Computational Geometry Algorithm Library

WenXin

December 13, 2018

Contents

1 二维几何

1.1 基础工具

```cpp
const int INF = 0x3f3f3f3f;
const double eps = 1e-6;
const double pi = acos(-1.0);

struct Point
{
        double x, y;
        Point(double x = 0, double y = 0) : x(x), y(y) {}
};

typedef Point Vector;

Vector operator+(Vector A, Vector B) { return Vector(A.x + B.x, A.y + B.y); }
Vector operator-(Point A, Point B) { return Vector(A.x - B.x, A.y - B.y); }
Vector operator*(Vector A, double p) { return Vector(A.x * p, A.y * p); }
Vector operator/(Vector A, double p) { return Vector(A.x / p, A.y / p); }
bool operator<(const Point &a, const Point &b)
{
        return a.x < b.x || (a.x == b.x && a.y < b.y);
}
int dcmp(double x)
{
        if (fabs(x) < eps)
                return 0;
        else
                return x < 0 ? -1 : 1;
}
bool operator==(const Point &a, const Point &b)
{
        return dcmp(a.x - b.x) == 0 && dcmp(a.y - b.y) == 0;
}
```

1.2 点与线段

```cpp
//向量点乘
double Dot(Vector A, Vector B) { return A.x * B.x + A.y * B.y; }
//向量模长
double Length(Vector A) { return sqrt(Dot(A, A)); }
//向量模长平方
double Length2(Vector A) { return Dot(A, A); }
//向量夹角
double Angle(Vector A, Vector B) { return acos(Dot(A, B) / Length(A) / Length(B)); }
//向量叉乘
double Cross(Vector A, Vector B) { return A.x * B.y - A.y * B.x; }
//三角形面积的 2 倍
double Area2(Point A, Point B, Point C) { return Cross(B - A, C - A); }
//向量旋转
Vector Rotate(Vector A, double rad)
```

```
{
    return Vector(A.x * cos(rad) - A.y * sin(rad), A.x * sin(rad) + A.y * cos(rad));
}
//向量单位法向量（逆时针 90°）
Vector Normal(Vector A)
{
    double L = Length(A);
    return Vector(-A.y / L, A.x / L);
}
//直线交点
Point GetLineIntersection(Point P, Vector v, Point Q, Vector w)
{
    Vector u = P - Q;
    double t = Cross(w, u) / Cross(v, w);
    return P + v * t;
}
//两点距离
double Distance(Point A, Point B) { return sqrt(Dot(B - A, B - A)); }
//两点距离平方
double Distance2(Point A, Point B) { return Dot(B - A, B - A); }
//点到直线的距离
double DistanceToLine(Point P, Point A, Point B)
{
    Vector v1 = B - A, v2 = P - A;
    return fabs(Cross(v1, v2)) / Length(v1);
}
//点到线段的距离
double DistanceToSegment(Point P, Point A, Point B)
{
    if (A == B)
        return Length(P - A);
    Vector v1 = B - A, v2 = P - A, v3 = P - B;
    if (dcmp(Dot(v1, v2)) < 0)
        return Length(v2);
    else if (dcmp(Dot(v1, v3)) > 0)
        return Length(v3);
    else
        return fabs(Cross(v1, v2)) / Length(v1);
}
//点在直线上的垂足
Point GetLineProjection(Point P, Point A, Point B)
{
    Vector v = B - A;
    return A + v * (Dot(v, P - A) / Dot(v, v));
}
//判断线段是否规范相交（不考虑端点）
bool SegmentProperIntersection(Point a1, Point a2, Point b1, Point b2)
{
    double c1 = Cross(a2 - a1, b1 - a1), c2 = Cross(a2 - a1, b2 - a1),
           c3 = Cross(b2 - b1, a1 - b1), c4 = Cross(b2 - b1, a2 - b1);
    return dcmp(c1) * dcmp(c2) < 0 && dcmp(c3) * dcmp(c4) < 0;
}
//判断点是否在线段上
```

```
bool OnSegment(Point p, Point a1, Point a2)
{
    return dcmp(Cross(a1 - p, a2 - p)) == 0 && dcmp(Dot(a1 - p, a2 - p)) <= 0;
}
//判断线段是否相交
bool SegmentIntersection(Point a1, Point a2, Point b1, Point b2)
{
    if (dcmp(Cross(a2 - a1, b2 - b1)) == 0)
        return OnSegment(a1, b1, b2) || OnSegment(a2, b1, b2) || OnSegment(b1, a1,
        ↪  a2) || OnSegment(b2, a1, a2);
    else
    {
        Point p = GetLineIntersection(a1, a2 - a1, b1, b2 - b1);
        return OnSegment(p, a1, a2) && OnSegment(p, b1, b2);
    }
}
```

