几何

圆的反演变换

给定反演圆(O,r),任意一点P和其反演点P'的关系为$OP*OP'=r^2$,且OPP'三点共线.

性质

除反演中心O以外,反演点唯一,且PP'互为反演点,反演圆上的点的反演点为自身
过O的任意直线反演后还是原直线
过O的任意圆反演后为不过O的直线,且直线平行于该圆在O的切线
不过O的任意直线反演后为过O的圆,且该圆在O的切线平行于原直线,与上一条互逆
不过O的任意圆反演后为不过O的圆
反演具有保角性,即原图在P点的夹角等于反演后在P'的夹角
正交的圆反演后仍正交,相切的圆反演后仍相切,若原切点为O,则反演后为一组平行线(直线可看成半径无穷大的圆)

对于不过O的圆(C_1,r_1),其反演圆(C_2,r_2)的半径$r_2=0.5(\frac{1}{OC_{1-r_1}}-\frac{1}{OC_{1+r_1}})r^2$

皮克定理

顶点均是整点的简单多边形，其面积S=内部格点数n + 边上格点数s的1/2 - 1

欧拉公式

凸多面体或平面图：顶点数+面数-边数=2
平面区域个数(面数)=边数-顶点数+联通块个数(上公式默认为1)+1

圆内接四边形面积公式

设四边形四边边长为a,b,c,d,设$p=\frac{a+b+c+d}{2}$
则其面积$S=\sqrt{(p-a)(p-b)(p-c)(p-d)}$
圆形上的扇形面积公式为$r^2\theta/2$,因椭圆面积公式为πab,故椭圆上扇形面积公式为$ab\theta/2$,这里的θ为映射到外接圆上的角度,即将y轴坐标缩放为等圆后的角度

切比雪夫距离

定义为各维向量的差值的最大值 $d=max(|x_1-x_2|,|y_1-y_2|)$
与曼哈顿距离的转换:
曼哈顿距离:$|x_1-x_2|+|y_1-y_2|=max(x_1-x_2+y_1-y_2,x_2-x_1+y_1-y_2,x_1-x_2+y_2-y_1,x_2-x_1+y_2-y_1)$
切比雪夫距离:$max(|x_1-x_2|,|y_1-y_2|)=max(x_1-x_2,x_2-x_1,y_1-y_2,y_2-y_1)$
于是作等价变化$x=\frac{x+y}{2},y=\frac{x-y}{2}$即可

重心

对于质地均匀的圆或圆环,重心在圆心
对于质量都在点上的多边形,点P_i的质量为w_i,那么重心就是$\frac{\sum P_i w_i}{w_i}$
对于质地均匀的三角形,重心为端点的平均值
对于质地均匀的多边形,把图形三角剖分,求出各三角形的重心,以这些重心为点,点权为三角形面积,转为质量都在点上的多边形
如果质地不均匀,积分

