|  |
| --- |
| Sun yen-set University |
| Geometry |
| SYSU\_TheRavenChaser |
|  |
| **doublehh** |
| **2014/12/3** |

目录

[Base.h 1](#_Toc405405134)

[Point.h 2](#_Toc405405135)

[Line.h 3](#_Toc405405136)

[Triangle.h 4](#_Toc405405137)

[Polygon.h 4](#_Toc405405138)

[Algorithm.h 6](#_Toc405405139)

[Point3.h 9](#_Toc405405140)

[Line3.h 11](#_Toc405405141)

[Ball.h 12](#_Toc405405142)

[Algorithm3.h 13](#_Toc405405143)

[Circle.h 14](#_Toc405405144)

Base.h

#include <cstdio>

#include <cstring>

#include <cmath>

#include <vector>

#include <utility>

#include <algorithm>

using namespace std;

const double inf = 1e10;

const double eps = 1e-8;

const double pi = acos(-1.0);

const int maxn = 100;

// about output

// %f format for g++, %lf format for c++, about printf

inline int dcmp(double x)

{

if (fabs(x) < eps) return 0;

return x < 0 ? -1: 1;

}

inline double adjust(double a)

{

while (a < 0\*pi) a += 2\*pi;

while (a > 2\*pi) a -= 2\*pi;

return a;

}

inline double adjust01(double x)

{

return min(1.0, max(0.0, x));

}

// default: 0-2\*pi

inline double normal(double rad, double center=pi)

{ return rad - 2\*pi \* floor((rad + pi - center) / (2\*pi)); }

inline double rand01() { return rand() / (double)RAND\_MAX; }

inline double randeps() { return (rand01() - .5) \* eps; }

inline double consinetheorem(double a, double b, double c, double C)

{

if (C == -1.0)

return acos((a\*a + b\*b - c\*c) / (2\*a\*b));

return -1.0;

}

Point.h

struct Point

{

double x, y;

Point() {}

Point(double x, double y): x(x), y(y) {}

void read()

{ scanf("%lf%lf", &x, &y); }

void print() const

{ printf("(%.2lf, %.2lf)\n", x, y); }

bool operator < (const Point &p) const

{ return dcmp(x - p.x) < 0 || dcmp(x - p.x) == 0 && dcmp(y - p.y) < 0; }

bool operator == (const Point &p) const

{ return !dcmp(x-p.x) && !dcmp(y-p.y); }

};

typedef Point Vector;

typedef const Point &CP;

typedef const Vector &CV;

Vector operator + (CV A, CV B) { return Vector(A.x+B.x, A.y+B.y); }

Vector operator - (CV A, CV B) { return Vector(A.x-B.x, A.y-B.y); }

Vector operator \* (CV A, double p) { return Vector(A.x\*p, A.y\*p); }

Vector operator / (CV A, double p) { return Vector(A.x/p, A.y/p); }

double Cross(CV A, CV B) { return A.x\*B.y-A.y\*B.x; }

double Dot(CV A, CV B) { return A.x\*B.x+A.y\*B.y; }

double Length2(CV v) { return Dot(v, v); }

double Length(CV v) { return sqrt(Dot(v, v)); }

double Angle(CV v) { return atan2(v.y, v.x); }

// 0-pi

double Angle(CV A, CV B) { return acos(Dot(A, B)/Length(A)/Length(B)); }

// -pi ~ pi

// { return fabs(atan2(Cross(A, B), Dot(A, B))); }

Vector Unit(CV A) { return A / Length(A); }

Vector Rotate(CV A, double rad)

{ return Vector(A.x\*cos(rad) - A.y\*sin(rad), A.x\*sin(rad) + A.y\*cos(rad)); }

// circumcircle

Point center(CP p1, CP p2, CP p3)

{

double d1 = Dot(p2-p1, p3-p1), d2 = Dot(p3-p2, p1-p2), d3 = Dot(p1-p3, p2-p3);

double c1 = d2 \* d3, c2 = d1 \* d3, c3 = d1 \* d2, c = c1 + c2 + c3;

if (!dcmp(c)) return p1;

return (p1 \* (c2 + c3) + p2 \* (c1 + c3) + p3 \* (c1 + c2)) / (2 \* c);