1.3 多边形

```
typedef Point *Polygon;
//or
//vector<Point>
//多边形面积
double PolygonArea(Point *p, int n)
{
        double area = 0;
        for (int i = 1; i < n - 1; i++)
                area += Cross(p[i] - p[0], p[i + 1] - p[0]);
        return area / 2;
}
//点在多边形内判定（用到叉积）
bool IsPointInPolygon(Point p, Polygon poly, int n)
{
        int wn = 0;
        for (int i = 0; i < n; ++i)
        {
                if (OnSegment(p, poly[(i + 1) % n], poly[i]))
                        return true;
                int k = dcmp(Cross(poly[(i + 1) % n] - poly[i], p - poly[i]));
                int d1 = dcmp(poly[i].y - p.y);
                int d2 = dcmp(poly[(i + 1) % n].y - p.y);
                if (k > 0 && d1 <= 0 && d2 > 0)
                        ++wn;
                if (k < 0 && d2 <= 0 && d1 > 0)
                        --wn;
        }
        if (wn != 0)
                return true;
        return false;
}
//计算凸包，输入点数组 p，个数为 n，输出点数组 ch。函数返回凸包顶点数。
//输入不能有重复点。函数执行完后输入点的顺序被破坏
//如果不希望在凸包的边上有输入点，把两个 <= 改成 <
```

```cpp
//在精度较高时建议使用 dcmp
int ConvexHull(Polygon p, int n, Polygon ch)
{
	sort(p, p + n);
	int m = 0;
	for (int i = 0; i < n; i++)
	{
		while (m > 1 && Cross(ch[m - 1] - ch[m - 2], p[i] - ch[m - 2]) <= 0)
			m--;
		ch[m++] = p[i];
	}
	int k = m;
	for (int i = n - 2; i >= 0; i--)
	{
		while (m > k && Cross(ch[m - 1] - ch[m - 2], p[i] - ch[m - 2]) <= 0)
			m--;
		ch[m++] = p[i];
	}
	if (n > 1)
		m--;
	return m;
}
// 点集凸包
// 如果不希望在凸包的边上有输入点，把两个 <= 改成 <
// 注意：输入点集会被修改
vector<Point> ConvexHull(vector<Point> &p)
{
	// 预处理，删除重复点
	sort(p.begin(), p.end());
	p.erase(unique(p.begin(), p.end()), p.end());

	int n = p.size();
	int m = 0;
	vector<Point> ch(n + 1);
	for (int i = 0; i < n; i++)
	{
		while (m > 1 && Cross(ch[m - 1] - ch[m - 2], p[i] - ch[m - 2]) <= 0)
			m--;
		ch[m++] = p[i];
	}
	int k = m;
	for (int i = n - 2; i >= 0; i--)
	{
		while (m > k && Cross(ch[m - 1] - ch[m - 2], p[i] - ch[m - 2]) <= 0)
			m--;
		ch[m++] = p[i];
	}
	if (n > 1)
		m--;
	ch.resize(m);
	return ch;
}
// 旋转卡壳
```

```cpp
// 返回点集直径的平方（对踵点对）
int diameter2(vector<Point> &points)
{
	vector<Point> p = ConvexHull(points);
	int n = p.size();
	if (n == 1)
		return 0;
	if (n == 2)
		return Distance2(p[0], p[1]);
	p.push_back(p[0]); // 免得取模
	int ans = 0;
	for (int u = 0, v = 1; u < n; u++)
	{
		// 一条直线贴住边 p[u]-p[u+1]
		for (;;)
		{
			// 当 Area(p[u], p[u+1], p[v+1]) <= Area(p[u], p[u+1], p[v]) 时停止旋转
			// 即 Cross(p[u+1]-p[u], p[v+1]-p[u]) - Cross(p[u+1]-p[u], p[v]-p[u]) <= 0
			// 根据 Cross(A,B) - Cross(A,C) = Cross(A,B-C)
			// 化简得 Cross(p[u+1]-p[u], p[v+1]-p[v]) <= 0
			int diff = Cross(p[u + 1] - p[u], p[v + 1] - p[v]);
			if (diff <= 0)
			{
				ans = max(ans, Distance2(p[u], p[v])); // u 和 v 是对踵点
				if (diff == 0)
					ans = max(ans, Distance2(p[u], p[v + 1])); // diff == 0 时 u 和 v+1 也是对踵点
				break;
			}
			v = (v + 1) % n;
		}
	}
	return ans;
}
```

