

# Ponto 2D / Vetor

---

Representação de pontos e vetores como pontos.

## Estrutura

```
const double EPS = 1e-9;
const double PI = acos(-1);

struct point{
    double x, y;

    point(double _x, double _y) : x(_x), y(_y) {} ;
    point(){ x=y=0.0; };

    // Operadores
    bool operator <( point other ) const {
        if(fabs( x-other.x )>EPS) return x<other.x;
        else return y < other.y;
    }
    bool operator ==(point other) const {
        return fabs(x-other.x) < EPS && fabs(y-other.y)<EPS;
    }
    point operator +(point other) const {
        return point( x+other.x, y+other.y);
    }
    point operator -(point other) const {
        return point( x-other.x, y-other.y );
    }
    point operator *(double k) const {
        return point(x*k, y*k);
    }

    // Norma do vetor em relação a origem
    double norm() { return hypot(x, y); }

    // Vetor normalizado
    point normalized(){ return point(x, y)*(1.0/norm()); }

    // Angulo do vetor com a origem
    double angle() { return atan2(y, x); }

    // Angulo polar em relação a origem
    double polarAngle(){
        double a = atan2(y, x);
        return a < 0 ? a + 2*PI : a;
    }
};
```

## Funções

```
// Distância entre 2 pontos
double dist(point p1, point p2){
    return hypot(p1.x-p2.x, p1.y-p2.y);
}

// Produto interno
double inner(point p1, point p2){
    return p1.x*p2.x+p1.y*p2.y;
}

// Produto vetorial entre 2 pontos (apenas a componente z)
double cross(point p1, point p2){
    return p1.x*p2.y-p1.y*p2.x;
}

// Teste se os pontos estão no sentido antihorário (DISCRIMINANTE)
bool ccw(point p, point q, point r){
    return cross(q-p, r-p)>0;
}

// Teste de colinearidade entre os pontos
bool collinear(point p, point q, point r){
    return fabs(cross(q-p, r-p))<EPS;
}

// Rotação do ponto em relação a origem
point rotate(point p, double rad){
    return point(p.x*cos(rad)-p.y*sin(rad),
                p.x*sin(rad)+p.y*cos(rad));
}

// Angulo formado entre vetores a e b com o de origem
double angle(point a, point o, point b){
    return acos(inner(a-o, b-o)/(dist(o, a)*dist(o,b)));
}

// Ponto q está dentro no segmento p r
bool between(point p, point q, point r){
    return collinear(p, q, r) && inner(p-q, r-q)<=0;
}

// Ponto formado pela intersecção das retas a b e A B
point lineIntersectSeg(point p, point q, point A, point B){
    double c = cross(A-B, p-q);
    double a = cross(A, B);
    double b = cross(p, q);
    return ( (p-q)*(a/c) ) - ( (A-B)*(b/c) );
}
```

```
// Teste de paralelidade
bool parallel(point a, point b){
    return fabs(cross(a, b))<EPS;
}

// Verifica se segmentos a b e p q se interceptam
bool segIntersects(point a, point b, point p, point q){
    if(parallel(a-b, p-q)){
        return between(a, p, q) || between(a, q, b) ||
            between(p, a, q) || between(p, b, q);
    }
    point i = lineIntersectSeg(a, b, p, q);
    return between(a, i, b) && between(p, i, q);
}

// Ponto mais próximo entre p e o segmento de reta a b
point closetToSegment(point p, point a, point b){
    double u = inner(p-a, b-a)/inner(b-a, b-a);
    if(u < 0.0) return a;
    if(u > 1.0) return b;
    return a+((b-a)*u);
}
```

# Polígono 2D

---

O polígono é representado por um vetor de pontos portanto não tem estrutura.