other

acos函数返回值在[0,pi]之间
atan()用来计算参数x 的反正切值 返回(-PI/2,PI/2) 之间的计算结果,误差较大
atan2(double y,double x)计算y/x,返回(-pi,pi],误差较大
asin 返回[- PI/2,PI/2] 之间
double精度15-16位，long double 18-19位
输出0的时候注意不要输出-0.0000
sqrt里面用int容易爆,注意加eps防止负数开根号
判断点是否是线段的端点时、尽量判距离，不要用点乘或叉乘，有精度误差
二分三分的上下界注意精度问题，尽量精准
投影变换不改变重心,也就是说,图形初始重心为A,进过投影变换f后,图形新的重心为B,那么有$B = f(A)$,即B是A投影变换过后的点(那么有,初始关于重心对称的点,不管怎么投影,仍会关于重心对称)
范围在R以内的整点构成的凸包的大小是$R^{2/3}$级别的
纯随机的点构成的凸包大小小于100

```
const double eps = 1e-8;
const double pi = acos(-1);
inline int sgn(double x){
    if(x < -eps) return -1;
    else if(x > eps) return 1;
    else return 0;
}
struct Point {
    double x,y;
    Point(double _x = 0.0,double _y = 0.0): x(_x),y(_y) {}
    Point operator + (const Point &b) const {
        return Point(x + b.x,y + b.y);
    }
    Point operator - (const Point &b) const {
        return Point(x - b.x,y - b.y);
    }
    double operator * (const Point &b) const { //点乘
        return (x * b.x + y * b.y);

    }
    double operator ^ (const Point &b) const { //叉乘，判断点的相对位置关系 左正右负
        return (x * b.y - y * b.x);
    }
    Point operator * (double b) {
        return Point(x * b,y * b);
    }
    bool operator < (const Point &b) const { //水平序比较
        if(sgn(x - b.x) == 0) return y < b.y;
        else return x < b.x;
    }
    bool operator == (const Point &b) const {
        return sgn(x - b.x) == 0 && sgn(y - b.y) == 0;
    }
    Point rot(double ang) { //旋转，输入角度
        return Point(x * cos(ang) - y * sin(ang),x * sin(ang) + y * cos(ang));
    }
    Point rot(double g1,double g2) { //旋转，输入正弦值，余弦值
        return Point(x * g2 - y * g1,x * g1 + y * g2);
    }
    double norm() { //求模
        return sqrt(x * x + y * y);
    }
```

```
    }
    Point unit() { //取单位向量
        if(sgn(x) == 0 && sgn(y) == 0) return Point(0.0,0.0);
        double ll = norm();
        return Point(x / ll,y / ll);
    }
};
struct Line {//两点式
    Point s,e;
    Line(){}
    Line(Point _s,Point _e){
        s = _s;
        e = _e;
    }
    //求两直线交点，-1重合，0相交，1平行
    pair<int,Point> operator &(Line b){
        if(sgn((s - e) ^ (b.s - b.e)) == 0){
            if(sgn((s - b.e) ^ (b.s - b.e)) == 0) return make_pair(-1,s);
            else return make_pair(1,s);
        }
        double t = ((s - b.s) ^ (b.s - b.e)) / ((s - e) ^ (b.s - b.e));
        return make_pair(0,Point(s.x + (e.x - s.x) * t,s.y + (e.y - s.y) * t));
    }
};

//极角排序 关于(0,0)点
bool cmp(const Point &A,const Point &B){
    return atan2(B.y,B.x) - atan2(A.y,A.x) > eps;//计算极角法
    //精度不足时可考虑在角度小于eps时进行叉乘判断 B ^ A < 0
}

double dist(Point a,Point b){
    return (a - b).norm();
}
//判断点p在线段l上,包含端点
bool isPointOnSegment(Point &p,Line &l){
    return sgn((p - l.s) ^ (l.s - l.e)) == 0 && sgn((p.x - l.s.x) * (p.x - l.e.x)) <= 0
    && sgn((p.y - l.s.y) * (p.y - l.e.y)) <= 0;
}
//判断点p在直线l上
bool isPointOnLine(Point &p,Line &l){
    return sgn((p - l.s) ^ (l.s - l.e)) == 0;
}
//判断两线段相交
bool seg_seg_inter(Line seg1,Line seg2){
    return
    sgn(max(seg1.s.x,seg1.e.x) - min(seg2.s.x,seg2.e.x)) >= 0 &&
    sgn(max(seg2.s.x,seg2.e.x) - min(seg1.s.x,seg1.e.x)) >= 0 &&
    sgn(max(seg1.s.y,seg1.e.y) - min(seg2.s.y,seg2.e.y)) >= 0 &&
    sgn(max(seg2.s.y,seg2.e.y) - min(seg1.s.y,seg1.e.y)) >= 0 &&
    sgn((seg2.s - seg1.e) ^ (seg1.s - seg1.e)) * sgn((seg2.e - seg1.e)
    ^ (seg1.s - seg1.e)) <= 0 &&
    sgn((seg1.s - seg2.e) ^ (seg2.s - seg2.e)) * sgn((seg1.e - seg2.e)
    ^ (seg2.s - seg2.e)) <= 0;
}
//判断直线与线段相交
bool seg_line_inter(Line l,Line seg){
    return sgn((seg.s - l.e) ^ (l.s - l.e)) * sgn((seg.e - l.e) ^ (l.s - l.e)) <= 0;
}
//点到直线距离，返回垂足
Point Point_to_Line(Point p,Line l){
    double t = ((p - l.s) * (l.e - l.s)) / ((l.e - l.s) * (l.e - l.s));
    return Point(l.s.x + (l.e.x - l.s.x) * t,l.s.y + (l.e.y - l.s.y) * t);
}
```

```
//点到线段距离，返回点到直线最近点
Point Point_to_Seg(Point p,Line seg){
    double t = ((p - seg.s) * (seg.e - seg.s)) / ((seg.e - seg.s) * (seg.e - seg.s));
    if(t >= 0 && t <= 1) return Point(seg.s.x + (seg.e.x - seg.s.x) * t,seg.s.y
    + (seg.e.y - seg.s.y) * t);
    else if(sgn(dist(p,seg.s) - dist(p,seg.e)) <= 0) return seg.s;
    else return seg.e;
}
//求向量夹角，小于等于pi
double angle(Point vA,Point vB){
    double tmp = vA.norm() * vB.norm();
    if(sgn(tmp) != 0) return acos((vA * vB) / tmp);
    else return 0.0;
}
//返回vA到vB的逆时针角度大小
double angle(Point vA,Point vB){
    double ang_1 = atan2(vA.y,vA.x);
    double ang_2 = atan2(vB.y,vB.x);
    ang_2 -= ang_1;
    if(sgn(ang_2) == -1) ang_2 += pi * 2.0;
    return ang_2;
}

//判断两圆关系，-1相离，0相交，1第一个圆内含第二个，2第二个圆内含第一个,3重合，4外切，
5内切第一个在内，6内切第二个在内
int dot_to_circle(Point o1,double r1,Point o2,double r2)
{
    if(sgn(dist(o1,o2)) == 0 && sgn(r1 - r2) == 0) return 3;
    int k = sgn(dist(o1,o2) - r1 - r2);
    if(k < 0)
    {
        k = sgn(dist(o1,o2) - fabs(r1 - r2));
        if(k > 0) return 0;
        if(k == 0)
        {
            if(sgn(r1 - r2) > 0) return 6;
            else return 5;
        }
        if(sgn(r1 - r2) > 0) return 1;
        else return 2;
    }
    else if(k == 0) return 4;
    else return -1;
}
//两圆面积交
double area_of_circle(Point o1,double r1,Point o2,double r2)
{
    int k = dot_to_circle(o1,r1,o2,r2);
    if(k == -1 || k == 4) return 0.0;
    else if(k != 0) return pi * min(r1,r2) * min(r1,r2);
    double d = dist(o1,o2);
    double ang_1 = acos((d * d + r1 * r1 - r2 * r2) / (2.0 * d * r1));
    double ang_2 = acos((d * d + r2 * r2 - r1 * r1) / (2.0 * d * r2));
    return ang_1 * r1 * r1 + ang_2 * r2 * r2 - r1 * d * sin(ang_1);
}
//求两圆交点,需先判断两圆相交
void point_of_circle(Point o1,double r1,Point o2,double r2,Point &p1,Point &p2)
{
    double l = dist(o1,o2);
    double d1 = (l * l - r2 * r2 + r1 * r1) / (2.0 * l);
    double d2 = sqrt(max(0.0,r1 * r1 - d1 * d1));
    Point mid = o1 + (o2 - o1).unit() * d1;
    Point vv = (o2 - o1).rot(pi / 2.0).unit() * d2;
    p1 = mid + vv;
```

```
    p2 = mid - vv;
}

//过点p到圆的切线,返回切线条数,pl表示切线的法向量
int getTangents(Point p,Point o,double r,Point pl[]) {
    Point u = o - p;
    double dist = u.norm();
    if(dist < r) return 0;
    else if(sgn(dist - r) == 0) {
        pl[0] = u.rot(pi * 0.5);
        return 1;
    } else {
        double ang = asin(r / dist);
        pl[0] = u.rot(-ang);
