Pontificia Universidad Católica del Perú - FCI

XieXieLucas Notebook - Froz/Phibrain/Ands

November 9, 2017

Contents

1 Convex Hull

```
//   INPUT:  a vector of input points, unordered.
//   OUTPUT: a vector of points in the convex hull,
//           counterclockwise, starting with bottom left

#define REMOVE_REDUNDANT

typedef double T;
const T EPS = 1e-7;
struct PT {
	T x, y;
	PT() {}
	PT(T x, T y) : x(x), y(y) {}
	bool operator<(const PT &rhs) const {
		return mp(y,x) < mp(rhs.y,rhs.x);
	}
	bool operator==(const PT &rhs) const {
		return mp(y,x) == mp(rhs.y,rhs.x);
	}
};
T cross(PT p, PT q) { return p.x*q.y-p.y*q.x; }
T area2(PT a, PT b, PT c) {
	return cross(a,b) + cross(b,c) + cross(c,a);
}

#ifdef REMOVE_REDUNDANT
bool between(const PT &a, const PT &b, const PT &c) {
  return (fabs(area2(a,b,c)) < EPS &&
                     (a.x-b.x)*(c.x-b.x) <= 0 &&
                     (a.y-b.y)*(c.y-b.y) <= 0);
}
#endif

void ConvexHull(vector<PT> &pts) {
	sort(pts.begin(), pts.end());
	pts.erase(unique(pts.begin(), pts.end()), pts.end());
	vector<PT> up, dn;
	for (int i = 0; i < pts.size(); i++) {
		while (up.size() > 1 &&
			area2(up[up.size()-2], up.back(), pts[i]) >= 0)
				up.pop_back();
		while (dn.size() > 1 &&
			area2(dn[dn.size()-2], dn.back(), pts[i]) <= 0)
				dn.pop_back();
		up.push_back(pts[i]);
		dn.push_back(pts[i]);
	}
	pts = dn;
	for (int i = (int) up.size() - 2; i >= 1; i--)
		pts.push_back(up[i]);

#ifdef REMOVE_REDUNDANT
	if (pts.size() <= 2) return;
	dn.clear();
```

```
        dn.push_back(pts[0]);
        dn.push_back(pts[1]);
        for (int i = 2; i < pts.size(); i++) {
                if (between(dn[dn.size()-2], dn[dn.size()-1], pts[i]))
                        dn.pop_back();
                dn.push_back(pts[i]);
        }
        if (dn.size() >= 3 && between(dn.back(), dn[0], dn[1])) {
                dn[0] = dn.back();
                dn.pop_back();
        }
  pts = dn;
#endif
}
```

2 Delaunay Triangulation

```
// Slow but simple Delaunay triangulation. Does not handle
// degenerate cases (from O'Rourke, Computational Geometry in C)
//
// Running time: O(n^4)
//
// INPUT:   x[] = x-coordinates
//          y[] = y-coordinates
//
// OUTPUT:  triples = a vector containing m triples of indices
//                    corresponding to triangle vertices

#include<vector>
using namespace std;

typedef double T;

struct triple {
    int i, j, k;
    triple() {}
    triple(int i, int j, int k) : i(i), j(j), k(k) {}
};

vector<triple> delaunayTriangulation(vector<T>& x, vector<T>& y) {
        int n = x.size();
        vector<T> z(n);
        vector<triple> ret;

        for (int i = 0; i < n; i++)
            z[i] = x[i] * x[i] + y[i] * y[i];

        for (int i = 0; i < n-2; i++) {
            for (int j = i+1; j < n; j++) {
                for (int k = i+1; k < n; k++) {
                    if (j == k) continue;
                    double xn = (y[j]-y[i])*(z[k]-z[i]) - (y[k]-y[i])*(z[j]-
                        z[i]);
                    double yn = (x[k]-x[i])*(z[j]-z[i]) - (x[j]-x[i])*(z[k]-
                        z[i]);
                    double zn = (x[j]-x[i])*(y[k]-y[i]) - (x[k]-x[i])*(y[j]-
                        y[i]);
                    bool flag = zn < 0;
                    for (int m = 0; flag && m < n; m++)
                        flag = flag && ((x[m]-x[i])*xn +
                                        (y[m]-y[i])*yn +
                                        (z[m]-z[i])*zn <= 0);
                    if (flag) ret.push_back(triple(i, j, k));
                }
            }
        }
        return ret;
}

int main(){
    T xs[]={0, 0, 1, 0.9};
    T ys[]={0, 1, 0, 0.9};
    vector<T> x(&xs[0], &xs[4]), y(&ys[0], &ys[4]);
    vector<triple> tri = delaunayTriangulation(x, y);

    //expected: 0 1 3
    //          0 3 2

    int i;
    for(i = 0; i < tri.size(); i++)
        printf("%d %d %d\n", tri[i].i, tri[i].j, tri[i].k);
    return 0;
}
```

