直线光栅化算法

- DDA算法
- Bresenham算法

圆光栅化算法

- 中点算法
- 中点整数算法
- 中点整数优化算法

2.1直线光栅化法

DDA算法 （ Digital Differential Analyzer)

- David F. Rogers 的描述 (适用于所有象限)
- James D. Foley的描述 (只适用于第一象限， 且 K<1)
- 本教程的描述 (适用于所有象限及任何端点)

Bresenham算法

- 基本原理
- Bresenham算法
- 整数 Bresenham算法
- 一般整数 Bresenham算法

2.1.1 DDA算法算法

1)David F. Rogers 描述描述

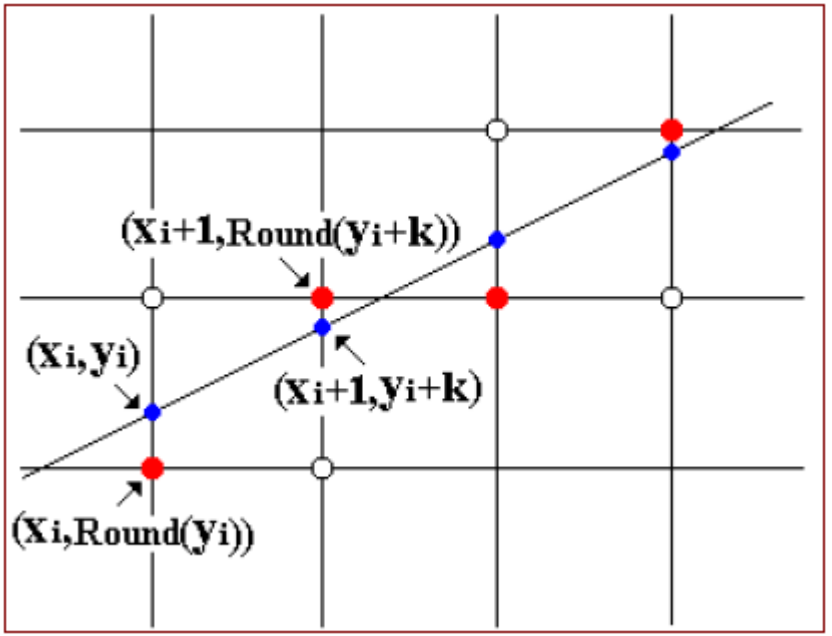
直线的基本微分方程是：

$$\frac{dy}{dx} = \text{常数 (k)}$$

设直线通过点 P1(x1,y1)和P2(x2,y2)，

则直线方程可表示为：

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



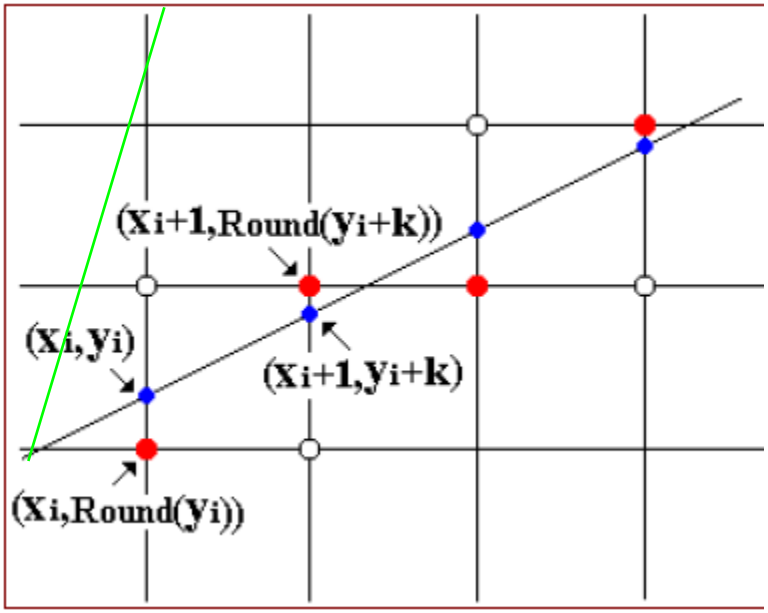
1)David F. Rogers 描述描述

- 如果已知第 i点的坐标， 可用步长 StepX 和 StepY得到

第 i+1点的坐标为：

  - xi+1=xi+ StepX
  - yi+1=yi+ StepY 或 yi+1=yi+ k * StepX

- 例图中
  - k<1
  - StepX=1
  - StepY=k
- 将算得的直线上每个点的当前坐标， 按四舍五入得到光栅点的位置



1) David F. Rogers 描述

```
// Digital Differential Analyzer (DDA ) routine for rasterizing a
      line
// The line end points are (xs,ys) and (xe,ye) assumed not equal.
// Round is the function.  Note: Many Round functions are
      floor functions,  i.e Round (-8.5)=-9 rather than -8. The
      algorithm assumes this is the case.
// Approximate the line length
If  (|xs- xe | ≥ | ys-ye |) then //插补长度
    Length ← |xs- xe |;
else
    Length ← | ys-ye |;
end if
```

1) David F. Rogers 描述描述

```
// Select the larger of  Δx or Δy to be one raster unit.
StepX = ( xe -xs) / Length;
StepY = ( ye -ys) / Length;
x = xs; //首点
y = ys;
i = 1; // Begin main loop
while (i≤ Length)
    WritePixel (Round(x), Round(y) ,value));
    x = x + StepX;
    y = y + StepY;
    i++;
end while
```

2) James D.Foley 描述描述

- 令 $$k=\frac{y_2-y_1}{x_2-x_1}$$

  有：

  y_{i+1}=yi+k * StepX

- 若 0<k<1， 即 Δx > Δy
  - 因光栅单位为 1，
  - 可以采用每次 x方向增加 1，
  - 而 y方向增加 k的办法得到下一个直线点 。

2) James D.Foley 描述描述

```
void Line ( //设 0≤ k≤ 1,xs<xe
        int xs,ys; //左端点
        int xe,ye; //右端点
        int value) //赋给线上的象数值
{
        int x; //x以步长为单位从 xs增长到 xe
        double dx =xe-xs;
        double dy =ye-ys;
        double k =dy/dx;  // 直线之斜率 k
        double y =ys;
        for (x=xs; x<=xe; x++)   {
                WritePixel(x,Round(y),value); //置象数值为 value
                y+=k; // y移动步长是斜率 k
        } // End of for
} // Line
```

