22. 计算几何

22.1 二维计算几何

[常数]

```
const double pi = acos(-1);
const double eps = 1e-8;
```

[浮点数的比较]

```
int cmp(double x, double y) {  // 比较浮点数
  if (fabs(x - y) < eps) return 0;  // 相等
  return x < y ? -1 : 1;
}
```

[符号函数]

```
int sgn(double x) {  // 符号函数
  if (fabs(x) < eps) return 0;  // 0
  return x < 0 ? -1 : 1;
}
```

[向量]

```
struct Point {
  double x, y;

  Point() {};
  Point(double _x, double _y) :x(_x), y(_y) {}

  void read() { cin >> x >> 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 ); };
  Point operator*(const Point& b)const { return Point(x * b.x, y * b.y ); };  // 点积
  Point operator^(const Point& b)const { return Point(x * b.y, y * b.x ); };  // 叉积
  Point operator*(const double k)const { return Point( k * x, k * y ); };
  bool operator==(const Point& b)const { return !cmp(x, b.x) && !cmp(y, b.y); };
};  // 点、起点在原点的向量
// 向量
Point vec_plus(Point& a, Point& b) { return Point{ a.x + b.x,a.y + b.y }; }  // 向量加法
Point vec_sub(Point& a, Point& b) { return Point{ a.x - b.x,a.y - b.y }; }  // 向量减法,向量a-向量b
Point vec_mul(double& a, Point& b) { return Point{ a * b.x,a * b.y }; }  // 向量数乘
double dot_product(Point& a, Point& b) { return a.x * b.x + a.y * b.y;}  // 向量点积
double cross_product(Point& a, Point& b) { return a.x * b.y - b.x * a.y; }  // 向量叉积
```

```
double get_length(Point& a) { return sqrt(dot_product(a, a)); }  // 向量的模
double get_angle(Point& a, Point& b) { return acos(dot_product(a, b) / get_length(a) / get_length(b)); };  // 向量夹角
double get_area(Point& a, Point& b, Point& c) { return cross_product(b - a, c - a); }  // 向量ab和ac围成的平行四边形的有向面积
Point rotate(Point& a, double angle) { return Point{ a.x * cos(angle) + a.y * sin(angle), -a.x * sin(angle) + a.y * cos(angle) }; }  // 向量a的终点绕原点旋转angle角
```

[直线与线段]

```
bool is_point_on_line(Point& a, Point& p, Point& v) {  // 判断点a是否在直线(点p+向量v)上
  return sgn(cross_product(a, v)) == 0;  // 叉积为0时点在线上
}

Point get_intersection(Point& p, Point& v, Point& q, Point& w) {  // 求直线(点p+向量v)与(点q+向量w)的交点
  if (sgn(cross_product(v, w)) == 0) return Point{ INF,INF };  // 两直线平行或重合

  Point u = p - q;
  double k = cross_product(w, u) / cross_product(v, w);  // 比例系数
  return p + v * k;
}

double distance_from_point_to_line(Point& p, Point& a, Point& b) {  // 求点p到经过点a、b的直线的距离
  Point v1 = b - a, v2 = p - a;
  return fabs(cross_product(v1, v2) / get_length(v1));
}

double distance_from_point_to_segment(Point& p, Point& a, Point& b) {  // 求点p到线段ab的距离
  if (a == b) return get_length(p - a);

  Point v1 = b - a, v2 = p - a, v3 = p - b;
  if (sgn(dot_product(v1, v2)) < 0) return get_length(v2);  // p在靠近线段外侧,靠近a
  if (sgn(dot_product(v1, v2)) > 0) return get_length(v3);  // p在靠近线段外侧,靠近b
  return distance_from_point_to_line(p, a, b);  // p在线段内侧
}

Point get_projection(Point& p, Point& a, Point& b) {  // 求点p在直线ab上的投影
  Point v = b - a;
  return a + v * (dot_product(v, p - a) / dot_product(v, v));
}

bool is_point_on_segment(Point& p, Point& a, Point& b) {  // 判断点p是否在线段ab上
  return sgn(cross_product(p - a, p - b)) == 0 && sgn(dot_product(p - a, p - b)) <= 0;
}

bool is_segment_intercection(Point& a1, Point& a2, Point& b1, Point& b2) {  // 判断线段a1a2和b1b2是否相交
  double c1 = cross_product(a2 - a1, b1 - a1), c2 = cross_product(a2 - a1, b2 - a1),
    c3 = cross_product(b2 - b1, a2 - b1), c4 = cross_product(b2 - b1, a1 - b1);
  return sgn(c1) * sgn(c2) <= 0 && sgn(c3) * sgn(c4) <= 0;  // 若非规范相交不算相交,则去掉等号
}
```

[三角形]

①重心:到三角形的三个顶点距离的平方和最小,到三边的距离之积最大.

②设三角形的三边分别为a,b,c.令$p=\dfrac{a+b+c}{2}$,则$S=\sqrt{p(p-a)(p-b)(p-c)}$.

[多边形]

①一般按逆时针存点.

②计算任意多边形的面积:在第一个顶点将n边形剖分为$(n-2)$个三角形,再将它们的有向面积相加,代码:

```
double get_polygon_area(Point p[], int n) {  // 求n边形的面积
  double res = 0;
  for (int i = 1; i + 1 < n; i++) res += cross_product(p[i] - p[0], p[i + 1] - p[i]);
  return res / 2;
}
```

③判断点是否在任意多边形内:

1)转角法:计算点沿多边形的顶点转一圈的转角和,若点在多边形内部,则转角和为360°.

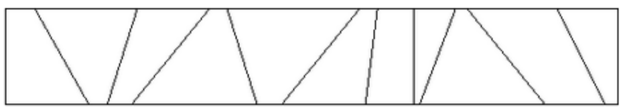
2)射线法:任作一条以该点为起点的、与所有边都不平行(可随机两次提高正确率)的射线,求它与多边形的交点个数.若交点个数为偶数,则该店在多边形外,否则在多边形内.

④判断点是否在凸多边形内:设逆时针为该凸多边形的正方向.用叉积判断该点是否在多边形的所有边的左侧.

⑤Pick定理:顶点都为整点的多边形的面积$S=a+\dfrac{b}{2}-1$,其中a为多边形内整点的个数,b为多边形边上整点的个数.

22.1.1 玩具

题意



给定如上图所示的长方形玩具收纳盒,求每个分区中玩具的数量.

有多组(不多于10组)测试数据.对每组测试数据,第一行包含六个整数n,m,x_1,y_1,x_2,y_2,表示有$n\ \ (1\leq n\leq 5000)$个纸板、$m\ \ (1\leq m\leq 5000)$个玩具,收纳盒左上角的坐标为$(x_1,y_1)\ \ (-1\mathrm{e}5\leq x_1,y_1\leq 1\mathrm{e}5)$,右下角的坐标为$(x_2,y_2)\ \ (-1\mathrm{e}5\leq x_2,y_2\leq 1\mathrm{e}5)$.接下来$n$行每行包含两个整数$u_i,l_i$,表示第$i$个纸盒的两端点坐标分别为$(u_i,y_1),(l_i,y_2)$,数据保证纸板间不相交,且按照从左往右的顺序给出.接下来m行每行包含两个整数(x_j,y_j),表示第j个玩具的坐标,玩具给出的顺序随机,数据保证玩具不落在纸板上或收纳盒外.输入由包含单个0的一行结束.

对每组测试数据,输出$n+1$行.n个纸板将收纳盒分为$(n+1)$个分区,从左往右依次编号$0\sim n$.按分区编号递增的顺序,每行输出$i:\ j$,其中i为分区编号,j为分区内的玩具数量.每组测试数据间用空行分隔.

思路

注意到点在其左边的所有向量的右侧,在其右边的所有向量的左侧,可二分出该点属于哪个区域,每组测试样例的时间复杂度为$O(n\log n)$..判断左右侧可用叉积,叉积>0在右侧,<0在左侧.

代码

```cpp
#define x first
#define y second
const int MAXN = 5005;
int n, m;  // 隔板数、玩具数
pll a[MAXN], b[MAXN];  // 隔板上坐标、隔板下坐标
int ans[MAXN];

ll cross(ll x1, ll y1, ll x2, ll y2) { return x1 * y2 - x2 * y1; }

ll get_area(pll a, pll b, pll c) { return cross(b.x - a.x, b.y - a.y, c.x - a.x, c.y - a.y); }  // 求有向面积

int find(ll x, ll y) {  // 二分出玩具在哪个区间
  int l = 0, r = n;
  while (l < r) {
    int mid = l + r >> 1;
    if (get_area(b[mid], a[mid], { x,y }) > 0) r = mid;  // 玩具在向量左边
    else l = mid + 1;
  }
  return r;
}

int main() {
  while (cin >> n, n) {
    ll x1, y1, x2, y2; cin >> m >> x1 >> y1 >> x2 >> y2;
    for (int i = 0; i < n; i++) {
      ll u, l; cin >> u >> l;
      a[i] = { u,y1 }, b[i] = { l,y2 };
    }
    a[n] = { x2,y1 }, b[n] = { x2,y2 };  // 最后一个向量是盒子的右边界

    memset(ans, 0, so(ans));
    while (m--) {
      ll x, y; cin >> x >> y;
      ans[find(x, y)]++;
    }

    for (int i = 0; i <= n; i++) printf("%d: %d\n", i, ans[i]);
    cout << endl;
  }
}
```

22.1.2 线段

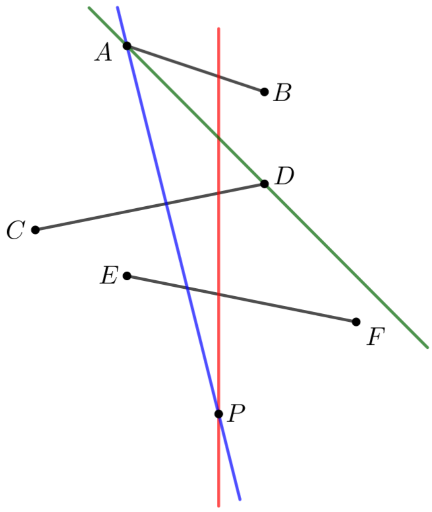
题意 ($2\ s$)

平面上有n条线段,判断是否存在一条直线使得将所有线段投影到该直线后,所有的投影线段至少有一个公共点.若存在,输出"Yes!",否则输出"No!".

有$T\ \ (1\leq T\leq 100)$组测试数据.每组测试数据第一行包含一个整数$n\ \ (1\leq n\leq 100)$,表示有n条线段.接下来n行每行包含四个实数$x_1,y_1,x_2,y_2\ \ (-1\text{e}9\leq x_1,y_1,x_2,y_2\leq 1\text{e}9)$,表示有一条端点为$(x_1,y_1)$和$(x_2,y_2)$的线段.

思路

若存在一条穿过所有线段的直线,过该直线上一点作其垂线,设垂足为P,则所有线段在垂线上的投影都有公共点P.若所有线段投影到某直线后投影线段有一个公共点P,过P作该直线的垂线,则垂线穿过所有线段,故两者等价.



任取一条直线(红色),绕其上一点P逆时针旋转直至其经过某线段的端点A(蓝色),再将其绕A逆时针旋转直至其经过另一线段的端点D(绿色).

枚举n个点中的两个点时间复杂度为$O(n^2)$,枚举n条线段时间复杂度为$O(n)$,总时间复杂度$O(n^3)$,最坏$100^3 \times 100 = 1e8, 2\ s$可过.但事实上无需枚举完$n^2$个点即可找到解,故实际的时间复杂度小于$O(n^3)$.

代码

```
#define x first
#define y second
const int MAXN = 205;  // 点数开两倍
int n;  // 线段数
pdd points[MAXN];  // 所有端点
pdd a[MAXN], b[MAXN];  // 分别存线段的两端点之一

int sgn(double x) {
  if (fabs(x) < eps) return 0;
  if (x < 0) return -1;
  else return 1;
}

int cmp(double x, double y) {
  if (fabs(x - y) < eps) return 0;
  if (x < y) return -1;
  else return 1;
}

double cross(double x1, double y1, double x2, double y2) { return x1 * y2 - x2 * y1; }

double get_area(pdd a, pdd b, pdd c) { return cross(b.x - a.x, b.y - a.y, c.x - a.x, c.y - a.y); }

bool check() {
  for (int i = 0; i < n * 2; i++) {  // 枚举第一个点
    for (int j = i + 1; j < n * 2; j++) {  // 枚举第二个点,保证i<j
      if (!cmp(points[i].x, points[j].x) && !cmp(points[i].y, points[j].y)) continue;  // 两点重合

```

```
      bool flag = 1;
      for (int k = 0; k < n; k++) {  // 判断点是否在线段两侧,若在同侧,则叉积同号
        if (sgn(get_area(points[i], points[j], a[k])) * sgn(get_area(points[i], points[j], b[k])) > 0) {
          flag = 0;
          break;
        }
      }

      if (flag) return 1;
    }
  }
  return 0;
}

int main() {
  CaseT{
    for (int i = 0, k = 0; i < n; i++) {
      double x1, y1, x2, y2; cin >> x1 >> y1 >> x2 >> y2;
      points[k++] = { x1,y1 }, points[k++] = { x2,y2 };
      a[i] = { x1,y1 }, b[i] = { x2,y2 };
    }

    if (check()) cout << "Yes!" << endl;
    else cout << "No!" << endl;
  }
}
```

22.1.3 围住奶牛

题意

给定$n\ \ (0\leq n\leq 1e4)$个点$(x_i,y_i)\ \ (-1e6\leq x_i,y_i\leq 1e6)$.求凸包的周长,保留两位小数.数据保证给定的点不都在同一直线上.

思路

[**Andrew算法**] 先将所有点以横坐标为第一关键字、纵坐标为第二关键字升序排列.Andrew算法将凸包分为上、下两部分,先从左往右求上半凸包,再从右往左求下半凸包.因凸包是凸多边形,则从凸包的一个顶点出发逆时针行进,过程中一直向左拐,可用叉积的政府判断.用栈维护凸包:①当栈内的点数$<\ 2$时直接入栈;②加入一个新点时,检查路径是否保持向左拐.若是,则栈顶点在凸包中,将新点入栈;否则栈顶点不在凸包中,弹出栈顶后将新点入栈.求出下半凸包后需用上半凸包的起点更新下半凸包的最后一个点,故求完上半凸包后将上半凸包的起点的$used[]$置为$false$.这会导致上半凸包的起点重复入栈,但这样求凸包周长更方便.总时间复杂度$O(n\log n)$.

代码

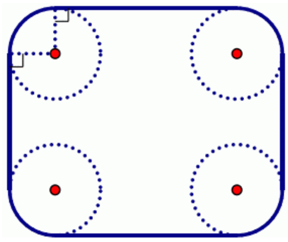
```
const int MAXN = 1e5 + 5;
int n;
int stk[MAXN], top;
bool used[MAXN];  // 记录每个点是否被用过

struct Point {  // 点类
  double x, y;

```

```cpp
  Point(double _x = 0, double _y = 0) :x(_x), y(_y) {}

  bool operator==(const Point& B)const { return cmp(x, B.x) == 0 && cmp(y, B.y) == 0; }
  bool operator<(const Point& B)const { return x == B.x ? y < B.y : x < B.x; }  // 按x升序排列,x相同时按y升序排列
}points[MAXN];
double get_dis(const Point& A, const Point& B) { return hypot(A.x - B.x, A.y - B.y); }

typedef Point Vector;  // 向量类
Vector operator+(Vector& A, Vector& B) { return Vector(A.x + B.x, A.y + B.y); }
Vector operator-(Vector& A, Vector& B) { return Vector(A.x - B.x, A.y - B.y); }
Vector operator*(Vector& A, double k) { return Vector(A.x * k, A.y * k); }
Vector operator/(Vector& A, double k) { return Vector(A.x / k, A.y / k); }
double operator*(Vector& A, Vector& B) { return A.x * B.x + A.y * B.y; }  // 点乘
double operator^(Vector& A, Vector& B) { return A.x * B.y - A.y * B.x; }  // 叉乘
double Dot_Product(Vector A, Vector B) { return A.x * B.x + A.y * B.y; }
double Cross_Product(Vector A, Vector B) { return A.x * B.y - A.y * B.x; }
double get_length(Vector& A) { return hypot(A.x, A.y); }  // 模长
double get_angle(Vector A, Vector B) { return acos(Dot_Product(A, B) / get_length(A) / get_length(B)); }  // 夹角(弧度)
double get_area(Point A, Point B, Point C) { return Cross_Product(B - A, C - A); }  // 向量AB与向量AC张成的平行四边形的面积
double get_area(Vector AB, Vector AC) { return Cross_Product(AB, AC); }  // 向量AB与向量AC张成的平行四边形的面积
Vector Rotate(Vector A, double rad) { return Vector(A.x * cos(rad) + A.y * sin(rad), -A.x * sin(rad) + A.y * cos(rad)); }  // 向量A逆时针旋转rad得到的向量
Vector Normal(Vector A) { return Vector(-A.y, A.x); }  // 向量A逆时针旋转90度得到的向量

double andrew() {
    top = 0;

    sort(points + 1, points + n + 1);

    stk[++top] = 1;  // 第一个点一定在凸包中
    for (int i = 2; i <= n; i++) {  // 求下半凸包
        while (top >= 2 && get_area(points[stk[top - 1]], points[stk[top]], points[i]) <= 0)
            used[stk[top--]] = false;

        stk[++top] = i, used[i] = true;  // 当前点入栈
    }

    int upsiz = top;  // 下半凸包的大小
    for (int i = n - 1; i >= 1; i--) {  // 求上半凸包
        if (!used[i]) {
            while (top > upsiz && get_area(points[stk[top - 1]], points[stk[top]], points[i]) <= 0)
                top--;  // 后续不会用到已出栈的点,故无需更新used[]

            stk[++top] = i;
        }
    }

    double res = 0;
    for (int i = 2; i <= top; i++) res += get_dis(points[stk[i - 1]], points[stk[i]]);
    return res;
}

```

```
void solve() {
    cin >> n;
    for (int i = 1; i <= n; i++) cin >> points[i].x >> points[i].y;

    cout << fixed << setprecision(2) << andrew();
}

int main() {
    solve();
}
```

22.1.4 信用卡凸包

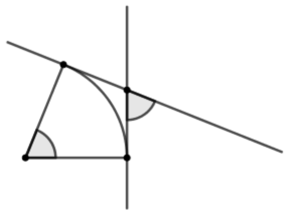
题意



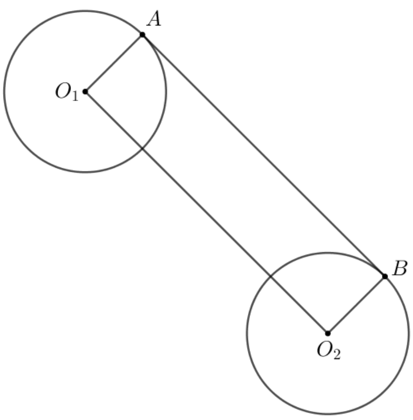
如上图所示,一张信用卡是一个矩形,其四个角作圆滑处理,使得它们都是与矩形的边相切的$\dfrac{1}{4}$圆.给定平面上的n张信用卡,求凸包周长.

