目录

//SuperPoint，点线的基本定义

//头文件和基本常数

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#include<math.h>
#include<algorithm>
using namespace std;
const double eps = 1e-8;
const double pi = acos(-1.0);//3.1415926...度数转弧度的时候别忘了乘
int dcmp(double x) {return (x > eps) - (x < -eps);} //消除浮点误差比较函数
inline double Sqr(double x) {return x * x;}//平方
/*不用std::max 和std::min 的时候*/
/*
inline double min(double a, double b) {return a < b ? a : b;}
inline double max(double a, double b) {return a > b ? a : b;}
*/
```

//Point 定义

```
struct Point
{
    /**********一般都要用的*********/
    double x, y;
    Point(){x = y = 0;}
    Point(double a, double b) {x = a, y = b;}

    inline Point operator-(const Point &b)const//重载减法，求Dis 要调用
    {return Point(x - b.x, y - b.y);}

    inline bool operator<(const Point &b)const//重载<，排序要用
    {return dcmp(x - b.x) ? x < b.x : y < b.y;}

    inline double dot(const Point &b)const//点积
    {return x * b.x + y * b.y;}

    inline double Dis(const Point &b)const//距离
    {return sqrt((*this - b).dot(*this - b));}

    inline double cross(const Point &b, const Point &c)const//三点叉积，(*this).cross(b, c)右手关系为正
    {return (b.x - x) * (c.y - y) - (c.x - x) * (b.y - y);}
    /**********点线关系有时候用的*********/
    /*
    bool Parallel()判断平行，struct 外的函数
    LineCross()直线交点，struct 外的函数
    SegCross()线段交点，struct 外的函数
    ToSeg()点到线段距离，声明在struct 里具体定义在struct 外面
    这四个里，下面的一般会调用上面的。
    */
    double ToSeg(const Point&, const Point&)const;//点到线段距离*/

    inline Point operator+(const Point &b)const
    {return Point(x + b.x, y + b.y);}

    inline Point operator*(const double &b)const//x、y 扩大常数倍
    {return Point(x * b, y * b);}

    inline Point operator-()
    {return Point(-x, -y);}

    inline bool InLine(const Point &b, const Point &c)const//三点共线
    {return !dcmp(cross(b, c));}

    inline bool OnSeg(const Point &b, const Point &c)const//点在线段上，包括端点
    {return InLine(b, c) && (*this - c).dot(*this - b) < eps;}
```

```cpp
    inline bool InSeg(const Point &b, const Point &c)const//点在线段上，不包括端点
    {return InLine(b, c) && (*this - c).dot(*this - b) < -eps;}

    /**********其他。遇到了再添加**********/
    inline bool operator>(const Point &b)const
    {return b < *this;}

    inline bool operator==(const Point &b)const
    {return !dcmp(x - b.x) && !dcmp(y - b.y);}

    Point RotePoint(const Point &p, double ang)//p 绕*this 逆时针旋转 ang 弧度
    {
        return Point((p.x - x) * cos(ang) - (p.y - y) * sin(ang) + x,
                (p.x - x) * sin(ang) + (p.y - y) * cos(ang) + y);
    }
};
/************************************************/
```

//判断两直线平行

```cpp
bool Parallel(const Point &a, const Point &b, const Point &c, const Point &d)
{return !dcmp(a.cross(b, a + d - c));}
```

//直线交点

```cpp
Point LineCross(const Point &a, const Point &b, const Point &c, const Point &d)
{
    double u = a.cross(b, c), v = b.cross(a, d);
    return Point((c.x * v + d.x * u) / (u + v), (c.y * v + d.y * u) / (u + v));
}
```

//点到线段距离

```cpp
double Point::ToSeg(const Point &b, const Point &c)const//点到线段距离
{
    Point t(x + b.y - c.y, y + c.x - b.x);
    if(cross(t, b) * cross(t, c) > eps)
        return min(Dis(b), Dis(c));
    return Dis(LineCross(*this, t, b, c));
}
```

//线段交点

```cpp
//包括端点，要不包括的话，"<="换成"<"
bool SegCross(const Point &a, const Point &b, const Point &c, const Point &d, Point &p)//线段交点，包括端点
{
    if(!Parallel(a, b, c, d) && a.cross(b, c) * a.cross(b, d) <= 0 && c.cross(d, a) * c.cross(d, b) <= 0)
    {
        p = LineCross(a, b, c, d);
        return true;
    }
    return false;
}
```

//经典问题

```
/***************************************************/
```

//半平面交

```
struct Line
{
    //double a, b, c;//ax + by + c = 0
    Point s, e;//s->e 向量表示有向直线，半平面在直线左侧
    double ang, d;
    inline void Read(Point s_, Point e_)
    {
        s = s_, e = e_;
        ang = atan2(e.y - s.y, e.x - s.x);
        if(dcmp(s.x - e.x)) d = (s.x * e.y - e.x * s.y) / fabs(s.x - e.x);
        else d = (s.x * e.y - e.x * s.y) / fabs(s.y - e.y);
        /**********/
        //如果需要a,b,c，在这里计算
        /**********/
    }
/*    void Read(double a_, double b_, double c_, bool up)//up 表示半平面在直线方程上方
    {
        //a = a_, b = b_, c = c_;
        if(b_ < -eps) a_ = -a_, b_ = -b_, c_ = -c_, up ^= 1;
        else if(!dcmp(b_) && a_ < -eps) a_ = -a_, c_ = -c_, up ^= 1;
        if(!dcmp(a_) && !dcmp(b_))
        {
            if(-c_ > eps && !up || -c_ < -eps && up) Read(Point(inf, inf), Point(-inf, inf));
            else Read(Point(inf, -inf), Point(-inf, -inf));
            return;
        }
        else if(!dcmp(a_)) Read(Point(0, -c_ / b_), Point(1, -c_ / b_));
        else if(!dcmp(b_)) Read(Point(-c_ / a_, 1), Point(-c_ / a_, 0));
        else Read(Point(0, -c_ / b_), Point(1, (-c_ - a_) / b_));
        if(!up) {Read(e, s);}
    }*/
    Line(){}
    Line(Point s_, Point e_){Read(s_, e_);}
//    Line(double a_, double b_, double c_, bool up){Read(a_, b_, c_, up);}
    inline bool Parallel(const Line &l)
    {return !dcmp( (e.x - s.x) * (l.e.y - l.s.y) - (e.y - s.y) * (l.e.x - l.s.x) );}
    Point operator*(const Line &l)const//求两不平行不重合直线交点
    {
        double u = s.cross(e, l.s), v = e.cross(s, l.e);
        return Point( (l.s.x * v + l.e.x * u) / (u + v),
                      (l.s.y * v + l.e.y * u) / (u + v) );
    }
    bool operator<(const Line &l)const//排序函数，优先极角，“左”边直线靠前
    {return dcmp(ang - l.ang) ? ang < l.ang : d < l.d;}
};
Line deq[maxn];
bool HalfPanelCross(Line l[], int n, Point cp[], int &m)
//l 为有向直线，核在直线左侧，n 为直线个数，cp 为保存的核顶点，逆时针。m 为保存的核顶点个数
{
    int i, tn;
    m = 0;
    std::sort(l, l + n);
    for(i = tn = 1; i < n; ++ i) if(dcmp(l[i].ang - l[i - 1].ang))l[tn ++] =l[i];
    n = tn;
    int front = 0, rear = 1;
    deq[0] = l[0], deq[1] = l[1];
    for(i = 2; i < n; ++ i)
    {
        if(deq[rear].Parallel(deq[rear - 1]) || deq[front].Parallel(deq[front + 1])) return false;
        while(front < rear && dcmp( l[i].s.cross(l[i].e, deq[rear] * deq[rear - 1]) ) < 0) -- rear;
        while(front < rear && dcmp( l[i].s.cross(l[i].e, deq[front] * deq[front + 1]) ) < 0) ++ front;
        deq[++ rear] = l[i];
```

