计算几何在算法竞赛与生活中的应用

尹玉文东，蔡越同，李灏冬，张钰晨

【摘要】方法 使用 C++语言编写计算几何工具库，利用其研究凸包和半平面交问题的解法。之后借助洛谷网等算法竞赛[1]相关资源网站，总结这两类问题的应用场景。 **结果** 介绍了 Andrew 算法和 Sort-and-Incremental 算法，简述了两种算法的思路与时间复杂度。通过对算法竞赛中相关典例的分析，指出了两类问题在算法竞赛和生活中的应用。

【关键词】计算几何；凸包；半平面交；旋转卡壳

Computational Geometry in Algorithm Competition and Life

[Abstract] Methods Using C++ language to program computational geometry tool library, using it to study convex hull and half-plane intersection problem. Then, with the help of resource websites related to algorithm competitions such as Luogu.com, the application scenarios of these two types of problems are summarized. **Results** The Andrew algorithm and the Sort-and-Incremental algorithm are introduced, and the ideas and time complexity of the two algorithms are briefly described. Through the analysis of the relevant examples in algorithm competition, the application of the two types of problems in algorithm competition and life is pointed out.

[Key words] computational geometry; convex Hull; half-plane intersection; rotating calipers

[1] 洛谷网：https://www.luogu.com.cn；其它资源网站包括：http://poj.org，https://www.spoj.com.

1. 引言

随着信息技术的大范围普及，算法竞赛的影响力与日俱增。计算几何在算法竞赛中有着重要应用，凸包和半平面交是计算几何中的经典问题。

计算几何是基于向量运算的体系。相较于解析几何，计算几何能有效控制计算机运算中的浮点数精度问题，更适用于算法竞赛。

作为算法竞赛中较为重要的模块，相关资料整理却不甚完全。我组决定借此机会整合资料，并对此类算法和问题进行学习研究，便于后续使用。

2. 计算几何库及有关说明

本课题组使用 C++语言编写了计算几何工具库，并利用其研究了这两类问题的解法，本文出现的代码可能直接调用库中函数。

计算几何库的完整代码，详见本文的 **9. 附件**部分。

2.1. 浮点数相关函数

为了避免浮点数运算造成的精度问题，编写了相关函数用于控制精度。

包含功能：判断一个数的正负、比较两个数的大小。

2.2. 平面向量类

平面向量类用于解决二维向量的相关运算问题。

2.2.1. 存储

```
struct Point{
        double x, y;
        Point(){};
        Point(double a, double b): x(a), y(b) {}
        Point(Point a, Point b): x(b.x - a.x), y(b.y - a.y) {}
};
```

2.2.2. 线性运算

支持加法、减法、数乘、点乘等操作。

以加法为例，通过 friend 友元函数实现重载运算符。

```
friend Point operator + (const Point &a, const Point &b) {
        return Point(a.x + b.x, a.y + b.y);
}
```

2.2.3. 叉乘

平面向量的叉乘，只求模长及方向。

根据右手法则可以推断出 **b** 相对于 **a** 的方向，逆时针方向竖坐标为正值，反之为负值。

2.2.4. 其它功能

平面向量类还可以：求向量的模长；求两向量之间的夹角；对一组向量按照字典序（坐标序）或极角排序。

其中，极角排序的原理如下。

取笛卡尔坐标系中的 x 轴作为极轴，以逆时针方向为正。

利用 C++语言中的 std::atan2(double y, double x) 进行极角排序，函数的返回值为向量 (x, y) 与 x 轴的极角 $\theta \in (-\pi, \pi]$.

2.3. 直线类

2.3.1. 存储

对于直线上任意两点 $A(x_a,y_a)$，$B(x_b,y_b)$，构造坐标对应的向量 $\vec{s}=(x_a,y_a)$，$\vec{t}=(x_b,y_b)$。

直线类利用向量类，记录这两个向量 $\vec{s}$ 和 $\vec{t}$。

```
struct Line{
        Point s, t;
        Line() {};
        Line(Point a, Point b) : s(a), t(b) {}
};
```

2.3.2. 求两直线交点

首先判断直线是否平行，若平行则不存在交点。

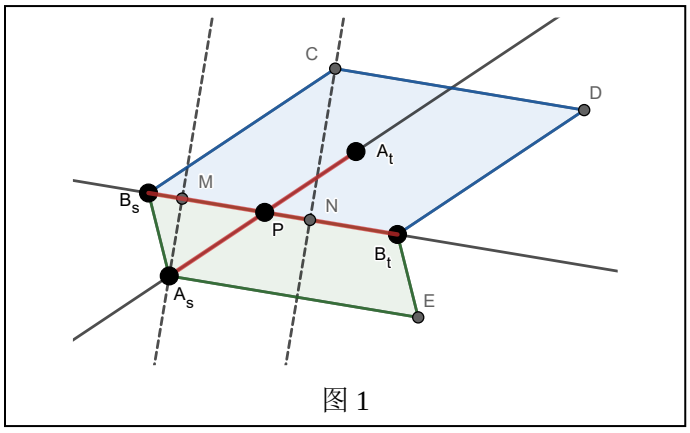

图 1

否则，如**图 1**，直线 A 为 (A_s,A_t)，直线 B 为 (B_s,B_t)，求 A 与 B 的交点。

函数只需求出 $t=\frac{|A_sP|}{|A_sA_t|}$，则交点 $\vec{P}=\overrightarrow{A_s}+t\cdot\overrightarrow{A_sA_t}$，证明如下。

注意到 $t=\frac{|A_sP|}{|A_sA_t|}=\frac{|\overrightarrow{B_sB_t}\times\overrightarrow{B_sA_s}|}{|\overrightarrow{B_sB_t}\times\overrightarrow{A_sA_t}|}$,

把 $\overrightarrow{A_sA_t}$ 平移到 $\overrightarrow{B_sC}$ 的位置，则可以把叉乘的模看作平行四边形面积。

记 $S_1=\left|\overrightarrow{B_sB_t}\times\overrightarrow{B_sA_s}\right|$，$S_2=\left|\overrightarrow{B_sB_t}\times\overrightarrow{A_sA_t}\right|$，因为两个平行四边形同底，所以有 $\frac{S_1}{S_2}=\frac{|A_sM|}{|CN|}$.

又因为 $\triangle A_sMP\sim\triangle CNB_s$，所以 $\frac{S_1}{S_2}=\frac{|A_sM|}{|CN|}=\frac{|A_sP|}{|CB_s|}=\frac{|A_sP|}{|A_sA_t|}$.