第一行输入一个整数$n\ \ (1\leq n\leq 1\mathrm{e}4)$.第二行输入三个实数$a,b,r\ \ \left(0.1\leq a,b\leq 1\mathrm{e}6,0\leq r\leq \min\left\{\dfrac{a}{4},\dfrac{b}{4}\right\}\right)$,分别表示信用卡圆滑处理前的竖直方向的长度、水平方向的长度、$\dfrac{1}{4}$圆的半径.之后的n行每行输入三个实数$x,y,\theta\ \ (|x|,|y|\leq 1\mathrm{e}6,0\leq\theta<2\pi)$,分别表示信用卡的中心(对角线交点)的坐标、信用卡绕中心逆时针旋转的弧度.

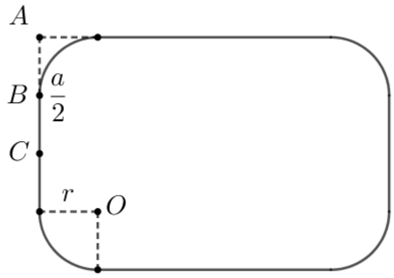
思路



先求所有圆弧的长度之和.如上图,由弦切角定理,每段弧的圆心角等于多边形的一个外角,而多边形的外角和为360°,故所有圆弧的长度之和为$2\pi r$.



再求所有线段的长度之和.如上图,注意到四边形ABO_2O_1是矩形,则$\overline{AB}=\overline{O_1O_2}$,即两圆间的线段长度等于圆心距,故对所有圆心求凸包周长即可.



如上图,C是矩形的高的中点,则$\overline{BC}=\dfrac{a}{2}-r$.故矩形的中心坐标为$\left(\dfrac{a}{2}-r,\dfrac{b}{2}-r\right)$.用中心点加上一个偏移向量来得到四个圆心点的坐标.

代码

```
const int MAXN = 1e5 + 5;
const int dx[] = { 1,1,-1,-1 }, dy[] = { 1,-1,-1,1 };  // 中心到四个圆心的偏移值
int n;  // 信用卡个数
double a, b, r;  // 矩形的高、宽、圆的半径
int cnt;  // 圆心点的个数
int stk[MAXN], top;
bool used[MAXN];  // 记录每个点是否被用过

struct Point {  // 点类
  double x, y;

  Point(double _x = 0, double _y = 0) :x(_x), y(_y) {}

  bool operator==(const Point& B)const { return cmp(x, B.x) == 0 && cmp(y, B.y) == 0; }
  bool operator<(const Point& B)const { return x == B.x ? y < B.y : x < B.x; }  // 按x升序排列,x相同时按y升序排列
}points[MAXN];
double get_dis(Point A, Point B) { return hypot(A.x - B.x, A.y - B.y); }

typedef Point Vector;  // 向量类
Vector operator+(Vector A, Vector B) { return Vector(A.x + B.x, A.y + B.y); }
Vector operator-(Vector A, Vector B) { return Vector(A.x - B.x, A.y - B.y); }
Vector operator*(Vector A, double k) { return Vector(A.x * k, A.y * k); }
Vector operator/(Vector A, double k) { return Vector(A.x / k, A.y / k); }
double operator*(Vector A, Vector B) { return A.x * B.x + A.y * B.y; }  // 点乘
double operator^(Vector A, Vector B) { return A.x * B.y - A.y * B.x; }  // 叉乘
double dot_product(Vector A, Vector B) { return A.x * B.x + A.y * B.y; }
double cross_product(Vector A, Vector B) { return A.x * B.y - A.y * B.x; }
double get_length(Vector A) { return hypot(A.x, A.y); }  // 模长
double get_angle(Vector A, Vector B) { return acos(dot_product(A, B) / get_length(A) / get_length(B)); }  // 夹角(弧度)
double get_area(Point A, Point B, Point C) { return cross_product(B - A, C - A); }  // 向量AB与向量AC张成的平行四边形的面积
double get_area(Vector AB, Vector AC) { return cross_product(AB, AC); }  // 向量AB与向量AC张成的平行四边形的面积
Vector rotate(Vector A, double rad) { return Vector(A.x * cos(rad) + A.y * sin(rad), -A.x * sin(rad) + A.y * cos(rad)); }  // 向量A逆时针旋转rad得到的向量
Vector normal(Vector A) { return Vector(-A.y, A.x); }  // 向量A逆时针旋转90度得到的向量

double andrew() {
```

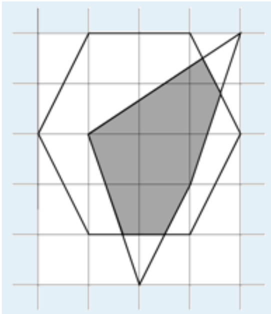
```cpp
    top = 0;

    sort(points + 1, points + cnt + 1);

    stk[++top] = 1;  // 第一个点一定在凸包中
    for (int i = 2; i <= cnt; i++) {  // 求下半凸包
        while (top >= 2 && get_area(points[stk[top - 1]], points[stk[top]], points[i]) <= 0)
            used[stk[top--]] = false;

        stk[++top] = i, used[i] = true;  // 当前点入栈
    }

    int upsiz = top;  // 下半凸包的大小
    for (int i = cnt - 1; i >= 1; i--) {  // 求上半凸包
        if (!used[i]) {
            while (top > upsiz && get_area(points[stk[top - 1]], points[stk[top]], points[i]) <= 0)
                top--;  // 后续不会用到已出栈的点,故无需更新used[]

            stk[++top] = i;
        }
    }

    double res = 0;
    for (int i = 2; i <= top; i++) res += get_dis(points[stk[i - 1]], points[stk[i]]);
    return res;
}

void solve() {
    cin >> n >> a >> b >> r;

    a = a / 2 - r, b = b / 2 - r;  // 中心坐标
    while (n--) {
        double x, y, theta; cin >> x >> y >> theta;
        for (int i = 0; i < 4; i++) {
            Vector tmp = rotate({dx[i] * b, dy[i] * a}, -theta);  // 注意-theta
            points[++cnt] = Point(x, y) + tmp;
        }
    }

    cout << fixed << setprecision(2) << andrew() + 2 * pi * r;
}

int main() {
    solve();
}
```

22.1.5 凸多边形

题意

逆时针给出n个凸多边形的顶点坐标,求它们的交的面积.

$n=2$时,两凸多边形的交如下图:



第一行输入一个整数$n\ \ (2\leq n\leq 10)$.第$i\ \ (1\leq i\leq n)$个多边形的第一行包含一个整数$m_i\ \ (3\leq m_i\leq 50)$,表示多边形的边数.接下来$m_i$行每行输入两个在$[-1000,1000]$内的整数,逆时针给出多边形各顶点的坐标.

输出它们交的面积,保留三位小数.

思路

将每个凸多边形视为若干个半平面覆盖平面后剩下的部分,求半平面交.

下面对每条直线保留其左侧的半平面,则方向相同的直线只需要保留最左侧的直线.

步骤:

①先将所有向量按极角升序排列.

②依次枚举所有向量,用双端队列维护当前半平面交的轮廓,当队列中元素≥ 2时向量会产生交点.每遍历到一个向量时维护交点数组,交点数组维护队尾向量与其上一个向量的交点.显然轮廓的向量方向是逆时针转的,则若当前向量在最后一个交点的左侧,则队尾向量出队,当前向量入队;否则当前向量直接入队.当凸包快封闭时,当前向量可能会影响队首向量,故每次都检查队首、队尾向量对应的交点是否在当前向量的右侧.凸包封闭后,用队首更新一遍队尾,再用队尾更新一遍队首.

代码

```
const int MAXN = 505;
namespace Geometry_2D {
  // 点类
  struct Point {
    double x, y;

    Point(double _x = 0, double _y = 0) :x(_x), y(_y) {}

    bool operator==(const Point& B)const { return cmp(x, B.x) == 0 && cmp(y, B.y) == 0; }
    bool operator<(const Point& B)const { return x == B.x ? y < B.y : x < B.x; }  // 按x升序排列,x相同时按y升序排列
  };
  double getDis(Point A, Point B) { return hypot(A.x - B.x, A.y - B.y); }

  // 向量类
  typedef Point Vector;
  Vector operator+(Vector A, Vector B) { return Vector(A.x + B.x, A.y + B.y); }
  Vector operator-(Vector A, Vector B) { return Vector(A.x - B.x, A.y - B.y); }
  Vector operator*(Vector A, double k) { return Vector(A.x * k, A.y * k); }
  Vector operator/(Vector A, double k) { return Vector(A.x / k, A.y / k); }
  double operator*(Vector A, Vector B) { return A.x * B.x + A.y * B.y; }  // 点乘
  double operator^(Vector A, Vector B) { return A.x * B.y - A.y * B.x; }  // 叉乘
```

```cpp
  double dotProduct(Vector A, Vector B) { return A.x * B.x + A.y * B.y; }
  double crossProduct(Vector A, Vector B) { return A.x * B.y - A.y * B.x; }
  double getLength(Vector A) { return hypot(A.x, A.y); }  // 模长
  double getAngle(Vector A, Vector B) { return acos(dotProduct(A, B) / getLength(A) / getLength(B)); }  // 夹角(弧度)
  double getArea(Point A, Point B, Point C) { return crossProduct(B - A, C - A); }  // 向量AB与向量AC张成的平行四边形的面积
  double getArea(Vector AB, Vector AC) { return crossProduct(AB, AC); }  // 向量AB与向量AC张成的平行四边形的面积
  Vector rotate(Vector A, double rad) { return Vector(A.x * cos(rad) + A.y * sin(rad), -A.x * sin(rad) + A.y * cos(rad)); }  // 向量A逆时针旋转rad得到的向量
  Vector normal(Vector A) { return Vector(-A.y, A.x); }  // 向量A逆时针旋转90度得到的向量

  // 直线类
  struct Line {
    Point p;  // 直线上一点
    Vector v;  // 方向向量

    Line() {}
    Line(Point _p, Vector _v) :p(_p), v(_v) {}

    Point getPoint(double t) { return p + v * t; }  // 直线上参数为t的点
  };
  double getAngle(Line l) {  // 求直线倾斜角
    Point q = l.p + l.v;
    return atan2(q.y - l.p.y, q.x - l.p.x);
  }
  bool isPointOnLine(Point P, Point A, Point B) { return sgn(crossProduct(P - A, P - B)) == 0; }  // 点P是否在直线AB上
  bool isPointOnSegment(Point P, Point A, Point B) { return isPointOnLine(P, A, B) && sgn(dotProduct(A - P, B - P)) < 0; }  // 点P是否在线段AB上
  double getDistanceToLine(Point P, Point A, Point B) {  // 点P到直线AB的距离
    Vector v1 = B - A, v2 = P - A;
    return fabs(crossProduct(v1, v2) / getLength(v1));
  }
  double getDistanceToSegment(Point P, Point A, Point B) {  // 点P到线段AB的距离
    if (A == B) return getLength(P - A);
    Vector v1 = B - A, v2 = P - A, v3 = P - B;
    if (sgn(dotProduct(v1, v2)) < 0) return getLength(v2);  // P的投影在线段AB外,且更靠近A
    if (sgn(dotProduct(v1, v3)) < 0) return getLength(v3);  // P的投影在线段AB外,且更靠近B
    return getDistanceToLine(P, A, B);  // P的投影在线段AB上
  }
  Point getLineIntersection(Point P, Vector v, Point Q, Vector w) {  // 求直线(P+vt)与(Q+vw)的交点,使用前需保证有交点
    Vector u = P - Q;
    double t = crossProduct(w, u) / crossProduct(v, w);
    return P + v * t;
  }
  Point getLineIntersection(Line A, Line B) { return getLineIntersection(A.p, A.v, B.p, B.v); }  // 求直线A与B的交点,使用前需保证有交点
  Point getPointProjection(Point P, Point A, Point B) {  // 点P在直线AB上的投影
    Vector v1 = B - A, v2 = P - A;
    return A + v1 * (dotProduct(v1, v2) / dotProduct(v1, v1));
  }
  bool isSegmentIntersect(Point A1, Point A2, Point B1, Point B2) {  // 判断线段A1A2和B1B2是否相交
    double c1 = crossProduct(A2 - A1, B1 - A1), c2 = crossProduct(A2 - A1, B2 - A1),
    c3 = crossProduct(B2 - B1, A2 - B1), c4 = crossProduct(B2 - B1, A1 - B1);
```

```
    return sgn(c1) * sgn(c2) <= 0 && sgn(c3) * sgn(c4) <= 0;  // 若非规范相交不算相交,则去掉等号
  }

  namespace HalfPlaneIntersection {
    Point points[MAXN];  // 多边形顶点
    Line lines[MAXN];  // 多边形的边所在直线
    int cnt = 0;  // 直线数
    int que[MAXN], hh = 0, tt = -1;  // 双端队列
    Point hull[MAXN];  // 凸包上的点

    // 半平面交
    bool isPointOnRight(Line A, Line B, Line C) {
      auto tmp = getLineIntersection(B, C);
      return sgn(getArea(A.p, A.p + A.v, tmp)) <= 0;
    }

    double half_plane_intersection() {  // 求半平面交,返回凸包面积
      sort(lines, lines + cnt, [&](const Line& A, const Line& B) {
        double a = getAngle(A), b = getAngle(B);
        if (!cmp(a, b)) return getArea(A.p, A.p + A.v, B.p) < 0;
        else return a < b;
      });  // 直线按倾斜角升序排列

      for (int i = 0; i < cnt; i++) {
        if (i && !cmp(getAngle(lines[i]), getAngle(lines[i - 1]))) continue;

        while (hh + 1 <= tt && isPointOnRight(lines[i], lines[que[tt - 1]], lines[que[tt]])) tt--;  // 检查队尾
        while (hh + 1 <= tt && isPointOnRight(lines[i], lines[que[hh]], lines[que[hh + 1]])) hh++;  // 检查队首

        que[++tt] = i;  // 当前点入队
      }

      while (hh + 1 <= tt && isPointOnRight(lines[que[hh]], lines[que[tt - 1]], lines[que[tt]])) tt--;  // 用队首更新队尾
      while (hh + 1 <= tt && isPointOnRight(lines[que[tt]], lines[que[hh]], lines[que[hh + 1]])) hh++;  // 用队尾更新队首

      que[++tt] = que[hh];  // 将队首插入队尾,方便求面积

      int idx = 0;
      for (int i = hh; i < tt; i++)  // 求凸包上的点
        hull[idx++] = getLineIntersection(lines[que[i]], lines[que[i + 1]]);

      double res = 0;
      for (int i = 1; i + 1 < idx; i++) res += getArea(hull[0], hull[i], hull[i + 1]);
      return res / 2;
    }
  }
};
using namespace Geometry_2D;
using namespace HalfPlaneIntersection;

void solve() {
  CaseT {
    int m; cin >> m;
```

```
    for (int i = 0; i < m; i++) cin >> points[i].x >> points[i].y;

    for (int i = 0; i < m; i++) lines[cnt++] = { points[i], points[(i + 1) %  m] - points[i] };
  }

  cout << fixed << setprecision(3) << half_plane_intersection();
}

int main() {
  solve();
}
```

22.1.6 赛车

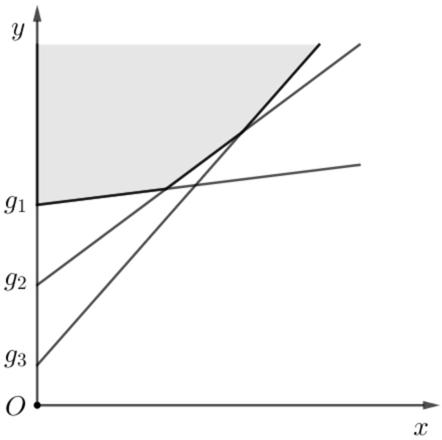
题意

赛车比赛有n辆车,分别记作$g_1,\cdots,g_n$.赛道是一条无线长的直线,不考虑赛车碰撞.初始时赛车g_i在距起跑线前进k_i的位置.比赛开始后,赛车g_i以v_i的速度匀速行驶.若一辆赛车在比赛过程中曾处于领跑位置,则该赛车最后获奖.问哪些车最后获奖.

第一行输入一个整数$n\ \ (1\leq n\leq 1\mathrm{e}4)$.第二行输入$n$个整数$k_1,\cdots,k_n\ \ (0\leq k_i\leq 1\mathrm{e}9)$.第三行输入$n$个整数$v_1,\cdots,v_n\ \ (0\leq v_i\leq 1\mathrm{e}9)$.

第一行输出获奖的车数.第二行按编号升序输出获奖车编号.