```
    }
    while(front < rear && dcmp( deq[front].s.cross(deq[front].e, deq[rear] * deq[rear - 1]) ) < 0) -- rear;
    while(front < rear && dcmp( deq[rear].s.cross(deq[rear].e, deq[front] * deq[front + 1]) ) < 0) ++ front;
    if(rear <= front + 1) return false;//两条以下直线，没有围住
    //保存核
    for(i = front; i < rear; ++ i) cp[m ++] = deq[i] * deq[i + 1];
    if(front < rear + 1) cp[m ++] = deq[front] * deq[rear];
    m = std::unique(cp, cp + m) - cp;//去掉重复点
//    for(i = 0; i < m; ++ i) cp[i].x = pz(cp[i].x), cp[i].y = pz(cp[i].y);//负0 误差修复
    return true;
}
/**************************************************/
```

//各边平移求新核

```
Point ParallelMove(Point a, Point b, Point ret, double L)//将ret 沿a->b 方向左侧垂直平移L
{
    Point tmp;
    double len = a.Dis(b);
    return ret + Point((a.y - b.y) / len * L, (b.x - a.x) / len * L);
}
void MakeNewPanels(Point p[], int n, Line l[], double L)//生成多边形的边向内平移L 后的半平面集
{
    p[n] = p[0];
    for(int i = 0; i < n; ++ i)
        l[i].Read(ParallelMove(p[i], p[i + 1], p[i], L),
            ParallelMove(p[i], p[i + 1], p[i + 1], L));
}
/**************************************************/
```

//凸包

```
int Graham(Point p[], int n, Point res[], int &top)//求凸包,结果为逆时针顺序
{
    int len, i;
    top = 1;
    if(n < 2) {res[0] = p[0]; return 1;}
    std::sort(p, p + n);
    res[0] = p[0], res[1] = p[1];
    for(i = 2; i < n; ++ i)
    {
        while(top && res[top - 1].cross(res[top], p[i]) <= 0)
            -- top;
        res[++ top] = p[i];
    }
    len = top;
    res[++ top] = p[n - 2];
    for(i = n - 3; i >= 0; -- i)
    {
        while(top != len && res[top - 1].cross(res[top], p[i]) <= 0)
            -- top;
        res[++ top] = p[i];
    }
    return top;
}
/*********************************************************************/
```

//简单多边形面积

```
double PolygonArea(Point p[], int n)
{
    if(n < 3) return 0.0;
    double s = p[0].y * (p[n - 1].x - p[1].x);
    p[n] = p[0];
    for(int i = 1; i < n; ++ i)
        s += p[i].y * (p[i - 1].x - p[i + 1].x);
    return fabs(s * 0.5);//顺时针方向s 为负
}
/*********************************************************************/
```

//凸多边形最远点对

```
//需要求点的时候把max换成判断更新。
double CPFMP(Point p[], int n)//ConvexPolygonFarMostPoints
{
    int i, j;
    double ans = 0;
    p[n] = p[0];
    for(i = 0, j = 1; i < n; ++ i)
    {
        while(p[i].cross(p[i + 1], p[j + 1]) > p[i].cross(p[i + 1], p[j]))
            j = (j + 1) % n;
        ans = max(ans, p[i].Dis(p[j]), p[i + 1].Dis(p[j + 1]));
    }
    return ans;
}
/**************************************************************************/
```

//两不相交凸多边形最近点对

```
//需要求点的时候把min换成判断更新。
//计算方法: double ans = min(CPMDTCP(p, n, q, m), CPMDTCP(q, m, p, n)));
inline double min(double a, double b, double c) {return min(a, min(b, c));}
double CPMDTCP(Point p[], int n, Point q[], int m)//ConvexPolygonMinDistanceToConvexPolygon
{
    int i, tp, tq;
    p[n] = p[0], q[m] = q[0];
    for(i = tp = 0; i < n; ++ i) if(p[i].y < p[tp].y) tp = i;
    for(i = tq = 0; i < m; ++ i) if(q[i].y > q[tq].y) tq = i;
    double tmp, mindis = p[tp].Dis(q[tq]);
    for(i = 0; i < n; ++ i)
    {
        while((tmp = p[tp].cross(p[tp + 1], q[tq]) - p[tp].cross(p[tp + 1], q[tq + 1])) < -eps)
            tq = (tq + 1) % m;
        if(tmp > eps) mindis = min(mindis, q[tq].ToSeg(p[tp], p[tp + 1]));
        else mindis = min(mindis,
                          min(p[tp].ToSeg(q[tq], q[tq + 1]), p[tp + 1].ToSeg(q[tq], q[tq + 1])),
                          min(q[tq].ToSeg(p[tp], p[tp + 1]), q[tq + 1].ToSeg(p[tp], p[tp + 1])));
        tp = (tp + 1) % n;
    }
    return mindis;
}
/*角度判断版
double CPMDTCP(Point p[], int n, Point q[], int m)//ConvexPolygonMinDistanceToConvexPolygon
{
    int i, j, inex, jnex, tp0, tq0;
    p[n] = p[0], q[m] = q[0];
    for(i = tp0 = 0; i < n; ++ i) if(p[i].y > p[tp0].y) tp0 = i;
    for(i = tq0 = 0; i < m; ++ i) if(q[i].y < q[tq0].y) tq0 = i;
    double mindis = p[tp0].Dis(q[tq0]);
    double angp = pi, angq = 0, angpnex, angqnex;
    bool flag = false;
    for(i = tp0, j = tq0;; )
    {
        inex = (i + 1) % n, jnex = (j + 1) % m;
        angpnex = atan2(p[inex].y - p[i].y, p[inex].x - p[i].x);
        angqnex = atan2(q[jnex].y - q[j].y, q[jnex].x - q[j].x);
        switch(dcmp(CADis(angp, angpnex) - CADis(angq, angqnex)))
        {
            case 0: mindis = min(mindis,
                                 min(p[i].ToSeg(q[j], q[jnex]), p[inex].ToSeg(q[j], q[jnex])),
                                 min(q[j].ToSeg(p[i], p[inex]), q[jnex].ToSeg(p[i], p[inex])));
                    i = inex, j = jnex; break;
            case 1: mindis = min(mindis,
                                 min(p[i].ToSeg(q[j], q[jnex]), p[inex].ToSeg(q[j], q[jnex])));
                    j = jnex; break;
            case -1:mindis = min(mindis,
                                 min(q[j].ToSeg(p[i], p[inex]), q[jnex].ToSeg(p[i], p[inex])));
                    i = inex; break;
```

