Contents

1 Util **2**
- 1.1 Comparison of floating point numbers 2
- 1.2 Point Class 2
- 1.3 Quadrant 3
- 1.4 Comparators 3
  - 1.4.1 Radial Comparator 4
  - 1.4.2 Polar Comparator 4

2 Distance **4**
- 2.1 Point to point 4
- 2.2 Point to line (or segment) 5

3 Angles **5**
- 3.1 Angle between vectors 5
- 3.2 Turn between vectors 5
- 3.3 Point in segment 6

4 Polygons **6**
- 4.1 Point in polygon 6
- 4.2 Convex Hull 7
- 4.3 Area of polygon 7
- 4.4 Intersection of Convex polygons 8
  - 4.4.1 Field test: Polygons 9

5 Lines **13**
- 5.1 Point of intersection 13
- 5.2 Segment intersection 13

6 Circles **14**
- 6.1 Diameter of the circumcircle of a triangle 14
- 6.2 Circumcenter of a triangle (Circle from three points) 14
- 6.3 Minimum spanning circle 14

1 Util

Functions and definitions necessary to use the presented algorithms. Whenever possible, methods will be presented as implementations of prototype functions that were presented on a minimal viable code *(Is that a thing? This "minimal viable code"? I meant the minimal implementation necessary to obtain basic functionality).*

1.1 Comparison of floating point numbers

```cpp
const double EPS = 1e-10;

inline int cmp(double x, double y = 0, double tol = EPS) {
    return (x <= y + tol) ? (x + tol < y) ? -1 : 0 : 1;
}
```

1.2 Point Class

Minimal Point Class definition for multiple purposes.
Requires 1.1:`cmp` and `iostream`.

```cpp
class Point {
 public:
  Point(double x_ = 0.0, double y_ = 0.0) : x(x_), y(y_) {}

  Point operator +(const Point &o) const { return Point(x + o.x, y + o.y); }
  Point operator -(const Point &o) const { return Point(x - o.x, y - o.y); }
  Point operator *(const double &m) const { return Point(m * x, m * y); }
  Point operator /(const double &m) const { return Point(x / m, y / m); }
  // Dot Product
  double operator *(const Point &o) const { return x * o.x + y * o.y; }
  // Cross Product
  double operator ^(const Point &o) const { return x * o.y - y * o.x; }

  int cmp(Point o) const {
    if (int t = ::cmp(x, o.x)) return t;
    return ::cmp(y, o.y);
  }
  bool operator ==(const Point &o) const { return cmp(o) == 0; }
  bool operator !=(const Point &o) const { return cmp(o) != 0; }
  bool operator < (const Point &o) const { return cmp(o) < 0; }

  //Not necessary for minimal Point declaration so they're declared as prototypes.
```

```
23    //Just to know they're there :3
24    double distance(const Point &o) const;
25    double distance(const Point &p1, const Point &p2, const bool &isSegment) const;
26
27    friend ostream& operator <<(ostream &o, Point p) {
28      return o << "(" << p.x << ", " << p.y << ")";
29    }
30
31    double x, y;
32    static Point pivot;
33  };
34
35  Point Point::pivot(0, 0);
36
37  double abs(Point p) {return hypot(p.x, p.y);}
38  double arg(Point p) {return atan2(p.y, p.x);}
```

1.3 Quadrant

Determine the quadrant where a point is positioned. The point $(0, 0)$ is classified as the fifth quadrant since it doesn't belong to any of them.
Requires 1.1:`cmp` and 1.2:`Point`.

```
1   int quadrant(const Point &p) {
2     if (::cmp(p.x) == 0 && ::cmp(p.y) == 0) return 5;
3     if (::cmp(p.y) == 1) {
4       if (::cmp(p.x) == 1) return 1;
5       return 2;
6     }
7     if (::cmp(p.y) == 0) {
8       if (::cmp(p.x) == 1 || ::cmp(p.x) == 0) return 1;
9       return 3;
10    }
11    if (::cmp(p.x) == -1) return 3;
12    return 4;
13  }
```

1.4 Comparators

Comparators to sort points to prepare them for use with more complex algorithms. To use them it is necessary to set the pivot of `Point` to the lowest (by operator $<$) in the vector. `*min_element(all(v))` is generally a good choice.