思路



i号赛车的位置$s_i=k_i+v_it$.如上图,作所有赛车对应的$s-t$图象,则在最上方的直线对应的赛车领跑,问题转化为求图中灰色区域的边界,即求所有直线的半平面交(保留直线的左侧部分),则在半平面交边界上的直线对应的赛车可获奖.注意要求的是半平面交在第一象限的部分,故可加入两条直线$x=0$和$y=0$.

注意到可能有不同的赛车对应的直线相同,可先将直线去重,记录每条直线对应的赛车.

注意多条直线穿过同一交点时该交点要留下,故isPointOnRight函数不取等.

本题精度要求高,要用long double,$\varepsilon=1\mathrm{e}-18$.

代码

```cpp
#define double long double
const double eps = 1e-18;
const int MAXN = 1e4 + 5;
int k[MAXN], v[MAXN];  // 起点、速度

namespace Geometry_2D {
    // 点类
    struct Point {
        double x, y;

        Point(double _x = 0, double _y = 0) :x(_x), y(_y) {}

        bool operator==(const Point& B)const { return cmp(x, B.x) == 0 && cmp(y, B.y) == 0; }
        bool operator<(const Point& B)const { return x == B.x ? y < B.y : x < B.x; }  // 按x升序排列,x相同时按y升序排列
        friend ostream& operator<<(ostream& out, const Point p) {
            out << '[' << p.x << ',' << p.y << ']';
            return out;
        }
    };
    double getDis(Point A, Point B) { return hypot(A.x - B.x, A.y - B.y); }

    // 向量类
    typedef Point Vector;
    Vector operator+(Vector A, Vector B) { return Vector(A.x + B.x, A.y + B.y); }
    Vector operator-(Vector A, Vector B) { return Vector(A.x - B.x, A.y - B.y); }
    Vector operator*(Vector A, double k) { return Vector(A.x * k, A.y * k); }
    Vector operator/(Vector A, double k) { return Vector(A.x / k, A.y / k); }
    double operator*(Vector A, Vector B) { return A.x * B.x + A.y * B.y; }  // 点乘
    double operator^(Vector A, Vector B) { return A.x * B.y - A.y * B.x; }  // 叉乘
    double dotProduct(Vector A, Vector B) { return A.x * B.x + A.y * B.y; }
    double crossProduct(Vector A, Vector B) { return A.x * B.y - A.y * B.x; }
    double getLength(Vector A) { return hypot(A.x, A.y); }  // 模长
    double getAngle(Vector A, Vector B) { return acos(dotProduct(A, B) / getLength(A) / getLength(B)); }  // 夹角(弧度)
    double getArea(Point A, Point B, Point C) { return crossProduct(B - A, C - A); }  // 向量AB与向量AC张成的平行四边形的面积
    double getArea(Vector AB, Vector AC) { return crossProduct(AB, AC); }  // 向量AB与向量AC张成的平行四边形的面积
    Vector rotate(Vector A, double rad) { return Vector(A.x * cos(rad) + A.y * sin(rad), -A.x * sin(rad) + A.y * cos(rad)); }  // 向量A逆时针旋转rad得到的向量
    Vector normal(Vector A) { return Vector(-A.y, A.x); }  // 向量A逆时针旋转90度得到的向量

    // 直线类
    struct Line {
       Point p;  // 直线上一点
        Vector v;  // 方向向量
        vi ids;  // 每条直线对应的赛车的编号

        Line() {}
        Line(Point _p, Vector _v) :p(_p), v(_v) {}
        Line(Point _p, Vector _v, vi& idx) :p(_p), v(_v), ids(idx) {}

        Point getPoint(double t) { return p + v * t; }  // 直线上参数为t的点
```

```
    };
    double getAngle(Line l) {  // 求直线倾斜角
        Point q = l.p + l.v;
        return atan2(q.y - l.p.y, q.x - l.p.x);
    }
    bool isPointOnLine(Point P, Point A, Point B) { return sgn(crossProduct(P - A, P - B)) == 0; }  // 点P是否在直线AB上
    bool isPointOnSegment(Point P, Point A, Point B) { return isPointOnLine(P, A, B) && sgn(dotProduct(A - P, B - P)) < 0; }  // 点P是否在线段AB上
    double getDistanceToLine(Point P, Point A, Point B) {  // 点P到直线AB的距离
        Vector v1 = B - A, v2 = P - A;
        return fabs(crossProduct(v1, v2) / getLength(v1));
    }
    double getDistanceToSegment(Point P, Point A, Point B) {  // 点P到线段AB的距离
        if (A == B) return getLength(P - A);
        Vector v1 = B - A, v2 = P - A, v3 = P - B;
        if (sgn(dotProduct(v1, v2)) < 0) return getLength(v2);  // P的投影在线段AB外,且更靠近A
        if (sgn(dotProduct(v1, v3)) < 0) return getLength(v3);  // P的投影在线段AB外,且更靠近B
        return getDistanceToLine(P, A, B);  // P的投影在线段AB上
    }
    Point getLineIntersection(Point P, Vector v, Point Q, Vector w) {  // 求直线(P+vt)与(Q+vw)的交点,使用前需保证有交点
        Vector u = P - Q;
        double t = crossProduct(w, u) / crossProduct(v, w);
        return P + v * t;
    }
    Point getLineIntersection(Line A, Line B) { return getLineIntersection(A.p, A.v, B.p, B.v); }  // 求直线A与B的交点,使用前需保证有交点
    Point getPointProjection(Point P, Point A, Point B) {  // 点P在直线AB上的投影
        Vector v1 = B - A, v2 = P - A;
        return A + v1 * (dotProduct(v1, v2) / dotProduct(v1, v1));
    }
    bool isSegmentIntersect(Point A1, Point A2, Point B1, Point B2) {  // 判断线段A1A2和B1B2是否相交
        double c1 = crossProduct(A2 - A1, B1 - A1), c2 = crossProduct(A2 - A1, B2 - A1),
            c3 = crossProduct(B2 - B1, A2 - B1), c4 = crossProduct(B2 - B1, A1 - B1);
        return sgn(c1) * sgn(c2) <= 0 && sgn(c3) * sgn(c4) <= 0;  // 若非规范相交不算相交,则去掉等号
    }

    namespace HalfPlaneIntersection {
        Point points[MAXN];  // 多边形顶点
        Line lines[MAXN];  // 多边形的边所在直线
        int cnt = 0;  // 直线数
        int que[MAXN], hh, tt;  // 双端队列
        Point hull[MAXN];  // 凸包上的点

        // 半平面交
        bool isPointOnRight(Line A, Line B, Line C) {
            auto tmp = getLineIntersection(B, C);
            return sgn(getArea(A.p, A.p + A.v, tmp)) < 0;  // 注意不取等
        }

        void half_plane_intersection() {  // 求半平面交
            hh = 0, tt = -1;

```

```
            sort(lines, lines + cnt, [&](const Line& A, const Line& B) {
                double a = getAngle(A), b = getAngle(B);
                if (!cmp(a, b)) return getArea(A.p, A.p + A.v, B.p) < 0;
                else return a < b;
            });  // 直线按倾斜角升序排列

            for (int i = 0; i < cnt; i++) {
                if (i && !cmp(getAngle(lines[i]), getAngle(lines[i - 1]))) continue;

                while (hh + 1 <= tt && isPointOnRight(lines[i], lines[que[tt - 1]], lines[que[tt]])) tt--;  // 检查队尾
                while (hh + 1 <= tt && isPointOnRight(lines[i], lines[que[hh]], lines[que[hh + 1]])) hh++;  // 检查队首

                que[++tt] = i;  // 当前点入队
            }

            while (hh + 1 <= tt && isPointOnRight(lines[que[hh]], lines[que[tt - 1]], lines[que[tt]])) tt--;  // 用队首更新队尾
            while (hh + 1 <= tt && isPointOnRight(lines[que[tt]], lines[que[hh]], lines[que[hh + 1]])) hh++;  // 用队尾更新队首

            vi res;
            for (int i = hh; i <= tt; i++)
                for (auto id : lines[que[i]].ids) res.push_back(id);

            sort(all(res));
            cout << res.size() << endl;
            for (int i = 0; i < res.size(); i++) cout << res[i] << " \n"[i == res.size() - 1];
        }
    }
};
using namespace Geometry_2D;
using namespace HalfPlaneIntersection;

map<pii, vi> mp;  // 直线对应的车的编号

void solve() {
    int n; cin >> n;
    for (int i = 1; i <= n; i++) cin >> k[i];
    for (int i = 1; i <= n; i++) cin >> v[i];

    for (int i = 1; i <= n; i++) mp[{k[i], v[i]}].push_back(i);

    lines[cnt++] = { {0, 0},{0, -1} }, lines[cnt++] = { {0, 0},{1, 0} };  // 指向y轴负向、x轴正向的向量
    for (auto& [p, idx] : mp) {
        Point A = { 0,p.first }, B = { 1,p.first + p.second };
        lines[cnt++] = { A,B - A,idx };
    }

    half_plane_intersection();
}

int main() {
    solve();
}
```

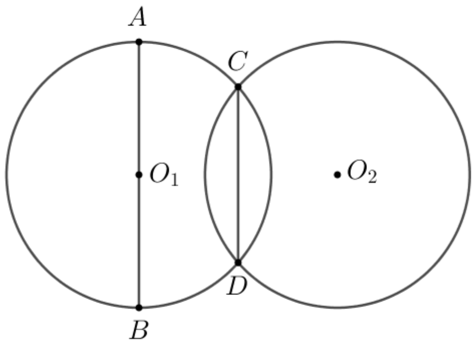
22.1.7 最小圆覆盖

题意

在二维平面上给定$n\ \ (2 \leq n \leq 1e5)$个点$(x_i, y_i)\ \ (-10000.0 \leq x_i, y_i \leq 10000.0)$,求一个最小的包含所有点的圆,输出半径和圆心,保留10位小数.

思路

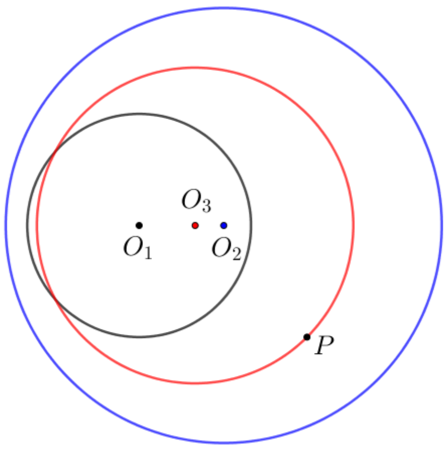
[**性质1**] 满足性质的最小圆唯一.



[**证**] 若不然,设存在如上图所示的两半径相等的圆O_1, O_2都可覆盖所有点,则所有点都在两圆的交集中.

取两圆的公共弦CD,显然其长度$<$直径AB的长度,以CD为直径的圆即可覆盖所有点,与圆O_1, O_2是最小圆覆盖矛盾.

[**性质2**] 若点P不在点集S的最小圆覆盖的内部,则P在点集$\{P\} \bigcup S$的边界上.



[**证**] 若不然,设O_1是点集S的最小覆盖圆,O_2是点集$\{P\} \bigcup S$的最小覆盖圆,点P在O_1的外部、O_2的内部.

显然$r_1 \leq r_2$,因S的最小圆覆盖唯一,故$r_1 < r_2$.

将圆O_2缩小,过程中保证O_2始终覆盖点集S的点,则最终圆O_2会缩小为圆O_1.

因点P在O_1的外部、O_2的内部,则缩小过程中存在一个时刻使得P在O_2的边界上.

设此时的圆为圆O_3.显然O_3是点集$\{P\} \bigcup S$的覆盖圆,且$r_3 \leq r_2$.

①若$r_3 = r_2$,类似于**性质1**的证明过程,可找到一个更小的圆覆盖,与圆O_2是最小圆覆盖矛盾.

②若$r_3 < r_2$,也与圆O_2是最小圆覆盖矛盾.

[**随机增量法**] 找出最小圆覆盖边界上的三个不共线的点.

步骤:

(1)将点集随机化.

(2)从前往后枚举每个点,设当前枚举到第$i\ \ (1\leq i\leq n)$个点:

```
for (int i = 2; i <= n; i++)
```

①$i=1$时,一个点的最小圆覆盖即本身.

②$i\geq 2$时,当前已确定前$(i-1)$个点的最小圆覆盖O_1.若此时第i个点已在O_1的内部,则跳过;否则第i个点在O_1的边界上,即找到了最小圆覆盖边界上的一个点.

③因最优解中第i个点在最小圆覆盖边界上,故只需找到使得第i个点在圆边界上的最小圆.

从第i个点的最小覆盖圆,即它本身开始枚举,设当前最小圆覆盖为第i个点.

枚举第$j\in[1,i-1]$个点:

```
for (int j = 1; j < i; j++)
```

设当前已求得能覆盖第$1\sim(j-1)$个点和第i个点,且第i个点在圆边界上的最小圆O_2.同理可证圆O_2也具有**性质1**和**性质2**.

现加入第j个点,若它在O_2的内部则跳过;否则它在能覆盖第$1\sim j$个点和第i个点,且第i个点在O_2边界上的最小圆O_3,同时第j个点也在O_3的边界上.同理可证圆O_3也具有**性质1**和**性质2**.

设当前最小圆覆盖为以第i、j个点为直径的圆.

枚举第$k\in[1,j-1]$个点:

```
for (int k = 1; k < j; k++)
```

设当前已求得能覆盖第$1\sim(k-1)$个点和第i、j个点,且第i、j个点在圆O_3边界上的最小圆O_4.同理可证圆O_4也具有**性质1**和**性质2**.

先加入第k个点,显然只需考虑第k个点在圆O_4外部的情况,此时它在能覆盖第$1\sim k$个点和第i、j个点,且第i、j个点在圆O_3边界上的最小圆O_5的边界上,此时点i、j、k确定一个圆.

④当$k=j-1$时,已求得能覆盖第$1\sim(j-1)$个点,且第i、j个点在圆边界上的最小圆覆盖.

⑤当$j=i-1$时,已求得能覆盖第$1\sim(i-1)$个点,且第i个点在圆边界上的最小圆覆盖,即第$1\sim i$个点的最小圆覆盖.

⑥当$i=n$时,即求得初始点集的最小圆覆盖.

上述算法的期望时间复杂度$O(n)$.

[**证**] 注意到不共线的三点确定一个圆,则并不是每次都会走到内层循环.

最内层循环时间复杂度$O(n)$.在n个点中,只有$\dfrac{3}{n}$的概率会枚举到最小圆覆盖边界上的点,则第二层循环的时间复杂度为$O(n)+\dfrac{3}{n}O(n)=O(n)$.同理第一层循环时间复杂度也为$O(n)$.

用三角形三边中垂线的交点来确定外接圆圆心,用圆心到任一顶点的距离来确定外接圆半径.

