Unit 3
Two-Dimensional Algorithms

Line Drawing Algorithms

The Cartesian slope-intercept equation of a straight line is:

$$y = mx + b \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (1)$$

Where m = slope of line and b = y-intercept.

For any two given points (x_1, y_1) and (x_2, y_2)

$$\text{m} = \frac{y_2 - y_1}{x_2 - x_1}$$

$\therefore$ (1) becomes,

$$b = y - \frac{y_2 - y_1}{x_2 - x_1} x$$

At any point (x_k, y_k),

$$y_k = mx_k + b \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (2)$$

At (x_{k+1}, y_{k+1}),

$$y_{k+1} = mx_{k+1} + b \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (3)$$

Subtracting (2) from (3) we get,

$y_{k+1} - y_k = m(x_{k+1} - x_k)$

Here $(y_{k+1} - y_k)$ is increment in y as corresponding increment in x.

$\therefore \Delta y = m.\Delta x$

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

DDA line Algorithm (Incremental algorithm)

The digital differential analyzer (DDA) is a scan conversion line drawing algorithm based on calculating either Δx or Δy form the equation,

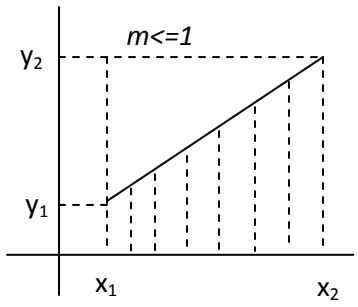
$$\Delta y = m.\Delta x$$

We sample the line at unit intervals in one co-ordinate and determine the corresponding integer values nearest to the line path for the other co-ordinates.

Consider first the line with positive slope.

If m<=1, we sample x co-ordinate. So $\Delta x = 1$ and compute each successive y value as:

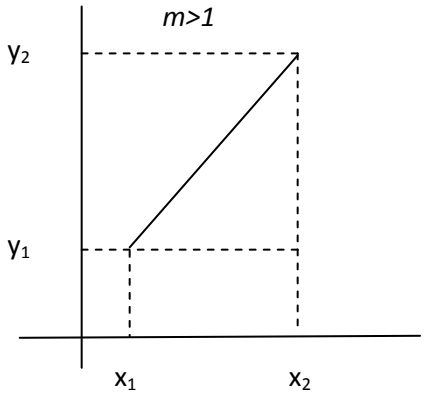
$$y_{k+1} = y_k + m \quad \because m = \frac{\Delta y}{\Delta x}, \Delta x = 1$$



Here *k* takes value from starting point and increase by 1 until final end point. *m* can be any real value between 0 and 1.

For line with positive slope greater than 1, we sample $\Delta y = 1$ and calculate corresponding x values as

$$x_{k+1} = x_k + \frac{1}{m} \quad \because m = \frac{\Delta y}{\Delta x}, \Delta y = 1$$



The above equations are under the assumption that the lines are processed from left to right i.e. left end point is starting. If the processing is from right to left, we can sample $\Delta y = -1$ for line |m|<1

$$\therefore y_{k+1} = y_k - m$$

If |m|>1, $\Delta y = -1$ and calculate

$$x_{k+1} = x_k - \frac{1}{m}.$$

Problem: Floating point multiplication & addition

C function for DDA algorithm

```
void lineDDA (in x1, int y1, int x2, int y2)
{
        int dx, dy, steps, k;
        float incrx, incry, x,y;
        dx = x2-x1;
        dy = y2-y1;
        if (abs(dx) > abs(dy))
                steps = abs(dx);
        else
                steps = abs(dy);
        incrx = dx/steps;
        incry = dy/steps;
        x = x1;   /* first point to plot */
        y = y1;
        putpixel(round(x), round(y),1);       //1 is color
parameter
        for (k = 1;k <= steps;k++)
        {
                x = x + incrx;
                y = y + incry;
                putpixel(round(x),round(y),1);
        }
}
```

The DDA algorithm is faster method for calculating pixel position but it has problems:

- *m* is stored in floating point number.
- round of error
- Error accumulates as we precede line.
- so line will move away from actual line path for long line

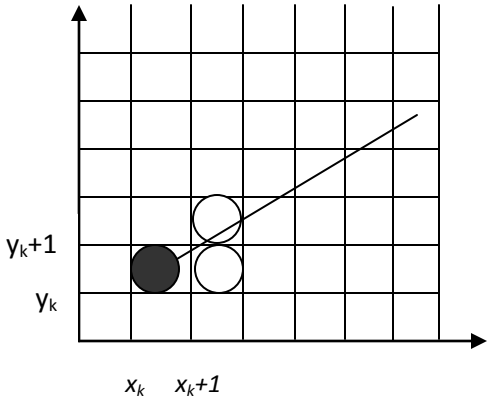
Bresenham's Line algorithm

An accurate and efficient line generating algorithm, developed by Bresenham that scan converts lines only using integer calculation to find the next *(x, y)* position to plot. It avoids incremental error accumulation.

