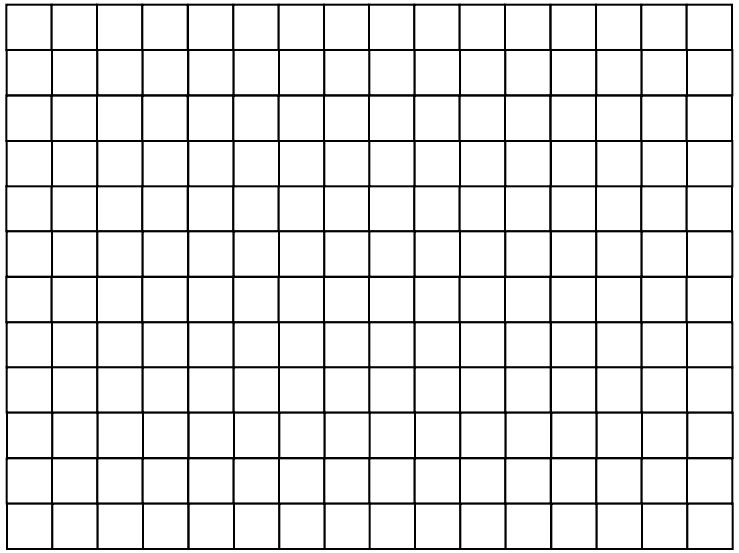
第二章 二维图元的生成

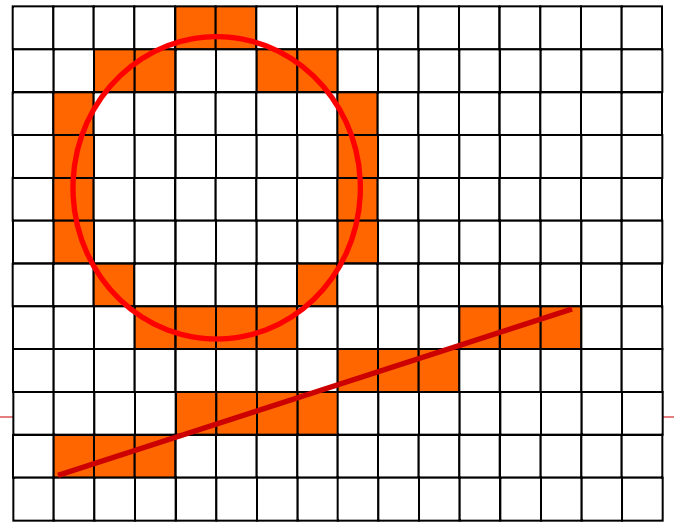
引言

- 光栅图形显示器可以看作一个象素的矩阵。在光栅显示器上显示任何一种图形，实际上都是一些具有一种或多种颜色的象素集合。



引言

- 确定最佳逼近图形的象素集合，并用指定属性写象素的过程，即指完成从图元的参数表示形式转换成点阵表示形式的过程称为图形的扫描转换或光栅化。
- 所谓图元的生成，是指完成图元的参数表示形式（由用户指定）到点阵表示形式（显示系统刷新时所需的表示形式）的转换。通常也称扫描转换图元。



本章内容

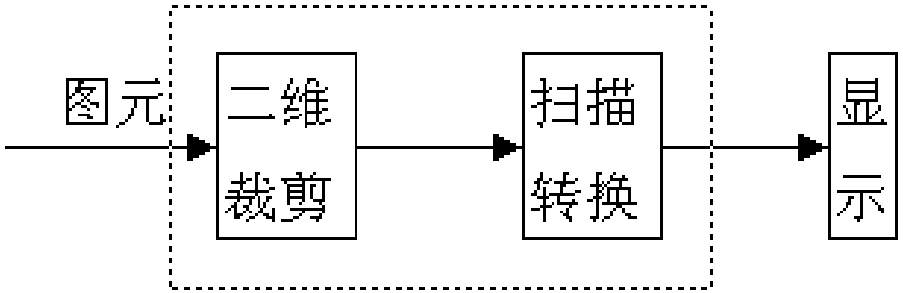
- 简单的二维图形显示流程
- 直线段的扫描转换
- 圆弧的扫描转换
- 易画曲线的正负法
- 线画图元的属性控制

掌握要点:

- 掌握扫描转换直线段的DDA算法、中点算法，以及中点算法在哪些方面对DDA算法做了改进;
- 掌握扫描转换直线段的Bresenham算法;
- 掌握圆弧的八对称性，掌握扫描转换圆弧的中点算法;
- 掌握生成圆弧的多边形迫近法;
- 了解正负法，掌握怎样利用正负法生成圆弧;
- 掌握扫描转换椭圆弧的中点算法;
- 了解线画图元的属性（线型、线宽）控制方法。

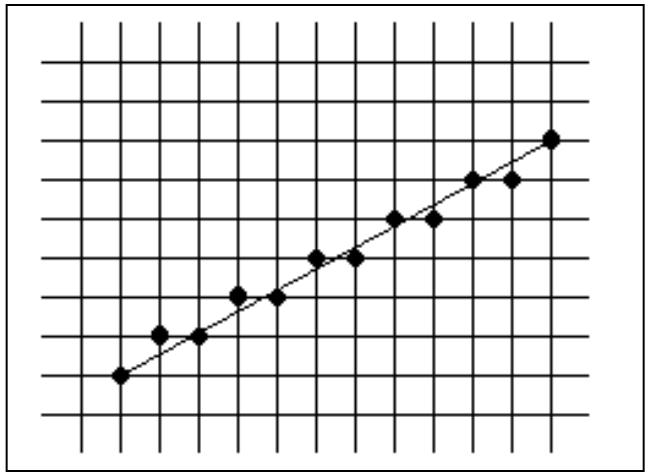
2.1 简单的二维图形显示流程

- 扫描转换：顶点(参数)表示的图形(来自用户) -------> 点阵表示的图形(图像，显示在显示器上)



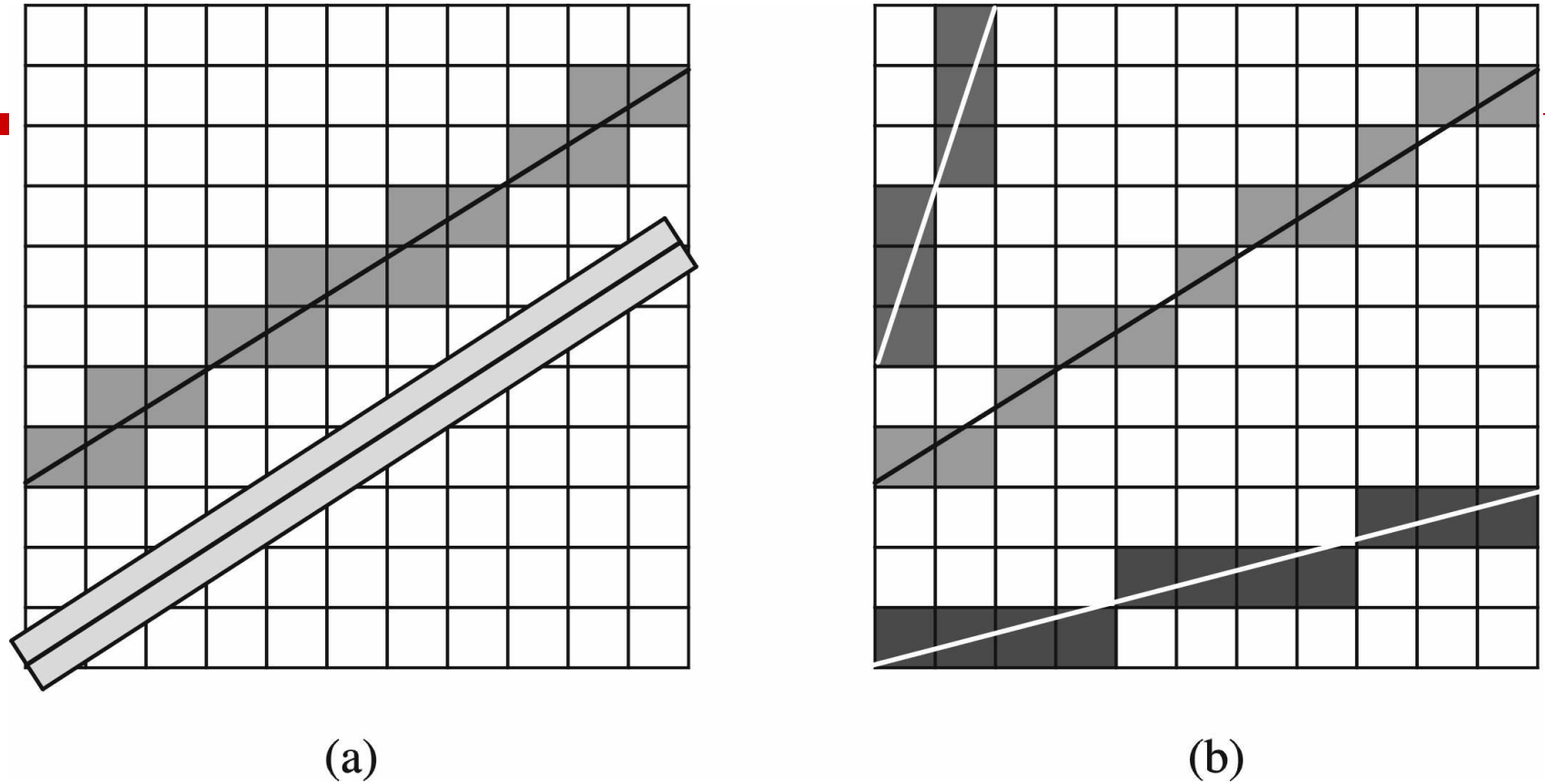
- 裁剪的顺序：
  - 先裁剪再扫描转换----常用，计算量小
  - 先扫描转换再裁剪----算法简单，对有快速测试方法或硬件支持情形有利

2.2 直线段的扫描转换



- 求与直线段充分接近的像素集
  - 像素组成均匀网格
  - 整型坐标系
- 两点假设
  - 直线段的宽度为1 (大于1的由专门的线宽控制算法处理)
  - 直线段的斜率:　　(绝对值大于1的，修改算法完成)

$$k \in [-1,1]$$



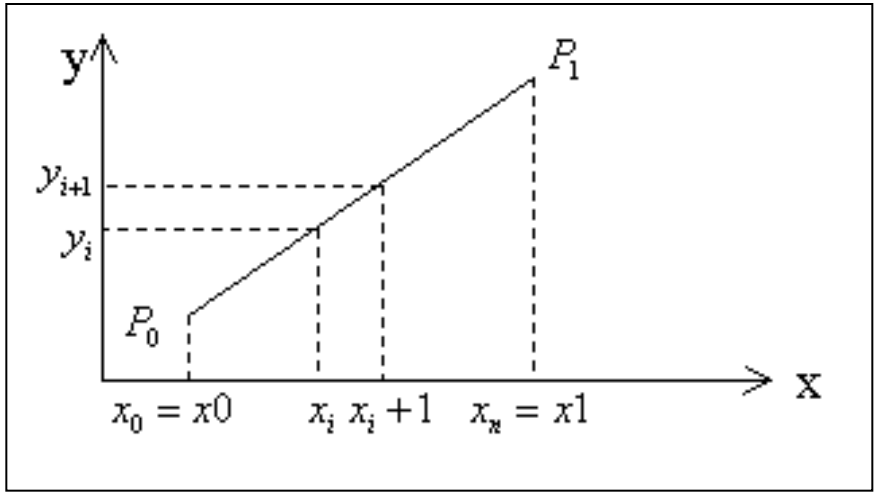
直线段的像素点表示
(a) 理论直线段及所有涉及到的像素点; (b) 应当选择的像素点

DDA (Digital Differential Analyzer) 算法

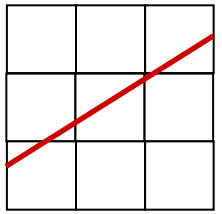
- 条件:
  - 待扫描转换的直线段: $P_0(x0, y0)P_1(x1, y1)$
  - 斜率: $k = \Delta y / \Delta x$

    $\Delta x = x1 - x0, \Delta y = y1 - y0$
  - 直线方程:

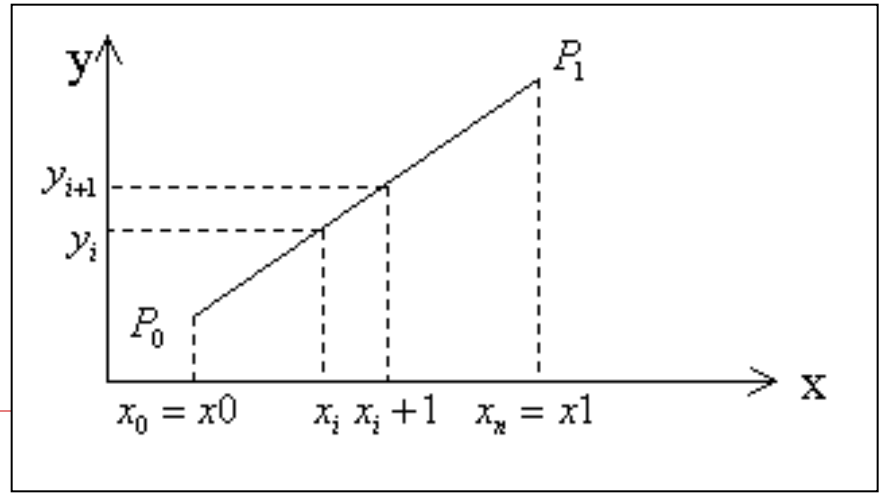
    $y = k \bullet x + B$



- 直接求交算法：
  - 划分区间[x0,x1]：$x_0, x_1, \cdots, x_n$，其中$x_{i+1} = x_i + 1$
  - 计算纵坐标：$y_i = k \bullet x_i + B$
  - 取整：$y_{i,r} = round(y_i) = (\text{int})(y_i + 0.5)$