代码

```cpp
const int MAXN = 1e5 + 5;
int n;
namespace Geometry_2D {
    // 点类
    struct Point {
        double x, y;

        Point(double _x = 0, double _y = 0) :x(_x), y(_y) {}

        bool operator==(const Point& B)const { return cmp(x, B.x) == 0 && cmp(y, B.y) == 0; }
        bool operator<(const Point& B)const { return x == B.x ? y < B.y : x < B.x; }  // 按x升序排列,x相同时按y升序排列
        friend ostream& operator<<(ostream& out, const Point p) {
            out << '[' << p.x << ',' << p.y << ']';
            return out;
        }
    }points[MAXN];
    double getDistance(Point A, Point B) { return hypot(A.x - B.x, A.y - B.y); }

    // 向量类
    typedef Point Vector;
    Vector operator+(Vector A, Vector B) { return Vector(A.x + B.x, A.y + B.y); }
    Vector operator-(Vector A, Vector B) { return Vector(A.x - B.x, A.y - B.y); }
    Vector operator*(Vector A, double k) { return Vector(A.x * k, A.y * k); }
    Vector operator/(Vector A, double k) { return Vector(A.x / k, A.y / k); }
    double operator*(Vector A, Vector B) { return A.x * B.x + A.y * B.y; }  // 点乘
    double operator^(Vector A, Vector B) { return A.x * B.y - A.y * B.x; }  // 叉乘
    double dotProduct(Vector A, Vector B) { return A.x * B.x + A.y * B.y; }
    double crossProduct(Vector A, Vector B) { return A.x * B.y - A.y * B.x; }
    double getLength(Vector A) { return hypot(A.x, A.y); }  // 模长
    double getAngle(Vector A, Vector B) { return acos(dotProduct(A, B) / getLength(A) / getLength(B)); }  // 夹角(弧度)
    double getArea(Point A, Point B, Point C) { return crossProduct(B - A, C - A); }  // 向量AB与向量AC张成的平行四边形的面积
    double getArea(Vector AB, Vector AC) { return crossProduct(AB, AC); }  // 向量AB与向量AC张成的平行四边形的面积
    Vector rotate(Vector A, double rad) { return Vector(A.x * cos(rad) + A.y * sin(rad), -A.x * sin(rad) + A.y * cos(rad)); }  // 向量A逆时针旋转rad得到的向量
    Vector normal(Vector A) { return Vector(-A.y, A.x); }  // 向量A逆时针旋转90度得到的向量

    // 直线类
    struct Line {
      Point p;  // 直线上一点
        Vector v;  // 方向向量

        Line() {}
        Line(Point _p, Vector _v) :p(_p), v(_v) {}

        Point getPoint(double t) { return p + v * t; }  // 直线上参数为t的点
    };
    double getAngle(Line l) {  // 求直线倾斜角
        Point q = l.p + l.v;
        return atan2(q.y - l.p.y, q.x - l.p.x);
    }
```

```
    bool isPointOnLine(Point P, Point A, Point B) { return sgn(crossProduct(P - A, P - B)) == 0; }  // 点P是否在直线AB上
    bool isPointOnSegment(Point P, Point A, Point B) { return isPointOnLine(P, A, B) && sgn(dotProduct(A - P, B - P)) < 0; }  // 点P是否在线段AB上
    double getDistanceToLine(Point P, Point A, Point B) {  // 点P到直线AB的距离
        Vector v1 = B - A, v2 = P - A;
        return fabs(crossProduct(v1, v2) / getLength(v1));
    }
    double getDistanceToSegment(Point P, Point A, Point B) {  // 点P到线段AB的距离
        if (A == B) return getLength(P - A);
        Vector v1 = B - A, v2 = P - A, v3 = P - B;
        if (sgn(dotProduct(v1, v2)) < 0) return getLength(v2);  // P的投影在线段AB外,且更靠近A
        if (sgn(dotProduct(v1, v3)) < 0) return getLength(v3);  // P的投影在线段AB外,且更靠近B
        return getDistanceToLine(P, A, B);  // P的投影在线段AB上
    }
    Point getLineIntersection(Point P, Vector v, Point Q, Vector w) {  // 求直线(P+vt)与(Q+vw)的交点,使用前需保证有交点
        Vector u = P - Q;
        double t = crossProduct(w, u) / crossProduct(v, w);
        return P + v * t;
    }
    Point getLineIntersection(Line A, Line B) { return getLineIntersection(A.p, A.v, B.p, B.v); }  // 求直线A与B的交点,使用前需保证有交点
    Point getPointProjection(Point P, Point A, Point B) {  // 点P在直线AB上的投影
        Vector v1 = B - A, v2 = P - A;
        return A + v1 * (dotProduct(v1, v2) / dotProduct(v1, v1));
    }
    bool isSegmentIntersect(Point A1, Point A2, Point B1, Point B2) {  // 判断线段A1A2和B1B2是否相交
        double c1 = crossProduct(A2 - A1, B1 - A1), c2 = crossProduct(A2 - A1, B2 - A1),
            c3 = crossProduct(B2 - B1, A2 - B1), c4 = crossProduct(B2 - B1, A1 - B1);
        return sgn(c1) * sgn(c2) <= 0 && sgn(c3) * sgn(c4) <= 0;  // 若非规范相交不算相交,则去掉等号
    }

    // 圆类
    struct Circle {
        Point c;  // 圆心
        double r;  // 半径

        Circle(Point _c, double _r = 0) :c(_c), r(_r) {}
    };

    bool isPointInCircle(Circle O, Point A) {  // 判断点A是否在圆O内部
        return cmp(O.r, getDistance(O.c, A)) < 0;
    }

    Line getPerpendicularBisector(Point A, Point B) {  // 求线段AB的中垂线
        return Line((A + B) / 2, rotate(B - A, pi / 2));
    }

    Circle getCircle(Point A, Point B, Point C) {  // 求三角形ABC的外接圆
        auto l1 = getPerpendicularBisector(A, B), l2 = getPerpendicularBisector(A, C);
        auto O = getLineIntersection(l1.p, l1.v, l2.p, l2.v);
        return Circle(O, getDistance(O, A));
    }
```

```cpp

    Circle minimumCircleCoverage() {  // 最小圆覆盖
        random_shuffle(points, points + n);

        Circle c({ points[0], 0});  // 当前圆即第一个点
        for (int i = 1; i < n; i++) {  // 枚举第一个点
            if (isPointInCircle(c, points[i])) {
                c = { points[i],0 };
                for (int j = 0; j < i; j++) {  // 枚举第二个点
                    if (isPointInCircle(c, points[j])) {
                        c = { (points[i] + points[j]) / 2,getDistance(points[i], points[j]) / 2 };  // 以点i、j为直径的圆
                        for (int k = 0; k < j; k++) {
                            if (isPointInCircle(c, points[k]))
                            c = getCircle(points[i], points[j], points[k]);
                        }
                    }
                }
            }
        }
        return c;
    }
};
using namespace Geometry_2D;

void solve() {
    cin >> n;
    for (int i = 0; i < n; i++) cin >> points[i].x >> points[i].y;

    auto ans = minimumCircleCoverage();
    cout << fixed << setprecision(10) << ans.r << endl;
    cout << fixed << setprecision(10) << ans.c.x << ' ' << ans.c.y << endl;
}

int main() {
    solve();
}
```

22.1.8 信号增幅仪

题意

给定$n\ \ (1\leq n\leq 5\mathrm{e}4)$个整点$(x_i,y_i)\ \ (0\leq |x|,|y|\leq 2\mathrm{e}8)$.求一个两焦点连线与$x$轴正方向成$\theta\ \ (0\leq\theta<180)$角度、且长轴长与短轴长之比为$p\ \ (1\leq p\leq 100)$的、短轴长最小的椭圆,使得它可覆盖所有点,输出其短半轴长,四舍五入保留三位小数.

思路

将坐标轴顺时针旋转θ角,此时椭圆的长轴在x轴上.为方便计算,设短半轴长放缩为1,则长半轴长为p,则椭圆方程为$\dfrac{x^2}{p^2}+y^2=1$.

令$x'=\dfrac{x}{p},y'=y$,将原坐标系xOy的中椭圆仿射为新坐标系$x'Oy'$中的圆$x'^2+y'^2=1$.转化为求最小圆覆盖.

代码

```
const int MAXN = 5e4 + 5;
int n;
namespace Geometry_2D {
    // 点类
    struct Point {
        double x, y;

        Point(double _x = 0, double _y = 0) :x(_x), y(_y) {}

        bool operator==(const Point& B)const { return cmp(x, B.x) == 0 && cmp(y, B.y) == 0; }
        bool operator<(const Point& B)const { return x == B.x ? y < B.y : x < B.x; }  // 按x升序排列,x相同时按y升序排列
        friend ostream& operator<<(ostream& out, const Point p) {
            out << '[' << p.x << ',' << p.y << ']';
            return out;
        }
    }points[MAXN];
    double getDistance(Point A, Point B) { return hypot(A.x - B.x, A.y - B.y); }

    // 向量类
    typedef Point Vector;
    Vector operator+(Vector A, Vector B) { return Vector(A.x + B.x, A.y + B.y); }
    Vector operator-(Vector A, Vector B) { return Vector(A.x - B.x, A.y - B.y); }
    Vector operator*(Vector A, double k) { return Vector(A.x * k, A.y * k); }
    Vector operator/(Vector A, double k) { return Vector(A.x / k, A.y / k); }
    double operator*(Vector A, Vector B) { return A.x * B.x + A.y * B.y; }  // 点乘
    double operator^(Vector A, Vector B) { return A.x * B.y - A.y * B.x; }  // 叉乘
    double dotProduct(Vector A, Vector B) { return A.x * B.x + A.y * B.y; }
    double crossProduct(Vector A, Vector B) { return A.x * B.y - A.y * B.x; }
    double getLength(Vector A) { return hypot(A.x, A.y); }  // 模长
    double getAngle(Vector A, Vector B) { return acos(dotProduct(A, B) / getLength(A) / getLength(B)); }  // 夹角(弧度)
    double getArea(Point A, Point B, Point C) { return crossProduct(B - A, C - A); }  // 向量AB与向量AC张成的平行四边形的面积
    double getArea(Vector AB, Vector AC) { return crossProduct(AB, AC); }  // 向量AB与向量AC张成的平行四边形的面积
    Vector rotate(Vector A, double rad) { return Vector(A.x * cos(rad) + A.y * sin(rad), -A.x * sin(rad) + A.y * cos(rad)); }  // 向量A逆时针旋转rad得到的向量
    Vector normal(Vector A) { return Vector(-A.y, A.x); }  // 向量A逆时针旋转90度得到的向量

    // 直线类
    struct Line {
      Point p;  // 直线上一点
        Vector v;  // 方向向量

        Line() {}
        Line(Point _p, Vector _v) :p(_p), v(_v) {}

        Point getPoint(double t) { return p + v * t; }  // 直线上参数为t的点
    };
    double getAngle(Line l) {  // 求直线倾斜角
        Point q = l.p + l.v;
        return atan2(q.y - l.p.y, q.x - l.p.x);
    }
```

```
    bool isPointOnLine(Point P, Point A, Point B) { return sgn(crossProduct(P - A, P - B)) == 0; }  // 点P是否在直线AB上
    bool isPointOnSegment(Point P, Point A, Point B) { return isPointOnLine(P, A, B) && sgn(dotProduct(A - P, B - P)) < 0; }  // 点P是否在线段AB上
    double getDistanceToLine(Point P, Point A, Point B) {  // 点P到直线AB的距离
        Vector v1 = B - A, v2 = P - A;
        return fabs(crossProduct(v1, v2) / getLength(v1));
    }
    double getDistanceToSegment(Point P, Point A, Point B) {  // 点P到线段AB的距离
        if (A == B) return getLength(P - A);
        Vector v1 = B - A, v2 = P - A, v3 = P - B;
        if (sgn(dotProduct(v1, v2)) < 0) return getLength(v2);  // P的投影在线段AB外,且更靠近A
        if (sgn(dotProduct(v1, v3)) < 0) return getLength(v3);  // P的投影在线段AB外,且更靠近B
        return getDistanceToLine(P, A, B);  // P的投影在线段AB上
    }
    Point getLineIntersection(Point P, Vector v, Point Q, Vector w) {  // 求直线(P+vt)与(Q+vw)的交点,使用前需保证有交点
        Vector u = P - Q;
        double t = crossProduct(w, u) / crossProduct(v, w);
        return P + v * t;
    }
    Point getLineIntersection(Line A, Line B) { return getLineIntersection(A.p, A.v, B.p, B.v); }  // 求直线A与B的交点,使用前需保证有交点
    Point getPointProjection(Point P, Point A, Point B) {  // 点P在直线AB上的投影
        Vector v1 = B - A, v2 = P - A;
        return A + v1 * (dotProduct(v1, v2) / dotProduct(v1, v1));
    }
    bool isSegmentIntersect(Point A1, Point A2, Point B1, Point B2) {  // 判断线段A1A2和B1B2是否相交
        double c1 = crossProduct(A2 - A1, B1 - A1), c2 = crossProduct(A2 - A1, B2 - A1),
            c3 = crossProduct(B2 - B1, A2 - B1), c4 = crossProduct(B2 - B1, A1 - B1);
        return sgn(c1) * sgn(c2) <= 0 && sgn(c3) * sgn(c4) <= 0;  // 若非规范相交不算相交,则去掉等号
    }

    // 圆类
    struct Circle {
        Point c;  // 圆心
        double r;  // 半径

        Circle(Point _c, double _r = 0) :c(_c), r(_r) {}
    };

    bool isPointInCircle(Circle O, Point A) {  // 判断点A是否在圆O内部
        return cmp(O.r, getDistance(O.c, A)) < 0;
    }

    Line getPerpendicularBisector(Point A, Point B) {  // 求线段AB的中垂线
        return Line((A + B) / 2, rotate(B - A, pi / 2));
    }

    Circle getCircle(Point A, Point B, Point C) {  // 求三角形ABC的外接圆
        auto l1 = getPerpendicularBisector(A, B), l2 = getPerpendicularBisector(A, C);
        auto O = getLineIntersection(l1.p, l1.v, l2.p, l2.v);
        return Circle(O, getDistance(O, A));
    }
```

```cpp

    Circle minimumCircleCoverage() {  // 最小圆覆盖
        random_shuffle(points, points + n);

        Circle c({ points[0], 0});  // 当前圆即第一个点
        for (int i = 1; i < n; i++) {  // 枚举第一个点
            if (isPointInCircle(c, points[i])) {
                c = { points[i],0 };
                for (int j = 0; j < i; j++) {  // 枚举第二个点
                    if (isPointInCircle(c, points[j])) {
                        c = { (points[i] + points[j]) / 2,getDistance(points[i], points[j]) / 2 };  // 以点i、j为直径的圆
                        for (int k = 0; k < j; k++) {
                            if (isPointInCircle(c, points[k]))
                            c = getCircle(points[i], points[j], points[k]);
                        }
                    }
                }
            }
        }
        return c;
    }
};
using namespace Geometry_2D;

void solve() {
    cin >> n;
    for (int i = 0; i < n; i++) cin >> points[i].x >> points[i].y;
    double theta, p; cin >> theta >> p;

    for (int i = 0; i < n; i++) {
        points[i] = rotate(points[i], theta / 180 * pi);  // 将坐标轴顺时针旋转theta角
        points[i].x /= p;  // 将椭圆仿射为圆
    }

    auto ans = minimumCircleCoverage();
    cout << fixed << setprecision(3) << ans.r << endl;
}

int main() {
    solve();
}
```

22.1.9 周游世界

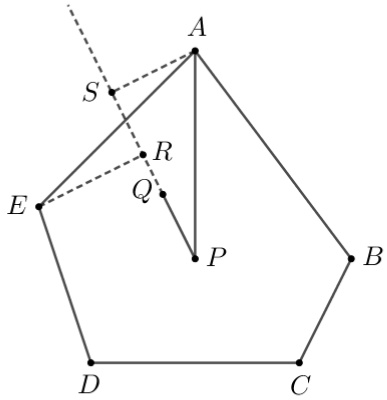
题意

给定平面上的$n\ \ (2\leq n\leq 5\mathrm{e}4)$个相异的整点$(x_i,y_i)\ \ (-1\mathrm{e}4\leq x_i,y_i\leq 1\mathrm{e}4)$,求相距最远的两点的距离的平方.

思路

平面最近点对可用分治求得,平面最远点对可用旋转卡壳求得.

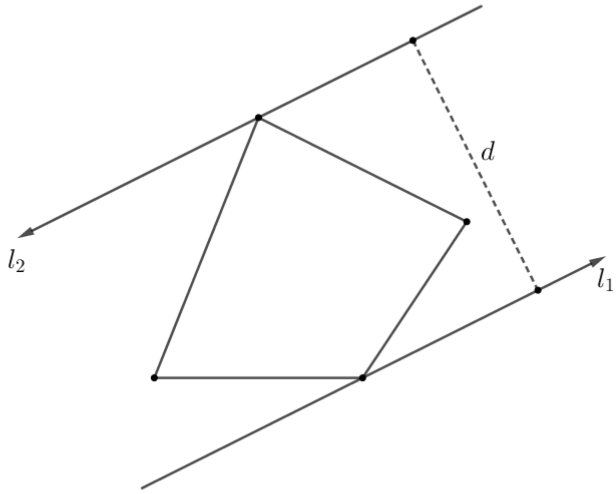
平面最远点对必在凸包上.



[**证**] 如上图,设凸包$ABCDE$,凸包内部的两点PQ是平面最远点对.

过A、E分别作射线PQ的垂线AS、ER,则$\overline{PQ} < \overline{PS} < \overline{PA}$,与$PQ$是平面最远点对矛盾.

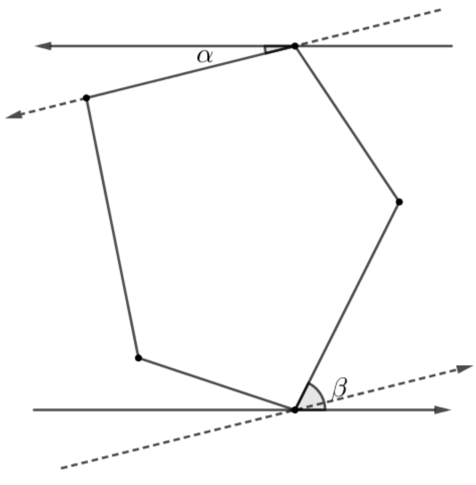
这表明:平面最远点对的一个点在凸包上.同理可证另一个点也在凸包上.



求平面最远点对转化为求凸包的直径,可用旋转卡壳求得.具体地,如上图,用两条平行线卡住凸包,逆时针旋转两平行线,过程中两平行线间的距离的最大值即凸包直径.

因角度连续,故枚举直线的角度计算量过大.考虑枚举两平行线与凸包的两交点,称它们为一对对踵点,则所有对踵点间的距离的最大值即凸包直径.

显然平行线旋转过程中对踵点对可能发生变化,故只需枚举对踵点发生变化的角度即可.



如上图,因$\alpha < \beta$,则上方的平行线会先与凸包的边重合,重合的时刻即对踵点改变的时刻.

为找到所有对踵点对,只需找到离凸包的每条边最远的凸包的顶点.可无需区分变化前和变化后的对踵点对,直接用两对点更新答案即可.

考虑如何快速求离凸包的每条边最远的凸包的顶点.注意到对一条固定的边,凸包上的点到其的距离先增加再减小,要求的点即取得距离峰值对应的点.因两平行线每次旋转相同角度,则对踵点对的两点都会单调地旋转,可用双指针快速求.

对一条固定的边,可通过它与凸包上另一点围成的三角形的面积来反映该点离该边的距离的大小.

当凸包只有两个点,即所有点共线时,凸包直径即第一个点和最后一点的距离.注意

该算法的瓶颈在求凸包上,用Andrew算法求凸包时总时间复杂度$O(n\log n)$.