代码实现详见 **9. 附件**

2.4. 三维向量类

2.4.1. 存储

用结构体同时记录三维坐标，方便重载运算符。

```cpp
struct Point3 {
    double x, y, z;
    Point3(){};
    Point3(double a, double b, double c) : x(a), y(b), z(c) {}
};
```

2.4.2. 基本运算

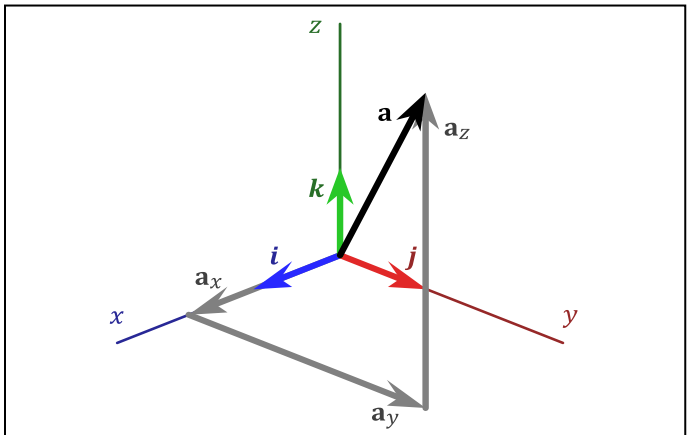

图 2

支持加法、减法和点乘等操作，代码实现与二维向量类似。

2.4.3. 叉乘

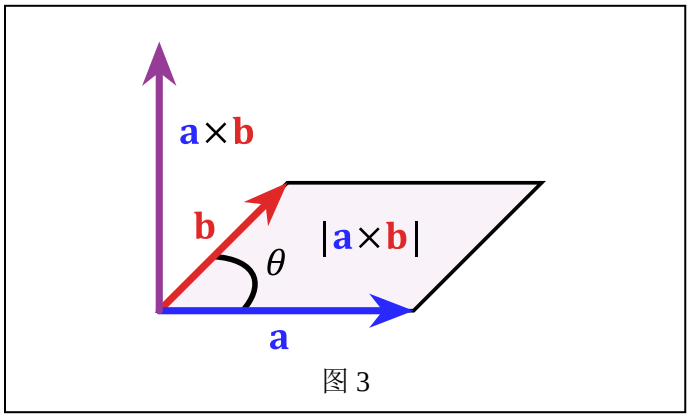

图 3

如**图 3**，$\mathbf{a}\times\mathbf{b}$ 的结果为一个三维向量 $\mathbf{c}$，$\mathbf{c}\perp\mathbf{a}$ 且 $\mathbf{c}\perp\mathbf{b}$，结果向量的模长为 $|\mathbf{a}||\mathbf{b}|\sin\langle\mathbf{a},\mathbf{b}\rangle$，代表以 $\mathbf{a}$、$\mathbf{b}$ 为两边的平行四边形的面积。

在三维向量类中，我们需要用坐标表示结果向量 $\mathbf{a}$，推导过程如下。（引用、同济大学高等数学）

右手坐标系中，基向量 $\mathbf{i},\mathbf{j},\mathbf{k}$ 满足以下等式：

$$\boldsymbol{i}\times\boldsymbol{j}=\boldsymbol{k}$$
$$\boldsymbol{j}\times\boldsymbol{i}=-\boldsymbol{k}$$
$$\boldsymbol{j}\times\boldsymbol{k}=\boldsymbol{i}$$
$$\boldsymbol{k}\times\boldsymbol{j}=-\boldsymbol{i}$$
$$\boldsymbol{k}\times\boldsymbol{i}=\boldsymbol{j}$$
$$\boldsymbol{i}\times\boldsymbol{k}=-\boldsymbol{j}$$

根据外积的定义可以得出：$\boldsymbol{i}\times\boldsymbol{i}=\boldsymbol{j}\times\boldsymbol{j}=\boldsymbol{k}\times\boldsymbol{k}=0$.

根据以上等式，结合外积的分配律，就可以确定任意向量的外积。

任取向量 $\boldsymbol{u}=u_1\boldsymbol{i}+u_2\boldsymbol{j}+u_3\boldsymbol{k}$ 和 $\boldsymbol{v}=v_1\boldsymbol{i}+v_2\boldsymbol{j}+v_3\boldsymbol{k}$，两者的外积 $\boldsymbol{u}\times\boldsymbol{v}$ 可以根据分配率展开：

$$\begin{aligned}\boldsymbol{u}\times\boldsymbol{v}&=(u_1\boldsymbol{i}+u_2\boldsymbol{j}+u_3\boldsymbol{k})\times(v_1\boldsymbol{i}+v_2\boldsymbol{j}+v_3\boldsymbol{k})\\&=u_1v_1(\boldsymbol{i}\times\boldsymbol{i})+u_1v_2(\boldsymbol{i}\times\boldsymbol{j})+u_1v_3(\boldsymbol{i}\times\boldsymbol{k})+\\&\quad u_2v_1(\boldsymbol{j}\times\boldsymbol{i})+u_2v_2(\boldsymbol{j}\times\boldsymbol{j})+u_2v_3(\boldsymbol{j}\times\boldsymbol{k})+\\&\quad u_3v_1(\boldsymbol{k}\times\boldsymbol{i})+u_3v_2(\boldsymbol{k}\times\boldsymbol{j})+u_3v_3(\boldsymbol{k}\times\boldsymbol{k})\end{aligned}$$

把前面的 6 个等式代入，则有：

$$\begin{aligned}\boldsymbol{u}\times\boldsymbol{v}&=-u_1v_1\boldsymbol{0}+u_1v_2\boldsymbol{k}-u_1v_3\boldsymbol{j}\\&\quad-u_2v_1\boldsymbol{k}-u_2v_2\boldsymbol{0}+u_2v_3\boldsymbol{i}\\&\quad+u_3v_1\boldsymbol{j}-u_3v_2\boldsymbol{i}-u_3v_3\boldsymbol{0}\\&=(u_2v_3-u_3v_2)\boldsymbol{i}+(u_3v_1-u_1v_3)\boldsymbol{j}+(u_1v_2-u_2v_1)\boldsymbol{k}\end{aligned}$$

因此结果向量 $\boldsymbol{s}=\boldsymbol{u}\times\boldsymbol{v}=s_1\boldsymbol{i}+s_2\boldsymbol{j}+s_3\boldsymbol{k}$ 的三维坐标为：

$$\begin{aligned}s_1&=u_2v_3-u_3v_2\\s_2&=u_3v_1-u_1v_3\\s_3&=u_1v_2-u_2v_1\end{aligned}$$

代码实现详见 **9. 附件**

2.5. 平面类

用三个向量表示一个三角形的平面。一个多面体可以通过三角剖分，用若干个三角形表示。

为了减少程序的空间占用，用 point3 p[N] 数组存储所有可能出现的向量，结构体 plane 只记录向量在 p 数组中的下标。

记录的三个向量按逆时针首尾相接，这样在判断方向时比较方便。

```
struct plane{
	int v[3];
	plane(){};
	plane(int a, int b, int c) {
		v[0] = a, v[1] = b, v[2] = c;
	}
};
```