1.4.1 Radial Comparator

The points are sorted by the direction of the turn between them. If they're collinear, the distance to the origin is used to "break the tie".
Requires 1.1:`cmp` and 1.2:`Point`.

```
bool radial_comp(const Point &p, const Point &q) {
  Point P = p - Point::pivot, Q = q - Point::pivot;
  double R = P ^ Q;
  if (::cmp(R)) return R > 0;
  return ::cmp(P * P, Q * Q) < 0;
}
```

1.4.2 Polar Comparator

The radial comparator in 1.4.1 could fail if the point do not lie in the same quadrant. This comparator takes care of this by finding the quadrant first and using it to determine the order of the points (if they differ).
Requires 1.1:`cmp` and 1.2:`Point`.

```
bool polar_comp(const Point &p, const Point &q) {
  Point P = p - Point::pivot, Q = q - Point::pivot;
  int q1 = Quadrant(P), q2 = Quadrant(Q);
  if (q1 != q2) return q1 < q2;
  double R = P ^ Q;
  if (::cmp(R)) return R > 0;
  return ::cmp(P * P, Q * Q) < 0;
}
```

2 Distance

2.1 Point to point

Euclidean distance defined as $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$.
Requires 1.2:`Point` and `cmath`.

```
double Point :: distance(const Point &o) const{
    double d1 = x - o.x, d2 = y - o.y;
    return sqrt(d1 * d1 + d2 * d2);
}
```

2.2 Point to line (or segment)

Compute the distance between the point and the line specified by the two other points given. If the boolean parameter `is_segment` is `true` those two points are treated as the endpoints of a line segment.
Requires 2.1:`distance (p2p)`.

```
double Point :: distance(const Point &p1, const Point &p2,
           const bool &isSegment) const {
    double dist = ((p2 - p1) ^ (*this - p1)) / p2.distance(p1);
    if (isSegment) {
      double dot1 = (*this - p2) * (p2 - p1);
    if (::cmp(dot1) > 0) return sqrt((p2 - *this) * (p2 - *this));
      double dot2 = (*this - p1) * (p1 - p2);
    if (::cmp(dot2) > 0) return sqrt((p1 - *this) * (p1 - *this));
    }
    return abs(dist);
}
```

3 Angles

3.1 Angle between vectors

Compute the angle between the vectors defined by `p-r` and `q-r`. The formula comes from the definition of dot and cross products:

$$A \cdot B = |A||B| \cos(c)$$

$$A \times B = |A||B| \sin(c)$$

$$\frac{\sin(c)}{\cos(c)} = \frac{A \times B}{A \cdot B} = \tan(c)$$

Requires 1.2:`Point`

```
inline double angle(const Point &p, const Point &q, const Point &r) {
  Point u = p - r, v = q - r;
  return atan2(u ^ v, u * v);
}
```

3.2 Turn between vectors

Determine the sign of the turn between the vectors defined by `p-r` and `q-r`. The cross product is negative it the turn is to the right and positive if the turn is to the left.
Requires 1.2:`Point` and 1.1:`cmp`

```cpp
inline int turn(const Point &p, const Point &q, const Point &r) {
  return ::cmp((p - r) ^ (q - r));
}
```

3.3 Point in segment

Determine whether a given point `r` is inside the segment defined by `p` and `q`. If the point is inside the segment, two conditions must be met:

- The turn between the vectors `p-q` and `r-q` is zero (they are parallel).
- The dot product between the vector formed by `p-r` and `q-r` (the testing point as the initial point for both vectors) is less than or equal to 0 (meaning the two vectors have opposite direction).