}

Line.h

struct Line

{

Point p;

Vector v;

Line() {}

Line(Point A, Point B): p(A), v(B-A) {}

Point point(double t) const

{ return p + v\*t; }

Point A() const

{ return p; }

Point B() const

{ return p+v; }

void print() const

{

p.print();

(p+v).print();

}

};

typedef const Line &CL;

typedef Line Segment;

typedef const Segment &CS;

Point GetLineIntersection(CL L1, CL L2)

{

Vector u = L1.p-L2.p;

double t = Cross(L2.v, u) / Cross(L1.v, L2.v);

return L1.p+L1.v\*t;

}

double DistanceToLine(CP P, CL L)

{

Vector v1 = L.v, v2 = P-L.p;

return fabs(Cross(v1, v2)) / Length(v1);

}

double DistanceToSegment(CP P, CP A, CP B)

{

if (A == B) return Length(P - A);

Vector v1 = B-A, v2 = P-A, v3 = P-B;

if (dcmp(Dot(v1, v2)) < 0) return Length(v2);

if (dcmp(Dot(v1, v3)) > 0) return Length(v3);

return fabs(Cross(v1, v2)) / Length(v1);

}

double UntouchedSegSegDistance(CP a, CP b, CP c, CP d)

{ return min(min(DistanceToSegment(a, c, d), DistanceToSegment(b, c, d)), min(DistanceToSegment(c, a, b), DistanceToSegment(d, a, b))); }

Point GetProjection(CP P, CL L)

{ return L.p + L.v\*(Dot(L.v, P-L.p) / Dot(L.v, L.v)); }

bool SegmentProperIntersection(CP a1, CP a2, CP b1, CP b2)

{

double c1 = Cross(a2-a1, b1-a1), c2 = Cross(a2-a1, b2-a1),

c3 = Cross(b2-b1, a1-b1), c4 = Cross(b2-b1, a2-b1);

return dcmp(c1) \* dcmp(c2) < 0 && dcmp(c3) \* dcmp(c4) < 0;

}

bool OnSegment(CP p, CP a1, CP a2)

{ return dcmp(Cross(a1-p, a2-p)) == 0 && dcmp(Dot(a1-p, a2-p)) < 0; }

Triangle.h

double Area(double a, double b, double c)

{

double p = (a+b+c)/2;

return sqrt(p\*(p-a)\*(p-b)\*(p-c));

}

double ExcircleRadium(double a, double b, double c)

{ return Area(a, b, c) \* 2/ (-a + b + c); }

double InCircleRadium(double a, double b, double c)

{ return Area(a, b, c) \* 2/ (a + b + c); }

Polygon.h

typedef vector<Point> Polygon;

bool PointInPolygon(Point p, Polygon poly)

{

int wn = 0;

int n = poly.size();

for (int i = 0; i < n; i++)

{

Point C = poly[i], D = poly[(i+1)%n];

if (OnSegment(p, C, D)) return true;

int k = dcmp(Cross(D-C, p-C));

int d1 = dcmp(C.y-p.y);

int d2 = dcmp(D.y-p.y);

if (k > 0 && d1 <= 0 && d2 > 0) wn++;

if (k < 0 && d2 <= 0 && d1 > 0) wn--;

}

return wn;

}

double PolygonArea(Point poly[], int n)

{

double area = 0;

poly[n] = poly[0];

for (int i = 0; i < n; i++)

area += Cross(poly[i], poly[i+1]) / 2;

return fabs(area);

}

// Andrew Convex Hull algorithm

int ConvexHull(Point p[], int n)

{

static Point q[maxn];

// important

sort(p, p+n);

// n = unique(p, p+n) - p;

int m = 0;

for (int i = 0; i < n; i++)

{

while (m > 1 && Cross(q[m-1]-q[m-2], p[i]-q[m-2]) <= 0) m--;

q[m++] = p[i];

