



Standard Code Library

Part4 - Geometry

Jiangxi Normal University
HeartFireY, eroengine, yezzz

November 7, 2022

Standard Code Library

HeartFireY, eroengine, yezzz

Jiangxi Normal University

November 7, 2022

Contents

Section.4 计算几何	2
二维几何: 点与向量	2
象限	2
线	3
点与线	3
线与线	3
多边形	4
面积、凸包	4
旋转卡壳	4
半平面交	5
圆	5
三点求圆心	6
圆线交点、圆圆交点	6
圆圆位置关系	6
圆与多边形交	6
圆的离散化、面积并	7
最小圆覆盖	9
圆的反演	10
三维计算几何	10
旋转	10
线、面	11
凸包	11

Section.4 计算几何

二维几何：点与向量

```
1  #define y1 yy1
2  #define nxt(i) ((i + 1) % s.size())
3  typedef double LD;
4  const LD PI = 3.14159265358979323846;
5  const LD eps = 1E-10;
6  int sgn(LD x) { return fabs(x) < eps ? 0 : (x > 0 ? 1 : -1); }
7  struct L;
8  struct P;
9  typedef P V;
10 struct P {
11     LD x, y;
12     explicit P(LD x = 0, LD y = 0): x(x), y(y) {}
13     explicit P(const L& l);
14 };
15 struct L {
16     P s, t;
17     L() {}
18     L(P s, P t): s(s), t(t) {}
19 };
20
21 P operator + (const P& a, const P& b) { return P(a.x + b.x, a.y + b.y); }
22 P operator - (const P& a, const P& b) { return P(a.x - b.x, a.y - b.y); }
23 P operator * (const P& a, LD k) { return P(a.x * k, a.y * k); }
24 P operator / (const P& a, LD k) { return P(a.x / k, a.y / k); }
25 inline bool operator < (const P& a, const P& b) {
26     return sgn(a.x - b.x) < 0 || (sgn(a.x - b.x) == 0 && sgn(a.y - b.y) < 0);
27 }
28 bool operator == (const P& a, const P& b) { return !sgn(a.x - b.x) && !sgn(a.y - b.y); }
29 P::P(const L& l) { *this = l.t - l.s; }
30 ostream &operator << (ostream &os, const P &p) {
31     return (os << "(" << p.x << ", " << p.y << ")");
32 }
33 istream &operator >> (istream &is, P &p) {
34     return (is >> p.x >> p.y);
35 }
36
37 LD dist(const P& p) { return sqrt(p.x * p.x + p.y * p.y); }
38 LD dot(const V& a, const V& b) { return a.x * b.x + a.y * b.y; }
39 LD det(const V& a, const V& b) { return a.x * b.y - a.y * b.x; }
40 LD cross(const P& s, const P& t, const P& o = P()) { return det(s - o, t - o); }
41 // -----
```

象限

```
1  // 象限
2  int quad(P p) {
3      int x = sgn(p.x), y = sgn(p.y);
4      if (x > 0 && y >= 0) return 1;
5      if (x <= 0 && y > 0) return 2;
6      if (x < 0 && y <= 0) return 3;
7      if (x >= 0 && y < 0) return 4;
8      assert(0);
9  }
10
11 // 仅适用于参照点在所有点一侧的情况
12 struct cmp_angle {
13     P p;
14     bool operator () (const P& a, const P& b) {
15         // int qa = quad(a - p), qb = quad(b - p);
16         // if (qa != qb) return qa < qb;
17         int d = sgn(cross(a, b, p));
18         if (d) return d > 0;
19         return dist(a - p) < dist(b - p);
20     }
21 };
```

线

```
1 // 是否平行
2 bool parallel(const L& a, const L& b) {
3     return !sgn(det(P(a), P(b)));
4 }
5 // 直线是否相等
6 bool l_eq(const L& a, const L& b) {
7     return parallel(a, b) && parallel(L(a.s, b.t), L(b.s, a.t));
8 }
9 // 逆时针旋转 r 弧度
10 P rotate(const P& p, const LD& r) { return P(p.x * cos(r) - p.y * sin(r), p.x * sin(r) + p.y * cos(r)); }
11 P RotateCCW90(const P& p) { return P(-p.y, p.x); }
12 P RotateCW90(const P& p) { return P(p.y, -p.x); }
13 // 单位法向量
14 V normal(const V& v) { return V(-v.y, v.x) / dist(v); }
```