Requires 1.1:`cmp`, 1.2:`Point`, 3.2:`turn`

```cpp
inline bool between(const Point &p, const Point &q, const Point &r) {
  return turn(p, r, q) == 0 && ::cmp((p - r) * (q - r)) <= 0;
}
```

4 Polygons

4.1 Point in polygon

Determine whether a given point `r` is positioned outside, on the frontier or inside (`0,-1,1`) respectively of the given polygon. ***The polygon must be ordered clockwise or counter-clockwise for this algorithm to work!.*** The underlying idea is to take the segment formed by each pair of adjacent points belonging to the polygon and determine if the given point is inside this segment (If so, a `-1` would be returned), if not, the angles formed by the three points are added together. For a point outside the polygon, this sum is zero because the angles cancel themselves.
Requires 1.1:`cmp`, 1.2:`Point`, 3.1:`angle`, 3.3:`between` and `vector`.

```cpp
/*Remember that THE FUCKING POLYGON SHOULD BE ORDERED CLOCKWISE
 OR COUNTER-CLOCKWISE FOR THIS ALGORITHM TO WORK.
*/
int in_polygon(const Point &p, const vector<Point> &T) {
  double a = 0; int N = T.size();
  for (int i = 0; i < N; ++i) {
    if (between(T[i], T[(i + 1) % N], p)) return -1;
    a += angle(T[i], T[(i + 1) % N], p);
  }
```

```
  return ::cmp(a) != 0;
}
```

4.2 Convex Hull

Requires 1.2:`Point`, 3.2:`turn`, 1.4.2:`polar_comp`, `vector` and `algorithm`.

```
vector <Point> convex_hull(vector <Point> T) {
  int j = 0, k, n = T.size(); vector <Point> U(n);
  Point::pivot = *min_element(T.begin(), T.end());
  sort(T.begin(), T.end(), RadialComp);
  for (k = n - 2; k >= 0 && turn(T[0], T[n - 1], T[k]) == 0; --k);
  reverse((k + 1) + T.begin(), T.end());
  for (int i = 0; i < n; ++i) {
    // Change >= for > to keep the colinear points.
    while (j > 1 && turn(U[j - 1], U[j - 2], T[i]) > 0) --j;
    U[j++] = T[i];
  }
  U.erase(j + U.begin(), U.end());
  return U;
}
```

4.3 Area of polygon

Find the area of the given polygon (**must be ordered in clockwise or counter-clockwise fashion**) by triangulation.
Requires 1.2:`Point` and `vector`.

```
/*Is the polygon sorted clockwise or counter-clockwise?
  if not, I'll find you. I swear, I'll find you and fucking kill you.
*/

double area(const vector<Point> &T) {
  double area = 0.0;
  for(int i = 1; i + 1 < T.size(); i++){
    area += (T[i] - T[0]) ^ (T[i + 1] - T[0]);
  }
  return abs(area / 2.0);
}
```

