第5章 计算几何

计算几何是几何学的一个重要分支，也是计算机科学的一个分支，研究解决几何问题的算法。在现代工程与数学、计算机图形学、机器人学、VLSI 设计、计算机辅助设计等学科领域中，计算几何都有重要应用。

计算几何问题的输入一般是关于一组几何物体（如点、线）的描述；输出常常是有关这些物体相关问题的回答，如直线是否相交、点围成的面积等问题。一般来说，计算几何问题可以分为如下 3 类。

1. 计算求解题

解这一类题除了需要有扎实的解析几何的基础，还要全面地看待问题，仔细地分析题目中的特殊情况，比如求直线的斜率时，直线的斜率为无穷大；求两条直线的交点时，这两条直线平行等。这些都要依靠平时学习的积累。

2. 存在性问题

这一类问题可以用计算的方法来直接求解，如果求得了可行解，则说明是存在的，否则就是不存在的，但是模型的效率同模型的抽象化程度有关，模型的抽象化程度越高，它的效率也就越高。几何模型的抽象化程度是非常低的，而且存在性问题一般在一个测试点上有好几组测试数据，几何模型的效率显然是远远不能满足要求的，这就需要对几何模型进行一定的变换，转换成高效率的模型。

3. 最佳值问题

这类问题是计算几何题中比较有难度的问题，一般没有什么非常有效的算法能够求得最佳解，最常用的是用近似算法去逼近最佳解，近似算法的优劣也完全取决于得出的解与最优解的近似程度。

5.1 矢量

专题讲解

矢量分析是高等数学的一个分支，主要应用于物理学（如力学分析）。在一些计算几何问题中，矢量和矢量运算的一些独特的性质往往能发挥出十分突出的作用，使问题的求解过程变得简洁而高效。熟练掌握一些矢量分析的方法，并灵活地加以运用，就能轻松地解决许

多看似复杂的计算几何题，或者会对求解这类题目有很大帮助，甚至还有一些计算几何问题是一定要用矢量方法解决的。

1. 直线

1）直线的方程

一般形式为：$ax+by+c=0$，或 $y=kx+b$。式中 k 称为直线的斜率，b 称为截矩。

如图 5.1 所示，直线 L_1 的方程为：$\frac{3}{4}x-y+1=0$，L_2：$y=3$，L_3：$x=4$。

2）直线的斜率

如图 5.2 所示，过两点 P_1、P_2 可以决定一条直线。斜率为：$k=\frac{y_2-y_1}{x_2-x_1}$。

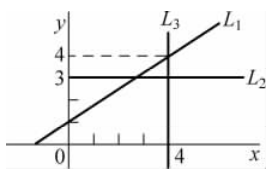

图 5.1　直线方程

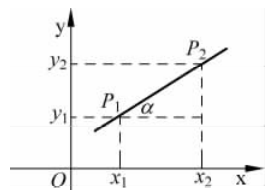

图 5.2　直线斜率

特别地：当 $y_1=y_2$ 时，斜率 $k=0$（如图 5.1 中的 L_2）。

当 $x_1=x_2$ 时，k 不存在（如图 5.1 中的 L_3）。

当 x_1 与 x_2 无限接近时，斜率 k 趋于无穷大，对这种情况在编程时要特别小心。

3）两条直线垂直

若两条直线垂直则它们的斜率乘积等于 -1。

2. 线段

1）凸组合

两个不同的点 $P_1(x_1,y_1)$、$P_2(x_2,y_2)$ 的凸组合是满足下列条件的点 $P_3(x_3,y_3)$：

对某个 $0\leqslant\sigma\leqslant1$，有

$$x_3=\sigma x_1+(1-\sigma)x_2,\ y_3=\sigma y_1+(1-\sigma)y_2$$

一般也可以写成：

$$P_3=\sigma P_1+(1-\sigma)P_2$$

从直观上看，P_3 通过直线 P_1P_2，并且处于 P_1 和 P_2 之间（也包括 P_1、P_2 两点）的任意点。

2）线段

线段 P_1P_2 是两个相异点 P_1、P_2 的凸组合的集合，其中 P_1、P_2 称为线段的端点。

3. 向量（矢量）的概念

1）矢量

从平面上取一固定点 $P_1(P_1=(X_1,Y_1))$，从 P_1 出发向某点 $P_2(P_2=(X_2,Y_2))$ 引一条线段，有方向且有长度，这样的线段叫做有向线段，记作 $\overrightarrow{P_1P_2}$，它的长度记作 $\overrightarrow{P_1P_2}=$

$\sqrt{(X_1-X_2)^2+(Y_1-Y_2)^2}$，点 P_1 叫做它的始点，点 P_2 叫做它的终点；如果 P_1 是原点(0，0)，则把有向线段$\overrightarrow{P_1P_2}$简记为向量 $\boldsymbol{P}_2$；如果不规定方向，则记线段为$\overrightarrow{P_1P_2}$。

2) 矢量的斜率

既然矢量是有方向的，那么矢量的斜率 k 就是有正负之分的，具体如图 5.3 所示。

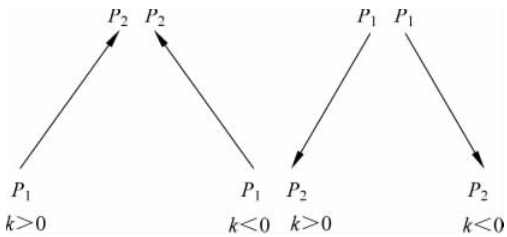

图 5.3　矢量斜率

3) $OA=\boldsymbol{a}$ 时(见图 5.4(a))

设 $OA=\boldsymbol{a}$，则有向线段 OA 的长度叫做向量(矢量)$\boldsymbol{a}$ 的长度或模。记作$|\boldsymbol{a}|$。

4) 夹角

两个非 0 矢量 $\boldsymbol{a}$、$\boldsymbol{b}$，在空间任取一点 O，作 $OA=\boldsymbol{a}$，$OB=\boldsymbol{b}$(见图 5.4(b))，则$\angle AOB$ 叫做矢量 $\boldsymbol{a}$ 与 $\boldsymbol{b}$ 的夹角，记作$<\boldsymbol{a},\boldsymbol{b}>$。若$<\boldsymbol{a},\boldsymbol{b}>=\pi/2$，则称 $\boldsymbol{a}$ 与 $\boldsymbol{b}$ 互相垂直，记作 $\boldsymbol{a}\perp\boldsymbol{b}$。

4. 矢量的加减法

以点 O 为起点、A 为端点作矢量 $\boldsymbol{a}$，以点 A 为起点、B 为端点作矢量 $\boldsymbol{c}$，则以点 O 为起点、B 为端点的矢量 $\boldsymbol{b}$ 称为矢量 $\boldsymbol{a}$ 与 $\boldsymbol{c}$ 的和 $\boldsymbol{a}+\boldsymbol{c}$，如图 5.4(b)所示。

从 A 点作 AB'，要求 AB'的模等于$|\boldsymbol{b}|$，方向与 $\boldsymbol{b}$ 相反，即 $AB'=-\boldsymbol{b}$，则以 O 为起点、B' 为端点的矢量称为 $\boldsymbol{a}$ 与 $\boldsymbol{b}$ 的差 $\boldsymbol{a}-\boldsymbol{b}$，如图 5.4(c)所示。

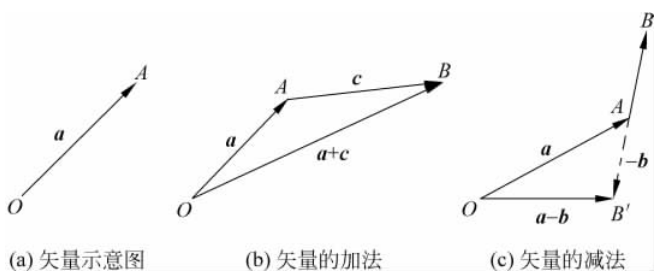

图 5.4　矢量加减法

5. 矢量的分解

定理：如果空间 3 个矢量 $\boldsymbol{a}$、$\boldsymbol{b}$、$\boldsymbol{c}$ 不共面，那么对任一矢量 $\boldsymbol{p}$，一定存在一个且仅一个有序实数组 x、y、z，使得：$\boldsymbol{p}=x\boldsymbol{a}+y\boldsymbol{b}+z\boldsymbol{c}$。含义与物理上的合力和力的分解一样。

6. 射影

已知向量 $AB=\boldsymbol{a}$ 和轴 1，$\boldsymbol{e}$ 是轴 1 上与轴 1 同方向的单位向量，作点 A 在 1 上的射影

A'，作点 B 在轴 1 上的射影 B'，则 $A'B'$ 叫做向量 AB 在轴 1 上或在 $\boldsymbol{e}$ 方向上的(正)射影，如图 5.5 所示。

可以证明：$A'B'=|AB|\cos<\boldsymbol{a},\boldsymbol{e}>$。

7. 矢量的数量积(点乘)

两个矢量的数量积是一个数，大小等于这两个矢量的模的乘积再乘以它们夹角的余弦。

$$\boldsymbol{a}\cdot\boldsymbol{b}=|\boldsymbol{a}||\boldsymbol{b}|\cos<\boldsymbol{a},\boldsymbol{b}>$$

用上面讲到的矢量的分解可以证明，数量积等于两个矢量的对应支量乘积之和。

$$\boldsymbol{a}\cdot\boldsymbol{b}=\boldsymbol{a}x\boldsymbol{b}x+\boldsymbol{a}y\boldsymbol{b}y+\boldsymbol{a}z\boldsymbol{b}z$$

数量积的性质：

(1) $\boldsymbol{a}\cdot\boldsymbol{e}=|\boldsymbol{a}||\boldsymbol{e}|\cos<\boldsymbol{a},\boldsymbol{e}>=|\boldsymbol{a}|\cos<\boldsymbol{a},\boldsymbol{e}>$。

(2) $\boldsymbol{a}\perp\boldsymbol{b}$ 等价于 $\boldsymbol{a}\cdot\boldsymbol{b}=0$，即 $\boldsymbol{a}x\boldsymbol{b}x+\boldsymbol{a}y\boldsymbol{b}y+\boldsymbol{a}z\boldsymbol{b}z=0$。

(3) 自乘：$|\boldsymbol{a}|^2=\boldsymbol{a}\cdot\boldsymbol{a}$。

(4) 结合律：$(\lambda\cdot\boldsymbol{a})\cdot\boldsymbol{b}=\lambda(\boldsymbol{a}\cdot\boldsymbol{b})$。

(5) 交换律：$\boldsymbol{a}\cdot\boldsymbol{b}=\boldsymbol{b}\cdot\boldsymbol{a}$。

(6) 分配律：$\boldsymbol{a}\cdot(\boldsymbol{b}+\boldsymbol{c})=\boldsymbol{a}\cdot\boldsymbol{b}+\boldsymbol{a}\cdot\boldsymbol{c}$。

8. 矢量的矢量积(叉乘、叉积)

(1) 矢量积的一般含义：两个矢量 $\boldsymbol{a}$ 和 $\boldsymbol{b}$ 的矢量积是一个矢量，记作 $\boldsymbol{a}\times\boldsymbol{b}$，其模等于由 $\boldsymbol{a}$ 和 $\boldsymbol{b}$ 做成的平行四边形的面积，方向与平行四边形所在平面垂直，当站在这个方向观察时，$\boldsymbol{a}$ 逆时针转过一个小于 π 的角到达 $\boldsymbol{b}$ 时的方向。

这个方向也可以用物理上的右手螺旋定则判断：右手 4 指弯向由 $\boldsymbol{a}$ 转到 $\boldsymbol{b}$ 的方向(转过的角小于 π)，大拇指指向的就是矢量积的方向，如图 5.6 所示。

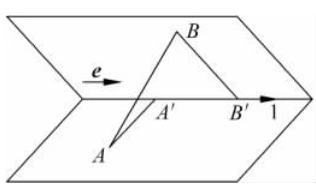

图 5.5　射影

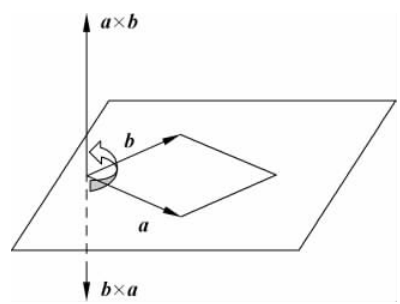

图 5.6　矢量积

(2) 叉积的等价且更有用的定义：把叉积定义为一个矩阵的行列式：

$$\boldsymbol{p}_1\times\boldsymbol{p}_2=\det\begin{bmatrix}x_1 & x_2\\ y_1 & y_2\end{bmatrix}=x_1y_2-x_2y_1$$

如图 5.7 所示，如果 $\boldsymbol{p}_1\times\boldsymbol{p}_2$ 为正数，则相对原点(0,0)来说，$\boldsymbol{p}_1$ 在 $\boldsymbol{p}_2$ 的顺时针方向；如果 $\boldsymbol{p}_1\times\boldsymbol{p}_2$ 为负数，则 $\boldsymbol{p}_1$ 在 $\boldsymbol{p}_2$ 的逆时针方向。如果 $\boldsymbol{p}_1\times\boldsymbol{p}_2$ 为 0，则 $\boldsymbol{p}_1$ 和 $\boldsymbol{p}_2$ 模相等且共线，方向相同或相反。也可以用矢量的分解证明：

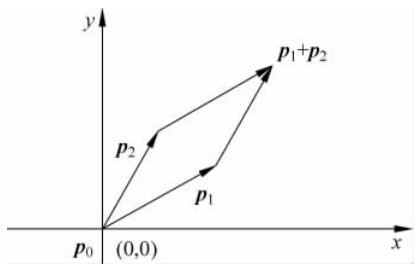

图 5.7 矢量乘积方向性

$$A \cdot B = \begin{vmatrix} i & j & k \\ Ax & Ay & Az \\ Bx & By & Bz \end{vmatrix} = |\ i\ | \begin{vmatrix} Ay & Az \\ By & Bz \end{vmatrix} - |\ j\ | \begin{vmatrix} Ax & Az \\ Bx & Bz \end{vmatrix} + |\ k\ | \begin{vmatrix} Ax & Ay \\ Bx & By \end{vmatrix}$$

即

$$(A \cdot B)x = AyBz - AzBy$$
$$(A \cdot B)y = AzBx - AxBz$$
$$(A \cdot B)z = AxBy - AyBx$$

注：i,j,k 分别为 x,y,z 方向上的单位矢量。

(3) 矢量的叉积对于计算几何有着重要的意义，是很多算法的核心。用叉积可以判断从一个矢量到另一个矢量的旋转方向，可以求同时垂直于两个矢量的直线(矢量)方向，还能用来计算面积……在计算几何中大有用武之地，下面的例子将有更详尽的说明。

用函数 multiply (P0,P1,P2)描述上述叉积的计算过程：

```
1.  /*************************************************************
2.  r = multiply(P0,P1,P2),得到(P1 - P0) * (P2 - P0)的叉积
3.  r>0: 则P0P1在P0P2的顺时针方向
4.  r = 0: 则P0P1与P0P2共线
5.  r<0: 则P0P1在P0P2的逆时针方向
6.   *************************************************************/
7.  double multiply(POINT P0,POINT P1,POINT P2)
8.  {
9.     return((P1.x - P0.x) * ( P2.y - P0.y) - ( P2.x - P0.x) * ( P1.y - P0.y));
10. }
```

