目录

一、三维基础操作

```
const double EPS=0.000001;

typedef struct Point_3D {
    double x, y, z;
    Point_3D(double xx = 0, double yy = 0, double zz = 0): x(xx), y(yy), z(zz) {}

    bool operator == (const Point_3D& A) const {
        return x==A.x && y==A.y && z==A.z;
    }
};
typedef Point_3D Vector_3D;

struct Line_3D   //空间直线
{
    Point_3D a, b;
};

struct Plane_3D   //空间平面
{
    Point_3D a, b, c;
    Plane_3D(){}
    Plane_3D( Point_3D a, Point_3D b, Point_3D c ):
    a(a), b(b), c(c) { }
};

Point_3D read_Point_3D() {
    double x,y,z;
    scanf("%lf%lf%lf",&x,&y,&z);
    return Point_3D(x,y,z);
}

Vector_3D operator + (const Vector_3D & A, const Vector_3D & B) {
    return Vector_3D(A.x + B.x, A.y + B.y, A.z + B.z);
}

Vector_3D operator - (const Point_3D & A, const Point_3D & B) {
    return Vector_3D(A.x - B.x, A.y - B.y, A.z - B.z);
}

Vector_3D operator * (const Vector_3D & A, double p) {
    return Vector_3D(A.x * p, A.y * p, A.z * p);
}

Vector_3D operator / (const Vector_3D & A, double p) {
    return Vector_3D(A.x / p, A.y / p, A.z / p);
}

//取平面法向量
```

```
Point_3D NormalVector( plane3 s )
{
    return Cross_3D( Subt( s.a, s.b ), Subt( s.b, s.c ) );
}
Point_3D NormalVector( Point3 a, Point3 b, Point3 c )
{
    return Cross_3D( Subt( a, b ), Subt( b, c ) );
}

//两点距离
double TwoPointDistance( Point3 p1, Point3 p2 )
{
    return sqrt( (p1.x - p2.x)*(p1.x - p2.x) + (p1.y - p2.y)*(p1.y - p2.y) + (p1.z - p2.z)*(p1.z - p2.z) );
}

```

1.1 三维点积(Dot3)

```
double Dot_3D(const Vector_3D & A, const Vector_3D & B) {
    return A.x * B.x + A.y * B.y + A.z * B.z;
}

double Length(const Vector_3D & A) {
    return sqrt(Dot(A, A));
}

double Angle(const Vector_3D & A, const Vector_3D & B) {
    return acos(Dot(A, B) / Length(A) / Length(B));
}
```

1.2 三维叉积(Cross3)

可以认为叉积同时垂直于v1和v2，当且仅当v1和v2平行时，叉积为0。

```
Vector_3D Cross(const Vector_3D & A, const Vector_3D & B) {
    return Vector_3D(A.y * B.z - A.z * B.y, A.z * B.x - A.x * B.z, A.x * B.y - A.y * B.x);
}
// 三角形abc面积的两倍
double Area2(const Point_3D & A, const Point_3D & B, const Point_3D & C) {
    return Length(Cross(B - A, C - A));
}
```