1.4 圆

```cpp
struct Line
{
	Point v;
	Vector p;
	Line(Point v, Vector p) : v(v), p(p) {}
	Point point(double t)
	{
		return v + p * t;
	}
};

struct Circle
{
```

```
14          Point c;
15          double r;
16          Circle(Point c, double r) : c(c), r(r) {}
17          Point point(double a)
18          {
19                  return Point(c.x + r * cos(a), c.y + r * sin(a));
20          }
21  };
22
23  int getLineCircleIntersection(Line L, Circle C, vector<Point> &sol)
24  {
25          double d = DistanceToLine(C.c, L.v, L.v + L.p);
26          if (dcmp(d - C.r) > 0)
27                  return 0;
28          else if (dcmp(d - C.r) == 0)
29          {
30                  Point P = L.v + L.p * (c - L.v, L.p) / Dot(L.p, L.p);
31                  sol.push_back(P);
32                  return 1;
33          }
34          else
35          {
36                  double l = sqrt(r * C.r - d * d);
37                  Point P = L.v + L.p * (c - L.v, L.p) / Dot(L.p, L.p);
38                  Vector e = Normal(L.p);
39                  sol.push_back(P - e * l);
40                  sol.push_bcak(P + e * l);
41                  return 2;
42          }
43  }
44  int getLineCircleIntersection(Line L, Circle C, double &t1, double &t2,
    ↪  vector<Point> &sol)
45  {
46          double a = L.v.x, b = L.p.x - C.c.x, c = L.v.y, d = L.p.y - C.c.y;
47          double e = a * a + c * c, f = 2 * (a * b + c * d), g = b * b + d * d - C.r *
            ↪  C.r;
48          double delta = f * f - 4 * e * g;
49          if (dcmp(delta) < 0)
50                  return 0;
51          if (dcmp(delta) == 0)
52          {
53                  t1 = t2 = -f / (2 * e);
54                  sol.push_back(L.point(t1));
55                  return 1;
56          }
57          t1 = (-f - sqrt(delta)) / (2 * e);
58          sol.push_back(L.point(t1));
59          t2 = (-f + sqrt(delta)) / (2 * e);
60          sol.push_back(L.point(t2));
61          return 2;
62  }
63
64  double angle(Vector v) { return atan2(v.y, v.x); }
```

```cpp

int getCircleCircleIntersection(Circle C1, Circle C2, vector<Point> &sol)
{
        double d = Length(C1.c - C2.c);
        if (dcmp(d) == 0)
        {
                if (dcmp(C1.r - C2.r) == 0)
                        return -1; //两圆重合
                return 0;
        }
        if (dcmp(C1.r + C2.r - d) < 0)
                return 0;
        if (dcmp(fabs(C1.r - C2.r) - d) > 0)
                return 0;

        double a = angle(C2.c - C1.c); //向量 C1C2 的极角
        double da = acos((C1.r * C1.r + d * d - C2.r * C2.r) / (2 * C1.r * d));
        //C1C2 到 C1P1 的角
        Point p1 = C1.point(a - da), p2 = C1.point(a + da);

        sol.push_back(p1);
        if (p1 == p2)
                return 1;
        sol.push_back(p2);
        return 2;
}
//过点 p 到圆 C 的切线。v[i] 是第 i 条切线的向量。返回切线条数
int getTangents(Point p, Circle C, Vector *v)
{
        Vector u = C.c - p;
        double dist = Length(u);