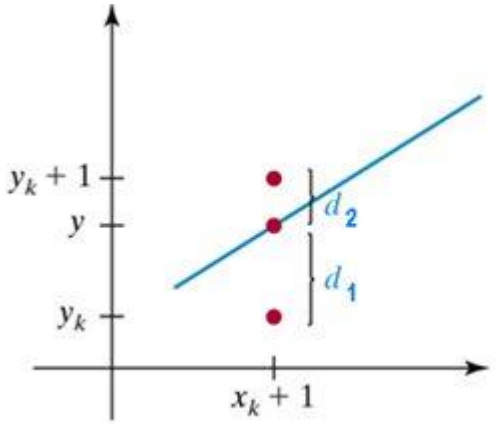
Line with positive slope less than 1 (0<m<1)

Pixel position along the line path is determined by sampling at unit x intervals. Starting from left end point, we step to each successive column and plot the pixel closest to line path.

Assume that (x_k, y_k) is pixel at k^{th} step then next point to plot may be either $(x_k + 1, y_k)$ or $(x_k + 1, y_k + 1)$.



At sampling position x_k+1, we label vertical pixel separation from line path as d_1 & d_2 as in figure.



The y-coordinate on the mathematical line path at pixel column x_k+1 is $y = m(x_k+1)+b$.

Then $d_1 = y - y_k = m(x_k + 1) + b - y_k$

$d_2 = (y_k + 1) - y = (y_k + 1) - m(x_k + 1) - b$

Now $d_1 - d_2 = 2m(x_k + 1) - (y_k + 1) - y_k + 2b = 2m(x_k + 1) - 2y_k + 2b - 1$

A decision parameter p_k for the k^{th} step in the line algorithm can be obtained by rearranging above equation so that it involves only integer calculations. We accomplish this by substituting $m = \frac{\Delta y}{\Delta x}$ in above eq^n and defining

$$p_k = \Delta x(d_1 - d_2) = \Delta x[2\frac{\Delta y}{\Delta x}(x_k + 1) - 2y_k + 2b - 1] = 2\Delta y.x_k - 2\Delta x.y_k + c$$

where the constant $c = 2\Delta y - \Delta x(2b-1)$ which is independent of the pixel position. Also, sign of p_k is same as the sign of d_1-d_2.

If decision parameter p_k is negative i.e. $d_1 < d_2$, pixel at y_k is closer to the line path than pixel at y_k+1. In this case we plot lower pixel *(x_k+1 , y_k), other wise plot upper pixel (x_k+1, y_k+1).*

Co-ordinate change along the line occur in unit steps in either x, or y direction. Therefore we can obtain the values of successive decision parameters using incremental integer calculations.
At step *k+1,* decision parameter p_{k+1} *is* evaluated as.

$$p_{k+1} = 2\Delta y.x_{k+1} - 2\Delta x y_{k+1} + c$$

$$\therefore p_{k+1} - p_k = 2\Delta y(x_{k+1} - x_k) - 2\Delta x(y_{k+1} - y_k)$$

Since $x_{k+1} = x_k + 1$

$$\therefore p_{k+1} = p_k + 2\Delta y - 2\Delta x(y_{k+1} - y_k)$$

The term $y_{k+1} - y_k$ is either 0 or 1 depending upon the sign of p_k.

The first decision parameter p_0 is evaluated as.

$$p_o = 2\Delta y - \Delta x$$

and successively we can calculate decision parameter as

$$p_{k+1} = p_k + 2\Delta y - 2\Delta x(y_{k+1} - y_k)$$

So if p_k is negative, $y_{k+1} = y_k$ so $p_{k+1} = p_k + 2\Delta y$

Otherwise $y_{k+1} = y_k + 1$, *then* $p_{k+1} = p_k + 2\Delta y - 2\Delta x$

<u>Algorithm:</u>

1. Input the two line endpoint and store the left endpoint at (x_o, y_o)
2. Load (x_o, y_o) in to frame buffer, i.e. Plot the first point.
3. Calculate constants $2\Delta x, 2\Delta y$ calculating $\Delta x, \Delta y$ and obtain first decision parameter value as
   $$p_o = 2\Delta y - \Delta x$$
4. At each x_k along the line, starting at k=0, perform the following test,
   if $p_k < 0$, next point is $(x_k + 1, y_k)$
   $$p_{k+1} = p_k + 2\Delta y$$
   otherwise
   next point to plot is $(x_k + 1, y_k + 1)$
   $$p_{k+1} = p_k + 2\Delta y - 2\Delta x$$
5. Repeat step (4) Δx times.