过不共线三点的平面，法向量为`Cross(p2 - p0, p1 - p0)`，我们在任取一个点即可得到平面的点法式。

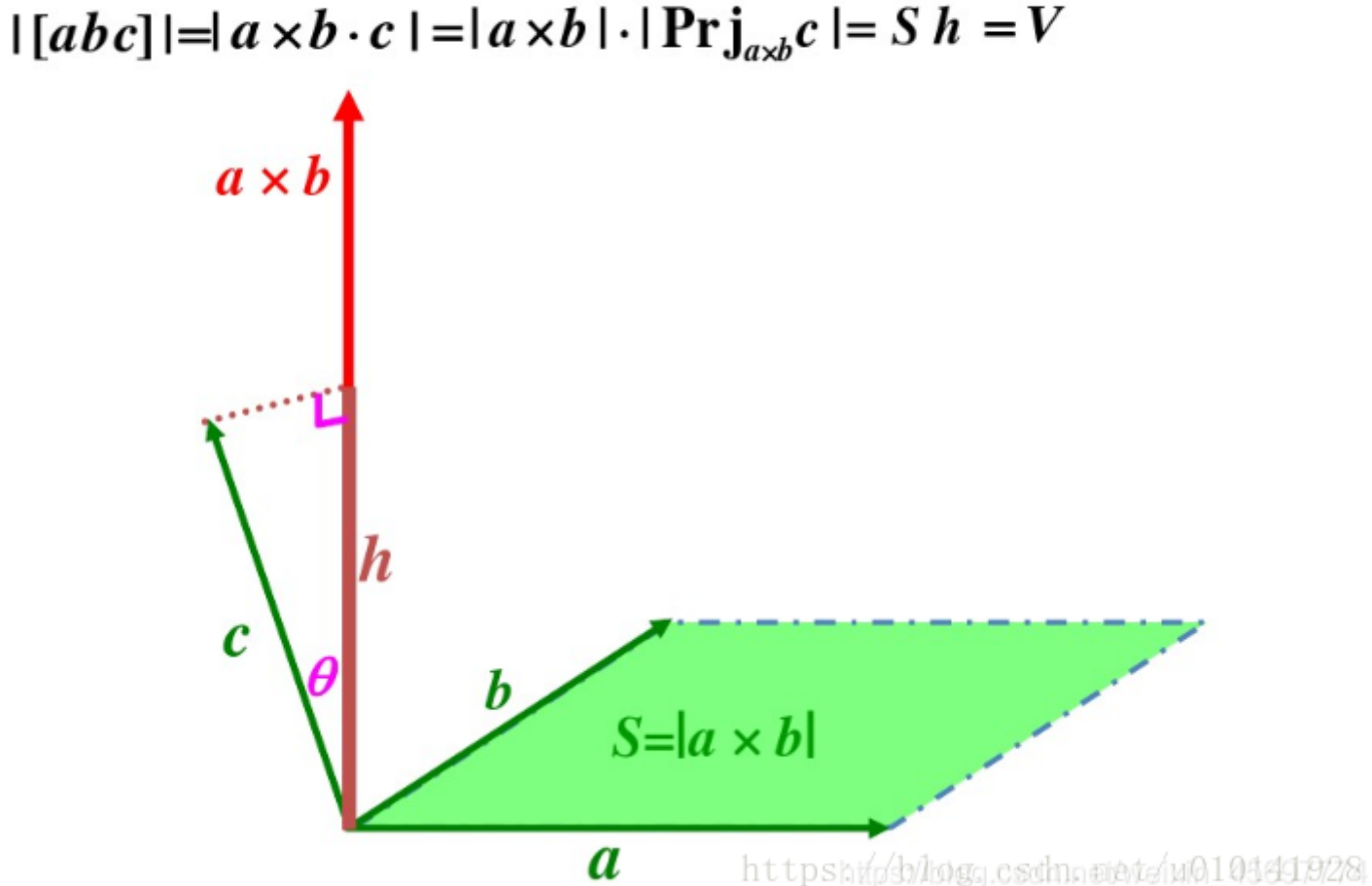
1.3 矢量差（Subt）

```
Point_3D Subt( Point3 u, Point3 v )
{
    Point_3D ret;
    ret.x = u.x - v.x;
    ret.y = u.y - v.y;
    ret.z = u.z - v.z;
    return ret;
}
```

1.4.1 返回ab，ac，ad的混合积(Volume6)

对于三个三维向量a, b, c，定义它们的混合积为$(a \times b) \cdot c$，其中$\times$ 表示叉乘，$\cdot$表示点乘，记为$[a\ b\ c]$