平面的法向量：是指垂直于该平面的三维向量。一个平面具有无限个法向量，这些法向量可以按方向分为两类。

根据叉积的性质，将三角形的两条邻边叉乘，得到的向量即为一条法向量。

利用法向量的模长，也可以算出三角形的面积。

代码实现详见 **9. 附件**

3. 平面凸包

3.1. 凸包

一个子集 S 被称为凸的，当且仅当对于任意两点 $p,q\in S$，线段 $\overline{pq}$ 都完全属于 S。

集合 S 的凸包 $\mathcal{CH}(S)$，是包含 S 的最小凸集，也就是包含 S 的所有凸集的交。

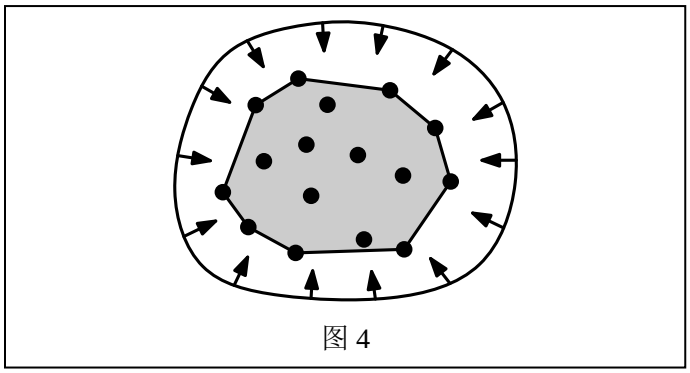

图 4

如**图 4**，不严谨地表示，平面凸包是平面上能包含所有给定点的最小凸多边形。

3.2. 效率较低的求解算法

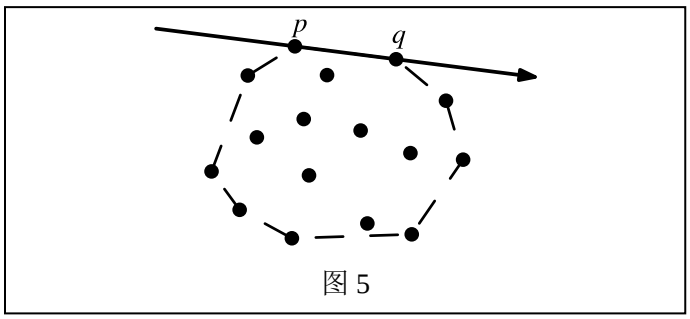

图 5

如**图 5**，任取一条线段 $\overline{pq}$，若其在凸包上，则点集 P 中的点均位于直线 $\overline{pq}$ 的同一侧。

若我们钦定 $p \to q$ 按顺时针方向，则有更强的限制，需要 P 中的点都在直线的右侧。

考虑向量的叉积，点 t 在 $\overrightarrow{pq}$ 右侧 $\Leftrightarrow \overrightarrow{pt} \times \overrightarrow{pq} > 0$。于是可以枚举有序点对 $(p,q) \in P \times P$，若 P 中的点都在有向线段 $\overrightarrow{pq}$ 的右侧，则 $\overline{pq}$ 是 $\mathcal{CH}(P)$ 中的一条边。

此算法时间复杂度为 $\mathcal{O}(n^3)$。其中 n 为点数。

3.3. Andrew 算法

3.3.1. 算法流程

首先把所有点排序，以横坐标为第一关键字，纵坐标为第二关键字。排序后，第一个点和末尾的点，一定在凸包上，容易通过反证法证明。

从左往右看，上下凸壳斜率的 单调性相反，即所旋转的方向不同，所以要分开求解。升序枚举求出下凸壳，然后 降序枚举求出上凸壳，这样凸包的每条边都是向逆时针方向旋转的。

设当前枚举到点 P，即将把其加入凸包；当前栈顶的点为 S_1，栈中第二个点为 S_2。

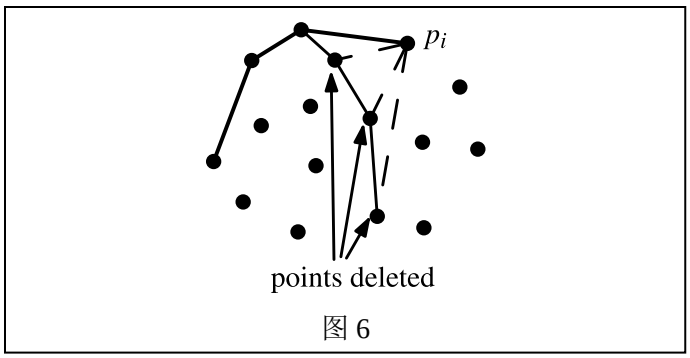

图 6

如**图 6**，求凸包时，若 P 与 S_1 构成的新线段是顺时针旋转的，即叉积满足：$\overrightarrow{S_2S_1} \times \overrightarrow{S_1P} < 0$，则弹出栈顶，继续检查，直到 $\overrightarrow{S_2S_1} \times \overrightarrow{S_1P} \geq 0$ 或者栈内仅剩一个元素为止。

记 $n = |P|$，则时间复杂度为 $\Theta(n\log n)$，瓶颈在排序部分。

相较于基础算法，Andrew 算法的效率大幅提升。

3. 3. 2. 参考代码

Convex_hull_2d 是求平面凸包的函数，需要传入三个参数。

下面的代码中，int n 代表点集大小， Point p[]是传入的点集， Point ret[]是用于记录凸包的点集。

```
inline bool check(Point s1, Point s2, Point p) {
	return Vec(s2, s1) * Vec(s1, p) > 0;
}
int Convex_hull_2d(int n, Point *p, Point *ret) {
	sort(p, p + n, cmp1);
	int top = -1;
	for (int i = 0; i < n; i++) {
		while (top > 0 && !check(ret[top], ret[top - 1], p[i]))
			top--;
		ret[++top] = p[i];
	}
	int k = top;
	for (int i = n - 2; i >= 0; i--) {
		while (top > k && !check(ret[top], ret[top - 1], p[i]))
			top--;
		ret[++top] = p[i];
	}
	return top;
}
```

3. 4. 凸包在算法竞赛中的应用

3. 4. 1. 求覆盖点集的最小凸多边形周长

此类问题即可转化为求凸包周长。

先求出按照顺时针排序的，构成凸包的点集 p，记 $n = |p|$。再把相邻两点组成的向量的模长求和，即：

$$l = \sum_{i=1}^{n} |\overrightarrow{p_i p_{i+1}}| + |\overrightarrow{p_1 p_n}|$$

代码实现见 **9. 附件**中的 double Convex_hull_2d_L 函数。

3.4.2. 求覆盖点集的最小凸边形的面积

此类问题即可转化为求凸包面积。

先求出按照顺时针排序的，构成凸包的点集 p，记 $n=|p|$。任取凸包内一点（一般取 p_1），则有：

$$s=\sum_{i=2}^{n-1}\text{area}(p_1,p_i,p_i+1)=\sum_{i=2}^{n-1}\frac{|(p_i-p_1)\times(p_{i+1}-p_1)|}{2}$$

代码实现见 **9. 附件**中的 double Convex_hull_2d_S 函数。

3.5. 凸包在生活中的应用

3.5.1. 包围盒算法

其广泛应用于与游戏物理引擎的制作中。

游戏过程中，游戏中的元素有时会发生碰撞，且这些元素的形状通常不规则。如果直接对不规则元素进行彭专检测会使得游戏效率低下，由此产生了一个近似算法，即包围盒算法。

其中包围盒指能完整覆盖元素的矩形。在游戏应用中为了使图形拟合更准确以及更简单地对游戏元素进行操作，通常会将以最小包围盒代替游戏中的不规则元素。

此问题即可转换为凸包问题进行求解。

3.5.2. 碰撞检测及避免算法

其广泛应用与 3D 游戏中。这可以使人物在场景中可以平滑移动、遇到一定高度的障碍物可以越过而遇到过高的障碍物则停下、在各种前进方向被挡住的情况下都要尽可能地让人物沿合理的方向滑动而不是被迫停下。在满足这些要求的同时还可以做到足够精确和稳定。

在这个算法中，多维凸包可以用于找出人物轮廓，并简化后续计算及判断过程。

4. 三维凸包

4.1. 三维凸包及性质

三维凸包的定义与二维凸包类似。

点集 P 的三维凸包，是由若干个点集内的点构成的凸多面体，任意两点构成的线段均在这个多面体内部。

4.2. S&I 算法随机增量法

S&I 算法，全称 Sort and Incremental 算法，是一种随机增量法。

4.2.1. 算法思想

用 p_i 表示点集中的第 i 个点，点集 $P_i=\{p_1,p_2,\cdots,p_i\}$，用 $\mathcal{CH}(A)$ 表示点集 A 的凸包。

和 Andrew 算法类似，考虑对每个点进行一次增量过程，每次把第 r 个点 p_r 加入到前 $r-1$ 个点的凸包中，并更新凸包。也就是说，通过某种操作，将 $\mathcal{CH}(P_{r-1})$ 转化为凸包 $\mathcal{CH}(P_r)$.