}

int k = m;

for (int i = n-2; i >= 0; i--)

{

while (m > k && Cross(q[m-1]-q[m-2], p[i]-q[m-2]) <= 0) m--;

q[m++] = p[i];

}

if (n > 1) m--;

for (int i = 0; i < m; i++)

p[i] = q[i];

return m;

}

struct line

{

Point p;

Vector v;

double ang;

line() {}

line(CP A, CP B): p(A), v(B-A) { ang = atan2(v.y, v.x); }

bool operator < (const line &L) const

{ return ang < L.ang; }

};

typedef const line &Cl;

bool OnLeft(Cl L, CP p)

{ return Cross(L.v, p-L.p) > 0; }

Point GetIntersection(Cl a, Cl b)

{

Vector u = a.p-b.p;

double t = Cross(b.v, u) / Cross(a.v, b.v);

return a.p + a.v \* t;

}

int HalfplaneIntersection(line L[], int n, Point poly[])

{

sort(L, L+n);

int first, last;

static Point p[maxn];

static line q[maxn];

q[first=last=0] = L[0];

for (int i = 1; i < n; i++)

{

while (first < last && !OnLeft(L[i], p[last-1])) last--;

while (first < last && !OnLeft(L[i], p[first])) first++;

q[++last] = L[i];

// Select inter

if (!dcmp(Cross(q[last].v, q[last-1].v)))

{

last--;

if (OnLeft(q[last], L[i].p)) q[last] = L[i];

}

if (first < last) p[last-1] = GetIntersection(q[last-1], q[last]);

}

while (first < last && !OnLeft(q[first], p[last-1])) last--;

if (last-first <= 1) return 0;

p[last] = GetIntersection(q[last], q[first]);

int m = 0;

for (int i = first; i <= last; i++)

poly[m++] = p[i];

return m;

}

// Euler Formula

// V + F - E = 2(2D-plane) V -> # of vertex, F -> # of region, E -> # of edge

// V + F - E = X(P) X(P) -> euler topological invariant (2 - 2\*h)

Algorithm.h

// p1 and p2 are both counterclockwise

// need min((p1, n1, p2, n2), (p2, n2, p1, n1))

// Max very similary

double MinDistcanceBetween2Polygon(Point p1[], int n1, Point p2[], int n2)

{

double ans = inf;

int i = 0, j = 0;

for (int k = 1; k < n1; k++) if (p1[k] < p1[i]) i = k;

for (int k = 1; k < n2; k++) if (p2[j] < p2[k]) j = k;

for (int t = 0; t < n1; t++)

{

for (;;)

{

// Area

int diff = dcmp(Cross(p1[i+1]-p1[i], p2[j+1]-p2[j]));

if (diff <= 0)

{

if (!diff) ans = min(ans, UntouchedSegSegDistance(p1[i], p1[i+1], p2[j], p2[j+1]));

else ans = min(ans, DistanceToSegment(p2[j], p1[i], p1[i+1]));

break;

}

j = (j+1) % n2;

}

i = (i+1) % n1;

}

return ans;

}

double shadow\_length(double alpha, Point a, Point b)

{

double dx = a.x - b.x;

double dy = a.y - b.y;

return fabs(dx \* cos(alpha) + dy \* sin(alpha));

}

void SmallestEnclosingRectangle(Point p[], int n, double &area, double &peri)

{

area = peri = inf;

Point \*q[4] = {NULL, NULL, NULL, NULL};

for (int i = 0; i < n; i++)

{

Point \*t = p+i;

if (!q[0] || t->y < q[0]->y || t->y == q[0]->y && t->x < q[0]->y) q[0] = t;

if (!q[1] || t->x > q[1]->x || t->x == q[1]->x && t->y < q[1]->y) q[1] = t;

if (!q[2] || t->y > q[2]->y || t->y == q[2]->y && t->x > q[2]->x) q[2] = t;

if (!q[3] || t->x < q[3]->x || t->x == q[3]->x && t->y > q[3]->y) q[3] = t;

}

double alpha = 0;

for (int k = 0; k < n+5; k++)