点与线

```
1 // 点在线段上 <= 0 包含端点 < 0 则不包含
2 bool p_on_seg(const P& p, const L& seg) {
3     P a = seg.s, b = seg.t;
4     return !sgn(det(p - a, b - a)) && sgn(dot(p - a, p - b)) <= 0;
5 }
6 // 点到直线距离
7 LD dist_to_line(const P& p, const L& l) {
8     return fabs(cross(l.s, l.t, p)) / dist(l);
9 }
10 // 点到线段距离
11 LD dist_to_seg(const P& p, const L& l) {
12     if (l.s == l.t) return dist(p - l);
13     V vs = p - l.s, vt = p - l.t;
14     if (sgn(dot(l, vs)) < 0) return dist(vs);
15     else if (sgn(dot(l, vt)) > 0) return dist(vt);
16     else return dist_to_line(p, l);
17 }
```

线与线

```
1 // 求直线交 需要事先保证有界
2 P l_intersection(const L& a, const L& b) {
3     LD s1 = det(P(a), b.s - a.s), s2 = det(P(a), b.t - a.s);
4     return (b.s * s2 - b.t * s1) / (s2 - s1);
5 }
6 // 向量夹角的弧度
7 LD angle(const V& a, const V& b) {
8     LD r = asin(fabs(det(a, b)) / dist(a) / dist(b));
9     if (sgn(dot(a, b)) < 0) r = PI - r;
10    return r;
11 }
12 // 线段和直线是否有交 1 = 规范, 2 = 不规范
13 int s_l_cross(const L& seg, const L& line) {
14     int d1 = sgn(cross(line.s, line.t, seg.s));
15     int d2 = sgn(cross(line.s, line.t, seg.t));
16     if ((d1 ^ d2) == -2) return 1; // proper
17     if (d1 == 0 || d2 == 0) return 2;
18     return 0;
19 }
20 // 线段的交 1 = 规范, 2 = 不规范
21 int s_cross(const L& a, const L& b, P& p) {
22     int d1 = sgn(cross(a.t, b.s, a.s)), d2 = sgn(cross(a.t, b.t, a.s));
23     int d3 = sgn(cross(b.t, a.s, b.s)), d4 = sgn(cross(b.t, a.t, b.s));
24     if ((d1 ^ d2) == -2 && (d3 ^ d4) == -2) { p = l_intersection(a, b); return 1; }
25     if (!d1 && p_on_seg(b.s, a)) { p = b.s; return 2; }
26     if (!d2 && p_on_seg(b.t, a)) { p = b.t; return 2; }
27     if (!d3 && p_on_seg(a.s, b)) { p = a.s; return 2; }
28     if (!d4 && p_on_seg(a.t, b)) { p = a.t; return 2; }
29     return 0;
30 }
```

多边形

面积、凸包

```
1  typedef vector<P> S;
2
3  // 点是否在多边形中 0 = 在外部 1 = 在内部 -1 = 在边界上
4  int inside(const S& s, const P& p) {
5      int cnt = 0;
6      FOR (i, 0, s.size()) {
7          P a = s[i], b = s[nxt(i)];
8          if (p_on_seg(p, L(a, b))) return -1;
9          if (sgn(a.y - b.y) <= 0) swap(a, b);
10         if (sgn(p.y - a.y) > 0) continue;
11         if (sgn(p.y - b.y) <= 0) continue;
12         cnt += sgn(cross(b, a, p)) > 0;
13     }
14     return bool(cnt & 1);
15 }
16 // 多边形面积, 有向面积可能为负
17 LD polygon_area(const S& s) {
18     LD ret = 0;
19     FOR (i, 1, (LL)s.size() - 1)
20         ret += cross(s[i], s[i + 1], s[0]);
21     return ret / 2;
22 }
23 // 构建凸包 点不可以重复 < 0 边上可以有点, <= 0 则不能
24 // 会改变输入点的顺序
25 const int MAX_N = 1000;
26 S convex_hull(S& s) {
27     // assert(s.size() >= 3);
28     sort(s.begin(), s.end());
29     S ret(MAX_N * 2);
30     int sz = 0;
31     FOR (i, 0, s.size()) {
32         while (sz > 1 && sgn(cross(ret[sz - 1], s[i], ret[sz - 2])) < 0) --sz;
33         ret[sz++] = s[i];
34     }
35     int k = sz;
36     FOR (i, (LL)s.size() - 2, -1) {
37         while (sz > k && sgn(cross(ret[sz - 1], s[i], ret[sz - 2])) < 0) --sz;
38         ret[sz++] = s[i];
39     }
40     ret.resize(sz - (s.size() > 1));
41     return ret;
42 }
43
44 P ComputeCentroid(const vector<P> &p) {
45     P c(0, 0);
46     LD scale = 6.0 * polygon_area(p);
47     for (unsigned i = 0; i < p.size(); i++) {
48         unsigned j = (i + 1) % p.size();
49         c = c + (p[i] + p[j]) * (p[i].x * p[j].y - p[j].x * p[i].y);
50     }
51     return c / scale;
52 }
```