情况一：p_r 在 $\mathcal{CH}(P_{r-1})$ 的内部或边界上，则凸包不变，$\mathcal{CH}(P_{r-1})\to\mathcal{CH}(P_r)$.

情况二：p_r 在 $\mathcal{CH}(P_{r-1})$ 外部。设想你站在 p_r 所在的位置，看向 $\mathcal{CH}(P_{r-1})$，$\mathcal{CH}(P_{r-1})$ 中的某些小平面会被看到，其余在背面的平面不会被看到。如**图 7**，从 p_r 可见的平面构成了一片连通的区域。

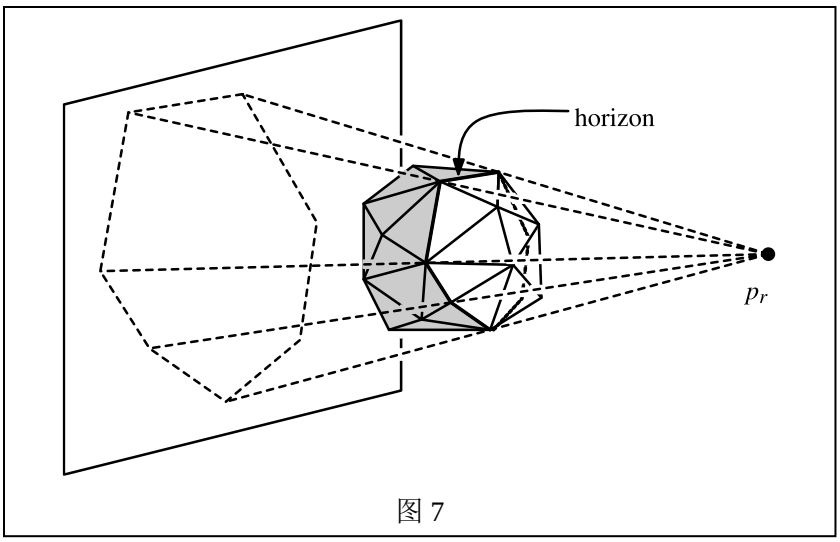

图 7

这片区域由一条封闭折线围成，称这条线为 p_r 在 $\mathcal{CH}(P_{r-1})$ 上的**边界线**(horizon)。

根据这条边界线，我们可以判断出，在原先 $\mathcal{CH}(P_{r-1})$ 表面上的哪些部分需要被保留，哪些需要被替换。

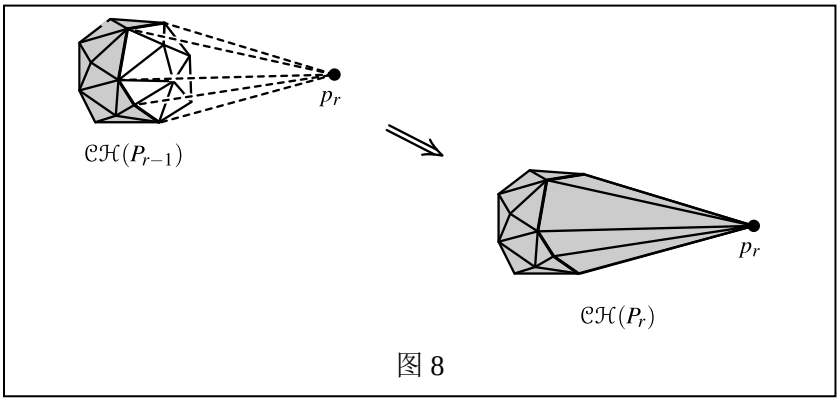

图 8

显然，不可见的平面在 $\mathcal{CH}(P_r)$ 中被保留，并且我们用 p_r 与边界线地平线之间连接出新的小平面，来替换所有可见的小平面，如**图 8**。

4.2.2. 判断平面对点的可见性

考虑如何用几何语言表达：“一个平面对 p_r 是可见的”。

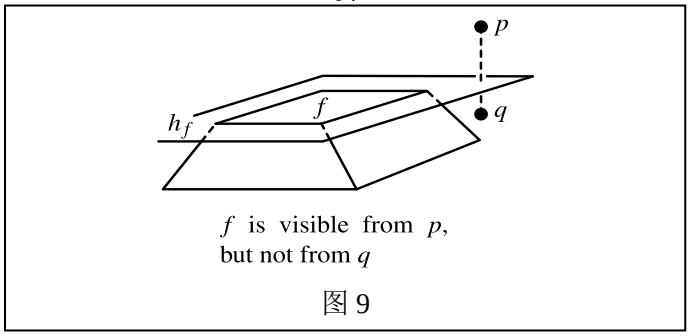

图 9

对于一个凸包上的小平面，它能将空间分为两半，一侧是凸包外部，一侧是凸包内部。容易发现，如果点 p 位于小平面的外侧，那么这个平面对于 p 点就是可见的，因为凸包上的其它平面都不会有遮挡。

形式化地，记 a,b,c 为平面三角形的三个顶点，从凸包外部看，三点按照逆时针排列。

利用叉乘的性质，记 $\mathbf{s}=\overrightarrow{ab}\times\overrightarrow{ac}$，则结果向量 $\mathbf{s}$ 是一个平面的法向量，且指向凸包外部。对于空间内任意一点 p，若 $\overrightarrow{ap}\times\mathbf{s}>0$，则这个平面对点 p 是可见的。

函数 bool is_above(Point3 A)定义在结构体 plane 中，用于判断点 A 是否位于平面的外侧。

4.2.3. 求出边界线

要想把凸包从 $\mathcal{CH}(P_{r-1})$ 转化为 $\mathcal{CH}(P_r)$，我们需要准确地求出凸包上的哪些边在边界线上。求出边界线之后，才能用 p 与边界线构成的小平面替换需要被删掉的小平面。

定义 bool g[N][N]，$g[i][j]$ 表示 $\overrightarrow{p_ip_j}$ 所在的平面是否可见。

若规定平面 (a,b,c) 只包含 $\overrightarrow{ab},\overrightarrow{bc},\overrightarrow{ca}$，则对于任意有序数对 (i,j)，向量 $\overrightarrow{p_ip_j}$ 最多被包含在一个平面内。

注意到，位于边界线上的向量 $\overrightarrow{p_ip_j}$ 一定满足 $g[i][j]=1$ 且 $g[j][i]=0$。所以我们只需对每个平面判断其可见性，并更新在 g 数组中对应的数值，即可求出边界线。

