基础

点(Point)

```cpp
struct Point {
    db x, y;
};

inline db dis(const Point &a, const Point &b) {
    db dx = a.x - b.x;
    db dy = a.y - b.y;
    return sqrt(dx * dx + dy * dy);
}
```

向量(Vector)

```cpp
using Vector = Point;

inline db dot(const Vector &a, const Vector &b) {
    return a.x * b.x + a.y * b.y;
}

inline db cross(const Vector &a, const Vector &b) {
    return a.x * b.y - a.y * b.x;
}

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

inline Vector operator-(const Point &a, const Point &b) {
    return Vector{a.x - b.x, a.y - b.y};
}

inline Vector operator*(const Vector &a, const db &b) {
    return Vector{a.x * b, a.y * b};
}
```

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)

```
struct Segment {
    Point a, b;
};
```

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 $\{P_1, \vec{v_1}\}\{P_2, \vec{v_2}\}$

$$\begin{cases} \dfrac{\|P_1Q\|}{\sin\alpha} = \dfrac{\|P_1P_2\|}{\sin\beta} \\ \|\vec{v_2} \times \vec{P_2P_1}\| = \|\vec{v_2}\|\|\vec{P_2P_1}\| \sin\alpha \end{cases}$$

$$\|\vec{v_1} \times \vec{v_2}\| = \|\vec{v_1}\|\|\vec{v_2}\| \sin\beta$$

$$\|P_1Q\| = \frac{\|\vec{v_2} \times \vec{P_2P_1}\|\|\vec{v_1}\|}{\|\vec{v_1} \times \vec{v_2}\|}$$

$$\vec{OQ} = \vec{OP_1} + \vec{P_1Q} = \vec{OP_1} + \frac{\|P_1Q\|}{\|\vec{v_1}\|}\vec{v_1} = \vec{OP_1} + \frac{\|\vec{v_2} \times \vec{P_2P_1}\|}{\|\vec{v_1} \times \vec{v_2}\|}\vec{v_1}$$

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

- 判断射线与线段是否相交

```
// 射线 r 与线段 s 是否相交
inline bool r_to_s(const Line &r, const Seg &s) {
    Vector d = r.v; // 射线方向
    Vector v = s.b - s.a; // 线段方向
    Vector w = r.p - s.a; // 射线起点到线段起点向量

    db denom = cross(v, d);
    if (fabs(denom) < eps) return false;

    db t = cross(w, v) / denom;
    db u = cross(w, d) / denom;

    return t >= -eps && u >= -eps && u <= 1 + eps;
}
```

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) $S=\frac{1}{2}\|\sum_{i=0}^{n-1}\vec{OP_I}\times\vec{OP_{(i+1)\,mod\,n}}\|$
- 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则点在多边形外 (优秀)

```

```

圆

模版

点(Point)

```cpp
template<typename T>
struct Point {
    T x, y;

    bool operator==(const Point &a) const { return (abs(x - a.x) <= eps && abs(y - a.y) <= eps); }
    Point operator+(const Point &a) const { return {x + a.x, y + a.y}; }
    Point operator-(const Point &a) const { return {x - a.x, y - a.y}; }
    Point operator-() const { return {-x, -y}; }
    Point operator*(const T k) const { return {k * x, k * y}; }
    Point operator/(const T k) const { return {x / k, y / k}; }
    T operator*(const Point &a) const { return x * a.x + y * a.y; } // Dot
    T operator^(const Point &a) const { return x * a.y - y * a.x; } // Cross
    bool operator<(const Point &a) const {
        if (abs(x - a.x) <= eps) return y < a.y - eps;
        return x < a.x - eps;
    }

    bool is_par(const Point &a) const { return abs((*this) ^ a) <= eps; } // 平行
    bool is_ver(const Point &a) const { return abs((*this) * a) <= eps; } // 垂直

    int toleft(const Point &a) const {
        auto t = (*this) ^ a;
        return (t > eps) - (t < -eps);
    }

    T len2() const { return (*this) * (*this); }
    T dis2(const Point &a) const { return (a - (*this)).len2(); }
    double len() const { return sqrt(len2()); }
    double dis(const Point &a) const { return (a - (*this)).len(); }
    double ang(const Point &a) const { return acos(((*this) * a) / (this->len() * a.len())); }
    Point rot(const double rad) const { return {x * cos(rad) - y * sin(rad), x * sin(rad) + y * cos(rad)}; }
};
```

线(Line)

```cpp
template<typename T>
struct Line {
    Point<T> p, v; //p+kv

    bool operator==(const Line &a) const { return v.is_par(a.v) && v.is_par(p - a.p); }
```

```
    bool is_par(const Line &a) const { return v.is_par(a.v) && !v.is_par(p - a.p); } // 排除共线
    bool is_ver(const Line &a) const { return v.is_ver(a.v); }
    bool is_on(const Point<T> &a) const { return v.is_par(a - p); }
    int toleft(const Point<T> &a) const { return v.toleft(a - p); }
    Point<T> inter(const Line &a) const { return p + v * ((a.v ^ (p - a.p)) / (v ^ a.v)); }
    double dis(const Point<T> &a) const { return abs(v ^ (a - p)) / v.len(); }
    Point<T> proj(const Point<T> &a) const { return p + v * ((v * (a - p)) / (v * v)); }

    bool operator<(const Line &a) const {
        if (abs(v ^ a.v) <= eps && v * a.v >= -eps) return toleft(a.p) == -1;
        return argcmp(v, a.v);
    }
};
```

多边形(Polygon)

```
template<typename T>
struct Polygon {
    vector<Point<T> > p;

    size_t nxt(const size_t i) const { return i == p.size() - 1 ? 0 : i + 1; }
    size_t pre(const size_t i) const { return i == 0 ? p.size() - 1 : i - 1; }
};
```

计算一条直线在一个多边形内的最长直线

```
int n;
Polygon<db> poly;
set<pair<Point<db>, Point<db> > > edges;

template<typename T>
db calc(const Line<T> &l) {
    vector<tuple<Point<db>, Point<db>, Point<db> > > vec;
    for (int i = 0; i < n; i++) {
        auto u = poly.p[i], v = poly.p[poly.nxt(i)];
        int c1 = l.toleft(u), c2 = l.toleft(v);
        if (c1 * c2 <= 0) {
            if (c1 == 0 && c2 == 0) {
                vec.emplace_back(u, u, v);
                vec.emplace_back(v, u, v);
            } else {
                auto s = l.inter({u, u - v});
                vec.emplace_back(s, u, v);
            }
        }
    }
```

```
    sort(vec.begin(), vec.end());
    int cnt = 0;
    Point pre = {1e12, 1e12};
    db len = 0, maxlen = 0;
    while (!vec.empty()) {
        auto [now,u,v] = vec.back();
        if (cnt || edges.count({now, pre})) {
            len += now.dis(pre);
        } else {
            maxlen = max(maxlen, len);
            len = 0;
        }
        while (!vec.empty() && get<0>(vec.back()) == now) {
            auto [p,u,v] = vec.back();
            vec.pop_back();
            if (l.toleft(u) == -1) cnt++;
            else if (l.toleft(v) == -1) cnt--;
        }
        pre = now;
    }
    return max(maxlen, len);
}
```