 1 + r^2 - 2r + 1 - r^2$$

$$= 3 - 2r$$

Thereafter, if $d_i < 0$, then only x is incremented.

$$x_{i+1} = x_i + 1 \quad d_{i+1} = d_i + 4x_i + 6$$

& if $d_i \geq 0$, then x & y are incremented

$x_{i+1} = x_i + 1$ $y_{i+1} = y_i + 1$

$d_{i+1} = d_i + 4(x_i - y_i) + 10$

Bresenham's Circle Algorithm:

Step1: Start Algorithm

Step2: Declare p, q, x, y, r, d variables

p, q are coordinates of the center of the circle

r is the radius of the circle

Step3: Enter the value of r

Step4: Calculate $d = 3 - 2r$

Step5: Initialize $x = 0$

$y = r$

Step6: Check if the whole circle is scan converted

 If $x \geq y$

 Stop

Step7: Plot eight points by using concepts of eight-way symmetry. The center is at (p, q) . Current active pixel is (x, y) .

 putpixel $(x+p, y+q)$

 putpixel $(y+p, x+q)$

 putpixel $(-y+p, x+q)$

 putpixel $(-x+p, y+q)$

 putpixel $(-x+p, -y+q)$

 putpixel $(-y+p, -x+q)$

 putpixel $(y+p, -x+q)$

 putpixel $(x+p, -y+q)$

Step8: Find location of next pixels to be scanned

 If $d < 0$

 then $d = d + 4x + 6$

 increment $x = x + 1$

 If $d \geq 0$

 then $d = d + 4(x - y) + 10$

 increment $x = x + 1$

 decrement $y = y - 1$

Step9: Go to step 6

Step10: Stop Algorithm

Example: Plot 6 points of circle using Bresenham Algorithm. When radius of circle is 10 units. The circle has centre $(50, 50)$.

Solution: Let $r = 10$ (Given)

Step1: Take initial point $(0, 10)$

$d = 3 - 2r$

$$\begin{aligned}
 d &= 3 - 2 * 10 = -17 \\
 d < 0 \therefore d &= d + 4x + 6 \\
 &= -17 + 4 (0) + 6 \\
 &= -11
 \end{aligned}$$

Step2: Plot (1, 10)

$$\begin{aligned}
 d &= d + 4x + 6 (\because d < 0) \\
 &= -11 + 4 (1) + 6 \\
 &= -1
 \end{aligned}$$

Step3: Plot (2, 10)

$$\begin{aligned}
 d &= d + 4x + 6 (\because d < 0) \\
 &= -1 + 4 \times 2 + 6 \\
 &= 13
 \end{aligned}$$

Step4: Plot (3, 9) d is > 0 so x = x + 1, y = y - 1

$$\begin{aligned}
 d &= d + 4 (x-y) + 10 (\because d > 0) \\
 &= 13 + 4 (3-9) + 10 \\
 &= 13 + 4 (-6) + 10 \\
 &= 23-24=-1
 \end{aligned}$$

Step5: Plot (4, 9)

$$\begin{aligned}
 d &= -1 + 4x + 6 \\
 &= -1 + 4 (4) + 6 \\
 &= 21
 \end{aligned}$$

Step6: Plot (5, 8)

$$\begin{aligned}
 d &= d + 4 (x-y) + 10 (\because d > 0) \\
 &= 21 + 4 (5-8) + 10 \\
 &= 21-12 + 10 = 19
 \end{aligned}$$

So P₁ (0,0)⇒(50,50)

P₂ (1,10)⇒(51,60)

P₃ (2,10)⇒(52,60)

P₄ (3,9)⇒(53,59)

P₅ (4,9)⇒(54,59)

P₆ (5,8)⇒(55,58)

Program to draw a circle using Bresenham's circle drawing algorithm:

```

1. #include <graphics.h>
2. #include <stdlib.h>
3. #include <stdio.h>
4. #include <conio.h>
5. #include <math.h>
6.
7. void EightWaySymmetricPlot(int xc,int yc,int x,int y)
8. {
9.     putpixel(x+xc,y+yc,RED);
10.    putpixel(x+xc,-y+yc,YELLOW);
11.    putpixel(-x+xc,-y+yc,GREEN);