代码

```
const int MAXN = 5e4 + 5;
int n;
namespace Geometry_2D {
    // 点类
    struct Point {
        double x, y;

        Point(double _x = 0, double _y = 0) :x(_x), y(_y) {}

        bool operator==(const Point& B)const { return cmp(x, B.x) == 0 && cmp(y, B.y) == 0; }
        bool operator<(const Point& B)const { return x == B.x ? y < B.y : x < B.x; }  // 按x升序排列,x相同时按y升序排列
        friend ostream& operator<<(ostream& out, Point& P) {
            out << '[' << P.x << ',' << P.y <<']';
            return out;
        }
    }points[MAXN];
    double getDistance(Point A, Point B) { return hypot(A.x - B.x, A.y - B.y); }
    double getDistance2(Point A, Point B) { return (A.x - B.x) * (A.x - B.x) + (A.y - B.y) * (A.y - B.y); }

    // 向量类
    typedef Point Vector;
    Vector operator+(Vector A, Vector B) { return Vector(A.x + B.x, A.y + B.y); }
    Vector operator-(Vector A, Vector B) { return Vector(A.x - B.x, A.y - B.y); }
    Vector operator*(Vector A, double k) { return Vector(A.x * k, A.y * k); }
    Vector operator/(Vector A, double k) { return Vector(A.x / k, A.y / k); }
    double operator*(Vector A, Vector B) { return A.x * B.x + A.y * B.y; }  // 点乘
    double operator^(Vector A, Vector B) { return A.x * B.y - A.y * B.x; }  // 叉乘
    double dotProduct(Vector A, Vector B) { return A.x * B.x + A.y * B.y; }
    double crossProduct(Vector A, Vector B) { return A.x * B.y - A.y * B.x; }
    double getLength(Vector A) { return hypot(A.x, A.y); }  // 模长
    double getAngle(Vector A, Vector B) { return acos(dotProduct(A, B) / getLength(A) / getLength(B)); }  // 夹角(弧度)
    double getArea(Point A, Point B, Point C) { return crossProduct(B - A, C - A); }  // 向量AB与向量AC张成的平行四边形的面积
    double getArea(Vector AB, Vector AC) { return crossProduct(AB, AC); }  // 向量AB与向量AC张成的平行四边形的面积
    Vector rotate(Vector A, double rad) { return Vector(A.x * cos(rad) + A.y * sin(rad), -A.x * sin(rad) + A.y * cos(rad)); }  // 向量A逆时针旋转rad得到的向量
    Vector normal(Vector A) { return Vector(-A.y, A.x); }  // 向量A逆时针旋转90度得到的向量

    // 直线类
    struct Line {
      Point p;  // 直线上一点
        Vector v;  // 方向向量
```

```

        Line() {}
        Line(Point _p, Vector _v) :p(_p), v(_v) {}

        Point getPoint(double t) { return p + v * t; }  // 直线上参数为t的点
    };
    double getAngle(Line l) {  // 求直线倾斜角
        Point q = l.p + l.v;
        return atan2(q.y - l.p.y, q.x - l.p.x);
    }
    bool isPointOnLine(Point P, Point A, Point B) { return sgn(crossProduct(P - A, P - B)) == 0; }  // 点P是否在直线AB上
    bool isPointOnSegment(Point P, Point A, Point B) { return isPointOnLine(P, A, B) && sgn(dotProduct(A - P, B - P)) < 0; }  // 点P是否在线段AB上
    double getDistanceToLine(Point P, Point A, Point B) {  // 点P到直线AB的距离
        Vector v1 = B - A, v2 = P - A;
        return fabs(crossProduct(v1, v2) / getLength(v1));
    }
    double getDistanceToSegment(Point P, Point A, Point B) {  // 点P到线段AB的距离
        if (A == B) return getLength(P - A);
        Vector v1 = B - A, v2 = P - A, v3 = P - B;
        if (sgn(dotProduct(v1, v2)) < 0) return getLength(v2);  // P的投影在线段AB外,且更靠近A
        if (sgn(dotProduct(v1, v3)) < 0) return getLength(v3);  // P的投影在线段AB外,且更靠近B
        return getDistanceToLine(P, A, B);  // P的投影在线段AB上
    }
    Point getLineIntersection(Point P, Vector v, Point Q, Vector w) {  // 求直线(P+vt)与(Q+vw)的交点,使用前需保证有交点
        Vector u = P - Q;
        double t = crossProduct(w, u) / crossProduct(v, w);
        return P + v * t;
    }
    Point getLineIntersection(Line A, Line B) { return getLineIntersection(A.p, A.v, B.p, B.v); }  // 求直线A与B的交点,使用前需保证有交点
    Point getPointProjection(Point P, Point A, Point B) {  // 点P在直线AB上的投影
        Vector v1 = B - A, v2 = P - A;
        return A + v1 * (dotProduct(v1, v2) / dotProduct(v1, v1));
    }
    bool isSegmentIntersect(Point A1, Point A2, Point B1, Point B2) {  // 判断线段A1A2和B1B2是否相交
        double c1 = crossProduct(A2 - A1, B1 - A1), c2 = crossProduct(A2 - A1, B2 - A1),
            c3 = crossProduct(B2 - B1, A2 - B1), c4 = crossProduct(B2 - B1, A1 - B1);
        return sgn(c1) * sgn(c2) <= 0 && sgn(c3) * sgn(c4) <= 0;  // 若非规范相交不算相交,则去掉等号
    }

    namespace ConvexHull {
        int stk[MAXN], top;
        bool used[MAXN];  // 记录每个点是否被用过

        void andrew() {  // 求凸包
            top = 0;

            sort(points + 1, points + n + 1);

            stk[++top] = 1;  // 第一个点一定在凸包中
            for (int i = 2; i <= n; i++) {  // 求下半凸包
```

```
                while (top >= 2 && getArea(points[stk[top - 1]], points[stk[top]], points[i]) <= 0) {
                    if (getArea(points[stk[top - 1]], points[stk[top]], points[i]) < 0) used[stk[top--]] = false;
                    else top--;
                }

                stk[++top] = i, used[i] = true;  // 当前点入栈
            }

            int upsiz = top;  // 下半凸包的大小
            for (int i = n - 1; i >= 1; i--) {  // 求上半凸包
                if (!used[i]) {
                    while (top > upsiz && getArea(points[stk[top - 1]], points[stk[top]], points[i]) <= 0)
                        top--;  // 后续不会用到已出栈的点,故无需更新used[]

                    stk[++top] = i;
                }
            }
            top--;
        }

        int rotating_calipers() {  // 求旋转卡壳,返回凸包直径的平方
            andrew();

            if (top <= 2) return getDistance2(points[1], points[n]);  // 所有点共线

            int res = 0;
            for (int i = 1, j = 2; i <= top; i++) {  // 枚举第i条边、第j个点
                auto u = points[stk[i]], v = points[stk[i + 1]];
                while (getArea(u, v, points[stk[j]]) < getArea(u, v, points[stk[j + 1]])) j = j % top + 1;
                res = max({ res,(int)getDistance2(u, points[stk[j]]),(int)getDistance2(v, points[stk[j]]) });
            }
            return res;
        }
    }
}
using namespace Geometry_2D;
using namespace ConvexHull;

void solve() {
    cin >> n;
    for (int i = 1; i <= n; i++) cin >> points[i].x >> points[i].y;

    cout << rotating_calipers();
}

int main() {
    solve();
}
```

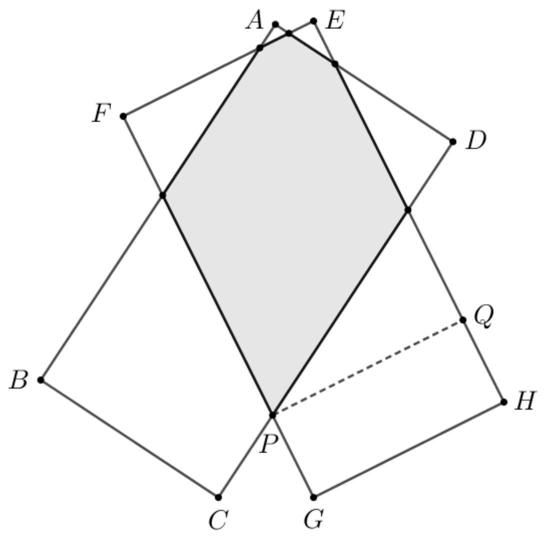
22.1.10 最小矩形覆盖

题意

给定平面上$n\ \ (3\leq n\leq 5\mathrm{e}4)$个点$(x_i,y_i)\ \ (1\leq i\leq n)$,求一个能覆盖所有点的面积最小的矩形,输出其面积并先输出y最小的顶点,再按逆时针顺序输出其四个顶点.若存在y相同的坐标,先输出x小的顶点.可以证明最小矩形覆盖唯一.

思路

最小矩形覆盖是唯一的.

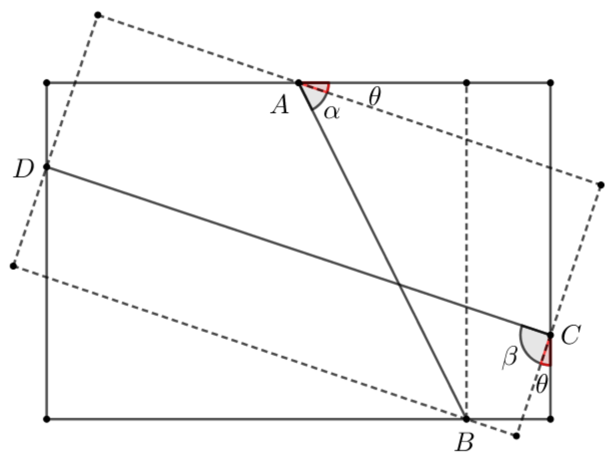


[**证**] 若不然,设矩形$ABCD$和矩形$EFGH$都是最小矩形覆盖,则所有点都在两者的交集中.

作$PQ\perp EH$于点Q,显然矩形$EFPQ$也是矩形覆盖,且其面积小于矩形$ABCD$的面积,与矩形$ABCD$是最小矩形覆盖矛盾.

显然最小矩形覆盖的每条边上都有凸包的顶点.

最小矩形覆盖中至少一条边与凸包的边共线.



[**证**] 若最小矩形覆盖的所有边上都只有一个点,如上图,设AB、CD分别与过A、B、C、D四点的矩形成$(\alpha+\theta)$、$(\beta+\theta)$角.

保持矩形过A、B、C、D四点,将矩形顺时针旋转θ.

设$x=\overline{AB},y=\overline{CD}$,则原矩形的面积为$x\sin{(\alpha+\theta)}\cdot y\sin{(\beta+\theta)}$,新矩形的面积为$x\sin\alpha\cdot y\beta$.

设$f(x)=\sin{(\alpha-x)}\sin{(\beta-x)}$,则$f'(x)=-\sin{(\alpha+\beta-2x)},f'(0)=-\sin{(\alpha+\beta)}$.

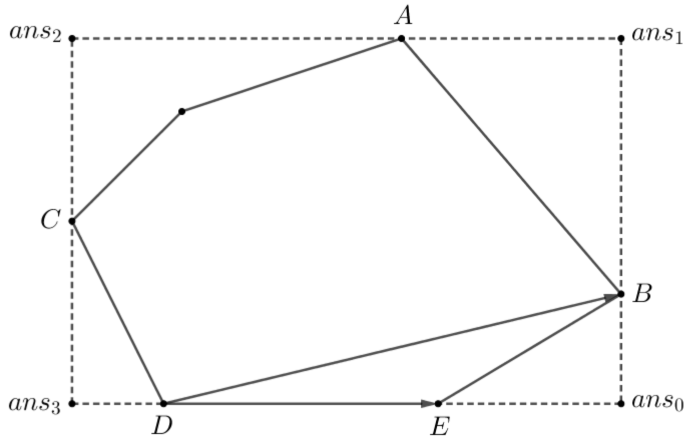
①$\alpha+\beta\leq\pi$时,$f'(0)\leq 0$,顺时针旋转可使得矩形面积减小.

②$\alpha+\beta<\pi$时,$f'(0)>0$,逆时针旋转可使得矩形面积减小.

注意矩形不能无限旋转,旋转至某一时刻时矩形会被凸包上的另一顶点卡住,此时矩形的一条边上有两个凸包上的顶点,即矩形的一条边与凸包的一条边共线.

注意到确定矩形的边经过哪些凸包上的顶点仍不足以唯一确定矩形,但确定矩形的一条边与凸包的一条边共线后可唯一确定矩形,故可枚举所有存在与一条与凸包的边共线的边的矩形即可.

对凸包的每条边确定的矩形,矩形的高用三角形的面积除以底求得,矩形的宽即向量在宽上的投影,用点积计算.显然点的移动是单调的,可用旋转卡壳求.本题中给定的三点至少构成一个三角形,进而至少存在一个对应的矩形,故无需判断所有点共线的情况.



$ans_0=D+\overrightarrow{DE}\cdot\dfrac{\overrightarrow{DB}\cdot\overrightarrow{DE}}{\left|\overrightarrow{DE}\right|}$,$ans_1=ans_0+h\cdot\vec{u}$,其中$\vec{u}$是垂直于$\overrightarrow{DE}$的单位向量.

代码

```
const int MAXN = 5e4 + 5;
int n;
namespace Geometry_2D {
    // 点类
    struct Point {
        double x, y;

        Point(double _x = 0, double _y = 0) :x(_x), y(_y) {}

        bool operator==(const Point& B)const { return cmp(x, B.x) == 0 && cmp(y, B.y) == 0; }
        bool operator<(const Point& B)const { return x == B.x ? y < B.y : x < B.x; }  // 按x升序排列,x相同时按y升序排列
        friend ostream& operator<<(ostream& out, Point& P) {
            out << '[' << P.x << ',' << P.y <<']';
            return out;
        }
    }points[MAXN];
    double getDistance(Point A, Point B) { return hypot(A.x - B.x, A.y - B.y); }
    double getDistance2(Point A, Point B) { return (A.x - B.x) * (A.x - B.x) + (A.y - B.y) * (A.y - B.y); }

    // 向量类
    typedef Point Vector;
```

```
    Vector operator+(Vector A, Vector B) { return Vector(A.x + B.x, A.y + B.y); }
    Vector operator-(Vector A, Vector B) { return Vector(A.x - B.x, A.y - B.y); }
    Vector operator*(Vector A, double k) { return Vector(A.x * k, A.y * k); }
    Vector operator/(Vector A, double k) { return Vector(A.x / k, A.y / k); }
    double operator*(Vector A, Vector B) { return A.x * B.x + A.y * B.y; }  // 点乘
    double operator^(Vector A, Vector B) { return A.x * B.y - A.y * B.x; }  // 叉乘
    double dotProduct(Vector A, Vector B) { return A.x * B.x + A.y * B.y; }
    double crossProduct(Vector A, Vector B) { return A.x * B.y - A.y * B.x; }
    double getLength(Vector A) { return hypot(A.x, A.y); }  // 模长
    double getAngle(Vector A, Vector B) { return acos(dotProduct(A, B) / getLength(A) / getLength(B)); }  // 夹角(弧度)
    double getArea(Point A, Point B, Point C) { return crossProduct(B - A, C - A); }  // 向量AB与向量AC张成的平行四边形的面积
    double getArea(Vector AB, Vector AC) { return crossProduct(AB, AC); }  // 向量AB与向量AC张成的平行四边形的面积
    Vector rotate(Vector A, double rad) { return Vector(A.x * cos(rad) + A.y * sin(rad), -A.x * sin(rad) + A.y * cos(rad)); }  // 向量A逆时针旋转rad得到的向量
    Vector normal(Vector A) { return Vector(-A.y, A.x); }  // 向量A逆时针旋转90度得到的向量
    double getProjection(Point A, Point B, Point C) { return (B - A) * (C - A) / getLength(B - A); }  // 求向量AC在向量BC上的投影
    Vector getUnitVector(Vector A) { return A / getLength(A); }  // 求向量A方向上的单位向量

    // 直线类
    struct Line {
      Point p;  // 直线上一点
        Vector v;  // 方向向量

        Line() {}
        Line(Point _p, Vector _v) :p(_p), v(_v) {}

        Point getPoint(double t) { return p + v * t; }  // 直线上参数为t的点
    };
    double getAngle(Line l) {  // 求直线倾斜角
        Point q = l.p + l.v;
        return atan2(q.y - l.p.y, q.x - l.p.x);
    }
    bool isPointOnLine(Point P, Point A, Point B) { return sgn(crossProduct(P - A, P - B)) == 0; }  // 点P是否在直线AB上
    bool isPointOnSegment(Point P, Point A, Point B) { return isPointOnLine(P, A, B) && sgn(dotProduct(A - P, B - P)) < 0; }  // 点P是否在线段AB上
    double getDistanceToLine(Point P, Point A, Point B) {  // 点P到直线AB的距离
        Vector v1 = B - A, v2 = P - A;
        return fabs(crossProduct(v1, v2) / getLength(v1));
    }
    double getDistanceToSegment(Point P, Point A, Point B) {  // 点P到线段AB的距离
        if (A == B) return getLength(P - A);
        Vector v1 = B - A, v2 = P - A, v3 = P - B;
        if (sgn(dotProduct(v1, v2)) < 0) return getLength(v2);  // P的投影在线段AB外,且更靠近A
        if (sgn(dotProduct(v1, v3)) < 0) return getLength(v3);  // P的投影在线段AB外,且更靠近B
        return getDistanceToLine(P, A, B);  // P的投影在线段AB上
    }
    Point getLineIntersection(Point P, Vector v, Point Q, Vector w) {  // 求直线(P+vt)与(Q+vw)的交点,使用前需保证有交点
        Vector u = P - Q;
        double t = crossProduct(w, u) / crossProduct(v, w);
        return P + v * t;
```

```
    }
    Point getLineIntersection(Line A, Line B) { return getLineIntersection(A.p, A.v, B.p, B.v); }  // 求直线A与B的交点,使用前需保证有交点
    Point getPointProjection(Point P, Point A, Point B) {  // 点P在直线AB上的投影
        Vector v1 = B - A, v2 = P - A;
        return A + v1 * (dotProduct(v1, v2) / dotProduct(v1, v1));
    }
    bool isSegmentIntersect(Point A1, Point A2, Point B1, Point B2) {  // 判断线段A1A2和B1B2是否相交
        double c1 = crossProduct(A2 - A1, B1 - A1), c2 = crossProduct(A2 - A1, B2 - A1),
            c3 = crossProduct(B2 - B1, A2 - B1), c4 = crossProduct(B2 - B1, A1 - B1);
        return sgn(c1) * sgn(c2) <= 0 && sgn(c3) * sgn(c4) <= 0;  // 若非规范相交不算相交,则去掉等号
    }

    namespace ConvexHull {
        int stk[MAXN], top;
        bool used[MAXN];  // 记录每个点是否被用过

        void andrew() {  // 求凸包
            top = 0;

            sort(points + 1, points + n + 1);

            stk[++top] = 1;  // 第一个点一定在凸包中
            for (int i = 2; i <= n; i++) {  // 求下半凸包
                while (top >= 2 && getArea(points[stk[top - 1]], points[stk[top]], points[i]) <= 0) {
                    if (getArea(points[stk[top - 1]], points[stk[top]], points[i]) < 0) used[stk[top--]] = false;
                    else top--;
                }

                stk[++top] = i, used[i] = true;  // 当前点入栈
            }

            int upsiz = top;  // 下半凸包的大小
            for (int i = n - 1; i >= 1; i--) {  // 求上半凸包
                if (!used[i]) {
                    while (top > upsiz && getArea(points[stk[top - 1]], points[stk[top]], points[i]) <= 0)
                        top--;  // 后续不会用到已出栈的点,故无需更新used[]

                    stk[++top] = i;
                }
            }
            top--;  // 删除重复的凸包起点
        }

        int rotating_calipers() {  // 求旋转卡壳,返回凸包直径的平方
            andrew();

            if (top <= 2) return getDistance2(points[1], points[n]);  // 所有点共线

            int res = 0;
            for (int i = 1, j = 2; i <= top; i++) {  // 枚举第i条边、第j个点
                auto u = points[stk[i]], v = points[stk[i + 1]];
```

```
                while (getArea(u, v, points[stk[j]]) < getArea(u, v, points[stk[j + 1]])) j = j % top + 1;
                res = max({ res,(int)getDistance2(u, points[stk[j]]),(int)getDistance2(v, points[stk[j]]) });
            }
            return res;
        }

        Point res[4];  // 最小矩形覆盖的四个顶点
        double area = INFF;  // 最小矩形覆盖的面积
        void minimumRectangleCoverage() {  // 求最小矩形覆盖
            andrew();

            for (int i = 1, a = 3, b = 2, c = 3; i <= top; i++) {
                auto u = points[stk[i]], v = points[stk[i + 1]];
                while (cmp(getArea(u, v, points[stk[a]]), getArea(u, v, points[stk[a + 1]])) < 0) a = a % top + 1;  // 找到距离该边最远的点
                while (cmp(getProjection(u, v, points[stk[b]]), getProjection(u, v, points[stk[b + 1]])) < 0) b = b % top + 1;  // 找到投影最大的点
                if (i == 1) c = a;
                while (cmp(getProjection(u, v, points[stk[c]]), getProjection(u, v, points[stk[c + 1]])) > 0) c = c % top + 1;  // 找到投影最小的点

                auto x = points[stk[a]], y = points[stk[b]], z = points[stk[c]];  // 三角形的顶点
                double h = getArea(u, v, x) / getLength(v - u);  // 矩形的高
                double w = ((y - z) * (v - u)) / getLength(v - u);  // 矩形的宽
                if (h * w < area) {
                    area = h * w;
                    res[0] = u + getUnitVector(v - u) * getProjection(u, v, y);
                    res[3] = u + getUnitVector(v - u) * getProjection(u, v, z);
                    auto t = getUnitVector(rotate(v - u, -pi / 2));  // 垂直于底边的单位向量
                    res[1] = res[0] + t * h, res[2] = res[3] + t * h;
                }
            }
        }
    }
}
using namespace Geometry_2D;
using namespace ConvexHull;

void solve() {
    cin >> n;
    for (int i = 1; i <= n; i++) cin >> points[i].x >> points[i].y;

    minimumRectangleCoverage();

    int k = 0;  // y最小且x最小的顶点
    for (int i = 1; i < 4; i++)
        if (cmp(res[i].y, res[k].y) < 0 || (!cmp(res[i].y, res[k].y) && cmp(res[i].x, res[k].x) < 0)) k = i;

    cout << fixed << setprecision(5) << area << endl;
    for (int i = 0; i < 4; i++, k++) {
        auto x = res[k % 4].x, y = res[k % 4].y;
        if (!sgn(x)) x = 0;
        if (!sgn(y)) y = 0;
        cout << fixed << setprecision(5) << x << ' ' << y << endl;
```

```
    }
}

int main() {
    solve();
}
```