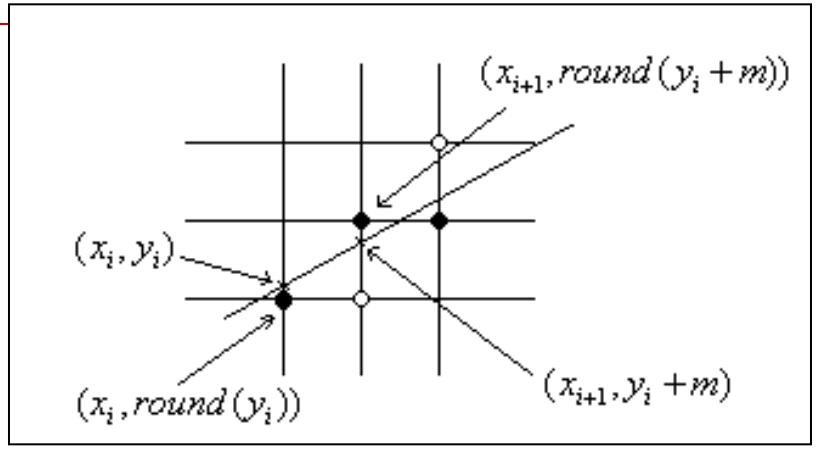
- 复杂度：$\{(x_i, y_i)\}_{i=0}^{n} \Rightarrow \{(x_i, y_{i,r})\}_{i=0}^{n}$
  - 乘法+加法+取整



DDA的优化

- 由于：

$$y_{i+1} = k \bullet x_{i+1} + B = k \bullet (x_i + 1) + B$$
$$= k \bullet x_i + B + k = y_i + k$$



- 即x增加1，y增加一个斜率值，因此我们可以直接从y_i计算y_{i+1}
- 计算中的乘法被省略，

  计算复杂度减少，

  该算法称为DDA算法

DDA画线算法程序

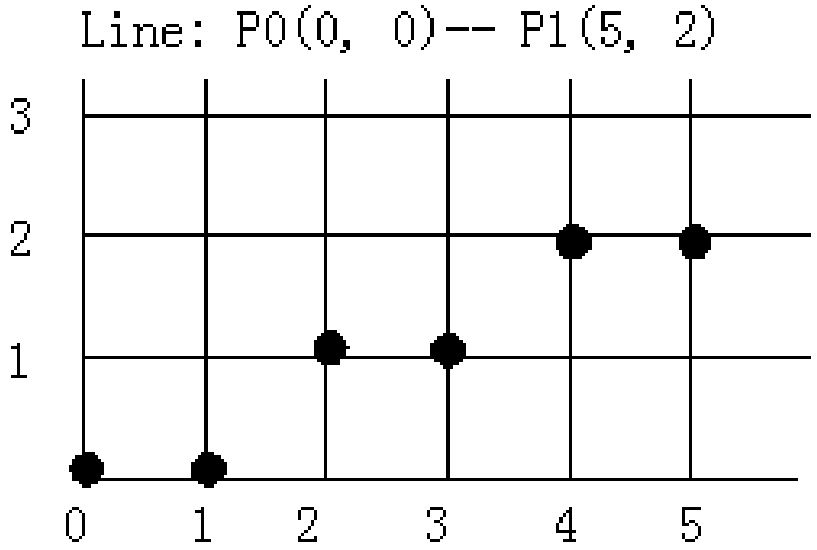
```
void LineDDA(int x0,int y0,int x1,int y1,int color)
 /* 假定x0<x1,-1<=k<=1 */
{ int x;
  float dx, dy, y, k;
  dx= x1-x0;
  dy=y1-y0;
  k=dy/dx;                          // 求斜率
  y=y0;
  for(x=x0; x<= x1, x++)            // x从x1到xn
  { Putpixel(x, int(y+0.5), color);
      y+=k;                         //y每次增加一个斜率值
   }
 }
```

举例

- 用DDA方法扫描转换连接两点P0（0,0）和P1（5,2）的直线段

k=0.4

X	int(y+0.5)	y
0	0	0
1	0	0.4
2	1	0.8
3	1	1.2
4	2	1.6



DDA法总结

- 注意上述分析的算法仅适用于| k| ≤1的情形。在这种情况下，x每增加1，y最多增加1。当 | k| > 1时，必须把x，y地位互换，y每增加1，x相应增加1/k。

- 在这个算法中，y与k必须用浮点数表示，而且每一步都要对y进行四舍五入后取整。这使得它不利于硬件实现。

中点算法

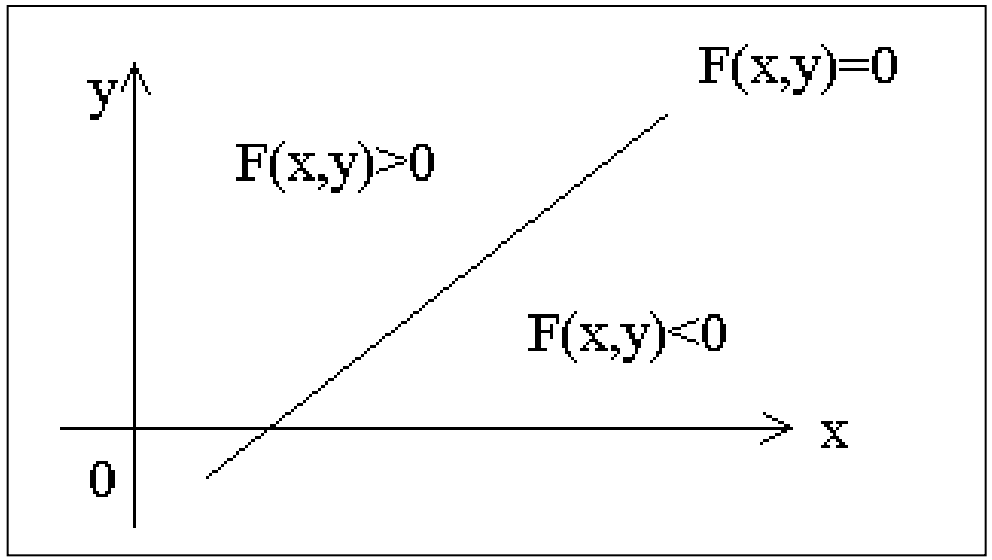
- 目标：消除DDA算法中的浮点运算（浮点数取整运算不利于硬件实现，导致DDA算法效率低）
  - 条件：
    - 同DDA算法
    - 斜率：$k \in [0,1]$
  - 以(x0,y0)(x1,y1)为端点的直线段的隐式方程：

    F(x,y)=ax+by+c=0

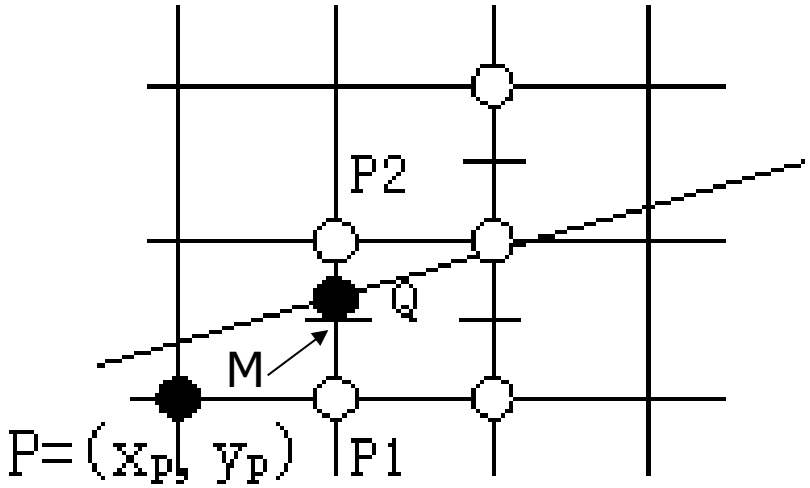
    式中 a=y0-y1, b=x1-x0, c=x0y1-x1y0

直线的正负划分性

- 点(x,y)与直线的关系　F(x,y)=ax+by+c=0
  - on: F(x,y)=0;
  - up: F(x, y)>0;
  - down: F(x, y)<0;



- 画直线段的过程中，当前象素点为(xp, yp)，下一个象素点有两种可选择点：p1(xp+1, yp)或p2(xp+1, yp+1)。
- 若M=(xp+1, yp+0.5)为p1与p2之中点，Q为理想直线与x=xp+1垂线的交点。当M在Q的下方，则P2 应为下一个象素点；M在Q的上方，应取P1 为下一点。就是中点画线法的基本原理。



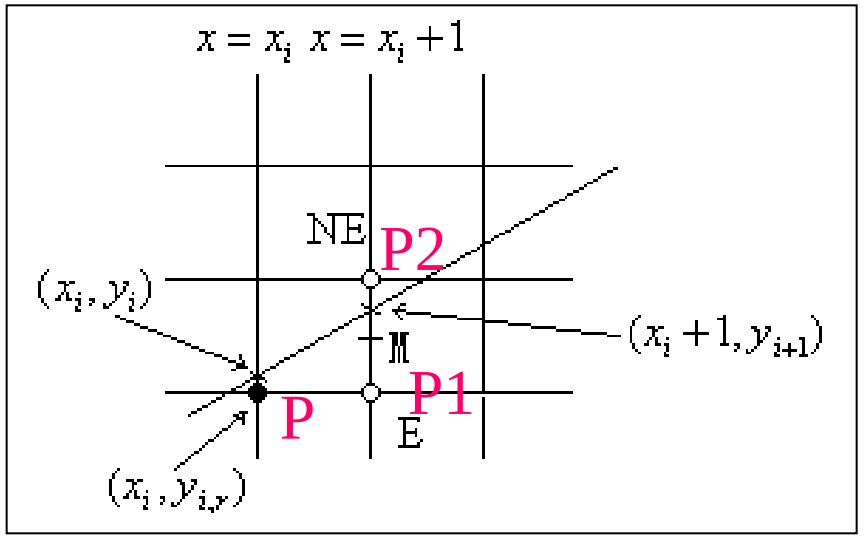
- 根据直线的正负性划分，欲判断中M在Q点的上方还是下方，只要把M代F(x，y)，并判断它的符号。
- 因此构造判别式:

  d=F(M)=F(xp+1, yp+0.5) =a(xp+1)+b(yp+0.5)+c

  - 当d<0，M在Q点下方，取P2为下一个象素;
  - 当d>0，M在Q点上方，取P1为下一个象素;
  - 当d=0，M在理想直线上， M,Q重合，选P1或P2均可，约定取P1为下一个象素

- 问题：判断距直线最近的下一个象素点
  - 构造判别式：d=F(M)=F(Xp+1,Yp+0.5)

由d的符号可判定下一个象素，



- 问题：计算判别式复杂，如何简单化?
  - 组织成增量形式。
- 要判定再下一个象素，分两种情形考虑：

1) 若d≥0，取正右方象素P1，再下一个象素判定的公式：

d1=F(Xp+2,Yp+0.5)=a(Xp+2)+b(Yp+0.5)+c=d+a，d的增量是a

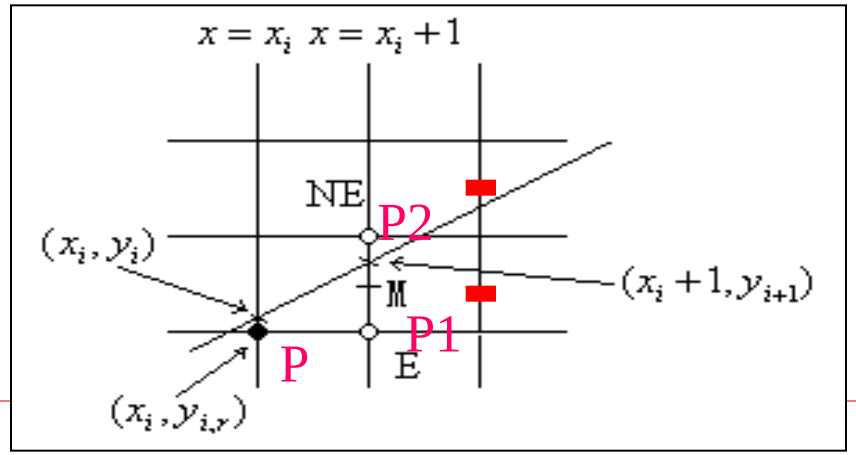
2) 若d＜0，取右上方象素P2，

再下一个象素的判定公式：

d2=F(Xp+2,Yp+1.5)

=d+a+b

d的增量为a+b



- d的增长方式已定，需要d的初始值
  - d0=F(X0+1,Y0+0.5)=F(X0,Y0)+a+0.5*b
  - 因(X0,Y0)在直线上，F(X0,Y0)=0,所以，d0=a+0.5*b
- d的增量a,b都是整数，只有初始值包含小数，可以用2d代替d，让d0也成为整数。 2a改写成a+a。
  - 这样d0=a+a+b，每次增量为a+a或a+a+b+b。
- 经过上述过程，算法中只有整数变量，不含乘除法，方便用硬件实现。