旋转卡壳

```
1  LD rotatingCalipers(vector<P>& qs) {
2      int n = qs.size();
3      if (n == 2)
4          return dist(qs[0] - qs[1]);
5      int i = 0, j = 0;
6      FOR (k, 0, n) {
7          if (!(qs[i] < qs[k])) i = k;
8          if (qs[j] < qs[k]) j = k;
9      }
10     LD res = 0;
11     int si = i, sj = j;
12     while (i != sj || j != si) {
```

```

13     res = max(res, dist(qs[i] - qs[j]));
14     if (sgn(cross(qs[(i+1)%n] - qs[i], qs[(j+1)%n] - qs[j])) < 0)
15         i = (i + 1) % n;
16     else j = (j + 1) % n;
17 }
18 return res;
19 }
20
21 int main() {
22     int n;
23     while (cin >> n) {
24         S v(n);
25         FOR (i, 0, n) cin >> v[i].x >> v[i].y;
26         convex_hull(v);
27         printf("%.0f\n", rotatingCalipers(v));
28     }
29 }

```

半平面交

```

1 struct LV {
2     P p, v; LD ang;
3     LV() {}
4     LV(P s, P t): p(s), v(t - s) { ang = atan2(v.y, v.x); }
5 }; // 另一种向量表示
6
7 bool operator < (const LV &a, const LV& b) { return a.ang < b.ang; }
8 bool on_left(const LV& l, const P& p) { return sgn(cross(l.v, p - l.p)) >= 0; }
9 P l_intersection(const LV& a, const LV& b) {
10     P u = a.p - b.p; LD t = cross(b.v, u) / cross(a.v, b.v);
11     return a.p + a.v * t;
12 }
13
14 S half_plane_intersection(vector<LV>& L) {
15     int n = L.size(), fi, la;
16     sort(L.begin(), L.end());
17     vector<P> p(n); vector<LV> q(n);
18     q[fi = la = 0] = L[0];
19     FOR (i, 1, n) {
20         while (fi < la && !on_left(L[i], p[la - 1])) la--;
21         while (fi < la && !on_left(L[i], p[fi])) fi++;
22         q[++la] = L[i];
23         if (sgn(cross(q[la].v, q[la - 1].v)) == 0) {
24             la--;
25             if (on_left(q[la], L[i].p)) q[la] = L[i];
26         }
27         if (fi < la) p[la - 1] = l_intersection(q[la - 1], q[la]);
28     }
29     while (fi < la && !on_left(q[fi], p[la - 1])) la--;
30     if (la - fi <= 1) return vector<P>();
31     p[la] = l_intersection(q[la], q[fi]);
32     return vector<P>(p.begin() + fi, p.begin() + la + 1);
33 }
34
35 S convex_intersection(const vector<P> &v1, const vector<P> &v2) {
36     vector<LV> h; int n = v1.size(), m = v2.size();
37     FOR (i, 0, n) h.push_back(LV(v1[i], v1[(i + 1) % n]));
38     FOR (i, 0, m) h.push_back(LV(v2[i], v2[(i + 1) % m]));
39     return half_plane_intersection(h);
40 }

```

圓

```

1 struct C {
2     P p; LD r;
3     C(LD x = 0, LD y = 0, LD r = 0): p(x, y), r(r) {}
4     C(P p, LD r): p(p), r(r) {}
5 };