C Implementation

```
void lineBresenham (int x1, int y1, int x2, int y2)
{
        int dx = abs(x2-x1), dy=abs(y2-y1);
        int  pk, xEnd;
        pk=2*dy-dx;
        //determine which point to use as start, which as end
        if(x1>x2){
                x = x2;
                y = y2;
                xEnd = x1;
        }
         else {
                x = x1;
                y = y1;
                xEnd = x2;
        }
        putixel (x,y,1);
        while (x < xEnd)
        {
                x++;
                if(pk<0)
                        pk=pk+2*dy;
                else
                {
                        Y++;
                        pk= pk+2*dy-2*dx
                 }
                putpixel (x,y,1);
         }
}
```

Bresenham's algorithm is generalized to lines with **arbitrary slope** by considering the symmetry between the various octants & quadrants of xy-plane.

Line with positive slope greater than 1 (m>1)

Here, we simply interchange the role of x & y in the above procedure i.e. we step along the y-direction in unit steps and calculate successive x values nearest the line path.

Circle generating algorithms

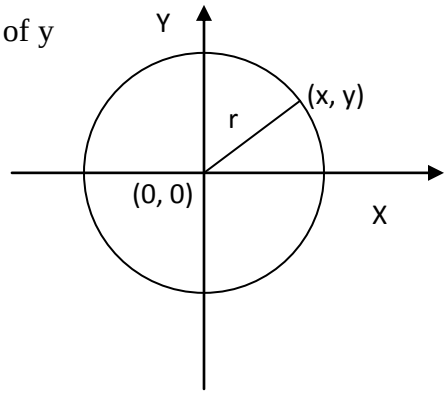
Circle is a frequently used component in pictures and graphs, a procedure for generating circular arcs or full circles is included in most graphics packages.

Simple Algorithm

The equation of circle centered at origin and radius r is given by $x^2 + y^2 = r^2$

$\Rightarrow y = \pm\sqrt{r^2 - x^2}$

- Increment x in unit steps and determine corresponding value of y from the equation above. Then set pixel at position *(x,y).*
- The steps are taken from *–r* to *+r*.
- In computer graphics, we take origin at upper left corner point on the display screen i.e. first pixel of the screen. So any visible circle drawn would be centered at point other than (0,0). If center of circle is *(xc, yc)* then the calculated points from origin center should be moved to pixel position by *(x+xc, y+yc).*



In general the equation of circle centered at *(xc, yc)* and radius r is

$$(x - xc)^2 + (y - yc)^2 = r^2$$

$$\Rightarrow y = yc \pm \sqrt{r^2 - (x - xc^2} \quad \text{..... (1)}$$

We use this equation to calculate the position of points on the circle. Take unit step from xc-r to xc+r for x value and calculate the corresponding value of y-position for pixel position (x, y). This algorithm is simple but,

- Time consuming – square root and squares computations.
- Non-uniform spacing, due to changing slope of curve. If non-uniform spacing is avoided by interchanging x and y for slope |m|>y, this leads to more computation.

Following program demonstrates the simple computation of circle using the above equation (1)

```
//program for circle (simple algorithm)
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#include<math.h>
#include<graphics.h>
#define SQUARE(x) ((x)*(x))
void drawcircle(int ,int,int);
void main()
{
        int gd=DETECT,gm;
        initgraph(&gd,&gm,"c:\\tc\\bgi");
        int xc,yc,r;
        xc=getmaxx()/2;
        yc=getmaxy()/2;
        r=50;
        drawcircle(xc,yc,r);
        getch();
        closegraph();
}//end main
void drawcircle(int xc,int yc,int r)
{
        int i,x,y,y1;
```

```
for(i=xc-r;i<=xc+r;i++)
{
        x=i;
        y=yc+sqrt(SQUARE(r)-SQUARE(x-xc));
        y1=yc-sqrt(SQUARE(r)-SQUARE(x-xc));
        putpixel(x,y,1);
        putpixel(x,y1,1);
}
```

Drawing circle using polar equations

If (x,y) be any point on the circle boundary with center (0,0) and radius r, then

$x = r\cos\theta$

$y = r\sin\theta$

i.e. $(x, y) = (r\cos\theta, r\sin\theta)$

To draw circle using these co-ordinates approach, just increment angle starting from 0 to 360. Compute (x,y) position corresponding to increment angle. Which draws circle centered at origin, but the circle centered at origin is not visible completely on the screen since (0, 0) is the starting pixel of the screen. If center of circle is given by (xc, yc) then the pixel position (x, y) on the circle path will be computed as