```
        }
        if(!flag && i != tp0 && j != tq0) flag = true;
        if(flag && (i == tp0 || j == tq0)) break;
    }
    while(i != tp0 && j == tq0)
    {
        inex = (i + 1) % n;
        mindis = min(mindis, min(q[j].ToSeg(p[i], p[inex]), q[jnex].ToSeg(p[i], p[inex])));
        i = inex;
    }
    while(j != tq0 && i == tp0)
    {
        jnex = (j + 1) % m;
        mindis = min(mindis, min(p[i].ToSeg(q[j], q[jnex]), p[inex].ToSeg(q[j], q[jnex])));
        j = jnex;
    }
    return mindis;
}
*/
/*************************************************/
```

//给定半径圆和散点集求圆覆盖最多点的个数

```
/*
这里添加角度区间模版（atan2 (-pi,pi]转区间覆盖处理)
struct Cov
int ctp;
int AScomp(const void *a, const void *b)//角度区间排序
void AngManage(double &x)//角度区间修正，(-pi, pi]
void AddAnSeg(double start, double end)
*/
int CircleCoverPoing(Point p[], int n, double R)
{
    int i, j, ans = 0, cnt;
    double dis, ang, ac, RR = R + R;
    for(i = 0 ; i < n; ++ i)
    {
        for(j = ctp = 0; j < n; ++ j)
            if(j != i && (dis = p[i].Dis(p[j])) < RR + eps)//包不包括圆上来确定 +/- eps
            {
                ang = atan2((double)p[j].y - p[i].y, (double)p[j].x - p[i].x);
                ac = acos(dis * 0.5 / R);
                AddAnSeg(ang - ac, ang + ac);
            }
        qsort(cover, ctp, sizeof(Cov), AScomp);
        for(j = cnt = 0; j < ctp; ++ j)
            ans = std::max(ans, cnt += cover[j].se);
    }
    return ans + 1;
}
/*************************************************/
```

//多边形有向边顺逆时针判断及反转

```
//支持简单多边形
void MakeCounterClock(Point p[], int n)//顺时针则反转多边形的有向边
{
    int i, id = 0;
    p[n] = p[0];
    for(i = 0; i < n; ++ i) if(p[i].x > p[id].x) id = i;
    if(p[id].cross(p[id + 1], p[(id + n - 1) % n]) > eps) return;
    Point tmp;
    for(i = n - 1 >> 1; i >= 0; -- i)
        tmp = p[i], p[i] = p[n - i - 1], p[n - i - 1] = tmp;
}
/*************************************************/
```

//判断点在简单多边形内

```
double AngCounterClock(double start, double end)//start 逆时针旋转到 end 的弧度
```

```
{return end - start + (end > start - eps ? 2 * pi : 0);}
bool InSimplePolygon(Point u, Point p[], int n/*double neg_inf*/)//判断点在简单多边形内，不包括边界，多边形点为逆时针
{
    double neg_inf = -1e20, angvu, angvp1, angvp2;
    Point v(0, u.y), p1, p2, tmp;
    int i, id;//距离u 最近交点对应线段起始点id
    bool flag = false;
    p[n] = p[0];
    //设u->v 为水平负向射线
/*    for(i = 0, neg_inf = 0; i < n; ++ i) neg_inf = min(neg_inf, p[i].x);
    neg_inf -= 100.0;    */
    v.x = neg_inf;
    for(i = 0; i < n; ++ i)
    {
        if(u.OnSeg(p[i], p[i + 1])) return false;
        if(!SegCross(u, v, p[i], p[i + 1], tmp)) continue;
        flag = true;
        if(tmp.x - v.x > eps) v.x = tmp.x, id = i;
    }
    if(!flag) return false;
    p1 = v == p[id] ? p[(id + n - 1) % n] : p[id];
    p2 = v == p[id + 1] ? p[(id + 2) % n] : p[id + 1];
    angvu = atan2(u.y - v.y, u.x - v.x);
    angvp1 = atan2(p1.y - v.y, p1.x - v.x);
    angvp2 = atan2(p2.y - v.y, p2.x - v.x);
    return AngCounterClock(angvu, angvp1) < AngCounterClock(angvp2, angvp1) - eps;
}
/**************************************************/
```

//简单多边形、凸多边形面积并

```
//不需要正规的三角剖分，用求多边形面积的思想，从一点出发连接多边形的边得到很多三
//角形，三角形有向边方向决定有向面积有正有负，相加得到多边形面积的正值或负值。
//把两个多边形都分成若干这样的三角形，求每对三角形的交，根据两三角形有向边顺逆时
//针关系确定相交面积的正负号，最后两多边形面积和减去相交面积。
double CPIA(Point a[], Point b[], int na, int nb)//ConvexPolygonIntersectArea
{
    Point p[maxisn], tmp[maxisn];
    int i, j, tn, sflag, eflag;
    a[na] = a[0], b[nb] = b[0];
    memcpy(p, b, sizeof(Point) * (nb + 1));
    for(i = 0; i < na && nb > 2; ++ i)
    {
        sflag = dcmp(a[i].cross(a[i + 1], p[0]));
        for(j = tn = 0; j < nb; ++ j, sflag = eflag)
        {
            if(sflag >= 0) tmp[tn ++] = p[j];
            eflag = dcmp(a[i].cross(a[i + 1], p[j + 1]));
            if((sflag ^ eflag) == -2)
                tmp[tn ++] = LineCross(a[i], a[i + 1], p[j], p[j + 1]);
        }
        memcpy(p, tmp, sizeof(Point) * tn);
        nb = tn, p[nb] = p[0];
    }
    if(nb < 3) return 0.0;
    return PolygonArea(p, nb);
}
double SPIA(Point a[], Point b[], int na, int nb)//SimplePolygonIntersectArea，要调用CPIA
//一般两个都写上，只调用SPIA
{
    int i, j;
    Point t1[4], t2[4];
    double res = 0, if_clock_t1, if_clock_t2;
    a[na] = t1[0] = a[0], b[nb] = t2[0] = b[0];
    for(i = 2; i < na; ++ i)
    {
        t1[1] = a[i - 1], t1[2] = a[i];
        if_clock_t1 = dcmp(t1[0].cross(t1[1], t1[2]));
        if(if_clock_t1 < -eps) std::swap(t1[1], t1[2]);
```