3)已有算法描述分析

Rogers 描述：

- 采用 x = x + StepX， y = y + StepY，
- 逼近点并不是直线的一个最好的逼近；

D.Foley描述： 可能引起积累误差

- 未分析直线端点不在象素点上的情况；
- 只给出 0— 45^0第一个八卦限的描述 。

为避免引起积累误差， D.Foley描述中采用

- double dx =xe-xs;
- double dy =ye-ys;
- double k =dy/dx; // 直线之斜率 k

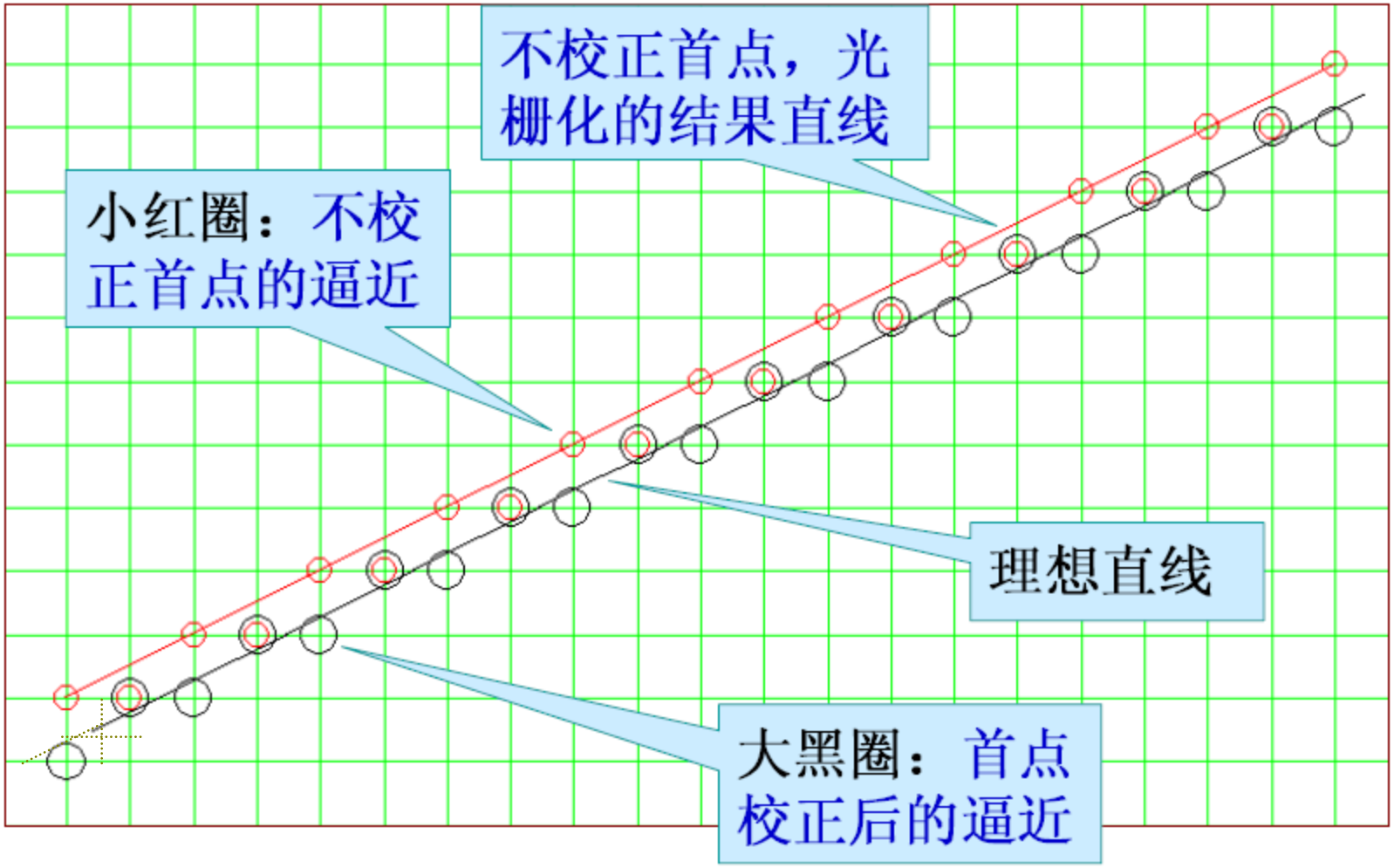
4)本教程描述——任意方向直线插补算法

```
void DDALine (
    float xs, ys;    //起点
    float xe, ye;    //终点
    int value) //赋给线上的象数值
{
    int n, ix, iy, idx, idy ;
    int Flag; //插补方向标记
    int Length; //插补长度
    float x, y, dx, dy;
```

```
dx=xe-xs;              dy=ye-ys;
if (fabs(dy)<fabs(dx)) {    //X方向长， 斜率 <=1
      Length=abs(Round(xe)-Round(xs));
      Flag=1; //最大的插补长度和方向标记
      ix= Round(xs); //初始 X点
      idx=isign(dx); //X方向单位增量
      y= ys+dy/dx*((float)(ix)-xs); //初始 Y点修正
      dy=dy/fabs(dx); //Y方向斜率增量
}
else { // Y方向长， 斜率 >1
      Length=abs(Round(ye)-Round(ys));
      Flag=0;
      iy= Round(ys); //初始 Y点
      idy=isign(dy); //Y方向单位增量
      x= xs+dx/dy*((float)(iy)-ys); //初始 X点修正
      dx=dx/fabs(dy); //X方向斜率增量
}
```

```
if  (Flag)  { //X方向单位增量
      for  (n=0; n<= Length; n++) {//X方向插补过程
            WritePixel(ix, Round(y), value);
            ix+=idx;
            y+=dy;
      }  //End of for
} //End of if
else { //Y方向斜率增量
      for (n=0; n<= Length; n++) {//Y方向插补过程
            WritePixel (Round(x), iy, value);
            iy+=idy;
            x+=dx;
      }  //End of for
} //End of else
}  //Finish
```

5)本教程描述 ——首点校正对逼近的影响



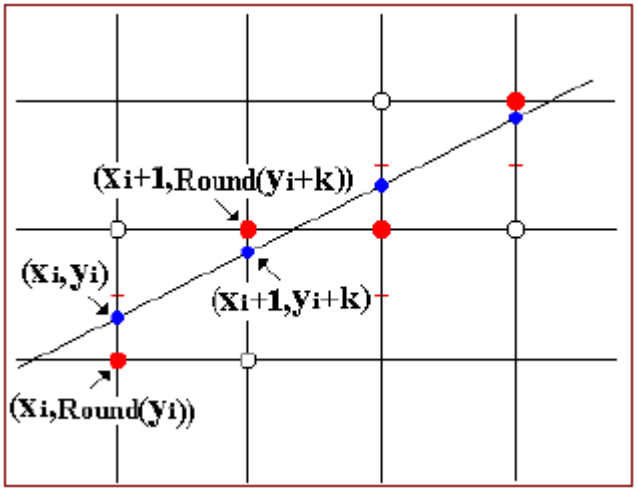
2.1.2 Bresenham算法

- Bresenham算法是计算机图形学典型的直线光栅化算法 。
- 从另一个角度看直线光栅化显示算法的原理 ：

由直线的斜率确定选择在 x方向或 y方向上每次递增 （ 减 ） 1个单位 ， 另一变量的递增 （ 减 ） 量为 0或 1， 它取决于实际直线与最近光栅网格点的距离 ， 这个距离的最大误差为 0.5。