        pl[1] = u.rot(+ang);
        return 2;
    }
}


//求两圆公切线,返回切线条数和切点，-1表示无限条
Point cir_rot(Point o,double r,double ang) {
    return Point(o.x + r * cos(ang),o.y + r * sin(ang));
}
int getTangents(Point o1,Point o2,double r1,double r2,Point a[],Point b[]) {
    int cnt = 0;
    if(r1 < r2) {
        swap(o1,o2);
        swap(r1,r2);
        swap(a,b);
    }
    double d = (o1 - o2).norm();
    double sr = r1 + r2,dr = r1 - r2;
    if(sgn(d - dr) < 0) return 0;//两圆内含
    double base = atan2((o2 - o1).y,(o2 - o1).x);
    if(sgn(d) == 0 && sgn(dr) == 0) return -1;//两圆重合
    if(sgn(d - dr) == 0) {  //两圆内切
        a[cnt] = cir_rot(o1,r1,base);
        b[cnt++] = cir_rot(o2,r2,base);
        return 1;
    }
    //有外切线
    double ang = acos(dr / d);
    a[cnt] = cir_rot(o1,r1,base + ang);
    b[cnt++] = cir_rot(o2,r2,base + ang);
    a[cnt] = cir_rot(o1,r1,base - ang);
    b[cnt++] = cir_rot(o2,r2,base - ang);
    if(sgn(d - sr) == 0) { //两圆外切
        a[cnt] = cir_rot(o1,r1,base);
        b[cnt++] = cir_rot(o2,r2,base + pi);
    } else if(sgn(d - sr) > 0){ //两圆外离
        double ang2 = acos(sr / d);
        a[cnt] = cir_rot(o1,r1,base + ang2);
        b[cnt++] = cir_rot(o2,r2,base + ang2 + pi);
        a[cnt] = cir_rot(o1,r1,base - ang2);
        b[cnt++] = cir_rot(o2,r2,base - ang2 + pi);
    }
    return cnt;
}


//圆(O,r)与直线(线段)l相交，num表示交点数，res存储交点
void Circle_cross_Segment(Point o,double r,Line l,Point res[],int &num)
{
```

```
    double dx = l.e.x - l.s.x;
    double dy = l.e.y - l.s.y;
    double A = dx * dx + dy * dy;
    double B = 2.0 * dx * (l.s.x - o.x) + 2.0 * dy * (l.s.y - o.y);
    double C = (l.s.x - o.x) * (l.s.x - o.x) + (l.s.y - o.y) * (l.s.y - o.y) - r * r;
    double delta = B * B - 4.0 * A * C;
    num = 0;
    if(sgn(delta) < 0) return;
    delta = sqrt(max(0.0,delta));
    double k1 = (-B - delta) / (2.0 * A);
    double k2 = (-B + delta) / (2.0 * A);
    //if(sgn(k1 - 1.0) <= 0 && sgn(k1) >= 0) //线段相交判断
    res[num++] = Point(l.s.x + k1 * dx,l.s.y + k1 * dy);
    //if(sgn(k2 - 1.0) <= 0 && sgn(k2) >= 0) //线段相交判断
    res[num++] = Point(l.s.x + k2 * dx,l.s.y + k2 * dy);
}

//三角形ABO与圆(O, r)的面积交
double Triangel_cross_Circle(Point A, Point B, Point O, double r) {
    double a,b,c,x,y,s = 0.5 * ((A - O) ^ (B - O));
    a = (B - O).norm();
    b = (A - O).norm();
    c = (A - B).norm();
    if(a <= r && b <= r) return s;
    else if(a < r && b >= r) {
        x = ((A - B) * (O - B) + sqrt(c * c * r * r - sqr((A - B) ^ (O - B)))) / c;
        return asin(s * (c - x) * 2.0 / c / b / r) * r * r * 0.5 +s * x / c;
    } else if(a >= r && b < r) {
        y = ((B - A) * (O - A) + sqrt(c * c * r * r - sqr((B - A) ^ (O - A)))) / c;
        return asin(s * (c - y) * 2.0 / c / a / r) * r * r * 0.5 +s * y / c;
    } else {
        if(fabs(2.0 * s) >= r * c || (B - A) * (O - A) <= 0 || (A - B) * (O - B) <= 0) {
            if((A - O) * (B - O) < 0) {
                if(((A - O) ^ (B - O)) < 0) return (-pi - asin(s * 2.0 / a / b)) * r * r * 0.5;
                else return (pi - asin(s *2.0 / a / b)) * r * r * 0.5;
            } else return asin(s * 2 / a / b) * r * r * 0.5;
        } else {
            x = ((A - B) * (O - B) + sqrt(c * c * r * r - sqr((A - B) ^ (O - B)))) / c;
            y = ((B - A) * (O - A) + sqrt(c * c * r * r - sqr((B - A) ^ (O - A)))) / c;
            return (asin(s * (1 - x / c) * 2 / r / b) + asin(s * (1 - y / c) * 2 / r / a)) * r * r * 0.5 + s * ((y + x) / c - 1);
        }
    }
}

//多边形与圆面积交
double Polygon_intersect_Circle(Point ploy[],int n,Point o,double r)
{
    ploy[n] = ploy[0];
    double res = 0.0;
    for(int i = 0;i < n; i++) res += Triangel_cross_Circle(ploy[i],ploy[i + 1],o,r);
    return fabs(res);
}

//最小圆覆盖---随机增量
void min_cover_circle(Point p[], int n, Point &c, double &r) {
    random_shuffle(p, p + n);
    c = p[0];
    r = 0;
    for(int i = 1; i < n; i++) {
        if((p[i] - c).norm() > r + eps) { //第一个点
            c = p[i];
            r = 0;
            for(int j = 0; j < i; j++)
                if((p[j] - c).norm() > r + eps) { //第二个点
```

```
                c = (p[i] + p[j]) * 0.5;
                r = (p[j] - c).norm();
                for(int k = 0; k < j; k++)
                    if((p[k] - c).norm() > r + eps) { //第三个点
                        //求外接圆圆心，三点必不共线
                        c = CircumCenter(p[i], p[j], p[k]);
                        r = (p[i] - c).norm();
                    }
            }
        }
    }
}

//圆面积k次交
double angle(Point vA) {
    double ang_1 = atan2(vA.y, vA.x);
    if(ang_1 < 0) ang_1 += pi * 2.0;
    return ang_1;
}
Point p[1005];
double r[1005];
double ans[1005];
vector<pair<double, double> > g;

int main() {
    int n, i, j, k, t;
    double ang_1, ang_2;
    Point p1, p2;
    scanf("%d", &n);
    for(i = 0; i < n; i++) scanf("%lf%lf%lf", &p[i].x, &p[i].y, &r[i]);
    for(i = 0; i < n; i++) {
        g.clear();
        t = 1;
        for(j = 0; j < n; j++) {
            if(i == j) continue;
            k = dot_to_circle(p[i], r[i], p[j], r[j]);
            if(k == 3) {
                if(i < j) t++;
            } else if(k == 5 || k == 2) t++;
            else if(k == 0) {
                point_of_circle(p[i], r[i], p[j], r[j], p1, p2);
                ang_1 = angle(p1 - p[i]);
                ang_2 = angle(p2 - p[i]);
                if(sgn(ang_1 - ang_2) == -1) t++;
                g.push_back(P(ang_1, -1));
                g.push_back(P(ang_2, 1));
            }
        }
        if(g.size() == 0) ans[t] += pi * r[i] * r[i] * 2.0;
        else {
            sort(g.begin(), g.end());
            g.push_back(P(g[0].first + pi * 2.0, g[0].second));
            for(j = 1; j < g.size(); j++) {
                t += g[j - 1].second;
                p1 = Point(r[i], 0.0).rot(g[j - 1].first) + p[i];
                p2 = Point(r[i], 0.0).rot(g[j].first) + p[i];
                ang_1 = g[j].first - g[j - 1].first;
                ans[t] += (p1 ^ p2) + (ang_1 - sin(ang_1)) * r[i] * r[i];
            }
        }
    }
    for(i = 1; i <= n; i++) {
        ans[i] -= ans[i + 1];
        printf("[%d] = %.3f\n", i, ans[i] * 0.5);
```

```
    }
    return 0;
}
```