矢量的旋转：实际应用中，经常需要从一个矢量转到另一个矢量，这个过程称为矢量的旋转，旋转的方向由叉积判定。很多例子都用到矢量的旋转这个方法和相应特性。

9. 坐标系中的矩形

若矩形边平行于坐标轴，如图 5.8(a)所示，此时给出 2 个点的坐标：$P_1(x_1,y_1)$，$P_2(x_2,y_2)$，则其他两点坐标为 $P_3(x_2,y_1)$和 $P_4(x_1,y_2)$。

若矩形边与坐标轴不平行，如图 5.8(b)所示，此时给出 3 点坐标：$P_1(x_1,y_1)$，$P_2(x_2,y_2)$，$P_3(x_3,y_3)$，则第 4 点坐标可以推得，利用切割法可以得到：

$$\begin{cases} x_4 + x_2 = x_1 + x_3 \\ y_4 + y_2 = y_1 + y_3 \end{cases} \Rightarrow \begin{cases} x_4 = x_1 + x_3 - x_2 \\ y_4 = y_1 + y_3 - y_2 \end{cases}$$

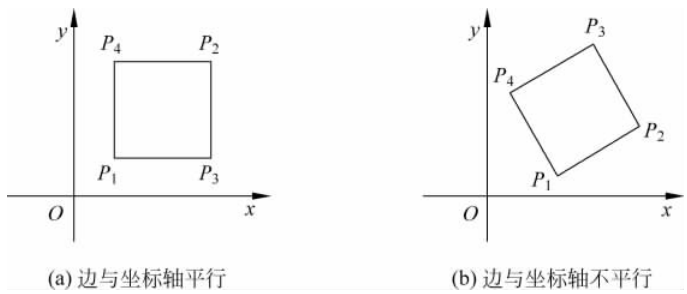

图 5.8　矢量旋转

另外，还有确定两条连续的有向线段$\overrightarrow{P_0P_1}$和$\overrightarrow{P_1P_2}$在 P_1 点是向左转还是向右转。

有了叉积的基础，无须通过对$\angle P_0P_1P_2$ 的计算来确定两条有向线段$\overrightarrow{P_0P_1}$和$\overrightarrow{P_1P_2}$在 P_1 点的转向，而仅需要添加一条辅助的有向线段$\overrightarrow{P_0P_2}$，并检查$\overrightarrow{P_0P_2}$是在有向线段$\overrightarrow{P_0P_1}$的顺时针方向还是逆时针方向。为了做到这一点，计算出叉积

$$(P_2 - P_0) * (P_1 - P_0) = (X_2 - X_0)(Y_1 - Y_0) - (X_1 - X_0)(Y_2 - Y_0)$$

若该叉积为正，则$\overrightarrow{P_0P_2}$在$\overrightarrow{P_0P_1}$的逆时针方向，即 P_1 点向左转，如图 5.9(a)所示。

若该叉积为负，则$\overrightarrow{P_0P_2}$在$\overrightarrow{P_0P_1}$的顺时针方向，即 P_1 点向右转，如图 5.9(b)所示。

若叉积为 0，说明 P_0 P_1 和 P_2 共线，如图 5.9(c)所示。

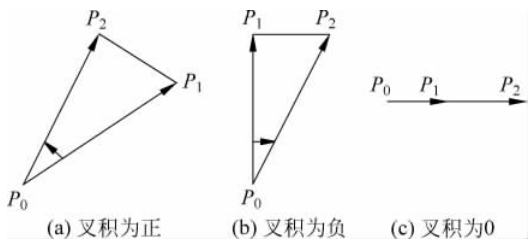

图 5.9　叉积计算结果

显然，可以通过调用函数 multiply(P_0,P_1,P_2)计算出叉积$(P_2-P_0)*(P_1-P_0)$，以确定两条连续的有向线段$\overrightarrow{P_0P_1}$和$\overrightarrow{P_1P_2}$在 P_1 点的转向。

例题 5.1

题意简述

任给 3 条直线的方程(斜率表示法，a_1,c_1,a_2,c_2,a_3,c_3)，求它们所围成的三角形的面积。

思路

设 3 条直线的方程分别为：$y=a_1*x+c_1$；$y=a_2*x+c_2$；$y=a_3*x+c_3$；首先要判断这 3 条直线是否能构成三角形，若能则求出 3 个交点，再由交点求出 3 条边的长度，再用海伦公式求出三角形的面积：$s=\text{sqrt}(p*(p-a)*(p-b)*(p-c))(p=(a+b+c)/2)$。

构成三角形的条件是：3 条直线的斜率互不相等，即 a_1、a_2、a_3 互不相等。

求两条直线的交点坐标，方法是求两个直线方程所列成的二元一次方程组的解，如：$y=a*x+b$，$y=c*x+d$，交点坐标为$((d-b)/(a-c),a*(d-b)/(a-c)+b)$。

由两个点的坐标求两点之间 r 距离只要用“两点间直线距离的公式”即可，如(a,b)、(c,d)两点间的距离为 $s=\mathrm{sqrt}(\mathrm{sqr}(a-c)+\mathrm{sqr}(b-d))$。

确定两条线段$\overrightarrow{P_1P_2}$和$\overrightarrow{P_3P_4}$是否相交。可分两步确定两条线段是否相交。

第一步：确定两条线段是否不相交——快速排斥试验，如图 5.10 所示。

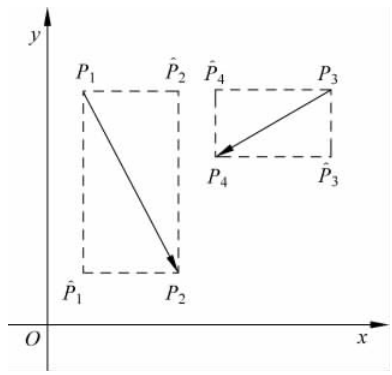

图 5.10　快速排斥试验

先通过下述方法求出两条线段的界限框（包含某线段且四边分别平行于 X 轴和 Y 轴的最小矩形）。

线段$\overrightarrow{P_1P_2}$的界限框用矩形$(\hat{P}_1,\hat{P}_2)$表示，其左下角的点为$\hat{P}_1=(\min(X_1,X_2),\min(Y_1,Y_2))$，右上角的点为$\hat{P}_2=(\max(X_1,X_2),\max(Y_1,Y_2))$。

线段$\overrightarrow{P_3P_4}$的界限框用矩形$(\hat{P}_3,\hat{P}_4)$表示，其左下角的点为$\hat{P}_3=(\min(X_3,X_4),\min(Y_3,Y_4))$，右上角的点为$\hat{P}_4=(\max(X_3,X_4),\max(Y_3,Y_4))$。

很显然，当$\overrightarrow{P_1P_2}$界限框右上角$\hat{P}_2$ 的 X 坐标$(\max(X_1, X_2))\geqslant\overrightarrow{P_3P_4}$界限框左下角$\hat{P}_3$ 的 X 坐标$(\min(X_3, X_4))$，并且$\overrightarrow{P_3P_4}$界限框右上角$\hat{P}_4$ 的 X 坐标 $(\max(X_3, X_4))\geqslant\overrightarrow{P_1P_2}$界限框左下角$\hat{P}_1$ 的 X 坐标$(\min(X_1, X_2))$，则两个矩形在 X 方向相交；同样，当$\overrightarrow{P_1P_2}$界限框右上角$\hat{P}_2$ 的 Y 坐标$(\max(Y_1, Y_2))\geqslant\overrightarrow{P_3P_4}$界限框左下角$\hat{P}_3$ 的 Y 坐标$(\min(Y_3, Y_4))$，并且$\overrightarrow{P_3P_4}$界限框右上角$\hat{P}_4$ 的 Y 坐标$(\max(Y_3, Y_4))\geqslant\overrightarrow{P_1P_2}$界限框左下角$\hat{P}_1$ 的 Y 坐标$(\min(Y_1, Y_2))$，则两个矩形在 Y 方向相交。

如果两条线段的界限框不能同时在 X 方向和 Y 方向相交，则这两条线段是不可能相交的。因此通过快速排斥试验，可以确定两条线段是否可能相交。

在两个矩形同时将在两个方向相交的前提下，做第二步：确定一条线段$\overrightarrow{P_1P_2}$（或$\overrightarrow{P_3P_4}$）是否跨立另一条线段$\overrightarrow{P_3P_4}$或（$\overrightarrow{P_1P_2}$）所在的直线——跨立实验。

所谓跨立是指某线段的两个端点是否分别处于另一线段所在直线的两旁，或者是否有其中一个端点处于另一线段所在的直线上。可以运用叉积的方法来确定。其设计思想是，从 P_1 出发向另一线段的两个端点 P_3 和 P_4 引出两条辅助线段$\overrightarrow{P_1P_3}$、$\overrightarrow{P_1P_4}$。然后计算两个叉积：

由$(P_3-P_1)*(P_2-P_1)$确定$\overrightarrow{P_1P_3}$在$\overrightarrow{P_1P_2}$的哪个方向；由$(P_4-P_1)*(P_2-P_1)$ 确定$\overrightarrow{P_1P_4}$在$\overrightarrow{P_1P_2}$的哪个方向。

若两个叉积的正负号相同，说明$\overrightarrow{P_1P_3}$和$\overrightarrow{P_1P_4}$同在$\overrightarrow{P_1P_2}$的一边，即$\overrightarrow{P_3P_4}$不能跨立$\overrightarrow{P_1P_2}$所在的直线，如图 5.11(b)所示。

若两个叉积的正负号相反，说明$\overrightarrow{P_1P_3}$和$\overrightarrow{P_1P_4}$同在$\overrightarrow{P_1P_2}$的两边，即$\overrightarrow{P_3P_4}$跨立$\overrightarrow{P_1P_2}$所在

的直线，如图 5.11(a)所示。

若任何一个叉积为 0，由于$\overrightarrow{P_1P_2}$和$\overrightarrow{P_3P_4}$已通过了快速排斥试验，因此 P_3 和 P_4 两点中必有一点处于线段$\overrightarrow{P_1P_2}$所在的直线上，如图 5.11(c)所示。

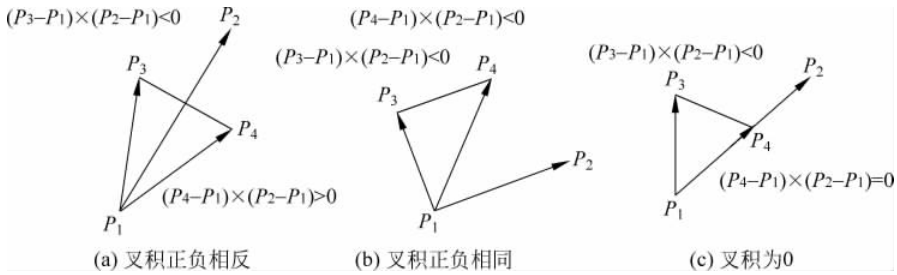

图 5.11 判断相交

显然两条线段相交当且仅当它们能够通过快速排斥试验并且每一条线段能够跨立另一条线段所在的直线。

代码

```
1.  //如果线段 u 和 v 相交(包括相交在端点处)时,返回 true
2.  bool intersect(LINESEG u,LINESEG v)
3.  {
4.      return( (max(u.s.x,u.e.x)>=min(v.s.x,v.e.x))&&        //排斥试验
5.      (max(v.s.x,v.e.x)>=min(u.s.x,u.e.x))&&
6.      (max(u.s.y,u.e.y)>=min(v.s.y,v.e.y))&&
7.      (max(v.s.y,v.e.y)>=min(u.s.y,u.e.y))&&
8.      (multiply(v.s,u.e,u.s)*multiply(u.e,v.e,u.s)>=0)&&  //跨立试验
9.      (multiply(u.s,v.e,v.s)*multiply(v.e,u.e,v.s)>=0));
10. }
```

例题 5.2 Intersecting Lines(POJ 1269)。

题意简述

在平面直角坐标系中，两点确定一条直线，题目给出 4 个点，前面 2 点和后面 2 点一共确定两条直线，求出这两条直线的交点。

思路

初步来看题目是比较简单的，根据两点能够求出直线方程，然后通过求解二元一次方程组求出交点。思路比较清晰，但是考虑得不够全面：首先根据两点确定的直线不一定存在斜率，因此直线方程应选择一般式，即 $ax+by+c=0$，这样比较严谨；另一方面，得到的两条直线同样存在很多情况，比如两条直线存在平行、相交、重合 3 种情况，因而必须分别考虑(当然判断是比较简单的)。

代码

```
1.  /* Accepted 176K 0MS C++ */
2.  #include<iostream>
```

```
3.  using namespace std;
4.  typedef struct
5.  {
6.     int x,y;
7.  } Point;
8.  typedef struct
9.  {
10.     int a,b,c;
11. } Line;
12. int line(int x1,int y1,int x2,int y2)                    //向量(x1,y1),(x2,y2)
13. {
14.     return x1 * y2 - y1 * x2;
15.     //结果为 0 时,向量平行
16. }
17. Line lineform(int x1,int y1,int x2,int y2)               //ax + by + c = 0;
18. {
19.     Line temp;
20.     temp.a = y2 - y1;
21.     temp.b = x1 - x2;
22.     temp.c = x2 * y1 - x1 * y2;
23.     return temp;
24. }
25. double x,y;
26. void lineintersect(Line l1,Line l2)                      //求两条直线的交点
27. {
28.     double d = l1.a * l2.b - l2.a * l1.b;
29.     x = (l2.c * l1.b - l1.c * l2.b) / d;
30.     y = (l2.a * l1.c - l1.a * l2.c) / d;
31. }
32. int main()
33. {
34.     int n;
35.     Point p1,p2,p3,p4;
36.     Line l1,l2;
37.     scanf("%d",&n);
38.     printf("INTERSECTING LINES OUTPUT\n");
39.     while(n--)
40.     {
41.         scanf("%d%d%d%d%d%d%d%d",&p1.x,&p1.y,&p2.x,&p2.y,&p3.x,&p3.y,&p4.x,
            &p4.y);
42.        if(line(p2.x-p1.x,p2.y-p1.y,p3.x-p1.x,p3.y-p1.y)==0&&line(p2.x-p1.x,p2.y
           -p1.y,p4.x-p1.x,p4.y-p1.y)==0)
43.             printf("LINE\n");
44.         else if(line(p2.x-p1.x,p2.y-p1.y,p4.x-p3.x,p4.y-p3.y)==0)
45.             printf("NONE\n");
46.         else {
47.             l1 = lineform(p1.x,p1.y,p2.x,p2.y);
48.             l2 = lineform(p3.x,p3.y,p4.x,p4.y);
49.             lineintersect(l1,l2);
50.             printf("POINT %.2lf %.2lf\n",x,y);
51.         }
```

```
52.     }
53.     printf("END OF OUTPUT\n");
54.     return 0;
55. }
```