1) Bresenham的基本原理

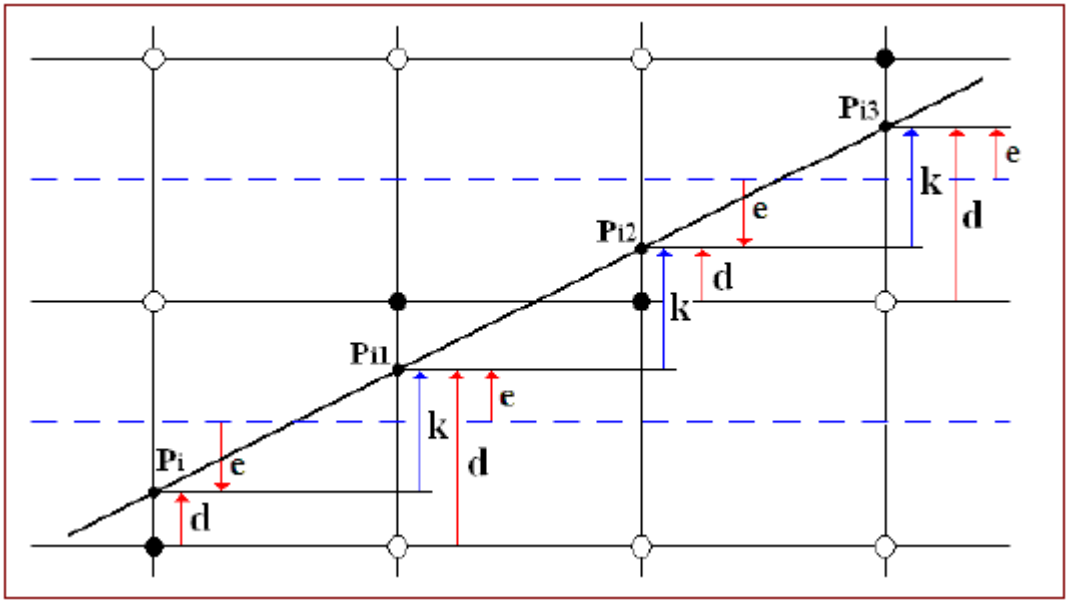
- 假定直线斜率 k在 0~1之间 。 此时， 只需考虑 x方向每次递增 1个单位， 决定 y方向每次递增 0或 1。
- 设直线的
  当前点为 (xi,y)
  当前光栅点为 (xi,yi)
- 下一个
  直线的点应为 (xi+1,y+k)
  直线的光栅点
  - 或为右光栅点 (xi+1,yi) （ y方向递增量 0）
  - 或为右上光栅点 (xi+1,yi+1)（ y方向递增量 1）



- 记 直线与它垂直方向最近的下光栅点的误差为 d，

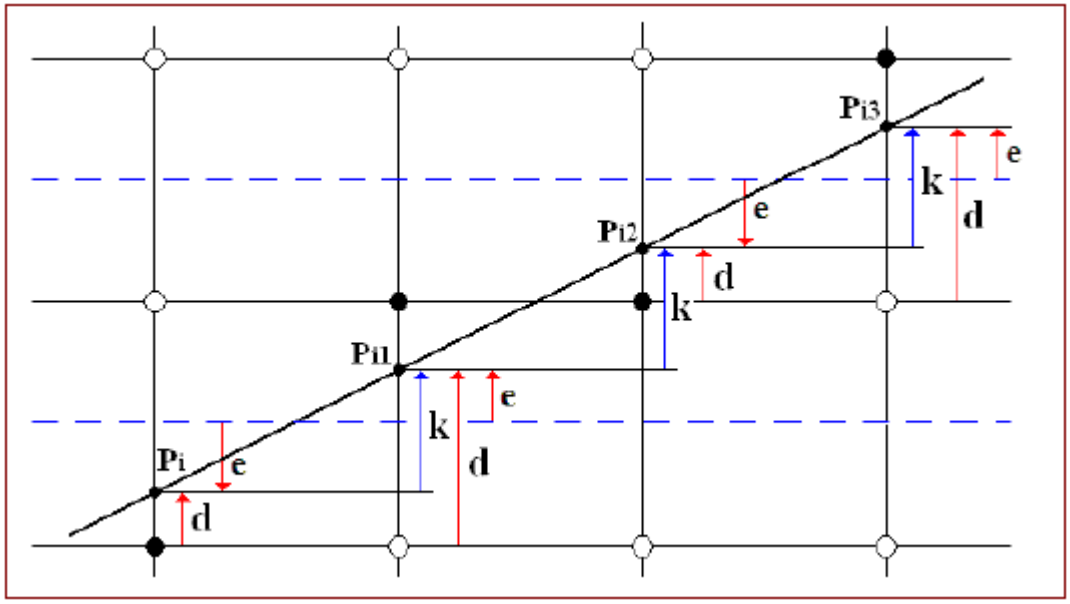
有： d=(y+k) –yi， 且

- 0≤ d≤ 1
- 当 d<0.5： 下一个象素应取右光栅点 (xi+1,yi)
- 当 d≥ 0.5： 下一个象素应取右上光栅点 (xi+1,yi+1)

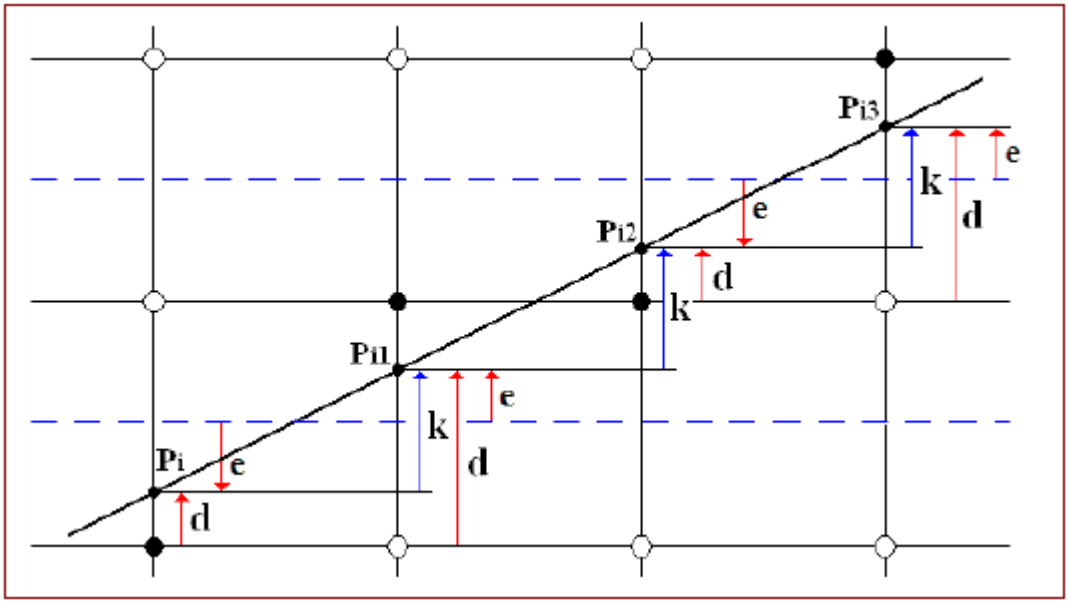


1) Bresenham的基本原理

- 如果直线的 （ 起 ） 端点在整数点上， 误差项 d的初值： d0＝ 0
- x坐标每增加 1， d的值相应递增直线的斜率值 k，即： d＝ d+k
- 一旦 d≥ 1， 就把它减去 1， 保证 d的相对性， 且在0-1之间 。



- 令 e=d-0.5， 关于 d的判别式和初值可简化成：
  - e的初值 e0= -0.5， 增量亦为 k;
  - e<0时， 取当前象素 (xi,yi)的右方象素 (xi+1,yi);
  - e>0 时， 取当前象素 (xi,yi) 的右上方象素 (xi+1,yi+1);
  - e=0时， 可任取上、 下光栅点显示 。