平面最近点对

```
//平面最近点对---分治
bool cmp1(const Point &a, const Point &b) {
    if(a.x != b.x) return a.x < b.x;
    else return a.y < b.y;
}

bool cmp2(const Point &a, const Point &b) {
    return a.y < b.y;
}

double solve(int l, int r) {
    double d = 1e15;
    int i, j;
    if(l + 3 >= r) {
        for(i = l; i < r; i++) {
            for(j = i + 1; j <= r; j++) d = min(d, dist(f[i], f[j]));
        }
        return d;
    }
    int mid = (l + r) >> 1;
    int k = 0;
    d = min(solve(l, mid), solve(mid + 1, r));
    for(i = l; i <= r; i++) {
        if(fabs(f[i].x - f[mid].x) <= d) g[k++] = f[i];
    }
    sort(g, g + k, cmp2);
    for(i = 0; i < k; i++) {
        for(j = i + 1; j < k; j++) {
            if(g[j].y - g[i].y > d) break;
            d = min(d, dist(g[i], g[j]));
        }
    }
    return d;
}

int main() {
    int n, i;
    while(~scanf("%d", &n) && n) {
        for(i = 0; i < n; i++) scanf("%lf%lf", &f[i].x, &f[i].y);
        sort(f, f + n, cmp1);
        printf("%.2f\n", solve(0, n - 1));
    }
    return 0;
}
```