        if (dist < C.r)
                return 0;
        else if (dcmp(dist - C.r) == 0)
        { //p 在圆上，只有一条切线
                v[0] = Rotate(u, pi / 2);
                return 1
        }
        else
        {
                double ang = asin(C.r / dist);
                v[0] = Rotate(u, -ang);
                v[1] = Rotate(u, +ang);
                return 2;
        }
}
//两圆公切线 返回切线的条数，-1 代表无穷条
//a[i] 和 b[i] 分别是第 i 条切线在圆 A 和圆 B 上的切点
int getTangents(Circle A, Circle B, Point *a, Point *b)
{
        int cnt = 0;
        if (A.r < B.r)
        {
```

```
118                 swap(A, B);
119                 swap(a, b);
120         }
121         int d2 = (A.x - B.x) * (A.x - B.x) + (A.y - B.y) * (A.y - B.y);
122         int rdiff = A.r - B.r;
123         int rsum = A.r + B.r;
124         if (d2 < rdiff * rdiff)
125                 return 0; //内含
126 
127         double base = atan2(B.y - A.y, B.x - A.x);
128         if (d2 == 0 && A.r == B.r)
129                 return -1; //无穷多条切线
130         if (d2 == rdiff * rdiff)
131         { //内切, 1 条切线
132                 a[cnt] = A.point(base);
133                 b[cnt] = B.point(base);
134                 cnt++;
135                 return 1;
136         }
137         //有外公切线
138         double ang = acos((A.r - B.r) / sqrt(d2));
139         a[cnt] = A.point(base + ang);
140         b[cnt] = B.point(base + ang);
141         cnt++;
142         a[cnt] = A.point(base - ang);
143         b[cnt] = B.point(base - ang);
144         cnt++;
145         if (d2 == rsum * rsum)
146         { //一条内公切线
147                 a[cnt] = A.point(base);
148                 b[cnt] = B.point(pi + base);
149                 cnt++;
150         }
151         else if (d2 > rsum * rsum)
152         { //两条内公切线
153                 double ang = acos((A.r + B.r) / sqrt(d2));
154                 a[cnt] = A.point(base + ang);
155                 b[cnt] = B.point(pi + base + ang);
156                 cnt++;
157                 a[cnt] = A.point(base - ang);
158                 b[cnt] = B.point(pi + base - ang);
159                 cnt++;
160         }
161         return cnt;
162 }
163 //三角形外接圆
164 Circle CircumscribedCircle(Point p1, Point p2, Point p3)
165 {
166         double Bx = p2.x - p1.x, By = p2.y - p1.y;
167         double Cx = p3.x - p1.x, Cy = p3.y - p1.y;
168         double D = 2 * (Bx * Cy - By * Cx);
169         double cx = (Cy * (Bx * Bx + By * By) - By * (Cx * Cx + Cy * Cy)) / D +
          ↪  p1.x;
```

```
        double cy = (Bx * (Cx * Cx + Cy * Cy) - Cy * (Bx * Bx + By * By)) / D +
        ↪  p1.y;
        Point p = Point(cx, cy);
        return Circle(p, Length(p1 - p));
}
//三角形内切圆
Circle InscribedCircle(Point p1, Point p2, Point p3)
{
        double a = Length(p2 - p3);
        double b = Length(p3 - p1);
        double c = Length(p1 - p2);
        Point p = (p1 * a + p2 * b + p3 * c) / (a + b + c);
        return Circle(p, DistanceToLine(p, p1, p2));
}
```