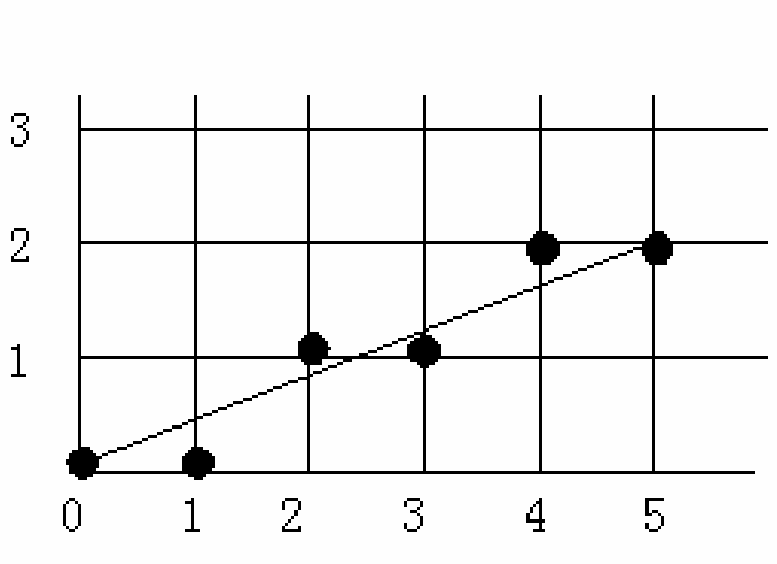
程序

```
Midpointline(x0,y0,x1,y1,color)
int x0,y0,x1,y1,color;
{
   int a,b,d1,d2,2,x,y;
       a = y0-y1; b = x1 – x0;  d = a + a + b;  //计算a,b,d0
       d1 = a + a;  d2 = a + a + b + b;          //计算可能的增量
       x = x0; y = y0;
       PutPixel(x,y,color);
       while (x<x1)
       {  if  (d<0)
                    { x++; y++; d +=d2;}  //如果d为负，取右上角点(y加1)
           else  { x++; d +=d1;}          //如果d为正，取右边点
           PutPixel(x,y,color);
       }
}
```

举例

- 用中点画线方法扫描转换连接两点P0（0,0）和P1（5,2）的直线段
  - a=y0-y1=-2; b=x1-x0=5;
  - d0=2*a+b=1; d1=2*a=-4; d2=2*(a+b)=6

x	y	d	下点位置
0	0	1	d>0,右点，增量d1
1	0	-3	d<0,右上，增量d2
2	1	3	d>0,右点，增量d1
3	1	-1	d<0,右上，增量d2
4	2	5	
5	2	1	



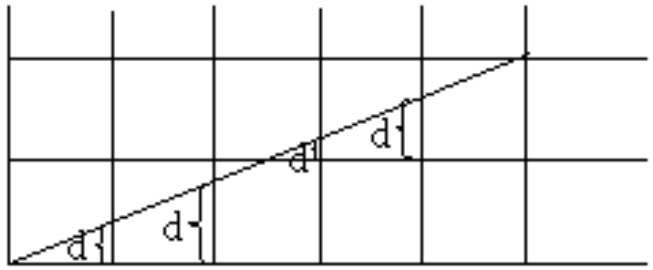
Bresenham算法

- Bresenham算法是计算机图形学领域使用最广泛的直线扫描转换算法。
- 该方法类似于中点法，由误差项符号决定下一个象素取右边点还是右上点。
- 算法原理如下：过各行各列象素中心构造一组虚拟网格线。按直线从起点到终点的顺序计算直线与各垂直网格线的交点，然后确定该列象素中与此交点最近的象素。
- 该算法的巧妙之处在于采用增量计算，使得对于每一列，只要检查一个误差项的符号，就可以确定该列的所求象素。

- 设直线方程为:

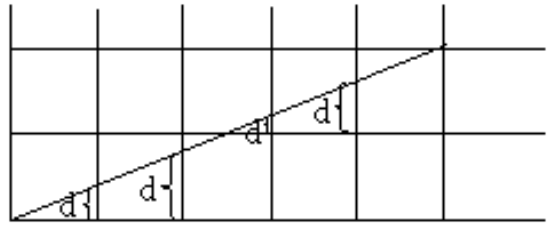
$$y_{i+1} = y_i + k \bullet (x_{i+1} - x_i) = y_i + k$$ 其中k=dy/dx。

- 假设已知象素(xi, yi)。那么下一个象素的x坐标为xi＋1，而y坐标要么不变(为yi)，要么递增1变为yi＋1)。是否增1取决于如图所示误差项d的值。
- 因为直线的起始点在象素中心，所以误差项d的初值d0＝0。



- X下标每增加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，这样保证d在0、1之间。
- 当d≥0.5时，直线与xi＋1列垂直网格交点最接近于当前象素（xi，yi）的右上方象素（xi＋1，yi＋1）；

  当d<0.5时，更接近于右方象素（xi＋1，yi）。



- 为方便计算，令e＝d－0.5，e的初值为－0.5，增量为k。

  当e≥0时，取当前象素（xi，yi）的右上方象素（xi＋1，yi＋1）；

  当e<0时，更接近于右方象素（xi＋1，yi）。

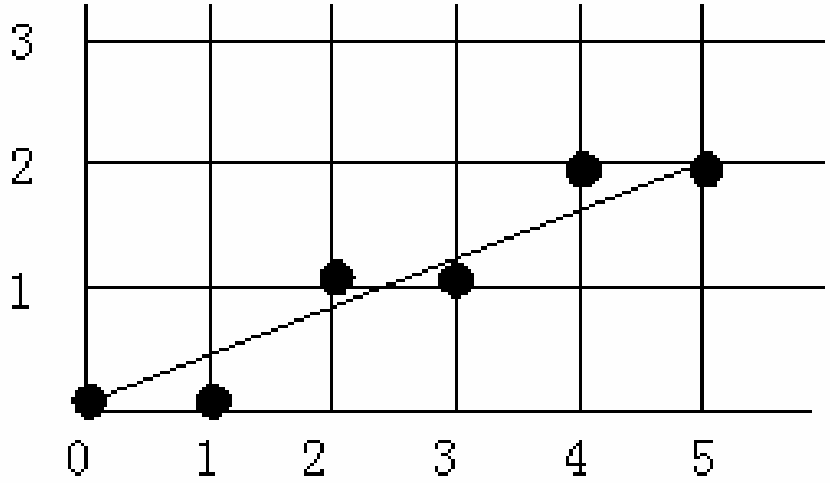
因此有如下程序：

```
void LineByE (int x0,int y0,int x1, int y1,int color)
{
    int x, y, dx, dy;
    float k, e;
    dx = x1-x0;    dy = y1- y0;   k=dy/dx;
    e=-0.5;                                    //e的初值为-0.5
    x=x0; y=y0;
    for (i=0; i<=dx; i++)    //  i从x0到x1也可，但速度是这个快
    {  Putpixel (x, y, color);
        x=x+1;
        e=e+k;                              //e每次的增量为k
        if (e>= 0)  { y++, e=e-1;} //若e>=0则：y增1，e-1
  }
}
```

举例

- 用上述方法扫描转换连接两点P0（0,0）和P1（5,2）的直线段。
  - k=0.4

x	y	e	下点位置
0	0	-0.5	
1	0	-0.1	
2	1	-0.7	0.3-1 y增
3	1	-0.3	
4	2	-0.9	0.1-1 y增
5	2	-0.5	



Bresenham算法

- 上述算法在计算直线斜率与误差项时用到小数与除法，可以改用整数以避免除法。改进后的算法称为bresenham算法。
- 由于算法中只用到误差项的符号，而且2dx为正，因此可作如下替换：e=e*2dx
  - e的初值为：e0 = -0.5*(2dx)=-dx
  - e的增量为：k*(2dx)=dy/dx*(2dx)=2dy
  - 当e大于0时，e的减量为：1*2dx= 2dx

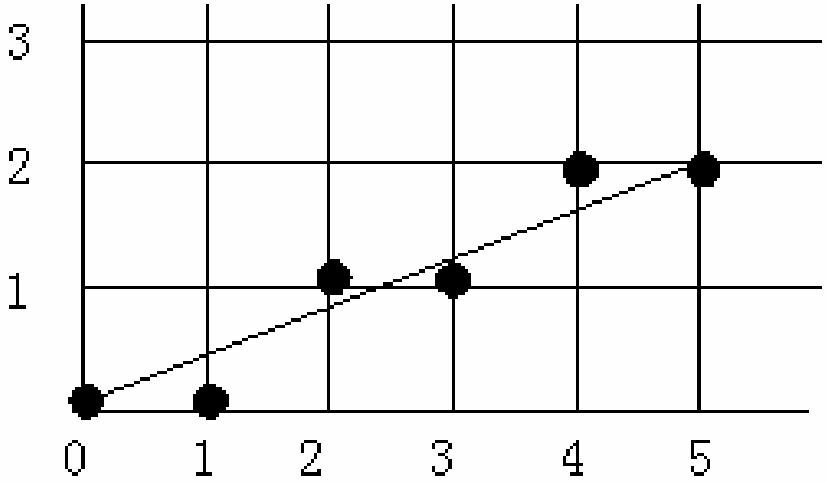
Bresenham算法程序

```
void Bresenhamline (int x0,int y0,int x1, int y1,int color)
{
    int x, y, dx, dy, e;
    dx = x1-x0;    dy = y1- y0;
    e=-dx;                                    //e的初值为-dx
    x=x0; y=y0;
    for (i=0; i<=dx; i++)     //  i从x0到x1也可，但速度是这个快
    {  Putpixel (x, y, color);
        x=x+1;
        e=e+2*dy;                             //e每次的增量为2*dy
        if (e>= 0)
            { y++, e=e-2*dx;}   //若e>=0则：y增1，e-2*dx
    }
}
```

Bresenham算法示例

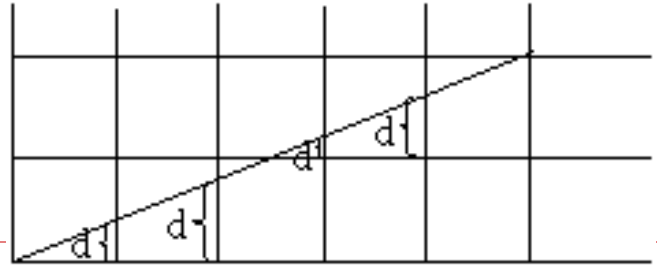
- 用Bresenham方法扫描转换连接两点P0（0,0）和P1（5,2）的直线段。
  - dx=5;dy=2;e=-dx=-5;增量2dy=4;e>0时减量2dx＝10

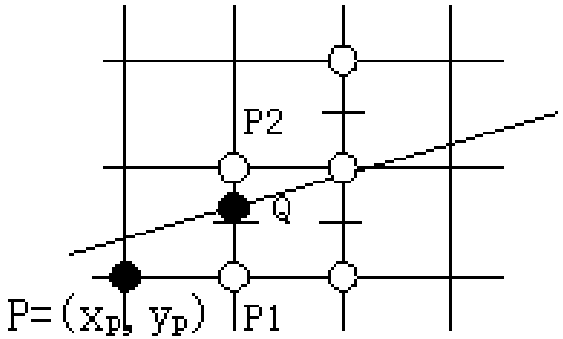
x	y	e	下点位置
0	0	-5	
1	0	-1	
2	1	-7	y+1
3	1	-3	
4	2	-9	y+1
5	2	-5	



三种直线绘制方法的总结：

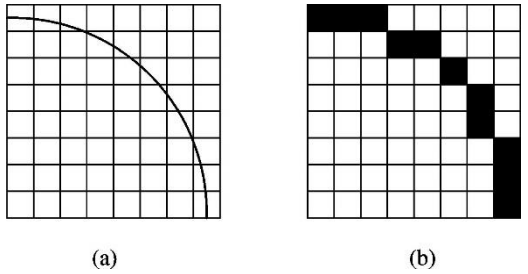
- DDA方法的原理：根据直线方程，X→Y
- 其他两种的原理：0<K<1时，下一像素只可能是当前像素的右点或右上点
  - 如何决定取右点或右上点？构造判别式（二者区别）
  - 中点算法：根据直线正负划分性，将中点坐标代入直线方程得到判别式；
  - Bresenham算法：误差项d->e