$x = xc + r\cos\theta$

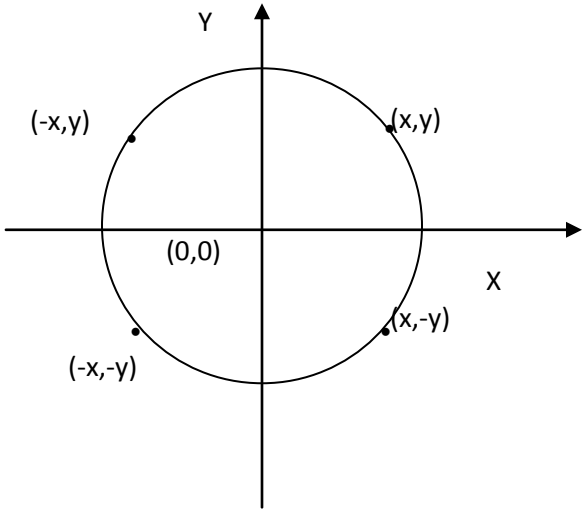
$y = yc + r\sin\theta$

polarcircle() Function to draw circle using the polar transformation:

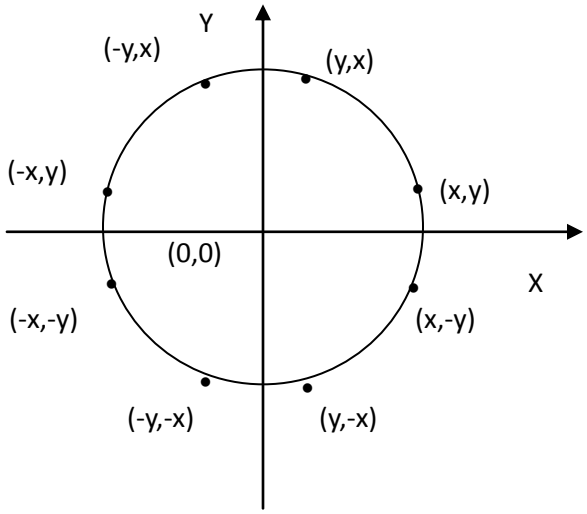
```
void polarcircle(int xc, int yc, int r)
{
        int x,y;
        float theta;
        const float PI=3.14;

        for(theta=0.0;theta<=360;theta+=1)
        {
                x= xc+r*cos(theta*PI/180.0);
                y= yc+r*sin(theta*PI/180.0);
                putpixel(x,y,1);

        }
}
```

Symmetry in circle scan conversion

We can reduce the time required for circle generation by using the symmetries in a circle e.g. 4-way or 8-way symmetry. So we only need to generate the points for one quadrant or octants and then use the symmetry to determine all the other points.



4-way symmetry



8-way symmetry

Problem of computation still persists using symmetry since there are square roots; trigonometric functions are still not eliminated in above algorithms.

Mid point circle Algorithm

In mid point circle algorithm, we sample at unit intervals and determine the closest pixel position to the specified circle path at each step.

For a given radius r, and screen center position (xc, yc), we can first set up our algorithm to calculate pixel positions around a circle path centered at (0, 0) and then each calculated pixel position (x, y) is moved to its proper position by adding xc to x and yc to y

$$i.e.\ x = x + xc,\ y = y + yc.$$

To apply the mid point method, we define a circle function as:

$$f_{circle} = x^2 + y^2 - r^2$$

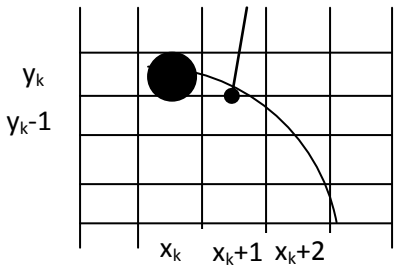
To summarize the relative position of point (x, y) by checking sign of f_{circle} function,

$$f_{circle}(x, y) \begin{cases} <0, \text{ if } (x, y) \text{ lies inside the circle boundary} \\ =0, \text{ if } (x, y) \text{ lies on the circle boundary} \\ >0, \text{ if } (x, y) \text{ lies outside the circle boundary.} \end{cases}$$

The circle function tests are performed for the mid positions between pixels near the circle path at each sampling step. Thus the circle function is decision parameter in mid point algorithm and we can set up incremental calculations for this functions as we did in the line algorithm.

Mid point

The figure, shows the mid point between the two candidate pixels at sampling position $x_k + 1$, Assuming we have just plotted the pixel (x_k, y_k), we next need to determine whether the pixel at position $(x_k + 1, y_k) or (x_k + 1, y_k - 1)$ is closer to the circle.



Our decision parameter is circle function evaluated at the mid point

$$p_k = f_{circle}(x_k + 1, y_k - \frac{1}{2})$$

$$= (x_k + 1)^2 + (y_k - \frac{1}{2})^2 - r^2 = (x_k + 1)^2 + y_k^2 - y_k + \frac{1}{4} - r^2$$

If $p_k < 0$, then mid-point lies inside the circle, so point at y_k is closer to boundary otherwise, $y_k - 1$ closer to choose next pixel position.