4.4 Intersection of Convex polygons

Return the polygon formed at the intersection of two **convex** polygons. Important: Remember to sort the points before attempting to find the intersection, use 1.4.1 `radial_comp`. See the field test for more information.

```
/*
 * Points in both polygons should be ordered clockwise.
 * Remember to use radial_comp before using this method.
 */

polygon poly_intersect(polygon &P, polygon &Q){
    int m = Q.size(), n = P.size();
    int a = 0, b = 0, aa = 0, ba = 0, inflag = 0;
    polygon R;
    while( (aa < n || ba < m) && aa < 2*n && ba < 2*m){
        Point p1 = P[a], p2 = P[(a+1) % n], q1 = Q[b], q2 = Q[(b+1) % m];
        Point A = p2 - p1, B = q2 - q1;
        int cross = cmp(A ^ B), ha = turn(p2, q2, p1), hb = turn(q2, p2, q1);
        if(cross == 0 && turn(p1, q1, p2) == 0 && cmp(A * B) < 0){
            if(between(p1, q1, p2)) R.push_back(q1);
            if(between(p1, q2, p2)) R.push_back(q2);
            if(between(q1, p1, q2)) R.push_back(p1);
            if(between(q1, p2, q2)) R.push_back(p2);
            if(R.size() < 2) return polygon();
            inflag = 1; break;
        }else if(cross != 0 && seg_intersect(p1, p2, q1, q2)) {
            if(inflag == 0) aa = ba = 0;
            R.push_back(intersection(p1, p2, q1, q2));
            inflag = (hb > 0) ? 1 : -1;
        }
        if(cross == 0 && hb < 0 && ha < 0) return R;
        bool t = cross == 0 && hb == 0 && ha == 0;
        if(t ? (inflag == 1) : (cross >= 0) ? (ha <= 0) : (hb > 0)){
            if(inflag == -1) R.push_back(q2);
            ba++; b++; b %= m;
        }else{
            if(inflag == 1) R.push_back(p2);
            aa++; a++; a %= n;
        }
    }

    if(inflag == 0){
        if (in_polygon(P[0], Q)) return P;
```

```
        if (in_polygon(Q[0], P)) return Q;
    }
    R.erase(unique(R.begin(), R.end()), R.end());
    if(R.size() > 1 && R.front() == R.back()) R.pop_back();
    return R;
}
```

4.4.1 Field test: Polygons

Source: *UVA 137*
Given two convex polygons that may or not intersect, find the total area that does not belong to their intersection.

```
#include<cmath>
#include<vector>
#include<iostream>
#include<iomanip>
#include<algorithm>

using namespace std;

const double EPS = 1e-10;

inline int cmp(double x, double y = 0, double tol = EPS) {
    return (x <= y + tol) ? (x + tol < y) ? -1 : 0 : 1;
}

class Point {
 public:
  Point(double x_ = 0.0, double y_ = 0.0) : x(x_), y(y_) {}

  Point operator +(const Point &o) const { return Point(x + o.x, y + o.y); }
  Point operator -(const Point &o) const { return Point(x - o.x, y - o.y); }
  Point operator *(const double &m) const { return Point(m * x, m * y); }
  Point operator /(const double &m) const { return Point(x / m, y / m); }
  double operator *(const Point &o) const { return x * o.x + y * o.y; }
  double operator ^(const Point &o) const { return x * o.y - y * o.x; }

  int cmp(Point o) const {
    if (int t = ::cmp(x, o.x)) return t;
    return ::cmp(y, o.y);
  }
  bool operator ==(const Point &o) const { return cmp(o) == 0; }
```

```
  bool operator !=(const Point &o) const { return cmp(o) != 0; }
  bool operator < (const Point &o) const { return cmp(o) < 0; }

  friend ostream& operator <<(ostream &o, Point p) {
    return o << "(" << p.x << ", " << p.y << ")";
  }

  double x, y;
  static Point pivot;
};

Point Point::pivot(0, 0);

double abs(Point p) {return hypot(p.x, p.y);}
double arg(Point p) {return atan2(p.y, p.x);}


bool seg_intersect(Point p, Point q, Point r, Point s){
    Point A = q - p, B = s - r, C = r - p, D = s - q;
    int a = cmp(A ^ C) + 2 * cmp(A ^ D);
    int b = cmp(B ^ C) + 2 * cmp(B ^ D);
    if( a == 3 || a == -3 || b == 3 || b == -3) return false;
    if( a || b || p == r || p == s || q == r || q == s) return true;
    int t = (p < r) + (p < s) + (q < r) + (q < s);
    return t != 0 && t != 4;
}


inline double angle(const Point &p, const Point &q, const Point &r) {
  Point u = p - r, v = q - r;
  return atan2(u ^ v, u * v);
}


inline int turn(const Point &p, const Point &q, const Point &r) {
  return ::cmp((p - r) ^ (q - r));
}

inline bool between(const Point &p, const Point &q, const Point &r) {
  return turn(p, r, q) == 0 && ::cmp((p - r) * (q - r)) <= 0;
}