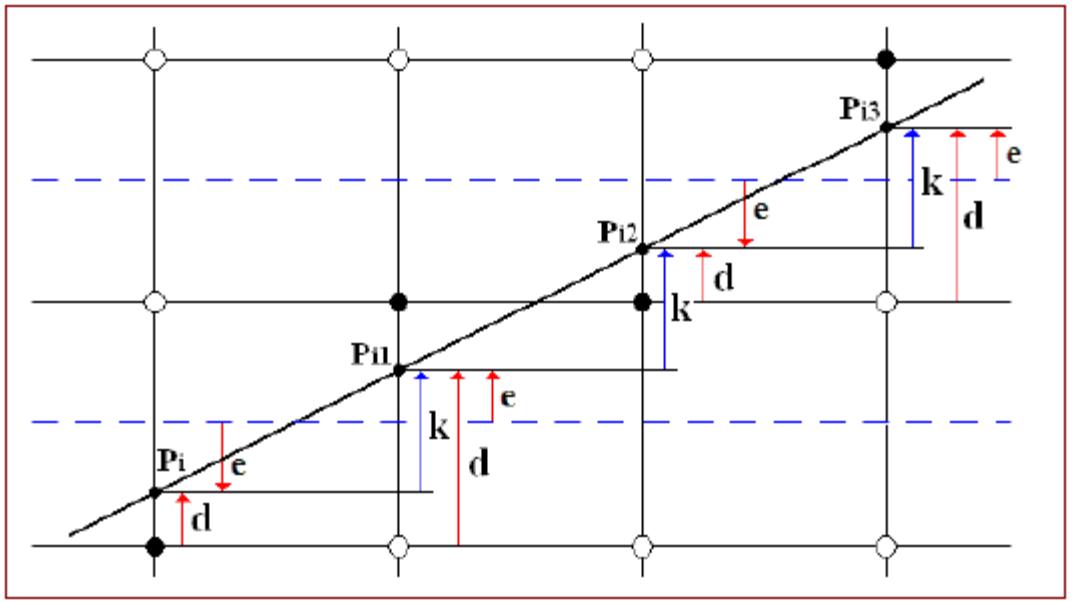


1) Bresenham的基本原理

- Bresenham算法的构思巧妙： 它引入动态误差 e， 当 x方向每次递增 1个单位， 可根据 e的符号决定 y方向每次递增 0或 1。
  - e<0， y方向不递增
  - e>0， y方向递增 1
  - x方向每次递增 1个单位， e = e + k

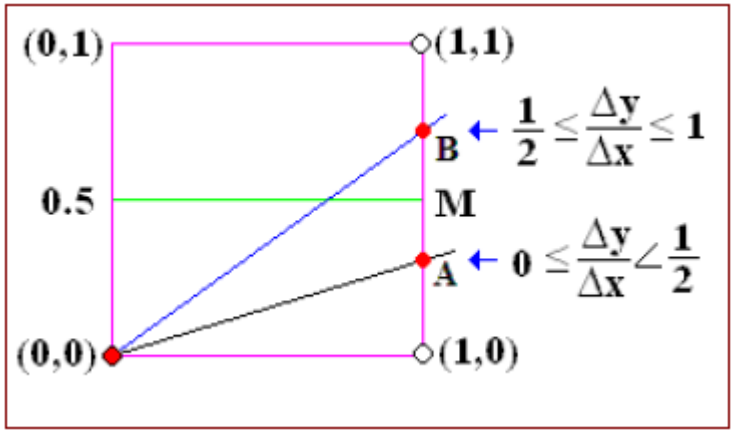
1) Bresenham的基本原理

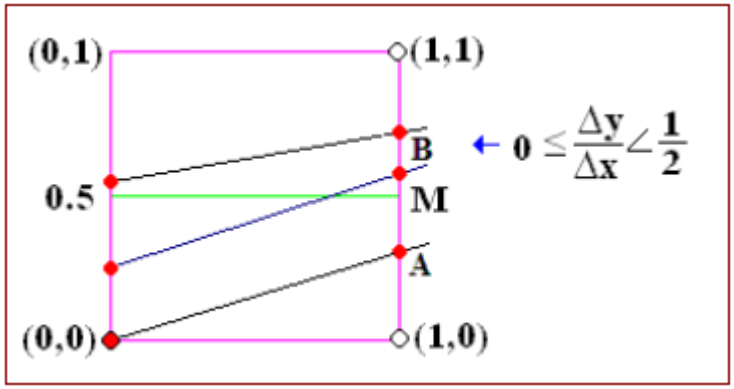
- 因为 e是相对量， 所以当 e>0时， 表明 e的计值将进入下一个参考点 （ 上升一个光栅点）， 此时须： e = e - 1



2) Bresenham算法的实施 ——Rogers 版

- 通过 (0,0)的所求直线的斜率大于 0.5， 它与 x=1直线的交点离 y=1 直线较近， 离 y=0直线较远， 因此取光栅点 (1,1)比 (1,0)更逼近直线；
- 如果 斜率 小于 0.5， 则反之；
- 当 斜率 等于 0.5， 没有确定的选择标准， 但本算法选择 (1,1)。





2) Bresenham算法的实施 ——Rogers 版

```
//Bresenham's line resterization algorithm for the first octal.
//The line end points are (xs,ys) and (xe,ye) assumed not equal.
// Round is the integer function.
// x,y, Δ x, Δ y are the integer, Error is the real.
//initialize variables
x=xs
y=ys
Δx = xe -xs
Δy = ye -ys
//initialize e to compensate for a nonzero intercept
Error =Δ y/Δx-0.5
```

2) Bresenham算法的实施 ——Rogers 版

```
//begin the main loop
for i=1 to Δx
        WritePixel (x, y, value)
        if  (Error ≥ 0) then
                y=y+1
                Error = Error -1        提问学生 why ?
        end if
        x=x+1
        Error = Error +Δ y/Δx
next i
finish
```

3)整数 Bresenham算法

- 上述 Bresenham算法在计算直线斜率和误差项时要用到浮点运算和除法， 采用整数算术运算和避免除法可以加快算法的速度 。
- 由于上述 Bresenham算法中只用到误差项（ 初值 Error =Δ y/Δx-0.5） 的符号
- 因此只需作如下的简单变换：

  NError = 2*Error*Δx
- 即可得到整数算法， 这使本算法便于硬件（ 固件 ） 实现

3)整数 Bresenham算法

//Bresenham's integer line resterization algorithm

for the first octal.

//The line end points are (xs,ys) and (xe,ye) assumed

not equal. All variables are assumed integer.