4.2.4. 利用边界线更新凸包

在上一步遍历小平面时，若遇到的小平面是不可见的，则把它加入新的凸包中；若可见，则单独记录。

之后遍历所有可见的小平面，若 $\overrightarrow{p_ip_j}$ 在边界线上，则把 (p_i,p_j,p_r) 加入凸包中。p_r 是新加入凸包的点。这样加入后的三点也满足逆时针排列。

4.2.5. 参考代码

函数返回值为三维凸包的平面数，plane ret[]的下标从 0 开始。

```
int Convex_hull_3d(int n, plane *ret) {
	plane tmp[N];
	bool g[N][N];
	for (int i = 0; i < n; i++) p[i].shake();
	int top = -1;
	ret[++top] = plane(0, 1, 2);
	ret[++top] = plane(0, 2, 1);
	for (int i = 3; i < n; i++) {
		int cnt = -1;
		for (int j = 0; j <= top; j++) {
			bool flag = ret[j].is_above(p[i]);
			if (!flag)
				tmp[++cnt] = ret[j];
			for (int k = 0; k < 3; k++)
				g[ret[j].v[k]][ret[j].v[(k + 1) % 3]] = flag;
		}
		for (int j = 0; j <= top; j++) {
			for (int k = 0; k < 3; k++) {
				int a = ret[j].v[k], b = ret[j].v[(k + 1) % 3];
				if (g[a][b] && !g[b][a])
					tmp[++cnt] = plane(a, b, i);
			}
		}
		for (int j = 0; j <= cnt; j++) ret[j] = tmp[j];
		top = cnt;
	}
	return (top + 1);
}
```

5. 旋转卡壳

5.1. 概述

旋转卡壳算法用于：在线性时间内，求凸包直径、最小矩形覆盖等与凸包性质相关的问题。求出凸包之后的算法时间复杂度是线性的。

5.2. 凸包直径

5.2.1. 问题描述

给定平面上的 n 个点，求所有点对之间的最长距离。

需要解决的数据范围满足：$2 \le n \le 50000, 0 \le |x|, |y| \le 10^4$.

5.2.2. 问题解法

可以遍历凸包上的边，对每条边 (a, b)，去找距离这条边最远的点 p。对于 p 点，距离它最远的点，一定是 a, b 中的一个。

我们发现，若逆时针遍历凸包上的边，那么随着边的转动，对应的最远点也在逆时针旋转。

因此，我们可以在逆时针枚举边的同时，实时维护最远点，利用单调性，复杂度为 $O(n)$.

下面的**图 10** 是旋转卡壳算法的示意流程图。

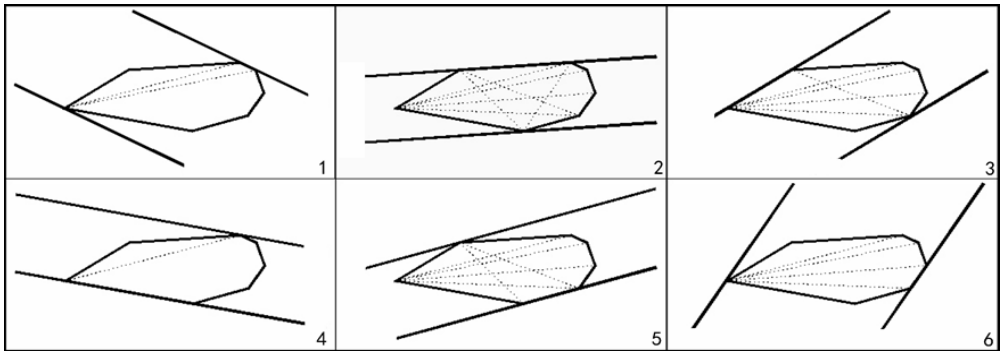

图 10

5.2.3. 参考代码

求凸包时，若使用 Andrew 算法，则凸包上的点已经按照逆时针排序了。问题在如何判断下一个点到当前边的距离是否更大。

一种方法是用点到直线距离公式，但利用叉积相关运算的精度更高，也更方便。

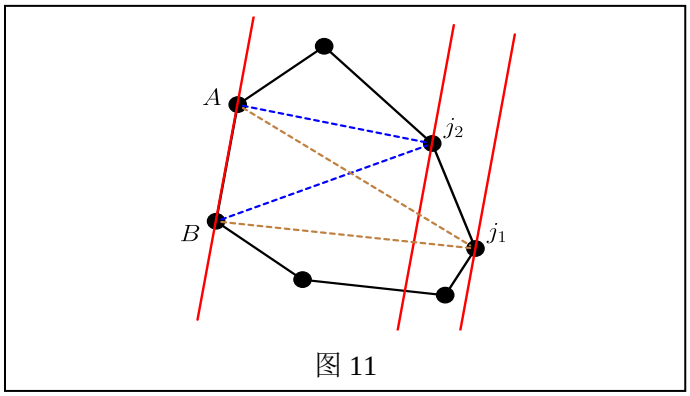

图 11

如**图 11**，$|\overrightarrow{AB} \times \overrightarrow{Bj_1}| > |\overrightarrow{AB} \times \overrightarrow{Bj_2}|$，因此 j_1 到直线 AB 的距离更远，代码如下。

```
inline int sqr_dis(Point a, Point b) { return (a - b).sqr_len(); }
int Get_Max(int n, Point *ch) {//传入 convex-hull
        int ret = 0;
        ch[n] = ch[0];
        int j = 1;
        for(int i = 0; i < n; i++) {
                while((ch[i] - ch[j+1]) * (ch[i+1] - ch[j+1]) >
                                (ch[i] - ch[j]) * (ch[i+1] - ch[j]))
                        j = (j + 1) % n;
                ret = max(ret, max(sqr_dis(ch[i], ch[j]), sqr_dis(ch[i+1], ch[j])));
        }
        return ret;
}
```

5.3. 最小矩形覆盖

5.3.1. 问题描述

给定 n 个点的坐标，求能够覆盖所有点的最小面积的矩形。

求出矩形面积、顶点坐标。

需要解决的数据范围满足：$3 \le n \le 50000$.

5.3.2. 问题解法

先求出凸包，之后用旋转卡壳维护三个边界点。
利用叉积和点积，可以求出矩形面积及四个顶点的坐标。

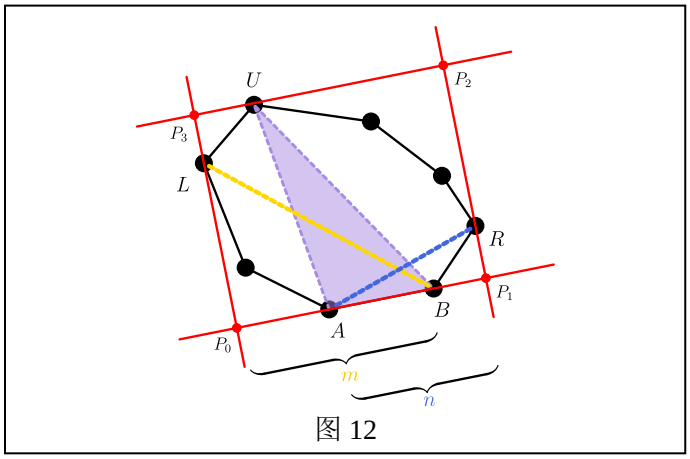

图 12

见**图 12**，下面是一种可行的表示方法，表示方法不唯一。
设 $H = |P_0P_3| = \frac{|\overrightarrow{AB} \times \overrightarrow{BU}|}{|AB|}$，$L = \frac{|\overrightarrow{BA} \cdot \overrightarrow{AL}|}{|BA|}$，$R = \frac{|\overrightarrow{AB} \cdot \overrightarrow{BR}|}{|AB|}$.
则矩形面积为：

$$S = H \times (L + |\overrightarrow{AB}| + R)$$

顶点坐标为：

$$\begin{aligned}
\overrightarrow{P_0} &= \overrightarrow{A} + L \times \frac{\overrightarrow{BA}}{|\overrightarrow{BA}|} \\
\overrightarrow{P_1} &= \overrightarrow{B} + R \times \frac{\overrightarrow{AB}}{|\overrightarrow{AB}|} \\
\overrightarrow{P_2} &= \overrightarrow{P_1} + H \times \frac{\overrightarrow{P_1R}}{|\overrightarrow{P_1R}|} \\
\overrightarrow{P_3} &= \overrightarrow{P_0} + H \times \frac{\overrightarrow{P_0L}}{|\overrightarrow{P_0L}|}
\end{aligned}$$