```

```
74  Point intersection(const Point &p, const Point &q, const Point &r,
75              const Point &s) {
76    Point a = q - p, b = s - r, c = Point(p ^ q, r ^ s);
77    return Point(Point(a.x, b.x) ^ c, Point(a.y, b.y) ^ c) / (a ^ b);
78  }
79
80  bool radial_comp(const Point &p, const Point &q) {
81    Point P = p - Point::pivot, Q = q - Point::pivot;
82    double R = P ^ Q;
83    if (::cmp(R)) return R > 0;
84    return ::cmp(P * P, Q * Q) < 0;
85  }
86  typedef vector<Point> polygon;
87
88  int in_polygon(const Point &p, const vector<Point> &T) {
89    double a = 0; int N = T.size();
90    for (int i = 0; i < N; ++i) {
91      if (between(T[i], T[(i + 1) % N], p)) return -1;
92      a += angle(T[i], T[(i + 1) % N], p);
93    }
94    return ::cmp(a) != 0;
95  }
96
97  double area(const vector<Point> &T) {
98    double area = 0.0;
99    for(int i = 1; i + 1 < T.size(); i++){
100     area += (T[i] - T[0]) ^ (T[i + 1] - T[0]);
101   }
102   return abs(area / 2.0);
103 }
104
105 polygon poly_intersect(polygon &P, polygon &Q){
106     int m = Q.size(), n = P.size();
107     int a = 0, b = 0, aa = 0, ba = 0, inflag = 0;
108     polygon R;
109     while( (aa < n || ba < m) && aa < 2*n && ba < 2*m){
110         Point p1 = P[a], p2 = P[(a+1) % n], q1 = Q[b], q2 = Q[(b+1) % m];
111         Point A = p2 - p1, B = q2 - q1;
112         int cross = cmp(A ^ B), ha = turn(p2, q2, p1), hb = turn(q2, p2, q1);
113         if(cross == 0 && turn(p1, q1, p2) == 0 && cmp(A * B) < 0){
114             if(between(p1, q1, p2)) R.push_back(q1);
115             if(between(p1, q2, p2)) R.push_back(q2);
116             if(between(q1, p1, q2)) R.push_back(p1);
```

```
117             if(between(q1, p2, q2)) R.push_back(p2);
118             if(R.size() < 2) return polygon();
119             inflag = 1; break;
120         }else if(cross != 0 && seg_intersect(p1, p2, q1, q2)) {
121             if(inflag == 0) aa = ba = 0;
122             R.push_back(intersection(p1, p2, q1, q2));
123             inflag = (hb > 0) ? 1 : -1;
124         }
125         if(cross == 0 && hb < 0 && ha < 0) return R;
126         bool t = cross == 0 && hb == 0 && ha == 0;
127         if(t ? (inflag == 1) : (cross >= 0) ? (ha <= 0) : (hb > 0)){
128             if(inflag == -1) R.push_back(q2);
129             ba++; b++; b %= m;
130         }else{
131             if(inflag == 1) R.push_back(p2);
132             aa++; a++; a %= n;
133         }
134     }
135
136     if(inflag == 0){
137         if (in_polygon(P[0], Q)) return P;
138         if (in_polygon(Q[0], P)) return Q;
139     }
140     R.erase(unique(R.begin(), R.end()), R.end());
141     if(R.size() > 1 && R.front() == R.back()) R.pop_back();
142     return R;
143 }
144
145 int main(){
146     freopen("137.in" , "r", stdin);
147     polygon p,q,r;
148     for(int n,m,x,y; cin>>n && n;){
149         p = polygon(n);
150         for(int i=0;i<n;i++){
151             cin>>x>>y;
152             p[i] = Point(x,y);
153         }
154         cin>>m;
155         q = polygon(m);
156         for(int i=0;i<m;i++){
157             cin>>x>>y;
158             q[i] = Point(x,y);
159         }
```

```
160
161         Point :: pivot = *min_element(p.begin(), p.end());
162         sort(p.begin(),p.end(), radial_comp);
163         Point :: pivot = *min_element(q.begin(), q.end());
164         sort(q.begin(),q.end(), radial_comp);
165
166         r = poly_intersect(q,p);
167
168         Point :: pivot = *min_element(r.begin(),r.end());
169         sort(r.begin(),r.end(), radial_comp);
170         double ans = area(p) + area(q) - 2.0 * area(r);
171         cout.width(8);
172         cout << fixed << setprecision(2) << ans;
173     }
174     cout << endl;
175     return 0;
176 }
```

