计算几何 1：点、向量、直线、凸包

Ebola

Institute of Mathematics,
Zhejiang University.

Jan, 2024

1. 二维几何基础
2. 二维凸包
3. 点与直线

点与向量

二维平面上的任何一个点，可以用坐标 (x, y) 表示。

点与向量

二维平面上的任何一个点，可以用坐标 (x, y) 表示。

向量是一个“具有方向和长度的箭头”，它不规定起点和终点。
二维平面上的任何一个向量，也可以用坐标 (x, y) 表示。

点与向量

二维平面上的任何一个点，可以用坐标 (x, y) 表示。

向量是一个“具有方向和长度的箭头”，它不规定起点和终点。
二维平面上的任何一个向量，也可以用坐标 (x, y) 表示。

计算机存储点与向量没有区别，所以我们都可以用下面的结构体来存储。

```
struct Point{
    double x,y;
    Point(double x=0, double y=0): x(x), y(y) {}
};
#define Vector Point
// 在计算机里，Vector 就是 Point，但为了从逻辑上区分，我们赋予它们不同的名字
```

浮点数比大小

浮点数是有限精度的，在运算过程中，难免会产生误差，相信大家深有被卡精度的体会。但是在计算几何中，我们经常需要判断浮点数的大小。

浮点数比大小

浮点数是有限精度的，在运算过程中，难免会产生误差，相信大家深有被卡精度的体会。但是在计算几何中，我们经常需要判断浮点数的大小。这里我们引入如下的比较函数：

```
#define eps 1e-12
int dcmp(double x)
{
    if(fabs(x)<=eps) return 0;
    else if(x<0) return -1;
    else return 1;
}
```

向量的基本运算

我们重载一些运算符来实现向量基本运算。

```
Vector operator + (Vector a, Vector b){return Vector(a.x+b.x, a.y+b.y);}
Vector operator - (Vector a, Vector b){return Vector(a.x-b.x, a.y-b.y);}
Vector operator - (Vector b){return Vector(-b.x, -b.y);}
Vector operator * (Vector a, double x){return Vector(a.x*x, a.y*x);}
Vector operator * (double x, Vector a){return Vector(a.x*x, a.y*x);}
double Angle(Vector a){return atan2(a.y, a.x);}

bool operator < (Point a, Point b){
    return a.x < b.x || (a.x == b.x && a.y < b.y);
}
bool operator == (Point a, Point b){
    return dcmp(a.x-b.x) == 0 && dcmp(a.y-b.y) == 0;
}

double Dot(Vector a, Vector b){return a.x*b.x + a.y*b.y;}
double Length(Vector a){return sqrt(a.x*a.x + a.y*a.y);}
```

向量的叉乘

二维向量叉乘写作 $\mathbf{a} \times \mathbf{b}$，定义如下：

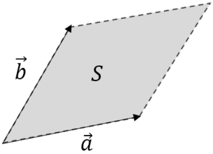
```
double Cross(Vector a,Vector b){return a.x*b.y - a.y*b.x;}
```

向量的叉乘

二维向量叉乘写作 $\mathbf{a} \times \mathbf{b}$，定义如下：

```
double Cross(Vector a,Vector b){return a.x*b.y - a.y*b.x;}
```

在几何中，叉乘是向量 a 与 b 构成的平行四边形的有向面积。如果 b 在 a 的**逆时针**方向，结果就是**正**的；逆时针方向就是负的；平行就是零。



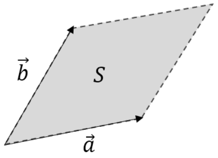
$\vec{a} \times \vec{b} = S \qquad \vec{b} \times \vec{a} = -S$

向量的叉乘

二维向量叉乘写作 a × b，定义如下：

```
double Cross(Vector a,Vector b){return a.x*b.y - a.y*b.x;}
```

在几何中，叉乘是向量 a 与 b 构成的平行四边形的有向面积。如果 b 在 a 的**逆时针**方向，结果就是**正**的；逆时针方向就是负的；平行就是零。



$\vec{a} \times \vec{b} = S$ $\vec{b} \times \vec{a} = -S$

当向量坐标都是整数时，用叉乘判断向量相对位置没有精度误差!