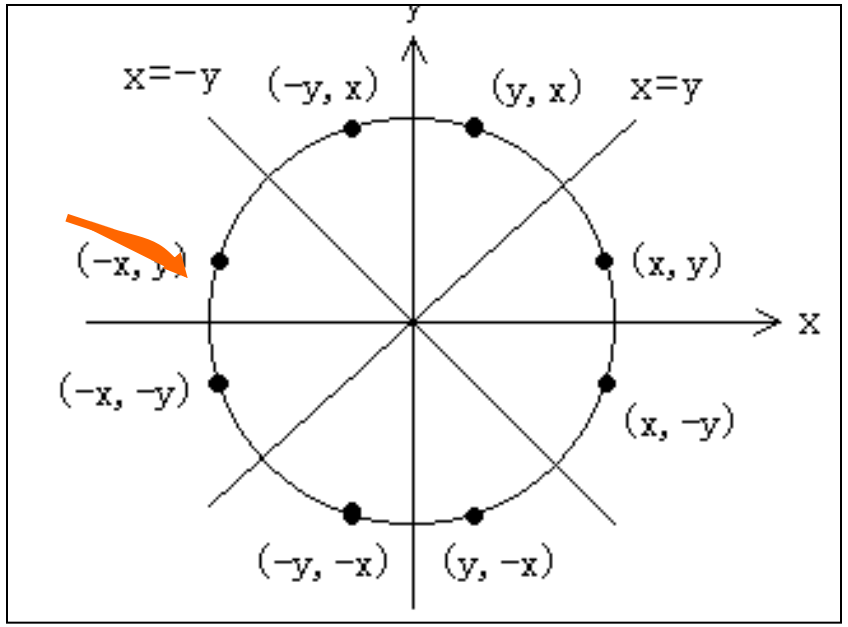




2.3 扫描转换圆弧

- 处理对象：圆心在原点的圆弧，如不在则使用坐标变换强制变换过来。
- 圆的八对称性
  - 因此我们只要转换八分之一圆弧就可以了。





显示圆弧上的八个对称点的算法

```
void CirclePoints(int x,int y,int color)
{
  Putpixel(x,y,color); Putpixel(y,x,color);
  Putpixel(-x,y,color); Putpixel(y,-x,color);
  Putpixel(x,-y,color); Putpixel(-y,x,color);
  Putpixel(-x,-y,color); Putpixel(-y,-x,color);
}
```

两种直接离散的圆弧转换方法：

- 离散点：

利用隐函数方程 $x^2 + y^2 = R^2$

$$(x_i, y_i = \sqrt{R^2 - x_i^2}) \xrightarrow{\text{取整}} (x_i,\ y_{i,r})$$

- 离散角度：

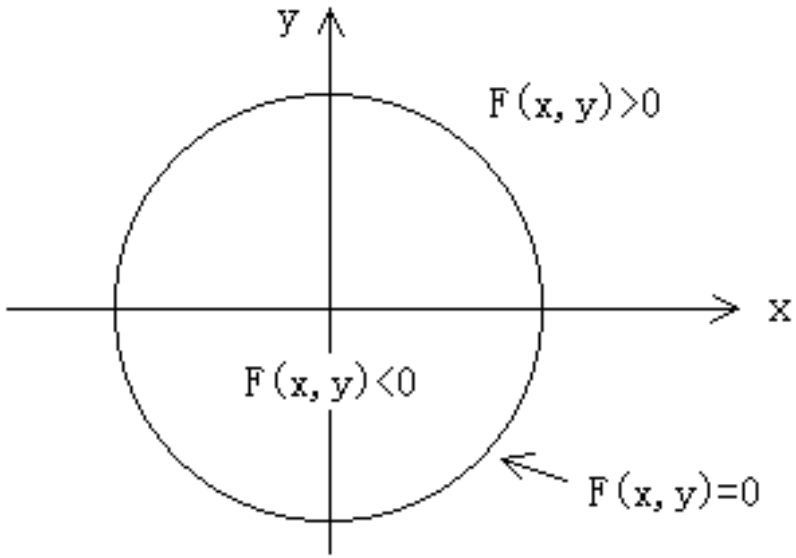
利用参数方程 $\begin{cases} x = R\cos\theta \\ y = R\sin\theta \end{cases}$

$$(round(R\cos\theta_i), round(R\sin\theta_i))$$

- 开根号，三角函数运算，计算量大，不可取。

圆弧的正负划分性

- 圆弧的隐函数：$F(X,Y)=X^2+Y^2-R^2=0$
  - 圆弧外的点：$F(X, Y) > 0$
  - 圆弧内的点：$F(X, Y) < 0$



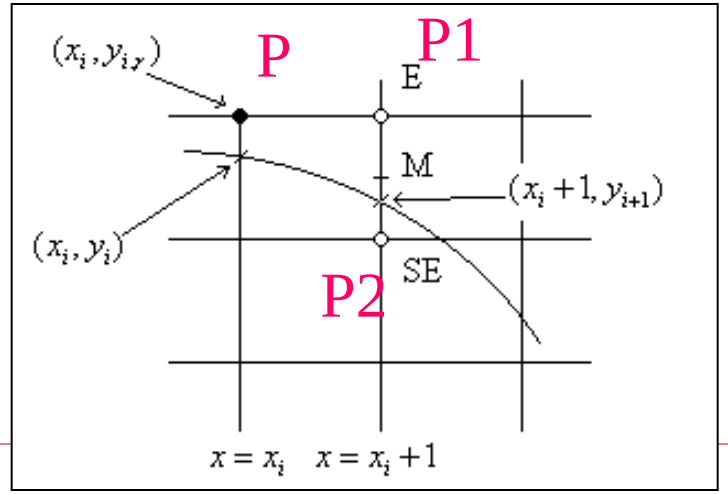
生成圆弧的中点算法

- 已知：半径(中点默认在原点)
- 圆弧的隐函数：$F(X,Y)=X^2+Y^2-R^2=0$

  考虑第一象限上侧1/8圆，则切线斜率k in [-1,0]，

  因此点P的下一像素只能在右侧点p1和右下侧点p2中选择。
- 中点 M=(Xp+1,Yp-0.5)
  - 当F(M) < 0时，M在圆内

    说明P1距离圆弧更近，取P1；
  - 当F(M) > 0时，取P2



算法

- M代入F(x,y)，构造判别式

$d=F(M)=F(Xp+1,Yp-0.5)=(Xp+1)^2+(Yp-0.5)^2-R^2$

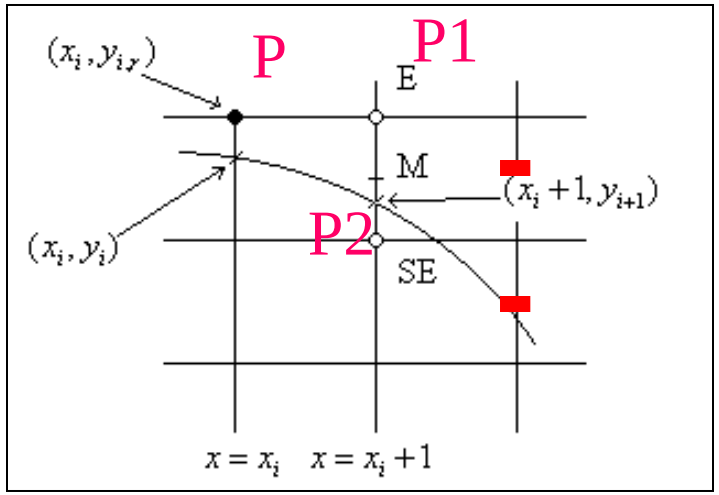
1）若d＜0，取P1(正右方向)，再下一个象素的判别式为：

d1=F(Xp+2,Yp-0.5)=d+2Xp+3，　　d的增量为2Xp+3；

2）若d≥0，取P2(沿右下方向)，再下一个象素的判别式为：

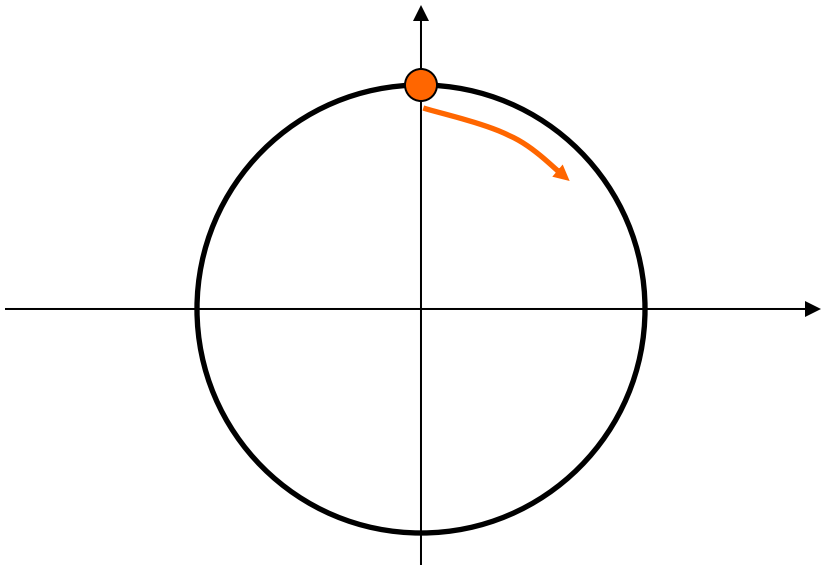
d2=F(Xp+2,Yp-1.5)=d+(2Xp+3)+(-2Yp+2)

d的增量为2(Xp-Yp)+5



- d的初始值：
  - 第一个象素(0,R)
  - 因此判别式初始值：

$$d0=F(1, R-0.5)=1.25-R$$



中点画圆算法的优化

- 上述算法中有浮点数，用e=d-0.25代替,所以：
  - 初始值：d0=1.25–R 对应于e0=1-R
  - 判别式：d<0对应于e<-0.25

    (因为e的初值e0为整数，运算过程中的增量2(Xp-Yp)+5或2Xp+3也为整数，故e始终为整数，所以 e<-0.25 等价于e<0)
- 经上述优化，程序里不存在浮点数。

优化后的中点画圆算法(正式算法)

```
void MidPointCircle(int r, int color)
{
    int x,y,d;
    x=0;   y=r;   e=1-r;                    // 初值e=1-r
    Circlepoints (x,y,color);                 // 画八分对称性的其他点
    while(x<=y)                              // 画到直线x=y结束
   {     if(e<0)   e+=2*x+3;                // d<0，取右侧点，d增
          else   { e+=2*(x-y)+5; y--;} // d>=0，取右下点，d增
          x++;
          Circlepoints (x,y,color);          // 画八分对称性的其他点
  }
}
```

举例

- 用中点画圆方法绘制圆心在(0,0),半径为6的圆
  - 起点坐标(0,r)即(0,6)
  - e=1-r=-5;
  - e<0时，取右点，判别式增量e1=2x+3
  - e>=0时，取右下点，判别式增量e1=2(x-y)+5

x	y	e	下点位置
0	6	-5	e<0,右点，增量3
1	6	-2	e<0,右点，增量5
2	6	3	e>0,右下点，增量-3
3	5	0	e>0,右下点，增量1
4	4	1	X>=Y，结束

每一步，根据圆的8对称性，
同时绘制(6,0),(0,-6), (0,6),(1,6),(-1,6),(1,-6)...

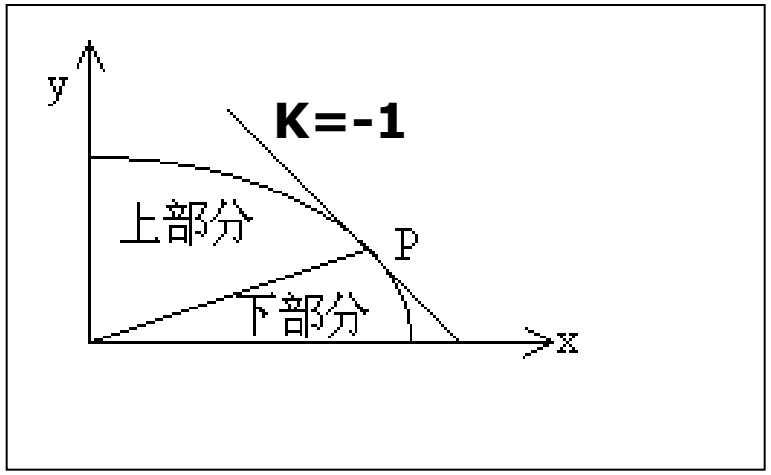
椭圆的扫描转换

- $F(x,y)=b^2x^2+a^2y^2-a^2b^2=0$
- 椭圆的四分对称性，只考虑第一象限椭圆弧生成
- 把椭圆弧分为上下两部分分别绘制，以切线斜率为-1的点作为分界点。(why?)
- 椭圆上一点处的法向：

  N(x,y)