5.2 确定任意一对线段是否相交

专题讲解

下面将讨论关于在一组线段中确定其中任意两条线段是否相交的问题。该算法的具体操作是，首先输入由若干条线段组成的集合，若这组线段中任意两条线段相交，则返回布尔标志 true；否则返回 false。

算法使用了一种称为“扫除”的技术，即假设一条垂直扫除线沿 X 轴方向从左到右顺序移动，穿过已知的若干线段。移动过程中，每遇到一个线段端点，它就将穿过扫除线的所有线段放入一个动态数据结构中，并利用它们之间的关系进行排序，核查是否有相交点存在。

为了使问题简化，算法做了两点假设：

(1) 没有输入线段是垂直的。

(2) 没有 3 条输入线段相交于同一点的情形。

由于不存在垂直的输入线段，所以任何与给定的垂直扫除线相交的输入线段与其只能有一个交点。可以将此时与垂直扫除线相交的线段其交点的 Y 坐标值单调递增的顺序放在一个序列 T 中。当线段的左端点遇到扫除线时，线段就进入序列 T，当其右端点遇到扫除线时使随之离开序列 T。

具体点讲，考查两条不相交线段 S_1 和 S_2。如果垂直扫除线平移至 X 位置时与这两条线段相交，则说这些线段在 X 是可比的。如果 S_1 和 S_2 在 X 可比并且 S_1 与在 X 的扫除线的交点比 S_2 在同一条扫除线的交点高，则说在 X 处 S_1 位于 S_2 之上，写作 S_1：>：S_2，反映在此时的 T 序列中，S_1 位于 S_2 前。

例如在图 5.12 中，有下述关系：

$a>r_c$，即在 r 处的 T 序列为$\{a, b, c\}$。

$a>t_b$，$b>t_c$，$a>t_c$，即在 t 处的 T 序列为$\{a, b, c\}$。

$b>u_c$，即在 u 处的 T 序列为$\{b,c\}$。

线段 d 与其他任何线段都不可比。

随着垂直扫除由左至右平移，T 序列中线段的数量和相对位置将由于下述原因而发生变化。

(1) 扫除线遇到某线段的左端点时，该线段进入 T 序列；遇到某线段的右端点时，该线段离开 T 序列；

(2) 扫除线穿过两线段交点后的某区域(即在该区域内的 T 序列中，这两条线段是相邻的)，它们在 T 序列的位置将对换。

例如在图 5.13 中，扫除线 V 和 W 分别位于线段 e 和 f 的交点的左边和右边。在交点的左边 r 时，$e>r_f$ 序列为$\{e,f\}$；而在交点右边的 w 时，$f>w_e$ 序列则变为$\{f, e\}$。由于没

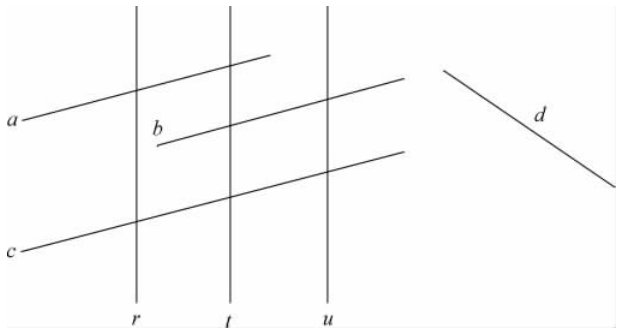

图 5.12 示例

有第三条线段相交于同一点，所以在交点附近必有一个连续区域（如图 5.13 中的阴影区域），使得垂直扫除线在该区域活动时，相交线段 e 和 f 在序列 T 中是连续的。

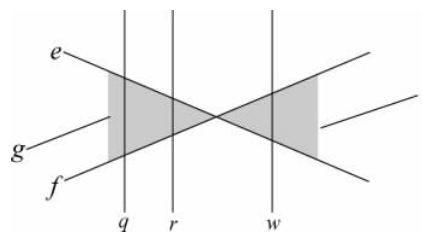

图 5.13 示例

由此可见，对于典型的扫除算法，要安排下列两个数据集合：

（1）由序列给出与扫除线相交的线段之间的关系，即线段按相交点 Y 坐标递增的顺序排列；

（2）由左至右定义扫除线的一系列暂停位置，称每个这样的暂停位置为一个事件点。一般来说，事件点为每条线段的端点。当遇到线段左端点时，就把线段按排序要求插入 T 序列中；当遇到线段右端点时，就把它从 T 序列中删除；每当两条线段在 T 序列中第一次变为相邻时，就检查它们是否相交。

用一个称为 any_sagment_intersect 的函数来描述算法流程。函数输入由 n 条线段组成的集合 S。如果 S 中任何一对线段相交.算法就返回布尔值 true；否则返回布尔值 false。由于在“扫除”过程中，T 序列需按排序要求频繁地进行插入或删除，因此将它放入一个动态的指针链表中。

```
1.  bool any_sagment_intersect (s)
2.  {
3.  T←ϕ;
4.     对集合 S 中的线段端点从左向右进行排序.对 X 坐标相同的端点,把 Y 坐标较小的放在前面;
5.  Foreach (p in 对端点排序表) {
6.     if (p 是线段 S 的左端点) {
7.           按排序要求将 S 线段插入序列 T;
8.           if ((s 在 T 中有左邻元素且接线段与 S 相交) ||(s 在 T 中有右邻元素且该线段与 s 相交))
9.              return true;
10.       }
11.       if ( p 是线段 S 的右端点 )
12.           if (S 在 T 中有左邻元素和右邻元素且两线段相交) {
13.              将线段 S 从序列 T 中删除;
14.              return true;
15.           }
```

```
16.        }
17.        return false;
18.  }
```

5.3 线段合并

专题讲解

关于线线相交，线与多边形相交等，只是知道线段相交的判断、求交点等是不够的。但是这是基础，靠模板就可以了，唯一要注意的是明白模板的输入、输出，确保模板的正确性。模板若是自己写的，要考虑到方方面面，先不求简练，而求正确。

当然，题目不会仅仅是简单的判断相交和求交点，往往需要综合运用“离散化”、“条件转化”等，使得对同一个问题的解决方式可以有很大的简化，并能减少相应的精度损失（根据经验，因为精度的问题不是很大。在采取方案时，能用 int 型不要用 double 型，尤其是不涉及求交点的一些题目，或是整点数目很多的情况）。

另外，单纯涉及线段的，往往要考虑重点、平行、重合、端点相交、重合线段的合并等方面，这些往往也是解题的关键。

这里给一段都是平行线段的合并代码，并且它们事先已经对 x 坐标进行过排序。s、e 分别为分线段的起始点。

```
1.  final[0] = shadow[0];
2.  fn = 0;
3.  for (int i = 1; i < sn; ++i) {
4.     if (final[fn].e.x < shadow[i].s.x) {
5.        final[++fn] = shadow[i];
6.     } else {
7.        final[fn].s.x = min(final[fn].s.x, shadow[i].s.x);
8.        final[fn].e.x = max(final[fn].e.x, shadow[i].e.x);
9.     }
10. }
```

如果对于没有排过序的线段来说，用一个 $O(n^2)$ 的时间复杂度可以对重合的线段进行合并。

```
1.  w.clear();
2.  for (int i = 0; i < n; ++i) {
3.     for (int j = 0; j <= (int)w.size(); ++j) {
4.        if (j == (int)w.size()) {
5.           w.push_back(v[i]);
6.           break;
7.        } else if (isoverlap(w[j], v[i])) {
8.           w[j] = overlap(w[j], v[i]);
9.           break;
```

```
10.         }
11.       }
12.  }
13.  v = w;
```

不只是线段，只要遇到计算几何的题目，小心总是不会错的。应尽可能地考虑到多种情况（特别是退化情况），处理细节要注意变量、索引等，因为代码量一大，不容易查错，一旦出错，找出错误就比较难了。

例题 5.3　Line of Sight（POJ 2074）。

题意简述

有一幢房子和一条马路，房子和马路之间有一些障碍物，房子、马路、障碍物都可以视作平行于 X 轴的线段，现在要求马路上最长的一段，这一段上所有的点都能看到房子，请输出这个长度。

思路

从反面着手，找出对于每一个线段障碍物，马路上看不见的阴影范围。将重叠的合并，最终将得到一些不相交的阴影线段，然后再依次寻找看得见的最长的部分。

代码

```
1.   /* Accepted 204K 0MS C++ */
2.   #include <cstdio>
3.   #include <cstring>
4.   #include <cmath>
5.   #include <climits>
6.   #include <vector>
7.   #include <algorithm>
8.   using namespace std;
9.
10.  #define EPS 1e-8
11.  #define MAXN 1000
12.  #define PI acos(-1.0)
13.
14.  int sgn(double d)
15.  {
16.      if (fabs(d) < EPS) return 0;
17.      return d > 0 ? 1 : -1;
18.  }
19.
20.  struct point {
21.      double x, y;
22.  };
23.
24.  struct seg {
25.      point s, e;
26.  };
27.
28.  double det(double x1, double y1, double x2, double y2)
```

```
{
    return x1 * y2 - x2 * y1;
}

double side(point a, point b, point p)
{
    return det(b.x-a.x, b.y-a.y, p.x-a.x, p.y-a.y);
}

point cross_point(point a, point b, point c, point d) {
   point its;
   double u = side(a, b, c), v = side(b, a, d);
   its.x = (c.x * v + d.x * u) / (u + v);
   its.y = (c.y * v + d.y * u) / (u + v);
   return its;
}

bool cmp(seg a, seg b)
{
    if (sgn(a.s.x - b.s.x) == 0) return a.e.x < b.e.x;
    return a.s.x < b.s.x;
}

seg h, p, ob[MAXN];
seg shadow[MAXN];
seg final[MAXN];
int sn, fn;

int work(int n, double &ans)
{
    seg unsee;
    sn = 0;
    for (int i = 0; i < n; ++i) {
        unsee.s = cross_point(h.e, ob[i].s, p.s, p.e);
        unsee.e = cross_point(h.s, ob[i].e, p.s, p.e);
        if (unsee.e.x < p.s.x) continue;
        if (unsee.s.x > p.e.x) continue;
        if (unsee.s.x < p.s.x) { unsee.s.x = p.s.x;    }
        if (unsee.e.x > p.e.x) { unsee.e.x = p.e.x; }
        shadow[sn++] = unsee;
    }

    //no projection
    if (sn == 0) {
        ans = p.e.x - p.s.x;
        return true;
    }

    sort(shadow, shadow + sn, cmp);

    final[0] = shadow[0];
```

```
80.       fn = 0;
81.       for (int i = 1; i < sn; ++ i) {
82.           if (final[fn].e.x < shadow[i].s.x) {
83.               final[ ++ fn] = shadow[i];
84.           } else {
85.               final[fn].s.x = min(final[fn].s.x, shadow[i].s.x);
86.               final[fn].e.x = max(final[fn].e.x, shadow[i].e.x);
87.           }
88.       }
89.
90.       ans = max(final[0].s.x - p.s.x, p.e.x - final[fn].e.x);
91.       for (int i = 0; i < fn; ++ i) {
92.           ans = max(ans, final[i + 1].s.x - final[i].e.x);
93.       }
94.
95.       return sgn(ans) > 0;
96.   }
97.
98.   int main()
99.   {
100.      int n;
101.      double x1, x2, y, res;
102.
103.      while (scanf("%lf%lf%lf", &x1, &x2, &y)) {
104.          if (!sgn(x1) && !sgn(x2) && !sgn(y)) break;
105.
106.          h.s.y = h.e.y = y;
107.          h.s.x = x1, h.e.x = x2;
108.
109.          scanf("%lf%lf%lf", &x1, &x2, &y);
110.          p.s.y = p.e.y = y;
111.          p.s.x = x1, p.e.x = x2;
112.          int on = 0;
113.          scanf("%d", &n);
114.          for (int i = 0; i < n; ++ i) {
115.              scanf("%lf%lf%lf", &x1, &x2, &y);
116.              if (y > h.s.y || y < p.s.y) continue;
117.
118.              ob[on].s.x = x1, ob[on].e.x = x2;
119.              ob[on].s.y = y, ob[on].e.y = y;
120.              on ++ ;
121.          }
122.
123.          if (work(on, res)) {
124.              printf("%.2f\n", res);
125.          } else {
126.              printf("No View\n");
127.          }
128.      }
129.      return 0;
130.  }
```

5.4 凸包

专题讲解

有一个点集 $Q=\{P_0,\cdots,P_{n-1}\}$，它的凸包 ch(q)是一个最小的凸多边形 P，且满足 Q 中的每个点或者在 P 的边界上，或者在 P 的内部。在直观上，可以把 Q 中的每个点看作露在板外的铁钉，那么凸包就是包含所有铁钉的一个拉紧的橡皮绳所构成的形状，如图 5.14 所示。

在计算一组点的凸包的基础上.可以引出一些计算几何学问题的算法。例如考虑二维的最远点对问题：已知平面上一组有 n 个点，并希望找出彼此间最远的两个点。很显然，这两个点必定是凸包的顶点。运用计算凸包的算法求最远点对问题，可使算法效率提高不少。

下面将阐述计算凸包的算法 CH(Q)，按逆时针方向输出凸包的顶点。

1. Graham 扫描法

首先必须明确，在点集 Q 处于最低位置(Y 坐标值最小)的一个点 P_0 是凸包 ch(q)的一个顶点。如果这样的顶点有多个，则选取最左边的点为 P_0，从 P_0 出发，按逆时针方向寻找凸包的顶点。寻找过程中，除 P_0 外的每个点都要被扫描一次，其顺序是依据各点在逆时针方向上相对 P_0 的极角的递增次序。极角的大小可以通过计算叉积

$$(P_i-P_0)*(P_j-P_0)$$

来确定：

若叉积>0，则说明相对 P_0 来说，P_j 的极角大于 P_i 的极角，P_i 先于 P_j 被扫描。

若叉积<0，则说明 P_j 的极角小于 P_i 的极角，P_j 先于 P_i 被扫描。

若叉积$=0$，则说明两个极角相等。在这种情况下，由于 P_i 和 P_j 中距离 P_0 较近的点不可能是凸包的顶点，因此只需扫描其中一个与 P_0 距离较远的点。

例如图 5.15 说明了图 5.14 小的点按相对于 P_0 的极角进行排序后得到的扫描序列$<P_1、P_2、P_3>$。

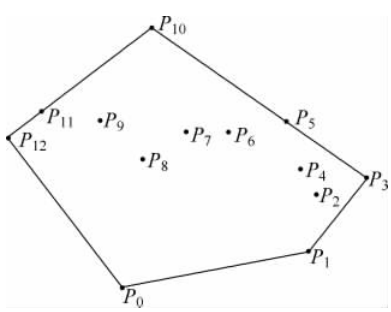

图 5.14 凸包

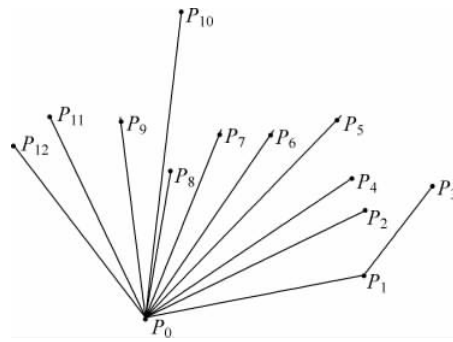

图 5.15 极角排序

接下来的问题是如何确定当前被扫描的点 $P_i(1\leqslant i\leqslant n-1)$是凸包上的点呢？设置一个关于候选点的堆栈 S 来解决凸包问题。初始时 P_0 和排序后的 P_1、P_2 作为初始凸包相继入栈。扫描过程中，Q 集合中的其他点都被推入堆栈一次，而不是凸包 ch(q)顶点的点最终