22.1.11 望远镜

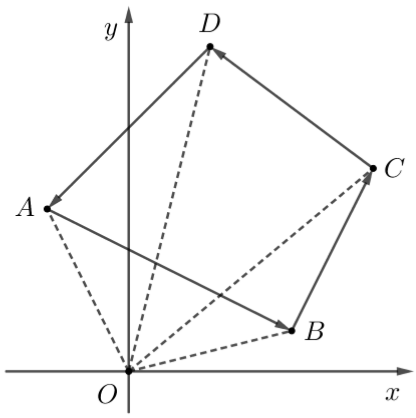
题意

给定平面上一个圆心在原点处的半径为r的圆和一个简单n边形,求两者的交的面积.

有多组测试数据.每组测试数据第一行输入一个实数$r\ \ (0.1\leq r\leq 1000)$.第二行输入一个整数$n\ \ (3\leq n\leq 50)$.接下来$n$行每行输入两个实数$x_i,y_i\ \ (-1000\leq x_i,y_i\leq 1000)$,表示多边形的一个顶点.相邻两行描述的顶点在多边形上也相邻.

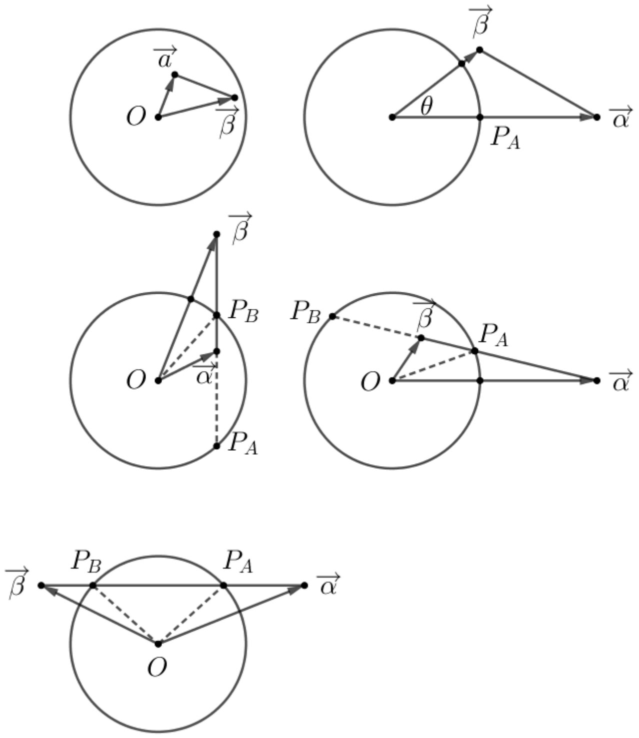
思路

[**求多边形面积的方法**] 将所有的相邻两顶点与原点连线,构成三角形,所有三角形的有向面积之和即多边形的面积.



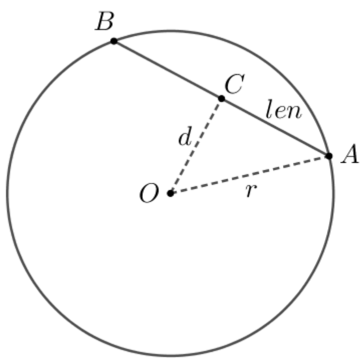
如上图,$S_{ABCD}=S_{\triangle OAD}+S_{\triangle OCD}+S_{\triangle OBC}-S_{\triangle OAB}$.

为求多边形与圆的交集的面积,对多边形三角剖分后求三角形与圆的交集的面积即可.



如上图,三角形与圆的面积交只有上述五种情况,将交集剖分为若干个三角形和扇形的面积和即可,其中三角形的面积可用叉积求,圆心角为θ的扇形的面积为$\dfrac{1}{2}R^2\theta$,其中$\theta = \arccos \dfrac{\overrightarrow{\alpha}\cdot\overrightarrow{\beta}}{\left|\overrightarrow{\alpha}\right|\cdot\left|\overrightarrow{\beta}\right|}$.

[求直线与圆的交点]



如上图,过O作$OC\perp AB$于C,先求$len = \overline{AC}$.设$\overrightarrow{BA}$、$\overrightarrow{AB}$方向的单位向量分别为$\overrightarrow{u}$、$\overrightarrow{v}$,则$A = C + len\overrightarrow{u}, B = C + len\overrightarrow{v}$.

题目输入的多边形顶点的顺序未必是逆时针,故最后要将答案取绝对值.本题中ε取$1\mathrm{e}-8$.

此外,求出面积交后,面积并=面积和−面积交.

代码

```
const int MAXN = 55;
int n;
namespace Geometry_2D {
    // 点类
    struct Point {
        double x, y;

        Point(double _x = 0, double _y = 0) :x(_x), y(_y) {}

```

```
    bool operator==(const Point& B)const { return cmp(x, B.x) == 0 && cmp(y, B.y) == 0; }
    bool operator<(const Point& B)const { return x == B.x ? y < B.y : x < B.x; }  // 按x升序排列,x相同时按y升序排列
    friend ostream& operator<<(ostream& out, Point& P) {
      out << '[' << P.x << ',' << P.y <<']';
      return out;
    }
  }points[MAXN];
  double getDistance(Point A, Point B) { return hypot(A.x - B.x, A.y - B.y); }
  double getDistance2(Point A, Point B) { return (A.x - B.x) * (A.x - B.x) + (A.y - B.y) * (A.y - B.y); }

  // 向量类
  typedef Point Vector;
  Vector operator+(Vector A, Vector B) { return Vector(A.x + B.x, A.y + B.y); }
  Vector operator-(Vector A, Vector B) { return Vector(A.x - B.x, A.y - B.y); }
  Vector operator*(Vector A, double k) { return Vector(A.x * k, A.y * k); }
  Vector operator/(Vector A, double k) { return Vector(A.x / k, A.y / k); }
  double operator*(Vector A, Vector B) { return A.x * B.x + A.y * B.y; }  // 点乘
  double operator^(Vector A, Vector B) { return A.x * B.y - A.y * B.x; }  // 叉乘
  double dotProduct(Vector A, Vector B) { return A.x * B.x + A.y * B.y; }
  double crossProduct(Vector A, Vector B) { return A.x * B.y - A.y * B.x; }
  double getLength(Vector A) { return hypot(A.x, A.y); }  // 模长
  double getAngle(Vector A, Vector B) { return acos(dotProduct(A, B) / getLength(A) / getLength(B)); }  // 夹角(弧度)
  double getArea(Point A, Point B, Point C) { return crossProduct(B - A, C - A); }  // 向量AB与向量AC张成的平行四边形的面积
  double getArea(Vector AB, Vector AC) { return crossProduct(AB, AC); }  // 向量AB与向量AC张成的平行四边形的面积
  Vector rotate(Vector A, double rad) { return Vector(A.x * cos(rad) + A.y * sin(rad), -A.x * sin(rad) + A.y * cos(rad)); }  // 向量A逆时针旋转rad得到的向量
  Vector normal(Vector A) { return Vector(-A.y, A.x); }  // 向量A逆时针旋转90度得到的向量
  double getProjection(Point A, Point B, Point C) { return (B - A) * (C - A) / getLength(B - A); }  // 求向量AC在向量BC上的投影
  Vector getUnitVector(Vector A) { return A / getLength(A); }  // 求向量A方向上的单位向量

  // 直线类
  struct Line {
   Point p;  // 直线上一点
    Vector v;  // 方向向量

    Line() {}
    Line(Point _p, Vector _v) :p(_p), v(_v) {}

    Point getPoint(double t) { return p + v * t; }  // 直线上参数为t的点
  };
  double getAngle(Line l) {  // 求直线倾斜角
    Point q = l.p + l.v;
    return atan2(q.y - l.p.y, q.x - l.p.x);
  }
  bool isPointOnLine(Point P, Point A, Point B) { return sgn(crossProduct(P - A, P - B)) == 0; }  // 点P是否在直线AB上
  bool isPointOnSegment(Point P, Point A, Point B) { return isPointOnLine(P, A, B) && sgn(dotProduct(A - P, B - P)) < 0; }  // 点P是否在线段AB上
  double getDistanceToLine(Point P, Point A, Point B) {  // 点P到直线AB的距离
    Vector v1 = B - A, v2 = P - A;
    return fabs(crossProduct(v1, v2) / getLength(v1));
```

```
    }
    double getDistanceToSegment(Point P, Point A, Point B) {  // 点P到线段AB的距离
        if (A == B) return getLength(P - A);
        Vector v1 = B - A, v2 = P - A, v3 = P - B;
        if (sgn(dotProduct(v1, v2)) < 0) return getLength(v2);  // P的投影在线段AB外,且更靠近A
        if (sgn(dotProduct(v1, v3)) < 0) return getLength(v3);  // P的投影在线段AB外,且更靠近B
        return getDistanceToLine(P, A, B);  // P的投影在线段AB上
    }
    Point getLineIntersection(Point P, Vector v, Point Q, Vector w) {  // 求直线(P+vt)与(Q+vw)的交点,使用前需保证有交点
        Vector u = P - Q;
        double t = crossProduct(w, u) / crossProduct(v, w);
        return P + v * t;
    }
    Point getLineIntersection(Line A, Line B) { return getLineIntersection(A.p, A.v, B.p, B.v); }  // 求直线A与B的交点,使用前需保证有交点
    Point getPointProjection(Point P, Point A, Point B) {  // 点P在直线AB上的投影
        Vector v1 = B - A, v2 = P - A;
        return A + v1 * (dotProduct(v1, v2) / dotProduct(v1, v1));
    }
    bool isSegmentIntersect(Point A1, Point A2, Point B1, Point B2) {  // 判断线段A1A2和B1B2是否相交
        double c1 = crossProduct(A2 - A1, B1 - A1), c2 = crossProduct(A2 - A1, B2 - A1),
            c3 = crossProduct(B2 - B1, A2 - B1), c4 = crossProduct(B2 - B1, A1 - B1);
        return sgn(c1) * sgn(c2) <= 0 && sgn(c3) * sgn(c4) <= 0;  // 若非规范相交不算相交,则去掉等号
    }

    // 圆类
    struct Circle {
        Point c;  // 圆心
        double r;  // 半径

        Circle(Point _c = { 0,0 }, double _r = 0) :c(_c), r(_r) {}
    }cir;
    bool isPointInCircle(Circle o, Point A) { return cmp(o.r, getDistance(o.c, A)) < 0; }  // 判断点A是否在圆O内部
    Line getPerpendicularBisector(Point A, Point B) { return Line((A + B) / 2, rotate(B - A, pi / 2)); }   // 求线段AB的中垂线
    Circle getCircle(Point A, Point B, Point C) {  // 求三角形ABC的外接圆
        auto l1 = getPerpendicularBisector(A, B), l2 = getPerpendicularBisector(A, C);
        auto o = getLineIntersection(l1.p, l1.v, l2.p, l2.v);
        return Circle(o, getDistance(o, A));
    }
    double getCircleLineIntersection(Circle cir, Point A, Point B, Point& PA, Point& PB) {  // 求直线AB与圆cir的交点PA和PB,返回圆心到线段AB的距离
        Point C = getLineIntersection(A, B - A, cir.c, rotate(B - A, pi / 2));  // 垂足
        double mindis = getDistance(cir.c, C);  // 圆心到线段AB的最短距离
        if (!isPointOnSegment(C, A, B)) mindis = min(getDistance(cir.c, A), getDistance(cir.c, B));

        if (cmp(cir.r, mindis) <= 0) return mindis;  // 线段AB与圆无交点

        double len = sqrt(cir.r * cir.r - getDistance(cir.c, C) * getDistance(cir.c, C));  // 半弦长
        PA = C + getUnitVector(A - B) * len, PB = C + getUnitVector(B - A) * len;
```

```
            return mindis;
        }
        double getSectorArea(Circle cir, Point A, Point B) {  // 求扇形OAB的有向面积,其中点A、B未必在圆上
            double theta = acos((A * B) / getLength(A) / getLength(B));  // 圆心角
            if (sgn(A ^ B) < 0) theta = -theta;  // 顺时针为负方向,有向面积为负数
            return cir.r * cir.r * theta / 2;
        }
        double getCircleTriangleArea(Circle cir, Point A, Point B) {  // 求三角形OAB与圆的交集的有向面积,其中点A、B未必在圆上
            double disa = getDistance(cir.c, A), disb = getDistance(cir.c, B);
            if (cmp(cir.r, disa) >= 0 && cmp(cir.r, disb) >= 0) return (A ^ B) / 2;  // 点A、B都在圆内或圆上
            if (!sgn(A ^ B)) return 0;  // O、A、B三点共线

            Point PA, PB;  // 直线AB与圆的交点
            double mindis = getCircleLineIntersection(cir, A, B, PA, PB);
            if (cmp(cir.r, mindis) <= 0) return getSectorArea(cir, A, B);  // 点A、B都在圆外且线段AB与圆无交点
            if (cmp(cir.r, disa) >= 0) return (A ^ PB) / 2 + getSectorArea(cir, PB, B);  // 点A在圆内,点B在圆外
            if (cmp(cir.r, disb) >= 0) return (PA ^ B) / 2 + getSectorArea(cir, A, PA);  // 点A在圆外,点B在圆内
            return (PA ^ PB) / 2 + getSectorArea(cir, A, PA) + getSectorArea(cir, PB, B);  // 点A、B都在圆外且线段AB与圆无交点
        }
}
using namespace Geometry_2D;

void solve() {
    while (cin >> cir.r >> n) {
        for (int i = 0; i < n; i++) cin >> points[i].x >> points[i].y;

        double ans = 0;
        for (int i = 0; i < n; i++) ans += getCircleTriangleArea(cir, points[i], points[(i + 1) % n]);
        cout << fixed << setprecision(2) << fabs(ans) << endl;
    }
}

int main() {
    solve();
}
```

22.1.12 扫描线

题意

给定平面上$n\ \ (1\leq n\leq 1000)$个矩形的左下角顶点(x_1,y_1)和右上角顶点(x_2,y_2),求它们的面积并.

思路

同线段树的扫描线思路,本题数据范围小,无需用线段树维护.

横坐标最多划分为n段,每一段内至多有n个区间,需对区间排序,总时间复杂度$O(n^2 \log n)$.