几何意义

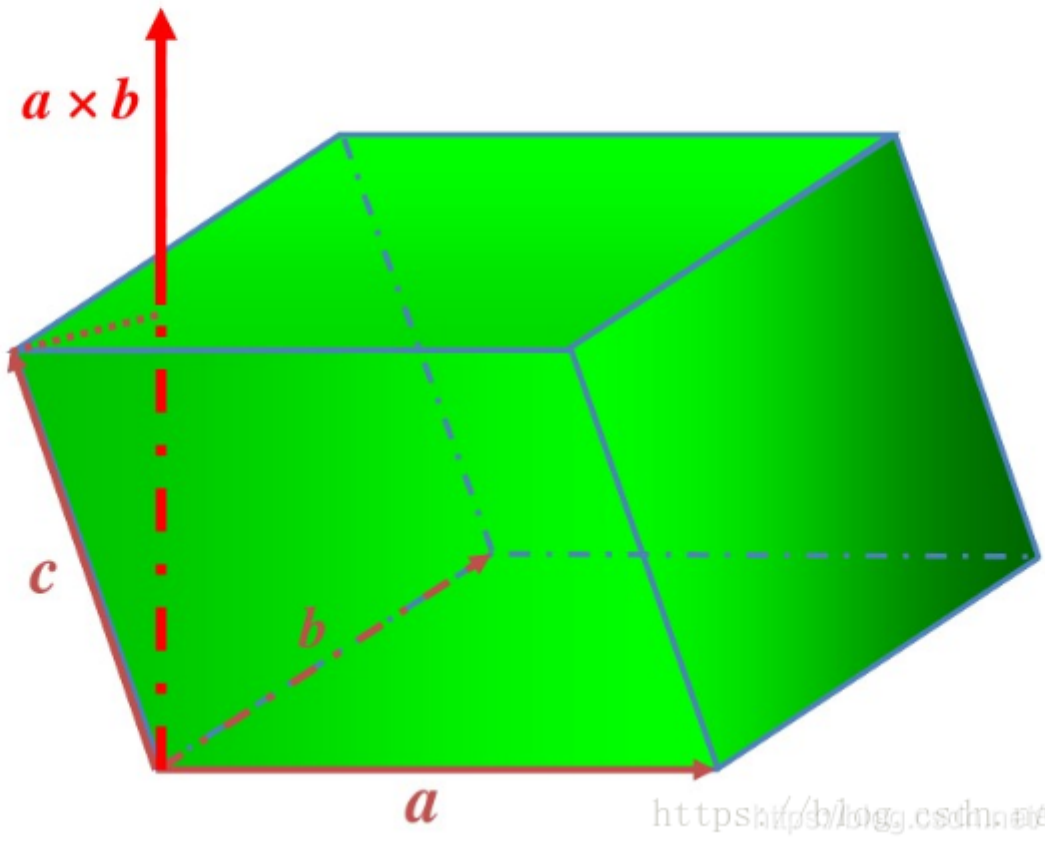


其中 $Prj_{a \times bc}$ 代表的是c这个向量在a×b这个向量上的投影

那么显然我们最后得到的是以这三个向量为三条临边的一个**六面体的体积**

几何意义 $|[abc]|=|a\times b\cdot c|=|a\times b|\cdot|\mathrm{Prj}_{a\times b}c|=Sh=V$

三矢 a, b, c 共面 $\Leftrightarrow$ 其混合积 $[abc]=0$




```cpp
// 返回ab，ac，ad的混合积。它等于四面体(三角形体)abcd的有
//向体积的6倍(六面体(正方形体)的体积)
double Volume6(const Point_3D & A, const Point_3D & B, const Point_3D & C, const Point_3D & D) {
    return Dot(D - A, Cross(B - A, C - A));
}
```

1.4.2 四面体体积(Volume6)

假设四面体的四个顶点分别为A,B,C,D，并设三个向量$a=B-A, b=C-A, c=D-A$，那么这个正四面体的体积就是$\frac{1}{6}[abc]$(混合积)

```cpp
// 返回ab，ac，ad的混合积。它等于四面体(三角形体)abcd的有
//向体积的6倍(六面体(正方形体)的体积)
double Volume6(const Point_3D & A, const Point_3D & B, const Point_3D & C, const Point_3D & D) {
    return Dot(D - A, Cross(B - A, C - A));
}
```

1.5 求四面体的重心(Centroid)

```cpp
// 四面体的重心
Point_3D Centroid(const Point_3D & A, const Point_3D & B, const Point_3D & C, const Point_3D & D)
{
    return (A + B + C + D) / 4.0;
}
```