将弹出堆栈。当算法结束时，堆栈 S 中仅包含 ch(q)的顶点，其顺序为各点在边界上出现的逆时针方向排列的顺序。

由于是沿逆时针方向通过凸包的，因此，如果栈顶元素是凸包的顶点，则它应该向左转指向当前被扫描的点 P_i。如果它不是向左转，则它不属于凸包中的顶点，应从堆栈 S 中移出。在弹出了所有非左转的顶点后，就把 P_i 推入堆栈 S，继续扫描序列中的下一个点 P_{i+1}。

判断当前栈顶元素是否向左转指向扫描点 P_j，只需计算下述叉积的值。

$(P_i - P_{top-1}) * (P_{top} - P_{top-1})$ （其中 top 为栈顶指针，P_{top-1} 为次栈顶元素）

若叉积大于等于 0，说明线段$\overline{P_{top-1}P_i}$在$\overline{P_{top-1}P_{top}}$的顺时针方向或共线，$P_{top}$未向左转。在依次扫描了 $P_3..P_{n-1}$后，堆栈 S 从底 P 到顶部依次是按逆时针方向排列的 ch(q)中的顶点。

算法如下：

```
1.  void graham_scan (q ) {
2.       求 q 中 Y 坐标值最小的点 P0.若具有 Y 坐标最小值的顶点有多个,则取最左边的点为 P0;
3.       对 q 的剩余点按逆时针相对 P0 的极角进行排序.若有数个极角相同的点,则保留其中一个
         与 P0 距离最远的点,得出序列<P1, P2, …, Pn-1>;
4.      P0, P1, P2 相继入栈 S;
5.      for( i = 3 ; i < n ; i ++ ) {
6.           while (由次栈顶元素、栈顶元素和 Pi 形成角不是向左转) {
7.                栈顶元素出栈 S;
8.                Pi 入栈 S;
9.           }
10.     }
11.     打印按逆时针方向排列的 ch(q)中的顶点 S[1..top];
12.     return;
13. }
```

例如 graham_scan(q)输入点集 q 后，其执行过程如图 5.16 所示：

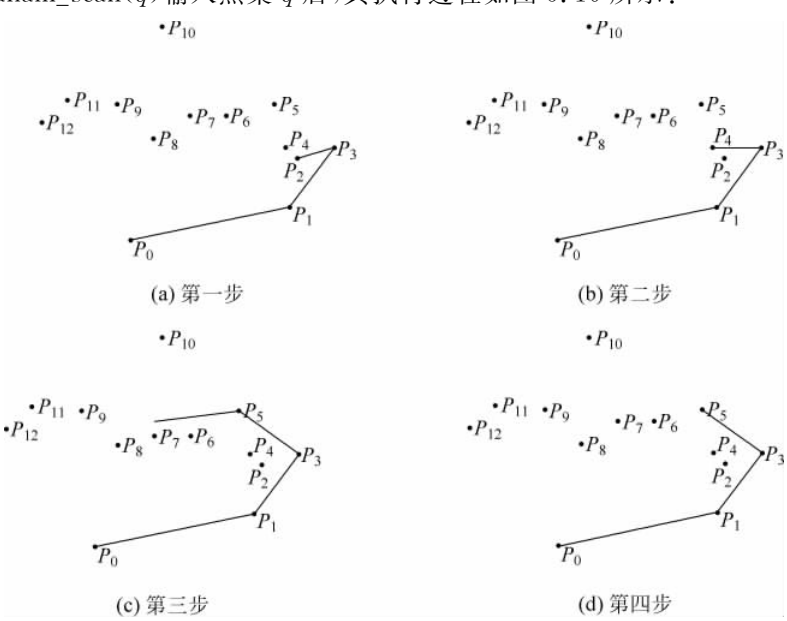

图 5.16 Graham 扫描法示例

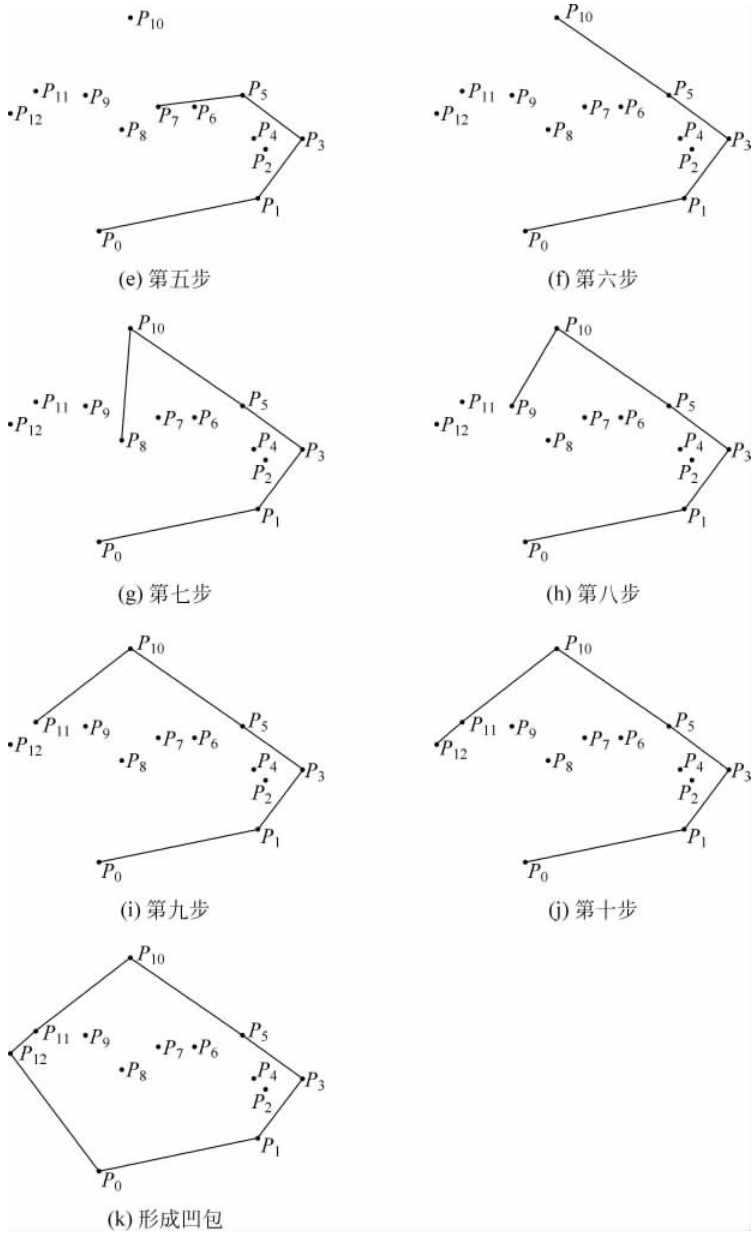

图 5.16（续）

一些试题的算法就是按上述求解凸包的思路构思的。

另外，还有以下几种凸包算法的思想。

1）Jarris 步进法

Jarris 步进法运用了一种称为打包的技术来计算一个点的集合 Q 的凸包 ch(q)。形象地看，该算法模拟在集合 Q 外紧紧地包了一层纸。开始时，把纸的末端粘在集合中最低的点上，即粘在与 Graham 扫描法开始相交的点 P_0 上。该点为凸包的一个顶点。然后把纸拉向右边使其绷紧，再将纸拉高一些直至碰到一个点。该点也必定是凸包的一个顶点。使纸

保持绷紧状态，用这种方法继续围绕顶点集合直至回到原始点 P_0。

2）增量式算法

逐次再点加入，然后检查之前的点是否在新的凸包上。由于每次都要检查所有之前的点，时间复杂度为 $O(n^2)$。

3）葛立恒扫描法

由最底的一点 A_1 开始，计算它跟其他各点的连线和 x 轴的角度，按从小至大将这些角度排序，称它们的对应点为 $A_2, A_3, \cdots, A_n$。这里的时间复杂度可达 $O(n\log n)$。

考虑最小的角度对应的点 A_3。若由 A_2 到 A_3 的路径相对 A_1 到 A_2 的路径是向右转的（可以想象一个人沿 A_1 走到 A_2，他站在 A_2 时，是向哪边改变方向），表示 A_3 不可能是凸包上的一点，考虑下一点由 A_2 到 A_4 的路径；否则就考虑 A_3 到 A_4 的路径是否向右转……直到回到 A_1。

这个算法的整体时间复杂度是 $O(n \log n)$，注意每点只会被考虑一次，而不像 Jarvis 步进法中会考虑多次。

这个算法由葛立恒在 1972 年发明。它的缺点是不能推广到二维以上的情况。

4）单调链

将点按 x 坐标的值排列，再按 y 坐标的值排列。

选择 x 坐标为最小值的点，在这些点中找出 y 坐标的值最大和 y 坐标的值最小的点。对于 x 坐标为最大值也是这样处理。将两组点中 y 坐标值较小的点连起。在这条线段下的点，找出它们之中 y 坐标值最大的点，又在它们之间找 x 坐标值再最小和最大的点……以此类推。

时间复杂度是 $O(n \log n)$。

分治法

将点集 X 分成两个不相交子集。求得两者的凸包后，计算这两个凸包的凸包，该凸包就是 X 的凸包。时间复杂度是 $O(n \log n)$。

5）快包法（Akl-Toussaint 启发式）

选择最左、最右、最上、最下的点，它们必组成一个凸四边形（或三角形）。这个四边形内的点必定不在凸包上。然后将其余的点按最接近的边分成四部分，再进行快包法（QuickHull）。

6）MELKMAN 方法

可以在线性时间 $O(n)$ 内求得凸包。其最大特点在于采用了双端队列，理解为两头都可以进出的栈。如果多边形是按顺序（顺时针或逆时针）输入的。只要在栈顶维护右手系，栈底维护左手系，经过一线性时间的处理，便得到了最终的凸包。

算法步骤如下：

（1）初始的 3 个点组成的三角形即凸包。此时栈里为 3－1－2－3，注意共线的情况。

（2）要保证栈顶右手，栈底左手，即为此要交换 1、2（如果需要的话）。

（3）对于下一个要考虑的点，如果该点在凸包内部，则无须考虑，否则维护两头栈。

While 当前点在栈顶两点所形成直线的右侧：删除栈顶的点。

While 当前点在栈底两点所形成直线的左侧：删除栈底的点。

在栈顶栈底分别加入当前点：

```
1.  //按顺时针或逆时针入栈
2.  void Melkman(int n, int &head, int &tail)
3.  {
4.      int i;
5.      head = tail = n;
6.      ch[tail++] = p[0];
7.      for (i = 1; i < n; ++i) {
8.          ch[tail] = p[i];
9.          if (side(ch[head], ch[head + 1], p[i + 1])) break;
10.         //first 3 points not in a line
11.     }
12.     if (n < 3) return;
13.     ch[++tail] = ch[--head] = p[++i];
14.     if (side(ch[n], ch[n + 1], ch[n + 2]) < 0) swap(ch[n], ch[n + 1]);
15.     for (++i; i < n; ++i) {
16.         if (side(ch[head + 1], ch[head], p[i]) < 0 &&
17.            side(ch[tail - 1], ch[tail], p[i]) > 0) continue;
18.         while (tail - head > 1 && side(ch[head + 1], ch[head], p[i]) >= 0) ++head;
19.         ch[--head] = p[i];
20.         while (tail - head > 1 && side(ch[tail - 1], ch[tail], p[i]) <= 0) --tail;
21.         ch[++tail] = p[i];
22.     }
23. }
```

例题 5.4　cows(POJ 3348)。

题意简述

有一些树,坐标已知,要用这些树作为篱笆的顶点圈一块多边形的地养牛,每头牛占地大小已知,问最多养多少头牛。

思路

本题实质上是求一个用这些树做篱笆桩所围成的多边行的最大面积,是一个简单的凸包问题,直接找出凸包,求出凸多边形面积,然后就基本完成。

代码

```
1.  /* Accepted 220K 0MS C++ */
2.  #include<iostream>
3.  #include<stack>
4.  #include<algorithm>
5.  using namespace std;
6.  #define ml 100005
7.  struct node {
8.     int x,y;
9.  };
10. node a[ml],b[ml];
11. stack<node> st;
12. int direction(node aa,node bb,node cc)
13. {
```

```
14.     int x1 = bb.x - aa.x , y1 = bb.y - aa.y , x2 = cc.x - aa.x , y2 = cc.y - aa.y;
15.     return x1 * y2 - x2 * y1;
16. }
17. bool cmp1(node aa,node bb)
18. {
19.     if(aa.y == bb.y) return aa.x < bb.x;
20.     return aa.y < bb.y;
21. }
22. int  cmp2(node aa,node bb)
23. {
24.     int d = direction(a[1],aa,bb);
25.     if(d > 0) return 1;
26.     return 0;
27. }
28. node next_to_top()
29. {
30.     node tn1,tn2;
31.     tn1 = st.top(),st.pop();
32.     tn2 = st.top();
33.     st.push(tn1);
34.     return tn2;
35. }
36. node st_top()
37. {
38.     return st.top();
39. }
40. int main()
41. {
42.     int i,k,n,x,y;
43.     node tn1,tn2;
44.     while(scanf("%d",&n)!= EOF) {
45.         for(i = 1; i <= n; i++)
46.             scanf("%d%d",&x,&y),a[i].x = x,a[i].y = y;
47.         if( n < 3 ) {
48.             printf("0\n");
49.             continue;
50.         }
51.         sort(a + 1,a + 1 + n,cmp1);
52.         sort(a + 2,a + 1 + n,cmp2);
53.         b[1] = a[1] , b[2] = a[2];
54.         for(k = 2 , i = 3; i <= n; i++) {
55.             if(direction(b[1],b[k],a[i]) == 0) {
56.                 if(abs( a[i].x - b[1].x )>= abs( b[k].x - b[1].x ) && abs(a[i].y - b[1].
                    y) >= abs(b[k].y - b[1].y) )
57.                     b[k] = a[i];
58.             }
59.             else
60.                 b[ ++k] = a[i];
61.         }
62.         while(!st.empty())
63.             st.pop();
```

```
64.          st.push(b[1]);
65.          st.push(b[2]);
66.          st.push(b[3]);
67.          for(i = 4; i <= k; i++) {
68.              while(1) {
69.                  tn1 = next_to_top();
70.                  tn2 = st_top();
71.                  int d = direction(tn1,b[i],tn2);
72.                  if (d < 0)
73.                      break;
74.                  st.pop();
75.              }
76.              st.push(b[i]);
77.          }
78.          k = st.size();
79.          for(i = k; i >= 1; i -- ) {
80.              b[i] = st.top();
81.              st.pop();
82.          }
83.          double sum = 0;
84.          for(i = 3; i <= k; i++)
85.              sum = sum + abs( direction(b[1],b[i],b[i - 1]) );
86.          printf(" % d\n",(int)(sum/100));
87.      }
88.      return 0;
89. }
```