Successive decision parameters are obtained by incremental calculation. The decision parameter for next position is calculated by evaluating circle function at sampling position $x_{k+1} + 1$ i.e. $x_k + 2$ as

$$p_{k+1} = f_{circle}(x_{k+1} + 1, y_{k+1} - \frac{1}{2})$$

$$= \{(x_{k+1} + 1)\}^2 + (y_{k+1} - \frac{1}{2})^2 - r^2$$

$$= (x_{k+1})^2 + 2x_{k+1} + 1 + (y_{k+1})^2 - (y_{k+1}) + \frac{1}{4} - r^2$$

Now, $p_{k+1} - p_k = 2x_{k+1} + (y^2{}_{k+1} - y_k^2) - (y_{k+1} - y_k) + 1$

i.e. $p_{k+1} = p_k + 2x_{k+1} + (y^2{}_{k+1} - y_k^2) - (y_{k+1} - y_k) + 1$

Where y_{k+1} is either y_k or $y_k - 1$ depending upon sign of p_k. and $x_{k+1} = x_k + 1$

If p_k is negative, $y_{k+1} = y_k$ so we get,

$$p_{k+1} = p_k + 2x_{k+1} + 1$$

If p_k is positive, $y_{k+1} = y_k - 1$ so we get,

$$p_{k+1} = p_k + 2x_{k+1} + 1 - 2y_{k+1}$$

Where $2x_{k+1} = 2x_k + 2$

$$2y_{k+1} = 2y_k - 2$$

At the start position, (0, r), these two terms have the values 0 and 2r, respectively. Each successive values are obtained by adding 2 to the previous value of 2x and subtracting 2 from previous value of 2y.

The initial decision parameter is obtained by evaluating the circle function at starting position $(x_0, y_0) = (0, r)$.

$$p_0 = f_{circle}(1, r - 1/2)$$
$$= 1 + (r - 1/2)^2 - r^2$$
$$= 1 + r^2 - r + \frac{1}{4} - r^2$$
$$= \frac{5}{4} - r$$

If p_0 is specified in integer,

$$p_0 = 1 - r.$$

Steps of Mid-point circle algorithm

1. Input radius r and circle centre (x_c, y_c) and obtain the first point on circle centered at origin as.
   $$(x_0, y_0) = (0, r).$$
2. Calculate initial decision parameter
   $$p_0 = 5/4 - r$$
3. At each x_k position, starting at $k = 0,$ perform the tests:
   If $p_k < 0$ next point along the circle centre at $(0,0)$ is $(x_k + 1, y_k)$
   $$p_{k+1} = p_k + 2x_{k+1} + 1)$$
   Otherwise, the next point along circle is $(x_k + 1, y_k - 1)$
   $$p_{k+1} = p_k + 2x_{k+1} + 1 - 2y_{k+1}$$
   Where $2x_{k+1} = 2x_k + 2.$ and $2y_{k+1} = 2y_k - 2.$
4. Determine symmetry point on the other seven octants.
5. Move each calculated pixels positions (x, y) in to circle path centered at (x_c, y_c) as
   $$x = x + x_c, y = y + y_c$$
6. Repeat 3 through 5 until $x \geq y.$

```
//mid-point circle algorithm
#include <graphics.h>
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
void drawpoints(int,int,int,int);
void drawcircle(int,int,int);

void main(void)
{
  /* request auto detection */
```

```
   int gdriver = DETECT, gmode;
   /* initialize graphics and local variables */
  initgraph(&gdriver, &gmode, "c:\\tc\\bgi");

  int xc,yc,r;
  printf("Enter the center co-ordinates:");
  scanf("%d%d",&xc,&yc);
  printf("Enter the radius");
  scanf("%d",&r);
  circlemidpoint(xc,yc,r);
  getch();
  closegraph();
}
void drawpoints(int x,int y, int xc,int yc)
{
        putpixel(xc+x,yc+y,1);
        putpixel(xc-x,yc+y,1);
        putpixel(xc+x,yc-y,1);
        putpixel(xc-x,yc-y,1);
        putpixel(xc+y,yc+x,1);
        putpixel(xc-y,yc+x,1);
        putpixel(xc+y,yc-x,1);
        putpixel(xc-y,yc-x,1);
}
void circlemidpoint(int xc,int yc,int r)
{
        int x = 0, y=r, p = 1-r;
        drawpoints(x,y,xc,yc);
        while(x<y)
        {
                x++;
                if(p<0)
                        p += 2*x+1;
                else
                {
                        y--;
                        p += 2*(x-y)+1;
                }
                drawpoints(x,y,xc,yc);
        }
}
```

Ellipse Algorithm generating algorithm

Direct Method

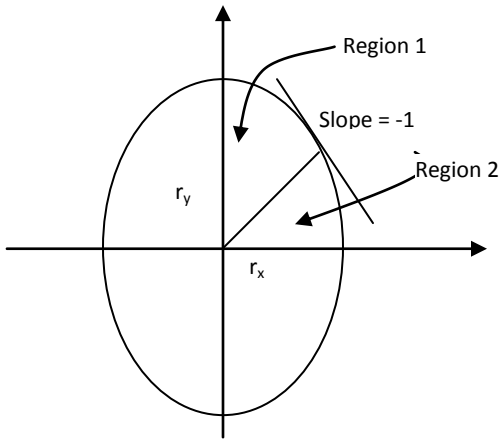
An ellipse is an elongated circle therefore the basic algorithm for drawing ellipse is same as circle computing x and y position at the boundary of the ellipse from the equation of ellipse directly.