//initialize variables

x=xs

y=ys

Δx = xe -xs

Δy = ye -ys

//initialize e to compensate for a nonzero intercept

NError =2*Δ y-Δx **(Error =Δ y/Δx-0.5)**

3)整数 Bresenham算法

```
//begin the main loop
for i=1 to Δx
        WritePixel (x, y)
        if (NError >=0) then
                y=y+1
                NError = NError –2*Δx          ( Error = Error -1 )


        end if
        x=x+1
        NError = NError +2*Δ y          ( Error = Error +Δ y/Δx )
next i
finish
```

4)一般 Bresenham算法算法

- 要使第一个八卦的 Bresenham算法适用于一般直线， 只需对以下 2点作出改造：
  - 当直线的斜率 |k|>1时， 改成 y的增量总是 1，再用 Bresenham误差判别式确定 x变量是否需要增加 1;
  - x或 y的增量可能是 “+1”或 “-1”， 视直线所在的象限决定 。

```
//Bresenham's integer line resterization algorithm for all quadrants
//The line end points are (xs,ys) and (xe,ye) assumed not equal. All variables
are assumed integer.
//initialize variables
x=xs
y=ys
Δx = abs(xe -xs)
Δy = abs(ye -ys)
sx = isign(xe -xs)
sy = isign(ye -ys)
//Swap Δ x and Δ y depending on the slope of the line.
if Δy>Δx then
        Swap(Δx,Δy)
        Flag=1
else
      Flag=0
end if
```

Δx = xe -xs

Δy = ye -ys

```
//initialize the error term to compensate for a nonezero
intercept
NError =2*Δy-Δx
//begin the main loop
for i=1 to Δx
    WritePixel(x, y , value)
    if (NError>=0) then
        if (Flag) then    //Δy>Δx
            x=x+sx
        else                          Y=Y+1
            y=y+sy
        end if // End of  Flag
        NError = NError –2*Δx
    end if // End of NError
```

4)一般 Bresenham算法算法

```
    if (Flag) then //Δy>Δ x
        y=y+sy
    else
        x=x+sx
    end if
        NError = NError +2*Δy
next i
finish
```

X=X+1

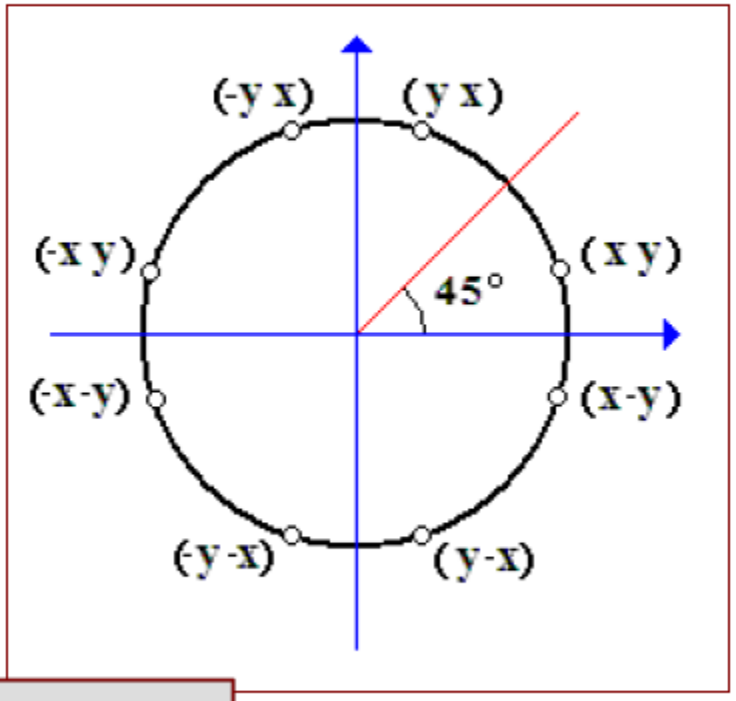
2.2 圆光栅化算法

2.2.1利用圆的八方对称性画圆

- 对圆的分析均假定圆心在坐标原点， 因为即使圆心不在原点， 可以通过一个简单的平移即可， 而对原理的叙述却方便了许多
- 即考虑圆的方程为： $x^2+y^2=R^2$

2.2.1利用圆的八方对称性画圆利用圆的八方对称性画圆

```
void CirclePoints (int x,int y, int value)
{
    WritePixel (x, y, value);
    WritePixel (-x, y, value);
    WritePixel (-x, -y, value);
    WritePixel (x, -y, value);
    WritePixel (y, x, value);
    WritePixel (-y,x, value);
    WritePixel (-y, -x, value);
    WritePixel (y, -x, value);
}
```



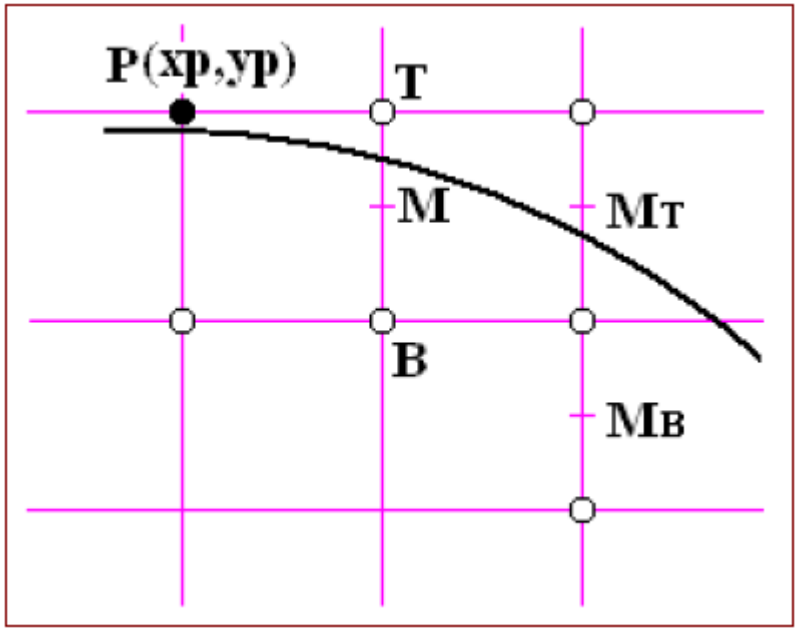
显然，当x=0或x=y或y=0时，圆上的对称点只有4个，因此，CirclePoints()需要修正。

2.2.2 中点圆算法——原理

- 设 d是点 p(x,y)到圆心的距离， 有：

$$d= F(x,y)=x^2+y^2-R^2$$

- 按照 Bresenham算法符号变量的思想， 以圆的下 2个可选象素 中点的函数值 d的符号 决定选择 2个可选象素 T和 B中哪一个更接近圆而作为圆的显示点 ？



2.2.2 中点圆算法——原理

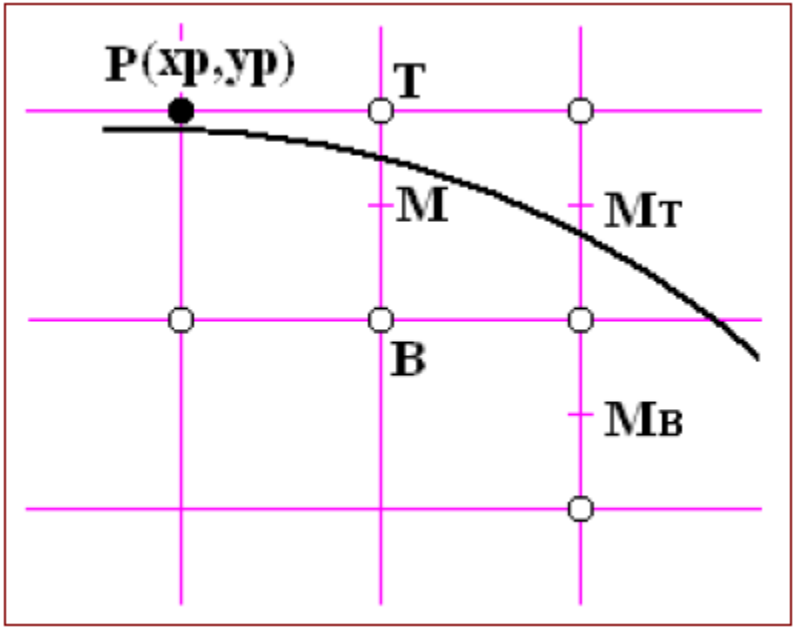
- $d_M= F(x_M,y_M)= F(x_p+1,y_p-0.5)=(x_p+1)^2+(y_p-0.5)^2-R^2$