5.5 寻找最近点对

专题讲解

通常将平面上两点 $P_i=(X_i,Y_i)$和 $P_j=(X_j,Y_j)$之间的距离称为欧氏距离。

$$d_{i,j}=\sqrt{(X_i-X_j)^2+(Y_i-Y_j)^2}$$

如果 P_i 和 P_j 重合，这种情况下它们之间的距离 d_{ij} 为 0。

现在考虑在 $n>2$ 个点的集合 Q 中寻找最近点对的问题。这个问题具有很大的实用价值，例如对于一个控制空中或海上交通的系统来说，可能需要了解两个最近的交通工具，以检测可能产生的相撞事故。

显然，n 个点的集合 Q 中共有 $c(n,2)$个点对。最原始的算法是通过计算所有这些点对的距离来找出最近点对。这种算法的效率直接依赖于 n 值，n 愈大，运行时间愈长。为了提高时效，将描述一种应用于该问题的分治算法。

算法每次递归调用的输入为点的子集 $P=(Pr..P_R)\subseteq Q$ 和数组 Y。P 中的所有点按其 Y 坐标单调递增的顺序排列，其编号序列存入数组 Y。同样，也得将所有点按 X 坐标单调递增的顺序排列，其编号序列存入数组 X。显然这种排序性质在每次递归调用中是保持不变的，因此只要在调用分治程序前进行一次预排序，分别求出数组 X 和数组 Y。

输入为 P、Y 的递归调用首先检查 $|P|<3$。如果是这样，则计算所有 $c(|P|,2)$ 个点对的距离并返回最近的点对及其距离。如果 $|P|>3$，则递归调用采取如下分治策略。

1. 划分

找出一条垂直平分线，将点集 P 划分为点数分别为 $P/2$（P 为偶数）或者 $[P/2]$ 和 $[P/2]+1$（P 为奇数）的两个点集 P_1 和 P_R。其中 P_1 上的所有点在垂直平分线上或其左侧；P_R 上的所有点在垂直平分线上或其右侧。数组 Y 被划分为两个数组 Y_1 和 Y_R。类似地，数组 X 也被划分为 X_1 和 X_R。划分后的 X_1 和 Y_1 包含了 P_1 中的点，并分别按 X 坐标和 Y 坐标单调递增的次序进行徘序；X_R 和 Y_R 包含了 P_R 中的点，也按 X 坐标和 Y 坐标单调递增的次序进行排序。

2. 解决

把 P 划分为 P_1 和 P_R 后再进行两次递归调用。

第一次：找出 P_1 中的最近点对，调用的输入为子集 P_1 和数组 Y_1，返回最近点对的距离 d_1。

第二次：找出 P_R 中的最近点对，调用的输入为子集 P_R 和数组 Y_R，返回最近点对的距离 d_R。

置 $d=\min(d_1,d_R)$。

3. 合并

问题是可能还存在其距离小于 d 的点对。如果存在这样的一个点对的话，则点对的一个点在 P_1 中，另一个点在 P_n 中，两个点必定相距垂直平分线 d 单位之内，即它们必定处于以垂直平分线为中心、宽度为 $2d$ 的垂直带形区域内。更精确一点讲，由于点对之间的垂直距离也少于 d 单位，因此，它们是在以垂直平分线为中心的某个 $d\times 2d$ 的矩形区域内，如图 5.17 所示。

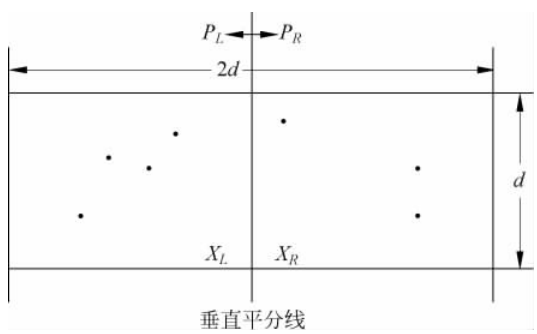

图 5.17　边界区域示例

为了找出这样的点对（如果存在），需要执行以下算法：

(1) 建立一个数组 Y'，Y' 仅含 Y 数组中所有在宽度为 $2d$ 的垂直带形区域内的点，这些点按其 Y 坐标单调递增的顺序存储于 Y'。

(2) 对数组 Y' 中的每个点 P，求出 P 到紧随 P 后的 7 个点间的距离，并记下 Y' 的所有

点对中找出的最近点对的距离 d'。为什么对点 P，在 Y' 中仅需考虑紧随 P 后的 7 个点呢？

设最近点对为 $X_L \in P_L, X_R \in P_R$。由于 X_L 和 X_R 只能在某个 $d \times 2d$ 的矩形区域内出现，且 P_L 中所有点对的距离和 P_R 中的所有点对的距离都至少为 d 单位，因此 P 中至多有 8 个点可能位于该 $d \times 2d$ 矩形内（垂直平分线上的点即属于 P_L，亦可属于 P_R），而 Y' 中 X_L 位于 X_R 之前，那么即使 X_L 在 Y' 中尽可能早出现而 X_R 尽可能晚出现，X_R 也必定是跟随 X_L 的 7 个位置中的一个。

（3）如果 $d' < d$，则垂直带形区域内的确包含比根据递归调用所找出的最近距离 d 更近的点对，于是返回该点对及其距离 d'；否则就返回递归调用中发现的最近点对及其距离 d。

综上所述，最近点对算法 Nearest(S)流程如下：

```
1.  double Nearest (S)
2.  {
3.      n = |S|;
4.      if (n < 2) return;
5.      1.m = S 中各点 x 间坐标的中位数;
6.        构造 S1 和 S2;
7.          //S1 = {p∈S|x(p)<=m}, S2 = {p∈S|x(p)> m}
8.      2.d1 = Nearest (S1);
9.         d2 = Nearest (S2);
10.     3.dm = min(d1,d2);
11.     4.设 P1 是 S1 中距垂直分割线 l 的距离在 dm 之内的所有点组成的集合;
12.        P2 是 S2 中距分割线 l 的距离在 dm 之内所有点组成的集合;
13.        将 P1 和 P2 中点依其 Y 坐标值排序;
14.        并设 X 和 Y 是相应的已排好序的点列;
15.     5.通过扫描 X 以及对于 X 中每个点检查 Y 中与其距离在 dm 之内的所有点(最多 6 个)可以完
          成合并;
16.        当 X 中的扫描指针逐次向上移动时,Y 中的扫描指针可在宽为 2dm 的区间内移动;
17.        设 dl 是按这种扫描方式找到的点对间的最小距离;
18.     6.d = min(dm,dl);
19.     return d;
20. }
```

例题 5.5　Raid（POJ 3714）。

题意简述

有若干车站和加油站，求车站和加油站的最小距离。

思路

求最近点对，更新距离时判断是否是同一类点。

代码

```
1.  /* Accepted 4892K 1844MS C++ */
2.  #include<iostream>
3.  #include<cmath>
4.  using namespace std;
```

```
const double max_distance = 1e100;
struct Node
{
    double x,y;
    bool flag;
};

Node pt[200005];
int y_sort[200005];
inline double Distance(Node &a,Node &b)
{
    if(a.flag!= b.flag)
        return sqrt((a.x - b.x) * (a.x - b.x) + (a.y - b.y) * (a.y - b.y));
    else
        return max_distance;
}
double min_n(double a,double b)
{
    if(a < b)
        return a;
    else
        return b;
}
int x_cmp(const void * a,const void * b)
{
    if(((Node * )a) -> x<((Node * )b) -> x)
        return -1;
    else
        return 1;
}
int y_cmp(const void * a,const void * b)
{
    if(pt[( * (int * )a)].y < pt[( * (int * )b)].y)
        return -1;
    else
        return 1;
}
double shortest_distance(int first,int last)
{
    if(last - first == 1)
        return Distance(pt[first],pt[last]);
    else if(last - first == 2)
        return min_n(min_n(Distance(pt[first],pt[first + 1]),Distance(pt[first],pt
        [first + 2])),Distance(pt[first + 1],pt[first + 2]));
    int mid = (first + last)/2;
    double min_dist = min_n(shortest_distance(first,mid),shortest_distance(mid + 1,
    last));

    if(min_dist == 0)
        return 0;
    int y_end = 0;
```

```
    for(int i = mid; pt[mid].x - pt[i].x < min_dist&&i >= first; i -- ) {
        y_sort[y_end ++ ] = i;
    }
    for(int i = mid + 1; pt[i].x - pt[mid + 1].x < min_dist&&i <= last; i++) {
        y_sort[y_end ++ ] = i;
    }
    qsort(y_sort,y_end,sizeof(y_sort[0]),y_cmp);
    for(int i = 0; i < y_end; i++)  {
        for(int j = i + 1; j < y_end&&pt[y_sort[j]].y - pt[y_sort[i]].y < min_dist; j++)  {
            min_dist = min_n(min_dist,Distance(pt[y_sort[i]],pt[y_sort[j]]));
        }
    }
    return min_dist;
}
int main()
{
    int t;
    scanf("%d",&t);
    while(t -- ) {
        int n;
        scanf("%d",&n);
        for(int i = 0; i < n; i++) {
            scanf("%lf%lf",&pt[i].x,&pt[i].y);
            pt[i].flag = false;
        }
        for(int i = n; i < 2 * n; i++) {
        scanf("%lf%lf",&pt[i].x,&pt[i].y);
        pt[i].flag = true;
    }
    qsort(pt,2 * n,sizeof(pt[0]),x_cmp);
    printf("%.3f\n",shortest_distance(0,2 * n - 1));
 }
 return 0;
}
```

5.6 半平面交

专题讲解

概括地说，半平面交就是一组线性不等式组，找出其满足的点集。

也可以理解为多边形的核（即在这个区域的点能“看到”多边形内部所有的点，想象一个放在这个区域里的摄像头，那么它没有看不到的死角）。

一个半平面可以用一个有向的线段来表示（其实应该把它想象成一条直线，只是用线段的这两点来表示这条直线而已），可以自己规定这条有向线段代表着它右侧的半平面或是左侧的半平面。这个在代码中只是简单的大于和小于的一个改变。

在求解按顺序（顺时针或逆时针）输入的多边形的凸包 MELKMAN 算法中，用到了双

端队列(也称为两头栈,两端均可 PUSH 和 POP)。而在半平面交算法中,同样也采用了两头栈,只是这次栈中的不是点,而是一个半平面(有向线段)。

算法描述如下:

(1) 对所有的半平面按角度排序。如(1,1)向量的角度为$\frac{\pi}{4}$,(−1, −1)则为$-\frac{3\pi}{4}$,用<math.h>中的 atan2 函数来求。

(2) 对于角度相同的,选择靠内侧的那个半平面。

(3) 使用一个双端队列,初始加入头两个半平面。

(4) 每考虑一个新的半平面时:

while 顶端的两个半平面的交点在当前半平面外:删除顶端的半平面。

while 底部的两个半平面的交点在当前半平面外:删除底部的半平面。

将新半平面加入顶端。

(5) 删除两端多余的半平面:

while 顶端的两个半平面的交点在底部半平面外:删除顶端的半平面。

while 底部的两个半平面的交点在顶端半平面外:删除底部的半平面。

该算法的时间复杂度为 $O(n \log n)$。

在了解 MELKMAN 后,其实也就不太难理解这个求半平面交的算法的步骤了。唯一多出的步骤,就是要对这些半平面的角度排一个序(因为对于 MELKMAN 来说,输入的多边形已经有序了。因此,如果这些半平面同样是多边形的边,按顺序输入的话,那么同样该算法可以降到 $O(n)$的复杂度)。

半平面的交,最终形成的是一个凸多边形,根据凸多边形的性质(整个凸多边形在任意一边的一侧),可以发现,在考虑新加入一个半平面的时候,之前半平面的交应该被包含在新考虑的这个半平面内(如何判断? 通过栈顶的两个半平面的交点来看该点是否被包含在新考虑的半平面中,不是的话一直退栈,直至满足条件为止)。每次考虑,都是两头同时维护,保证条件成立。

这样最终就得到了一个半平面(直线)的集合。按次序求这些直线的 u 交点,可以得到该半平面的交集。

需要注意的是,有时候得到的解会是个无穷大的区域,因为可以人为地加上 4 个半平面表示边界。有时候会无解,此时栈中少于 2 个平面。

```
1.  int half_planes_cross(vec * v, int vn)
2.  {
3.      int i, n;
4.      sort(v, v + vn, cmp);
5.      for (i = n = 1; i < vn; ++ i) {
6.          if (sgn(v[i].ag - v[i - 1].ag) == 0) continue;
7.          v[n ++ ] = v[i];
8.      }
9.      int head = 0, tail = 1;
10.     deq[0] = v[0], deq[1] = v[1];
11.
12.     // > for right side of the half plane
```

```
    //change > to < for left side
    for (i = 2; i < n; ++i) {
        if (parallel(deq[tail - 1], deq[tail]) ||
parallel(deq[head], deq[head + 1])) return 0;
        while (head < tail && sgn(side(v[i].s, v[i].e, cross_point(deq[tail - 1], deq
        [tail]))) > 0) --tail;
        while (head < tail && sgn(side(v[i].s, v[i].e, cross_point(deq[head], deq[head +
        1]))) > 0) ++head;
        deq[++tail] = v[i];
    }
    while (head < tail && sgn(side(deq[head].s, deq[head].e, cross_point(deq[tail - 1],
    deq[tail]))) > 0) --tail;
    while (head < tail && sgn(side(deq[tail].s, deq[tail].e, cross_point(deq[head], deq
    [head + 1]))) > 0)) ++head;
    /* judging whether exist or not */
    //not exist the area
    if (tail <= head + 1) return 0;
    //return 1;
    /* get the convex polygon area */
    //res is a struct polygon (with n for number of the vertices)
    res.n = 0;
    for (i = head; i < tail; ++i)
        res.v[res.n++] = cross_point(deq[i], deq[i + 1]);
    res.v[res.n++] = cross_point(deq[head], deq[tail]);
    res.n = unique(res.v, res.v + res.n) - res.v;
    res.v[res.n] = res.v[0];
    return 1;
}
```

例题 5.6 Feng Shui（POJ 3384）。

题意简述

用一个按顺时针方向输入的多边形表示一个房间，现在用两个半径为 r 的地毯来覆盖这个多边形。地毯可以重叠，但必须完整地存在于多边形内。问能覆盖的最大面积时，两块地毯圆心的位置。题目保证必存在一个解。

思路

把多边形的每条边向内平移 r 的距离，则将得到一凸多边形的区域，在这个区域内，放置两个圆均可满足圆在多边形内的条件。那么如何使得覆盖面积最大呢？只需在得到的区域中，相距最远的那两个顶点上放置两个圆的圆心，即为最终解。简化为求多边形的半平面交，在得到的凸包中求最远点对。