3 Geometry

```
// C++ routines for computational geometry.

const double INF = 1e100;
const double EPS = 1e-12;
const double PI = acos(-1);

struct PT {
  double x, y;
  PT() {}
  PT(double x, double y) : x(x), y(y) {}
  PT(const PT &p) : x(p.x), y(p.y) {}
  PT operator + (const PT &p) const { return PT(x+p.x, y+p.y); }
  PT operator - (const PT &p) const { return PT(x-p.x, y-p.y); }
  PT operator * (double c)  const { return PT(x*c, y*c ); }
  PT operator / (double c)  const { return PT(x/c, y/c ); }
};

double dot(PT p, PT q)  { return p.x*q.x+p.y*q.y; }
double norm(PT p) { return sqrt(dot(p,p)); }
double dist2(PT p, PT q) { return dot(p-q,p-q); }
double cross(PT p, PT q) { return p.x*q.y-p.y*q.x; }
ostream &operator<<(ostream &os, const PT &p) {
  os << "(" << p.x << "," << p.y << ")";
}

double angle(PT p){
    double res = acos(p.x / norm(p));
    if (p.y > 0) return res;
    else return 2*PI - res;
}

// rotate a point CCW or CW around the origin
PT RotateCCW90(PT p) { return PT(-p.y,p.x); }
PT RotateCW90(PT p)  { return PT(p.y,-p.x); }
PT RotateCCW(PT p, double t) {
  return PT(p.x*cos(t)-p.y*sin(t), p.x*sin(t)+p.y*cos(t));
}

// project point c onto line through a and b
// assuming a != b
PT ProjectPointLine(PT a, PT b, PT c) {
  return a + (b-a)*dot(c-a, b-a)/dot(b-a, b-a);
}

// project point c onto line segment through a and b
PT ProjectPointSegment(PT a, PT b, PT c) {
  double r = dot(b-a,b-a);
  if (fabs(r) < EPS) return a;
  r = dot(c-a, b-a)/r;
  if (r < 0) return a;
  if (r > 1) return b;
  return a + (b-a)*r;
}

// distance from c to segment between a and b
double DistancePointSegment(PT a, PT b, PT c) {
  return sqrt(dist2(c, ProjectPointSegment(a, b, c)));
}

// distance between point (x,y,z) and plane ax+by+cz=d
double DistancePointPlane(double x, double y, double z,
                double a, double b, double c, double d)
{
  return fabs(a*x+b*y+c*z-d)/sqrt(a*a+b*b+c*c);
}

// Whethes lines (a,b), (c,d) are parallel/collinear
bool LinesParallel(PT a, PT b, PT c, PT d) {
  return fabs(cross(b-a, c-d)) < EPS;
}
bool LinesCollinear(PT a, PT b, PT c, PT d) {
  return LinesParallel(a, b, c, d)
      && fabs(cross(a-b, a-c)) < EPS
      && fabs(cross(c-d, c-a)) < EPS;
}

// determine if line segment from a to b intersects with
// line segment from c to d
bool SegmentsIntersect(PT a, PT b, PT c, PT d) {
        if (LinesCollinear(a, b, c, d)) {
                if (dist2(a, c) < EPS || dist2(a, d) < EPS ||
                        dist2(b, c) < EPS || dist2(b, d) < EPS) return true;
                if (dot(c-a, c-b) > 0 && dot(d-a, d-b) > 0 &&
                         dot(c-b, d-b) > 0) return false;
                return true;
  }
  if (cross(d-a, b-a) * cross(c-a, b-a) > 0) return false;
```

```
  if (cross(a-c, d-c) * cross(b-c, d-c) > 0) return false;
  return true;
}

// compute intersection of line passing through a and b
// with line passing through c and d, assuming that unique
// intersection exists; for segment intersection, check if
// segments intersect first
PT ComputeLineIntersection(PT a, PT b, PT c, PT d) {
  b=b-a; d=c-d; c=c-a;
  assert(dot(b, b) > EPS && dot(d, d) > EPS);
  return a + b*cross(c, d)/cross(b, d);
}

// compute center of circle given three points
PT ComputeCircleCenter(PT a, PT b, PT c) {
  b=(a+b)/2;
  c=(a+c)/2;
  return ComputeLineIntersection(b, b+RotateCW90(a-b), c, c+RotateCW90(a-c
      ));
}

// determine if point is in a possibly non-convex polygon
// (by William Randolph Franklin); returns 1 for strictly
// interior points, 0 for strictly exterior points, and 0 or 1
// for the remaining points. Note that it is possible to
// convert this into an *exact* test using integer arithmetic
// by taking care of the division appropriately (making sure
// to deal with signs properly) and then by writing exact
// tests for checking point on polygon boundary
bool PointInPolygon(const vector<PT> &p, PT q) {
  bool c = 0;
  for (int i = 0; i < p.size(); i++){
    int j = (i+1)%p.size();
    if ((p[i].y <= q.y && q.y < p[j].y ||
      p[j].y <= q.y && q.y < p[i].y) &&
      q.x < p[i].x + (p[j].x - p[i].x) *(q.y - p[i].y) /