{

int j = -1;

double Min = inf;

for (int i = 0; i < 4; i++)

{

double tmp = adjust(Angle(q[i][1] - q[i][0]) - (alpha + i\*pi/2));

if (tmp < Min)

{

j = i;

Min = tmp;

}

}

if (++q[j] == p + n) q[j] = p + 0;

alpha = adjust(alpha + Min);

double a = shadow\_length(alpha + pi / 2, \*q[0], \*q[2]);

double b = shadow\_length(alpha, \*q[1], \*q[3]);

area = min(area, a\*b);

peri = min(peri, 2\*(a+b));

if (dcmp(alpha - pi / 2) > 0) break;

}

}

const int maxp = 1e4;

struct Edge

{

int from, to;

double ang;

Edge(int from, int to, double ang):

from(from), to(to), ang(ang) {}

};

struct PSLG

{

int n, m;

Point p[maxp];

vector<int> G[maxp];

vector<Edge> edges;

int prev[maxp<<1];

bool vis[maxp<<1];

double getAngle(int from, int to)

{ return Angle(p[to] - p[from]); }

void init(int n, Point inter[])

{

this->n = n;

for (int i = 0; i < n; i++)

{

G[i].clear();

p[i] = inter[i];

}

edges.clear();

}

void addEdge(int from, int to)

{

edges.push\_back(Edge(from, to, getAngle(from, to)));

edges.push\_back(Edge(to, from, getAngle(to, from)));

m = edges.size();

G[from].push\_back(m-2);

G[to].push\_back(m-1);

}

// PSLG build the graph and output is on faces

// all face are counter-clockwise

void build(vector<Polygon> &faces)

{

// clear the faces

faces.clear();

// calculate prev

for (int u = 0; u < n; u++)

{

int sz = G[u].size();

for (int i = 0; i < sz; i++) for (int j = i + 1; j < sz; j++)

if (edges[G[u][i]].ang > edges[G[u][j]].ang) swap(G[u][i], G[u][j]);

for (int i = 0; i < sz; i++)

prev[G[u][(i+1)%sz]] = G[u][i];

}

// find plane region

memset(vis, false, sizeof(vis));

// infinite border

for (int u = 0; u < n; u++)

{

if (G[u].size() == 1)

{

int e = G[u][0];

if (!vis[e])

{

Polygon poly;

for (;;)

{

int from = edges[e].from;

if (from != u && G[from].size() == 1) break;

vis[e] = true;

poly.push\_back(p[from]);

e = prev[e^1];

}

int from = edges[e].from;

poly.push\_back(p[from]);

faces.push\_back(poly);

}

}

}

// finite region

for (int u = 0; u < n; u++)

{

for (int i = 0; i < G[u].size(); i++)

{

int e = G[u][i];

if (!vis[e])

{

Polygon poly;

for (;;)

{

vis[e] = true;

// left[e] = face\_cnt;

int from = edges[e].from;

poly.push\_back(p[from]);

e = prev[e^1];

if (e == G[u][i]) break;

}

faces.push\_back(poly);

}

}

}

}

} solver;

Point3.h

struct Point3

{

double x, y, z;

Point3(double x=0, double y=0, double z=0):

x(x), y(y), z(z) {}

void read()

{ scanf("%lf%lf%lf", &x, &y, &z); }

void print()

{ printf("(%.2lf, %.2lf, %.2lf)\n", x, y, z); }

};

typedef Point3 Vector3;

typedef const Point3 &CP3;

typedef const Vector3 &CV3;

typedef Point3 Triangle[3];

typedef Point3 Tetrahedron[4];

bool operator == (CV3 A, CV3 B) { return !dcmp(A.x-B.x) && !dcmp(A.y-B.y) && !dcmp(A.z-B.z); }

Vector3 operator + (CV3 A, CV3 B) { return Vector3(A.x+B.x, A.y+B.y, A.z+B.z); }

Vector3 operator - (CV3 A, CV3 B) { return Vector3(A.x-B.x, A.y-B.y, A.z-B.z); }

Vector3 operator \* (CV3 A, double b) { return Vector3(A.x\*b, A.y\*b, A.z\*b); }