代码

```
/* Accepted 228K 0MS C++ */
#include <cstdio>
#include <cstring>
#include <cmath>
#include <climits>
```

```
#include <vector>
#include <algorithm>
using namespace std;
#define EPS 1e-8
#define MAXN 1000
#define PI acos(-1.0)
struct point {
    double x, y;
    bool operator == (const point &b) const {
        if (fabs(x - b.x) < EPS && fabs(y - b.y) < EPS) return true;
        return false;
    }
};
struct seg {
    point s, e;
    double ag;
};
struct polygon {
    point v[MAXN];
    int n;
}pg, res;
typedef point pvec;
typedef seg vec;
typedef seg line;
int sgn(double d)
{
    if (fabs(d) < EPS) return 0;
    return d > 0 ? 1 : -1;
}
double det(double x1, double y1, double x2, double y2)
{
    return x1 * y2 - x2 * y1;
}
//judge which side p is towards directed-seg ab
//pos for left, neg for right
double side(point a, point b, point p)
{
    return det(b.x-a.x, b.y-a.y, p.x-a.x, p.y-a.y);
}
double dis(point a, point b)
{
    return sqrt((a.x - b.x) * (a.x - b.x) + (a.y - b.y) * (a.y - b.y));
}
point cross_point(const vec &a, const vec &b) {
   point res;
   double u = side(a.s, a.e, b.s), v = side(a.e, a.s, b.e);
   res.x = (b.s.x * v + b.e.x * u) / (u + v);
   res.y = (b.s.y * v + b.e.y * u) / (u + v);
   return res;
}
/*****************************
```

```
57.  *     half_planes_cross
58.  ***************************/
59.  int parallel(line a, line b)
60.  {
61.      double u = (a.e.x - a.s.x) * (b.e.y - b.s.y) - (a.e.y - a.s.y) * (b.e.x - b.s.
         x);
62.      return sgn(u) == 0;
63.  }
64.  void set_vector(double x1, double y1, double x2, double y2, vec &v)
65.  {
66.     v.s.x = x1; v.s.y = y1;
67.     v.e.x = x2; v.e.y = y2;
68.     v.ag = atan2(y2 - y1, x2 - x1);
69.  }
70.  vec vct[MAXN];
71.  vec deq[MAXN];
72.  //sgn > for left
73.  //sgn < for right
74.  bool cmp(vec a, vec b)
75.  {
76.      if (sgn(a.ag - b.ag) == 0)
77.          return sgn(side(b.s, b.e, a.s)) < 0;
78.      return a.ag < b.ag;
79.  }
80.  int half_planes_cross(vec * v, int vn)
81.  {
82.      int i, n;
83.      //sort(v, v + vn, cmp);
84.
85.      for (i = n = 1; i < vn; ++ i) {
86.          if (sgn(v[i].ag - v[i - 1].ag) == 0) continue;
87.          v[n ++ ] = v[i];
88.      }
89.
90.      int head = 0, tail = 1;
91.      deq[0] = v[0], deq[1] = v[1];
92.
93.      // > for right side of the half plane
94.      //change > to < for left side
95.      for (i = 2; i < n; ++ i) {
96.          if (parallel(deq[tail - 1], deq[tail]) || parallel(deq[head], deq[head + 1]))
             return 0;
97.          while (head < tail && sgn(side(v[i].s, v[i].e, cross_point(deq[tail - 1], deq
             [tail]))) > 0) -- tail;
98.          while (head < tail && sgn(side(v[i].s, v[i].e, cross_point(deq[head], deq[head
             + 1]))) > 0) ++ head;
99.          deq[ ++ tail] = v[i];
100.     }
101.     while (head < tail && sgn(side(deq[head].s, deq[head].e, cross_point(deq[tail - 1],
         deq[tail])) > 0)) -- tail;
102.     while (head < tail && sgn(side(deq[tail].s, deq[tail].e, cross_point(deq[head], deq
```

```
        [head + 1])) > 0)) ++head;
103.    /* judging whether exist or not */
104.    //not exist the area
105.    if (tail <= head + 1) return 0;
106.    //return 1;
107.    /* get the convex polygon area */
108.    //res is a struct polygon (with n for number of the vertices)
109.    res.n = 0;
110.    for (i = head; i < tail; ++i) {
111.        res.v[res.n++] = cross_point(deq[i], deq[i + 1]);
112.    }
113.    res.v[res.n++] = cross_point(deq[head], deq[tail]);
114.    res.n = unique(res.v, res.v + res.n) - res.v;
115.    res.v[res.n] = res.v[0];
116.    return 1;
117. }
118. void moving(vec *v, int vn, double r)
119. {
120.    for (int i = 0; i < vn; ++i) {
121.        double dx = v[i].e.x - v[i].s.x;
122.        double dy = v[i].e.y - v[i].s.y;
123.        dx = dx / dis(v[i].s, v[i].e) * r;
124.        dy = dy / dis(v[i].s, v[i].e) * r;
125.        v[i].s.x = v[i].s.x + dy;
126.        v[i].e.x = v[i].e.x + dy;
127.        v[i].s.y = v[i].s.y - dx;
128.        v[i].e.y = v[i].e.y - dx;
129.    }
130. }
131. int main()
132. {
133.    int n;
134.    double r;
135.    while (scanf("%d%lf", &n, &r) == 2) {
136.        for (int i = 0; i < n; ++i) {
137.            scanf("%lf%lf", &pg.v[i].x, &pg.v[i].y);
138.        }
139.        pg.v[n] = pg.v[0];
140.
141.        for (int i = 0; i < n; ++i) {
142.            set_vector(pg.v[i].x, pg.v[i].y, pg.v[i + 1].x, pg.v[i + 1].y, vct[i]);
143.        }
144.        moving(vct, n, r);
145.        half_planes_cross(vct, n);
146.
147.        int ix = 0, jx = 0;
148.        double maxdis = 0;
149.        for (int i = 0; i < res.n; ++i) {
150.            for (int j = 0; j < res.n; ++j) {
151.                if (i == j) continue;
152.                double t = dis(res.v[i], res.v[j]);
```

```
153.                    if (sgn(t - maxdis) > 0) {
154.                        maxdis = t;
155.                        ix = i, jx = j;
156.                    }
157.                }
158.            }
159.            printf("%.4f %.4f %.4f %.4f\n", res.v[ix].x, res.v[ix].y, res.v[jx].x,
                res.v[jx].y);
160.        }
161.        return 0;
162.    }
```

5.7 旋转卡壳

专题讲解

在旋转卡壳中，对踵点是一个很重要的概念，也有着很重要的应用。

切线：给定一个凸多边形 P，一条直线 L 与 P 相交，且 P 在 L 的一侧，则 L 为 P 的一条切线。

有向切线：有方向的切线称为有向切线，有向切线与平面交有莫大的联系，与凸包给出的点是逆时针还是顺时针有关系。

对踵点：如果两个点 p、q 在一对平行切线上，那么 p 和 q 是一对对踵点。

对踵点对的形式有 3 种，如图 5.18 所示。

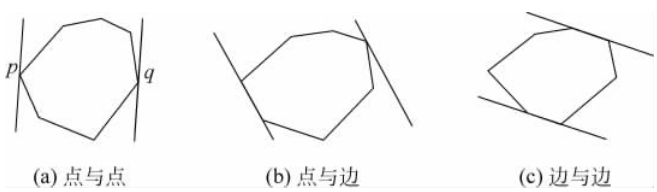

图 5.18 对踵点对

在如图 5.18(b)所示的形式中就存在两对对踵点，而在图 5.8(c)中就存在 4 对对踵点。

比如，给出平面上的一个点集，包含 N 个点，要求最远点对的距离。

思路 1：最容易想到的方法，就是枚举。完全可以枚举所有的点对，求出它们的欧几里得距离，找出其中的最远点对，如果点的数目不是很大的话。这个算法的复杂度是 $O(n^2)$，当点的数目稍大，该方法就不太可取了。

思路 2：点集中的最远点对必定是凸包的顶点。可以用反证法很容易地证明。于是，就变成了先求出凸包，然后再枚举凸包上面的任意两点，取最大距离的对点。但是在最坏情况下，该算法的复杂度依然可以达到 $O(n^2)$。当然，在平均情况下，还是要优于第一种思路的。

思路 3：通过上述的对踵点的概念，采取旋转卡壳的方法。对于最远点，在上述思路中提到了最远点是在凸包的顶点上，同样也可以证明最远点对一定在对踵点集中(也可以很直观地从图形上看出该思路的正确可能性大小)。

下面给出一个可以在$O(n)$时间内找到凸多边形的所有对踵点对的伪代码。

```
1.  p0 = pn;
2.  q = NEXT[p];
3.  while (Area(p,NEXT[p],NEXT[q]) > Area(p,NEXT[p],q)) {
4.     q = NEXT[q];
5.     q0 = q;
6.     while (q != p0) {
7.         p = NEXT[p];
8.          Print(p,q);
9.          while (Area(p,NEXT[p],NEXT[q]) > Area(p,NEXT[p],q) {
10.              q = NEXT[q];
11.              if ((p,q) != (q0,p0)) Print(p,q)
12.              else return
13.          };
14.          if (Area(p,NEXT[p],NEXT[q]) = Area(p,NEXT[p],q)) {
15.              if ((p,q) = (q0,p0)) Print(p,NEXT[q])
16.              else Print(NEXT[p],q)
17.          }
18.      }
19. }
```

这个算法看起来还是比较烦琐，并不如旋转卡壳那么直观，但是它避免了浮点计算，而且实质上是相同的。

更直观的旋转卡壳算法是这样的：

计算凸多边形y方向上的极值点，记为y_{min}、y_{max}。

经过y_{min}、y_{max}作两条平行于x轴的水平线。这两点已经就是对踵点，计算它们的距离并维护最大值。

同时旋转两条直线，直至某一条线与多边形的一条边重合。

此时产生了新的一对对踵点，计算新的距离并比较，如有必要则更新。

重复前两个步骤，直至重新回到y_{min}、y_{max}这对对踵点。

输出最大直径的对踵点对和它们之间的距离。

当然，代码实现起来会更简洁一些。直接从凸多边形的起始点开始找起，然后通过面积比较，找到最远的点对，这样，第一对对踵点就找到了，然后计算它们之间的距离。之后，每次下移一个点，计算相邻的两对对踵点的距离，更新，直至多边形的最后一个点。

```
1.  //O(n)
2.  int dia_rotating_calipers(int cn)
3.  {
4.       int dia = 0;
5.       for (int i = 0, q = 1; i < cn; ++i) {
6.            while (side(ch[i], ch[i + 1], ch[q + 1]) > side(ch[i], ch[i + 1], ch[q]))
7.                 q = (q + 1) % n;
8.            dia = max(dia, max(square_dis(ch[i], ch[q]), square_dis(ch[i + 1], ch[q])));
9.       }
```

```
10.     return dia;
11. }
```

另外，通过上面的算法伪代码（包含了很多的信息和思路），还可以以此求出凸多边形的宽度，更进一步，还可以求出两个不相交的凸包之间的最长距离和最短距离等。

例题 5.7 Beauty Contest（POJ 2187）。

题意简述

有一个农场有 N 个房子（$2<=N<=50\ 000$），问最远的房子相距多少距离。

思路

这就是旋转卡壳的一个最基础的应用，通过寻找所有的对踵点，找出最远的点对。

代码

```
1.  /* Accepted 564K 125MS C++ */
2.  #include<cstdio>
3.  #include<cstring>
4.  #include<cstdlib>
5.  #include<algorithm>
6.  using namespace std;
7.
8.  typedef long long ll;
9.  const int MAXN = 50000;
10.
11. struct point {
12.     int x, y;
13. }p[MAXN], ch[MAXN];
14.
15. int det(int x1, int y1, int x2, int y2)
16. {
17.     return x1 * y2 - x2 * y1;
18. }
19. int side(point a, point b, point p)
20. {
21.     return det(b.x - a.x, b.y - a.y, p.x - a.x, p.y - a.y);
22. }
23. int square_dis(point a, point b)
24. {
25.     return (a.x - b.x) * (a.x - b.x) + (a.y - b.y) * (a.y - b.y);
26. }
27.
28. bool cmp(point a, point b)
29. {
30.     if (a.y == b.y) return a.x<b.x;
31.     return a.y<b.y;
32. }
33. int convex_hull(point p[], int n)
34. {
35.     sort(p, p + n, cmp);
```

```
    ch[0] = p[0];
    if (n == 1) { ch[1] = ch[0]; return 1; }
    ch[1] = p[1];
    if (n == 2) { ch[2] = ch[0]; return 2; }
    int ix = 2;
    for (int i = 2; i < n; ++i) {
        while (ix > 1 && side(ch[ix - 2], ch[ix - 1], p[i]) <= 0) --ix; //<= 0 for no
        more than 3 points in a line
        ch[ix++] = p[i];
    }
    int t = ix;
    ch[ix++] = p[n - 2];
    for (int i = n - 3; i >= 0; --i) {
        while (ix > t && side(ch[ix - 2], ch[ix - 1], p[i]) <= 0) --ix;
        ch[ix++] = p[i];
    }
    return ix - 1;
}
//O(n^2)
int dia_numerator(int cn)
{
    int dia = 0;
    for (int i = 0; i < cn; ++i) {
        for (int j = 0; j < cn; ++j) {
            int t = square_dis(ch[i], ch[j]);
            dia = t > dia ? t : dia;
        }
    }
    return dia;
}
//O(n)
int dia_rotating_calipers(int n)
{
    int dia = 0;
    int q = 1;
    for (int i = 0; i < n; ++i) {
        while (side(ch[i], ch[i + 1], ch[q + 1]) > side(ch[i], ch[i + 1], ch[q]))
            q = (q + 1) % n;
        dia = max(dia, max(square_dis(ch[i], ch[q]), square_dis(ch[i + 1], ch[q + 1])));
    }
    return dia;
}
int main()
{
    int n, cn;
    scanf("%d", &n);
    for (int i = 0; i < n; ++i)
        scanf("%d%d", &p[i].x, &p[i].y);
    cn = convex_hull(p, n);
    printf("%d\n", dia_rotating_calipers(cn));
    return 0;
}
```

5.8 扫描线

专题讲解

扫描线是计算几何领域的基础手法，可以用来解决许多计算几何的问题，有如图论中的BFS与DFS一样经典。它并不是一个物品，而是一个概念。

如图5.19所示，一条(或两条)无限长平行线，沿其垂直方向不断移动，从画面一端移动到另一端，只在顶点处停留。

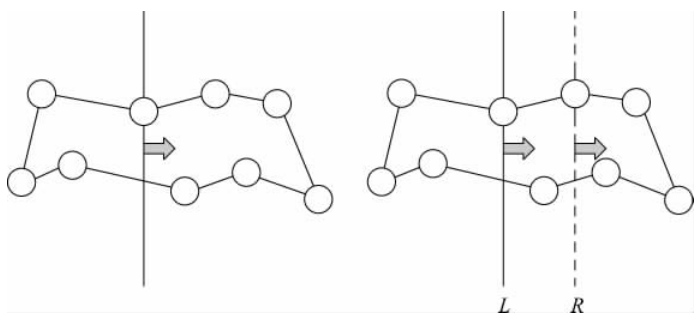

图5.19 扫描线

实际操作时，通常是先按坐标大小排序所有顶点，然后以两索引值，记录平行线的位置在哪个顶点上面。两条平行线，一条为主，穿越整个画面；一条为辅，跟着主线的状况进行平移。这是阴阳的道理。