We have equation of ellipse centered at origin (0,0) is

$\frac{x^2}{r_x^2} + \frac{y^2}{r_y^2} = 1$ which gives

$y = \pm \frac{r_y}{r_x} \sqrt{(r_x^2 - x^2)}$ ..... (1)

Stepping unit interval in x direction from $-r_x$ to r_x we can get corresponding y value at each x position which gives the ellipse boundary co-ordinates. Plotting these computed points we can get the ellipse.



If center of ellipse is any arbitrary point *(xc, yc)* then the equation of ellipse can be written as

$$\frac{(x - xc)^2}{r_x^2} + \frac{(y - yc)^2}{r_y^2} = 1$$

i.e. $y = yc \pm \frac{r_y}{r_x} \sqrt{r_x^2 - (x - xc)^2}$ ----------(2)

For any point *(x, y)* on the boundary of the ellipse If major axis of ellipse is along X-axis, then algorithm based on the direct computation of ellipse boundary points can be summarized as,

1. Input the center of ellipse *(xc, yc)* , x-radius *xr* and y-radius *yr*.
2. For each x position starting from xc-r and stepping unit interval along x-direction, compute corresponding y positions as

   $y = yc \pm \frac{r_y}{r_x} \sqrt{r_x^2 - (x - xc)^2}$
3. Plot the point (x, y).
4. Repeat step 2 to 3 until x >= xc+xr.

Computation of ellipse using polar co-ordinates

Using the polar co-ordinates for ellipse, we can compute the (x,y) position of the ellipse boundary using the following parametric equations

$$x = xc + r\cos\theta$$
$$y = yc + r\sin\theta$$

The algorithm based on these parametric equations on polar co-ordinates can be summarized as below.

1. Input center of ellipse *(xc,yc)* and radii xr and yr.
2. Starting θ from angle 0^o step minimum increments and compute boundary point of ellipse as $x = xc + r\cos\theta$
   $y = yc + r\sin\theta$
3. Plot the point at position (round(x), round(y))
4. Repeat until θ is greater or equal to 360^o.

C implementation

```
void drawellipse(int xc,int yc,int rx,int ry)
{
        int x,y;
        float theta;
        const float PI=3.14;
        for(theta=0.0;theta<=360;theta+=1)
        {
                x= xc+rx*cos(theta*PI/180.0);
                y= yc+ry*sin(theta*PI/180.0);
                putpixel(x,y,1);

        }
}
```

The methods of drawing ellipses explained above are not efficient. The method based on direct equation of ellipse must perform the square and square root operations due to which there may be floating point number computation which cause rounding off to plot the pixels. Due to the changing slope of curve along the path of ellipse, there may be un-uniform separation of pixel when slope changes. Although , the method based on polar co-ordinate parametric equation gives the uniform spacing of pixel due to uniform increment of angle but it also take extra computation to evaluate the trigonometric functions. So these algorithms are **not efficient** to construct the ellipse. We have another algorithm called mid-point ellipse algorithm similar to raster mid-point circle algorithm and is **efficient** one.

Mid-Point Ellipse Algorithm

The mid-point ellipse algorithm decides which point near the boundary (i.e. path of the ellipse) is closer to the actual ellipse path described by the ellipse equation. That point is taken as next point.

- This algorithm is applied to the first quadrant in two parts as in fig Region 1 and Region 2. We process by taking unit steps in x-coordinates direction and finding the closest value for y for each x-step in region 1.
- In first quadrant at region 1, we start at position *(0, r_y)* and incrementing x and calculating y closer to the path along clockwise direction. When slope becomes -1

then shift unit step in x to y and compute corresponding x closest to ellipse path at Region 2 in same direction.

- Alternatively, we can start at position *(r_x, 0)* and select point in counterclockwise order shifting unit steps in y to unit step in x when slope becomes greater than -1.

Here, to implement mid-point ellipse algorithm, we take start position at *(0, r_y)* and step along the ellipse path in clockwise position throughout the first quadrant.

We define ellipse function center at origin i.e. *(xc, yc) = (0, 0)* as

$$f_{ellipse}(x, y) = r_y^2 x^2 + r_x^2 y^2 - r_x^2 r_y^2$$

$$f_{ellipse}(x, y) = \begin{cases} < 0, \text{ if (x, y) lies inside boundary of ellipse} \\ = 0 \text{ if (x, y lies on the boundary of ellipse} \\ >0 \text{ if(x, y) lies outside the boundary of ellipse} \end{cases}$$

So $f_{ellipse}$ function serves as decision parameter in ellipse algorithm at each sampling position. We select the next pixel position according to the sign of decision parameter.

