

基础

点(Point)

```
1 struct Point {
2     db x, y;
3 };
4
5 inline db dis(const Point &a, const Point &b) {
6     db dx = a.x - b.x;
7     db dy = a.y - b.y;
8     return sqrt(dx * dx + dy * dy);
9 }
```

向量(Vector)

```
1 using Vector = Point;
2
3 inline db dot(const Vector &a, const Vector &b) {
4     return a.x * b.x + a.y * b.y;
5 }
6
7 inline db cross(const Vector &a, const Vector &b) {
8     return a.x * b.y - a.y * b.x;
9 }
10
11 inline Vector operator+(const Point &a, const Point &b) {
12     return Vector{a.x + b.x, a.y + b.y};
13 }
14
15 inline Vector operator-(const Point &a, const Point &b) {
16     return Vector{a.x - b.x, a.y - b.y};
17 }
18
19 inline Vector operator*(const Vector &a, const db &b) {
20     return Vector{a.x * b, a.y * b};
21 }
```

基础用法

- 将一个向量 \vec{a} 逆时针旋转 θ 度

$$\begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix} \times \begin{bmatrix} a_x \\ b_x \end{bmatrix} = \begin{bmatrix} \cos \theta a_x - \sin \theta a_y \\ \sin \theta a_x + \cos \theta a_y \end{bmatrix}$$

点积(Dot)

$$\vec{a} \cdot \vec{b} = a_x b_x + a_y b_y$$

叉积(Cross)

$$\vec{a} \times \vec{b} = a_x b_y - a_y b_x$$

- 平行四边形面积: $\|\vec{a}\| \|\vec{b}\| |\sin \theta| = \|\vec{a} \times \vec{b}\|$
- 向量平行: $\vec{a} \times \vec{b} = 0$
- **TO_LEFT测试**
 - 判断点 P 在向量 AB 的左侧还是右侧
 - $\vec{a} \times \vec{b} > 0$ P 在向量 AB 左侧
 - $\vec{a} \times \vec{b} < 0$ P 在向量 AB 右侧
 - $\vec{a} \times \vec{b} = 0$ P 在向量 AB 上

线段(Segment)

```
1 struct Segment {  
2     Point a, b;  
3 };
```

基础用法

- 判断点 P 是否在线段 AB 上(含端点)
 - $\vec{a} \times \vec{b} = \vec{0}$
 - $\vec{a} \cdot \vec{b} \leq 0$
- 判断线段 AB, CD 是否相交
 - 特判三点共线和四点共线
 - 通过叉积判断
 - 点 C 和点 D 在线段 AB 的不同侧
 - 点 A 和点 B 在线段 CD 的不同侧

直线

点向式 `struct Line { Point p; Vector v; };`

基础用法

- 输入直线 (P, \vec{v}) 与点 A , 求 A 到直线距离

$$\|\vec{AB}\| = \|\vec{PA}\| |\sin \theta| = \frac{\|\vec{v} \times \vec{PA}\|}{\|\vec{v}\|}$$

- 输入直线(P, \vec{v})与点A, 求A在直线上的投影点B 点乘算投影
- 两直线求交点 直线 $\{P_1, \vec{v}_1\}\{P_2, \vec{v}_2\}$

$$\begin{cases} \frac{\|P_1Q\|}{\sin \alpha} = \frac{\|P_1P_2\|}{\sin \beta} \\ \|\vec{v}_2 \times P_2\vec{P}_1\| = \|\vec{v}_2\| \|P_2\vec{P}_1\| \sin \alpha \end{cases}$$

$$\|\vec{v}_1 \times \vec{v}_2\| = \|\vec{v}_1\| \|\vec{v}_2\| \sin \beta$$

$$\|P_1Q\| = \frac{\|\vec{v}_2 \times P_2\vec{P}_1\| \|\vec{v}_1\|}{\|\vec{v}_1 \times \vec{v}_2\|}$$

$$\vec{OQ} = \vec{OP}_1 + P_1\vec{Q} = \vec{OP}_1 + \frac{\|P_1Q\|}{\|\vec{v}_1\|} \vec{v}_1 = \vec{OP}_1 + \frac{\|\vec{v}_2 \times P_2\vec{P}_1\|}{\|\vec{v}_1 \times \vec{v}_2\|} \vec{v}_1$$

```

1 inline Point l_to_l(const Line &l1, const Line &l2) {
2     Vector w = l2.p - l1.p; // P2 - P1
3     db denom = cross(l1.v, l2.v);
4     if (fabs(denom) < 1e-9) return {1e18, 1e18};
5     db t = cross(w, l2.v) / denom;
6     return {l1.p.x + t * l1.v.x, l1.p.y + t * l1.v.y};
7 }