如图5.20所示，一条(或两条)从原点射出的射线，做360°旋转，只在顶点处停留。

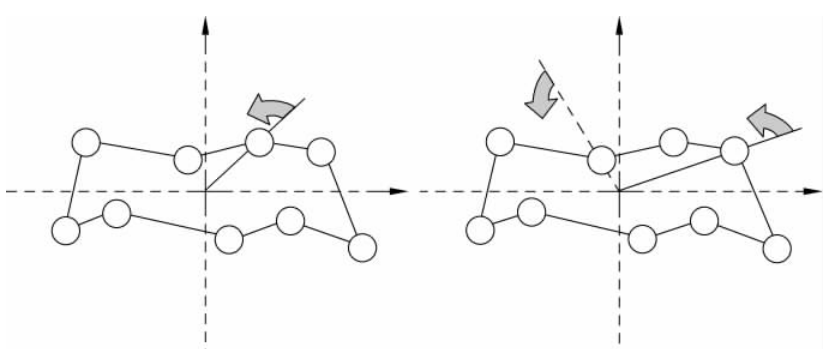

图5.20 旋转的扫描线

实际操作时，通常是先按极角大小排序所有顶点，然后以两索引值，记录射线的位置在哪个顶点上面。两条射线，一条为主，转过整个画面；一条为辅，跟着主线的状况进行平移。

用SweepLine算法的步骤如下：

扫描线的基本精神就是先排序、再搜寻。在二维平面上，有一个重要的特性就是“区域性”。比如说，两点之间，会被距离更近的点隔开，如图5.21所示。

排序的目的，也就是建立“区域性”。有了“区域性”，搜寻的条件就更精确了，搜寻的速

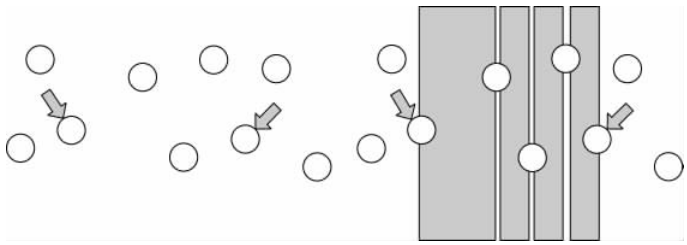

图 5.21 SweepLine 算法步骤

度也就更快了。观察问题是否有“不重叠”、“不相交”、“间隔”、“相聚”之类的性质。然后选定平移的或者旋转的扫描线，进行扫描。换句话说，对所有的顶点排序，进行搜寻。

扫描线可以解决许多问题，平移扫描线可以在诸如“找出所有最近点对”、“找出所有线段的所有交点”、“找出两个多边形的交集、联集、差集”、“Voronoi 图”、“三角剖分”等问题中使用；旋转的扫描线在之前的凸包问题中的解决也用到。

另外，求矩形面积的过程中，会用到离散化的思想和线段树的数据结构。在学习过线段树之后，没看明确的说明前，的确还是想不通为什么要用到线段树。现在想想，线段树不过是一个有特点的数据结构，只是一个优化的工具而已，并不是非用不可。

在求矩形面积的过程中，对所有的 x 坐标离散化，对 y 的水平线段排序，扫描线自下而上。矩形的下边标为+1，上边标为−1，当扫描过后，累加的结果为正时，正说明在矩形的内部，这时用线段树来维护到达每一条 y 坐标下的水平扫描线时，在内的线段有效总长，乘以相邻扫描线的宽度，便得到这一部分的矩形面积，直至最后一条扫描线。

```
1.      int m = 0;
2.      for (int i = 0; i < n; ++ i) {
3.          double a, b, c, d;
4.          scanf("%lf%lf%lf%lf", &a, &b, &c, &d);
5.          dc[m] = a;
6.          rs[m++] = seg(a, c, b, 1);
7.          dc[m] = c;
8.          rs[m++] = seg(a, c, d, -1);
9.      }
10.     sort(dc, dc + m);
11.     sort(rs, rs + m);
12.     int k = unique(dc, dc + m) - dc;                    //disconcrete
13.     memset(cnt, 0, sizeof(cnt));
14.     memset(sum, 0, sizeof(sum));
15.     double res = 0;
16.     for (int i = 0; i < m - 1; ++ i) {
17.         int l = bins(rs[i].l, k, dc);
18.         int r = bins(rs[i].r, k, dc) - 1;               //an interval not a point
19.         if (l <= r) update(1, 0, k - 1, l, r, rs[i].s);
20.         res += sum[1] * (rs[i + 1].h - rs[i].h);
21.     }
```

例题 5.8 Atlantis(POJ 1151)。

题意简述

去一个地方探险,有很多地图描述这个地方,每张地图描述了一个边平行于坐标轴的矩形区域,但是这些地图可能有重叠,求被描述的地方的面积。

思路

扫描从下至上,找出每条扫描线时,总区间的有效长度,乘以相邻扫描线的间距,就是这一部分间的总面积。扫完之后,便得到了总面积。对于每个矩形边,下边记为 1,上边记为 −1,则当标记大于 0 时,这段边是有效的,要算到总和里去,每扫到一个位置,更新一下总长,这用线段树来操作会有较大的优化。

注意对 x 离散排序,对上下边按 y 轴排序。一遍扫描便得结果。

代码

```
1.  /* Accepted 224K 0MS C++ */
2.  #include<cstdio>
3.  #include<cstring>
4.  #include<cmath>
5.  #include<climits>
6.  #include<vector>
7.  #include<algorithm>
8.  using namespace std;
9.  #define EPS 1e-8
10. #define MAXN 400
11. #define PI acos(-1.0)
12. struct seg {
13.     double l, r;
14.     double h;                                          //y coordinate
15.     int s;                                             //above or below
16.     seg() {}
17.     seg(double a, double b, double c, int d) : l(a), r(b), h(c), s(d) {}
18.     bool operator<(const seg &b) const {
19.         return h<b.h;
20.     }
21. }rs[MAXN];
22. double dc[MAXN];
23. double sum[MAXN<<2];
24. int cnt[MAXN<<2];
25. void up(int rt, int l, int r)
26. {
27.     if (cnt[rt]) {
28.         sum[rt] = dc[r + 1] - dc[l];
29.     } else if (l == r) {
30.         sum[rt] = 0;
31.     } else {
32.         sum[rt] = sum[rt<<1] + sum[rt<<1|1];
33.     }
34. }
35. void update(int rt, int l, int r, int L, int R, int c)
```

```
{
    if (L <= l && r <= R) {
        cnt[rt] += c;
        up(rt, l, r);
        return;
    }
    int m = (l + r) >> 1;
    if (L <= m) update(rt << 1, l, m, L, R, c);
    if (R > m) update(rt << 1 | 1, m + 1, r, L, R, c);
    up(rt, l, r);
}
int bins(double key, int n, double arr[])
{
    int l = 0, r = n - 1;
    while (l <= r) {
        int m = (l + r) >> 1;
        if (arr[m] == key) return m;
        if (arr[m] < key) l = m + 1;
        else r = m - 1;
    }
    return -1;
}
int main()
{
    int n, c = 0;;
    while (scanf("%d", &n) && n) {
        int m = 0;
        for (int i = 0; i < n; ++i) {
            double a, b, c, d;
            scanf("%lf%lf%lf%lf", &a, &b, &c, &d);
            dc[m] = a;
            rs[m++] = seg(a, c, b, 1);
            dc[m] = c;
            rs[m++] = seg(a, c, d, -1);
        }
        sort(dc, dc + m);
        sort(rs, rs + m);
        int k = unique(dc, dc + m) - dc;
        memset(cnt, 0, sizeof(cnt));
        memset(sum, 0, sizeof(sum));
        double res = 0;
        for (int i = 0; i < m - 1; ++i) {
            int l = bins(rs[i].l, k, dc);
            int r = bins(rs[i].r, k, dc) - 1;
            if (l <= r) update(1, 0, k - 1, l, r, rs[i].s);
            res += sum[1] * (rs[i + 1].h - rs[i].h);
        }
        printf("Test case #%d\n", ++c);
        printf("Total explored area: %.2f\n\n", res);
    }
    return 0;
}
```