多边形

```
//判断多边形与线段交即判断多边形的每条边与线段交
//判断线段在多边形内即判断线段的两个端点在多边形内部
//应先判相交，再判内含关系……

//多边形面积
double Calarea(Point ploy[], int n) {
    double res = 0.0;
    ploy[n] = ploy[0];
    for(int i = 0; i < n; i++) res += (ploy[i] ^ ploy[i + 1]);
    return fabs(res * 0.5);
```

```
}

//判断多边形是否是凸多边形,即对每条边判断其相邻两点是否在同侧即可

//判断点在凸多边形内，-1在多边形外，1在内，0在边上
int inConvexPoly(Point a, Point p[], int n) {
    p[n] = p[0];
    for(int  i = 0; i < n; i++) {
        if(sgn((p[i] - a) ^ (p[i + 1] - a)) < 0) return -1;//若凸包为顺时针<改为>

        else if(isPointOnSegment(a, Line(p[i], p[i + 1]))) return 0;
    }
    return 1;
}



//判断点在任意多边形内：(需先特判点在多边形上)
作出要判断的点的水平线，对于多边形每条边作出判断：
如果两个端点一上一下或一下一个在水平线上，则视为有一个交点；否则视为无交点
求出所有交点，若在要判断点两侧的交点的个数都是奇数个，则点在多边形内部

//直线与简单多边形交,以下代码用于求直线在多边形内(包括边界)最长的连续部分
//整体思想为判断每一个线段是否在多边形内,用判断点是否在多边形内的方法实现
typedef pair<Point, int> Pt;
vector<Pt> g;
double solve(Line l, int n) {
    int i, a, b;
    g.clear();
    for(i = 0; i < n; i++) {
        if(!seg_line_inter(l, Line(p[i], p[i + 1]))) continue;
        pair<int, Point> e = l & Line(p[i], p[i + 1]);
        if(e.fi == -1) {
            g.push_back(Pt((p[i] + p[i + 1]) * 0.5, -1)); //边界与直线重合,加辅助点
            continue;
        }
        a = sgn((l.e - l.s) ^ (p[i] - l.s));
        b = sgn((l.e - l.s) ^ (p[i + 1] - l.s));
        if(a + b == 0) g.push_back(Pt(e.se, 2));  //一上一下
        else if(a + b == 1) g.push_back(Pt(e.se, 1));  //一下一个在水平线上
        else g.push_back(Pt(e.se, 0));  //无贡献点
    }
    int m = g.size();
    if(m == 0) return 0;
    sort(g.begin(), g.end());
    double res = 0, pre = 0;
    for(i = a = 0; i + 1 < m; i++) {
        if(g[i].se > 0) a++;  //左边的有效点个数
        if(g[i].fi == g[i + 1].fi) continue;  //重点,跳过
        if(g[i].se == -1 || g[i + 1].se == -1) pre += (g[i].fi - g[i + 1].fi).norm(); //这条线段是边界且在直线上,视为在内部
        else {
            if(a & 1) pre += (g[i].fi - g[i + 1].fi).norm(); //左边点为奇数个---在内部
            else {
                res = max(res, pre);  //在外部,更新答案
                pre = 0;
            }
        }
    }
    res = max(res, pre);
    return res;
}
```