```
        for(j = 2; j < nb; ++ j)
        {
            t2[1] = b[j - 1], t2[2] = b[j];
            if_clock_t2 = dcmp(t2[0].cross(t2[1], t2[2]));
            if(if_clock_t2 < -eps) std::swap(t2[1], t2[2]);
            res += CPIA(t1, t2, 3, 3) * if_clock_t1 * if_clock_t2;
        }
    }
    return PolygonArea(a, na) + PolygonArea(b, nb) - res;//res 是面积交，CPIA 可同理求交
}

/***************************************************/
```

//简单多边形与圆面积交

```
//要用到Sqr(double x)，同时重载了Sqr(Point p)，两个都要。
inline double Sqr(const Point &p) {return p.dot(p);}
double LineCrossCircle(const Point &a, const Point &b, const Point &r, double R, Point &p1, Point &p2)
{
    Point fp = LineCross(r, Point(r.x + a.y - b.y, r.y + b.x - a.x), a, b);
    double rtol = r.Dis(fp), rtos = fp.OnSeg(a, b) ? rtol : min(r.Dis(a), r.Dis(b)), atob = a.Dis(b);
    double fptoe = sqrt(R * R - rtol * rtol) / atob;
    if(rtos > R - eps) return rtos;
    p1 = fp + (a - b) * fptoe;
    p2 = fp + (b - a) * fptoe;
    return rtos;
}
double SectorArea(const Point &r, const Point &a, const Point &b, double R)//不大于180 度扇形面积，r->a->b
逆时针
{
    double A2 = Sqr(r - a), B2 = Sqr(r - b), C2 = Sqr(a - b);
    return R * R * acos((A2 + B2 - C2) * 0.5 / sqrt(A2) / sqrt(B2)) * 0.5;
}
double TACIA(const Point &r, const Point &a, const Point &b, double R)//TriangleAndCircleIntersectArea，
逆时针，p[0]为圆心
{
    double adis = r.Dis(a), bdis = r.Dis(b);
    if(adis < R + eps && bdis < R + eps) return r.cross(a, b) * 0.5;
    Point ta, tb;
    if(r.InLine(a, b)) return 0.0;
    double rtos = LineCrossCircle(a, b, r, R, ta, tb);
    if(rtos > R - eps) return SectorArea(r, a, b, R);
    if(adis < R + eps) return r.cross(a, tb) * 0.5 + SectorArea(r, tb, b, R);
    if(bdis < R + eps) return r.cross(ta, b) * 0.5 + SectorArea(r, a, ta, R);
    return r.cross(ta, tb) * 0.5 + SectorArea(r, a, ta, R) + SectorArea(r, tb, b, R);
}
double SPICA(Point p[], int n, Point r, double R)//SimplePolygonIntersectCircleArea
//一般两个都写上，只调用SPICA
{
    int i;
    Point a, b;
    double res = 0, if_clock_t;
    p[n] = p[0];
    for(i = 1; i <= n; ++ i)
    {
        a = p[i - 1], b = p[i];
        if_clock_t = dcmp(r.cross(a, b));
        if(if_clock_t < 0) std::swap(a, b);
        res += TACIA(r, a, b, R) * if_clock_t;
    }
    return fabs(res);
}
```

//模拟退火代码模型

```
//POJ1379，范围内距离点集最近距离最远点样例
const int maxsam = 20;      //采样点个数
const int faillimit = 20;   //测试次数限制
const double leps = 1e-2;    //步长限制
const double depace = 0.9;  //步长缩减率
```

```
Point sam[maxsam];          //采样点
double CalMinDis(Point)//枚举所有点计算要求的量
void SA()
{
   int i, j;
   double pace;//初始步长要足够大
   Point tmp;
   for(i = 0; i < maxsam; ++ i)
   {
       sam[i].x = rand() / 32767.0 * X;
       sam[i].y = rand() / 32767.0 * Y;
       sam[i].mindis = CalMinDis(sam[i]);
   }
   for(pace = sqrt((double)X * Y); pace > leps; pace *= depace)//缩减步长
       for(i = 0; i < maxsam; ++ i)//枚举采样点
           for(j = 0; j < faillimit; ++ j)//采样点扩展
           {
               tmp.x = sam[i].x + cos((double)rand()) * pace;
               tmp.y = sam[i].y + cos((double)rand()) * pace;
               if(tmp.x <= X && tmp.y <= Y && tmp.x >= 0 && tmp.y >= 0)
               {
                   tmp.mindis = CalMinDis(tmp);
                   if(tmp.mindis > sam[i].mindis /*|| rand() / 30.0 < exp((tmp.mindis - sam[i].mindis) / 
pace)*/)
                       sam[i] = tmp;
               }
           }
   for(i = j = 0; i < maxsam; ++ i)
       if(sam[i].mindis > sam[j].mindis) j = i;
   printf("The safest point is (%.1f, %.1f).\n", sam[j].x, sam[j].y);
}
```

//有关圆，更多参考其他模板

```
/*************************************************************************/
```

//atan2 (-pi,pi]转区间覆盖处理

```
int ctp;
struct Cov
{
    double site;
    int se;
    bool operator<(const Cov &b)const
    {
        if(!dcmp(site - b.site)) return se > b.se;
        return site < b.site;
    }
} cover[maxn <<2];
void AngManage(double &x)//角度区间修正，(-pi, pi]
{
    while(x + pi < eps) x += 2 * pi;
    while(x - pi > eps) x -= 2 * pi;
}
void AddAnSeg(double start, double end)//圆心角转区间
{
    AngManage(start), AngManage(end);
    if(start - end > eps) AddAnSeg(start, pi), AddAnSeg(-pi + eps * 2, end);
    else
    {
        cover[ctp].site = start, cover[ctp].se = 1;++ ctp;
        cover[ctp].site = end, cover[ctp].se = -1;++ ctp;
    }
}
int SumCov(Cov cover[], int ctp)
{
    int i, cnt, ans;
    for(i = cnt = ans = 0; i < ctp; ++ i)
    {
        cnt += cover[i].se;
        ans = max(ans, cnt);
    }
    return ans;
}
/*************************************************************************/
```

//判断圆 i 在圆 j 中（包括内切）

