Codings

alex4814*

February 24, 2014

Contents

1 Geometry

1.1 2-dimension

1.1.1 Structures

```
#include <cstdio>
#include <cmath>
#include <algorithm>
using namespace std;

#define MAXN 100005

const double EPS = 1e-8;
const double PI = M_PI;

inline int sgn(double x) {
    return (x > EPS) - (x < -EPS);
}

typedef struct Point {
  double x, y;
```

*alex4814.fu@gmail.com

```
  Point(double x = 0, double y = 0): x(x), y(y) {}
} Vector;

struct Segment {
  Point a, b;
  Segment(Point a, Point b): a(a), b(b) {}
};

struct Line {
  Point p;
  Vector v;
  double a;
  Line(Point p = Point(), Vector v = Vector(1, 0)): p(p), v(v) { a = atan2(v.y, v.x); }
  bool operator < (const Line &b) const { return a < b.a; }
};

struct Circle {
  Point c;
  double r;
  Circle(Point c = Point(), double r = 0): c(c), r(r) {}
};
```

1.1.2 Point & Vector

```
Vector operator + (Vector a, Vector b) {
  return Vector(a.x + b.x, a.y + b.y);
}
Vector operator - (Vector a, Vector b) {
  return Vector(a.x - b.x, a.y - b.y);
}
Vector operator * (Vector a, double k) {
  return Vector(a.x * k, a.y * k);
}
Vector operator / (Vector a, double k) {
  return Vector(a.x / k, a.y / k);
}
bool operator == (const Point &a, const Point &b) {
  return sgn(a.x - b.x) == 0 && sgn(a.y - b.y) == 0;
}
bool operator < (const Point &a, const Point &b) {
  return a.y < b.y || (a.y == b.y && a.x < b.x);
}

inline double dot(Vector a, Vector b) {
  return a.x * b.x + a.y * b.y;
}
inline double cross(Vector a, Vector b) {
  return a.x * b.y - a.y * b.x;
}
inline double xmult(Point a, Point b, Point c) {
  return cross(b - a, c - a);
}
inline double length(Vector a) {
  return sqrt(dot(a, a));
}
inline double angle(Vector a, Vector b) {
  return acos(dot(a, b) / length(a) / length(b));
}
inline Vector rotate(Vector a, double rad) {
  return Vector(a.x * cos(rad) - a.y * sin(rad), a.x * sin(rad) + a.y * cos(rad));
}
```

1.1.3 Distance

```
inline double dis_pp(Point a, Point b) {
  return sqrt((a.x - b.x) * (a.x - b.x) + (a.y - b.y) * (a.y - b.y));
}
inline double dis_pl(Point p, Point a, Point b) {
```

```
  Vector v1 = b - a, v2 = p - a;
  return fabs(cross(v1, v2)) / length(v1);
}
inline double dis_ps(Point p, Point a, Point b) {
  Vector v1 = b - a, v2 = p - a, v3 = p - b;
  if (a == b) return length(p - a);
  if (sgn(dot(v1, v2)) < 0) return length(v2);
  if (sgn(dot(v1, v3)) > 0) return length(v3);
  return fabs(cross(v1, v2)) / length(v1);
}
```