Vector3 operator / (CV3 A, double b) { return Vector3(A.x/b, A.y/b, A.z/b); }

double Dot(CV3 A, CV3 B) { return A.x\*B.x + A.y\*B.y + A.z\*B.z; }

Vector3 Cross(CV3 A, CV3 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); }

double Length(CV3 A) { return sqrt(Dot(A, A)); }

double torad(double deg) { return deg/180 \* pi; }

double Area2(CP3 A, CP3 B, CP3 C) { return Length(Cross(B-A, C-A)); }

double Volume6(CP3 A, CP3 B, CP3 C, CP3 D) { return Dot(D-A, Cross(B-A, C-A)); }

// lat == latitude, lng == longitude

// north latitude > 0, east longitude > 0

Vector3 get\_coord(double R, double lat, double lng)

{

lat = torad(lat);

lng = torad(lng);

return Point3(R\*cos(lat)\*sin(lng), R\*cos(lat)\*cos(lng), R\*sin(lat));

}

// R is ball radium, o is (0, 0, 0)

double DistanceOnBall(CV3 A, CV3 B, double R)

{

double d = Length(A-B);

double alpha = 2\*asin(d/2./R);

return alpha \* R;

}

// (jingdu, weidu)

// (lag, lat)

// A(alpha1, belta1) B(alpha2, belta2)

// double DistanceOnBall(double alpha1, double alpha2, double belta1, double belta2)

// { return R \* acos(cos(belta1)\*cos(belta2)\*cos(alpha1-alpha2)+sin(belta1)\*sin(belta2)); }

double DistanceToPlane(CP3 p, CP3 p0, CV3 n)

{ return fabs(Dot(p-p0, n)) / Length(n); }

// p's projection in plane (p0, n)

Point3 GetPlaneProjection(CP3 p, CP3 p0, CV3 n)

{

double d = Dot(p-p0, n) / Length(n);

return p - n \* (d / Length(n));

}

// P must in plane(tri)

bool PointInTri(CP3 P, Triangle tri)

{

double area1 = Area2(P, tri[0], tri[1]);

double area2 = Area2(P, tri[1], tri[2]);

double area3 = Area2(P, tri[2], tri[0]);

return !dcmp(area1 + area2 + area3 - Area2(tri[0], tri[1], tri[2]));

}

Line3.h

struct Line3

{

Point3 p;

Vector3 v;

Line3(CP3 A, CP3 B): p(A), v(B-A) {}

Point3 A() { return p; }

Point3 B() { return p + v; }

Point3 point(double t) { return p + v \* t; }

};

typedef Line3 Segment3;

Point3 GetLineProjection(CP3 P, Segment3 seg)

{ return seg.p + seg.v \* (Dot(seg.v, P-seg.p) / Dot(seg.v, seg.v)); }

double DistanceToLine(Point3 P, Segment3 seg)

{ return Length(Cross(seg.v, P - seg.p)) / Length(seg.v); }

Point3 LinePlaneIntersection(Line3 L, CP3 p0, CV3 n)

{

double t = (Dot(n, p0-L.p) / Dot(n, L.v));

return L.p + L.v\*t;

}

// the distance between two line in different planes

// return s in p1 + s \* v1(the intersection)

bool LineDistance3D(Line3 L1, Line3 L2, double &s)

{

double b = Dot(L1.v, L1.v) \* Dot(L2.v, L2.v) - Dot(L1.v, L2.v) \* Dot(L1.v, L2.v);

if (!dcmp(b)) return false; // parallel or coincide

double a = Dot(L1.v, L2.v) \* Dot(L2.v, L1.p-L2.p) - Dot(L2.v, L2.v) \* Dot(L1.v, L1.p-L2.p);

s = a / b;

return true;

}

double PointToSegment3D(CP3 p, Segment3 seg)