5.9 计算几何基本算法代码集锦

```
#include<cmath>
#include<cstdio>
#include<memory.h>
#include<algorithm>
using namespace std;
typedef double TYPE;
#define Abs(x) (((x)>0)?(x):(-(x)))
#define Sgn(x) (((x)<0)?(-1):(1))
#define Max(a,b) (((a)>(b))?(a):(b))
#define Min(a,b) (((a)<(b))?(a):(b))
#define Epsilon 1e-8
#define Infinity 1e+10
#define Pi 3.14159265358979323846
TYPE Deg2Rad(TYPE deg) {return (deg * Pi / 180.0);}
TYPE Rad2Deg(TYPE rad) {return (rad * 180.0 / Pi);}
TYPE Sin(TYPE deg) {return sin(Deg2Rad(deg));}
TYPE Cos(TYPE deg) {return cos(Deg2Rad(deg));}
TYPE ArcSin(TYPE val) {return Rad2Deg(asin(val));}
TYPE ArcCos(TYPE val) {return Rad2Deg(acos(val));}
TYPE Sqrt(TYPE val) {return sqrt(val);}

//点
struct POINT
{
    TYPE x;
    TYPE y;
    POINT() : x(0), y(0) {};
    POINT(TYPE _x_, TYPE _y_) : x(_x_), y(_y_) {};
};
//两个点的距离
TYPE Distance(const POINT & a, const POINT & b)
{
    return Sqrt((a.x - b.x) * (a.x - b.x) + (a.y - b.y) * (a.y - b.y));
}
//线段
struct SEG
{
    POINT a;                                                    //起点
    POINT b;                                                    //终点
    SEG() {};
    SEG(POINT _a_, POINT _b_):a(_a_),b(_b_) {};
};
//直线(两点式)
struct LINE
{
```

```
    POINT a;
    POINT b;
    LINE() {};
    LINE(POINT _a_, POINT _b_) : a(_a_), b(_b_) {};
};
//直线(一般式)
struct LINE2
{
    TYPE A,B,C;
    LINE2() {};
    LINE2(TYPE _A_, TYPE _B_, TYPE _C_) : A(_A_), B(_B_), C(_C_) {};
};
//两点式化一般式
LINE2 Line2line(const LINE & L)
{
    LINE2 L2;
    L2.A = L.b.y - L.a.y;
    L2.B = L.a.x - L.b.x;
    L2.C = L.b.x * L.a.y - L.a.x * L.b.y;
    return L2;
}
//返回直线 Ax + By + C = 0  的系数
void Coefficient(const LINE & L, TYPE & A, TYPE & B, TYPE & C)
{
    A = L.b.y - L.a.y;
    B = L.a.x - L.b.x;
    C = L.b.x * L.a.y - L.a.x * L.b.y;
}
void Coefficient(const POINT & p,const TYPE a,TYPE & A,TYPE & B,TYPE & C)
{
    A = Cos(a);
    B = Sin(a);
    C = - (p.y * B + p.x * A);
}
//直线相交的交点
POINT Intersection(const LINE & A, const LINE & B)
{
    TYPE A1, B1, C1;
    TYPE A2, B2, C2;
    Coefficient(A, A1, B1, C1);
    Coefficient(B, A2, B2, C2);
    POINT I(0, 0);
    I.x = - (B2 * C1 - B1 * C2) / (A1 * B2 - A2 * B1);
    I.y =   (A2 * C1 - A1 * C2) / (A1 * B2 - A2 * B1);
    return I;
}
//点到直线的距离
TYPE Ptol(const POINT p,const LINE2 L)
{
    return fabs( L.A*p.x + L.B*p.y + L.C ) / Sqrt( L.A*L.A + L.B*L.B );
}
```

```
97.  //两线段叉乘
98.  TYPE Cross2(const SEG & p, const SEG & q)
99.  {
100.     return (p.b.x - p.a.x) * (q.b.y - q.a.y) - (q.b.x - q.a.x) * (p.b.y - p.a.y);
101. }
102. //有公共端点 O 叉乘,并判拐,若正 p0 -> p1 -> p2 拐向左
103. TYPE Cross(const POINT & a, const POINT & b, const POINT & o)
104. {
105.     return (a.x - o.x) * (b.y - o.y) - (b.x - o.x) * (a.y - o.y);
106. }
107. //判等(值,点,直线)
108. bool IsEqual(TYPE a, TYPE b)
109. {
110.     return (Abs(a - b) < Epsilon);
111. }
112. bool IsEqual(const POINT & a, const POINT & b)
113. {
114.     return (IsEqual(a.x, b.x) && IsEqual(a.y, b.y));
115. }
116. bool IsEqual(const LINE & A, const LINE & B)
117. {
118.     TYPE A1, B1, C1;
119.     TYPE A2, B2, C2;
120.     Coefficient(A, A1, B1, C1);
121.     Coefficient(B, A2, B2, C2);
122.     return IsEqual(A1 * B2, A2 * B1) &&
123.         IsEqual(A1 * C2, A2 * C1) &&
124.         IsEqual(B1 * C2, B2 * C1);
125. }
126. //判断点是否在线段上
127. bool IsOnSeg(const SEG & seg, const POINT & p)
128. {
129.     return (IsEqual(p, seg.a) || IsEqual(p, seg.b)) ||
130.         (((p.x - seg.a.x) * (p.x - seg.b.x) < 0 ||
131.         (p.y - seg.a.y) * (p.y - seg.b.y) < 0) &&
132.         (IsEqual(Cross(seg.b, p, seg.a), 0)));
133. }
134. //判断两条线断是否相交,端点重合算相交
135. bool IsIntersect(const SEG & u, const SEG & v)
136. {
137.     return (Cross(v.a, u.b, u.a) * Cross(u.b, v.b, u.a) >= 0) &&
138.         (Cross(u.a, v.b, v.a) * Cross(v.b, u.b, v.a) >= 0) &&
139.         (Max(u.a.x, u.b.x) >= Min(v.a.x, v.b.x)) &&
140.         (Max(v.a.x, v.b.x) >= Min(u.a.x, u.b.x)) &&
141.         (Max(u.a.y, u.b.y) >= Min(v.a.y, v.b.y)) &&
142.         (Max(v.a.y, v.b.y) >= Min(u.a.y, u.b.y));
143. }
144. //判断线段 P 和直线 Q 是否相交,端点重合算相交
145. bool Isonline(const SEG & p, const LINE & q)
146. {
147.     SEG p1,p2,q0;
```

```
    p1.a = q.a;p1.b = p.a;
    p2.a = q.a;p2.b = p.b;
    q0.a = q.a;q0.b = q.b;
    return (Cross2(p1,q0) * Cross2(q0,p2)>= 0);
}

//判断两条线段是否平行
bool IsParallel(const LINE & A, const LINE & B)
{
    TYPE A1, B1, C1;
    TYPE A2, B2, C2;
    Coefficient(A, A1, B1, C1);
    Coefficient(B, A2, B2, C2);
    //共线不算平行
/*     return (A1 * B2 == A2 * B1) &&
        ((A1 * C2 != A2 * C1) || (B1 * C2 != B2 * C1)); */
    //共线算平行
    return (A1 * B2 == A2 * B1);
}

//判断两条直线是否相交
bool IsIntersect(const LINE & A, const LINE & B)
{
    return !IsParallel(A, B);
}

//矩形
//矩形的线段
//        2
//    ---------------- b
//    |                |
//  3 |                |  1
//  a ----------------
//        0
struct RECT
{
    POINT a;                                          //左下点
    POINT b;                                          //右上点
    RECT() {};
    RECT(const POINT & _a_, const POINT & _b_)
    {
        a = _a_;
        b = _b_;
    }
};
//矩形化标准
RECT Stdrect(const RECT & q) {
    TYPE t;
    RECT p = q;
    if(p.a.x > p.b.x) {t = p.a.x ; p.a.x = p.b.x ; p.b.x = t;}
    if(p.a.y > p.b.y) {t = p.a.y ; p.a.y = p.b.y ; p.b.y = t;}
```

```
    return p;
}
//根据下标返回矩形的边
SEG Edge(const RECT & rect, int idx)
{
    SEG edge;
    while (idx < 0) idx += 4;
    switch (idx % 4) {
    case 0:
        edge.a = rect.a;
        edge.b = POINT(rect.b.x, rect.a.y);
        break;
    case 1:
        edge.a = POINT(rect.b.x, rect.a.y);
        edge.b = rect.b;
        break;
    case 2:
        edge.a = rect.b;
        edge.b = POINT(rect.a.x, rect.b.y);
        break;
    case 3:
        edge.a = POINT(rect.a.x, rect.b.y);
        edge.b = rect.a;
        break;
    default:
        break;
    }
    return edge;
}
//矩形的面积
TYPE Area(const RECT & rect)
{
    return (rect.b.x - rect.a.x) * (rect.b.y - rect.a.y);
}
//两个矩形的公共面积
TYPE CommonArea(const RECT & A, const RECT & B)
{
    TYPE area = 0.0;
    POINT LL(Max(A.a.x, B.a.x), Max(A.a.y, B.a.y));
    POINT UR(Min(A.b.x, B.b.x), Min(A.b.y, B.b.y));
    if ((LL.x <= UR.x) && (LL.y <= UR.y)) {
        area = Area(RECT(LL, UR));
    }
    return area;
}
//圆
struct CIRCLE {
    TYPE x;
    TYPE y;
    TYPE r;
    CIRCLE() {}
```

```
250.     CIRCLE(TYPE _x_, TYPE _y_, TYPE _r_) : x(_x_), y(_y_), r(_r_) {}
251. };
252. //圆心
253. POINT Center(const CIRCLE & circle)
254. {
255.     return POINT(circle.x, circle.y);
256. }
257. //圆的面积
258. TYPE Area(const CIRCLE & circle)
259. {
260.     return Pi * circle.r * circle.r;
261. }
262. //两个圆的公共面积
263. TYPE CommonArea(const CIRCLE & A, const CIRCLE & B)
264. {
265.     TYPE area = 0.0;
266.     const CIRCLE & M = (A.r > B.r) ? A : B;
267.     const CIRCLE & N = (A.r > B.r) ? B : A;
268.     TYPE D = Distance(Center(M), Center(N));
269.     if ((D < M.r + N.r) && (D > M.r - N.r)) {
270.         TYPE cosM = (M.r * M.r + D * D - N.r * N.r) / (2.0 * M.r * D);
271.         TYPE cosN = (N.r * N.r + D * D - M.r * M.r) / (2.0 * N.r * D);
272.         TYPE alpha = 2.0 * ArcCos(cosM);
273.         TYPE beta  = 2.0 * ArcCos(cosN);
274.         TYPE TM = 0.5 * M.r * M.r * Sin(alpha);
275.         TYPE TN = 0.5 * N.r * N.r * Sin(beta);
276.         TYPE FM = (alpha / 360.0) * Area(M);
277.         TYPE FN = (beta / 360.0) * Area(N);
278.         area = FM + FN - TM - TN;
279.     } else if (D <= M.r - N.r) {
280.         area = Area(N);
281.     }
282.     return area;
283. }
284. //判断圆是否在矩形内(不允许相切)
285. bool IsInCircle(const CIRCLE & circle, const RECT & rect)
286. {
287.     return (circle.x - circle.r > rect.a.x) &&
288.         (circle.x + circle.r < rect.b.x) &&
289.         (circle.y - circle.r > rect.a.y) &&
290.         (circle.y + circle.r < rect.b.y);
291. }
292. //判断矩形是否在圆内(不允许相切)
293. bool IsInRect(const CIRCLE & circle, const RECT & rect)
294. {
295.     POINT c,d;
296.     c.x = rect.a.x; c.y = rect.b.y;
297.     d.x = rect.b.x; d.y = rect.a.y;
298.     return (Distance( Center(circle) , rect.a ) < circle.r) &&
299.         (Distance( Center(circle) , rect.b ) < circle.r) &&
300.         (Distance( Center(circle) , c ) < circle.r) &&
```

```
301.          (Distance( Center(circle) , d ) < circle.r);
302. }
303. //判断矩形是否与圆相离(不允许相切)
304. bool Isoutside(const CIRCLE & circle, const RECT & rect)
305. {
306.     POINT c,d;
307.     c.x = rect.a.x; c.y = rect.b.y;
308.     d.x = rect.b.x; d.y = rect.a.y;
309.     return (Distance( Center(circle) , rect.a ) > circle.r) &&
310.          (Distance( Center(circle) , rect.b ) > circle.r) &&
311.          (Distance( Center(circle) , c ) > circle.r) &&
312.          (Distance( Center(circle) , d ) > circle.r) &&
313.          (rect.a.x > circle.x || circle.x > rect.b.x || rect.a.y > circle.y || circle.y >
             rect.b.y) ||
314.          ((circle.x - circle.r > rect.b.x) ||
315.          (circle.x + circle.r < rect.a.x) ||
316.          (circle.y - circle.r > rect.b.y) ||
317.          (circle.y + circle.r < rect.a.y));
318. }
319. //三角形
320. struct TRIANGLE {
321.     POINT a;
322.     POINT b;
323.     POINT c;
324.     TRIANGLE() {};
325.     TRIANGLE(const POINT & _a_, const POINT & _b_,  const POINT & _c_)
326.     {
327.          a = _a_;
328.          b = _b_;
329.          c = _c_;
330.     }
331. };
332. //三角形重心
333. POINT InCenter(const TRIANGLE & t)
334. {
335.     return POINT((t.a.x + t.b.x + t.c.x)/3,(t.a.y + t.b.y + t.c.y)/3);
336. }
337. //三角形外心
338. POINT CcCenter(const TRIANGLE & t)
339. {
340.     POINT u,v;
341.     LINE A,B;
342.     A.a = t.a;A.b = t.b;B.a = t.b;B.b = t.c;
343.     TYPE A1, B1, C1;
344.     TYPE A2, B2, C2;
345.     Coefficient(A, B1, A1, C1);
346.     Coefficient(B, B2, A2, C2);
347.     B1 =- B1;B2 =- B2;
348.     C1 =- ( (A.a.x + A.b.x) * A1 + (A.a.y + B.a.y) * B1 ) / 2;
349.     C2 =- ( (B.a.x + B.b.x) * A2 + (B.a.y + B.b.y) * B2 ) / 2;
350.     POINT I(0, 0);
```

```
    I.x = -(B2 * C1 - B1 * C2) / (A1 * B2 - A2 * B1);
    I.y =  (A2 * C1 - A1 * C2) / (A1 * B2 - A2 * B1);
    return I;
}
//三角形内切圆面积
TYPE InArea(const TRIANGLE & t)
{
    TYPE p,a,b,c,s;
    a = Distance(t.a,t.b);
    c = Distance(t.a,t.c);
    b = Distance(t.c,t.b);
    s = (a + b + c)/2;
    p = s * (s - a) * (s - b) * (s - c);
    s = p * Pi/s/s;
    return s;
}
//三角形外接圆面积
TYPE OutArea(const TRIANGLE & t)
{
    TYPE a,b,c;
    a = (t.a.x - t.b.x) * (t.a.x - t.b.x) + (t.a.y - t.b.y) * (t.a.y - t.b.y);
    b = (t.a.x - t.c.x) * (t.a.x - t.c.x) + (t.a.y - t.c.y) * (t.a.y - t.c.y);
    c = (t.c.x - t.b.x) * (t.c.x - t.b.x) + (t.c.y - t.b.y) * (t.c.y - t.b.y);
    a = Pi * sqrt(c/(1 - (a + b - c) * (a + b - c)/a/b/4));
    return a;
}

//多边形,逆时针或顺时针给出 x,y
struct POLY {
    int n;                                          //n 个点
    TYPE * x;                                       //x,y 为点的指针,首尾必须重合
    TYPE * y;
    POLY() : n(0), x(NULL), y(NULL) {};
    POLY(int _n_, const TYPE * _x_, const TYPE * _y_)
    {
        n = _n_;
        x = new TYPE[n + 1];
        memcpy(x, _x_, n * sizeof(TYPE));
        x[n] = _x_[0];
        y = new TYPE[n + 1];
        memcpy(y, _y_, n * sizeof(TYPE));
        y[n] = _y_[0];
    }
};
//多边形顶点
POINT Vertex(const POLY & poly, int idx)
{
    idx %= poly.n;
    return POINT(poly.x[idx], poly.y[idx]);
}
//多边形的边
```

```
SEG Edge(const POLY & poly, int idx)
{
    idx % = poly.n;
    return SEG(POINT(poly.x[idx], poly.y[idx]),
        POINT(poly.x[idx + 1], poly.y[idx + 1]));
}
//多边形的周长
TYPE Perimeter(const POLY & poly)
{
    TYPE p = 0.0;
    for (int i = 0; i < poly.n; i++)
        p = p + Distance(Vertex(poly, i), Vertex(poly, i + 1));
    return p;
}
//求多边形面积
TYPE Area(const POLY & poly)
{
    if (poly.n < 3) return TYPE(0);
    double s = poly.y[0] * (poly.x[poly.n - 1] - poly.x[1]);
    for (int i = 1; i < poly.n; i++) {
        s += poly.y[i] * (poly.x[i - 1] - poly.x[(i + 1) % poly.n]);
    }
    return s/2;
}
//多边形的重心
POINT InCenter(const POLY & poly)
{
    TYPE S,Si,Ax,Ay;
    POINT p;
    Si = (poly.x[poly.n - 1] * poly.y[0] - poly.x[0] * poly.y[poly.n - 1]);
    S = Si;
    Ax = Si * (poly.x[0] + poly.x[poly.n - 1]);
    Ay = Si * (poly.y[0] + poly.y[poly.n - 1]);
    for(int i = 1;i < poly.n;i++) {
       Si = (poly.x[i - 1] * poly.y[i] - poly.x[i] * poly.y[i - 1]);
       Ax += Si * ( poly.x[i - 1] + poly.x[i] );
       Ay += Si * ( poly.y[i - 1] + poly.y[i] );
       S += Si;
    }
    S = S * 3;
    return POINT(Ax/S,Ay/S);
}

//判断点是否在多边形上
bool IsOnPoly(const POLY & poly, const POINT & p)
{
    for (int i = 0; i < poly.n; i++) {
        if (IsOnSeg(Edge(poly, i), p)) {
            return true;
        }
    }
```

```
453.     return false;
454. }
455. //判断点是否在多边形内部
456. bool IsInPoly(const POLY & poly, const POINT & p)
457. {
458.     SEG L(p, POINT(Infinity, p.y));
459.     int count = 0;
460.     for (int i = 0; i < poly.n; i++) {
461.         SEG S = Edge(poly, i);
462.         if (IsOnSeg(S, p)) {
463.             return true;     //如果在 poly 上则返回 true
464.         }
465.         if (!IsEqual(S.a.y, S.b.y)) {
466.             POINT & q = (S.a.y > S.b.y)?(S.a):(S.b);
467.             if (IsOnSeg(L, q)) {
468.                 ++count;
469.             } else if (!IsOnSeg(L, S.a) && !IsOnSeg(L, S.b) && IsIntersect(S, L)) {
470.                 ++count;
471.             }
472.         }
473.     }
474.     return (count % 2 != 0);
475. }
476. //判断是否为简单多边形
477. bool IsSimple(const POLY & poly)
478. {
479.     if (poly.n < 3)
480.         return false;
481.     SEG L1, L2;
482.     for (int i = 0; i < poly.n - 1; i++) {
483.         L1 = Edge(poly, i);
484.         for (int j = i + 1; j < poly.n; j++) {
485.             L2 = Edge(poly, j);
486.             if (j == i + 1) {
487.                 if (IsOnSeg(L1, L2.b) || IsOnSeg(L2, L1.a))
488.                     return false;
489.             }
490.             else if (j == poly.n - i - 1) {
491.                 if (IsOnSeg(L1, L2.a) || IsOnSeg(L2, L1.b))
492.                     return false;
493.             } else {
494.                 if (IsIntersect(L1, L2))
495.                     return false;
496.             }
497.         } //for j
498.     } //for i
499.     return true;
500. }
501. //点阵的凸包,返回一个多边形()
502. POLY ConvexHull(const POINT * set, int n)       // 不适用于点少于 3 个的情况
503. {
```

```
504.     POINT * points = new POINT[n];
505.     memcpy(points, set, n * sizeof(POINT));
506.     TYPE * X = new TYPE[n];
507.     TYPE * Y = new TYPE[n];
508.     int i, j, k = 0, top = 2;
509.     for(i = 1; i < n; i++) {
510.         if ((points[i].y < points[k].y) ||
511.             ((points[i].y == points[k].y) &&
512.             (points[i].x < points[k].x))) {
513.             k = i;
514.         }
515.     }
516.     std::swap(points[0], points[k]);
517.     for (i = 1; i < n - 1; i++) {
518.         k = i;
519.         for (j = i + 1; j < n; j++) {
520.             if ((Cross(points[j], points[k], points[0]) > 0) ||
521.                 ((Cross(points[j], points[k], points[0]) == 0) &&
522.                 (Distance(points[0], points[j]) < Distance(points[0], points[k])))) {
523.                 k = j;
524.             }
525.         }
526.         std::swap(points[i], points[k]);
527.     }
528.     X[0] = points[0].x; Y[0] = points[0].y;
529.     X[1] = points[1].x; Y[1] = points[1].y;
530.     X[2] = points[2].x; Y[2] = points[2].y;
531.     for (i = 3; i < n; i++)  {
532.         while (Cross(points[i], POINT(X[top], Y[top]),
533.             POINT(X[top - 1], Y[top - 1])) >= 0) {
534.             top--;
535.         }
536.         ++top;
537.         X[top] = points[i].x;
538.         Y[top] = points[i].y;
539.     }
540.     delete[] points;
541.     POLY poly(++top, X, Y);
542.     delete[] X;
543.     delete[] Y;
544.     return poly;
545. }
```