1.6 凸多面体的重心

我们首先考虑凸多边形的重心

对于三角形，它的重心就是它所有坐标的平均值

那么对于凸多边形，我们把它三角剖分了，记第i块的重心为 a_i，面积为 m_i，那么凸多边形的重心就是

$$\frac{\sum_{i=1}^{n} a_i \times m_i}{\sum_{i=1}^{n} m_i}$$

那么凸多面体也差不多了，我们把它给四面体剖分了，然后也差不多按上面的算就好了

关于四面体剖分，具体的说我们在多面体中随便选一个点，比方说是p1，然后把每一个面给三角剖分，那么三角形就和选定的点构成了一个四面体。设vi表示第i个四面体的体积，ai表示重心，则最终多面体的重心为

$$\frac{\sum_{i=1}^{n} a_i \times v_i}{\sum_{i=1}^{n} v_i}$$

1.7 二面角

简单来说就是两个平面的夹角

我们假设现在有a,b,c三个向量，要求ab这个平面和ac这个平面的二面角

那么求出ab和ac的法向量（法向量可以直接用叉积算），两个法向量之间的夹角就是二面角了，法向量之间的夹角直接用点积除以长度计算(转换成单位法向量)

二、三维点线面

三维的直线仍然可以用参数方程（点+向量表示），同时射线和线段为“参数有取值范围”的直线

```
struct Line_3D    //空间直线
{
    Point_3D a, b;
};
```

三维平面通常使用点法式(p_0, n)表示，其中p_0是平面上的一个点，向量n是平面的法向量。（我们的平面把空间分成了两个部分，一般取法向量所背离的半空间）

法向量垂直于平面上所有直线，意味着平面上的任意点p满足$Dot(n, p - p0) = 0$。

我们设$p(x, y), p_0(x_0, y_0)$，法向量$n(A, B, C)$

化简得到平面的一般式：$Ax + By + Cz - D = 0(D = -(Ax_0 + By_0 + Cz_0))$

当$Ax + By + Cz - D > 0$时，上述点积大于0，说明点$p(x, y, z)$在版空间(p_0, n)之外。

```
struct Plane_3D    //空间平面
{
    Point_3D a, b, c;
    Plane_3D(){}
    Plane_3D( Point_3D a, Point_3D b, Point_3D c ):
    a(a), b(b), c(c) { }
};
```

2.1 取平面法向量(NormalVector)

```
Point3 NormalVector( plane_3D s )
{
    return Cross3( Subt( s.a, s.b ), Subt( s.b, s.c ) );
}
Point3 NormalVector( Point_3D a, Point_3D b, Point_3D c )
{
    return Cross3( Subt( a, b ), Subt( b, c ) );
}
```

2.2 求两点距离(TwoPointDistance)

```
double TwoPointDistance( Point_3D p1, Point_3D p2 )
{
    return sqrt( (p1.x - p2.x)*(p1.x - p2.x) + (p1.y - p2.y)*(p1.y - p2.y) + (p1.z - p2.z)*(p1.z - p2.z) );
}
```

2.3 点p到平面的距离(DistanceToPlane)

```
// 点p到平面p0-n的距离。n必须为单位向量
double DistanceToPlane(const Point_3D & p, const Point_3D & p0, const Vector_3D & n)
{
    return fabs(Dot(p - p0, n)); // 如果不取绝对值，得到的是有向距离
}
```

2.4 点p在平面上的投影(GetPlaneProjection)

```
// 点p在平面p0-n上的投影。n必须为单位向量(如果不是单位向量就/Length(n)嘛)
Point_3D GetPlaneProjection(const Point_3D & p, const Point_3D & p0, const Vector_3D & n)
{
    return p - n * Dot(p - p0, n);
}
```

2.5 直线与平面的交点(LinePlaneIntersection)

```
//直线p1-p2 与平面p0-n的交点
Point_3D LinePlaneIntersection(Point_3D p1, Point_3D p2, Point_3D p0, Vector_3D n)
{
    Vector_3D v= p2 - p1;
    double t = (Dot(n, p0 - p1) / Dot(n, p2 - p1)); //分母为0，直线与平面平行或在平面上
    return p1 + v * t; //如果是线段 判断t是否在0~1之间
}
```