```

```

12. putpixel(-x+xc,y+yc,YELLOW);
13. putpixel(y+xc,x+yc,12);
14. putpixel(y+xc,-x+yc,14);
15. putpixel(-y+xc,-x+yc,15);
16. putpixel(-y+xc,x+yc,6);
17. }
18.
19. void BresenhamCircle(int xc,int yc,int r)
20. {
21.     int x=0,y=r,d=3-(2*r);
22.     EightWaySymmetricPlot(xc,yc,x,y);
23.
24.     while(x<=y)
25.     {
26.         if(d<=0)
27.         {
28.             d=d+(4*x)+6;
29.         }
30.         else
31.         {
32.             d=d+(4*x)-(4*y)+10;
33.             y=y-1;
34.         }
35.         x=x+1;
36.         EightWaySymmetricPlot(xc,yc,x,y);
37.     }
38. }
39.
40. int main(void)
41. {
42.     /* request auto detection */
43.     int xc,yc,r,gdriver = DETECT, gmode, errorcode;
44.     /* initialize graphics and local variables */
45.     initgraph(&gdriver, &gmode, "C:\\TURBOC3\\BGI");
46.
47.     /* read result of initialization */
48.     errorcode = graphresult();
49.
50.     if (errorcode != grOk) /* an error occurred */
51.     {
52.         printf("Graphics error: %s\n", grapherrormsg(errorcode));
53.         printf("Press any key to halt:");
54.         getch();
55.         exit(1); /* terminate with an error code */

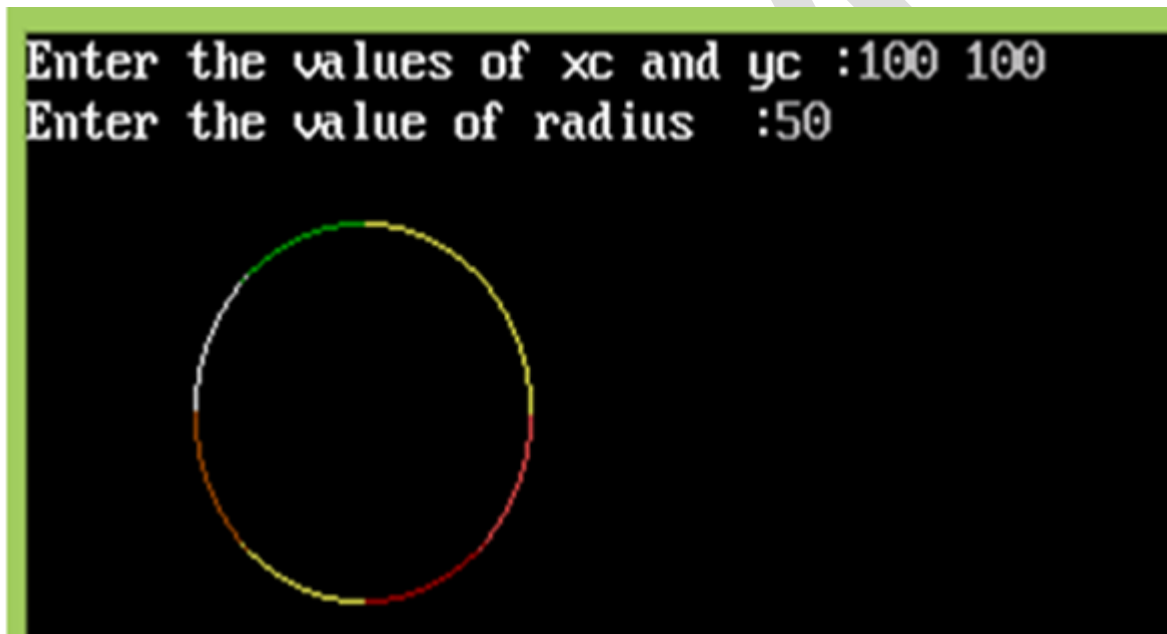
```

```

56. }
57.     printf("Enter the values of xc and yc :");
58.     scanf("%d%d",&xc,&yc);
59.     printf("Enter the value of radius :");
60.     scanf("%d",&r);
61.     BresenhamCircle(xc,yc,r);
62.
63.     getch();
64.     closegraph();
65.     return 0;
66. }