多边形面积交或并

分解成三角形，两两求面积交，用半平面交求凸多边形面积交

多个凸包面积并(不如上java)

```
typedef pair<double,int> DI;
double seg(Point o, Point a, Point b) {
    if (sgn(b.x - a.x) == 0) return (o.y - a.y) / (b.y - a.y);
    return (o.x - a.x) / (b.x - a.x);
}
vector<Point> p[110];
DI s[2000200];
double polyunion(int n) { //求n个多凸包的面积并
    double ret = 0;
    for (int i = 0; i < n; i++) {
        int sz = p[i].size();
        for (int j = 0; j < sz; j++) {
            int m = 0;
            s[m++] = DI(0, 0);
            s[m++] = DI(1, 0);
            Point a = p[i][j], b = p[i][(j + 1) % sz];
            for (int k = 0; k < n; k++) {
                if (i != k) {
                    int siz = p[k].size();
                    for (int ii = 0; ii < siz; ii++) {
                        Point c = p[k][ii], d = p[k][(ii + 1) % siz];
                        int c1 = sgn((b - a) ^ (c - a));
                        int c2 = sgn((b - a) ^ (d - a));
                        if (c1 == 0 && c2 == 0) {
                            if (sgn((b - a) * (d - c)) > 0 && i > k) {
                                s[m++] = DI(seg(c, a, b), 1);
                                s[m++] = DI(seg(d, a, b), -1);
                            }
                        } else {
                            double s1 = (d - c) ^ (a - c);
                            double s2 = (d - c) ^ (b - c);
                            if (c1 >= 0 && c2 < 0) s[m++] = DI(s1 / (s1 - s2), 1);
                            else if (c1 < 0 && c2 >= 0) s[m++] = DI(s1 / (s1 - s2), -1);
                        }
                    }
                }
            }
            sort(s, s + m);
            double pre = min(max(s[0].first, 0.0), 1.0), now;
            double sum = 0;
            int cov = s[0].second;
            for (int j = 1; j < m; j++) {
                now = min(max(s[j].first, 0.0), 1.0);
                if (!cov) sum += now - pre;
                cov += s[j].second;
                pre = now;
            }
            ret += (a ^ b) * sum;
        }
    }
    return ret * 0.5;
}
```

旋转卡壳--两个多边形--可适用于单个多边形，要求多边形为凸

对踵点对，两点旋转方向不同、、并踵点对，两点旋转方向相同(两点形成的边平行),以下内容均基于对踵点对、、凸包公切线一定在并踵点对上
凸包直径，凸包间最大距离用b[fb],b[fa+1]两点到a[fa],a[fa+1]两点的最大距离，即4个点对距离
凸包宽度用b[fb]到直线a[fa]--a[fa+1]的距离
凸包间最小距离用b[fb],b[fb+1]两点到线段a[fa]--a[fa+1]的距离(如下模板)
求凸包最小面积外接矩形和最小周长外接矩形的方法(两者的答案并不一定是同一个矩形)：定义fc为x坐标最小点，fd为x坐标最大点，依旧枚举fa--fa+1

边，跑fb的旋转卡壳，求出fb到fa--fa+1交点u，对fb--u跑fd的卡壳，对u--fb跑fc的卡壳(待验证)，然后对4条切线求矩形、
求凸包内面积最大的内接三角形,枚举固定一个点,旋转枚举另一个点,第三个点是单增的,复杂度n^2
错误做法,枚举一个点,同时旋转其它两个点求得答案