2.6 空间直线距离(LineToLine)

```
//空间直线距离,tmp为两直线的公共法向量
double LineToLine( Line_3D u, Line_3D v, Point_3D& tmp )
{
    tmp = Cross3( Subt( u.a, u.b ), Subt( v.a, v.b ) );
    return fabs( Dot3( Subt(u.a, v.a), tmp ) ) / VectorLenth(tmp);
}
```

2.7 点到直线的距离(DistanceToLine)

```
// 点P到直线AB的距离
double DistanceToLine(const Point_3D & P, const Point_3D & A, const Point_3D & B)
{
    Vector_3D v1 = B - A, v2 = P - A;
    return Length(Cross(v1, v2)) / Length(v1);
}
```

2.8 点到线段的距离(DistanceToSeg)

```
double DistanceToSeg(Point_3D p, Point_3D a, Point_3D b)
{
    if(a == b)
        return Length(p - a);
    Vector_3D v1 = b - a, v2 = p - a, v3 = p - b;
    if(Dot(v1, v2) + EPS < 0)
        return Length(v2);
    else{
        if(Dot(v1, v3) - EPS > 0)
            return Length(v3);
        else
            return Length(Cross(v1, v2)) / Length(v1);
    }
}
```

2.9 求异面直线与的公垂线对应的s(LineDistance3D)

```
//求异面直线 p1+s*u与p2+t*v的公垂线对应的s 如果平行|重合，返回false
bool LineDistance3D(Point_3D p1, Vector_3D u, Point_3D p2, Vector_3D v, double & s)
{
    double b = Dot(u, u) * Dot(v, v) - Dot(u, v) * Dot(u, v);
    if(abs(b) ≤ EPS)
        return false;
    double a = Dot(u, v) * Dot(v, p1 - p2) - Dot(v, v) * Dot(u, p1 - p2);
    s = a / b;
    return true;
}
```

2.10 p1和p2是否在线段a-b的同侧(SameSide)

```
bool SameSide(const Point_3D & p1, const Point_3D & p2, const Point_3D & a, const Point_3D & b){
    return Dot(Cross(b - a, p1 - a), Cross(b - a, p2 - a)) - EPS ≥ 0;
}
```

2.11 判断点P是否在三角形中(PointInTri)

另法详见《训练指南》P288

```
//点P在三角形P0, P1, P2中
bool PointInTri(const Point_3D & P, const Point_3D & P0, const Point_3D & P1, const Point_3D &
P2){
    return SameSide(P, P0, P1, P2) && SameSide(P, P1, P0, P2) && SameSide(P, P2, P0, P1);
}
```

2.12 三角形P0、P1、P2是否和线段AB相交(TriSegIntersection)

```
bool TriSegIntersection(const Point_3D & P0, const Point_3D & P1, const Point_3D & P2, const
Point_3D & A, const Point_3D & B, Point_3D & P)
{
    Vector_3D n = Cross(P1 - P0, P2 - P0);

```

```
    if(abs(Dot(n, B - A)) ≤ EPS)
        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(t + EPS < 0 || t - 1 - EPS > 0)
            return false;    // 不在线段AB上
        P = A + (B - A) * t; // 交点
        return PointInTri(P, P0, P1, P2);
    }
}
```

2.13 空间两三角形是否相交(TriTriIntersection)

```
bool TriTriIntersection(Point_3D * T1, Point_3D * T2)
{
    Point_3D 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;
}
```

2.14 空间两直线上最近点对 返回最近距离(SegSegDistance)

```
//空间两直线上最近点对 返回最近距离 点对保存在ans1 ans2中
double SegSegDistance(Point_3D a1, Point_3D b1, Point_3D a2, Point_3D b2, Point_3D& ans1,
Point_3D& ans2)
{
    Vector_3D v1 = (a1 - b1), v2 = (a2 - b2);
    Vector_3D N = Cross(v1, v2);
    Vector_3D ab = (a1 - a2);
    double ans = Dot(N, ab) / Length(N);
    Point_3D p1 = a1, p2 = a2;
    Vector_3D d1 = b1 - a1, d2 = b2 - a2;
    double t1, t2;
    t1 = Dot((Cross(p2 - p1, d2)), Cross(d1, d2));
    t2 = Dot((Cross(p2 - p1, d1)), Cross(d1, d2));
    double dd = Length((Cross(d1, d2)));
    t1 /= dd * dd;
    t2 /= dd * dd;
    ans1 = (a1 + (b1 - a1) * t1);
    ans2 = (a2 + (b2 - a2) * t2);
    return fabs(ans);
}
```