叉乘的应用：将凸多边形的顶点按逆时针排序

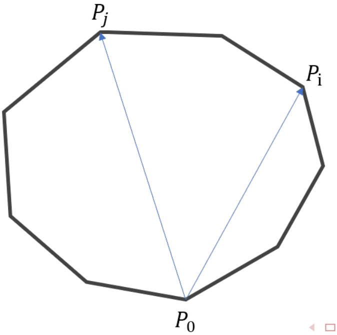
给定凸 n 边形的所有顶点，请将它们按逆时针排序，起点随意。
（提示：使用 sort 函数，考虑如何定义 cmp）

叉乘的应用：将凸多边形的顶点按逆时针排序

给定凸 n 边形的所有顶点，请将它们按逆时针排序，起点随意。
（提示：使用 sort 函数，考虑如何定义 cmp）

先随意固定一个起点 P_0，P_i 排在 P_j 前面，当且仅当

$$\overrightarrow{P_0P_i} \times \overrightarrow{P_0P_j} > 0.$$

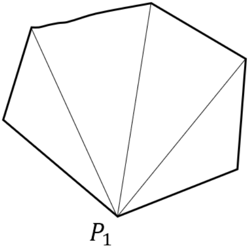


叉乘的应用：求凸多边形的面积

给定凸 n 边形的所有顶点，它们已经按逆时针排好了序，求图形的面积。

叉乘的应用：求凸多边形的面积

给定凸 n 边形的所有顶点，它们已经按逆时针排好了序，求图形的面积。



依次叉乘并累加即可。

```
double area = 0;
for(int i = 2; i <= n-1; i++)
    area += 0.5 * cross(p[i]-p[1], p[i+1]-p[1]);
```

1 二维几何基础

2 二维凸包

3 点与直线

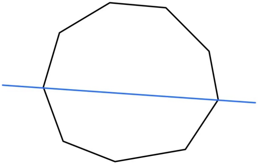
凸包

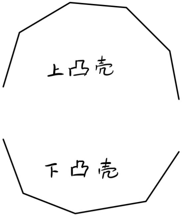
给定 n 个点，你需要从中选取若干个点构成一个凸多边形，并且这个凸多边形包住了所有的点。

模板题：P2742 [USACO5.1] 圈奶牛

凸包的拆分

我们通常将凸多边形拆分成两个部分：上凸壳和下凸壳。上下凸壳的分界点是凸多边形最左与最右的顶点。



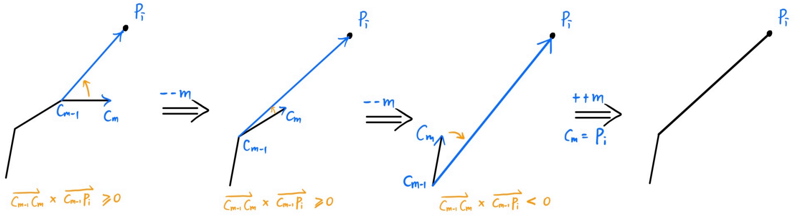


上凸壳的维护

维护上凸壳的一个基本想法是：用一个栈来存储当前上凸壳，然后考虑添加一个新的点。为了维护凸性，我们先弹出栈顶的一些元素，然后再将这个点加入。栈顶元素是否需要弹出可以根据叉乘的符号来判断。

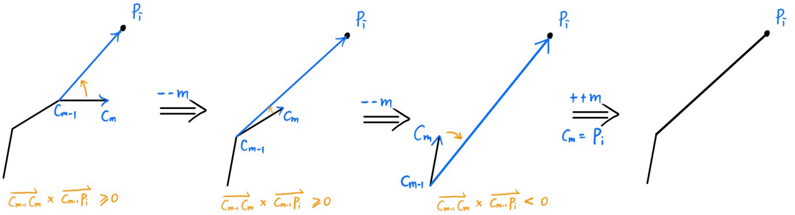
上凸壳的维护

维护上凸壳的一个基本想法是：用一个栈来存储当前上凸壳，然后考虑添加一个新的点。为了维护凸性，我们先弹出栈顶的一些元素，然后再将这个点加入。栈顶元素是否需要弹出可以根据叉乘的符号来判断。



上凸壳的维护

维护上凸壳的一个基本想法是：用一个栈来存储当前上凸壳，然后考虑添加一个新的点。为了维护凸性，我们先弹出栈顶的一些元素，然后再将这个点加入。栈顶元素是否需要弹出可以根据叉乘的符号来判断。