1.1.4 Position

```
// change < to <=, if improper
inline bool is_proper_intersection_ss(Point a1, Point a2, Point b1, Point b2) {
  double d1 = cross(a2 - a1, b1 - a1), d2 = cross(a2 - a1, b2 - a1);
  double d3 = cross(b2 - b1, a1 - b1), d4 = cross(b2 - b1, a2 - b1);
  return sgn(d1) * sgn(d2) < 0 && sgn(d3) * sgn(d4) < 0;
}
inline bool is_proper_intersection_ls(Point a1, Point a2, Point b1, Point b2) {
  double d1 = cross(a2 - a1, b1 - a1), d2 = cross(a2 - a1, b2 - a1);
  return sgn(d1) * sgn(d2) < 0;
}
inline bool is_on_ps(Point p, Point a, Point b) {
  return sgn(cross(a - p, b - p)) == 0 && sgn(dot(a - p, b - p)) < 0;
}
inline bool is_on_pl(Point p, Point a, Point b) {
  return sgn(cross(a - p, b - p)) == 0;
}
inline bool is_parallel_ll(Point a1, Point a2, Point b1, Point b2) {
  return sgn(cross(a2 - a1, b2 - b1)) == 0;
}
inline bool is_left_pl(Point p, Line l) {
  return cross(l.v, p - l.p) > 0;
}
// p[] for polygon, p[n] = p[0]
// function: whether point a is inside or on the boundry of polygon p
int is_in_pp(Point a, Point *p, int n) {
  int wn = 0;
  for (int i = 0; i < n; ++i) {
    if (is_on_ps(a, p[i], p[i + 1])) return -1;     // on edge;
    int k = sgn(xmult(p[i], p[i + 1], a));
    int d1 = sgn(p[i].y - a.y);
    int d2 = sgn(p[i + 1].y - a.y);
    if (k > 0 && d1 <= 0 && d2 > 0) ++wn;
    if (k < 0 && d2 <= 0 && d1 > 0) --wn;
  }
  return wn != 0;
}

// -1 for d > r, 0 for d == r, 1 for d < r
inline int relative_position_cl(Circle c, Point p, Vector v) {
  double d = dis_pl(c.c, p, p + v);
  return sgn(c.r - d);
}
// -1 for outside, 0 for on, 1 for inside
inline int relative_position_cp(Circle c, Point p) {
  double d = dis_pp(c.c, p);
  return sgn(c.r - d);
}
```

1.1.5 Intersection

```
Point intersection_ll(Point a, Vector i, Point b, Vector j) {
  Vector k = a - b;
  double t = cross(j, k) / cross(i, j);
  return a + i * t;
}
```

```
Point intersection_ll(Line a, Line b) {
  Vector u = a.p - b.p;
  double t = cross(b.v, u) / cross(a.v, b.v);
  return a.p + a.v * t;
}
Point projection_pl(Point p, Point a, Point b) {
  Vector v = b - a;
  return a + v * (dot(v, p - a) / dot(v, v));
}
```

1.1.6 Convex Hull

```
// p[] for original points in polygon(sorted)
// ch[] for final points in convex hull
// return nums of points
// ch[ix] = ch[0]
// <= 0 for all points in a line, < otherwise
int convex_hull(Point ch[], Point p[], int n)
{
  sort(p, p + n);
  int ix = 0;
  for (int i = 0; i < n; ++i) {
    while (ix > 1 && xmult(ch[ix - 2], ch[ix - 1], p[i]) < 0) --ix;
    ch[ix++] = p[i];
  }
  int t = ix;
  for (int i = n - 2; i >= 0; --i) {
    while (ix > t && xmult(ch[ix - 2], ch[ix - 1], p[i]) < 0) --ix;
    ch[ix++] = p[i];
  }
  return n > 1 ? ix - 1 : ix;
}
```

1.1.7 Area

```
inline double area_of_triangle(Point a, Point b, Point c) {
    return length(cross(b - a, c - a)) / 2;
}
```