{

Vector3 v1 = seg.v, v2 = p - seg.A(), v3 = p - seg.B();

if (dcmp(Dot(v1, v2)) <= 0) return Length(v2);

if (dcmp(Dot(v1, v3)) >= 0) return Length(v3);

return Length(Cross(v1, v2)) / Length(v1);

}

double SegmentDistance3D(Segment3 seg1, Segment3 seg2)

{

double s, t;

if (LineDistance3D(seg1, seg2, s) && dcmp(s) >= 0 && dcmp(s-1) <= 0

&& LineDistance3D(seg2, seg1, t) && dcmp(t) >= 0 && dcmp(t-1) <= 0)

return Length(seg1.point(s) - seg2.point(t));

return min(min(PointToSegment3D(seg1.A(), seg2), PointToSegment3D(seg1.B(), seg2)),

min(PointToSegment3D(seg2.A(), seg1), PointToSegment3D(seg2.B(), seg1)));

}

double SegmentTriDistance3D(Segment3 seg, Triangle tri)

{

Vector3 n = Cross(tri[1]-tri[0], tri[2]-tri[0]);

Vector3 proj1 = GetPlaneProjection(seg.A(), tri[0], n);

Vector3 proj2 = GetPlaneProjection(seg.B(), tri[0], n);

double ans = inf;

if (PointInTri(proj1, tri))

ans = min(ans, Length(seg.A()-proj1));

if (PointInTri(proj2, tri))

ans = min(ans, Length(seg.B()-proj2));

for (int i = 0; i < 3; i++)

ans = min(ans, SegmentDistance3D(seg, Segment3(tri[i], tri[(i+1)%3])));

return ans;

}

double TriDistance3D(Triangle tri1, Triangle tri2)

{

double ans = inf;

for (int i = 0; i < 3; i++)

{

ans = min(ans, SegmentTriDistance3D(Segment3(tri1[i], tri1[(i+1)%3]), tri2));

ans = min(ans, SegmentTriDistance3D(Segment3(tri2[i], tri2[(i+1)%3]), tri1));

}

return ans;

}

double TetrahedronDistance(Tetrahedron tet[2])

{

static Triangle tri[2][4];

for (int i = 0; i < 2; i++)

{

tri[i][0][0] = tet[i][0], tri[i][0][1] = tet[i][1], tri[i][0][2] = tet[i][2];

tri[i][1][0] = tet[i][0], tri[i][1][1] = tet[i][1], tri[i][1][2] = tet[i][3];

tri[i][2][0] = tet[i][0], tri[i][2][1] = tet[i][2], tri[i][2][2] = tet[i][3];

tri[i][3][0] = tet[i][1], tri[i][3][1] = tet[i][2], tri[i][3][2] = tet[i][3];

}

double ans = inf;

for (int i = 0; i < 4; i++) for (int j = 0; j < 4; j++)

ans = min(ans, TriDistance3D(tri[0][i], tri[1][j]));

return ans;

}

Ball.h

struct Ball

{

Point3 o;

double r;

Ball(Point3 o, double r): o(o), r(r) {}

};

bool SegmentBallIntersection(Segment3 seg, Ball ball, double &t1, double &t2)

{

double dist = DistanceToLine(ball.o, seg);

if (dcmp(dist - ball.r) >= 0)

return false;

double t = Dot(seg.v, ball.o-seg.p) / Dot(seg.v, seg.v);

double dt = sqrt(ball.r\*ball.r - dist\*dist) / Length(seg.v);

t1 = adjust01(t-dt);

t2 = adjust01(t+dt);

if (!dcmp(t1 - t2))

return false;

if (dcmp(t1 - t2) > 0)

swap(t1, t2);

return true;

}

Algorithm3.h

struct Face

{

int v[3];

Face(int a, int b, int c)

{ v[0] = a, v[1] = b, v[2] = c; }

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; }

};

Point3 add\_noise(CP3 p)

{ return Point3(p.x + randeps(), p.y + randeps(), p.z + randeps()); }

// increment method

vector<Face> CH3D(Point3 q[], int n)