1.5 球面

```
//角度转弧度
double torad(double deg)
{
        return deg / 180 * acos(-1); //acos(-1)==pi
}
//经纬度（角度）转化为空间坐标
void get_coord(double R, double lat, double lng, double &x, double &y, double &z)
{
        lat = torad(lat);
        lng = torad(lng);
        x = R * cos(lat) * cos(lng);
        y = R * cos(lat) * sin(lng);
        z = R * sin(lat);
}
```

1.6 半平面交

```
// 有向直线 它的左边就是对应的半平面
struct Line
{
        Point P;        // 直线上任意一点
        Vector v;   // 方向向量
        double ang; // 极角，即从 x 正半轴旋转到向量 v 所需要的角（弧度）
        Line() {}
        Line(Point P, Vector v) : P(P), v(v) { ang = atan2(v.y, v.x); }
        bool operator<(const Line &L) const
        {
                return ang < L.ang;
        }
};
// 点 p 在有向直线 L 的左边（线上不算）
bool OnLeft(const Line &L, const Point &p)
{
        return Cross(L.v, p - L.P) > 0;
}
```

```
// 二直线交点，假定交点惟一存在
Point GetLineIntersection(const Line &a, const Line &b)
{
        Vector u = a.P - b.P;
        double t = Cross(b.v, u) / Cross(a.v, b.v);
        return a.P + a.v * t;
}
const double INF = 1e8;
const double eps = 1e-6;
// 半平面交主过程
vector<Point> HalfplaneIntersection(vector<Line> L)
{
        int n = L.size();
        sort(L.begin(), L.end()); // 按极角排序

        int first, last;        // 双端队列的第一个元素和最后一个元素的下标
        vector<Point> p(n); // p[i] 为 q[i] 和 q[i+1] 的交点
        vector<Line> q(n);  // 双端队列
        vector<Point> ans;  // 结果

        q[first = last = 0] = L[0]; // 双端队列初始化为只有一个半平面 L[0]
        for (int i = 1; i < n; i++)
        {
                while (first < last && !OnLeft(L[i], p[last - 1]))
                        last--;
                while (first < last && !OnLeft(L[i], p[first]))
                        first++;
                q[++last] = L[i];
                if (fabs(Cross(q[last].v, q[last - 1].v)) < eps)
                { // 两向量平行且同向，取内侧的一个
                        last--;
                        if (OnLeft(q[last], L[i].P))
                                q[last] = L[i];
                }
                if (first < last)
                        p[last - 1] = GetLineIntersection(q[last - 1], q[last]);
        }
        while (first < last && !OnLeft(q[first], p[last - 1]))
                last--; // 删除无用平面
        if (last - first <= 1)
                return
                 ↪  ans;
                 ↪  // 空集
        p[last] = GetLineIntersection(q[last], q[first]); // 计算首尾两个半平面的交点

        // 从 deque 复制到输出中
        for (int i = first; i <= last; i++)
                ans.push_back(p[i]);
        return ans;
}
//LA 3890
//二分法求凸 n 边形内距边界最远点，输出距离
int n;
```

```
while (scanf("%d", &n) == 1 && n)
{
        vector<Vector> p, v, normal;

        int m, x, y;
        for (int i = 0; i < n; i++)
        {
                scanf("%d%d", &x, &y);
                p.push_back(Point(x, y));
        }
        if (PolygonArea(p) < 0)
                reverse(p.begin(), p.end());

        for (int i = 0; i < n; i++)
        {
                v.push_back(p[(i + 1) % n] - p[i]);
                normal.push_back(Normal(v[i]));
        }

        double left = 0, right = 20000;
        while (right - left > 1e-6)
        {
                vector<Line> L;
                double mid = left + (right - left) / 2;
                for (int i = 0; i < n; i++)
                        L.push_back(Line(p[i] + normal[i] * mid, v[i]));
                vector<Point> poly = HalfplaneIntersection(L);
                if (poly.empty())
                        right = mid;
                else
                        left = mid;
        }
        printf("%.6lf\n", left);
}
```

2 三维几何

2.1 基础工具

```
struct Point3
{
        double x, y, z;
        Point3(double x = 0, double y = 0, double z = 0) : x(x), y(y), z(z) {}
};

typedef Point3 Vector3;

Vector3 operator+(const Vector3 &A, const Vector3 &B) { return Vector3(A.x + B.x, A.y + B.y, A.z + B.z); }
Vector3 operator-(const Point3 &A, const Point3 &B) { return Vector3(A.x - B.x, A.y - B.y, A.z - B.z); }
Vector3 operator*(const Vector3 &A, double p) { return Vector3(A.x * p, A.y * p, A.z * p); }
Vector3 operator/(const Vector3 &A, double p) { return Vector3(A.x / p, A.y / p, A.z / p); }

const double eps = 1e-8;
int dcmp(double x)
{
        if (fabs(x) < eps)
                return 0;
        else
                return x < 0 ? -1 : 1;
}

double Dot(const Vector3 &A, const Vector3 &B) { return A.x * B.x + A.y * B.y + A.z * B.z; }
double Length(const Vector3 &A) { return sqrt(Dot(A, A)); }
double Angle(const Vector3 &A, const Vector3 &B) { return acos(Dot(A, B) / Length(A) / Length(B)); }
Vector3 Cross(const Vector3 &A, const Vector3 &B) { return Vector3(A.y * B.z - A.z * B.y, A.z * B.x - A.x * B.z, A.x * B.y - A.y * B.x); }

Point3 read_point3()
{
        Point3 p;
        scanf("%lf%lf%lf", &p.x, &p.y, &p.z);
        return p;
}
```

