Geometria Computacional

Ponto 2D / Vetor

Representação de pontos e vetores como pontos.

Estrutura

Descrição

- **(double)** norm: Calcula norma do vetor em relação a origem
- **(double)** normalized: Retorna vetor normalizado (Norma 1)
- **(double)** angle: Angulo do vetor com a origem
- **(double)** polarAngle: Angulo polar em relação a origem

```
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; };

    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);
    }

    double norm() { return hypot(x, y); }
    point normalized(){ return point(x, y)*(1.0/norm()); }
    double angle() { return atan2(y, x); }
    double polarAngle(){
        double a = atan2(y, x);
        return a < 0 ? a + 2*PI : a;
    }
};
```

Funções

Descrição

- **(double)** dist: Distância entre 2 pontos
- **(double/int)** inner: Produto interno entre 2 vetores
- **(double/int)** cross: Produto vetorial entre 2 vetores (componente z)
- **(double/int)** ccw: Verifica se os pontos estão no sentido antihorário (DISCRIMINANTE)
- **(double/int)** collinear: Teste de colinearidade entre os pontos
- **(double)** rotate: Rotação do ponto em relação a origem
- **(double)** angle: Angulo formado entre vetores a e b com o de origem
- **(double/int)** between: Verifica se o ponto q está dentro no segmento p r
- **(double)** lineIntersectSeg: Retorna ponto formado pela intersecção das retas a b e A B
- **(double/int)** parallel: Teste de paralelidade
- **(double)** segIntersects: Verifica se segmentos a b e p q se interceptam
- **(double)** closetToSegment: Ponto mais próximo entre p e o segmento de reta a b

```
double dist(point p1, point p2){
    return hypot(p1.x-p2.x, p1.y-p2.y);
}
double inner(point p1, point p2){
    return p1.x*p2.x+p1.y*p2.y;
}
double cross(point p1, point p2){
    return p1.x*p2.y-p1.y*p2.x;
}
bool ccw(point p, point q, point r){
    return cross(q-p, r-p)>0;
}
bool collinear(point p, point q, point r){
    return fabs(cross(q-p, r-p))<EPS;
}
point rotate(point p, double rad){
    return point(p.x*cos(rad)-p.y*sin(rad),
                 p.x*sin(rad)+p.y*cos(rad));
}
double angle(point a, point o, point b){
    return acos(inner(a-o, b-o)/(dist(o, a)*dist(o,b)));
}
bool between(point p, point q, point r){
    return collinear(p, q, r) && inner(p-q, r-q)<=0;
}
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) );
}
bool parallel(point a, point b){ return fabs(cross(a, b))<EPS; }
```

```
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);
}
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);
}
```

Círculo 2D

Estrutura

Descrição

- **(double)** area: Calcula área do círculo
- **(double)** chord: Calcula o comprimento da corda com ângulo em radiandos
- **(double)** sector: Calula o comprimento do setor com ângulo radianos
- **(double)** intersects: Verifica se os círculos se interceptem
- **(double)** contains: Verifica se um ponto p está dentro do círculo
- **(double)** getTangentPoint: Retorna os 2 pontos que formam as retas com p tangentes ao círculo, (A função asin retorna nan se o ponto estiver dentro do círculo)
- **(double)** getIntersectionPoints: Retorna os 2 pontos de intersecção entre os círculos

```
const double PI = acos(-1);
const double EPS = 1e-9;
struct circle{
    point c;
    double r;

    circle() { c=point(); r=0; }
    circle(point _c, double _r) : c(_c), r(_r) {}

    double area() { return PI*r*r; }
    double chord (double rad){ return 2*r*sin(rad/2.0); }
    double sector (double rad) { return 0.5*rad*area()/PI; }
    bool intersects(circle other){
        return dist(c, other.c) < r+other.r;
    }
    bool contains(point p) { return dist(c, p) <= r + EPS; }
    pair<point, point> getTangentPoint( point p ){
        double d1 = dist(p, c), theta = asin(r/d1);
        point p1 = rotate(c-p, -theta);
        point p2 = rotate(c-p, theta);
        p1 = p1*(sqrt(d1*d1-r*r)/d1)+p;
        p2 = p2*(sqrt(d1*d1-r*r)/d1)+p;
        return make_pair(p1, p2);
    }
    pair<point, point> getIntersectionPoints(circle other){
        assert(intersects(other));
        double d = dist(c, other.c);
        double u = acos((other.r*other.r + d*d - r*r)/(2*other.r*d));
        point dc = ((other.c - c).normalized()) * r;
        return make_pair(c + rotate(dc, u), c + rotate(dc, -u));
    }
};
```