```
inline bool IinJ(int i, int j, double ijdis)
{return dcmp(ra[i].r + ijdis - ra[j].r) <= 0;}
```

//判断圆 i 与圆 j 相交（包括外切）

```
inline bool IcutJ(int i, int j, double ijdis)
{return dcmp(ijdis - ra[i].r + ra[j].r) <= 0;}
```

//左圆 i 与右圆 j 的公切线

```
void CalCirCutCir(int i, int j)//左圆i 与右圆j 的公切线，计算的反正切皆为-pi~pi。
{
    double ijdis = CalDis(ra[i].x - ra[j].x, ra[i].y - ra[j].y);
    double xlij = atan2(ra[i].y - ra[j].y, ra[i].x - ra[j].x);//右心->左心反正切
    double xlji = atan2(ra[j].y - ra[i].y, ra[j].x - ra[i].x);//左心->右心反正切
    double asimj = asin((ra[i].r - ra[j].r) / ijdis);
    double asipj = asin((ra[i].r + ra[j].r) / ijdis);
    double at1 = AngManage(xlji - asipj), at2 = AngManage(xlji - asimj);
        //左上->右下切线的反正切、左上->右上切线的反正切
    double at3 = AngManage(xlji + asipj), at4 = AngManage(xlji + asimj);
        //左下->右上切线的反正切、左下->右下切线的反正切
    //ji 即左->右，+pj 即下->上，+mj 即下->下，反之则反。 Manage(at1), Manage(at2);
    //以第一象限左右为例推出计算切点切线的方法，可以推广到其他方位和象限。
```

```
}
```

//圆外点到圆切线

```
void TLTP(Point p, Point r, double R, Point &p1, Point &p2)
//p->p1 p->p2 为切线向量，但非切点。p1->p2 逆时针，绕点旋转法
{
   double ang = asin(R / p.Dis(r));

   if(!dcmp(R)) {p1 = p2 = r; return;}
   p1 = p.RotePoint(r, 2 * pi - ang);
   p2 = p.RotePoint(r, ang);
}
/*
void TLTP(Point p, Point r, double R, Point &p1, Point &p2)
//圆外点到圆切线，p1~p2 逆时针，定比分点法，p1、p2 为切点
{
   double tc = Sqr(R) / p.Dis(r);
   Point tmp = r + (p - r) * (tc / p.Dis(r));
   Point tx(tmp.x + p.y - r.y, tmp.y + r.x - p.x);
   double lin = (sqrt(Sqr(R) - Sqr(tc)) / tx.Dis(tmp));
   p1 = tmp - (tx - tmp) * lin;
   p2 = tmp + (tx - tmp) * lin;
}
*/
```

//圆的扫描线样例

```
//求一系列不相切不相交的圆最深嵌套
int LineNow, ltp, n, cnt[maxn];
struct Cir//圆
{
   int x;
   int y;
   int r;
}c[maxn];
struct Line//从左向右扫描节点
{
   int id;
   bool in;
   void Read(int id_, bool in_){id = id_, in = in_;}
   inline int GetSite()const{return c[id].x + (in ? -c[id].r : c[id].r);}
   bool operator<(const Line &b)const{return GetSite() < b.GetSite();}
}l[maxn << 1];
struct Node//从上至下排序节点
{
   int id;
   bool up;
   Node(){}
   Node(int id_, bool up_){id = id_, up = up_;}
   bool operator<(const Node &b)const
   {
       double y1 = c[id].y + sqrt(Sqr(c[id].r) - Sqr(LineNow - c[id].x)) * (up ? 1 : -1);
       double y2 = c[b.id].y + sqrt(Sqr(c[b.id].r) - Sqr(LineNow - c[b.id].x)) * (b.up ? 1 : -1);
       return dcmp(y1 - y2) ? y1 > y2 : up > b.up;
   }
};
set<Node> s;
set<Node>::iterator iti, itn;
void ReadData(int n)
{
   int i;
   for(ltp = i = 0; i < n; ++ i)
   {
       scanf("%d%d%d", &c[i].x, &c[i].y, &c[i].r);
       l[ltp ++].Read(i, true);
       l[ltp ++].Read(i, false);
   }
}
int MakeAns()
```

```
{
    int i, ans = 0;
    sort(l, l + ltp);
    s.clear();
    for(i = 0; i < ltp; ++ i)
    {
        LineNow = l[i].GetSite();
        if(!l[i].in)
        {
            s.erase(Node(l[i].id, true));
            s.erase(Node(l[i].id, false));
        }
        else
        {
            iti = itn = s.insert(Node(l[i].id, true)).first;
            itn ++;
            if(iti == s.begin() || itn == s.end()) cnt[l[i].id] = 1;
            else
            {
                iti --;
                if((*iti).id == (*itn).id) cnt[l[i].id] = cnt[(*iti).id] + 1;
                else cnt[l[i].id] = max(cnt[(*iti).id], cnt[(*itn).id]);
            }
            ans = max(ans, cnt[l[i].id]);
            s.insert(Node(l[i].id, false));
        }
    }
    return ans;
}
```

//三维几何

//三维坐标变换