1.1.8 Circle

```
// 计算直线与圆的交点保证直线与圆有交点,
// 计算线段与圆的交点可用这个函数后判点是否在线段上
void intersection_cl(Circle c, Point p, Vector v, Point &p1, Point &p2) {
  Point l1 = p, l2 = p + v;
  Vector u = Vector(-v.y, v.x);
  Point p0 = intersection_ll(p, v, c.c, u);
  double d1 = dis_pp(p0, c.c);
  double d2 = dis_pp(l1, l2);
  double t = sqrt(c.r * c.r - d1 * d1)/ d2;
  p1.x = p0.x + (l2.x - l1.x) * t;
  p1.y = p0.y + (l2.y - l1.y) * t;
  p2.x = p0.x - (l2.x - l1.x) * t;
  p2.y = p0.y - (l2.y - l1.y) * t;
}

// 前提: 保证圆与圆有交点, 圆心不重合
void intersection_cc(Circle c1, Circle c2, Point& p1, Point& p2){
  double d = dis_pp(c1.c, c2.c);
  double t = (1.0 + (c1.r * c1.r - c2.r * c2.r) / d / d) / 2;
  Point u = Point(c1.c.x + (c2.c.x - c1.c.x) * t, c1.c.y + (c2.c.y - c1.c.y) * t);
  Point v = Point(u.x + c1.c.y - c2.c.y, u.y - c1.c.x + c2.c.x);
  intersection_cl(c1, u, v - u, p1, p2);
}

// 计算圆外一点与圆的两个切点
void point_of_tangency_cp(Circle c, Point p, Point &p1, Point &p2) {
  double d = dis_pp(c.c, p);
```

```
  double theta = asin(c.r / d);
  Vector v1 = rotate(c.c - p, theta);
  Vector v2 = rotate(c.c - p, 2 * PI - theta);
  p1 = p + v1 / length(v1) * d * cos(theta);
  p2 = p + v2 / length(v2) * d * cos(theta);
}

// 计算最小的圆覆盖平面上的点集
Circle min_circle_cover(Point *p, int n) {
  Point c = p[0]; double r = 0;
  for (int i = 1; i < n; ++i) {
    if (sgn(dis_pp(c, p[i]) - r) <= 0) continue;
    c = p[i], r = 0;
    for (int j = 0; j < i; ++j) {
      if (sgn(dis_pp(c, p[j]) - r) <= 0) continue;
      c.x = (p[i].x + p[j].x) / 2;
      c.y = (p[i].y + p[j].y) / 2;
      r = dis_pp(c, p[j]);
      for (int k = 0; k < j; ++k) {
        if (sgn(dis_pp(c, p[k]) - r) <= 0) continue;
        c = circumcenter(p[i], p[j], p[k]);
        r = dis_pp(c, p[k]);
      }
    }
  }
  return Circle(c, r);
}
```

1.1.9 Segment

```
int seg_union(Segment *s, int &n) {
  int m = 0;
  for (int i = 1; i < n; ++i) {
    if (s[m].b.x < s[i].a.x) {
      s[++m] = s[i];
    } else {
      s[m].a.x = min(s[m].a.x, s[i].a.x);
      s[m].b.x = max(s[m].b.x, s[i].b.x);
    }
  }
  return n = m + 1;
}
```

1.1.10 Symmetry

```
Point symmetry_pl(Point p, Point a, Point b) {
  Vector v1 = b - a, v2 = Vector(-v1.y, v1.x);
  Point m = intersection_ll(a, v1, p, v2);
  return p + (m - p) * 2;
}
```

1.1.11 Triangle

```
Point circumcenter(Point a, Point b, Point c) {
  double x1 = b.x - a.x, y1 = b.y - a.y, e1 = (x1 * x1 + y1 * y1) / 2;
  double x2 = c.x - a.x, y2 = c.y - a.y, e2 = (x2 * x2 + y2 * y2) / 2;
  double _d = x1 * y2 - x2 * y1;
  double _x = a.x + (e1 * y2 - e2 * y1) / _d;
  double _y = a.y + (x1 * e2 - x2 * e1) / _d;
  return Point(_x, _y);
}
```

1.2 3-dimension

1.2.1 Structures

```
#include <cstdio>
```

```cpp
#include <cmath>
#include <algorithm>
using namespace std;

const double EPS = 1e-8;
const double PI = acos(-1.0);

inline int sgn(double x) {
  return (x > EPS) - (x < -EPS);
}

typedef struct Point3 {
  double x, y, z;
  Point3(double x = 0, double y = 0, double z = 0): x(x), y(y), z(z) {}
} Vector3 ;
```