凸包的闽科夫斯基和

给定平面上两个凸多边形P和Q，P和Q的矢量和，记为P+Q定义如下：P+Q={a+b|a∈P,b∈Q}
性质：P+Q是一个凸包，同时也是P和Q的并踵点对的和集，P+Q顶点数不超过P和Q的顶点和
其差P-Q是一个凸包，同时也是P和Q的对踵点对的差集，P-Q顶点数不超过P和Q的顶点和，若P和Q相交，则P-Q包含原点

```
//枚举a凸包的边，旋转b凸包,凸包为逆时针顺序
double rot_solve(Point a[], Point b[], int n, int m) {
    int fa, fb, i;
    Point u;
    //寻找y轴最远点对
    for(fa = i = 0; i < n; i++) {
        if(a[i].y < a[fa].y) fa = i;
    }
    for(fb = i = 0; i < m; i++) {
        if(b[i].y > b[fb].y) fb = i;
    }
    a[n] = a[0];
    b[m] = b[0];
    double ans = MAX;
    for(i = 0; i < n; i++) { //旋转卡壳，寻找对踵点
        while(sgn(((a[fa + 1] - a[fa]) ^ (b[fb + 1] - a[fa])) - ((a[fa + 1] -
                a[fa]) ^ (b[fb] - a[fa]))) == 1) fb = (fb + 1) % m;
        u = Point_to_Seg(b[fb], Line(a[fa], a[fa + 1]));
        ans = min(ans, (b[fb] - u).norm());
        u = Point_to_Seg(b[fb + 1], Line(a[fa], a[fa + 1]));
        ans = min(ans, (b[fb + 1] - u).norm());
        fa = (fa + 1) % n;
    }
    return ans;
}



//求凸包，返回凸包点数
int convexhull(Point p[], int n, Point q[]) {
    sort(p, p + n);
    int i, m = 0;
    for(i = 0; i < n; i++) {
        while(m > 1 && sgn((q[m - 1] - q[m - 2]) ^ (p[i] - q[m - 2])) <= 0) m--;
        q[m++] = p[i];
    }
    int k = m;
    for(i = n - 2; i >= 0; i--) {
        while(m > k && sgn((q[m - 1] - q[m - 2]) ^ (p[i] - q[m - 2])) <= 0) m--;
        q[m++] = p[i];
    }
    if(n > 1) m--;
    return m;
}
```

```
//半平面交,可用于凸包缩小放大
//半平面交的直线结构体
struct Line { //两点式,定义方向向量由s指向e
    Point s, e;
    double ang;
    Line() {}
    Line(Point _s, Point _e) {
        s = _s;
        e = _e;
```

```
        ang = atan2(e.y - s.y, e.x - s.x);
    }
    //求两直线交点，-1重合，0相交，1平行
    pair<int, Point> operator &(Line b) {
        if(sgn((s - e) ^ (b.s - b.e)) == 0) {
            if(sgn((s - b.e) ^ (b.s - b.e)) == 0) return make_pair(-1, s);
            else return make_pair(1, s);
        }
        double t = ((s - b.s) ^ (b.s - b.e)) / ((s - e) ^ (b.s - b.e));
        return make_pair(0, Point(s.x + (e.x - s.x) * t, s.y + (e.y - s.y) * t));
    }
    bool operator < (const Line &b) const {
        if(sgn(ang - b.ang) == 0) return ((s - b.s) ^ (s - e)) > 0;
        else return sgn(ang - b.ang) < 0;
    }
};
//判断点在直线左边
bool OnLeft(Line l, Point p) {
    return sgn((l.e - l.s) ^ (p - l.s)) > 0;
}
// 半平面交 左半平面
// 存在重合的方向相反的直线可能仍会得到一个凸包，但是面积为0，需特判
Line que[maxn * 2];
int HalfplaneIntersection(Line l[], int n, Point p[]) {
    sort(l, l + n); //极角排序
    int i, tot = 1;
    for(i = 1; i < n; i++) {
        if(sgn(abs(l[i].ang - l[i - 1].ang)) != 0) l[tot++] = l[i];
    }
    int head = 0, tail = 1;
    que[0] = l[0];
    que[1] = l[1];
    for(i = 2; i < tot; i++) {
        while(head < tail && sgn(((que[tail] & que[tail - 1]).second - l[i].s)
                                 ^ (l[i].e - l[i].s)) > 0) --tail;
        while(head < tail && sgn(((que[head] & que[head + 1]).second - l[i].s)
                                 ^ (l[i].e - l[i].s)) > 0) ++head;
        que[++tail] = l[i];
    }
    while(head < tail && sgn(((que[tail] & que[tail - 1]).second - que[head].s)
                             ^ (que[head].e - que[head].s)) > 0) --tail;
    while(head < tail && sgn(((que[head] & que[head + 1]).second - que[tail].s)
                             ^ (que[tail].e - que[tail].s)) > 0) ++head;
    if(tail <= head + 1) return 0;//无解
    int m = 0;
    for(i = head; i < tail; i++)  p[m++] = (que[i] & que[i + 1]).second;
    if(head < tail - 1) p[m++] = (que[head] & que[tail]).second;
    return m;
}


//半平面ax+by+c>0转化为两点式
if(sgn(a) == 0) {
    if(sgn(b) == 0) {
        if(sgn(c) != 1) break;
        else continue;
    }
    L[k++] = Line(Point(0.0, -c / b), Point(sgn(b), -c / b));
} else {
    if(sgn(b) == 0) L[k++] = Line(Point(-c / a, 0.0), Point(-c / a, -sgn(a)));
    else L[k++] = Line(Point(0.0, -c / b), Point(sgn(b), -(c + a * sgn(b)) / b));
}
```

求点集内是否存在面积为S的三角形

复杂度n^2logn

```
struct node {
    int u, v;
    bool operator <(const node &b) const {
        return ((p[u] - p[v]) ^ (p[b.u] - p[b.v])) > 0;
    }
};
vector<node> f;
int id[maxn];
int main() {
    int n, i, j, l, r;
    LL s;
    scanf("%d%lld", &n, &s);
    s *= 2;
    for(i = 0; i < n; i++) scanf("%d%d", &p[i].x, &p[i].y);
    for(i = 0; i < n; i++) id[i] = i;
    sort(p, p + n);
    for(i = 0; i < n; i++) {
        for(j = i + 1; j < n; j++) f.push_back(node{i, j});
    }
    sort(f.begin(), f.end());
    for(auto e : f) {
        i = e.u;
        j = e.v;
        if(id[i] > id[j]) swap(i, j);
        w = p[id[i]] - p[id[j]];
        l = 0, r = id[i] - 1;
        while(l < r) {
            int mid = (l + r + 1) >> 1;
            if(abs((p[mid] - p[id[j]]) ^ w) >= s) l = mid;
            else r = mid - 1;
        }
        if(abs((p[l] - p[id[j]]) ^ w) == s) {
            printf("Yes\n%d %d\n%d %d\n%d %d\n", p[id[i]].x, p[id[i]].y, p[id[j]].x, p[id[j]].y, p[l].x, p[l].y);
            return 0;
        }
        l = id[j] + 1, r = n - 1;
        while(l < r) {
            int mid = (l + r) >> 1;
            if(abs((p[mid] - p[id[j]]) ^ w) >= s) r = mid;
            else l = mid + 1;
        }
        if(abs((p[l] - p[id[j]]) ^ w) == s) {
            printf("Yes\n%d %d\n%d %d\n%d %d\n", p[id[i]].x, p[id[i]].y, p[id[j]].x, p[id[j]].y, p[l].x, p[l].y);
            return 0;
        }
        swap(id[i], id[j]);
        swap(p[id[i]], p[id[j]]);
    }
    puts("No");
    return 0;
}
```

```
//三角形重心
Point MassCenter(Point A, Point B, Point C) {
    return (A + B + C) * (1.0 / 3.0);
}
//三角形内心
Point InnerCenter(Point A, Point B, Point C) {
    double a = dist(B, C), b = dist(A, C), c = dist(A, B);
    return (A * a + B * b + C * c) * (1.0 / (a + b + c));
```

```
}
//三角形外心
Point CircumCenter(Point A, Point B, Point C) {
    Point t1 = B - A, t2 = C - A, t3((t1 * t1) * 0.5, (t2 * t2) * 0.5);
    swap(t1.y, t2.x);
    return A + Point(t3 ^ t2, t1 ^ t3) * (1.0 / (t1 ^ t2));
}
//三角形垂心
Point OrthoCenter(Point A, Point B, Point C) {
    return MassCenter(A, B, C) * 3.0 - CircumCenter(A, B, C) * 2.0;
}
```