  如果 $d_M<0$， 表示下一中点 M在圆内， 用 T点逼近， 得

- $d_{MT}= F(x_{MT}, y_{MT})= F(x_p+2,y_p-0.5)=(x_p+2)^2+(y_p-0.5)^2-R^2$

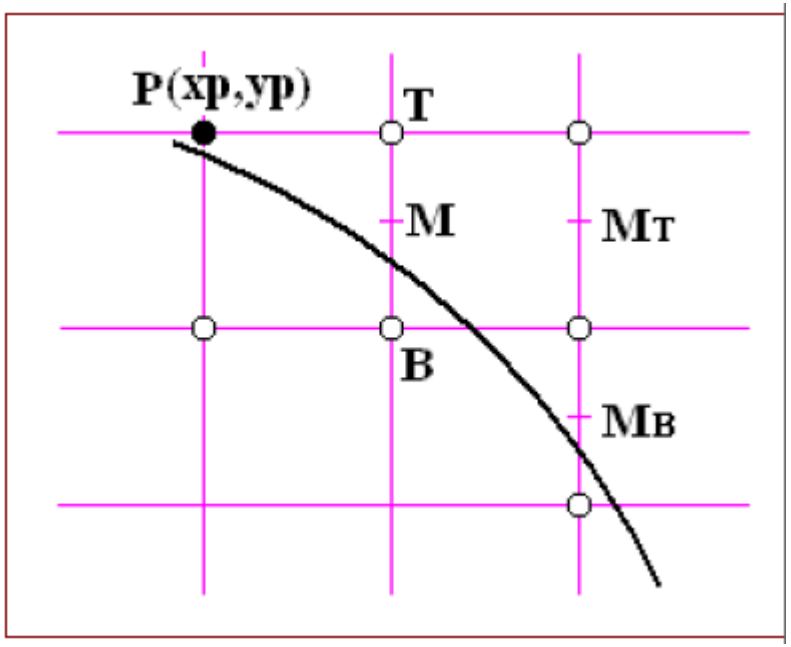
- $\Delta d_{MT}= d_{MT} - d_M= 2x_p+3$

注意：$x_p^2+y_p^2-R^2$ 并不等于零



如果 $d_M>0$， 表示下一中点 M在圆外， 用 B点逼近， 得

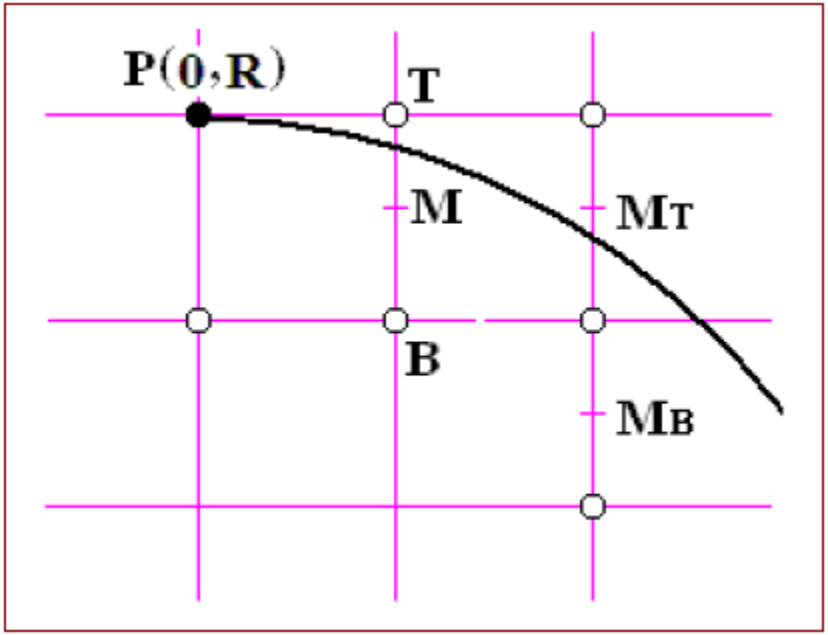
- $d_{MB}= F(x_{MB}, y_{MB})= F(x_p+2,y_p-1.5)=(x_p+2)^2+(y_p-1.5)^2-R^2$
- $\Delta d_{MB}= d_{MB} - d_M=2x_p-2y_p + 5$
- 结论：
- 根据中点 d的值， 决定
  - 显示的光栅点 （ T或 B）
  - 新的 Δd（ Δd_{MT}或 Δd_{MB} ）
  - 更新 d



2.2.2 中点圆算法——原理

初值

- 由 $x_0=0$， $y_0 = R$
- 得 $x_{M0}=0+1$， $y_{M0} = R-0.5$
- $d_{M0}= F(x_{M0}, y_{M0})= F(1,R-0.5)= 1^2+(R-0.5)^2-R^2=1.25-R$



2.2.3 中点圆算法 ——实施

```
//中点圆算法 （ 假设圆的中心在原点 ）
void MidPointCircle(int radius,int value)
{
    int x=0;
    int y= radius;
    double d=1.25- radius;
```

2.2.3 中点圆算法——实施

```
While (y>x)  {
    CirclePoints(x, y,value);
    if (d<0) //选择 T
        d+=2.0*x+3.0;
    else { //选择 B
        d+=2.0*(x-y)+5.0;
        y--;
    }
    x++;
    } //End of while
}
```

2.2.4 中点圆整数算法——原理

- 中点圆算法 的半径是整数， 而 用于该算法

  符号判别的变量 d（ 初值 d=1.25- radius）

  采用浮点运算， 会花费较多的时间 。
- 为了将其改造成整数计算， 定义新变量：

  D =d - 0.25
- 那么判别式 d<0等价于 D<-0.25。
- 在 D为整数情况下， D<-0.25和 D<0等价
- 仍将 D写成 d（ 新的初值 d=1-radius）， 可得到 中点圆整数算法 。