1.2.2 Point & Vector

```cpp
Vector3 operator + (Vector3 a, Vector3 b) {
  return Vector3(a.x + b.x, a.y + b.y, a.z + b.z);
}
Vector3 operator - (Point3 a, Point3 b) {
  return Vector3(a.x - b.x, a.y - b.y, a.z - b.z);
}
Vector3 operator * (Vector3 a, double k) {
  return Vector3(a.x * k, a.y * k, a.z * k);
}
Vector3 operator / (Vector3 a, double k) {
  return Vector3(a.x / k, a.y / k, a.z / k);
}
bool operator == (const Point3 &a, const Point3 &b) {
  return sgn(a.x - b.x) == 0 && sgn(a.y - b.y) == 0 && sgn(a.z - b.z) == 0;
}
inline double dot(Vector3 a, Vector3 b) {
  return a.x * b.x + a.y * b.y + a.z * b.z;
}
inline double length(Vector3 a) {
  return sqrt(dot(a, a));
}
inline double angle(Vector3 a, Vector3 b) {
  return acos(dot(a, b) / length(a) / length(b));
}
inline Vector3 cross(Vector3 a, Vector3 b) {
  return Vector3(a.y * b.z - a.z * b.y, a.z * b.x - a.x * b.z, a.x * b.y - a.y * b.x);
}
```

1.2.3 Distance

```cpp
inline double dis_pp(Point3 a, Point3 b) {
  return length(a - b);
}
inline double dis_pf(Point3 p, Point3 p0, Vector3 n) {
  return fabs(dot(p - p0, n));
}
inline Point3 projection_pf(Point3 p, Point3 p0, Vector3 n) {
  return p - n * dot(p - p0, n);
}
inline Point3 intersection_lf(Point3 a, Point3 b, Point3 p0, Vector3 n) {
  Vector3 v = b - a;
  double t = dot(n, p0 - a) / dot(n, b - a);
  return a + v * t;
}

inline double dis_pl(Point3 p, Point3 a, Point3 b) {
  Vector3 v1 = b - a, v2 = p - a;
  return length(cross(v1, v2)) / length(v1);
}
inline double dis_ps(Point3 p, Point3 a, Point3 b) {
```

```
  if (a == b) return length(p - a);
  Vector3 v1 = b - a, v2 = p - a, v3 = p - b;
  if (sgn(dot(v1, v2)) < 0) return length(v2);
  if (sgn(dot(v1, v3)) > 0) return length(v3);
  return length(cross(v1, v2)) / length(v1);
}
```

1.2.4 Convex Hull

```
struct Face {
  int v[3];
  Vector3 normal(Point3 *p) const {
    return cross(p[v[1]] - p[v[0]], p[v[2]] - p[v[0]]);
  }
  bool cansee(Point3 *p, int i) const {
    return dot(p[i] - p[v[0]], normal(p)) > 0;
  }
};

vector<Face> convex_hull3(Point3 *p, int n) {
  static int vis[MAXN][MAXN];
  vector<Face> ch;
  ch.push_back((Face){{ 0, 1, 2 }});
  ch.push_back((Face){{ 2, 1, 0 }});
  for (int i = 3; i < n; ++i) {
    vector<Face> next;
    // calculate the left visibility of each edge
    for (int j = 0; j < ch.size(); ++j) {
      Face &f = ch[j];
      int ret = f.cansee(p, i);
      if (!ret) next.push_back(f);
      for (int k = 0; k < 3; ++k) {
        vis[ f.v[k] ][ f.v[(k + 1) % 3] ] = ret;
      }
    }
    for (int j = 0; j < ch.size(); ++j) {
      for (int k = 0; k < 3; ++k) {
        int a = ch[j].v[k], b = ch[j].v[(k + 1) % 3];
        if (vis[a][b] != vis[b][a] && vis[a][b]) {  // (a, b)'s left is visible to p[i]
          next.push_back((Face){{ a, b, i }});
        }
      }
    }
    ch = next;
  }
  return ch;
}
```