```

三点求圆心

```
1 P compute_circle_center(P a, P b, P c) {
2     b = (a + b) / 2;
3     c = (a + c) / 2;
4     return l_intersection({b, b + RotateCW90(a - b)}, {c, c + RotateCW90(a - c)});
5 }
```

圆线交点、圆圆交点

- 圆和线的交点关于圆心是顺时针的

```
1 vector<P> c_l_intersection(const L& l, const C& c) {
2     vector<P> ret;
3     P b(l), a = l.s - c.p;
4     LD x = dot(b, b), y = dot(a, b), z = dot(a, a) - c.r * c.r;
5     LD D = y * y - x * z;
6     if (sgn(D) < 0) return ret;
7     ret.push_back(c.p + a + b * (-y + sqrt(D + eps)) / x);
8     if (sgn(D) > 0) ret.push_back(c.p + a + b * (-y - sqrt(D)) / x);
9     return ret;
10 }
11
12 vector<P> c_c_intersection(C a, C b) {
13     vector<P> ret;
14     LD d = dist(a.p - b.p);
15     if (sgn(d) == 0 || sgn(d - (a.r + b.r)) > 0 || sgn(d + min(a.r, b.r) - max(a.r, b.r)) < 0)
16         return ret;
17     LD x = (d * d - b.r * b.r + a.r * a.r) / (2 * d);
18     LD y = sqrt(a.r * a.r - x * x);
19     P v = (b.p - a.p) / d;
20     ret.push_back(a.p + v * x + RotateCCW90(v) * y);
21     if (sgn(y) > 0) ret.push_back(a.p + v * x - RotateCCW90(v) * y);
22     return ret;
23 }
```

圆圆位置关系

```
1 // 1: 内含 2: 内切 3: 相交 4: 外切 5: 相离
2 int c_c_relation(const C& a, const C& v) {
3     LD d = dist(a.p - v.p);
4     if (sgn(d - a.r - v.r) > 0) return 5;
5     if (sgn(d - a.r - v.r) == 0) return 4;
6     LD l = fabs(a.r - v.r);
7     if (sgn(d - l) > 0) return 3;
8     if (sgn(d - l) == 0) return 2;
9     if (sgn(d - l) < 0) return 1;
10 }
```

圆与多边形交

- HDU 5130
- 注意顺时针逆时针（可能要取绝对值）

```
1 LD sector_area(const P& a, const P& b, LD r) {
2     LD th = atan2(a.y, a.x) - atan2(b.y, b.x);
3     while (th <= 0) th += 2 * PI;
4     while (th > 2 * PI) th -= 2 * PI;
5     th = min(th, 2 * PI - th);
6     return r * r * th / 2;
7 }
8
9 LD c_tri_area(P a, P b, P center, LD r) {
10     a = a - center; b = b - center;
11     int ina = sgn(dist(a) - r) < 0, inb = sgn(dist(b) - r) < 0;
12     // dbg(a, b, ina, inb);
13     if (ina && inb) {
14         return fabs(cross(a, b)) / 2;
15     } else {
16         auto p = c_l_intersection(L(a, b), C(0, 0, r));
```



```

17     if (ina ^ inb) {
18         auto cr = p_on_seg(p[0], L(a, b)) ? p[0] : p[1];
19         if (ina) return sector_area(b, cr, r) + fabs(cross(a, cr)) / 2;
20         else return sector_area(a, cr, r) + fabs(cross(b, cr)) / 2;
21     } else {
22         if ((int) p.size() == 2 && p_on_seg(p[0], L(a, b))) {
23             if (dist(p[0] - a) > dist(p[1] - a)) swap(p[0], p[1]);
24             return sector_area(a, p[0], r) + sector_area(p[1], b, r)
25                 + fabs(cross(p[0], p[1])) / 2;
26         } else return sector_area(a, b, r);
27     }
28 }
29 }
30
31 typedef vector<P> S;
32 LD c_poly_area(S poly, const C& c) {
33     LD ret = 0; int n = poly.size();
34     FOR (i, 0, n) {
35         int t = sgn(cross(poly[i] - c.p, poly[(i + 1) % n] - c.p));
36         if (t) ret += t * c_tri_area(poly[i], poly[(i + 1) % n], c.p, c.r);
37     }
38     return ret;
39 }