现在问题是：按什么样的顺序考虑新的点，才能保证正确地求出上凸壳？

点的加入顺序

答案是**从左到右、从下到上**。即将所有点按 x 为第一关键字、y 为第二关键字进行排序。这样为什么是对的？

点的加入顺序

答案是**从左到右、从下到上**。即将所有点按 x 为第一关键字、y 为第二关键字进行排序。这样为什么是对的？

其实我们只需要保证上凸壳上的点的访问顺序是从左到右，并且上凸壳最右边的点排在最后一个即可。对于不在上凸壳上的点，它们的顺序不重要，因为都会被弹出。显然，“从左到右、从下到上”符合上述要求。

下凸壳的维护

维护下凸壳也很简单，只要把点的访问顺序倒过来即可，其余部分完全一样。

完整代码

```
bool operator < (const Point &a,const Point &b){
    return a.x < b.x || (a.x == b.x && a.y < b.y);
}
int ConvexHull(Point a[],int n,Point b[]){
    sort(a+1, a+n+1);
    int m = 0;
    for(int i = 1; i <= n; i++){
        while(m > 1 && cross(b[m]-b[m-1], a[i]-b[m-1]) >= 0) --m;
        b[++m] = a[i];
    }
    int k = m;
    for(int i = n-1; i >= 1; i--){
        while(m > k && cross(b[m]-b[m-1], a[i]-b[m-1]) >= 0) --m;
        b[++m] = a[i];
    }
    return m-1;
}
```

完整代码

```
bool operator < (const Point &a,const Point &b){
    return a.x < b.x || (a.x == b.x && a.y < b.y);
}
int ConvexHull(Point a[],int n,Point b[]){
    sort(a+1, a+n+1);
    int m = 0;
    for(int i = 1; i <= n; i++){
        while(m > 1 && cross(b[m]-b[m-1], a[i]-b[m-1]) >= 0) --m;
        b[++m] = a[i];
    }
    int k = m;
    for(int i = n-1; i >= 1; i--){
        while(m > k && cross(b[m]-b[m-1], a[i]-b[m-1]) >= 0) --m;
        b[++m] = a[i];
    }
    return m-1;
}
```

这样得到的凸包顶点是按顺时针方向排序的，如果要按逆时针方向排序，可以 reverse 一下，或者直接把上面代码里的 >= 改成 <=。

判断点与线段的位置关系

考虑如何判断点 P 是否在线段 AB 上。(不要去想斜率，因为算斜率会引入精度误差，而且斜率无穷大时还要特判)

判断点与线段的位置关系

考虑如何判断点 P 是否在线段 AB 上。(不要去想斜率，因为算斜率会引入精度误差，而且斜率无穷大时还要特判)

$$\overrightarrow{AB} \times \overrightarrow{AP} = 0 \tag{1}$$

$$\overrightarrow{PA} \cdot \overrightarrow{PB} < 0 \tag{2}$$

判断点与线段的位置关系

考虑如何判断点 P 是否在线段 AB 上。(不要去想斜率，因为算斜率会引入精度误差，而且斜率无穷大时还要特判)

$$\overrightarrow{AB} \times \overrightarrow{AP} = 0 \tag{1}$$

$$\overrightarrow{PA} \cdot \overrightarrow{PB} < 0 \tag{2}$$

```
bool PointInSegment(Point p, Point a, Point b){
    return Cross(b-p, a-p) == 0 && Dot(b-p, a-p) < 0;
}
```

判断点与线段的位置关系

考虑如何判断点 P 是否在线段 AB 上。（不要去想斜率，因为算斜率会引入精度误差，而且斜率无穷大时还要特判）

$$\overrightarrow{AB} \times \overrightarrow{AP} = 0 \tag{1}$$

$$\overrightarrow{PA} \cdot \overrightarrow{PB} < 0 \tag{2}$$

```
bool PointInSegment(Point p, Point a, Point b){
    return Cross(b-p, a-p) == 0 && Dot(b-p, a-p) < 0;
}
```

其实通过 $\overrightarrow{AB} \times \overrightarrow{AP}$ 的符号，我们还可以判断 P 的方位：大于零时，在 $\overrightarrow{AB}$ 左侧；小于零时，在 $\overrightarrow{AB}$ 右侧。