```

- 判断射线与线段是否相交

```

1 // 射线 r 与线段 s 是否相交
2 inline bool r_to_s(const Line &r, const Seg &s) {
3     Vector d = r.v; // 射线方向
4     Vector v = s.b - s.a; // 线段方向
5     Vector w = r.p - s.a; // 射线起点到线段起点向量
6
7     db denom = cross(v, d);
8     if (fabs(denom) < eps) return false;
9
10    db t = cross(w, v) / denom;
11    db u = cross(w, d) / denom;
12
13    return t >= -eps && u >= -eps && u <= 1 + eps;
14 }

```

多边形

`struct Polygon { vector<Point> p; };` 一般默认按照逆时针排序

基础用法

- 计算多边形面积(三角剖分) $S = \frac{1}{2} \|\sum_{i=0}^{n-1} \vec{OP_i} \times \vec{OP_{(i+1) \bmod n}}\|$
- 判断点是否在多边形内部
 1. 从该点引出一条射线，如果与多边形有奇数个交点，则在内部，否则在多边形外部 (不能交到顶点)
 2. 遍历多边形的点，如果转动圈数为0，点在多边形外部，否则在内部 (计算角度有精度误差)
 3. 水平引出一条射线，逆时针依次遍历边，如果边从上向下穿过射线，val--，否则val++，如果val=0则点不在多边形内 (优秀)

1 |

圆

模版

点(Point)

```
1  template<typename T>
2  struct Point {
3      T x, y;
4
5      bool operator==(const Point &a) const { return (abs(x - a.x) <= eps && abs(y - a.y)
6      <= eps); }
7      Point operator+(const Point &a) const { return {x + a.x, y + a.y}; }
8      Point operator-(const Point &a) const { return {x - a.x, y - a.y}; }
9      Point operator-() const { return {-x, -y}; }
10     Point operator*(const T k) const { return {k * x, k * y}; }
11     Point operator/(const T k) const { return {x / k, y / k}; }
12     T operator*(const Point &a) const { return x * a.x + y * a.y; } // Dot
13     T operator^(const Point &a) const { return x * a.y - y * a.x; } // Cross
14     bool operator<(const Point &a) const {
15         if (abs(x - a.x) <= eps) return y < a.y - eps;
16         return x < a.x - eps;
17     }
18
19     bool is_par(const Point &a) const { return abs((*this) ^ a) <= eps; } // 平行
20     bool is_ver(const Point &a) const { return abs((*this) * a) <= eps; } // 垂直
21
22     int toleft(const Point &a) const {
23         auto t = (*this) ^ a;
24         return (t > eps) - (t < -eps);
25     }
26
27     T len2() const { return (*this) * (*this); }
28     T dis2(const Point &a) const { return (a - (*this)).len2(); }
29     double len() const { return sqrt(len2()); }
30     double dis(const Point &a) const { return (a - (*this)).len(); }
31     double ang(const Point &a) const { return acos((( *this) * a) / ((*this)->len() *
32     a.len())); } // 普通夹角 (只返回大小 [0, π])
33     double signed_ang(const Point &a) const { return atan2((*this) ^ a, (*this) * a); }
34     // 带方向夹角: 逆时针为正, 顺时针为负, 范围 (-π, π]
35     Point rot(const double rad) const { return {x * cos(rad) - y * sin(rad), x *
36     sin(rad) + y * cos(rad)}; }
37 };
38
39 template<typename T>
40 bool is_on(const Point<T> p, const Point<T> a, const Point<T> b) {
41     return abs((p - a) ^ (p - b)) <= eps && (p - a) * (p - b) <= eps;
42 }
43
44 template<typename T>
45 bool is_in(const Point<T> p, const Point<T> a, const Point<T> b, const Point<T> c) {
46     if (is_on(p, a, b) || is_on(p, b, c) || is_on(p, c, a)) return true;
```

```

44     if ((b - a).toleft(p - a) > 0 && (c - b).toleft(p - b) > 0 && (a - c).toleft(p - c)
    > 0) return true;
45     if ((b - a).toleft(p - a) < 0 && (c - b).toleft(p - b) < 0 && (a - c).toleft(p - c)
    < 0) return true;
46     return false;
47 }