```

圖的离散化、面积并

SPOJ: CIRU, EOJ: 284

- 版本 1: 复杂度 $O(n^3 \log n)$ 。虽然常数小，但还是难以接受。
- 优点？想不出来。
- 原理上是用竖线进行切分，然后对每一个切片分别计算。
- 扫描线部分可以魔改，求各种东西。

```

1  inline LD rt(LD x) { return sgn(x) == 0 ? 0 : sqrt(x); }
2  inline LD sq(LD x) { return x * x; }
3
4  // 圆弧
5  // 如果按照 x 离散化，圆弧是 " 横着的 "
6  // 记录圆弧的左端点、右端点、中点的坐标，和圆弧所在的圆
7  // 调用构造要保证 c.x - x.r <= xl < xr <= c.y + x.r
8  // t = 1 下圆弧 t = -1 上圆弧
9  struct CV {
10     LD yl, yr, ym; C o; int type;
11     CV() {}
12     CV(LD yl, LD yr, LD ym, C c, int t)
13         : yl(yl), yr(yr), ym(ym), type(t), o(c) {}
14 };
15
16 // 辅助函数 求圆上纵坐标
17 pair<LD, LD> c_point_eval(const C& c, LD x) {
18     LD d = fabs(c.p.x - x), h = rt(sq(c.r) - sq(d));
19     return {c.p.y - h, c.p.y + h};
20 }
21 // 构造上下圆弧
22 pair<CV, CV> pairwise_curves(const C& c, LD xl, LD xr) {
23     LD yl1, yl2, yr1, yr2, ym1, ym2;
24     tie(yl1, yl2) = c_point_eval(c, xl);
25     tie(ym1, ym2) = c_point_eval(c, (xl + xr) / 2);
26     tie(yr1, yr2) = c_point_eval(c, xr);
27     return {CV(yl1, yr1, ym1, c, 1), CV(yl2, yr2, ym2, c, -1)};
28 }
29
30 // 离散化之后同一切片内的圆弧应该是不相交的
31 bool operator < (const CV& a, const CV& b) { return a.ym < b.ym; }
32 // 计算圆弧和连接圆弧端点的线段构成的封闭图形的面积
33 LD cv_area(const CV& v, LD xl, LD xr) {
34     LD l = rt(sq(xr - xl) + sq(v.yr - v.yl));
35     LD d = rt(sq(v.o.r) - sq(l / 2));
36     LD ang = atan(l / d / 2);
37     return ang * sq(v.o.r) - d * l / 2;

```

```

38 }
39
40 LD circle_union(const vector<C>& cs) {
41     int n = cs.size();
42     vector<LD> xs;
43     FOR (i, 0, n) {
44         xs.push_back(cs[i].p.x - cs[i].r);
45         xs.push_back(cs[i].p.x);
46         xs.push_back(cs[i].p.x + cs[i].r);
47         FOR (j, i + 1, n) {
48             auto pts = c_c_intersection(cs[i], cs[j]);
49             for (auto& p: pts) xs.push_back(p.x);
50         }
51     }
52     sort(xs.begin(), xs.end());
53     xs.erase(unique(xs.begin(), xs.end(), [](LD x, LD y) { return sgn(x - y) == 0; }), xs.end());
54     LD ans = 0;
55     FOR (i, 0, (int) xs.size() - 1) {
56         LD xl = xs[i], xr = xs[i + 1];
57         vector<CV> intv;
58         FOR (k, 0, n) {
59             auto& c = cs[k];
60             if (sgn(c.p.x - c.r - xl) <= 0 && sgn(c.p.x + c.r - xr) >= 0) {
61                 auto t = pairwise_curves(c, xl, xr);
62                 intv.push_back(t.first); intv.push_back(t.second);
63             }
64         }
65         sort(intv.begin(), intv.end());
66
67         vector<LD> areas(intv.size());
68         FOR (i, 0, intv.size()) areas[i] = cv_area(intv[i], xl, xr);
69
70         int cc = 0;
71         FOR (i, 0, intv.size()) {
72             if (cc > 0) {
73                 ans += (intv[i].yl - intv[i - 1].yl + intv[i].yr - intv[i - 1].yr) * (xr - xl) / 2;
74                 ans += intv[i - 1].type * areas[i - 1];
75                 ans -= intv[i].type * areas[i];
76             }
77             cc += intv[i].type;
78         }
79     }
80     return ans;
81 }