2.15判断P是否在三角形A, B, C中，并且到三条边的距离都至少为mindist(InsideWithMinDistance)

```
// 判断P是否在三角形A, B, C中，并且到三条边的距离都至少为mindist。保证P, A, B, C共面
bool InsideWithMinDistance(const Point_3D & P, const Point_3D & A, const Point_3D & B, const Point_3D & C, double mindist)
{
    if(!PointInTri(P, A, B, C))
        return false;
    if(DistanceToLine(P, A, B) < mindist)
        return false;
    if(DistanceToLine(P, B, C) < mindist)
        return false;
    if(DistanceToLine(P, C, A) < mindist)
        return false;
    return true;
}
```

2.16判断P是否在凸四边形中，并且到四条边的距离都至少为mindist(InsideWithMinDistance)

```
// 判断P是否在凸四边形ABCD（顺时针或逆时针）中，并且到四条边的距离都至少为mindist。保证P, A, B, C, D共面
bool InsideWithMinDistance(const Point_3D & P, const Point_3D & A, const Point_3D & B, const Point_3D & C, const Point_3D & D, double mindist)
{
    if(!PointInTri(P, A, B, C))
        return false;
    if(!PointInTri(P, C, D, A))
        return false;
    if(DistanceToLine(P, A, B) < mindist)
        return false;
    if(DistanceToLine(P, B, C) < mindist)
        return false;
    if(DistanceToLine(P, C, D) < mindist)
        return false;
    if(DistanceToLine(P, D, A) < mindist)
        return false;
    return true;
}

```

三、三维凸包

3.1 加干扰防止多点共面(add_noise)

```cpp
//加干扰防止多点共面
double rand01()
{
    return rand() / (double)RAND_MAX;
}
double randeps()
{
    return (rand01() - 0.5) * EPS;
}
Point_3D add_noise(const Point_3D & p)
{
    return Point_3D(p.x + randeps(), p.y + randeps(), p.z + randeps());
}
```

3.2 凸包的定义(Face)

```cpp
struct Face
{
    int v[3];
    Face(int a, int b, int c)
    {
        v[0] = a;
        v[1] = b;
        v[2] = c;
    }//逆时针旋转
    Vector_3D Normal(const vector<Point_3D> & 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<Point_3D> & P, int i) const
    {
        return Dot(P[i] - P[v[0]], Normal(P)) > 0;
    }
};
```

3.3 增量法求三维凸包(CH3D)

```cpp
// 增量法求三维凸包
// 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(Point_3D* p, int n)//所有面的点集和点数
vector<Face> CH3D(const vector<Point_3D> & 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;
}

```

3.4 凸多面体(ConvexPolyhedron)

```
struct ConvexPolyhedron
{
    int n;
    vector<Point_3D> P, P2;
    vector<Face> faces;

    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_Point_3D();
            P2[i] = add_noise(P[i]);
        }

        faces = CH3D(P2);
        return true;
    }

    //三维凸包重心
    Point_3D centroid()
    {
        Point_3D C = P[0];
        double totv = 0;
        Point_3D tot(0, 0, 0);

        for(int i = 0; i < faces.size(); i++)
        {
            Point_3D p1 = P[faces[i].v[0]], p2 = P[faces[i].v[1]], p3 = P[faces[i].v[2]];
            double v = -Volume6(p1, p2, p3, C);
            totv += v;
            tot = tot + Centroid(p1, p2, p3, C) * v;
        }

        return tot / totv;
    }
    //凸包重心到表面最近距离
    double mindist(Point_3D C)
    {
        double ans = 1e30;

        for(int i = 0; i < faces.size(); i++)
        {
            Point_3D p1 = P[faces[i].v[0]], p2 = P[faces[i].v[1]], p3 = P[faces[i].v[2]];
            ans = min(ans, fabs(-Volume6(p1, p2, p3, C) / Area2(p1, p2, p3)));
        }

        return ans;
    }
};
```