  $= (F)'_x\, i + (F)'_y\, j$

  $= 2b^2 x\, i + 2a^2 y\, j$

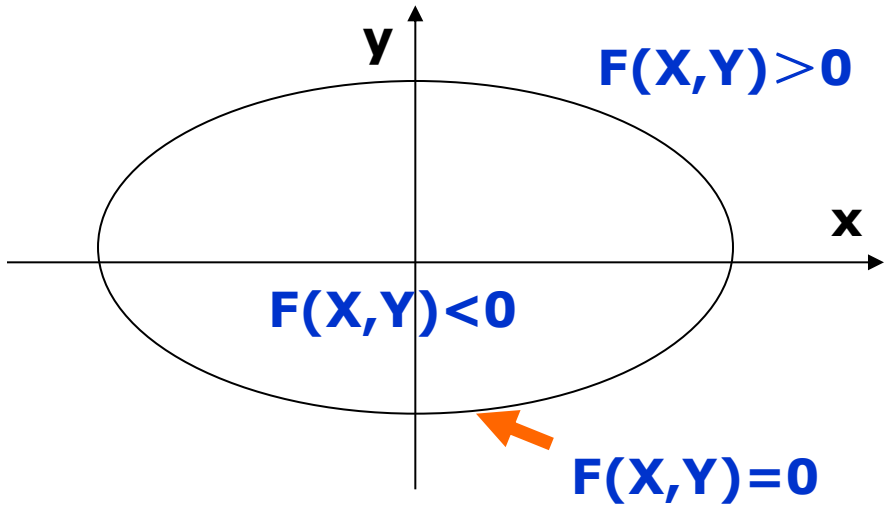


椭圆的正负性划分

- $F(x, y) = b^2x^2 + a^2y^2 - a^2b^2 = 0$
  - 椭圆弧外的点：

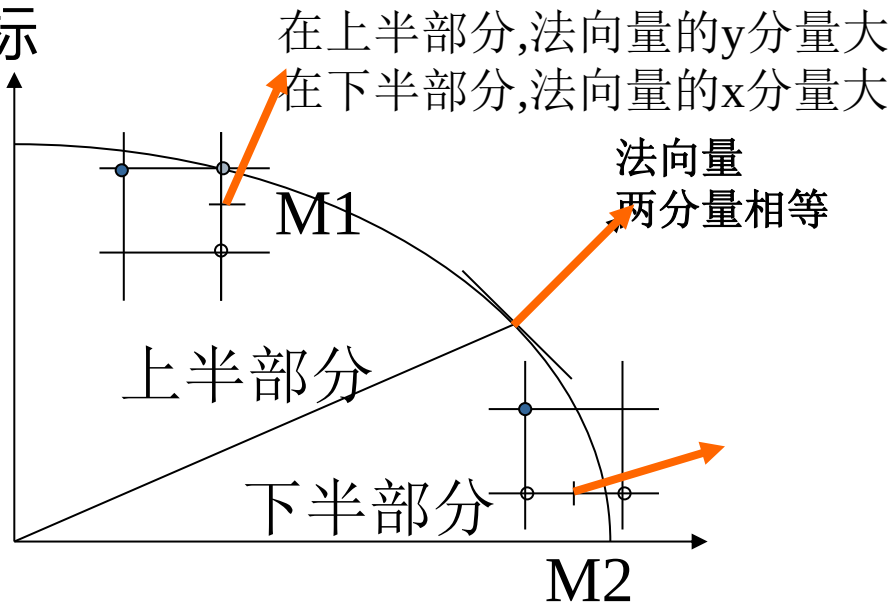
    F(X，Y)＞0
  - 椭圆弧内的点：

    F(X，Y)＜0



何时从上段进入下段

- 在当前中点处，法向量（ 2b2 (Xp+1)， 2a2 (Yp-0.5))的y分量比x分量大,即：b2 (Xp+1) < a2 (Yp-0.5), 而在下一中点，不等式改变方向，则说明椭圆弧已从上部分转入下部分。
- 或按书上，直接计算分界点P的坐标

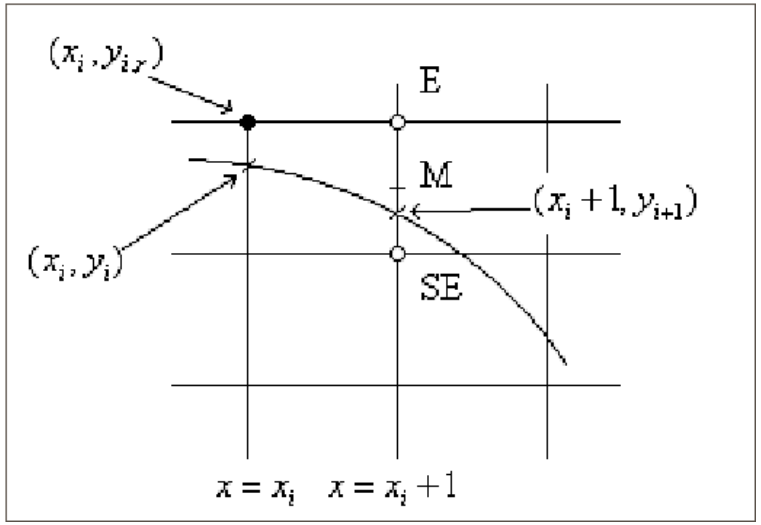


椭圆的中点画法

- 与圆弧中点算法类似：确定一个象素后，接着在两个候选象素的中点计算一个判别式的值，由判别式的符号确定更近的点。
- 先讨论椭圆弧的上部分
  - 已知(Xp，Yp)，求下一点
  - 切线斜率在[-1,0]间，下一点是右点或右下点
  - 中点(Xp+1,Yp-0.5)
  - 构造判别式：

d1=F(Xp+1,Yp-0.5)

$= b^2(Xp+1)^2+a^2(Yp-0.5)^2-a^2b^2$



- 根据d1的符号来决定下一像素是取正右方的那个，还是右下方的那个。
  - 若d1＜0，中点在椭圆内，取正右方象素，再下一个象素的判别式为：

    d1'=F(Xp+2,Yp-0.5)=d1+b^2(2Xp+3)

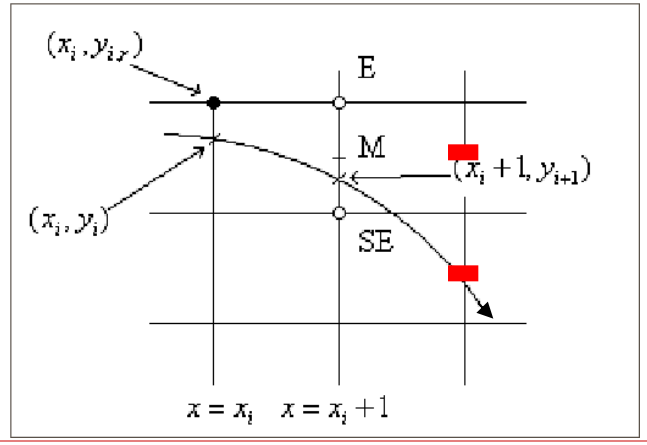
    d1的增量为：b^2(2Xp+3)
  - 当d1≥0，中点在椭圆外，取右下方象素，再下一个象素的判别式为：

    d1'=F(Xp+2,Yp-1.5)

    =d1+b^2(2Xp+3) +a^2(-2Yp+2)

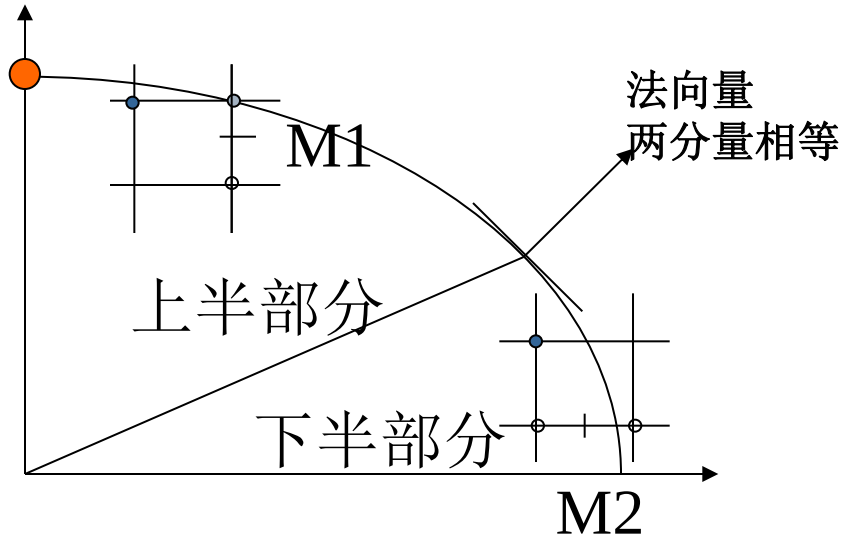
    d1的增量为：

    b^2(2Xp+3)+a^2(-2Yp+2)



- d1的初值：
  - 椭圆弧起点为(0，b)，第一个中点为(1,b-0.5)
  - 判别式d1的初值：

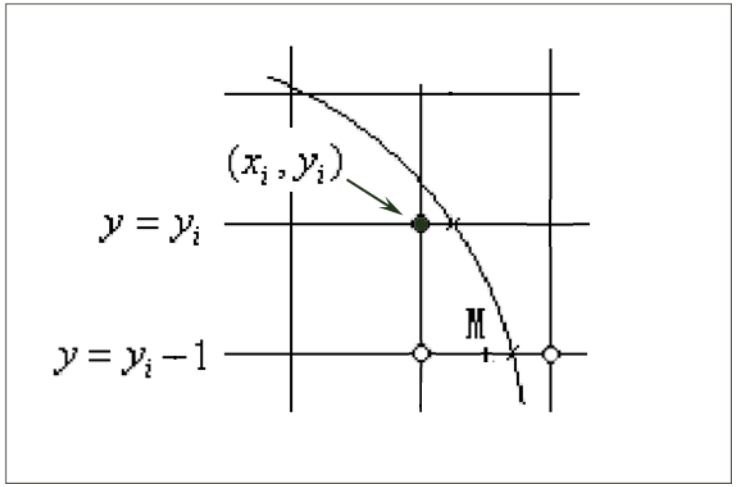
$$d1_0=F(1,b-0.5)=b*b+a*a(-b+0.25)$$



- 讨论椭圆弧的下半部分
  - 已知(Xi，Yi)，求下一点
  - 切线斜率在[-1,-∞]间，因此现在我们关心的是y递减1时，x递增多少，并对x取整。
  - (Xi，Yi)的下一像素可能是下点或右下点。
  - 中点M(Xi+0.5,Yi-1)
  - 构造判别式：

    d2 = F(Xi+0.5,Yi-1)

    = $b^2(Xi+0.5)^2+a^2(Yi-1)^2-a^2b^2$



- 若d2<0,则中点在椭圆弧内，选择右下点，再下一个象素的判别式为：

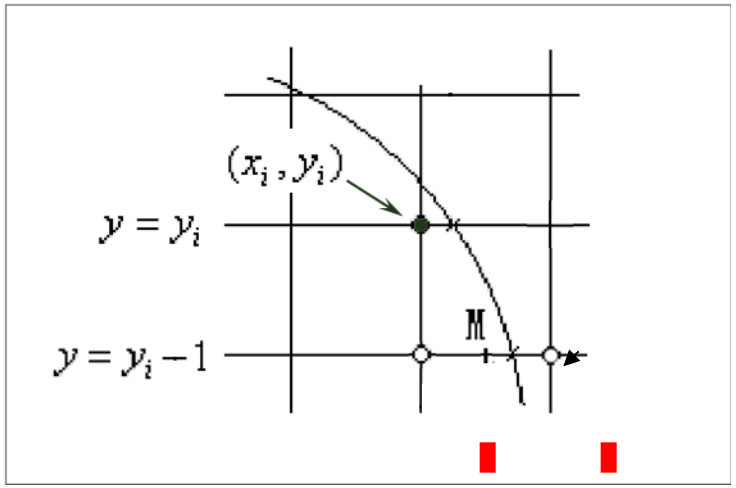
  d2′ = F(Xi+1.5,Yi-2)

  = d2 + $b^2(2Xi+2)+a^2(-2Yi+3)$

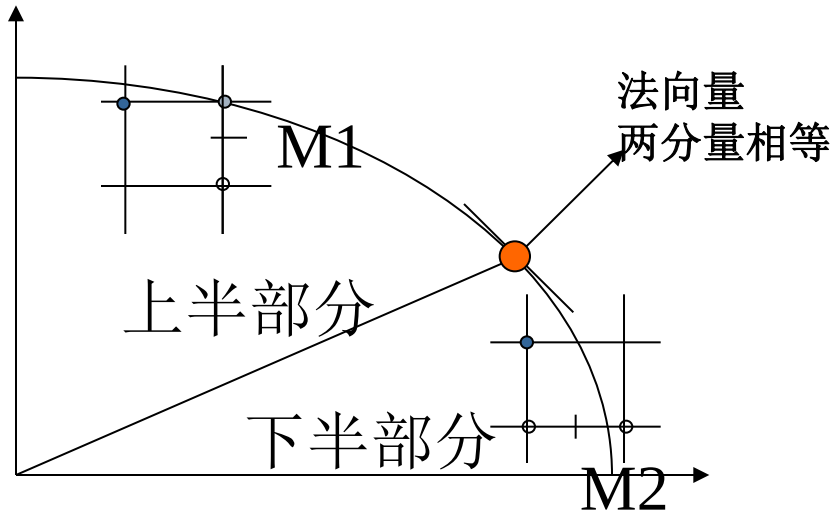
- 若d2>=0,则中点在椭圆弧外，选择下点，再下一个象素的判别式为：

  d2′ = F(Xi+0.5,Yi-2)

  = d2 + $a^2(-2Yi+3)$



- d2的初值：
  - $d2= b^2(xp+0.5)^2 +a^2(yp-1)^2– a^2b^2$
  - 其中(xp,yp)为分界点P的坐标，也是在程序中，椭圆弧上半段算毕，跳出循环时的坐标。
- 下半部分弧的终止条件为 y=0