2.2.4中点圆整数算法——实施

```
//中点圆算法 （ 假设圆的中心在原点 ）
void MidPointCircle(int radius,int value)
{
    int x=0;
    int y= radius;
    int d=1- radius;
    //CirclePoints(x, y, value);
```

d=1.25-radius

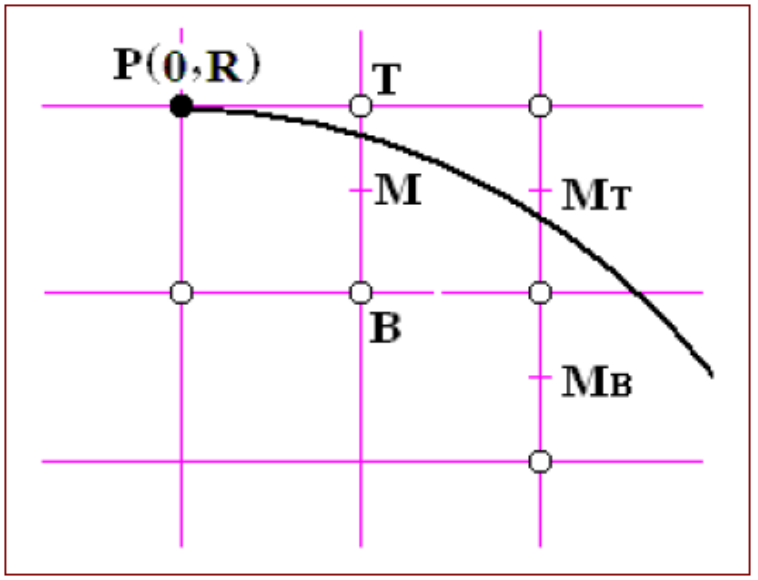
2.2.4中点圆整数算法——实施

```
While (y>x)  {
    CirclePoints(x, y,value);
    if (d<0) //选择 T
        d+=2*x+3;
    else { //选择 B
        d+=2*(x-y)+5;
        y--;
    } //End of else
    x++;
    //CirclePoints(x, y,value);
} //End of while
}
```

d+=2.0*x+3.0

d+=2.0*(x-y)+5.0

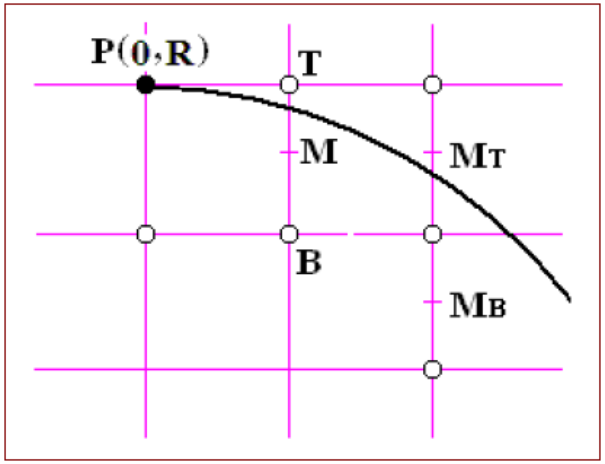
2.2.5中点圆整数优化算法 — 原理



- 用 Δd修正 d
- 1) 选择 T点 （ $x_p \leftarrow x_p+1$）：
  - d的增量 （ 一次差分 ）：
    - $\Delta d_T = 2x_p+3$
  - Δd的增量 （ 二次差分 ）：
    - $\Delta^2 d_T = 2(x_p+1)+3-(2x_p+3)=2$
    - $\Delta^2 d_B = 2(x_p+1)-2y_p+5-(2x_p-2y_p+5) = 2$

2.2.5中点圆整数优化算法 — 原理

- 2) 选择 B点 （ $x_p \leftarrow x_p+1$， $y_p \leftarrow y_p-1$）：
  - d的增量 （ 一次差分 ）：
    - $\Delta d_B = 2x_p-2y_p+5$
  - Δd的增量 （ 二次差分 ）：
    - $\Delta^2 d_T = 2(x_p+1)+3-(2x_p+3)=2$
    - $\Delta^2 d_B = 2(x_p+1)-2(y_p-1)+5-(2x_p-2y_p+5)=4$



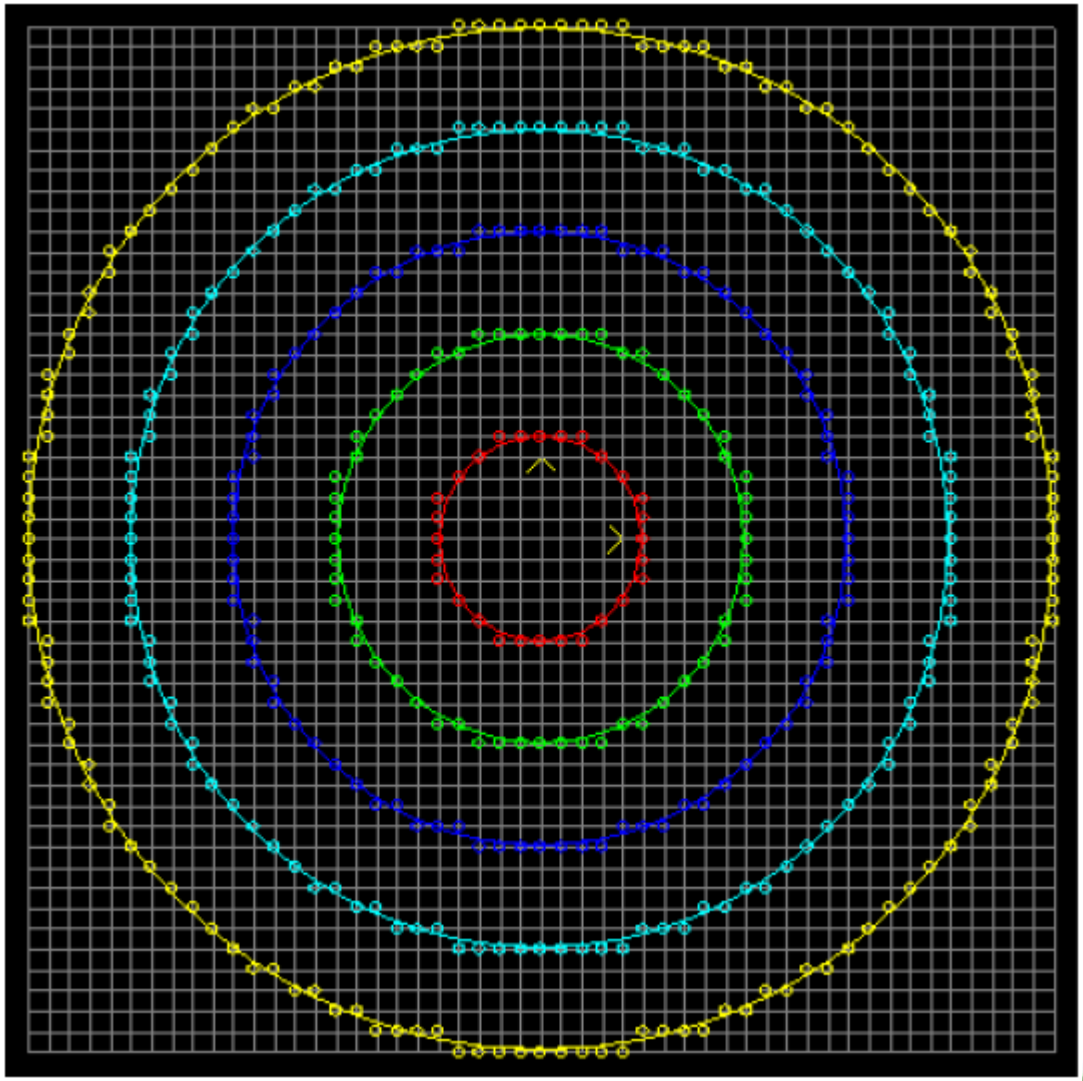
2.2.5中点圆整数优化算法 — 实施

```
//中点圆整数优化算法 （ 假设圆的中心在原点 ）
void MidPointCircleInt(int radius,int value)
{
    int x=0;
    int y= radius;
    int d=1- radius;
    int dt=3;
    int db= -2*radius+5;
```

```
        While (y>=x)  {
            CirclePoints(x, y,value);
            if (d<0) //选择 T
                d=d+dt;
                dt+=2;
                db+=2;

            else { //选择 B
                d=d+db;
                dt+=2;
                db+=4;
                y--;
            }
            x++;
        } //End of while
} //Finish
```