3.5 给三维凸包求出重心到各面的最小距离

给我们一个三维凸包，让我们求出重心到各个面的最小距离。

```
const int MAXN=550;
const double eps=1e-8;
struct Point
{
    double x,y,z;
    Point(){}

    Point(double xx,double yy,double zz):x(xx),y(yy),z(zz){}

```

```
    //两向量之差
    Point operator -(const Point p1)
    {
        return Point(x-p1.x,y-p1.y,z-p1.z);
    }

    //两向量之和
    Point operator +(const Point p1)
    {
        return Point(x+p1.x,y+p1.y,z+p1.z);
    }

    //叉乘
    Point operator *(const Point p)
    {
        return Point(y*p.z-z*p.y,z*p.x-x*p.z,x*p.y-y*p.x);
    }

    Point operator *(double d)
    {
        return Point(x*d,y*d,z*d);
    }

    Point operator / (double d)
    {
        return Point(x/d,y/d,z/d);
    }

    //点乘
    double  operator ^(Point p)
    {
        return (x*p.x+y*p.y+z*p.z);
    }
};

struct CH3D
{
    struct face
    {
        //表示凸包一个面上的三个点的编号
        int a,b,c;
        //表示该面是否属于最终凸包上的面
        bool ok;
    };
    //初始顶点数
    int n;
    //初始顶点
    Point P[MAXN];
    //凸包表面的三角形数
    int num;
    //凸包表面的三角形
    face F[8*MAXN];
    //凸包表面的三角形
    int g[MAXN][MAXN];
    //向量长度
```

```
65      double vlen(Point a)
66      {
67          return sqrt(a.x*a.x+a.y*a.y+a.z*a.z);
68      }
69      //叉乘
70      Point cross(const Point &a,const Point &b,const Point &c)
71      {
72          return Point((b.y-a.y)*(c.z-a.z)-(b.z-a.z)*(c.y-a.y),
73                       (b.z-a.z)*(c.x-a.x)-(b.x-a.x)*(c.z-a.z),
74                       (b.x-a.x)*(c.y-a.y)-(b.y-a.y)*(c.x-a.x)
75                       );
76      }
77      //三角形面积*2
78      double area(Point a,Point b,Point c)
79      {
80          return vlen((b-a)*(c-a));
81      }
82      //四面体有向体积*6
83      double volume(Point a,Point b,Point c,Point d)
84      {
85          return (b-a)*(c-a)^(d-a);
86      }
87      //正：点在面同向
88      double dblcmp(Point &p,face &f)
89      {
90          Point m=P[f.b]-P[f.a];
91          Point n=P[f.c]-P[f.a];
92          Point t=p-P[f.a];
93          return (m*n)^t;
94      }
95      void deal(int p,int a,int b)
96      {
97          int f=g[a][b];//搜索与该边相邻的另一个平面
98          face add;
99          if(F[f].ok)
100         {
101             if(dblcmp(P[p],F[f])>eps)
102                 dfs(p,f);
103             else
104             {
105                 add.a=b;
106                 add.b=a;
107                 add.c=p;//这里注意顺序，要成右手系
108                 add.ok=true;
109                 g[p][b]=g[a][p]=g[b][a]=num;
110                 F[num++]=add;
111             }
112         }
113     }
114     void dfs(int p,int now)//递归搜索所有应该从凸包内删除的面
115     {
116          F[now].ok=0;
117          deal(p,F[now].b,F[now].a);
118          deal(p,F[now].c,F[now].b);
119          deal(p,F[now].a,F[now].c);
```

```
    }
    bool same(int s,int t)
    {
        Point &a=P[F[s].a];
        Point &b=P[F[s].b];
        Point &c=P[F[s].c];
        return fabs(volume(a,b,c,P[F[t].a]))<eps &&
               fabs(volume(a,b,c,P[F[t].b]))<eps &&
               fabs(volume(a,b,c,P[F[t].c]))<eps;
    }
    //构建三维凸包
    void create()
    {
        int i,j,tmp;
        face add;

        num=0;
        if(n<4)return;
    //********************************************
        //此段是为了保证前四个点不共面
        bool flag=true;
        for(i=1;i<n;i++)
        {
            if(vlen(P[0]-P[i])>eps)
            {
                swap(P[1],P[i]);
                flag=false;
                break;
            }