```
//平移、旋转、倍增
//Matrix 结构体中进行每种类型操作的成员函数都返回一个Matrix 值，并不改变自身。
//x、y、z 分别为Mt[0][0]、Mt[0][1]、Mt[0][2]，运算前Mt[0][3]置1.
const double pi = acos(-1.0);
const double eps = 1e-6;
inline double pz(double x)
{return fabs(x) < eps ? 0.0 : x;}
struct Matrix
{
    double Mt[4][4];
    void init0(){memset(Mt, 0, sizeof(Mt));}//初始化0
    void init1()//初始化单位阵
    {init0(), Mt[0][0] = Mt[1][1] = Mt[2][2] = Mt[3][3] = 1;}
    Matrix(){init0();}//默认初始化0
    Matrix(int num)//按数值初始化为对角阵:Matrix a(1);
    {init0();Mt[0][0] = Mt[1][1] = Mt[2][2] = Mt[3][3] = num;}
    Matrix operator *(const Matrix &b)//重载乘法
    {
        int i, j, k;Matrix res;
        for(i = 0; i < 4; ++ i)
            for(j = 0; j < 4; ++ j)
                for(k = 0; k < 4; ++ k)
                    res.Mt[i][j] += Mt[i][k] * b.Mt[k][j];
        return res;
    }
    void read(double x, double y, double z)//赋值
    {init0(), Mt[0][0] = x, Mt[0][1] =y, Mt[0][2] = z, Mt[0][3] = 1;}
    void scan()    //输入
    {double x, y, z;scanf("%lf%lf%lf", &x, &y, &z);read(x, y, z);}
    void prin()    //输出
    {printf("%.2f %.2f %.2f\n", pz(Mt[0][0]), pz(Mt[0][1]), pz(Mt[0][2]));}
    Matrix Trans(double x, double y, double z)//坐标平移
    {
        Matrix b(1);
        b.Mt[3][0] = x, b.Mt[3][1] = y, b.Mt[3][2] = z;
        return *this * b;
    }
    Matrix Scale(double x, double y, double z)//坐标加倍
    {
        Matrix b(1);
        b.Mt[0][0] = x, b.Mt[1][1] = y, b.Mt[2][2] = z;
        return *this * b;
    }
    Matrix Rotate(double x, double y, double z, double deg)
    //绕轴旋转，从x、y、z 看向原点逆时针，deg 为弧度
    {
        Matrix b(1);
        double len = sqrt(x * x + y * y + z * z);
        double co = cos(deg), si = sin(deg);
        x /= len, y /= len, z /= len; //归一化处理
        b.Mt[0][0] = (1 - co) * x * x + co;
        b.Mt[0][1] = (1 - co) * x * y + si * z;
        b.Mt[0][2] = (1 - co) * x * z - si * y;
        b.Mt[1][0] = (1 - co) * y * x - si * z;
        b.Mt[1][1] = (1 - co) * y * y + co;
        b.Mt[1][2] = (1 - co) * y * z + si * x;
        b.Mt[2][0] = (1 - co) * z * x + si * y;
        b.Mt[2][1] = (1 - co) * z * y - si * x;
        b.Mt[2][2] = (1 - co) * z * z + co;
        return *this * b;
    }
    Matrix Rep(int p)//矩阵连乘快速p 次幂
    {
```

```
        Matrix b = *this, res(1);
        if(p == 0) return res;
        if(p == 1) return b;
        while(p > 1)
        {
            if(p & 1) res = res * b;
            b = b * b;
            p >>= 1;
        }
        return b * res;
    }
};
```

//分数形式的三维坐标点计算

```
struct Point3   //三维坐标点
{
    LL x, y, z;
    Point3(){x = y = z = 0;}
    Point3(LL a, LL b, LL c)
    {x = a, y = b, z = c;}
    Point3 operator*(const Point3 &p)const//叉积
    {
        return Point3(y * p.z - z * p.y,
                      z * p.x - x * p.z,
                      x * p.y - y * p.x);
    }
    Point3 operator-(const Point3 &p)const
    {return Point3(x - p.x, y - p.y, z - p.z);}
    Point3 operator+(const Point3 &p)const
    {return Point3(x + p.x, y + p.y, z + p.z);}
    Point3 operator-()const
    {return Point3(-x, -y, -z);}
    LL dot(const Point3 &p)const    //点积
    {return x * p.x + y * p.y + z * p.z;}
};
struct Dis                  //能用分数表示距离的距离结构体
{
    LL fz, fm;//分子分母
    void yf()
    {
        if(fz == 0) {fm = 1;return;}
        LL t = gcd(fz, fm);
        fz /= t;
        fm /= t;
    }
    bool operator<(const Dis &p)const
    {return fz * p.fm < p.fz * fm;}
};
bool Paral(Point3 a, Point3 b, Point3 c, Point3 d)//判平行
{
    Point3 tmp = (b - a) * (d - c);
    return !tmp.dot(tmp);
}
bool JudgeCZ(Point3 a, Point3 b, Point3 c, Point3 d)
//判垂足是否在线段上，点到线的情况传入两相同点即可，否则为线段公垂线情况
{
    LL A1, B1, C1, A2, B2, C2;
    A1 = (b - a).dot(b - a);
    B1 = -(d - c).dot(b - a);
    C1 = -(a - c).dot(b - a);
    A2 = (b - a).dot(d - c);
    B2 = -(d - c).dot(d - c);
    C2 = -(a - c).dot(d - c);
    double bl1 = dcmp(A2 * B1 - A1 * B2) ? ((double)C2 * B1 - C1 * B2) / (A2 * B1 - A1 * B2) : (A1 ? (double)C1 / A1 : (A2 ? (double)C2 / A2 : 0));
    double bl2 = dcmp(B2 * A1 - B1 * A2) ? ((double)C2 * A1 - C1 * A2) / (B2 * A1 - B1 * A2) : (B1 ? (double)C1 / B1 : (B2 ? (double)C2 / B2 : 0));
    return bl1 > -eps && bl1 < 1 + eps && bl2 > -eps && bl2 < 1 + eps;
```

```
}
Dis CalPtoL(Point3 p, Point3 a, Point3 b)//点线距离
{
    Point3 t = (a - p) * (b - a);
    Dis tmp;
    tmp.fz = t.dot(t);
    tmp.fm = (b - a).dot(b - a);
    tmp.yf();
    return tmp;
}
Dis CalPtoP(Point3 a, Point3 b)//两点距离
{
    Dis tmp;
    tmp.fz = (b - a).dot(b - a);
    tmp.fm = 1;
    tmp.yf();
    return tmp;
}
Dis CalLtoL(Point a, Point b, Point c, Point d)//两线距离
{
//    if(!Paral(a, b, c, d) && JudgeCZ(a, b, c, d))
    Point3 t = (b - a) * (d - c);
    ans.fz = Sqr((c - a).dot(t));
    ans.fm = t.dot(t);
    ans.yf();
    return ans;
}
```

//多面体切割（附带各种空间点、面操作）

```
//非凸多棱柱、凸多面体切割，O(n^2)三角剖分，三维坐标点多种操作
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#include<math.h>
#include<vector>
#include<list>
#include<algorithm>
#include<iostream>
using namespace std;
const int maxn = 3011;
const double eps = 1e-8;
inline int dcmp(double x)
{
    return x > eps ? 1 : (x < -eps ? -1 : 0);
}
inline double pz(double x)
{
    return dcmp(x) ? x : 0;
}
/*************************************************************************/
//定义三维点
struct Point3
{
    double x, y, z;
    Point3()
    {
        x = y = z = 0;
    }
    Point3(double a, double b, double c)
    {
        x = a, y = b, z = c;
    }
    Point3 cross(Point3 p)
    {
        return Point3(y * p.z - p.y * z,
                      z * p.x - x * p.z,
                      x * p.y - y * p.x);
    }
```