程序

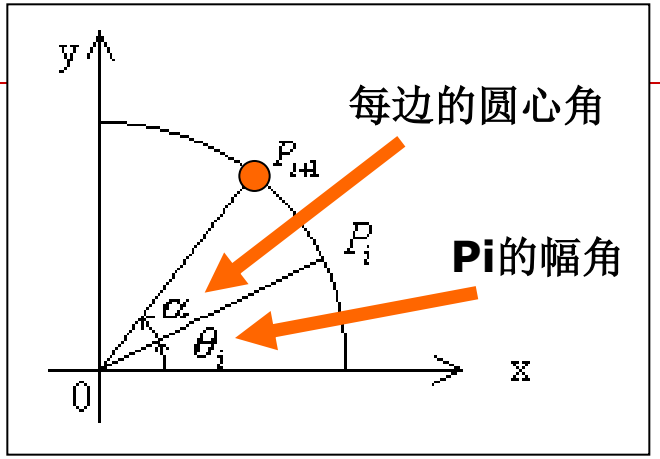
```
MidpointEllipe(int a, int b, int color)
{  int x,y;  float d1,d2;
    x = 0; y = b;
    d1 = b*b +a*a*(-b+0.25);     //d1置初值
    putpixel(x,y,color);
   while( b*b*(x+1) < a*a*(y-0.5))    //直到上下分界点
   {
        if (d1<0) {d1 +=b*b*(2*x+3); x++;} //取右点,d1增
         else  {d1 +=(b*b*(2*x+3)+a*a*(-2*y+2));
                  x++;  y--;}    //取右下点，d1增
           putpixel(x,y,color);
   }//上半部分结束
```

```
//下半部分开始
    d2=sqr(b*(x+0.5)) +sqr(a*(y-1))–sqr(a*b);
                          //d2置初值，sqr为平方函数
    while(y >0)           //直到x坐标轴
      {
        if(d2<0) {d2 +=b*b*(2*x+2)+a*a*(-2*y+3);
                        x++; y--; }     //取右下点
         else  {d2 += a*a*(-2*y+3);  y--;}  //取下点
         putpixel(x,y,color);
     }
}
```

生成圆弧的多边形逼近法

- 当一个正多边形边数足够多的时候，该多边形可以和圆任意接近。因此在允许的范围内，可以用正多边形代替圆。
- 得到各边端点坐标后，显示多边形的边可用扫描转换直线段的算法来实现。
- 分类:
  - 圆的正内接多边形迫近法
  - 圆的等面积正多边形迫近法

圆的正内接多边形迫近法



- 原理

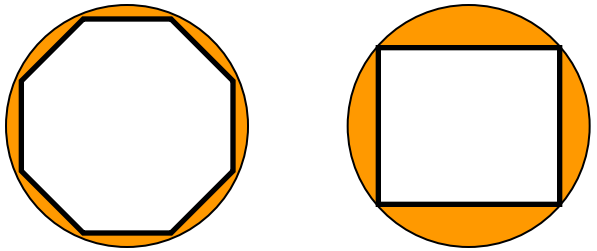
$$\lim_{n\to\infty}(n\text{边正内接多边形})=\text{圆}$$

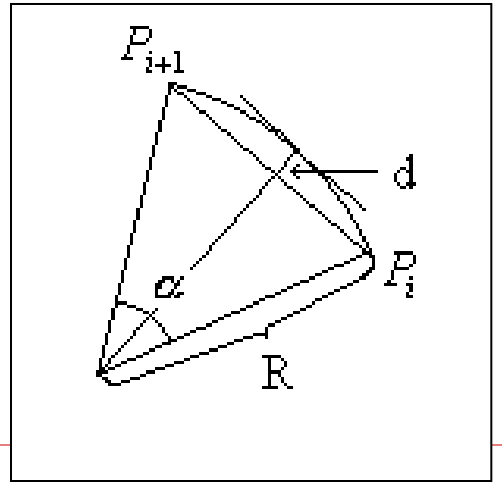
- 计算多边形各顶点的递推公式：

$$\begin{bmatrix} x_{i+1} \\ y_{i+1} \end{bmatrix} = \begin{bmatrix} R\cos(\theta_i+\alpha) \\ R\sin(\theta_i+\alpha) \end{bmatrix} = \begin{bmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix}$$

- 因为a是常数，sina, cosa只在开始时计算一次，所以一个顶点只需4次乘法，一个n边形的计算共4n次乘法，加直线段的中点算法的计算量。

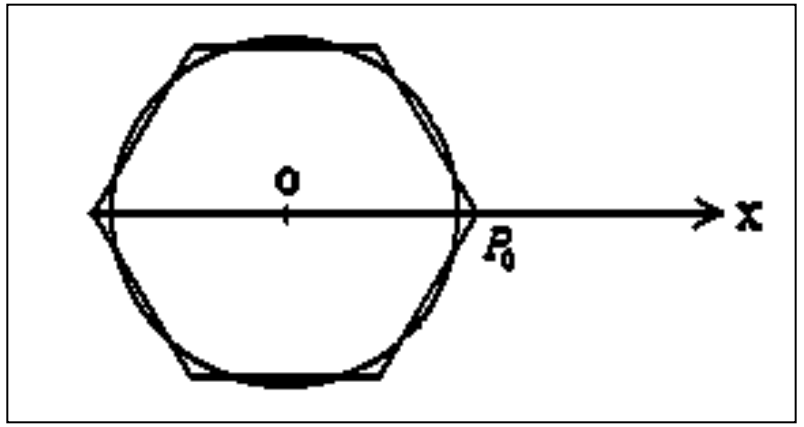
- 问题：给定最大逼近误差(最大距离) DELTA ， 如何确定多边形的边数n或a
  - 即要求：d=R–Rcos(a/2) <= DELTA ⟹ cos(a/2) >= (R–DELTA)/R
    ⟹ a<=2arccos (R–DELTA)/R ⟹ a不能超过 2arc cos (R–DELTA)/R
  - n = 360 /a
- 另外，用矢量运算可以简化计算，推出求顶点的逆推公式



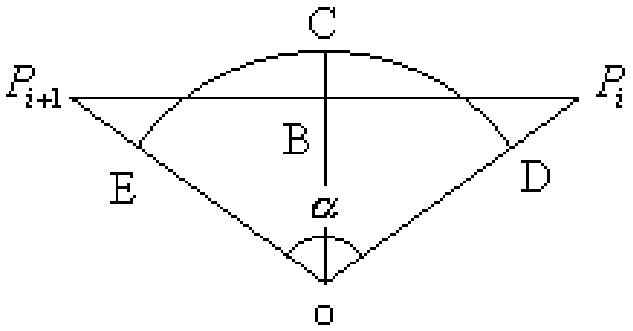


圆的等面积正多边形逼近法

- 步骤：
  - 根据右下图，求多边形径长OP_0 ($OP_0 > R$)
  - 根据用户给定的逼近误差值，确定边所对应的圆心角a
  - 然后根据递推公式求所有顶点坐标标值。



圆的等面积正多边形

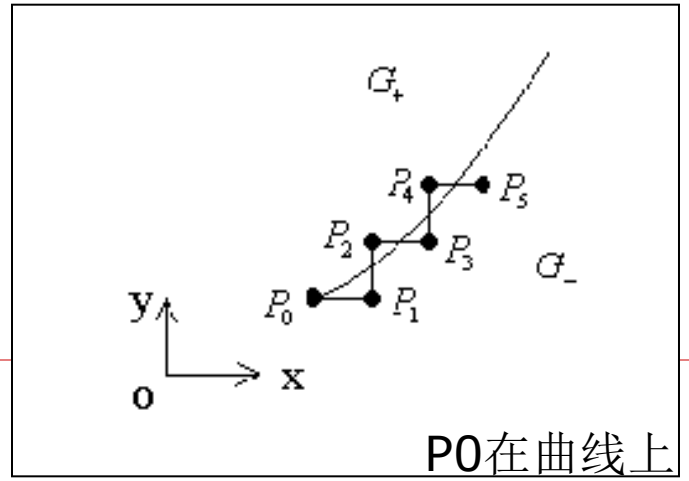


扇形ODCE的面积=三角形OP_iP_{i+1}的面积

生成圆弧的正负法

- 正负法基本原理
  - 设曲线方程为F(x,y)=0,它具有正负划分性，将平面分成了三个点集：
    - G_+={(x,y)| F(x,y)>0)};
    - G_0={(x,y)| F(x,y)=0)};
    - G_-={(x,y)| F(x,y)<0)};
  - 假定初始点P0∈G0，沿某方向（假定为X轴）前进△X时，到达G+或G-(假定为G-)中的P1,再沿另外一方向(Y轴)前进△Y,到达P2。

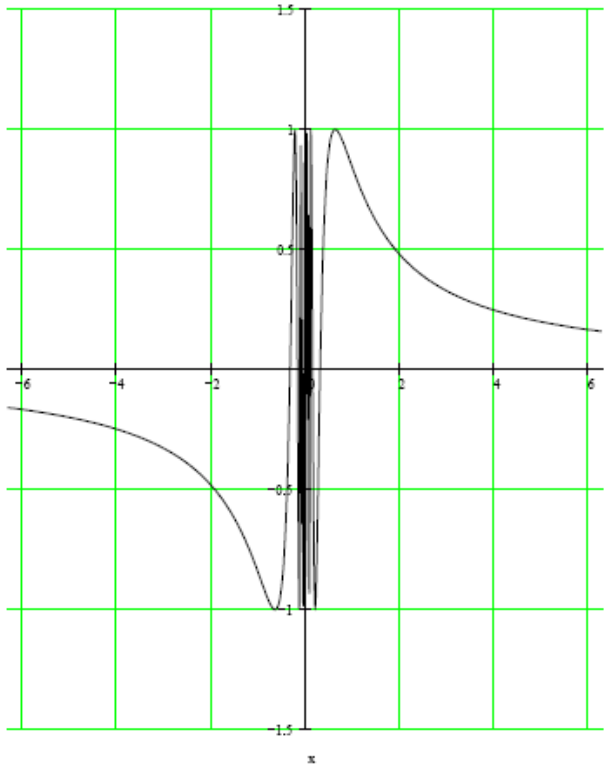
    若P2 ∈G+，则改变前进方向，否则继续向G+前进…



P0在曲线上

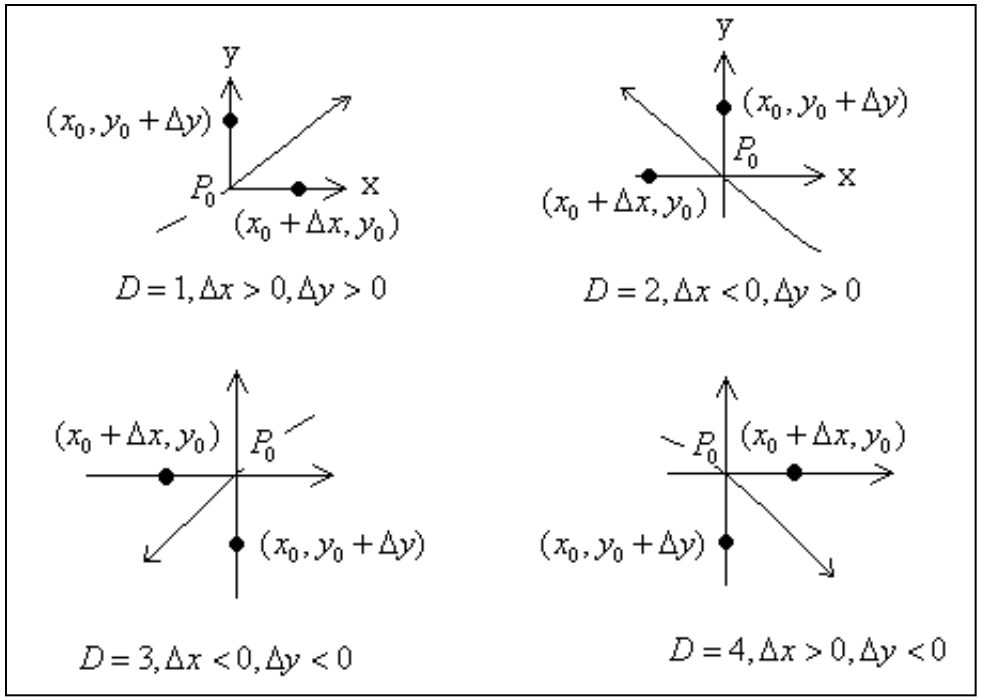
- 上述方法只有在一定条件下才能顺利进行，例如对y=sin1/x，近0处难以跟踪。
- 易画曲线
  - F(x,y)具有正负划分性
  - F(x,y)二阶导数连续
  - 曲线上各点的曲率半径大于步长，即在步长度量下，曲线是平坦的。
- 我们只考虑单调曲线

非单调分割成几段单调的再处理



初始定向

- 指从起始点出发向哪个方向前进，
- 或：以起始点为原点的局部坐标系中向哪个象限前进。
- 象限号决定 $\Delta x, \Delta y$ 的正负号。



前进规则

- 起始点P0在曲线上，第一步不妨先走 Δx，到达P1(x0+Δx,y0)；
- 第二步必须走 Δy，到达P2(x0+Δx,y0+ Δy)点；
- 接下去如何走则要根据F(P2)的符号确定。
- 假定当前在Pi(xi,yi)点上，决定下一步是前进 Δx 或 Δy 的规则称为前进规则。