Funções

Descrição

- **(double)** insideCircle: Retorna 0 caso o ponto esteja dentro, 1 caso esteja na borda e 2 caso esteja fora do círculo
- **(double)** circumcircle: Círculo circunscrito no triângulo
- **(double)** incircle: Círculo inscrito do triângulo

```
int insideCircle(point p, circle c) {
    if (fabs(dist(p, c.c) - c.r)<EPS) return 1;
    else if(dist(p, c.c) < c.r) return 0;
    else return 2;
} // 0 = inside/ 1 = border/ 2 = outside
circle circumcircle(point a, point b, point c) {
    circle ans;
    point u = point((b-a).y, -(b-a).x);
    point v = point((c-a).y, -(c-a).x);
    point n = (c-b)*0.5;
    double t = cross(u, n)/cross(v, u);
    ans.c = ((a+c)*0.5) + (v*t);
    ans.r = dist(ans.c, a);
    return ans;
}
circle incircle( point p1, point p2, point p3){
    double m1 = dist(p2, p3);
    double m2 = dist(p1, p3);
    double m3 = dist(p1, p2);
    point c = (p1*m1+p2*m2+p3*m3)*(1/(m1+m2+m3));
    double s = 0.5*(m1+m2+m3);
    double r = sqrt(s*(s-m1)*(s-m2)*(s-m3))/s;
    return circle(c, r);
}
```

Triângulo 2D

Estrutura

Descrição

- **(double)** perimeter: Calcula perímetro do triângulo
- **(double)** semiPerimeter: Calcula semi-perímetro do triângulo
- **(double)** area: Calcula área do triângulo
- **(double)** rInCircle: Calcula o raio do círculo inscrito no triângulo
- **(double)** inCircle: Retorna o círculo inscrito do triângulo
- **(double)** rCircumCircle: Calcula raio do círculo circunscrito no triângulo
- **(double)** circumCircle: Retorna o círculo circunscrito do triângulo
- **(double/int)** isInside: Retorna 0 caso o ponto esteja dentro do triângulo, 1 caso esteja na borda e 2 caso esteja fora

```
struct triangle{
    point a, b, c;

    triangle() { a=b=c=point(); }
    triangle(point _a, point _b, point _c) : a(_a), b(_b), c(_c) {}

    double perimeter() { return dist(a, b) + dist(b, c) + dist(c, a); }
    double semiPerimeter() { return perimeter()/2.0; }
    double area(){
        double s = semiPerimeter(), ab = dist(a, b), bc = dist(b, c), ca =
dist(c, a);
        return sqrt(s*(s-ab)*(s-bc)*(s-ca));
    }
    double rInCircle() {
        return area()/semiPerimeter();
    }
    circle inCircle() { return incircle(a, b, c); }
    double rCircumCircle() {
        return dist(a, b)*dist(b, c)*dist(c, a)/(4.0*area());
    }
    circle circumCircle(){ return circumcircle(a, b, c); }
    bool isInside(point p){
        double u = cross(b-a, p-a)*cross(b-a, c-a);
        double v = cross(c-b, p-b)*cross(c-b, a-b);
        double w = cross(a-c, p-c)*cross(a-c, b-c);
        if (u>0.0 and v>0.0 and w>0.0) return 0;
        if (u<0.0 or v<0.0 or w<0.0) return 2;
        else return 1;
    }// 0 = inside/ 1 = border/ 2=outside
};
```

Polígono 2D

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

Funções

Descrição

- **(double/int)** signedArea: Retorna área do polígono (Pode dar valor negativo)
- **(double/int)** area: Retorna área positiva do polígono
- **(double/int)** leftmostIndex: Índice do ponto mais a esquerda
- **(double/int)** make_polygon: Coloca o índice do ponto mais a esquerda como 0 (Necessário para as funções que chamam polygon)
- **(double)** perimeter: Calcula o perímetro do polígono
- **(double/int)** isConvex: Verifica se polígono é convexo
- **(double)** inPolytgon: Verifica se um ponto está no polígono
- **(double)** cutPolygon: Polígono a direita que é formado pelo corte do polígono P pela reta formada pelos pontos a e b

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

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;
}
double area(polygon &P){
    return fabs(signedArea(P));
}
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;
}
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;
}
```

```
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;
}
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;
}
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;
}
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 presice consederar os pontos no meio de uma aresta trocar ccw para >=0. CUIDADO: Se todos os pontos forem colineares, vai dar RTE.

```
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;
}
```

Monotone chain

- Com esse algorítimo 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 presice consederar os pontos no meio de uma aresta trocar ccw para >=0**
- Nessa rotina é preciso usar a struct point como **long long.**

```
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()-1);
    return lower;
}
```