```

- 版本 2: 复杂度 $O(n^2 \log n)$ 。
- 原理是: 认为所求部分是一个奇怪的多边形 + 若干弓形。然后对于每个圆分别求贡献的弓形, 并累加多边形有向面积。
- 同样可以魔改扫描线的部分, 用于求周长、至少覆盖 k 次等等。
- 内含、内切、同一个圆的情况, 通常需要特殊处理。
- 下面的代码是 k 圆覆盖。

```

1 inline LD angle(const P& p) { return atan2(p.y, p.x); }
2
3 // 圆弧上的点
4 // p 是相对于圆心的坐标
5 // a 是在圆上的 atan2 [-PI, PI]
6 struct CP {
7     P p; LD a; int t;
8     CP() {}
9     CP(P p, LD a, int t): p(p), a(a), t(t) {}
10 };
11 bool operator < (const CP& u, const CP& v) { return u.a < v.a; }
12 LD cv_area(LD r, const CP& q1, const CP& q2) {
13     return (r * r * (q2.a - q1.a) - cross(q1.p, q2.p)) / 2;
14 }
15
16 LD ans[N];
17 void circle_union(const vector<C>& cs) {
18     int n = cs.size();
19     FOR (i, 0, n) {
20         // 有相同的圆的话只考虑第一次出现

```

```

21     bool ok = true;
22     FOR (j, 0, i)
23         if (sgn(cs[i].r - cs[j].r) == 0 && cs[i].p == cs[j].p) {
24             ok = false;
25             break;
26         }
27     if (!ok) continue;
28     auto& c = cs[i];
29     vector<CP> ev;
30     int belong_to = 0;
31     P bound = c.p + P(-c.r, 0);
32     ev.emplace_back(bound, -PI, 0);
33     ev.emplace_back(bound, PI, 0);
34     FOR (j, 0, n) {
35         if (i == j) continue;
36         if (c_c_relation(c, cs[j]) <= 2) {
37             if (sgn(cs[j].r - c.r) >= 0) // 完全被另一个圆包含, 等于说叠了一层
38                 belong_to++;
39             continue;
40         }
41         auto its = c_c_intersection(c, cs[j]);
42         if (its.size() == 2) {
43             P p = its[1] - c.p, q = its[0] - c.p;
44             LD a = angle(p), b = angle(q);
45             if (sgn(a - b) > 0) {
46                 ev.emplace_back(p, a, 1);
47                 ev.emplace_back(bound, PI, -1);
48                 ev.emplace_back(bound, -PI, 1);
49                 ev.emplace_back(q, b, -1);
50             } else {
51                 ev.emplace_back(p, a, 1);
52                 ev.emplace_back(q, b, -1);
53             }
54         }
55     }
56     sort(ev.begin(), ev.end());
57     int cc = ev[0].t;
58     FOR (j, 1, ev.size()) {
59         int t = cc + belong_to;
60         ans[t] += cross(ev[j - 1].p + c.p, ev[j].p + c.p) / 2;
61         ans[t] += cv_area(c.r, ev[j - 1], ev[j]);
62         cc += ev[j].t;
63     }
64 }
65 }

```

最小圆覆盖

- 随机增量。期望复杂度 $O(n)$ 。

```

1  P compute_circle_center(P a, P b) { return (a + b) / 2; }
2  bool p_in_circle(const P& p, const C& c) {
3      return sgn(dist(p - c.p) - c.r) <= 0;
4  }
5  C min_circle_cover(const vector<P> &in) {
6      vector<P> a(in.begin(), in.end());
7      dbg(a.size());
8      random_shuffle(a.begin(), a.end());
9      P c = a[0]; LD r = 0; int n = a.size();
10     FOR (i, 1, n) if (!p_in_circle(a[i], {c, r})) {
11         c = a[i]; r = 0;
12         FOR (j, 0, i) if (!p_in_circle(a[j], {c, r})) {
13             c = compute_circle_center(a[i], a[j]);
14             r = dist(a[j] - c);
15             FOR (k, 0, j) if (!p_in_circle(a[k], {c, r})) {
16                 c = compute_circle_center(a[i], a[j], a[k]);
17                 r = dist(a[k] - c);
18             }
19         }
20     }
21     return {c, r};