## Funções

```
using polygon = vector<point>;
const double PI = acos(-1);

// Área do polígono, com sinal
double signedArea(polygon &P){
    double result = 0.0;
    int n = P.size();
    for(int i=0; i<n; i++){
        result+=cross(P[i], P[(i+1)%n]);
    }
    return result/2.0;
}

// Área positiva do polígono
double area(polygon &P){
    return fabs(signedArea(P));
}

// Índice do ponto mais a esquerda do vetor de pontos
int leftmostIndex(vector<point>& P){
    int ans = 0;
    for(int i=1; i<int(P.size()); i++){
        if (P[i]<P[ans]) ans = i;
    }
    return ans;
}

// Retorna o polígono sem cruzamento de pontos (Eu acho kkk)
polygon make_polygon(vector<point> P){
    if(signedArea(P)<0.0)
        reverse(P.begin(), P.end());

    int li = leftmostIndex(P);
    rotate(P.begin(), P.begin()+li, P.end());
    return P;
}

// Perímetro do pol- distígono
double perimeter(polygon &P){
    double result = 0.0;
    int n = P.size();
    for(int i=0; i<n; i++) result+=dist(P[i], P[(i+1)%n]);
    return result;
}
```

```

// Verifica se polígono é convexo
bool isConvex(polygon& P){
    int n = P.size();
    if( n<3 ) return false;
    bool left = ccw(P[0], P[1], P[2]);
    for(int i=1; i<n; i++){
        if(ccw(P[i], P[(i+1)%n], P[(i+2)%n]) != left)
            return false;
    }
    return true;
}

// Verifica se um ponto está dentro do polígono
bool inPolytgon(polygon &P, point p){
    if (P.size() == 0u) return false;
    double sum = 0.0;
    int n = P.size();
    for(int i=0; i<n; i++){
        if(P[i] == p || between(P[i], p, P[(i+1)%n]))
            return true;
        if(ccw(p, P[i], P[(i+1)%n]))
            sum+=angle(P[i], p, P[(i+1)%n]);
        else
            sum-=angle(P[i], p, P[(i+1)%n]);
    }
    return fabs(fabs(sum)-2*PI) < EPS;
}

// Polígono a direita que é formado pelo corte do polígono P pela reta
// formada pelos pontos a e b
polygon cutPolygon(polygon &P, point a, point b){
    vector<point> R;
    double left1, left2;
    int n = P.size();
    for(int i=0; i<n; i++){
        left1 = cross(b-a, P[i]-a);
        left2 = cross(b-a, P[(i+1)%n]-a);
        if (left1 > -EPS) R.push_back(P[i]);
        if (left1 * left2 < -EPS)
            R.push_back(lineIntersectSeg(P[i], P[(i+1)%n], a, b));
    }
    return make_polygon(R);
}

```

## Convex Hull

Dado um conjunto de pontos retorna o polígono que contém todos os pontos em  $O(n \log n)$ . Caso precise considerar os pontos no meio de uma aresta trocar ccw para  $\geq 0$ . CUIDADO: Se todos os pontos forem colineares, vai dar RTE.

- Pré requisitos struct point
  - construtores
  - operadores
- Pré requisitos funções point
  - dist
  - inner
  - cross
  - ccw
  - collinear
- Pré requisitos funções polygon
  - leftmostIndex

```
point pivot(0, 0);

bool angleCmp(point a, point b){
    if(collinear(pivot, a, b))
        return inner(pivot-a, pivot-a) < inner(pivot-b, pivot-b);
    return cross(a-pivot, b-pivot) >=0;
}

polygon convexHull(vector<point> P){
    int i, j, n = P.size();
    if( n<=2 ) return P;
    int P0 = leftmostIndex(P);
    swap(P[0], P[P0]);
    pivot = P[0];
    sort(++P.begin(), P.end(), angleCmp);
    vector<point> S;
    S.push_back(P[n-1]);
    S.push_back(P[0]);
    S.push_back(P[1]);
    for(i=2; i<n;){
        j=int(S.size()-1);
        if(ccw(S[j-1], S[j], P[i]))
            S.push_back(P[i++]);
        else S.pop_back();
    }
    reverse(S.begin(), S.end());
    S.pop_back();
    reverse(S.begin(), S.end());
    return S;
}
```

## Convex Hull (Algoritmo de inteiros)

Com esse algoritmo é possível conseguir o polígono que contém todos os pontos em  $O(n \log n)$  com todos os pontos consistindo como inteiros e não ponto flutuante. Caso precise considerar os pontos no meio de uma aresta trocar ccw para  $\geq 0$ ;

Nessa rotina é preciso reescrever a struct point para que ela trabalhe com **long long** e não **double**

```
using polygon = vector<point>;

polygon monotone_chain(const vector<point> points){
    vector<point> P(points);
    sort(P.begin(), P.end());

    vector<point> lower, upper;
    for(const auto& p: P){
        int n = int(lower.size());
        while(n>=2 and ccw(lower[n-2], lower[n-1], p)){
            lower.pop_back();
            n = int(lower.size());
        }
        lower.push_back(p);
    }
    reverse(P.begin(), P.end());
    for(const auto& p: P){
        int n = int(upper.size());
        while(n>=2 and ccw(upper[n-2], upper[n-1], p)){
            upper.pop_back();
            n = int(upper.size());
        }
        upper.push_back(p);
    }
    lower.pop_back();
    lower.insert(lower.end(), upper.begin(), upper.end());
    return lower;
}
```

```
struct point{
    long long x;
    long long y;
    point(long long _x, long long _y) : x(_x), y(_y) {};
    point(){ x=y=0; }
    bool operator<(point other) const {
        return x == other.x ? y<other.y : x<other.x;
    }
    point operator -(point other) const{
        return point(x-other.x, y-other.y);
    }
};
```

Além disso também é preciso reescrever algumas funções

```
long long cross(point p1, point p2){  
    return p1.x*p2.y-p1.y*p2.x;  
}  
// Discriminante  
bool ccw(point p, point q, point r){  
    return cross(q-p, r-p)>0;  
}
```