```
double dot(Point3 p)
{
    return x * p.x + y * p.y + z * p.z;
}
Point3 operator-(const Point3 &p)const
{
    return Point3(x - p.x, y - p.y, z - p.z);
}
Point3 operator-()const
{
    return Point3(-x, -y, -z);
}
Point3 operator+(const Point3 &p)const
{
    return Point3(x + p.x, y + p.y, z + p.z);
}
Point3 operator*(const double b)const
{
    return Point3(x * b, y * b, z * b);
}
bool operator==(const Point3 &p)const
{
    return !dcmp(x - p.x) && !dcmp(y - p.y) && !dcmp(z - p.z);
}
bool operator!=(const Point3 &p)const
{
    return !(*this == p);
}
bool operator<(const Point3 &p)const
{
    if(!dcmp(x - p.x))
    {
        if(!dcmp(y - p.y))
            return dcmp(z - p.z) < 0;
        return dcmp(y - p.y) < 0;
    }
    return dcmp(x - p.x) < 0;
}
bool operator>(const Point3 &p)const
{
    return p < *this;
}
bool operator>=(const Point3 &p)const
{
    return !(*this < p);
}
bool operator<=(const Point3 &p)const
{
    return !(*this > p);
}
Point3 fxl(Point3 b, Point3 c)//平面法向量
{
    return (*this - b).cross(b - c);
}
double Dis(Point3 b)
{
    return sqrt((*this - b).dot(*this - b));
}
double vlen()
{
    return sqrt(dot(*this));
}
bool PinLine(Point3 b, Point3 c)//三点共线
{
    return fxl(b, c).vlen() < eps;
}
bool PonPlane(Point3 b, Point3 c, Point3 d)//四点共面
{
    return !dcmp(fxl(b, c).dot(d - *this));
```

```
    }
    bool PonSeg(Point3 b, Point3 c)//点在线段上，包括端点
    {
        return !dcmp((*this - b).cross(*this - c).vlen()) &&
               (*this - b).dot(*this - c) <= 0;
    }
    bool PinSeg(Point3 b, Point3 c)//点在线段上，不包括端点
    {
        return PonSeg(b, c) && *this != b && *this != c;
    }
    double PtoLine(Point3 b, Point3 c)//点到直线距离
    {
        return (*this - b).cross(c - b).vlen() / b.Dis(c);
    }
    double PtoPlane(Point3 b, Point3 c, Point3 d)//点到平面距离
    {
        return fabs(b.fxl(c, d).dot(*this - b)) / b.fxl(c, d).vlen();
    }
};
/*************************************************************************/
//定义平面+空间平面凸包
struct Plane
{
    double a, b, c, d;
    bool outplane;//计入表面积的面
    vector<Point3> p;
    Plane()
    {
        a = b = c = d = 0, outplane = false;
        p.clear();
    }
    inline void init(double a_, double b_, double c_, double d_)
    {
        a = a_, b = b_, c = c_, d = d_;
        p.clear();
    }
    inline void init(Point3 pa, Point3 pb, Point3 pc)
    {
        Point3 t = (pa - pb).cross(pa - pc);
        a = t.x, b = t.y, c = t.z;
        d = -(pa.x * t.x + pa.y * t.y + pa.z * t.z);
        p.clear();
    }
    Plane(double a_, double b_, double c_, double d_)
    {
        init(a_, b_, c_, d_);
    }
    Plane(Point3 pa, Point3 pb, Point3 pc)
    {
        init(pa, pb, pc);
    }
    double PtoPlane(const Point3 &pa)const//点面距离
    {
        return fabs(Sub(pa)) / sqrt(a * a + b * b + c * c);
    }
    double Sub(const Point3 &p)const//点代入方程
    {
        return p.x * a + p.y * b + p.z * c + d;
    }
    Point3 PcrossPlane(Point3 a, Point3 b)
    {
        return a + (b - a) * (PtoPlane(a) / (PtoPlane(a) + PtoPlane(b)));//定比分点求法，所以 a、b 在同侧的时
候要变减号
    }
    int Parallel(Plane pl)//面平行
    {
        if(!dcmp(a * pl.b - b * pl.a) && !dcmp(a * pl.c - c * pl.a) && !dcmp(b * pl.c - c * pl.b))
            return dcmp(pl.Sub(p[0])) > 0 ? 1 : -1;
        return 0;
```

```
    }
    bool Cut(Plane &pl)//平面切割
    {
        switch(Parallel(pl))
        {
        case -1:
            return true;
        case 1:
            return false;
        }
        int i, j, k, n = p.size();
        bool flag1, flag2;
        Point3 p1, p2;
        vector<Point3> ret;
        for(i = flag1 = flag2 = 0; i < n; ++ i)
        {
            flag1 |= pl.Sub(p[i]) < 0;
            flag2 |= pl.Sub(p[i]) > 0;
        }
        if(flag1 != flag2) return flag1;
        if(!flag1) return true;
        for(i = 0; pl.Sub(p[i]) >= 0; ++ i);
        for(; pl.Sub(p[i]) < 0; i = (i + 1) % n);
        for(j = i; pl.Sub(p[j]) >= 0; j = (j + 1) % n);
        for(k = j; k != i; k = (k + 1) % n)
            ret.push_back(p[k]);
        p1 = pl.PcrossPlane(p[i], p[(i + n - 1) % n]);
        p2 = pl.PcrossPlane(p[j], p[(j + n - 1) % n]);
        ret.push_back(p1), ret.push_back(p2);
        p = ret;
        pl.p.push_back(p1), pl.p.push_back(p2);
        return true;
    }
    void Graham()//求空间平面凸包
    {
        int len, i, n, top = 1;
        vector<Point3> res;
        Point3 fx(a, b, c);
        sort(p.begin(), p.end());
        vector<Point3>::iterator iter = unique(p.begin(), p.end());
        p.erase(iter, p.end());
        n = p.size();
        res.push_back(p[0]), res.push_back(p[1]);
        for(i = 2; i < n; ++ i)
        {
            while(top && dcmp((res[top] - res[top - 1]).cross(p[i] - res[top]).dot(fx)) <= 0)
                res.pop_back(), -- top;
            res.push_back(p[i]), ++ top;
        }
        len = top;
        res.push_back(p[n - 2]), ++ top;
        for(i = n - 3; i >= 0; -- i)
        {
            while(top != len && dcmp((res[top] - res[top - 1]).cross(p[i] - res[top]).dot(fx)) <= 0)
                res.pop_back(), -- top;
            res.push_back(p[i]), ++ top;
        }
        res.pop_back();
        p = res;
    }
    double PolygonArea()//空间平面凸包面积
    {
        int n = p.size();
        double ret = 0;
        for(int i = 2; i < n; ++ i)
            ret += (p[i - 1] - p[0]).cross(p[i] - p[0]).vlen();
        return ret * 0.5;
    }
};
```