判断两条线段是否相交

考虑如何判断线段 AB 与线段 CD 是否相交。

判断两条线段是否相交

考虑如何判断线段 AB 与线段 CD 是否相交。

两条线段相交当且仅当下面两个条件同时满足：

- A, B 在 l_{CD} 的两侧；
- C, D 在 l_{AB} 的两侧。

判断两条线段是否相交

考虑如何判断线段 AB 与线段 CD 是否相交。

两条线段相交当且仅当下面两个条件同时满足:

- A, B 在 l_{CD} 的两侧;
- C, D 在 l_{AB} 的两侧。

```
bool SegmentProperIntersection(Point a1, Point a2, Point b1, Point b2)
{
    long long c1 = Cross(b1-a1, a2-a1);
    long long c2 = Cross(b2-a1, a2-a1);
    long long c3 = Cross(a1-b1, b2-b1);
    long long c4 = Cross(a2-b1, b2-b1);
    return c1*c2 < 0 && c3 * c4 < 0;
}
```

判断点是否在凸多边形内部

考虑如何判断点 P 是否在凸多边形的内部（或者边界上）。其中凸多边形的顶点 $A_1, \dots, A_n$ 按照逆时针顺序给出。

判断点是否在凸多边形内部

考虑如何判断点 P 是否在凸多边形的内部（或者边界上）。其中凸多边形的顶点 $A_1, \ldots, A_n$ 按照逆时针顺序给出。

判断是否在边界上很简单，用刚刚的 `PointInSegment` 即可。

判断点是否在凸多边形内部

考虑如何判断点 P 是否在凸多边形的内部（或者边界上）。其中凸多边形的顶点 $A_1, ..., A_n$ 按照逆时针顺序给出。

判断是否在边界上很简单，用刚刚的 `PointInSegment` 即可。

判断是否在内部只需检查以下条件：

$$\overrightarrow{A_iA_{i+1}} \times \overrightarrow{A_iP} > 0 \tag{3}$$

对所有的 i 都成立（当 $i = n$ 时，$i + 1$ 用 1 代替）。

判断点是否在任意多边形内部

考虑如何判断点 P 是否在任意多边形的内部（或者边界上）。其中多边形的顶点 $A_1, \ldots, A_n$ 按照逆时针顺序给出。

判断点是否在任意多边形内部

考虑如何判断点 P 是否在任意多边形的内部（或者边界上）。其中多边形的顶点 $A_1, \ldots, A_n$ 按照逆时针顺序给出。

射线法：从 P 向任意方向引出一条射线，如果射线与多边形边界有奇数个交点，说明在内部，否则就在外部。（写程序时，一般取水平向右的射线）

判断点是否在任意多边形内部

```
bool PointInPolygon(Point p,Point* res,int cnt)
{
    int wn=0;
    for (int i=0;i<cnt;i++)
    {
        if(res[i]==p||res[(i+1)%cnt]==p||PointInSegment(p,res[i],res[(i+1)%cnt]))
            return 1;
        // 射线法
        int k=Cross(res[(i+1)%cnt]-res[i],p-res[i]);
        int d1=res[i].y-p.y;
        int d2=res[(i+1)%cnt].y-p.y;
        if(k>0&&d1<=0&&d2>0) wn++;
        if(k<0&&d2<=0&&d1>0) wn--;
    }
    if(wn) return 1;
    return 0;
}
```

[UVA10256] 判断两个点集能否分离

给定两组点集（每组最多 500 个点），问是否存在一条直线能将它们分离？

[UVA10256] 判断两个点集能否分离

给定两组点集（每组最多 500 个点），问是否存在一条直线能将它们分离？

分别求凸包，转换为两个凸包是否相交的判定问题。

[UVA10256] 判断两个点集能否分离

给定两组点集（每组最多 500 个点），问是否存在一条直线能将它们分离？

分别求凸包，转换为两个凸包是否相交的判定问题。

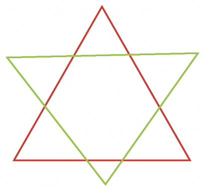
枚举凸包 A 的所有顶点，判断是否在凸包 B 内部，然后反过来再来一次。（这样是否充分？）

[UVA10256] 判断两个点集能否分离

给定两组点集（每组最多 500 个点），问是否存在一条直线能将它们分离？

分别求凸包，转换为两个凸包是否相交的判定问题。

枚举凸包 A 的所有顶点，判断是否在凸包 B 内部，然后反过来再来一次。(这样是否充分 ?)



显然不充分，所以还要枚举凸包 A 与凸包 B 的所有边，然后调用线段相交的判定方法。

Thank You