                (p[j].y - p[i].y))
      c = !c;
  }
  return c;
}

// determine if point is on the boundary of a polygon
bool PointOnPolygon(const vector<PT> &p, PT q) {
  for (int i = 0; i < p.size(); i++)
    if (dist2(ProjectPointSegment(p[i], p[(i+1)%p.size()], q), q) < EPS)
      return true;
    return false;
}

// compute intersection of line through points a and b with
// circle centered at c with radius r > 0
vector<PT> CircleLineIntersection(PT a, PT b, PT c, double r) {
  vector<PT> ret;
  b = b-a;
  a = a-c;
  double A = dot(b, b);
  double B = dot(a, b);
  double C = dot(a, a) - r*r;
  double D = B*B - A*C;
  if (D < -EPS) return ret;
  ret.push_back(c+a+b*(-B+sqrt(D+EPS))/A);
  if (D > EPS)
    ret.push_back(c+a+b*(-B-sqrt(D))/A);
  return ret;
}

// compute intersection of circle centered at a with radius r
// with circle centered at b with radius R
vector<PT> CircleCircleInter(PT a, PT b, double r, double R) {
  vector<PT> ret;
  double d = sqrt(dist2(a, b));
  if (d > r+R || d+min(r, R) < max(r, R)) return ret;
  double x = (d*d-R*R+r*r)/(2*d);
  double y = sqrt(r*r-x*x);
  PT v = (b-a)/d;
  ret.push_back(a+v*x + RotateCCW90(v)*y);
  if (y > 0)
    ret.push_back(a+v*x - RotateCCW90(v)*y);
  return ret;
}

// Area or centroid of a (possibly nonconvex) polygon,
// assuming the coordinates are listed in a clockwise or
// counterclockwise order. Note that the centroid is often
// known as the "center of gravity" or "center of mass".
double ComputeSignedArea(const vector<PT> &p) {
  double area = 0;
  for(int i = 0; i < p.size(); i++) {
```

```
    int j = (i+1) % p.size();
    area += p[i].x*p[j].y - p[j].x*p[i].y;
  }
  return area / 2.0;
}

double ComputeArea(const vector<PT> &p) {
  return fabs(ComputeSignedArea(p));
}

PT ComputeCentroid(const vector<PT> &p) {
  PT c(0,0);
  double scale = 6.0 * ComputeSignedArea(p);
  for (int i = 0; i < p.size(); i++){
    int j = (i+1) % p.size();
    c = c + (p[i]+p[j])*(p[i].x*p[j].y - p[j].x*p[i].y);
  }
  return c / scale;
}

// Whether or not a given (CW or CCW) polygon is simple
bool IsSimple(const vector<PT> &p) {
  for (int i = 0; i < p.size(); i++) {
    for (int k = i+1; k < p.size(); k++) {
      int j = (i+1) % p.size();
      int l = (k+1) % p.size();
      if (i == l || j == k) continue;
      if (SegmentsIntersect(p[i], p[j], p[k], p[l]))
        return false;
    }
  }
  return true;
}

// Computes the circumcenter of a Triangle PQR
PT circumcenter(PT p, PT q, PT r) {
        PT a = p-r, b = q-r;
        PT c = PT(dot(a, (p + r)) / 2, dot(b, (q + r)) / 2);
        return PT(dot(c, RotateCW90(PT(a.y, b.y))),
                dot(PT(a.x, b.x), RotateCW90(c))) / dot(a, RotateCW90(b));
}


//Check if a polygon is convex
bool isConvex(const vector<point> &P) {
        int sz = (int)P.size();
        if (sz <= 3) return false;
        bool isLeft = ccw(P[0], P[1], P[2]);
        for (int i = 1; i < sz-1; i++)
                if (ccw(P[i], P[i+1], P[(i+2) == sz ? 1 : i+2]) != isLeft)
                    return false;
        return true;
}
```

4 Minkowski Sum

```
//Calcula suma de Minkowski en O(n + m)
//A y B deben estar en sentido antihorario
inline bool compare(PT a, PT b){
        // mas abajo, mas a la izquierda
        if(a.y < b.y) return 1;
        if(a.y == b.y) return a.x < b.x;
        return 0;
}

vector<PT> minkow_sum(const vector<PT>& a, const vector<PT>& b){
        vector< PT > out;
        out.clear();
        int lena = int(a.size());
        int lenb = int(b.size());
        int i = 0, j = 0;
        for(int q = 0; q < lena; ++q) if(compare(a[q], a[i])) i = q;
        for(int q = 0; q < lenb; ++q) if(compare(b[q], b[j])) j = q;
        ll pr;
        int nxti, nxtj;
        do{
                out.pb(a[i] + b[j]);
                nxti = (i + 1) % lena;
                nxtj = (j + 1) % lenb;
                pr = cross(a[nxti] - a[i], b[nxtj] - b[j]);
                if(pr > 0) i = nxti;
                else if(pr < 0) j = nxtj;
                else i = nxti, j = nxtj; // paralelas, subo en ambas
        }while((a[i] + b[j]) != out[0]);
        return out;
}
```