```

线(Line)

```

1  template<typename T>
2  struct Line {
3      Point<T> p, v; //p+kv
4
5      bool operator==(const Line &a) const { return v.is_par(a.v) && v.is_par(p - a.p); }
6      bool is_par(const Line &a) const { return v.is_par(a.v) && !v.is_par(p - a.p); } //
    排除共线
7      bool is_ver(const Line &a) const { return v.is_ver(a.v); }
8      bool is_on(const Point<T> &a) const { return v.is_par(a - p); }
9      int toleft(const Point<T> &a) const { return v.toleft(a - p); }
10     Point<T> inter(const Line &a) const { return p + v * ((a.v ^ (p - a.p)) / (v ^
    a.v)); }
11     double dis(const Point<T> &a) const { return abs(v ^ (a - p)) / v.len(); }
12     Point<T> proj(const Point<T> &a) const { return p + v * ((v * (a - p)) / (v * v)); }
13
14     bool operator<(const Line &a) const {
15         if (abs(v ^ a.v) <= eps && v * a.v >= -eps) return toleft(a.p) == -1;
16         return argcmp(v, a.v);
17     }
18 };

```

多边形(Polygon)

```

1  template<typename T>
2  struct Polygon {
3      vector<Point<T> > p;
4
5      size_t nxt(const size_t i) const { return i == p.size() - 1 ? 0 : i + 1; }
6      size_t pre(const size_t i) const { return i == 0 ? p.size() - 1 : i - 1; }
7
8      // 计算绕多边形一圈转了几圈
9      pair<bool, int> winding(const Point<T> &a) const {
10         int cnt = 0;
11         for (size_t i = 0; i < p.size(); i++) {
12             Point<T> u = p[i], v = p[nxt(i)];
13             if (is_on(a, u, v)) return {true, 0};
14             if (abs(u.y - v.y) <= eps) continue;
15             Line<T> uv = {u, v - u};

```

```

16         if (u.y < v.y - eps && uv.toleft(a) <= 0) continue;
17         if (u.y > v.y + eps && uv.toleft(a) >= 0) continue;
18         if (u.y < a.y - eps && v.y >= a.y - eps) cnt++;
19         if (u.y >= a.y - eps && v.y < a.y - eps) cnt--;
20     }
21     return {false, cnt};
22 }
23 };

```

计算一条直线在一个多边形内的最长直线

```

1  int n;
2  Polygon<db> poly;
3  set<pair<Point<db>, Point<db> > > edges;
4
5  template<typename T>
6  db calc(const Line<T> &l) {
7      vector<tuple<Point<db>, Point<db>, Point<db> > > vec;
8      for (int i = 0; i < n; i++) {
9          auto u = poly.p[i], v = poly.p[poly.nxt(i)];
10         int c1 = l.toleft(u), c2 = l.toleft(v);
11         if (c1 * c2 <= 0) {
12             if (c1 == 0 && c2 == 0) {
13                 vec.emplace_back(u, u, v);
14                 vec.emplace_back(v, u, v);
15             } else {
16                 auto s = l.inter({u, u - v});
17                 vec.emplace_back(s, u, v);
18             }
19         }
20     }
21     sort(vec.begin(), vec.end());
22     int cnt = 0;
23     Point pre = {1e12, 1e12};
24     db len = 0, maxlen = 0;
25     while (!vec.empty()) {
26         auto [now,u,v] = vec.back();
27         if (cnt || edges.count({now, pre})) {
28             len += now.dis(pre);
29         } else {
30             maxlen = max(maxlen, len);
31             len = 0;
32         }
33         while (!vec.empty() && get<0>(vec.back()) == now) {
34             auto [p,u,v] = vec.back();
35             vec.pop_back();
36             if (l.toleft(u) == -1) cnt++;
37             else if (l.toleft(v) == -1) cnt--;
38         }
39         pre = now;
40     }
41     return max(maxlen, len);

```