代码

```
const int MAXN = 1005;
int n;
pii l[MAXN], r[MAXN];  // 矩形的左下角、右上角

ll getArea(int a, int b) {  // 求x∈[a,b]中的矩形的面积并
    vii segs;  // 区间
    for (int i = 0; i < n; i++) {
        if (l[i].first <= a && r[i].first >= b)  // 有交集
            segs.push_back({ l[i].second,r[i].second });
    }

    if (segs.empty()) return 0;  // 无交集

    sort(all(segs));
    ll res = 0;
    int st = segs[0].first, ed = segs[0].second;  // 当前合并区间的左、右端点
    for (auto [x, y] : segs) {
        if (x <= ed) ed = max(ed, y);
        else {
            res += ed - st;
            st = x, ed = y;
        }
    }
    res += ed - st;  // 最后一个区间
    return res * (b - a);
}

void solve() {
    cin >> n;
    vi x;  // 所有顶点的x坐标,用于去重
    for (int i = 0; i < n; i++) {
        cin >> l[i].first >> l[i].second >> r[i].first >> r[i].second;
        x.push_back(l[i].first), x.push_back(r[i].first);
    }

    sort(all(x));
    ll ans = 0;
    for (int i = 0; i + 1 < x.size(); i++) {
        if (x[i] != x[i + 1])  // 去重
            ans += getArea(x[i], x[i + 1]);
    }
    cout << ans;
}

int main() {
    solve();
}
```

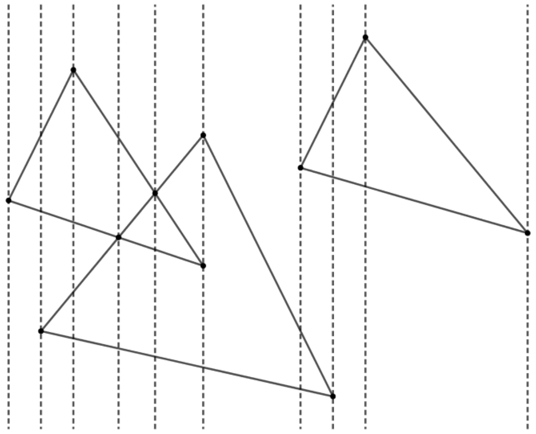
22.1.13 三角形面积并

题意 (2 s)

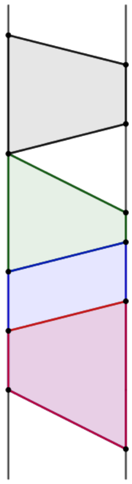
给定平面上$n\ \ (1\leq n\leq 100)$个三角形的三个顶点,求它们的面积并,保留两位小数.三角形的顶点坐标都是不超过1e6的实数.

思路

以所有三角形的顶点和交点将x轴划分为段:



每一段内都是若干个梯形:



由梯形面积公式,梯形的下底和上底分别是左边界上和右边界上区间合并后的区间长度,梯形的高即两边界间的距离.

总时间复杂度$O(n^3\log n)$.

代码

```
const int MAXN = 105;
namespace Geometry_2D {
    // 点类
    struct Point {
        double x, y;

        Point(double _x = 0, double _y = 0) :x(_x), y(_y) {}

        bool operator==(const Point& B)const { return cmp(x, B.x) == 0 && cmp(y, B.y) ==
0; }
        bool operator<(const Point& B)const { return x == B.x ? y < B.y : x < B.x; }  //
按x升序排列,x相同时按y升序排列
```

```
        friend ostream& operator<<(ostream& out, Point& P) {
            out << '[' << P.x << ',' << P.y <<']';
            return out;
        }
    };
    double getDistance(Point A, Point B) { return hypot(A.x - B.x, A.y - B.y); }
    double getDistance2(Point A, Point B) { return (A.x - B.x) * (A.x - B.x) + (A.y - B.y) * (A.y - B.y); }

    // 向量类
    typedef Point Vector;
    Vector operator+(Vector A, Vector B) { return Vector(A.x + B.x, A.y + B.y); }
    Vector operator-(Vector A, Vector B) { return Vector(A.x - B.x, A.y - B.y); }
    Vector operator*(Vector A, double k) { return Vector(A.x * k, A.y * k); }
    Vector operator/(Vector A, double k) { return Vector(A.x / k, A.y / k); }
    double operator*(Vector A, Vector B) { return A.x * B.x + A.y * B.y; }  // 点乘
    double operator^(Vector A, Vector B) { return A.x * B.y - A.y * B.x; }  // 叉乘
    double dotProduct(Vector A, Vector B) { return A.x * B.x + A.y * B.y; }
    double crossProduct(Vector A, Vector B) { return A.x * B.y - A.y * B.x; }
    double getLength(Vector A) { return hypot(A.x, A.y); }  // 模长
    double getAngle(Vector A, Vector B) { return acos(dotProduct(A, B) / getLength(A) / getLength(B)); }  // 夹角(弧度)
    double getArea(Point A, Point B, Point C) { return crossProduct(B - A, C - A); }  // 向量AB与向量AC张成的平行四边形的面积
    double getArea(Vector AB, Vector AC) { return crossProduct(AB, AC); }  // 向量AB与向量AC张成的平行四边形的面积
    Vector rotate(Vector A, double rad) { return Vector(A.x * cos(rad) + A.y * sin(rad), -A.x * sin(rad) + A.y * cos(rad)); }  // 向量A逆时针旋转rad得到的向量
    Vector normal(Vector A) { return Vector(-A.y, A.x); }  // 向量A逆时针旋转90度得到的向量
    double getProjection(Point A, Point B, Point C) { return (B - A) * (C - A) / getLength(B - A); }  // 求向量AC在向量BC上的投影
    Vector getUnitVector(Vector A) { return A / getLength(A); }  // 求向量A方向上的单位向量

    // 直线类
    struct Line {
      Point p;  // 直线上一点
        Vector v;  // 方向向量

        Line() {}
        Line(Point _p, Vector _v) :p(_p), v(_v) {}

        Point getPoint(double t) { return p + v * t; }  // 直线上参数为t的点
    };
    double getAngle(Line l) {  // 求直线倾斜角
        Point q = l.p + l.v;
        return atan2(q.y - l.p.y, q.x - l.p.x);
    }
    bool isPointOnLine(Point P, Point A, Point B) { return sgn(crossProduct(P - A, P - B)) == 0; }  // 点P是否在直线AB上
    bool isPointOnSegment(Point P, Point A, Point B) { return isPointOnLine(P, A, B) && sgn(dotProduct(A - P, B - P)) < 0; }  // 点P是否在线段AB上
    double getDistanceToLine(Point P, Point A, Point B) {  // 点P到直线AB的距离
        Vector v1 = B - A, v2 = P - A;
        return fabs(crossProduct(v1, v2) / getLength(v1));
    }
    double getDistanceToSegment(Point P, Point A, Point B) {  // 点P到线段AB的距离
        if (A == B) return getLength(P - A);
        Vector v1 = B - A, v2 = P - A, v3 = P - B;
```

```
        if (sgn(dotProduct(v1, v2)) < 0) return getLength(v2);  // P的投影在线段AB外,且更靠近A
        if (sgn(dotProduct(v1, v3)) < 0) return getLength(v3);  // P的投影在线段AB外,且更靠近B
        return getDistanceToLine(P, A, B);  // P的投影在线段AB上
    }
    Point getLineIntersection(Point P, Vector v, Point Q, Vector w) {  // 求直线(P+vt)与(Q+vw)的交点,使用前需保证有交点
        Vector u = P - Q;
        double t = crossProduct(w, u) / crossProduct(v, w);
        return P + v * t;
    }
    Point getLineIntersection(Line A, Line B) { return getLineIntersection(A.p, A.v, B.p, B.v); }  // 求直线A与B的交点,使用前需保证有交点
    Point getPointProjection(Point P, Point A, Point B) {  // 点P在直线AB上的投影
        Vector v1 = B - A, v2 = P - A;
        return A + v1 * (dotProduct(v1, v2) / dotProduct(v1, v1));
    }
    bool isSegmentIntersect(Point A1, Point A2, Point B1, Point B2) {  // 判断线段A1A2和B1B2是否相交
        double c1 = crossProduct(A2 - A1, B1 - A1), c2 = crossProduct(A2 - A1, B2 - A1),
            c3 = crossProduct(B2 - B1, A2 - B1), c4 = crossProduct(B2 - B1, A1 - B1);
        return sgn(c1) * sgn(c2) <= 0 && sgn(c3) * sgn(c4) <= 0;  // 若非规范相交不算相交,则去掉等号
    }
    Point getSegmentIntersection(Point P, Vector v, Point Q, Vector w) {  // 求两线段的交点,无交点返回Point(INF,INF)
        auto is_point_on_segment = [](Point P, Point A, Point B) {  // 特殊的判断点是否在线段上
            return sgn((P - A) * (P -B)) <= 0;
        };

        if (!sgn(v ^ w)) return { INF,INF };  // 两线段平行,无交点

        Vector u = P - Q;
        double t = (w ^ u) / (v ^ w);
        Point res = P + v * t;
        if (!is_point_on_segment(res, P, P + v) || !is_point_on_segment(res, Q, Q + w)) return { INF,INF };  // 交点不在线段上
        return res;
    }
}
using namespace Geometry_2D;

int n;
Point tri[MAXN][3];  // 三角形的三个顶点

double getLineArea(double a, int side) {  // 左边界side=1,右边界side=0
    vector<Point> tmp;
    for (int i = 0; i < n; i++) {
        auto t = tri[i];
        if (cmp(t[0].x, a) > 0 || cmp(t[2].x, a) < 0) continue;  // 无交集
        if (!cmp(t[0].x, a) && !cmp(t[1].x, a)) {  // 特判三角形的一条边在左边界上的情况
            if (side) tmp.push_back({ t[0].y,t[1].y });
        }
        else if (!cmp(t[2].x, a) && !cmp(t[1].x, a)) {  // 特判三角形的一条边在右边界上的情况
            if (!side) tmp.push_back({ t[2].y,t[1].y });
        }
```

```
        else {
            vector<double> ys;  // 存边界上的区间的y坐标
            for (int j = 0; j < 3; j++) {
                Point P = getSegmentIntersection(t[j], t[(j + 1) % 3] - t[j], { a,-INF }, { 0,INF * 2 });
                if (cmp(P.x, INF)) ys.push_back(P.y);
            }

            if(ys.size()) {
                sort(all(ys));
                tmp.push_back({ ys[0],ys.back() });
            }
        }
    }

    if (tmp.empty()) return 0;  // 无交点

    for (auto& [x, y] : tmp)  // 保证交点的x<y,方便排序:注意取引用
        if (x > y) swap(x, y);
    sort(all(tmp));
    double res = 0;
    double st = tmp[0].x, ed = tmp[0].y;  // 当前合并的区间的左右端点
    for (int i = 1; i < tmp.size(); i++) {
        if (tmp[i].x <= ed) ed = max(ed, tmp[i].y);
        else {
            res += ed - st;
            st = tmp[i].x, ed = tmp[i].y;
        }
    }
    res += ed - st;  // 最后一个区间
    return res;
}

double getRangeArea(double a, double b) {  // 求x∈[a,b]的梯形的面积并
    return (getLineArea(a, 1) + getLineArea(b, 0)) * (b - a) / 2;
}

void solve() {
    cin >> n;
    vector<double> xs;  // 存所有三角形的顶点和交点的x坐标,用于去重
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < 3; j++) {
            cin >> tri[i][j].x >> tri[i][j].y;
            xs.push_back(tri[i][j].x);
        }
        sort(tri[i], tri[i] + 3);  // 将三角形的顶点排序
    }

    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) {
            for (int x = 0; x < 3; x++) {
                for (int y = 0; y < 3; y++) {
                    auto P = getSegmentIntersection(tri[i][x], tri[i][(x + 1) % 3] - tri[i][x],
                                                    tri[j][y], tri[j][(y + 1) % 3] - tri[j][y]);
                    if (cmp(P.x, INF)) xs.push_back(P.x);  // 有交点才存下x坐标
                }
```

```cpp
            }
        }
    }

    sort(all(xs));
    double ans = 0;
    for (int i = 0; i + 1 < xs.size(); i++)
        if (cmp(xs[i], xs[i + 1])) ans += getRangeArea(xs[i], xs[i + 1]);
    cout << fixed << setprecision(2) << ans;
}

int main() {
    solve();
}
```

22.1.14 自适应Simpson积分

题意

给定两实数$a,b\ \ (1\leq a<b\leq 100)$,求积分$\displaystyle\int_a^b \frac{\sin x}{x}\mathrm{d}x$,结果保留六位小数.

思路

[二次函数积分公式,Simpson公式] 设二次函数$f(x)=ax^2+bx+c$,则$\displaystyle\int_l^r f(x)\mathrm{d}x=\frac{r-l}{6}\left[f(l)+f(r)+4f\left(\frac{l+r}{2}\right)\right]$.

普通Simpson法将积分区间细分,将每个小区间上的积分用二次函数的积分代替.为权衡精度和效率,自适应Simpson法在递归过程中根据精度调整递归次数,具体地,若一段图象与二次函数相似则直接求积分,不再往下递归.判断一段图象是否与二次函数相似的方法:先整段代入公式求积分res,再将该段平分后分别求积分$left$和$right$,若res与$(left+right)$充分接近,则认为该段图象与二次函数相似.

自适应Simpson法递归过程中除检查精度外还需强制执行最少的迭代次数.

代码

```cpp
double f(double x) {
    return sin(x) / x;
}

double simpson(double l, double r) {
    double mid = (l + r) / 2;
    return (r - l) * (f(l) + 4 * f(mid) + f(r)) / 6;
}

// eps为当前的精度,res为将函数在[l,r]上的图象视为二次函数时求得的答案,step为当前步数
double adaptive(double l, double r, double eps, double res, int step) {
    double mid = (l + r) / 2;
    double left = simpson(l, mid), right = simpson(mid, r);
    if (fabs(left + right - res) <= 15 * eps && step < 0) return left + right + (left + right - res) / 15;
    else return adaptive(l, mid, eps / 2, left, step - 1) + adaptive(mid, r, eps / 2, right, step - 1);
}
```

```

void solve() {
    double l, r; cin >> l >> r;
    cout << fixed << setprecision(6) << adaptive(l, r, eps, simpson(l, r), 12);
}

int main() {
    solve();
}
```

22.1.15 圆的面积并

题意 (5 s)

给定平面上的$n\ \ (1\leq n\leq 1000)$个圆,每个圆用三个整数$x,y,r\ \ (-1000\leq x,y\leq 1000m,0\leq r\leq 1000)$描述.求所有圆的面积并,误差不超过$1\mathrm{e}-2.$

思路

圆的面积并可用扫描线求,但中间扇形的面积并难计算.一般图形有弧线时用Simpson积分方便实现.

考虑用Simpson积分求.构造函数$f(x)\ \ s.t.\ \ f(x_0)$表示直线$x=x_0$被圆覆盖的长度,则$ans=\displaystyle\int_{-\infty}^{+\infty}f(x)\mathrm{d}x$,积分区间取$[-2000,2000]$即可.这样可以保证精度,因为圆的方程是二次的.

下面求一条直线被圆覆盖的长度.只需求该直线与所有圆交集的区间,做区间合并即可.

Simpson积分难处理离散的数据,如两圆分别为$\{(0,0),1\}$和$\{(2,0),1\}$时可能会求得面积并为0.对此,可先将所有圆投影到x轴上,做区间合并后对每个区间分别做Simpson积分.

本题卡时间,迭代时不要令$\varepsilon/=2$,也不要执行最小迭代次数.