```

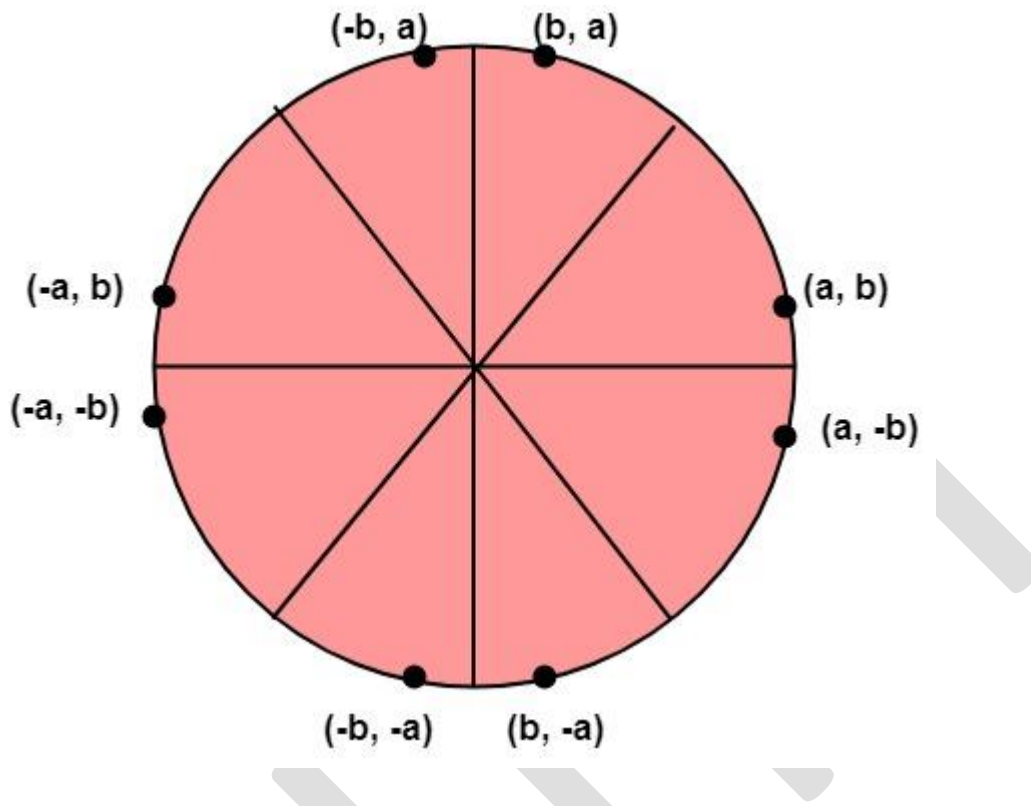
Output:



MidPoint Circle Algorithm

It is based on the following function for testing the spatial relationship between the arbitrary point (x, y) and a circle of radius r centered at the origin:

$$f(x, y) = x^2 + y^2 - r^2 \quad \left[\begin{array}{l} < 0 \text{ for } (x, y) \text{ inside the circle} \\ = 0 \text{ for } (x, y) \text{ on the circle} \\ > 0 \text{ for } (x, y) \text{ outside the circle} \end{array} \right] \dots \text{equation 1}$$



Now, consider the coordinates of the point halfway between pixel T and pixel S

This is called midpoint $(x_{i+1}, y_i - \frac{1}{2})$ and we use it to define a decision parameter:

$$P_i = f(x_{i+1}, y_i - \frac{1}{2}) = (x_{i+1})^2 + (y_i - \frac{1}{2})^2 - r^2 \dots \text{equation 2}$$

If P_i is -ve \Rightarrow midpoint is inside the circle and we choose pixel T

If P_i is +ve \Rightarrow midpoint is outside the circle (or on the circle) and we choose pixel S.

The decision parameter for the next step is:

$$P_{i+1} = (x_{i+1} + 1)^2 + (y_{i+1} - \frac{1}{2})^2 - r^2 \dots \text{equation 3}$$

Since $x_{i+1} = x_{i+1}$, we have

$$\begin{aligned}
P_{i+1} - P_i &= ((x_i + 1) + 1)^2 - (x_i + 1)^2 + (y_{i+1} - \frac{1}{2})^2 - (y_i - \frac{1}{2})^2 \\
&= x_i^2 + 4 + 4x_i - x_i^2 + 1 - 2x_i + y_{i+1}^2 + \frac{1}{4} - y_{i+1} - y_i^2 - \frac{1}{4} - y_i \\
&= 2(x_i + 1) + 1 + (y_{i+1}^2 - y_i^2) - (y_{i+1} - y_i) \dots \text{equation 4} \\
P_{i+1} &= P_i + 2(x_i + 1) + 1 + (y_{i+1}^2 - y_i^2) - (y_{i+1} - y_i) \dots \text{equation 4}
\end{aligned}$$