{

static Point3 p[maxn];

for (int i = 0; i < n; i++)

p[i] = add\_noise(q[i]);

static bool vis[maxn][maxn];

vector<Face> cur;

cur.push\_back(Face(0, 1, 2));

cur.push\_back(Face(2, 1, 0));

for (int i = 3; i < n; i++)

{

vector<Face> nex;

for (int j = 0; j < cur.size(); j++)

{

Face &f = cur[j];

bool flag = f.cansee(p, i);

if (!flag) nex.push\_back(f);

for (int k = 0; k < 3; k++) vis[f.v[k]][f.v[(k+1)%3]] = flag;

}

for (int j = 0; j < cur.size(); j++)

{

for (int k = 0; k < 3; k++)

{

int a = cur[j].v[k], b = cur[j].v[(k+1)%3];

if (vis[a][b] != vis[b][a] && vis[a][b])

nex.push\_back(Face(a, b, i));

}

}

cur = nex;

}

return cur;

}

Circle.h

struct Circle

{

Point o;

double r;

Circle() {}

Circle(Point o, double r): o(o), r(r) {}

bool operator < (const Circle &C) const

{ return o < C.o || o == C.o && dcmp(r - C.r) < 0; }

bool operator == (const Circle &C) const

{ return o == C.o && dcmp(r - C.r) == 0; }

Point point(double a) const

{ return Point(o.x + cos(a)\*r, o.y + sin(a)\*r); }

void read()

{

o.read();

scanf("%lf", &r);

}

void print() const

{

o.print();

printf("r = %.2lf\n", r);

}

};

typedef const Circle &CC;

int GetLineCircleIntersection(Line L, CC C, double &t1, double &t2, vector<Point> &sol)

{

double a = L.v.x, b = L.p.x - C.o.x, c = L.v.y, d = L.p.y - C.o.y;

double e = a\*a + c\*c, f = 2\*(a\*b + c\*d), g = b\*b + d\*d - C.r\*C.r;

double delta = f\*f - 4\*e\*g;

if (dcmp(delta) < 0) return 0;

if (!dcmp(delta))

{

t1 = t2 = -f / (2\*e); sol.push\_back(L.point(t1));

return 1;

}

t1 = (-f - sqrt(delta)) / (2 \* e); sol.push\_back(L.point(t1));

t2 = (-f + sqrt(delta)) / (2 \* e); sol.push\_back(L.point(t2));

return 2;

}

int GetCircleSegmentIntersection(CC C, Line L, double &t1, double &t2)

{

double a = L.v.x, b = L.p.x - C.o.x, c = L.v.y, d = L.p.y - C.o.y;

double e = a\*a + c\*c, f = 2\*(a\*b + c\*d), g = b\*b + d\*d - C.r\*C.r;

double delta = f\*f - 4\*e\*g;

if (delta < 0) return 0;

int tot = 0;

if (!dcmp(delta))

{

t1 = t2 = -f / (2\*e);

if (dcmp(t1) >= 0 && dcmp(t1-1) <= 0) tot++;

// else t1 = 0, t2 = 1, tot++;

return tot;

}

t1 = (-f - sqrt(delta)) / (2 \* e);

if (dcmp(t1) > 0 && dcmp(t1-1) < 0) tot++;

// else t1 = 0, tot++;

t2 = (-f + sqrt(delta)) / (2 \* e);

if (dcmp(t2) > 0 && dcmp(t2-1) < 0) tot++;

// else t2 = 1, tot++;

return tot;

}

bool PointInCircle(CP p, const Circle &c)

{ return dcmp(Length(p-c.o) - c.r) < 0; }

bool CircleInCircle(const Circle &c1, const Circle &c2)

{ return dcmp((Length(c1.o-c2.o) + c1.r) - c2.r) <= 0; }

bool GetCircleCircleIntersection(const Circle &c1, const Circle &c2, double &a1, double &a2)

{

double d = Length(c1.o - c2.o);

// external

if (dcmp(c1.r+c2.r - d) < 0) return false;