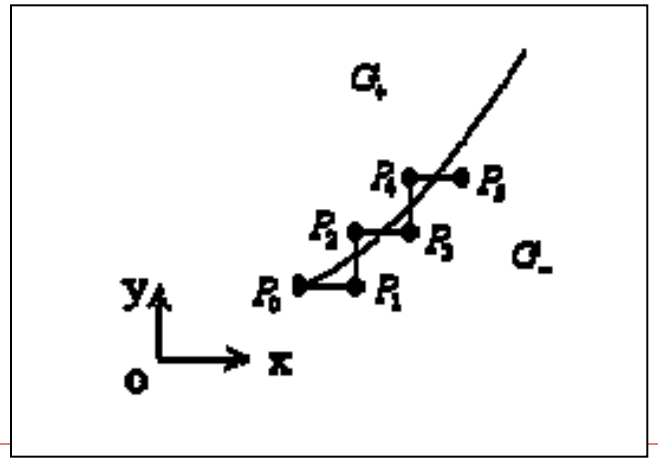
前进规则

- 单调曲线的前进规则：
  (1)当P_i与P_1在同侧时，前进Δy
  (2)当P_i与P_1在异侧时，前进Δx
- Pi与P1同侧即F(P1)和F(Pi)同号，因此取判定式

$$D(P_i)=F(P_i)\bullet F(P_1)=F(x_i,y_i)\bullet F(x_0+\Delta x,y_0)$$

(1)当$D(P_i)\geq 0$时，前进Δy

(2)当$D(P_i)\leq 0$时，前进Δx

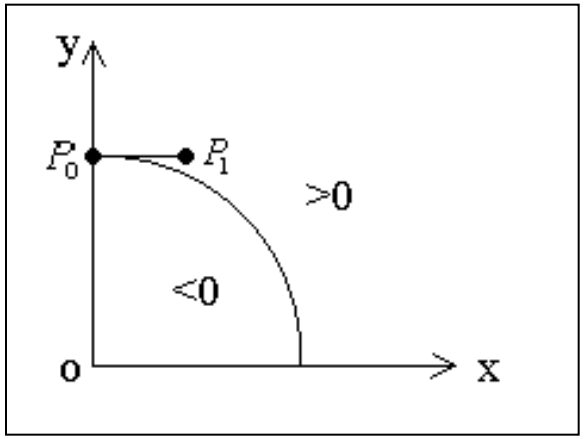


正负法生成圆弧

- 考虑第一像限圆弧段
- 圆弧是易画曲线
  - 取初始点P0(x0,y0) = (0,R)
  - 初始定向为：D = 4, △X=1, △y =-1,
  - 第二点：P1为(x0+1,y0) = (1,R)
  - 判别式：D(Pi) = F(Pi)F(P1) = F(Pi)F(1,R)

    F(1,R)为正，上式等价于： D(Pi) = F(Pi) =F(x,y)

    判别式： $D(Pi) = x_i^2 + y_i^2 - R^2$



正负法生成圆弧

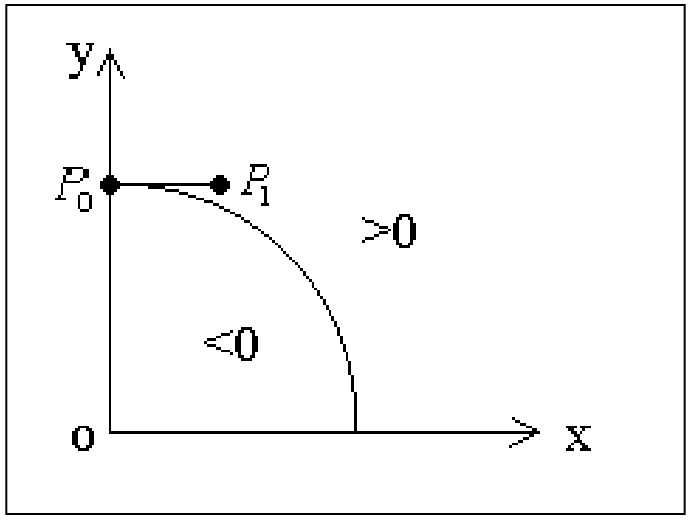
- 当D(Pi) >=0时，前进△y

  $x_{i+1}=x_i$, $y_{i+1}=y_i-1$;

- 当D(Pi) <0时，前进△x

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

- 为更加快速，可推导递推公式。(略)

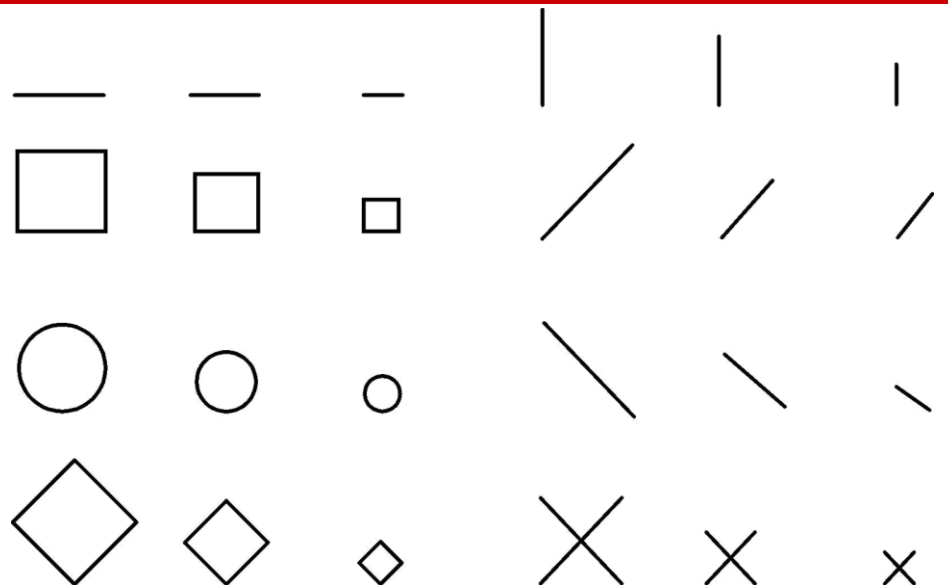


2.5 线画图元的属性控制

- 线宽控制：
  - 如何画线宽大于一个像素的线?
    - 选取宽度适当的刷子在屏幕上绘图，刷子轨迹即为所求。
  - 问题：刷子形状如何?
    - 圆形，长方形…
  - 对于非圆形刷子，绘制方式如何?
    - 水平、垂直、旋转…
  - 线型如何?
    - 实线，虚线，点划线…

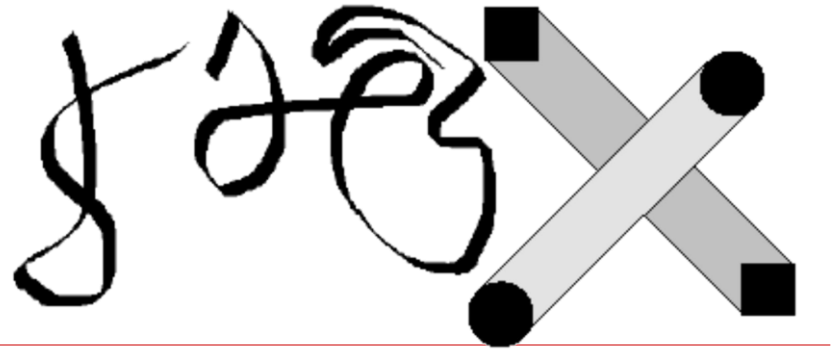
刷子

- 图形是由具有一定尺寸大小的像素点构成的。
- 绘制无宽度的线条时我们每算出一个点就用最小的像素点表示。这时用具有一定尺寸的点表示，就可以绘出具有一定宽度的线条。
- 有尺寸的点也可以有不同的几何形状。大小、形状不同的点就构成了绘制线条时不同的刷子。
- 刷子的形状可以是方形的、圆形的或线形的。通过对这些基本画笔形状的旋转、放大、缩小，甚至改变颜色，就可得到不同的画笔。