If pixel T is chosen $\Rightarrow P_i < 0$

We have $y_{i+1} = y_i$

If pixel S is chosen $\Rightarrow P_i \geq 0$

We have $y_{i+1} = y_i - 1$

$$\text{Thus, } P_{i+1} = \begin{cases} P_i + 2(x_i + 1) + 1, & \text{if } P_i < 0 \\ P_i + 2(x_i + 1) + 1 - 2(y_i - 1), & \text{if } P_i \geq 0 \end{cases} \dots \text{equation 5}$$

We can continue to simplify this in terms of (x_i, y_i) and get

$$P_{i+1} = \begin{cases} P_i + 2x_i + 3, & \text{if } P_i < 0 \\ P_i + 2(x_i - y_i) + 5, & \text{if } P_i \geq 0 \end{cases} \dots \text{equation 6}$$

Now, initial value of $P_i (0, r)$ from equation 2

$$\begin{aligned}
P_1 &= (0 + 1)^2 + (r - \frac{1}{2})^2 - r^2 \\
&= 1 + \frac{1}{4} - r^2 = \frac{5}{4} - r
\end{aligned}$$

We can put $\frac{5}{4} \cong 1$
 $\therefore r$ is an integer
 So, $P_1 = 1 - r$

Algorithm:

Step1: Put $x = 0, y = r$ in equation 2
 We have $p = 1 - r$

Step2: Repeat steps while $x \leq y$
 Plot (x, y)
 If $(p < 0)$
 Then set $p = p + 2x + 3$
 Else
 $p = p + 2(x - y) + 5$

```
y = y - 1 (end if)
x = x + 1 (end loop)
```

Step3: End

Program to draw a circle using Midpoint Algorithm:

```
1. #include <graphics.h>
2. #include <stdlib.h>
3. #include <math.h>
4. #include <stdio.h>
5. #include <conio.h>
6. #include <iostream.h>
7.
8. class bresen
9. {
10.     float x, y, a, b, r, p;
11.     public:
12.     void get ();
13.     void cal ();
14. };
15. void main ()
16. {
17.     bresen b;
18.     b.get ();
19.     b.cal ();
20.     getch ();
21. }
22. Void bresen :: get ()
23. {
24.     cout<<"ENTER CENTER AND RADIUS";
25.     cout<<"ENTER (a, b)";
26.     cin>>a>>b;
27.     cout<<"ENTER r";
28.     cin>>r;
29. }
30. void bresen :: cal ()
31. {
32.     /* request auto detection */
33.     int gdriver = DETECT, gmode, errorcode;
34.     int midx, midy, i;
35.     /* initialize graphics and local variables */
36.     initgraph (&gdriver, &gmode, " ");
37.     /* read result of initialization */
38.     errorcode = graphresult ();
39.     if (errorcode != grOK)    /*an error occurred */
```

```

40. {
41.     printf("Graphics error: %s \n", grapherrormsg (errorcode));
42.     printf ("Press any key to halt:");
43.     getch ();
44.     exit (1); /* terminate with an error code */
45. }
46. x=0;
47. y=r;
48. putpixel (a, b+r, RED);
49. putpixel (a, b-r, RED);
50. putpixel (a-r, b, RED);
51. putpixel (a+r, b, RED);
52. p=5/4)-r;
53. while (x<=y)
54. {
55.     If (p<0)
56.         p+= (4*x)+6;
57.     else
58.     {
59.         p+=(2*(x-y))+5;
60.         y--;
61.     }
62.     x++;
63.     putpixel (a+x, b+y, RED);
64.     putpixel (a-x, b+y, RED);
65.     putpixel (a+x, b-y, RED);
66.     putpixel (a-x, b-y, RED);
67.     putpixel (a+x, b+y, RED);
68.     putpixel (a-x, b-y, RED);
69.     putpixel (a-x, b+y, RED);
70.     putpixel (a-x, b-y, RED);
71. }
72. }

```

Output:

ENTER CENTER AND RADIUS

ENTER (a, b) 319, 239

ENTER r 100