5 Lines

5.1 Point of intersection

Find the point of intersection between the lines defined by the points given (pairwise).
Requires 1.2:Point

```
1 Point intersection(const Point &p, const Point &q, const Point &r,
2             const Point &s) {
3   Point a = q - p, b = s - r, c = Point(p ^ q, r ^ s);
4   return Point(Point(a.x, b.x) ^ c, Point(a.y, b.y) ^ c) / (a ^ b);
5 }
```

5.2 Segment intersection

Find if the segments determined by `pq` and `rs` have points in common.
Requires 1.2:Point

```
1 bool seg_intersect(Point p, Point q, Point r, Point s){
2     Point A = q - p, B = s - r, C = r - p, D = s - q;
3     int a = cmp(A ^ C) + 2 * cmp(A ^ D);
4     int b = cmp(B ^ C) + 2 * cmp(B ^ D);
5     if( a == 3 || a == -3 || b == 3 || b == -3) return false;
6     if( a || b || p == r || p == s || q == r || q == s) return true;
```

```
    int t = (p < r) + (p < s) + (q < r) + (q < s);
    return t != 0 && t != 4;
}
```

6 Circles

6.1 Diameter of the circumcircle of a triangle

$$D = \frac{abc}{2area} = \frac{|AB||BC||CA|}{2|\Delta ABC|} = \frac{abc}{2\sqrt{s(s-a)(s-b)(s-c)}}$$

Where a,b and c are the lengths of the sides and $s = \frac{a+b+c}{2}$ is the semiperimeter. $2\sqrt{s(s-a)(s-b)(s-c)}$ is the area of the triangle by **Heron's Formula**.

6.2 Circumcenter of a triangle (Circle from three points)

Find the center of the circle that goes trough the three given points. **Watch out for collinear points!**
Requires 1.2:`Point`

```
/*
 * Returns the circumcenter of the triangle formed by the given points.
 * If the points are collinear returs shit.
 */
Point circumcenter(Point p, Point q, Point r){
    Point a = p-r, b = q-r, c = Point(a * (p + r) /2, b * (q+r) /2);
    return Point(c ^ Point(a.y, b.y), Point(a.x, b.x) ^ c ) / (a ^ b);
}
```

6.3 Minimum spanning circle

Find the minimum circle that contains all the given points.
1.2:`Point`, `algorithm`, `vector`

```
typedef pair<Point, double> circle;

bool in_circle(circle C, Point p){
    return cmp(abs(p - C.first), C.second) <= 0;
}

circle spanning_circle(vector<Point> &T){
```

```
8       int n = T.size();
9       random_shuffle(T.begin(),T.end());
10      circle C(Point(), -oo);
11      for(int i=0;i<n;i++) if(!in_circle(C,T[i])){
12          C = circle(T[i],0);
13          for(int j=0; j<i; j++) if (!in_circle(C, T[j])){
14              C = circle((T[i] + T[j] ) / 2, abs(T[i] - T[j]) / 2);
15              for(int k=0; k<j; k++) if(!in_circle(C,T[k])){
16                  Point o = circumcenter(T[i], T[j], T[k]);
17                  C = circle(o, abs(o - T[k]));
18              }
19          }
20      }
21      return C;
22  }
```