Starting at (0, r_y) , we take unit step in x-direction until we reach the boundary between the region 1 and region 2. Then we switch unit steps in y over the remainder of the curve in first quadrant. At each step, we need to test the slope of curve. The slope of curve is calculated as;

$$\frac{dy}{dx} = -\frac{2r_y^2 x}{2r_x^2 y}$$

At the boundary between region 1 and region 2,

$\frac{dy}{dx} = -1$ and $2r_y^2 = 2r_x^2$ Therefore , we move out of region 1 when $2r_y^2 x \geq 2r_x^2$

Assuming the position (x_k, y_k) is filled, we move x_{k+1} to determine next pixel. The corresponding y value for x_{k+1} position will be either y_k or $y_k - 1$ depending upon the sign of decision parameter. So the decision parameter for region 1 is tested at mid point of $(x_k + 1, y_k)$ and $(x_k + 1, y_k - 1)$ i.e.

$$p_{1k} = f_{ellipse}(x_{k+1}, y_k - \frac{1}{2})$$

or $$p_{1k} = r_y^2 (x_{k+1})^2 + r_x^2 (y_k - \frac{1}{2})^2 - r_x^2 r_y^2$$

or $$p_{1k} = r_y^2 (x_{k+1})^2 + r_x^2 y_k^2 - r_x^2 y_k + \frac{r_x^2}{4} - r_x^2 r_y^2 \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (1)$$

if $p_{1k} < 0,$ the mid point lies inside boundary, so next point to plot is $(x_k + 1, y_k)$ otherwise, next point to plot will be $(x_k + 1, y_k - 1)$

The successive decision parameter is computed as

$$p_{1k+1} = f_{ellipse}(x_{k+1} + 1, y_{k+1} - \frac{1}{2})$$

$$= r_y^{\,2}(x_{k+1}+1)^2 + r_x^{\,2}(y_{k+1}-\frac{1}{2})^2 - r_x^{\,2}r_y^{\,2}$$

Or, $p_{1k+1} = r_y^{\,2}(x_{k+1}^2 + 2x_{k+1} + 1) + r_x^{\,2}(y_{k+1}^2 - y_{k+1} + \frac{1}{4}) - r_x^{\,2}r_y^{\,2}$

Or, $p_{1k+1} = r_y^{\,2}x_{k+1}^2 + 2r_y^{\,2}x_{k+1} + r_y^2 + r_x^{\,2}y_{k+1}^2 - r_x^{\,2}y_{k+1} + \frac{r_x^{\,2}}{4} - r_x^{\,2}r_y^{\,2}$ .....(2)

Subtracting (2) - (1)

$p_{1k+1} - p_{1k} = 2r_y^{\,2}x_{k+1} + r_y^2 + r_x^{\,2}(y_{k+1}^2 - y_k^2) - r_x^2(y_{k+1} - y_k)$

if $p_{1k} < 0,\ y_{k+1} = y_k$ then,

$\therefore p_{1k+1} = p_{1k} + 2r_y^{\,2}x_{k+1} + r_y^{\,2}$

Otherwise $y_{k+1} = y_k - 1$ then we get,

$p_{1k+1} = p_{1k} + 2r_y^{\,2}x_{k+1} + r_y^{\,2} - 2r_x^{\,2}y_{k+1}$

At the initial position, $(0, r_y)$ $2r_y^{\,2}x = 0$ and $2r_x^{\,2}y = 2r_x^2r_y$

In region 1, initial decision parameter is obtained by evaluating ellipse function at $(0, r_y)$ as

$p_{10} = f_{ellipse}(1, r_y - \frac{1}{2})$

Or, $p_{10} = f_{ellipse}(1, r_y - \frac{1}{2})$

$= r_y^2 - r_x^2r_y + \frac{1}{4}r_x^{\,2}$

Similarly, over the region 2, the decision parameter is tested at mid point of $(x_k, y_k - 1)$ and $(x_k + 1, y_k - 1)$ i.e.

$p_{2k} = f_{ellipse}(x_k + \frac{1}{2}, y_k - 1)$

$= r_y^{\,2}(x_k + \frac{1}{2})^2 + r_x^{\,2}(y_k - 1)^2 - r_x^{\,2}r_y^{\,2}$

$\therefore p_{2k} = r_y^{\,2}x_k^{\,2} + r_y^2x_k + \frac{r_y^2}{4} + r_x^{\,2}(y_k - 1)^2 - r_x^{\,2}r_y^{\,2}$ .....(3)

if $p_{2k} > 0,$ the mid point lies outside the boundary, so next point to plot is $(x_k, y_k - 1)$ otherwise, next point to plot will be $(x_k + 1, y_k - 1)$

The successive decision parameter is computed as evaluating ellipse function at mid point of