```

```
22 }
```

圖的反演

```
1 C inv(C c, const P& o) {
2     LD d = dist(c.p - o);
3     assert(sgn(d) != 0);
4     LD a = 1 / (d - c.r);
5     LD b = 1 / (d + c.r);
6     c.r = (a - b) / 2 * R2;
7     c.p = o + (c.p - o) * ((a + b) * R2 / 2 / d);
8     return c;
9 }
```

三维计算几何

```
1 struct P;
2 struct L;
3 typedef P V;
4
5 struct P {
6     LD x, y, z;
7     explicit P(LD x = 0, LD y = 0, LD z = 0): x(x), y(y), z(z) {}
8     explicit P(const L& l);
9 };
10
11 struct L {
12     P s, t;
13     L() {}
14     L(P s, P t): s(s), t(t) {}
15 };
16
17 struct F {
18     P a, b, c;
19     F() {}
20     F(P a, P b, P c): a(a), b(b), c(c) {}
21 };
22
23 P operator + (const P& a, const P& b) { return P(a.x + b.x, a.y + b.y, a.z + b.z); }
24 P operator - (const P& a, const P& b) { return P(a.x - b.x, a.y - b.y, a.z - b.z); }
25 P operator * (const P& a, LD k) { return P(a.x * k, a.y * k, a.z * k); }
26 P operator / (const P& a, LD k) { return P(a.x / k, a.y / k, a.z / k); }
27 inline int operator < (const P& a, const P& b) {
28     return sgn(a.x - b.x) < 0 || (sgn(a.x - b.x) == 0 && (sgn(a.y - b.y) < 0 ||
29         (sgn(a.y - b.y) == 0 && sgn(a.z - b.z) < 0)));
30 }
31 bool operator == (const P& a, const P& b) { return !sgn(a.x - b.x) && !sgn(a.y - b.y) && !sgn(a.z - b.z); }
32 P::P(const L& l) { *this = l.t - l.s; }
33 ostream &operator << (ostream &os, const P &p) {
34     return (os << "(" << p.x << ", " << p.y << ", " << p.z << ")");
35 }
36 istream &operator >> (istream &is, P &p) {
37     return (is >> p.x >> p.y >> p.z);
38 }
39
40 // -----
41 LD dist2(const P& p) { return p.x * p.x + p.y * p.y + p.z * p.z; }
42 LD dist(const P& p) { return sqrt(dist2(p)); }
43 LD dot(const V& a, const V& b) { return a.x * b.x + a.y * b.y + a.z * b.z; }
44 P cross(const P& v, const P& w) {
45     return P(v.y * w.z - v.z * w.y, v.z * w.x - v.x * w.z, v.x * w.y - v.y * w.x);
46 }
47 LD mix(const V& a, const V& b, const V& c) { return dot(a, cross(b, c)); }
```

旋转

```
1 // 逆时针旋转 r 弧度
2 // axis = 0 绕 x 轴
3 // axis = 1 绕 y 轴
4 // axis = 2 绕 z 轴
```

```

5 P rotation(const P& p, const LD& r, int axis = 0) {
6     if (axis == 0)
7         return P(p.x * cos(r) - p.z * sin(r), p.y * sin(r) + p.z * cos(r));
8     else if (axis == 1)
9         return P(p.z * cos(r) - p.x * sin(r), p.y, p.z * sin(r) + p.x * cos(r));
10    else if (axis == 2)
11        return P(p.x * cos(r) - p.y * sin(r), p.x * sin(r) + p.y * cos(r), p.z);
12 }
13 // n 是单位向量 表示旋转轴
14 // 模板是顺时针的
15 P rotation(const P& p, const LD& r, const P& n) {
16     LD c = cos(r), s = sin(r), x = n.x, y = n.y, z = n.z;
17     // dbg(c, s);
18     return P((x * x * (1 - c) + c) * p.x + (x * y * (1 - c) + z * s) * p.y + (x * z * (1 - c) - y * s) * p.z,
19             (x * y * (1 - c) - z * s) * p.x + (y * y * (1 - c) + c) * p.y + (y * z * (1 - c) + x * s) * p.z,
20             (x * z * (1 - c) + y * s) * p.x + (y * z * (1 - c) - x * s) * p.y + (z * z * (1 - c) + c) * p.z);
21 }