2.2.5中点圆整数中点圆整数优化算法 — 例子



总结

- 直线光栅化算法
  - DDA算法
  - Bresenham算法
- 圆光栅化算法
  - 中点算法
  - 中点整数算法
  - 中点整数优化算法
- 基本方法
  - 增量算法
  - 符号算法

2.3 椭圆光栅化算法

2.3.1 椭圆的扫描转换

中点画圆法可以推广到一般二次曲线的生成，下面以中心在原点的标准椭圆的扫描转换为例说明。 设椭圆的方程为

$$F(x,y)=b^2x^2+a^2y^2-a^2b^2=0$$

其中，a为沿x轴方向的长半轴长度，b为y轴方向的短半轴长度，a、 b均为整数。 不失一般性，我们只讨论第一象限椭圆弧的生成。需要注意的是，在处理这段椭圆时，必须以弧上斜率为-1的点(即法向量两个分量相等的点)作为分界把它分为上部分和下部分，如图2.6所示。

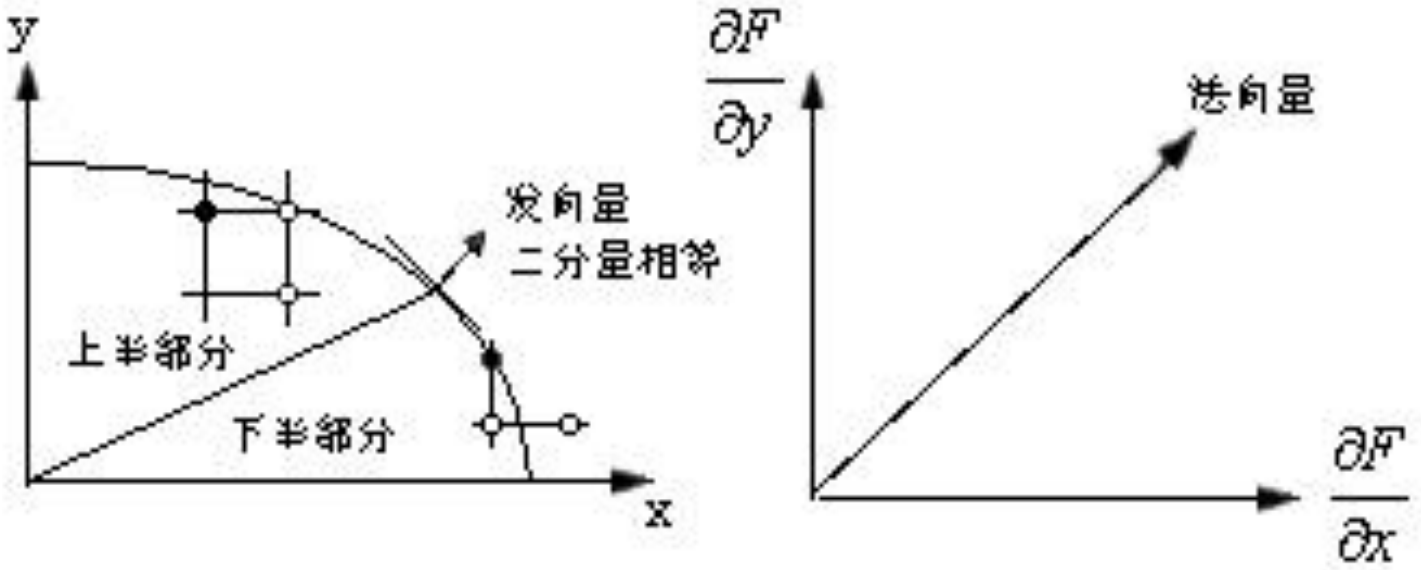


图 2.6 第一象限的椭圆弧

该椭圆上一点(x, y)处的法向量为

$$N(x,y)=\frac{\partial F}{\partial x}i+\frac{\partial F}{\partial y}j=2b^2xi+2a^2yj$$

其中，i和j分别为沿x轴和y轴方向的单位向量。 从图2.6可看出，在上部分，法向量的y分量更大，而在下部分，法向量的x分量更大，因而，在上部分若当前最佳逼近理想椭圆弧的像素(x_P,y_P)满足下列不等式

$$b^2(x_P+1)<a^2(y_P-0.5)$$

而确定的下一个像素不满足上述不等式，则表明椭圆弧从上部分转入下部分。

在上部分，假设横坐标为x_P的像素中与椭圆弧更接近点是(x_P, y_P)，那么下一对候选像素的中点是$(x_P+1, y_P-0.5)$。因此判别式为

$$d_1=F(x_P+1, y_P-0.5)=b^2(x_P+1)^2+a^2(y_P-0.5)^2-a^2b^2$$

若$d_1<0$，中点在椭圆内，则应取正右方像素，且判别式应更新为

$$d'_1 = F(x_P+2, y_P-0.5)=b^2(x_P+2)^2+a^2(y_P-0.5)^2-a^2b^2$$

$$=d_1+b^2(2x_P+3)$$

当$d_1 \geq 0$, 中点在椭圆之外, 这时应取右下方像素, 并且更新判别式为

$$d'_1=F(x_P+2, y_P-1.5)=b^2(x_P+2)^2+a^2(y_P-1.5)^2-a^2b^2$$

$$=d_1+b^2(2x_P+3)+a^2(-2y_P+2)$$

由于弧起点为$(0, b)$, 因此, 第一中点是$(1, b-0.5)$, 对应的判别式是

$$d_{10}=F(1, b-0.5)=b^2+a^2(b-0.5)^2-a^2b^2$$

$$=b^2+a^2(-b+0.25)$$

在下部分,应改为从正下方和右下方两个像素中选择下一像素。 如果在上部分所选择的最后一像素是(x_P, y_P), 则下部分的中点判别式d_2的初始值为

$$d_{20}=F(x_P+0.5, y_P-1)=b^2(x_P+0.5)^2+a^2(y_P-1)^2-a^2b^2$$

d_2在正下方向与右下方向的增量计算与上部分类似,这里不再赘述。 下部分弧的终止条件是$y=0$。

第一象限椭圆弧的扫描转换中点算法的伪C描述如下:

```
 void MidpointEllipse(a, b, color)
int a, b, color;
{ int x, y;
  float d1, d2;
  x=0; y=b;
  d1=b*b+a*a*(-b+0.25);
  putpixel(x, y, color);
  while(b*b*(x+1)<a*a(y-0.5))
 { if(d1<0)
  { d1+=b*b*(2*x+3);
    x++;
  }
```

```
else { d1+=(b*b*(2*x+3)+a*a*(-2*y+2));
        x++; y--;
      }
   putpixel(x, y, color);
  }/*上半部分*/
  d2=sqr(b*(x+0.5))+sqr(a*(y-1))-sqr(a*b);
  while(y>0)
   { if(d2<0)
     {  d2+=b*b(2*x+2)+a*a*(-2*y+3);
       x++;
       y--;
     }
     else { d2+=a*a*(-2*y+3);
```

sqr() 为平方函数

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