2.2 距离/面积/体积

```
double Area2(const Point3 &A, const Point3 &B, const Point3 &C) { return Length(Cross(B - A, C - A)); }

//点 P 到直线 AB 的距离
double DistanceToLine(Point3 P, Point3 A, Point3 B)
```

```
{
        Vector3 v1 = B - A, v2 = P - A;
        reutrn Length(Cross(v1, v2)) / Length(v1);
}
//点 P 到线段 AB 的距离
double DistanceToSegment(Point3 P, Point3 A, Point3 B)
{
        if (A == B)
                return Length(P - A);
        Vector3 v1 = B - A, v2 = P - A, v3 = P - B;
        if (dcmp(Dot(v1, v2)) < 0)
                return Length(v2);
        else if (dcmp(Dot(v1, v3)) > 0)
                return Length(v3);
        else
                return Length(Cross(v1, v2)) / Length(v1);
}
//返回 AB, AC, AD 的混合积。它也等于四面体 ABCD 的有向面积的 6 倍
double Volume6(Point3 A, Point3 B, Point3 C, Point3 D)
{
        return Dot(D - A, Cross(B - A, C - A));
}
```

2.3 三角形

```
//空间三点重心
Point3 Centroid(const Point3 &A, const Point3 &B, const Point3 &C, const Point3 &D)
 ↪  { return (A + B + C + D) / 4.0; }
// p1 和 p2 是否在线段 a-b 的同侧
bool SameSide(const Point3 &p1, const Point3 &p2, const Point3 &a, const Point3 &b)
{
        return dcmp(Dot(Cross(b - a, p1 - a), Cross(b - a, p2 - a))) >= 0;
}
// 点在三角形 P0, P1, P2 中
bool PointInTri(const Point3 &P, const Point3 &P0, const Point3 &P1, const Point3
 ↪  &P2)
{
        return SameSide(P, P0, P1, P2) && SameSide(P, P1, P0, P2) && SameSide(P, P2,
         ↪  P0, P1);
}
// 三角形 P0P1P2 是否和线段 AB 相交
bool TriSegIntersection(const Point3 &P0, const Point3 &P1, const Point3 &P2, const
 ↪  Point3 &A, const Point3 &B, Point3 &P)
{
        Vector3 n = Cross(P1 - P0, P2 - P0);
        if (dcmp(Dot(n, B - A)) == 0)
                return false; // 线段 A-B 和平面 P0P1P2 平行或共面
        else
        { // 平面 A 和直线 P1-P2 有惟一交点
                double t = Dot(n, P0 - A) / Dot(n, B - A);
                if (dcmp(t) < 0 || dcmp(t - 1) > 0)
                        return false;        // 不在线段 AB 上
                P = A + (B - A) * t; // 交点
```

```
                return PointInTri(P, P0, P1, P2);
        }
}
//判断空间三角形相交
bool TriTriIntersection(Point3 *T1, Point3 *T2)
{
        Point3 P;
        for (int i = 0; i < 3; i++)
        {
                if (TriSegIntersection(T1[0], T1[1], T1[2], T2[i], T2[(i + 1) % 3],
                ↪  P))
                        return true;
                if (TriSegIntersection(T2[0], T2[1], T2[2], T1[i], T1[(i + 1) % 3],
                ↪  P))
                        return true;
        }
        return false;
}
```