```

线、面

函数相互依赖，所以交织在一起了。

```

1 // 点在线段上 <= 0 包含端点 < 0 则不包含
2 bool p_on_seg(const P& p, const L& seg) {
3     P a = seg.s, b = seg.t;
4     return !sgn(dist2(cross(p - a, b - a))) && sgn(dot(p - a, p - b)) <= 0;
5 }
6 // 点到直线距离
7 LD dist_to_line(const P& p, const L& l) {
8     return dist(cross(l.s - p, l.t - p)) / dist(l);
9 }
10 // 点到线段距离
11 LD dist_to_seg(const P& p, const L& l) {
12     if (l.s == l.t) return dist(p - l.s);
13     V vs = p - l.s, vt = p - l.t;
14     if (sgn(dot(l, vs)) < 0) return dist(vs);
15     else if (sgn(dot(l, vt)) > 0) return dist(vt);
16     else return dist_to_line(p, l);
17 }
18
19 P norm(const F& f) { return cross(f.a - f.b, f.b - f.c); }
20 int p_on_plane(const F& f, const P& p) { return sgn(dot(norm(f), p - f.a)) == 0; }
21
22 // 判两点在线段异侧 点在线段上返回 0 不共面无意义
23 int opposite_side(const P& u, const P& v, const L& l) {
24     return sgn(dot(cross(P(l), u - l.s), cross(P(l), v - l.s))) < 0;
25 }
26
27 bool parallel(const L& a, const L& b) { return !sgn(dist2(cross(P(a), P(b)))); }
28 // 线段相交
29 int s_intersect(const L& u, const L& v) {
30     return p_on_plane(F(u.s, u.t, v.s), v.t) &&
31            opposite_side(u.s, u.t, v) &&
32            opposite_side(v.s, v.t, u);
33 }

```

凸包

增量法。先将所有的点打乱顺序，然后选择四个不共面的点组成一个四面体，如果找不到说明凸包不存在。然后遍历剩余的点，不断更新凸包。对遍历到的点做如下处理。

1. 如果点在凸包内，则不更新。
2. 如果点在凸包外，那么找到所有原凸包上所有分隔了这个点可见面和不可见面的边，以这样的边的两个点和新的点创建新的面加入凸包中。

```

1
2 struct FT {
3     int a, b, c;
4     FT() {}
5     FT(int a, int b, int c) : a(a), b(b), c(c) {}

```

```

6   };
7
8   bool p_on_line(const P& p, const L& l) {
9       return !sgn(dist2(cross(p - l.s, P(l))));
10  }
11
12  vector<F> convex_hull(vector<P> &p) {
13      sort(p.begin(), p.end());
14      p.erase(unique(p.begin(), p.end()), p.end());
15      random_shuffle(p.begin(), p.end());
16      vector<FT> face;
17      FOR (i, 2, p.size()) {
18          if (p_on_line(p[i], L(p[0], p[1]))) continue;
19          swap(p[i], p[2]);
20          FOR (j, i + 1, p.size())
21              if (sgn(mix(p[1] - p[0], p[2] - p[1], p[j] - p[0]))) {
22                  swap(p[j], p[3]);
23                  face.emplace_back(0, 1, 2);
24                  face.emplace_back(0, 2, 1);
25                  goto found;
26              }
27      }
28  found:
29      vector<vector<int>> mk(p.size(), vector<int>(p.size())));
30      FOR (v, 3, p.size()) {
31          vector<FT> tmp;
32          FOR (i, 0, face.size()) {
33              int a = face[i].a, b = face[i].b, c = face[i].c;
34              if (sgn(mix(p[a] - p[v], p[b] - p[v], p[c] - p[v])) < 0) {
35                  mk[a][b] = mk[b][a] = v;
36                  mk[b][c] = mk[c][b] = v;
37                  mk[c][a] = mk[a][c] = v;
38              } else tmp.push_back(face[i]);
39          }
40          face = tmp;
41          FOR (i, 0, tmp.size()) {
42              int a = face[i].a, b = face[i].b, c = face[i].c;
43              if (mk[a][b] == v) face.emplace_back(b, a, v);
44              if (mk[b][c] == v) face.emplace_back(c, b, v);
45              if (mk[c][a] == v) face.emplace_back(a, c, v);
46          }
47      }
48      vector<F> out;
49      FOR (i, 0, face.size())
50          out.emplace_back(p[face[i].a], p[face[i].b], p[face[i].c]);
51      return out;
52  }

```