        }
        if(flag)return;
        flag=true;
        //使前三个点不共线
        for(i=2;i<n;i++)
        {
            if(vlen((P[0]-P[1])*(P[1]-P[i]))>eps)
            {
                swap(P[2],P[i]);
                flag=false;
                break;
            }
        }
        if(flag)return;
        flag=true;
        //使前四个点不共面
        for(int i=3;i<n;i++)
        {
            if(fabs((P[0]-P[1])*(P[1]-P[2])^(P[0]-P[i]))>eps)
            {
                swap(P[3],P[i]);
                flag=false;
                break;
            }
        }
        if(flag)return;
```

```
175     //*****************************************
176         for(i=0;i<4;i++)
177         {
178             add.a=(i+1)%4;
179             add.b=(i+2)%4;
180             add.c=(i+3)%4;
181             add.ok=true;
182             if(dblcmp(P[i],add)>0)swap(add.b,add.c);
183             g[add.a][add.b]=g[add.b][add.c]=g[add.c][add.a]=num;
184             F[num++]=add;
185         }
186         for(i=4;i<n;i++)
187         {
188             for(j=0;j<num;j++)
189             {
190                 if(F[j].ok&&dblcmp(P[i],F[j])>eps)
191                 {
192                     dfs(i,j);
193                     break;
194                 }
195             }
196         }
197         tmp=num;
198         for(i=num=0;i<tmp;i++)
199           if(F[i].ok)
200             F[num++]=F[i];
201 
202     }
203     //表面积
204     double area()
205     {
206         double res=0;
207         if(n==3)
208         {
209             Point p=cross(P[0],P[1],P[2]);
210             res=vlen(p)/2.0;
211             return res;
212         }
213         for(int i=0;i<num;i++)
214           res+=area(P[F[i].a],P[F[i].b],P[F[i].c]);
215         return res/2.0;
216     }
217     double volume()
218     {
219         double res=0;
220         Point tmp(0,0,0);
221         for(int i=0;i<num;i++)
222            res+=volume(tmp,P[F[i].a],P[F[i].b],P[F[i].c]);
223         return fabs(res/6.0);
224     }
225     //表面三角形个数
226     int triangle()
227     {
228         return num;
229     }
```

```
    //表面多边形个数
    int polygon()
    {
        int i,j,res,flag;
        for(i=res=0;i<num;i++)
        {
            flag=1;
            for(j=0;j<i;j++)
              if(same(i,j))
              {
                  flag=0;
                  break;
              }
            res+=flag;
        }
        return res;
    }
    //三维凸包重心
    Point barycenter()
    {
        Point ans(0,0,0),o(0,0,0);
        double all=0;
        for(int i=0;i<num;i++)
        {
            double vol=volume(o,P[F[i].a],P[F[i].b],P[F[i].c]);
            ans=ans+(o+P[F[i].a]+P[F[i].b]+P[F[i].c])/4.0*vol;
            all+=vol;
        }
        ans=ans/all;
        return ans;
    }
    //点到面的距离
    double ptoface(Point p,int i)
    {
        return fabs(volume(P[F[i].a],P[F[i].b],P[F[i].c],p)/vlen((P[F[i].b]-P[F[i].a])*
(P[F[i].c]-P[F[i].a])));
    }
};
CH3D hull;
int main()
{
    while(scanf("%d",&hull.n)==1)
    {
        for(int i=0;i<hull.n;i++)
        {
            scanf("%lf%lf%lf",&hull.P[i].x,&hull.P[i].y,&hull.P[i].z);
        }
        hull.create();
        Point p=hull.barycenter();//求重心
        double ans1=1e20;
        for(int i=0;i<hull.num;i++)
        {
            ans1=min(ans1,hull.ptoface(p,i));
        }
        printf("%.3f\n",ans1);
```

```
284         }
285         return 0;
286     }
```