$p_{2k+1} = f_{ellipse}(x_{k+1} + \frac{1}{2}, y_{k+1} - 1)$ with $y_{k+1} = y_k - 1$

$$p_{2k+1} = r_y^2(x_{k+1} + \frac{1}{2})^2 + r_x^2[(y_k - 1) - 1]^2 - r_x^2 r_y^2$$

Or $p_{2k+1} = r_y^2 x_{k+1}^2 + r_y^2 x_{k+1} + \frac{r_y^2}{4} + r_x^2(y_k - 1)^2 - 2r_x^2(y_k - 1) + r_x^2 - r_x^2 r_y^2$ .....(4)

Subtracting (4)-(3)

$$p_{2k+1} - p_{2k} = r_y^2(x_{k+1}^2 - x_k^2) + r_y^2(x_{k+1} - x_k) - 2r_x^2(y_k - 1) + r_x^2$$

Or $p_{2k+1} = p_{2k} + r_y^2(x_{k+1}^2 - x_k^2) + r_y^2(x_{k+1} - x_k) - 2r_x^2(y_k - 1) + r_x^2$

if $p_{2k} > 0$, $x_{k+1} = x_k$ then

$$p_{2k+1} = p_{2k} - 2r_x^2(y_k - 1) + r_x^2$$

Otherwise $x_{k+1} = x_k + 1$ then

$$p_{2k+1} = p_{2k} + r_y^2[(x_k + 1)^2 - x_k^2)] + r_y^2(x_k + 1 - x_k) - 2r_x^2(y_k - 1) + r_x^2$$

Or, $p_{2k+1} = p_{2k} + r_y^2(2x_k + 1) + r_y^2 - 2r_x^2(y_k - 1) + r_x^2$

Or, $p_{2k+1} = p_{2k} + r_y^2(2x_k + 2) - 2r_x^2(y_k - 1) + r_x^2$

Or, $p_{2k+1} = p_{2k} + 2r_y^2 x_{k+1} - 2r_x^2 y_{k+1} + r_x^2$ where $x_{k+1} = x_k + 1$ and $y_{k+1} = y_k - 1$

The initial position for region 2 is taken as last position selected in region 1 say which is (x_0, y_0) then initial decision parameter in region 2 is obtained by evaluating ellipse function at mid point of (x_0, y_0-1) and (x_0+1, y_0-1) as

$$p_{20} = f_{ellipse}(x_0 + \frac{1}{2}, y_0 - 1)$$

$$= r_y^2(x_0 + \frac{1}{2})^2 + r_x^2(y_0 - 1)^2 - r_x^2 r_y^2$$

Now the mid-point **ellipse algorithm is summarized** as;

1. Input center (xc, yc) and r_x and r_y for the ellipse and obtain the first point as $(x_0, y_0) = (0, r_y)$
2. Calculate initial decision parameter value in Region 1 as

   $$P_{10} = r_y^2 - r_x^2 r_y + \frac{1}{4} r_x^2$$

3. At each x_k position, in Region 1, starting at k = 0 , compute $x_{k+1} = x_k + 1$

   If $p_{1k} < 0$, then the next point to plot is

   $$p_{1k+1} = p_{1k} + 2r_y^2 x_{k+1} + r_y^2$$

   $y_{k+1} = y_k$

   Otherwise next point to plot is

   $y_{k+1} = y_k - 1$

$$p_{1k+1} = p_{1k} + 2r_y{}^2 x_{k+1} + r_y{}^2 - 2r_x{}^2 y_{k+1} \qquad \text{with } x_{k+1} = x_k + 1 \text{ and } y_{k+1} = y_k - 1$$

4. Calculate the initial value of decision parameter at region 2 using last calculated point say (x_0,y_0) in region 1 as

$$p_{20} = r_y^2(x_0 + \frac{1}{2})^2 + r_x^2(y_0 - 1)^2 - r_x^2 r_y^2$$

5. At each y_k position in Region 2 starting at k = 0, perform computation $y_{k+1} = y-1$;
   if $p_{2k} > 0$, then

   $x_{k+1} = x_k$

   $$p_{2k+1} = p_{2k} - 2r_x^2(y_k - 1) + r_x^2$$

   Otherwise

   $x_{k+1} = x_k + 1$

   $p_{2k+1} = p_{2k} + 2r_y^2 x_{k+1} - 2r_x^2 y_{k+1} + r_x^2$ where $x_{k+1} = x_k + 1$ and $y_{k+1} = y_k - 1$

6. Determine the symmetry points in other 3 quadrants.
7. Move each calculated point (x_k, y_k) on to the centered (xc,yc) ellipse path as

   $x_k = x_k + xc;$

   $y_k = y_k + yc$

8. Repeat the process for region 1 until $2r_y^2 x_k \geq 2r_x^2 y_k$ and region until $(x_k, y_k) = (r_x,0)$