若不需要求出顶点坐标，也可以这样表示矩形面积：

$$S = \frac{|\overrightarrow{AB} \times \overrightarrow{BU}| \times (|\overrightarrow{AD} \cdot \overrightarrow{AB}| + |\overrightarrow{BC} \cdot \overrightarrow{BA}| - |\overrightarrow{AB} \cdot \overrightarrow{BA}|)}{|\overrightarrow{AB} \cdot \overrightarrow{BA}|}$$

设 $|\overrightarrow{P_0B}| = m$，$|\overrightarrow{AP_1}| = n$ 后易证。

5.3.3. 参考代码

代码实现见 **9. 附件**中的 double Get_Min 函数。

5.4. 点集能构成的最大三角形

5.4.1. 问题描述

给定平面上的 n 个点，从其中任选三个点作为顶点，求能构成的最大三角形面积。
需要解决的数据范围满足：$1 \le n \le 50000$.

5.4.2. 问题解法

显然，三角形的顶点一定都在这 n 个点的凸包上，所以先求出凸包。

考虑旋转卡壳。

这题中不能固定一条边，再枚举另外两个点，因为三角形的边不一定在凸包上。

因此，先逆时针枚举一个固定点 i，再逆时针旋转另外两个顶点 j 和 k.·

由于凸包上的一些单调性，我们旋转 j 时停止的条件是 $S_{\triangle ijk}$ 最大，停止后固定 j，以相同停止条件旋转 k.

$S_{\triangle ijk}$ 最大，是指 $S_{\triangle i,j\_\text{last},k} < S_{\triangle ijk}$ 且 $S_{\triangle ijk} > S_{\triangle i,j\_\text{next},k}$·

代码与 **5.2 凸包直径**类似。

6. 半平面交

6.1. 相关定义

6.1.1. 半平面

半平面是一条直线和直线的一侧，是一个点集。

当点集包含直线时，称为闭半平面；当不包含直线时，称为开半平面。

6.1.2. 半平面交

半平面交是多个半平面的交集。

半平面交是一个点集，并且是一个凸集。在直角坐标系上是一个区域。

半平面交在代数意义下，是若干个线性约束条件，每个约束条件形如：

$$a_i x + b_i y \le c_i$$

其中 a_i, b_i, c_i 为常数，且 a_i 和 b_i 不都为零。

半平面交有**图 13** 中的五种可能情况，每个半平面位于边界直线的阴影一侧。

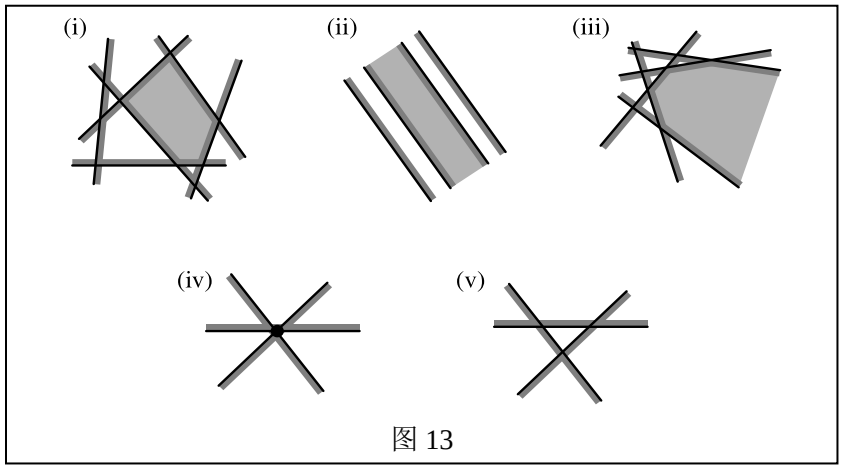

图 13

(iii)和 (v) 比较特殊，我们一般假设产生的交集总是有界或空的。

6.2. Sort-and-Incremental 算法

6.2.1. 算法思想

S&I 算法利用了性质：半平面交是凸集。

因为半平面交是凸集，所以我们维护凸壳。

若我们把半平面按极角排序，那么在过程中，只可能删除队首或队尾的元素，因此使用双端队列维护。

6.2.2. 算法流程

a) 把半平面按照极角序排序，需要 $O(n\log n)$ 的时间。
b) 对每个半平面，执行一次增量过程，每次根据需要弹出双端队列的头部或尾部元素。这是线性的，因为每个半平面只会被增加或删除一次。
c) 最后，通过双端队列中的半平面，在线性时间内求出半平面交（一个凸多边形，用若干顶点描述）。

这样，我们得到了一个时间复杂度为 $O(n\log n)$ 的算法，瓶颈在于排序。因此，若题目给定的半平面满足极角序，则我们可以在线性的时间内求出半平面交。

6.2.3. 参考代码

实现过程中，用直线类表示半平面。直线 (S,T) 表示 ST 左侧的半平面。

形式化地，若点 P 位于直线 (S,T) 表示的半平面中，则 $|\overrightarrow{ST}\times\overrightarrow{SP}|>0$.

```
bool Halfplane_intersection(int n, Line *hp, Point *p) {
    if(n < 3) return false;
    sort(hp, hp + n, cmp);
    Halfplane_unique(n, hp);
    st = 0; ed = 1;
    que[0] = 0; que[1] = 1;
    if(parallel(hp[0], hp[1])) return false;
    Calc_intersection(hp[0], hp[1], p[1]);
    for(int i = 2; i < n; i++) {
        while(st < ed &&
              sgn((hp[i].t-hp[i].s) * (p[ ed ]-hp[i].s)) < 0)
            ed--;
        while(st < ed &&
              sgn((hp[i].t-hp[i].s) * (p[st+1]-hp[i].s)) < 0)
            st++;
        que[++ed] = i;
        assert(ed >= 1);
        if(parallel(hp[i], hp[que[ed - 1]])) return false;
        Calc_intersection(hp[i], hp[que[ed - 1]], p[ed]);
    }
    while(st < ed &&
          sgn((hp[que[st]].t-hp[que[st]].s) * (p[ ed ]-hp[que[st]].s)) < 0)
        ed--;
    while(st < ed &&
          sgn((hp[que[ed]].t-hp[que[ed]].s) * (p[st+1]-hp[que[ed]].s)) < 0)
        st++;
    if(st + 1 >= ed) return false;
    return true;
}
```