2.4 三维凸包

```
double rand01() { return rand() / (double)RAND_MAX; }
double randeps() { return (rand01() - 0.5) * eps; }
//加噪声
Point3 add_noise(const Point3 &p)
{
        return Point3(p.x + randeps(), p.y + randeps(), p.z + randeps());
}
//面
struct Face
{
        int v[3];
        Face(int a, int b, int c)
        {
                v[0] = a;
                v[1] = b;
                v[2] = c;
        }
        Vector3 Normal(const vector<Point3> &P) const
        {
                return Cross(P[v[1]] - P[v[0]], P[v[2]] - P[v[0]]);
        }
        // f 是否能看见 P[i]
        int CanSee(const vector<Point3> &P, int i) const
        {
                return Dot(P[i] - P[v[0]], Normal(P)) > 0;
        }
};
// 增量法求三维凸包
//遍历所有面，判断是否可见；然后遍历所有边，判断是否在阴影边界上
// 注意：没有考虑各种特殊情况（如四点共面）。实践中，请在调用前对输入点进行微小扰动
vector<Face> CH3D(const vector<Point3> &P)
{
```

```
        int n = P.size();
        vector<vector<int>> vis(n);
        for (int i = 0; i < n; i++)
                vis[i].resize(n);

        vector<Face> cur;
        cur.push_back(Face(0, 1, 2)); // 由于已经进行扰动，前三个点不共线
        cur.push_back(Face(2, 1, 0));
        for (int i = 3; i < n; i++)
        {
                vector<Face> next;
                // 计算每条边的“左面”的可见性
                for (int j = 0; j < cur.size(); j++)
                {
                        Face &f = cur[j];
                        int res = f.CanSee(P, i);
                        if (!res)
                                next.push_back(f);
                        for (int k = 0; k < 3; k++)
                                vis[f.v[k]][f.v[(k + 1) % 3]] = res;
                }
                for (int j = 0; j < cur.size(); j++)
                        for (int k = 0; k < 3; k++)
                        {
                                int a = cur[j].v[k], b = cur[j].v[(k + 1) % 3];
                                if (vis[a][b] != vis[b][a] && vis[a][b]) // (a,b) 是
                                ↪ 分界线，左边对 P[i] 可见
                                        next.push_back(Face(a, b, i));
                        }
                cur = next;
        }
        return cur;
}
//三维凸包结构体
struct ConvexPolyhedron
{
        int n;
        vector<Point3> P, P2;
        vector<Face> faces;
        //读入
        bool read()
        {
                if (scanf("%d", &n) != 1)
                        return false;
                P.resize(n);
                P2.resize(n);
                for (int i = 0; i < n; i++)
                {
                        P[i] = read_point3();
                        P2[i] = add_noise(P[i]);
                }
                faces = CH3D(P2);
                return true;
```

```
85          }
86          //求重心
87          Point3 centroid()
88          {
89                  Point3 C = P[0];
90                  double totv = 0;
91                  Point3 tot(0, 0, 0);
92                  for (int i = 0; i < faces.size(); i++)
93                  {
94                          Point3 p1 = P[faces[i].v[0]], p2 = P[faces[i].v[1]], p3 =
                            ↪ P[faces[i].v[2]];
95                          double v = -Volume6(p1, p2, p3, C);
96                          totv += v;
97                          tot = tot + Centroid(p1, p2, p3, C) * v;
98                  }
99                  return tot / totv;
100         }
101         //求重心到所有面的最短距离
102         double mindist(Point3 C)
103         {
104                 double ans = 1e30;
105                 for (int i = 0; i < faces.size(); i++)
106                 {
107                         Point3 p1 = P[faces[i].v[0]], p2 = P[faces[i].v[1]], p3 =
                            ↪ P[faces[i].v[2]];
108                         ans = min(ans, fabs(-Volume6(p1, p2, p3, C) / Area2(p1, p2,
                            ↪ p3)));
109                 }
110                 return ans;
111         }
112 };
```