```
/**************************************************************************/
//定义变量
struct Polyhedron//立方体面集合
{
   list<Plane> pl;
};
list<Polyhedron> pol;
int n, h, m;//点个数，高度，切割次数
struct PointChain
{
   Point3 p;
   int pre, nex;
   bool outside;//标记发向nex 的边为外部边，抬起的面计入总面积
} PC[maxn];
/**************************************************************************/
//判相交，辅助判断三角剖分合法性
inline double det(double x1, double y1, double x2, double y2)
{
   return x1 * y2 - x2 * y1;
}
inline double cross2(Point3 a, Point3 b, Point3 c)//平面叉积
{
   return det(b.x - a.x, b.y - a.y, c.x - a.x, c.y - a.y);
}
inline bool SegCross(Point3 u1, Point3 u2, Point3 v1, Point3 v2)
{
   return ((dcmp(cross2(u1, u2, v1)) ^ dcmp(cross2(u1, u2, v2))) == -2 &&
           (dcmp(cross2(v1, v2, u1)) ^ dcmp(cross2(v1, v2, u2))) == -2) ||
          ((v1.PinSeg(u1, u2) || v2.PinSeg(u1, u2)) &&
           (u1.PonSeg(v1, v2) || u2.PonSeg(v1, v2))) ||
          ((v1.PonSeg(u1, u2) || v2.PonSeg(u1, u2)) &&
           (u1.PinSeg(v1, v2) || u2.PinSeg(v1, v2)));
}
/**************************************************************************/
bool CanDo(int s, int e)//判断角是否可切
{
   int tp = PC[s].nex;
   while(tp != s)
   {
       if(SegCross(PC[s].p, PC[e].p, PC[tp].p, PC[PC[tp].pre].p))
           return false;
       tp = PC[tp].nex;
   }
   return true;
}
void AddPolyhedron(int A, int B, int C)//三角剖分添加三棱柱
{
   Polyhedron t;
   Plane pl;
   Point3 a = PC[A].p, b = PC[B].p, c = PC[C].p;
   Point3 a_ = Point3(a.x, a.y, h), b_ = Point3(b.x, b.y, h), c_ = Point3(c.x, c.y, h);
   pl.init(a, b, c);
   pl.p.push_back(a);
   pl.p.push_back(b);
   pl.p.push_back(c);
   pl.outplane = true;
   t.pl.push_front(pl);
   pl.init(a_, b_, c_);
   pl.p.push_back(a_);
   pl.p.push_back(b_);
   pl.p.push_back(c_);
   t.pl.push_front(pl);
   pl.init(a, b, b_);
   pl.p.push_back(a);
   pl.p.push_back(b);
   pl.p.push_back(b_);
   pl.p.push_back(a_);
   pl.outplane = PC[A].outside, PC[A].outside = false;
   t.pl.push_front(pl);
```

```
    pl.init(b, c, c_);
    pl.p.push_back(b);
    pl.p.push_back(c);
    pl.p.push_back(c_);
    pl.p.push_back(b_);
    pl.outplane = PC[B].outside, PC[B].outside = false;
    t.pl.push_front(pl);
    pl.init(c, a, a_);
    pl.p.push_back(c);
    pl.p.push_back(a);
    pl.p.push_back(a_);
    pl.p.push_back(c_);
    if(PC[C].nex == A && PC[C].outside)
        pl.outplane = true, PC[C].outside = false;
    else pl.outplane = false;
    t.pl.push_front(pl);
    pol.push_front(t);
}
void Triangulation()//循环枚举相邻三点划分三角形
{
    int tp = 0;
    while(PC[PC[tp].nex].nex != tp)
    {
        while(dcmp(cross2(PC[tp].p, PC[PC[tp].nex].p, PC[PC[PC[tp].nex].nex].p)) <= 0 ||
            !CanDo(tp, PC[PC[tp].nex].nex)) tp = PC[tp].nex;
        AddPolyhedron(tp, PC[tp].nex, PC[PC[tp].nex].nex);
        PC[PC[PC[tp].nex].nex].pre = tp;
        PC[tp].nex = PC[PC[tp].nex].nex;
    }
}
void init()//读入数据，建立链表以备三角剖分
{
    double x, y;
    int i, tp;
    pol.clear();
    for(i = 0; i < n; ++ i)
    {
        scanf("%lf%lf", &x, &y);
        PC[i].p = Point3(x, y, 0);
        PC[i].pre = i - 1;
        PC[i].nex = i + 1;
        PC[i].outside = true;
    }
    PC[0].pre = n - 1, PC[n - 1].nex = 0;
//    for(i = -2, tp = 0; i <= n; ++ i, tp = PC[tp].nex)
//        while(PC[tp].p.PinLine(PC[PC[tp].nex].p, PC[PC[PC[tp].nex].nex].p))
//            PC[PC[PC[tp].nex].nex].pre = tp, PC[tp].nex = PC[PC[tp].nex].nex;
    Triangulation();
}
/*******************************************************************************/
list<Polyhedron>::iterator iti;
list<Plane>::iterator itj;
double CalSumVol()//求总体积
{
    double ans = 0;
    Point3 tmp;
    for(iti = pol.begin(); iti != pol.end(); ++ iti)
    {
        for(itj = (*iti).pl.begin(), tmp = (*itj).p[0]; itj != (*iti).pl.end(); ++ itj)
            ans += (*itj).PolygonArea() * (*itj).PtoPlane(tmp);
    }
    return ans / 3;
}
double CalSumArea()//求总面积
{
    double ans = 0;
    for(iti = pol.begin(); iti != pol.end(); ++ iti)
        for(itj = (*iti).pl.begin(); itj != (*iti).pl.end(); ++ itj)
            if((*itj).outplane) ans += (*itj).PolygonArea();
```

```
    return ans;
}

void MakeAns()
{
    double a, b, c, d;
    bool flag;
    while(m --)
    {
        scanf("%lf%lf%lf%lf", &a, &b, &c, &d);
        for(iti = pol.begin(); iti != pol.end();)
        {
            Plane tmp(a, b, c, d);
            tmp.outplane = true;
            for(itj = (*iti).pl.begin(), flag = false; itj != (*iti).pl.end();)
            {
                if(!(*itj).Cut(tmp))
                {
                    list<Plane>::iterator tmpj = itj;
                    ++ itj;
                    (*iti).pl.erase(tmpj);
                }
                else flag = true, ++ itj;
            }
            if(!flag)
            {
                list<Polyhedron>::iterator tmpi = iti;
                ++ iti;
                pol.erase(tmpi);
            }
            else if(tmp.p.size())
            {
                tmp.Graham();
                (*iti).pl.push_front(tmp);
                ++ iti;
            }
            else ++ iti;
        }
        printf("%.3f %.3f\n", CalSumVol(), CalSumArea());
    }
}
```