6.3. 半平面交的应用

6.3.1. 赛车问题

有一种赛车的获奖方式为若在比赛过程中一辆赛车存在一个时刻处于领跑位置上即可获奖。

为简化问题，视每辆赛车均匀速行驶。则每辆车的 $x-t$ 图像是一条直线。由图像的几何意义，若取每条直线的左侧半平面，再与 x 轴上侧平面、y 轴右侧平面一起求交，则所有凸多边形上的直线可以获奖。问题转化为半平面交问题。

6.3.2. 光的传播问题

此类问题指给定若干障碍物，并规定光的传播方向，求哪些障碍物是可见的。

最基础的模型即为所有障碍物均为直线。即若在 x-y 直角坐标平面上有 n 条直线 $L_1, L_2, \dots L_n$，若在 y 值为正无穷大处往下看，求出所有可见的直线每条直线 L_i 均可表示为 $y = k \times x + b$ 的形式。

从上往下看到的图像一定是一个凸的图像，因此可以转化为半平面交问题。

7. 结语

本文对凸包与半平面交两类问题在算法竞赛及生活中的应用进行了研究。除此之外，这两类问题还在其他多个领域有所应用，特别是图像处理相关行业。

对凸包和半平面交问题的典例整理，可以作为他人学习时的参考资料，使人们在应对算法竞赛中的相关问题时，有更灵活的方法。

8. 参考文献

[1] BERG M, KREVELD M, OVERMARS M H. Computational Geometry: Algorithms and Applications[M]. 2008.

[2] Preparata F P, Shamos M I. Computational geometry: an introduction[M]. Springer Science & Business Media, 2012.

[3] TOUSSAINT G T. Solving geometric problems with the rotating calipers[C]//Proc. IEEE Melecon.1983:A10.

[4] TOUSSAINT G T. The rotating calipers: An efficient, multipurpose, computational tool[C]//The International Conference on Computing Technology and Information Management (ICCTIM).Citeseer,2014:215.

[5] THOMAS H C, CHARLES,E.LEISERSON,RONALD,L.RIVEST,CLIFFORD,STEIN,殷建平,徐云,王刚,刘晓光,苏明,邹恒明,王宏志 2013. 算法导论(原书第 3 版). 计算机教育 [J]: 1.

[6] 董永建. 信息学奥赛一本通[M].科学技术文献出版社,2013.

[7] PRATA S, 张海龙, 袁国忠. C++ Primer Plus（第 6 版）中文版[M]. 人民邮电出版社, 2012.

[8] 同济大学数学系.高等数学.下册[M].北京: 高等教育出版社,2014.1-23.

[9] 人民教育出版社 课程教材研究所 中学数学教材实验研究组.普通 高中教科书 数学（B 版）必修 第二册[M].北京:人民教育出版社,2019:131-172.