2.5 有理数类

```
1  //有理数类
2  typedef long long LL;
3
4  LL gcd(LL a, LL b) { return b ? gcd(b, a % b) : a; }
5  LL lcm(LL a, LL b) { return a / gcd(a, b) * b; }
6
7  struct Rat
8  {
9          LL a, b;
10         Rat(LL a = 0) : a(a), b(1) {}
11         Rat(LL x, LL y) : a(x), b(y)
12         {
13                 if (b < 0)
14                         a = -a, b = -b;
15                 LL d = gcd(a, b);
16                 if (d < 0)
17                         d = -d;
18                 a /= d;
19                 b /= d;
```

```
        }
};

Rat operator+(const Rat &A, const Rat &B)
{
        LL x = lcm(A.b, B.b);
        return Rat(A.a * (x / A.b) + B.a * (x / B.b), x);
}

Rat operator-(const Rat &A, const Rat &B) { return A + Rat(-B.a, B.b); }
Rat operator*(const Rat &A, const Rat &B) { return Rat(A.a * B.a, A.b * B.b); }
//下为工具函数
void updatemin(Rat &A, const Rat &B)
{
        if (A.a * B.b > B.a * A.b)
                A.a = B.a, A.b = B.b;
}
// 点 P 到线段 AB 的距离的平方
Rat Rat_Distance2ToSegment(const Point3 &P, const Point3 &A, const Point3 &B)
{
        if (A == B)
                return Length2(P - A);
        Vector3 v1 = B - A, v2 = P - A, v3 = P - B;
        if (Dot(v1, v2) < 0)
                return Length2(v2);
        else if (Dot(v1, v3) > 0)
                return Length2(v3);
        else
                return Rat(Length2(Cross(v1, v2)), Length2(v1));
}
//异面直线最小距离
// 求异面直线 p1+su 和 p2+tv 的公垂线对应的 s。如果平行/重合，返回 false
bool Rat_LineDistance3D(const Point3 &p1, const Vector3 &u, const Point3 &p2, const
↪  Vector3 &v, Rat &s)
{
        LL b = (LL)Dot(u, u) * Dot(v, v) - (LL)Dot(u, v) * Dot(u, v);
        if (b == 0)
                return false;
        LL a = (LL)Dot(u, v) * Dot(v, p1 - p2) - (LL)Dot(v, v) * Dot(u, p1 - p2);
        s = Rat(a, b);
        return true;
}
//由线段参数形式得点的坐标
void Rat_GetPointOnLine(const Point3 &A, const Point3 &B, const Rat &t, Rat &x, Rat
↪  &y, Rat &z)
{
        x = Rat(A.x) + Rat(B.x - A.x) * t;
        y = Rat(A.y) + Rat(B.y - A.y) * t;
        z = Rat(A.z) + Rat(B.z - A.z) * t;
}
//两点距离平方
Rat Rat_Distance2(const Rat &x1, const Rat &y1, const Rat &z1, const Rat &x2, const
↪  Rat &y2, const Rat &z2)
```

```
{
        return (x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 - y2) + (z1 - z2) * (z1 -
        ↪  z2);
}
//线段最短距离
int main()
{
        int T;
        scanf("%d", &T);
        LL maxx = 0;
        while (T--)
        {
                Point3 A = read_point3();
                Point3 B = read_point3();
                Point3 C = read_point3();
                Point3 D = read_point3();
                Rat s, t;
                bool ok = false;
                Rat ans = Rat(1000000000);
                if (Rat_LineDistance3D(A, B - A, C, D - C, s))
                        if (s.a > 0 && s.a < s.b && Rat_LineDistance3D(C, D - C, A,
                        ↪  B - A, t))
                                if (t.a > 0 && t.a < t.b)
                                {
                                        ok = true; // 异面直线/相交直线
                                        Rat x1, y1, z1, x2, y2, z2;
                                        Rat_GetPointOnLine(A, B, s, x1, y1, z1);
                                        Rat_GetPointOnLine(C, D, t, x2, y2, z2);
                                        ans = Rat_Distance2(x1, y1, z1, x2, y2, z2);
                                }
                if (!ok)
                { // 平行直线/重合直线
                        updatemin(ans, Rat_Distance2ToSegment(A, C, D));
                        updatemin(ans, Rat_Distance2ToSegment(B, C, D));
                        updatemin(ans, Rat_Distance2ToSegment(C, A, B));
                        updatemin(ans, Rat_Distance2ToSegment(D, A, B));
                }
                printf("%lld %lld\n", ans.a, ans.b);
        }
        return 0;
}
```