代码

```
const int MAXN = 1005;
int n;
namespace Geometry_2D {
    // 点类
    struct Point {
        double x, y;

        Point(double _x = 0, double _y = 0) :x(_x), y(_y) {}

        bool operator==(const Point& B)const { return cmp(x, B.x) == 0 && cmp(y, B.y) == 0; }
        bool operator<(const Point& B)const { return x == B.x ? y < B.y : x < B.x; }  // 按x升序排列,x相同时按y升序排列
        friend ostream& operator<<(ostream& out, Point& P) {
            out << '[' << P.x << ',' << P.y <<']';
            return out;
        }
    }points[MAXN];
    double getDistance(Point A, Point B) { return hypot(A.x - B.x, A.y - B.y); }
    double getDistance2(Point A, Point B) { return (A.x - B.x) * (A.x - B.x) + (A.y - B.y) * (A.y - B.y); }

```

```cpp
    // 向量类
    typedef Point Vector;
    Vector operator+(Vector A, Vector B) { return Vector(A.x + B.x, A.y + B.y); }
    Vector operator-(Vector A, Vector B) { return Vector(A.x - B.x, A.y - B.y); }
    Vector operator*(Vector A, double k) { return Vector(A.x * k, A.y * k); }
    Vector operator/(Vector A, double k) { return Vector(A.x / k, A.y / k); }
    double operator*(Vector A, Vector B) { return A.x * B.x + A.y * B.y; }  // 点乘
    double operator^(Vector A, Vector B) { return A.x * B.y - A.y * B.x; }  // 叉乘
    double dotProduct(Vector A, Vector B) { return A.x * B.x + A.y * B.y; }
    double crossProduct(Vector A, Vector B) { return A.x * B.y - A.y * B.x; }
    double getLength(Vector A) { return hypot(A.x, A.y); }  // 模长
    double getAngle(Vector A, Vector B) { return acos(dotProduct(A, B) / getLength(A) / getLength(B)); }  // 夹角(弧度)
    double getArea(Point A, Point B, Point C) { return crossProduct(B - A, C - A); }  // 向量AB与向量AC张成的平行四边形的面积
    double getArea(Vector AB, Vector AC) { return crossProduct(AB, AC); }  // 向量AB与向量AC张成的平行四边形的面积
    Vector rotate(Vector A, double rad) { return Vector(A.x * cos(rad) + A.y * sin(rad), -A.x * sin(rad) + A.y * cos(rad)); }  // 向量A逆时针旋转rad得到的向量
    Vector normal(Vector A) { return Vector(-A.y, A.x); }  // 向量A逆时针旋转90度得到的向量
    double getProjection(Point A, Point B, Point C) { return (B - A) * (C - A) / getLength(B - A); }  // 求向量AC在向量BC上的投影
    Vector getUnitVector(Vector A) { return A / getLength(A); }  // 求向量A方向上的单位向量

    // 直线类
    struct Line {
      Point p;  // 直线上一点
        Vector v;  // 方向向量

        Line() {}
        Line(Point _p, Vector _v) :p(_p), v(_v) {}

        Point getPoint(double t) { return p + v * t; }  // 直线上参数为t的点
    };
    double getAngle(Line l) {  // 求直线倾斜角
        Point q = l.p + l.v;
        return atan2(q.y - l.p.y, q.x - l.p.x);
    }
    bool isPointOnLine(Point P, Point A, Point B) { return sgn(crossProduct(P - A, P - B)) == 0; }  // 点P是否在直线AB上
    bool isPointOnSegment(Point P, Point A, Point B) { return isPointOnLine(P, A, B) && sgn(dotProduct(A - P, B - P)) < 0; }  // 点P是否在线段AB上
    double getDistanceToLine(Point P, Point A, Point B) {  // 点P到直线AB的距离
        Vector v1 = B - A, v2 = P - A;
        return fabs(crossProduct(v1, v2) / getLength(v1));
    }
    double getDistanceToSegment(Point P, Point A, Point B) {  // 点P到线段AB的距离
        if (A == B) return getLength(P - A);
        Vector v1 = B - A, v2 = P - A, v3 = P - B;
        if (sgn(dotProduct(v1, v2)) < 0) return getLength(v2);  // P的投影在线段AB外,且更靠近A
        if (sgn(dotProduct(v1, v3)) < 0) return getLength(v3);  // P的投影在线段AB外,且更靠近B
        return getDistanceToLine(P, A, B);  // P的投影在线段AB上
    }
    Point getLineIntersection(Point P, Vector v, Point Q, Vector w) {  // 求直线(P+vt)与(Q+vw)的交点,使用前需保证有交点
        Vector u = P - Q;
```

```
        double t = crossProduct(w, u) / crossProduct(v, w);
        return P + v * t;
    }
    Point getLineIntersection(Line A, Line B) { return getLineIntersection(A.p, A.v, B.p, B.v); }  // 求直线A与B的交点,使用前需保证有交点
    Point getPointProjection(Point P, Point A, Point B) {  // 点P在直线AB上的投影
        Vector v1 = B - A, v2 = P - A;
        return A + v1 * (dotProduct(v1, v2) / dotProduct(v1, v1));
    }
    bool isSegmentIntersect(Point A1, Point A2, Point B1, Point B2) {  // 判断线段A1A2和B1B2是否相交
        double c1 = crossProduct(A2 - A1, B1 - A1), c2 = crossProduct(A2 - A1, B2 - A1),
            c3 = crossProduct(B2 - B1, A2 - B1), c4 = crossProduct(B2 - B1, A1 - B1);
        return sgn(c1) * sgn(c2) <= 0 && sgn(c3) * sgn(c4) <= 0;  // 若非规范相交不算相交,则去掉等号
    }

    // 圆类
    struct Circle {
        Point c;  // 圆心
        double r;  // 半径

        Circle(Point _c = { 0,0 }, double _r = 0) :c(_c), r(_r) {}
    }cirs[MAXN];
    bool isPointInCircle(Circle O, Point A) { return cmp(O.r, getDistance(O.c, A)) < 0; }  // 判断点A是否在圆O内部
    Line getPerpendicularBisector(Point A, Point B) { return Line((A + B) / 2, rotate(B - A, pi / 2)); }  // 求线段AB的中垂线
    Circle getCircle(Point A, Point B, Point C) {  // 求三角形ABC的外接圆
        auto l1 = getPerpendicularBisector(A, B), l2 = getPerpendicularBisector(A, C);
        auto O = getLineIntersection(l1.p, l1.v, l2.p, l2.v);
        return Circle(O, getDistance(O, A));
    }
    double getCircleLineIntersection(Circle cir, Point A, Point B, Point& PA, Point& PB) {  // 求直线AB与圆cir的交点PA和PB,返回圆心到线段AB的距离
        Point C = getLineIntersection(A, B - A, cir.c, rotate(B - A, pi / 2));  // 垂足
        double mindis = getDistance(cir.c, C);  // 圆心到线段AB的最短距离
        if (!isPointOnSegment(C, A, B)) mindis = min(getDistance(cir.c, A), getDistance(cir.c, B));

        if (cmp(cir.r, mindis) <= 0) return mindis;  // 线段AB与圆无交点

        double len = sqrt(cir.r * cir.r - getDistance(cir.c, C) * getDistance(cir.c, C));  // 半弦长
        PA = C + getUnitVector(A - B) * len, PB = C + getUnitVector(B - A) * len;
        return mindis;
    }
    double getSectorArea(Circle cir, Point A, Point B) {  // 求扇形OAB的有向面积,其中点A、B未必在圆上
        double theta = acos((A * B) / getLength(A) / getLength(B));  // 圆心角
        if (sgn(A ^ B) < 0) theta = -theta;  // 顺时针为负方向,有向面积为负数
        return cir.r * cir.r * theta / 2;
    }
    double getCircleTriangleArea(Circle cir, Point A, Point B) {  // 求三角形OAB与圆的交集的有向面积,其中点A、B未必在圆上
        double disa = getDistance(cir.c, A), disb = getDistance(cir.c, B);
        if (cmp(cir.r, disa) >= 0 && cmp(cir.r, disb) >= 0) return (A ^ B) / 2;  // 点A、B都在圆内或圆上
```

```
        if (!sgn(A ^ B)) return 0;  // O、A、B三点共线

        Point PA, PB;  // 直线AB与圆的交点
        double mindis = getCircleLineIntersection(cir, A, B, PA, PB);
        if (cmp(cir.r, mindis) <= 0) return getSectorArea(cir, A, B);  // 点A、B都在圆外且线段AB与圆无交点
        if (cmp(cir.r, disa) >= 0) return (A ^ PB) / 2 + getSectorArea(cir, PB, B);  // 点A在圆内,点B在圆外
        if (cmp(cir.r, disb) >= 0) return (PA ^ B) / 2 + getSectorArea(cir, A, PA);  // 点A在圆外,点B在圆内
        return (PA ^ PB) / 2 + getSectorArea(cir, A, PA) + getSectorArea(cir, PB, B);  // 点A、B都在圆外且线段AB与圆无交点
    }
}
using namespace Geometry_2D;

namespace Simpson {
    double f(double x) {  // 直线x=x0被圆覆盖的长度
        vector<pdd> segs;
        for (int i = 0; i < n; i++) {
            double dx = fabs(x - cirs[i].c.x), r = cirs[i].r;
            if (cmp(dx, r) < 0) {
                double dy = sqrt(r * r - dx * dx);
                segs.push_back({ cirs[i].c.y - dy,cirs[i].c.y + dy});
            }
        }

        if (segs.empty()) return 0;  // 无交点

        sort(all(segs));
        double res = 0;
        double st = segs[0].first, ed = segs[0].second;
        for (int i = 1; i < segs.size(); i++) {
            if (segs[i].first <= ed) ed = max(ed, segs[i].second);
            else {
                res += ed - st;
                st = segs[i].first, ed = segs[i].second;
            }
        }
        return res + ed - st;
    }

    double simpson(double l, double r) {
        double mid = (l + r) / 2;
        return (r - l) * (f(l) + 4 * f(mid) + f(r)) / 6;
    }

    // eps为当前的精度,res为将函数在[l,r]上的图象视为二次函数时求得的答案,step为当前步数
    double adaptive(double l, double r, double eps, double res, int step) {
        double mid = (l + r) / 2;
        double left = simpson(l, mid), right = simpson(mid, r);
        if (fabs(left + right - res) <= 15 * eps && step < 0) return left + right + (left + right - res) / 15;
        else return adaptive(l, mid, eps, left, step - 1) + adaptive(mid, r, eps, right, step - 1);  // 此处不要eps/2
    }
}
using namespace Simpson;
```

```cpp

void merge(vector<pdd>& segs) {  // 区间合并
    sort(all(segs));

    vector<pdd> res;
    double l = -INF, r = -INF;
    for (auto& seg : segs) {
        if (cmp(seg.first, r) <= 0) r = max(r, seg.second);
        else {
            if (cmp(l, -INF)) res.push_back({ l,r });
            l = seg.first, r = seg.second;
        }
    }
    res.push_back({ l,r });  // 最后一个区间
    segs = res;
}

void solve() {
    cin >> n;
    vector<pdd> segs;  // 圆投影到x轴上的区间
    for (int i = 0; i < n; i++) {
        cin >> cirs[i].c.x >> cirs[i].c.y >> cirs[i].r;
        segs.push_back({ cirs[i].c.x - cirs[i].r,cirs[i].c.x + cirs[i].r });
    }

    merge(segs);

    double ans = 0;
    for (auto& seg : segs)
        ans += adaptive(seg.first, seg.second, eps, simpson(seg.first, seg.second), 1); // 此处只迭代1次
    cout << fixed << setprecision(3) << ans;
}

int main() {
    solve();
}
```

22.2 三维计算几何

[叉积] 向量$\overrightarrow{\alpha}=(x_1,y_1,z_1),\overrightarrow{\beta}=(x_2,y_2,z_2)$,则$\overrightarrow{\alpha}\times\overrightarrow{\beta}=\left(\begin{vmatrix}y_1 & z_1\\ y_2 & z_2\end{vmatrix},-\begin{vmatrix}x_1 & z_1\\ x_2 & z_2\end{vmatrix},\begin{vmatrix}x_1 & z_1\\ x_2 & z_2\end{vmatrix}\right)$.

[求平面法向量] 取平面上不共线的两向量,求它们的叉积.

[判断点是否在平面上] 取平面上两不共线的向量$\overrightarrow{\alpha},\overrightarrow{\beta}$,求法向量$\overrightarrow{\gamma}=\overrightarrow{\alpha}\times\overrightarrow{\beta}$.判断点$P$是否在平面上时,取平面上一点$Q$,判断$\overrightarrow{PQ}$在$\overrightarrow{\gamma}$上的投影的长度是否为0,即$\overrightarrow{PQ}\cdot\overrightarrow{\gamma}$是否为0.显然根据点积的正负还可判断点在平面的上方还是下方.

[求点到平面的距离] 取平面上两不共线的向量$\overrightarrow{\alpha},\overrightarrow{\beta}$,求法向量$\overrightarrow{\gamma}=\overrightarrow{\alpha}\times\overrightarrow{\beta}$.求点$P$到平面的距离时,取平面上一点$Q$,求$\overrightarrow{PQ}$在$\overrightarrow{\gamma}$上的投影的长度即可,即$\dfrac{\overrightarrow{PQ}\cdot\overrightarrow{\gamma}}{\left|\overrightarrow{\gamma}\right|}$.

[多面体Euler定理] 顶点数-棱长数+面数=2.

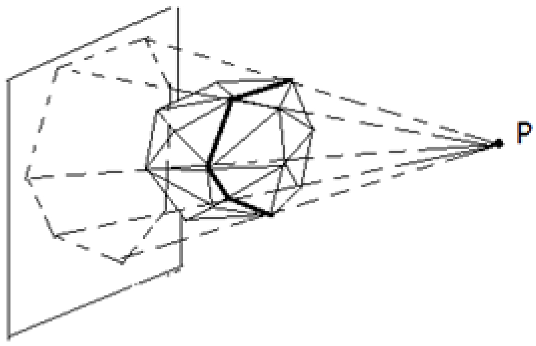
22.2.1 最佳包裹

题意

给定空间中相异的$n\ \ (1\leq n\leq 2000)$个点(x_i,y_i,z_i),求凸包的面积,四舍五入保留小数点后六位.

思路

将凸包的所有面存下来,因所有面都是多边形,则都可以三角形的形式存下来.为避免出现四点共面的情况,可对所有点微扰.注意微扰后不能再通过$<\varepsilon$来判断是否共面.



现新加入一个点P,若该点在凸包内部,则跳过;否则考虑点P向四周发光,删除凸包上能被光线照到的面和与光线平行的面,其余面保留.显然删除的面与保留的面存在分界线(如上图加粗的线),将分割线上的顶点与点P构成的面加入凸包中.

判断一个平面能否被光源照到即判断光源是否在平面上方.

是分界线的边两侧的平面一个能被照到,一个不能被照到,可用有个bool数组$see[\ ][\ ]$维护,其中$see[i][j]=true$表示点i指向点j的向量代表的面能被照到.逆时针存三角形的三条边的向量,则边$\overline{ij}$的两侧的平面对应的向量即$\overrightarrow{ij}$和$\overrightarrow{ji}$.

总时间复杂度为$O(n^2)$.

[证]

[面数与点数的关系] 每个面(三角形)有三条边,每条边会被两个面数到,则$3\times$ 面数 $=2\times$ 边数,即面数 $=\dfrac{2}{3}$边数.

设凸包顶点数为n,由多面体Euler公式知:至多有$(3n-6)$条边,$(2n-4)$个面.

每加入一个点都要判断$O(n)$个面的去留,故总时间复杂度$O(n^2)$.

本题ε取$1\mathrm{e}-10$.

代码

```
const double eps = 1e-10;
const int MAXN = 2005 << 1;  // 两倍空间
int n, m;  // 顶点数、面数
namespace Geometry_3D {
    double rand_eps() {  // 生成微扰
        return ((double)rand() / RAND_MAX - 0.5) * eps;
    }

    // 点类
    struct Point {
        double x, y, z;

```

```cpp
        Point(double _x = 0, double _y = 0, double _z = 0) :x(_x), y(_y), z(_z) {}

        bool operator==(const Point& B)const { return cmp(x, B.x) == 0 && cmp(y, B.y) == 0 && cmp(z, B.z); }
        friend ostream& operator<<(ostream& out, const Point p) {
            out << '[' << p.x << ',' << p.y << ',' << p.z << ']';
            return out;
        }

        void shake() { x += rand_eps(), y += rand_eps(), z += rand_eps(); }  // 微扰
    }points[MAXN];
    double getDistance(Point A, Point B) { return hypot(hypot(A.x - B.x, A.y - B.y), A.z - B.z); }

    // 向量类
    typedef Point Vector;
    Vector operator+(Vector A, Vector B) { return Vector(A.x + B.x, A.y + B.y, A.z + B.z); }
    Vector operator-(Vector A, Vector B) { return Vector(A.x - B.x, A.y - B.y, A.z - B.z); }
    Vector operator*(Vector A, double k) { return Vector(A.x * k, A.y * k, A.z * k); }
    Vector operator/(Vector A, double k) { return Vector(A.x / k, A.y / k, A.z / k); }
    double operator*(Vector A, Vector B) { return A.x * B.x + A.y * B.y + A.z * B.z; }  // 点乘
    Vector operator^(Vector A, Vector B) { return { A.y * B.z - B.y * A.z,A.z * B.x - B.z * A.x,A.x * B.y - B.x * A.y }; }  // 叉乘
    double dotProduct(Vector A, Vector B) { return A.x * B.x + A.y * B.y + A.z * B.z; }
    Vector crossProduct(Vector A, Vector B) { return { A.y * B.z - B.y * A.z,A.z * B.x - B.z * A.x,A.x * B.y - B.x * A.y }; }
    double getLength(Vector A) { return hypot(hypot(A.x, A.y), A.z); }  // 模长

    // 平面类
    struct Plane {
        int v[3];  // 逆时针三角形三个顶点的编号

        Vector getNormalVector() { return (points[v[1]] - points[v[0]]) ^ (points[v[2]] - points[v[0]]); }  // 求平面法向量
        double getArea() { return getLength(getNormalVector()) / 2; }  // 求三角形面积
        bool isPointAbove(Point P) { return (P - points[v[0]]) * getNormalVector() >= 0; }  // 判断点P是否在平面上方
    }planes[MAXN], tmpplanes[MAXN];

    bool see[MAXN][MAXN];  // s[i][j]表示向量ij代表的平面能否被照到

    double getConvexHull() {  // 求凸包,返回凸包面积
        for (int i = 0; i < n; i++) points[i].shake();  // 微扰

        planes[m++] = { 0,1,2 }, planes[m++] = { 2,1,0 };  // 三角形的正面和反面的顶点编号

        for (int i = 3; i < n; i++) {  // 枚举顶点
            int cnt = 0;
            for (int j = 0; j < m; j++) {  // 枚举现有的面
                bool flag = planes[j].isPointAbove(points[i]);  // 判断该面能否被照到
                if (!flag) tmpplanes[cnt++] = planes[j];  // 保留不能被照到的面

                for (int k = 0; k < 3; k ++)  // 记录三角形三边对应的平面能否被照到
                    see[planes[j].v[k]][planes[j].v[(k + 1) % 3]] = flag;
            }
```

```cpp

            for (int j = 0; j < m; j++) {  // 枚举每个面
                for (int k = 0; k < 3; k++) {  // 枚举三角形的三条边
                    int a = planes[j].v[k], b = planes[j].v[(k + 1) % 3];
                    if (see[a][b] && !see[b][a])  // 是分界线
                        tmpplanes[cnt++] = { a,b,i };  // 加入光源与分界线构成的平面
                }
            }

            // 记录答案
            m = cnt;
            for (int j = 0; j < m; j++) planes[j] = tmpplanes[j];
        }

        double res = 0;
        for (int i = 0; i < m; i++) res += planes[i].getArea();
        return res;
    }
};
using namespace Geometry_3D;

void solve() {
    cin >> n;
    for (int i = 0; i < n; i++) cin >> points[i].x >> points[i].y >> points[i].z;

    cout << fixed << setprecision(6) << getConvexHull();
}

int main() {
    solve();
}
```