立体几何

```
struct Point3 {
    double x,y,z;
    Point3(double _x = 0.0,double _y = 0.0,double _z = 0.0): x(_x),y(_y),z(_z) {}
    Point3 operator +(const Point3 &b) const {
        return Point3(x + b.x,y + b.y,z + b.z);
    }
    Point3 operator -(const Point3 &b) const {
        return Point3(x - b.x,y - b.y,z - b.z);
    }
    double operator *(const Point3 &b) const { //点乘
        return (x * b.x + y * b.y + z * b.z);
    }
    Point3 operator ^(const Point3 &b) const { //叉乘
        return Point3(y * b.z - z * b.y,z * b.x - x * b.z,x * b.y - y * b.x);
    }
    Point3 operator *(const double &k) const {
        return Point3(x * k,y * k,z * k);
    }
    bool operator ==(const Point3 &b) const {
        return sgn(x - b.x) == 0 && sgn(y - b.y) == 0 && sgn(z - b.z) == 0;
    }
    double norm() { //求模
        return sqrt(x * x + y * y + z * z);
    }
    Point3 unit() {
        if(sgn(x) == 0 && sgn(y) == 0 && sgn(z) == 0) return Point3(0.0,0.0,0.0);
        double ll = 1.0 / norm();
        return Point3(x * ll,y * ll,z * ll);
    }
};
//s指向e
struct Line3 {
    Point3 s,e;
    Line3() {}
    Line3(Point3 _s,Point3 _e): s(_s),e(_e) {}
};

struct Plane {
    Point3 sa,sb,sc,e;//e是法向量
    Plane() {}
    Plane(Point3 _sa,Point3 _sb,Point3 _sc): sa(_sa),sb(_sb),sc(_sc) {
        e = (sa - sb) ^ (sb - sc);
    }
};
//两点距离
double dis_point(Point3 p1,Point3 p2) {
    return (p1 - p2).norm();
}
//判断三点共线，同判断点在直线上
```

```
bool dots_inline(Point3 p1,Point3 p2,Point3 p3) {
    return sgn(((p1 - p2) ^ (p2 - p3)).norm()) == 0;
}
//判断点在平面上,同判断4点共面
bool dots_oneplane(Plane PL,Point3 p) {
    return sgn(PL.e * (p - PL.sa)) == 0;
}

//判断点在线段上
bool dot_online(Line3 L,Point3 p) {
    if(!dots_inline(p,L.e,L.s)) return false; //先判点在直线上
    if(sgn((L.e - p) * (L.s - p)) <= 0) return true;
    else return false;
}

//判断点在空间三角形上,包括边界,(利用面积相等法判定)
bool dot_in_Triangle(Point3 p,Plane PL) {
    return sgn(((PL.sa - PL.sb) ^ (PL.sa - PL.sc)).norm() - ((p - PL.sa) ^ (p - PL.sb)).norm()
               - ((p - PL.sb) ^ (p - PL.sc)).norm() - ((p - PL.sc) ^ (p - PL.sa)).norm()) == 0;
}

//判断两点与线段位置关系，必须点线共面，不然无意义
//1 同侧; -1 异侧; 0 有点在线段所属直线上
int Point_Position_Seg(Point3 a,Point3 b,Line3 l) {
    return sgn(((l.s - l.e) ^ (a - l.e)) * ((l.s - l.e) ^ (b - l.e)));
}

//判断两点与平面位置关系
//1 同侧; -1 异侧; 0 有点在平面上
int Point_Position_Plane(Point3 a,Point3 b,Plane PL) {
    return sgn((PL.e * (a - PL.sa)) * (PL.e * (b - PL.sa)));
}

//判断两直线平行
bool Parallel_Line(Line3 l1,Line3 l2) {
    return sgn(((l1.s - l1.e) ^ (l2.s - l2.e)).norm()) == 0;
}

//判断两平面平行
bool Parallel_Plane(Plane PL1,Plane PL2) {
    return sgn((PL1.e ^ PL2.e).norm()) == 0;
}

//判断直线与平面平行
bool Parallel_L_P(Line3 l,Plane PL) {
    return sgn((l.s - l.e) * PL.e) == 0;
}

//判断两直线垂直
bool Vertical_Line(Line3 l1,Line3 l2) {
    return sgn((l1.s - l1.e) * (l2.s - l2.e)) == 0;
}

//判断两平面垂直
bool Vertical_Plane(Plane PL1,Plane PL2) {
    return sgn(PL1.e * PL2.e)== 0;
}

//判断直线与平面垂直
bool Vertical_L_P(Line3 l,Plane PL) {
    return sgn(((l.s - l.e) ^ PL.e).norm()) == 0;
}

//判断两线段相交
```

```
bool Interset_Seg(Line3 l1,Line3 l2) {
    if(!dots_oneplane(Plane(l1.s,l1.e,l2.s),l2.e)) return false;
    if(!dots_inline(l1.s,l1.e,l2.s) || !dots_inline(l1.s,l1.e,l2.e))
        return (Point_Position_Seg(l1.s,l1.e,l2) <= 0 && Point_Position_Seg(l2.s,l2.e,l1) <= 0);
    else return dot_online(l2,l1.s) || dot_online(l2,l1.e) || dot_online(l1,l2.s) || dot_online(l1,l2.e);
}

//判断线段与空间三角形相交
bool Interset_Triangle(Line3 l,Plane PL) {
    return Point_Position_Plane(l.s,l.e,PL) <= 0 && Point_Position_Plane(PL.sa,PL.sb,Plane(l.s,l.e,PL.sc)) <= 0
    && Point_Position_Plane(PL.sb,PL.sc,Plane(l.s,l.e,PL.sa)) <= 0 && Point_Position_Plane(PL.sc,PL.sa,Plane(l.s,l.e,PL.sb)) <= 0;
}

//求两直线交点,需保证直线共面且不平行
//求线段交点需保证线段共面和相交且不平行
Point3 Interset_L_L(Line3 l1,Line3 l2) {
    Point3 ret = l1.s;
    double t = ((l1.s.x - l2.s.x) * (l2.s.y - l2.e.y) - (l1.s.y - l2.s.y) * (l2.s.x - l2.e.x)) /
    ((l1.s.x - l1.e.x) * (l2.s.y - l2.e.y) - (l1.s.y - l1.e.y) * (l2.s.x - l2.e.x));
    ret = ret + (l1.e - l1.s) * t;
    return ret;
}

//求直线与平面交点，需保证直线与平面不平行，且平面的三点不能三点共线
//若以判断线段与空间三角形相交，可用于求线段与空间三角形交点
Point3 Interset_L_P(Line3 l,Plane PL) {
    double t = PL.e * (PL.sa - l.s) / (PL.e * (l.e - l.s));
    return (l.s + (l.e - l.s) * t);
}

//求两平面交线,需先判断两平面是否平行，且每个平面的三点都不共线
Line3 Interset_P_P(Plane P1,Plane P2) {
    Line3 ret;
    ret.s = Parallel_L_P(Line3(P1.sa,P1.sb),P2) ? Interset_L_P(Line3(P1.sb,P1.sc),P2) : Interset_L_P(Line3(P1.sa,P1.sb),P2);
    ret.e = Parallel_L_P(Line3(P1.sc,P1.sa),P2) ? Interset_L_P(Line3(P1.sb,P1.sc),P2) : Interset_L_P(Line3(P1.sc,P1.sa),P2);
    return ret;
}

//点到直线距离
double ptoline(Point3 p,Line3 l) {
    return ((l.s - p) ^ (l.e - p)).norm() / dis_point(l.s,l.e);
}

//点到直线的投影，即点到直线最短距离的那个交点
Point3 PointToLine(Point3 p,Line3 l) {
    Point3 pp = (l.s - p) ^ (l.e - p);
    Point3 vec = (l.e - l.s) ^ pp;
    return vec.unit() * (pp.norm() / dis_point(l.s,l.e)) + p;
}

//点到线段距离1
double ptoseg(Point3 p,Line3 l) {
    Point3 pp = PointToLine(p,l);
    if(dot_online(l,pp)) return dis_point(pp,p);
    else return min(dis_point(l.s,p),dis_point(l.e,p));
}

//点到线段距离2
double ptoseg(Point3 p,Line3 l) {
    if(l.e == l.s) return dis_point(l.e,p);
    Point3 v1 = l.e - l.s,v2 = p - l.s,v3 = p - l.e;
    if(sgn(v1 * v2) < 0) return v2.norm();
    else if(sgn(v1 * v3) > 0) return v3.norm();
    else return (v1 ^ v2).norm() / v1.norm();
```

```
}

//点到平面距离
double ptoplane(Point3 p,Plane PL) {
    return fabs((PL.e * (p - PL.sa)) / PL.e.norm());
}

//直线到直线距离
double linetoline(Line3 l1,Line3 l2) {
    if(Parallel_Line(l1,l2)) return ptoline(l1.s,l2);
    Point3 ret = (l1.s - l1.e) ^ (l2.s - l2.e);
    return fabs((l1.s - l2.s) * ret / ret.norm());
}

//点p绕着直线l的法向量逆时针旋转弧度ang
Point3 rotate(Point3 p,Line3 l,double ang) {
    if(dots_inline(p,l.e,l.s)) return p;
    Point3 fa1 = (l.e - l.s) ^ (p - l.s);
    Point3 fa2 = (l.e - l.s) ^ fa1;
    double len = fabs(((l.e - p) ^ (l.s - p)).norm()) / dis_point(l.e,l.s);
    fa1 = fa1.unit() * len;
    fa2 = fa2.unit() * len;
    Point3 h = p + fa2;
    Point3 pp = h + fa1;
    return (h + (p - h) * cos(ang) + (pp - h) * sin(ang));
}

//求两直线的公垂线，p1表示直线l1与公垂线的交点，p2表示直线l2与公垂线的交点
void Vertical(Line3 l1,Line3 l2,Point3 &p1,Point3 &p2) {
    Point3 e = l2.e + ((l1.s - l1.e) ^ (l2.s - l2.e));
    p1 = Interset_L_P(l1,Plane(l2.s,l2.e,e));
    p2 = PointToLine(p1,l2);
}

//四面体面积
//ab叉乘ac点乘ad
double get_vol(Point3 a,Point3 b,Point3 c,Point3 d) {
    return ((b - a) ^ (c - a)) * (d - a) / 6;
}
```