不同大小、形状的刷子

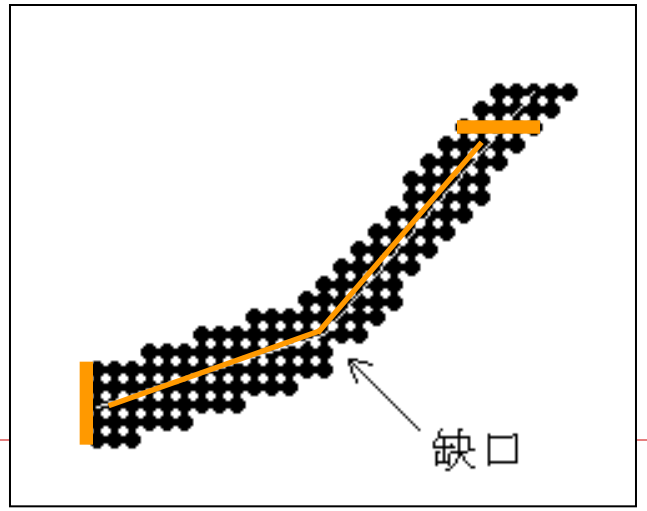
移动不同形状画笔绘制的线条

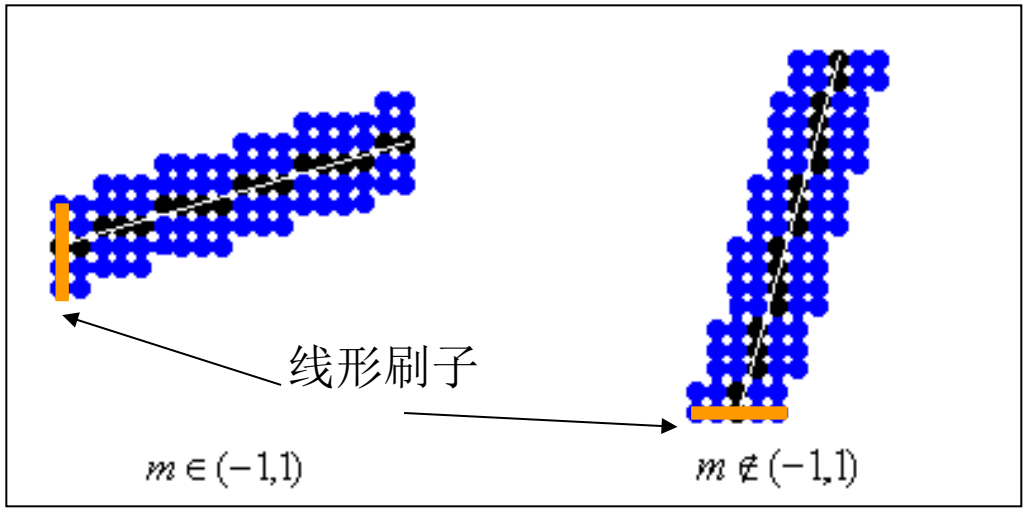


像素复制方法（线刷子）

- 斜率在(-1,1)之间用垂直刷子，否则用水平刷子。
- 优点：实现简单，效率高
- 缺点：1 线段两端要么为水平的，要么是竖直的；

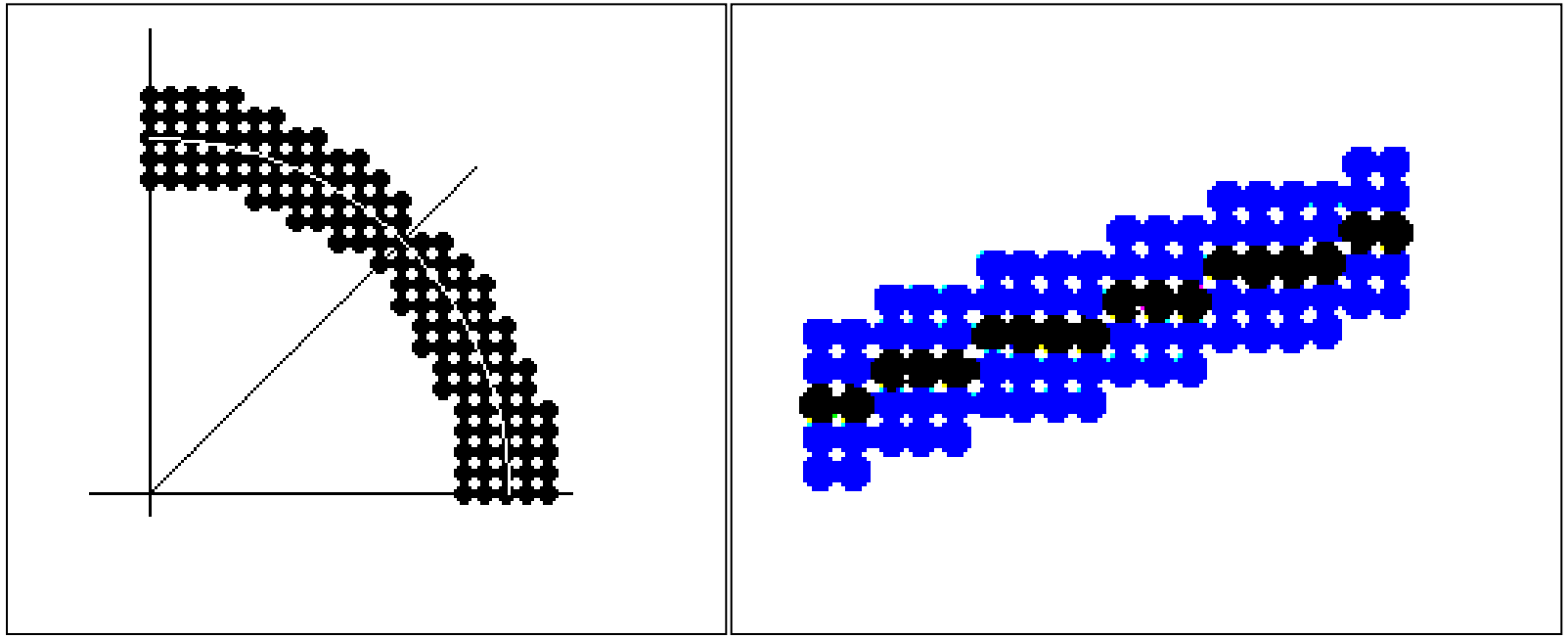
  2 折线顶点处有缺口





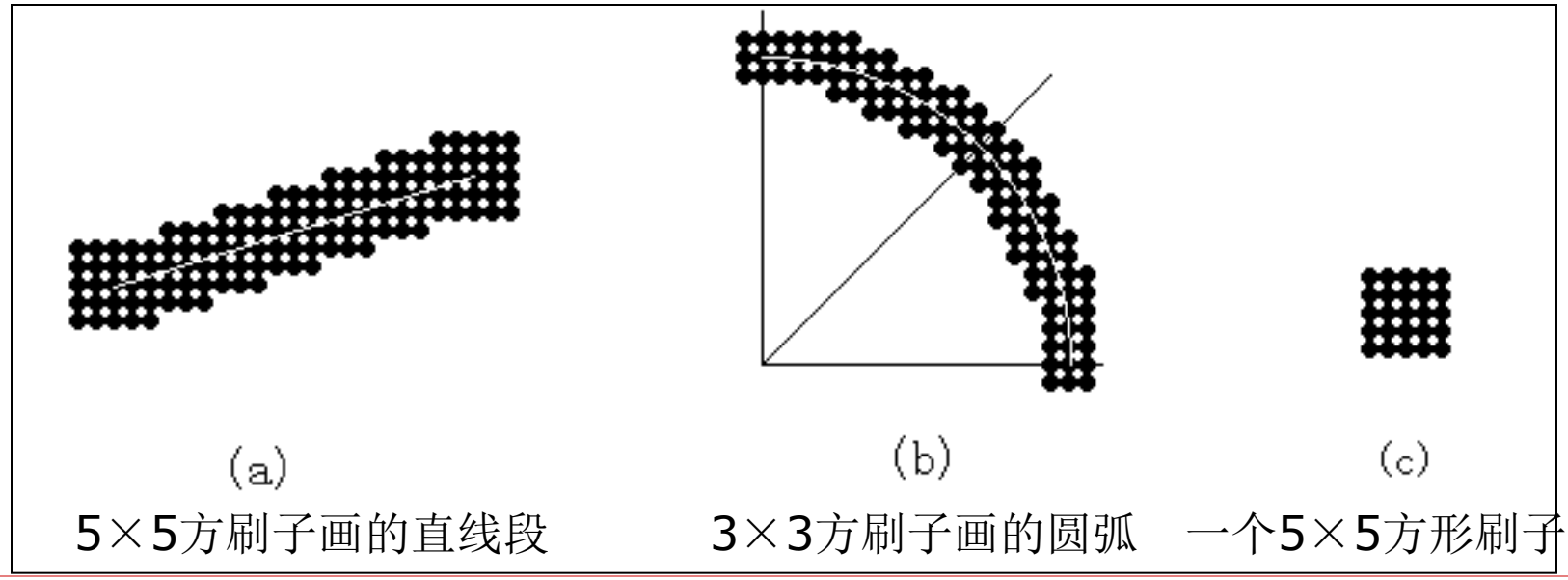
3 图元的宽度不均匀

4 产生宽度为偶数像素的图元效果不好



移动刷子

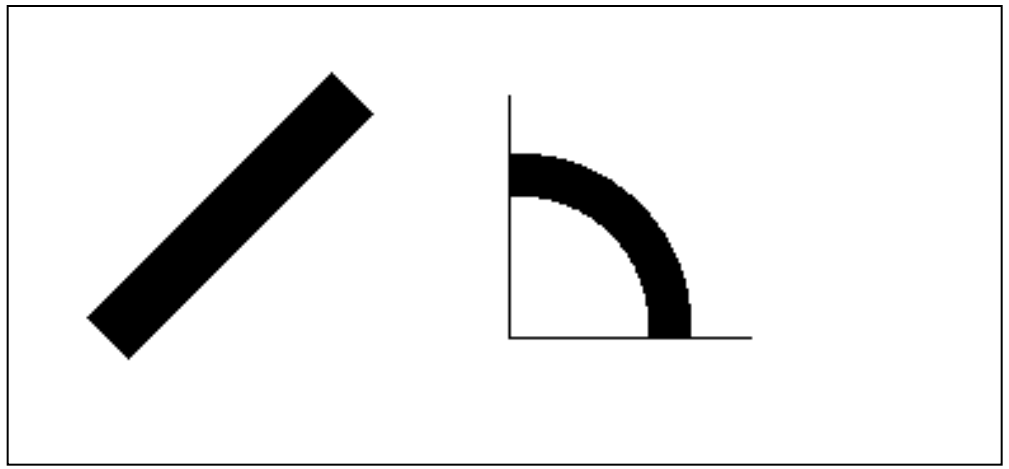
- 把宽度为指定线宽的刷子的中心沿直线移动，获得相应宽图元。



- 优点：
  - 实现简单
  - 通过改变刷子形状可以不同的线宽效果
- 缺点：
  - 重复写象素，但可以通过改进算法避免
  - 方形刷子也有宽度不均问题

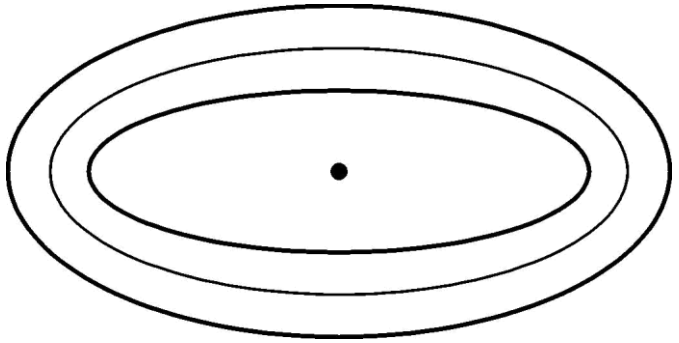
利用填充图元

- 要产生宽度为k的图元，可以首先计算出距离原始的理想图元k/2的两条等距线，将其连接起来并填充。



利用填充图元

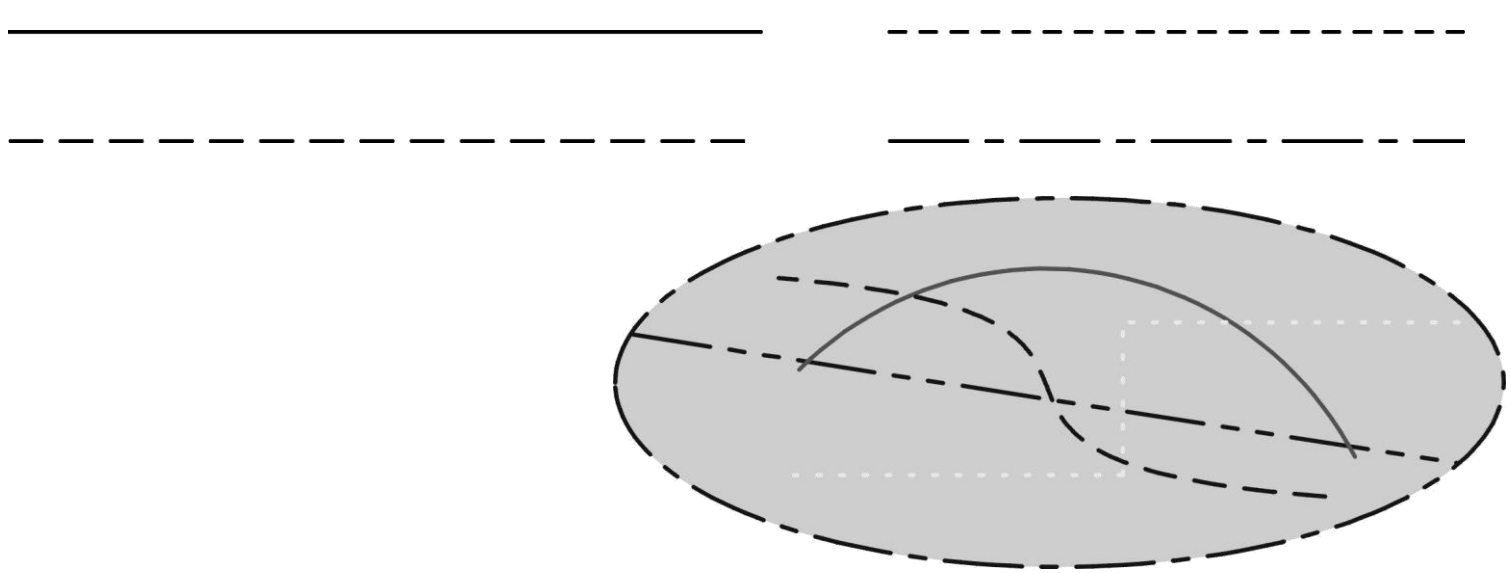
- 优点：
  - 生成的图形质量高
- 缺点
  - 计算量大
  - 有些图形的等距线难以获得(如椭圆)



有宽度的椭圆曲线

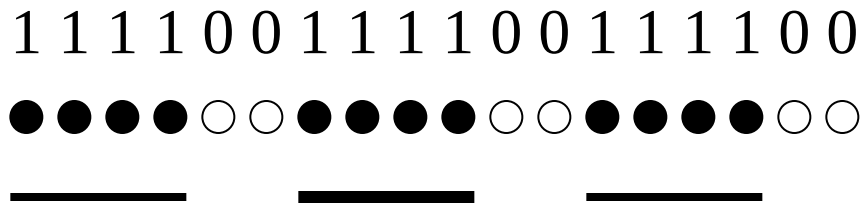
线型控制

- 不同线型的各类线条



线型控制

- 一般用位屏蔽器实现，如：用一个16位整数表示一个位串，当对应为1时，显示像素，否则不显示。



- 以16个像素为周期重复，在程序实现时将无条件写像素Putpixel(x,y,color)改为有条件：

```
if(位串[i%16]) Putpixel(x,y,color);
```

其中i为整型计数器，指示当前像素的序号。

END

提示：WIN API中的绘图方法

```
#include<windows.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>


long WINAPI WndProc(HWND hWnd,UINT iMessage,WPARAM wParam,LPARAM lParam);
BOOL InitWindowsClass(HINSTANCE hInstance);
BOOL InitWindows(HINSTANCE hInstance,int nCmdShow);
HWND hWndMain;


LRESULT CALLBACK WndProc(HWND,UINT,WPARAM,LPARAM);
```

```
//program starting.
int WINAPI WinMain(HINSTANCE hInstance,HINSTANCE hPrevInst,LPSTR lpszCmdLine,int nCmdShow)
{
	MSG   msg;
	if(!InitWindowsClass(hInstance))
		return FALSE;

	if(!InitWindows(hInstance,nCmdShow))
		return FALSE;

	//Core message looping
	while(GetMessage(&msg,NULL,0,0))
	{
		TranslateMessage(&msg);
		DispatchMessage(&msg);
	}

	return msg.wParam;

}
```

```
//main wndProc function: message looping

long WINAPI WndProc(HWND hWnd,UINT iMessage,WPARAM wParam,LPARAM
    lParam)
{
    HDC hDC;
    HBRUSH hBrush;
    HPEN hPen;
    PAINTSTRUCT PtStr;

    switch(iMessage)
    {
```

```
case WM_PAINT:
        //First draw,a black line
        hDC=BeginPaint(hWnd,&PtStr);
        hPen=(HPEN)GetStockObject(NULL_PEN);//get empty brush
        SelectObject(hDC,hPen);
        hBrush=(HBRUSH)GetStockObject(BLACK_BRUSH);
        SelectObject(hDC,hBrush);

        hPen=CreatePen(PS_SOLID,2,RGB(255,0,0));//create pen
        SelectObject(hDC,hPen);

        DDALine(0,0,200,200,1,hDC);

        DeleteObject(hPen);
        DeleteObject(hBrush);
        EndPaint(hWnd,&PtStr);

        return 0;
```

```
case WM_LMouseButtonDown:
        ……
        return 0;
case WM_LMouseButtonUp:
        ……
        return 0;
case WM_DESTROY:
        PostQuitMessage(0);
        return 0;


default:
        return DefWindowProc(hWnd,iMessage,wParam,lParam); }}
```

```
//Init the Window to show out.
BOOL InitWindows(HINSTANCE hInstance,int nCmdShow)
{       HWND hWnd;
        hWnd=CreateWindow("WinFill",
                                "图形绘制算法示例程序",
                                        WS_OVERLAPPEDWINDOW,
                                CW_USEDEFAULT,
                                0,
                                CW_USEDEFAULT,
                                0,
                                NULL,
                                NULL,
                                hInstance,
                                NULL
                        );
        if(!hWnd)
                return FALSE;
        hWndMain=hWnd;
        ShowWindow(hWnd,nCmdShow);
        UpdateWindow(hWnd);
        return TRUE; }
```

```
//Set wndClass Propertity
BOOL InitWindowsClass(HINSTANCE hInstance)
{
	WNDCLASS wndClass;

	wndClass.cbClsExtra=0;
	wndClass.cbWndExtra=0;
	wndClass.hbrBackground=(HBRUSH)GetStockObject(WHITE_BRUSH);
	wndClass.hCursor=LoadCursor(NULL,IDC_ARROW);
	wndClass.hIcon=LoadIcon(NULL,"END");
	wndClass.hInstance=hInstance;
	wndClass.lpfnWndProc=WndProc;
	wndClass.lpszClassName="WinFill";
	wndClass.lpszMenuName=NULL;
	wndClass.style=CS_HREDRAW|CS_VREDRAW;

	return RegisterClass(&wndClass);
}
```