9. 附件：程序源代码

```
//1.浮点数相关函数
const int N = 5000;
const double pi = 3.1415926535898;
const double eps = 1e-10;
inline int sgn(double x) {
    if(fabs(x) < eps) return 0;
    else return x > 0 ? 1 : -1;
}
inline int fcmp(double x, double y) {
    if(fabs(x - y) < eps) return 0;
    else return x > y ? 1 : -1;
}
inline double rd() { return (rand()) % 2 ? eps : -eps; }
//2.平面向量类
struct Point{
    int x, y;
    Point(){};
    Point(int a, int b): x(a), y(b) {}
    Point(Point a, Point b): x(b.x - a.x), y(b.y - a.y) {}
    friend Point operator + (const Point &a, const Point &b) {
        return Point(a.x + b.x, a.y + b.y);
    }
    friend Point operator - (const Point &a, const Point &b) {
        return Point(a.x - b.x, a.y - b.y);
    }
    friend Point operator * (const double &a, const Point &b) {
        return Point(a * b.x, a * b.y);
    }
    friend Point operator * (const Point &a, const double &b) {
        return Point(a.x * b, a.y * b);
    }
    friend int operator * (const Point &a, const Point &b) {
        return a.x * b.y - a.y * b.x;
    }
    friend int operator & (const Point &a, const Point &b) {
        return a.x * b.x + a.y * b.y;
    }
    friend bool operator < (const Point &a, const Point &b) {
        return fcmp(a.x, b.x ) == -1 ||
              (fcmp(a.x, b.x)== 0 && fcmp(a.y, b.y) == -1);
    }
    friend bool operator == (const Point &a, const Point &b) {
        return fcmp(a.x, b.x) == 0 && fcmp(a.y, b.y == 0);
    }
    inline double len() {
        return sqrt(1.0 * x * x + 1.0 * y * y);
    }
    int sqr_len() {
        return x * x + y * y;
    }
    Point unit() {
        return Point(x / len(), y / len());
    }
};
typedef Point Vec;
bool cmp1(Point a, Point b) {
    return (fcmp(a.x, b.x) == -1) ||
```

```
            (!fcmp(a.x, b.x) && fcmp(a.y, b.y) == -1);
}
bool cmp2(Point a, Point b) {
    if(atan2(a.y, a.x) - atan2(b.y, b.x) == 0)
        return a.x < b.x;
    return atan2(a.y, a.x) < atan2(b.y, b.x);
}
inline double dis(Point &a, Point &b) {
    return (a - b).len();
}
//平面凸包
inline bool check(Point s1, Point s2, Point p) {
    return Vec(s2, s1) * Vec(s1, p) > 0;
}
int Convex_hull_2d(int n, Point *p, Point *ret) {
    sort(p, p + n, cmp1);
    int top = -1;
    for (int i = 0; i < n; i++) {
        while (top > 0 && !check(ret[top], ret[top - 1], p[i]))
            top--;
        ret[++top] = p[i];
    }
    int k = top;
    for (int i = n - 2; i >= 0; i--) {
        while (top > k && !check(ret[top], ret[top - 1], p[i]))
            top--;
        ret[++top] = p[i];
    }
    return top;
}
double Convex_hull_2d_L(int n, Point *p) {
    Point convex[N];
    int siz = Convex_hull_2d(n, p, convex);
    double ans = dis(convex[0], convex[siz - 1]);
    for (int i = 1; i < siz; i++)
        ans += dis(convex[i - 1], convex[i]);
    return ans;
}
double Convex_hull_2d_S(int n, Point *p) {
    Point convex[N];
    int siz = Convex_hull_2d(n, p, convex);
    double ans = 0;
    for (int i = 2; i < siz; i++)
        ans += area(convex[0], convex[i - 1], convex[i]);
    return ans;
}
//3.三维向量类
struct Point3 {
    double x, y, z;
    Point3(){};
    Point3(double a, double b, double c) : x(a), y(b), z(c) {}
    Point3 operator + (const Point3 &b) {
        return Point3(x + b.x, y + b.y, z + b.z);
    }
    Point3 operator - (const Point3 &b) {
        return Point3(x - b.x, y - b.y, z - b.z);
    }
    Point3 operator * (const Point3 &b) {
        return Point3(y*b.z - z*b.y, z*b.x - x*b.z, x*b.y - y*b.x);
    }
```

```
    double operator & (const Point3 &b) {
        return x * b.x + y * b.y + z * b.z;
    }
    bool operator == (const Point3 &b) {
        return fcmp(x, b.x) == 0 && fcmp(y, b.y) == 0 && fcmp(z, b.z) == 0;
    }
    double len() {
        return sqrt(x * x + y * y + z * z);
    }
    void shake() { x += rd(), y += rd(), z += rd(); }
} p[N];
//4.平面类
struct plane{
    int v[3];
    plane(){};
    plane(int a, int b, int c) { v[0] = a, v[1] = b, v[2] = c; }
    Point3 normal() {
        return (p[v[1]] - p[v[0]]) * (p[v[2]] - p[v[0]]);
    }
    bool is_above(Point3 A) {
        return (normal() & (A - p[v[0]])) >= 0;
    }
    double area() {
        return normal().len() / 2.0;
    }
};
int Convex_hull_3d(int n, plane *ret) {
    plane tmp[N];
    bool g[N][N];
    for (int i = 0; i < n; i++) p[i].shake();
    int top = -1;
    ret[++top] = plane(0, 1, 2);
    ret[++top] = plane(0, 2, 1);
    for (int i = 3; i < n; i++)
    {
        int cnt = -1;
        for (int j = 0; j <= top; j++)
        {
            bool flag = ret[j].is_above(p[i]);
            if (!flag)
                tmp[++cnt] = ret[j];
            for (int k = 0; k < 3; k++)
                g[ret[j].v[k]][ret[j].v[(k + 1) % 3]] = flag;
        }
        for (int j = 0; j <= top; j++)
        {
            for (int k = 0; k < 3; k++)
            {
                int a = ret[j].v[k], b = ret[j].v[(k + 1) % 3];
                if (g[a][b] && !g[b][a])
                    tmp[++cnt] = plane(a, b, i);
            }
        }
        for (int j = 0; j <= cnt; j++) ret[j] = tmp[j];
        top = cnt;
    }
    return (top + 1);
}
double Convex_hull_3d_S(int n) {
    plane convex[N];
```

```
    int siz = Convex_hull_3d(n, convex);
    double ret = 0;
    for (int i = 0; i < siz; i++) ret += convex[i].area();
    return ret;
}

//5.旋转卡壳
//凸包直径
inline int sqr_dis(Point a, Point b) {
    return (a - b).sqr_len();
}
int Get_Max(int n, Point *ch) {//传入 convex-hull
    int ret = 0;
    ch[n] = ch[0];
    int j = 1;
    for(int i = 0; i < n; i++) {
        while((ch[i] - ch[j+1]) * (ch[i+1] - ch[j+1]) >
              (ch[i] - ch[j  ]) * (ch[i+1] - ch[j  ]))
            j = (j + 1) % n;
        ret = max(ret, max(sqr_dis(ch[i], ch[j]), sqr_dis(ch[i+1], ch[j])));
    }
    return ret;
}
//最小矩形覆盖
double Get_Min(int n, Point *ch, Point *Ans) {
    ch[n] = ch[0];
    int u = 2, l, r = 2;
    //u 是距离 AB 最远的点；在 AB 为底时，l 和 r 是两个最靠边的点
    double ret = 1e100, H, L, R, S;
    for(int i = 0; i < n; i++) {
        Point A = ch[i], B = ch[i+1]; Vec AB = B - A, BA = A - B;
        while((AB * Vec(B, ch[u+1])) >= (AB * Vec(B, ch[u])))
            u = (u + 1) % n;
        while((AB & Vec(B, ch[r+1])) >= (AB & Vec(B, ch[r])))
            r = (r + 1) % n;
        if(i == 0) l = r;
        while((AB & Vec(B, ch[l+1])) <= (AB & Vec(B, ch[l])))
            l = (l + 1) % n;
        H = (AB * Vec(B, ch[u])) / AB.len(); //以 AB 所在直线为底边，矩形的高
        L = (BA & Vec(A, ch[l])) / BA.len(); //A 距离左侧顶点的距离
        R = (AB & Vec(B, ch[r])) / AB.len(); //B 距离右侧顶点的距离
        S = H * (L + AB.len() + R); //矩形面积
        if(S < ret) { //求矩形顶点坐标
            ret = S;
            Ans[0] = A + L * BA.unit();
            Ans[1] = B + R * AB.unit();
            Ans[2] = Ans[1] + H * (ch[r]-Ans[1]).unit();
            Ans[3] = Ans[0] + H * (ch[l]-Ans[0]).unit();
        }
    }
    return ret;
}
//6.直线类
struct Line{
    Point s, t;
    Line() {};
    Line(Point a, Point b) : s(a), t(b) {}
    double ang() { return atan2((t - s).y, (t - s).x); };
    friend bool parallel(const Line &A, const Line &B) {
```

```
        return sgn((A.s - A.t) * (B.s - B.t)) == 0;
    }
    friend bool Calc_intersection(Line &A, Line &B, Point &res) {
        if(parallel(A, B)) return false;
        double s1 = (B.t - B.s) * (B.s - A.s);
        double s2 = (B.t - B.s) * (A.t - A.s);
        res = A.s + (A.t - A.s) * (s1 / s2);
        return true;
    }
} L[N];
inline bool cmp_Line(Line A, Line B) {
    //极角相等时，位置靠右的排在前面
    if(!sgn(A.ang() - B.ang())) return (A.t - A.s) * (B.t - A.s) > 0;
    return A.ang() < B.ang();
}
void Halfplane_unique(int &n, Line *hp) {
    int m = 0;
    for(int i = 0; i < n - 1; i++) {
        if(sgn(hp[i].ang() - hp[i + 1].ang()) == 0) continue;
        hp[m++] = hp[i];
    }
    hp[m++] = hp[n - 1];
    n = m;
}
int que[N], st, ed;
bool Halfplane_intersection(int n, Line *hp, Point *p) {
    if(n < 3) return false;
    sort(hp, hp + n, cmp_Line);
    Halfplane_unique(n, hp);
    st = 0; ed = 1;
    que[0] = 0; que[1] = 1;
    if(parallel(hp[0], hp[1])) return false;
    Calc_intersection(hp[0], hp[1], p[1]);
    for(int i = 2; i < n; i++) {
        while(st < ed &&
              sgn((hp[i].t - hp[i].s) * (p[ed] - hp[i].s)) < 0)
            ed--;
        while(st < ed &&
              sgn((hp[i].t - hp[i].s) * (p[st + 1] - hp[i].s)) < 0)
            st++;
        que[++ed] = i;
        if(parallel(hp[i], hp[que[ed - 1]])) return false;
        Calc_intersection(hp[i], hp[que[ed - 1]], p[ed]);
    }
    while(st < ed &&
          sgn((hp[que[st]].t - hp[que[st]].s) * (p[ed] - hp[que[st]].s)) < 0)
        ed--;
    while(st < ed &&
          sgn((hp[que[ed]].t - hp[que[ed]].s) * (p[st + 1] - hp[que[ed]].s)) < 0)
        st++;
    if(st + 1 >= ed) return false;
    return true;
}
int Get_convex_hull(Line *hp, Point *p, Point *ch) {
    Calc_intersection(hp[que[st]], hp[que[ed]], p[st]);
    for(int i = 0, j = st; j <= ed; i++, j++) ch[i] = p[j];
    return ed - st + 1;
}
double area(Point &a, Point &b, Point &c) {
    return (b - a) * (c - a) / 2.0;
```

```
}
double Calc_area(int n, Point *ch) {
    double ans = 0;
    for (int i = 2; i < n; i++)
        ans += area(ch[0], ch[i - 1], ch[i]);
    return ans;
}
```