// coincide or internal

if (!dcmp(d) || dcmp(fabs(c1.r - c2.r) - d) > 0) return false;

double alpha = Angle(c2.o - c1.o), beta = consinetheorem(c1.r, d, c2.r, -1);

a1 = adjust(alpha - beta), a2 = adjust(alpha + beta);

return true;

}

typedef pair<double, int> Rad;

bool CirclesIntersect(vector<Circle> &circle, double &area, Point &P)

{

bool ret = false;

area = 0;

sort(circle.begin(), circle.end());

circle.erase(unique(circle.begin(), circle.end()), circle.end());

for (int i = 0; i < circle.size(); i++)

{

vector<Rad> a;

a.push\_back(Rad(0, 0));

a.push\_back(Rad(2\*pi, 0));

int k = 0;

for (int j = 0; j < circle.size(); j++) if (i != j)

{

double a1, a2;

if (CircleInCircle(circle[i], circle[j])) { k++; continue; }

if (!GetCircleCircleIntersection(circle[i], circle[j], a1, a2)) continue;

a.push\_back(Rad(a1, 1));

a.push\_back(Rad(a2, -1));

if (a1 > a2) k++;

}

sort(a.begin(), a.end());

for (int j = 0; j+1 < a.size(); j++)

{

k += a[j].second;

if (k + 1 == circle.size())

{

ret = true;

double angle = a[j+1].first - a[j].first;

area += .5 \* circle[i].r\*circle[i].r \* (angle - sin(angle));

area += .5 \* Cross(circle[i].point(a[j].first), circle[i].point(a[j+1].first));

}

}

}

return ret;

}

void MinimalCoverCircle(vector<Point> points, Point &o, double &r)

{

random\_shuffle(points.begin(), points.end());

o = points[0]; r = 0;

for (int i = 1; i < points.size(); ++i)

{

if (dcmp(Length(points[i] - o) - r) <= 0) continue;

o = points[i]; r = 0;

for (int j = 0; j < i; ++j)

{

if (dcmp(Length(points[j] - o) - r) <= 0) continue;

o = (points[i] + points[j]) \* .5; r = Length(points[j] - o);

for (int k = 0; k < j; ++k)

{

if (dcmp(Length(points[k] - o) - r) <= 0) continue;

o = center(points[i], points[j], points[k]); r = Length(points[k] - o);

}

}

}

}

double CircleIntersectTriangle(Point p1, Point p2, CC C)

{

p1 = p1 - C.o;

p2 = p2 - C.o;

if (dcmp(0.25 \* Dot(p1+p2, p1+p2) - C.r \* C.r) < 0)

return Cross(p1, p2) / 2;

else

{

double ang = adjust(atan2(p2.y, p2.x) - atan2(p1.y, p1.x));

return C.r \* C.r \* ang / 2;

}

}

double CircleIntersectPolygon(Point poly[], int n, CC C)

{

static Point p[3\*maxn];

int m = 0;

for (int i = 0; i < n; i++)

{

vector<Point> sol;

double t1, t2;

p[m++] = poly[i];

if (GetCircleSegmentIntersection(C, Line(poly[i], poly[(i+1)%n]), t1, t2))

{

p[m++] = Line(poly[i], poly[(i+1)%n]).point(t1);

p[m++] = Line(poly[i], poly[(i+1)%n]).point(t2);

}

}

double area = 0;

for (int i = 0; i < m; i++)

area += CircleIntersectTriangle(p[i], p[i+1], C);

return fabs(area);

}

// Circles Union in n\*m plant

double CirclesUnion(Circle C[maxn], int tot, int n, int m)

{

static Point border[4];

static Circle tC[maxn];

border[0] = Point(0, 0);

border[1] = Point(n, 0);

border[2] = Point(n, m);

border[3] = Point(0, m);

int ttot = 0;

for (int i = 0; i < tot; i++)

{

bool flag = true;

for (int j = 0; j < tot; j++) if (i != j)

{

double d = Length(C[i].o - C[j].o);

if (!dcmp(d))

{

if (dcmp(C[i].r - C[j].r) < 0 || dcmp(C[i].r - C[j].r) == 0 && i < j)

flag = false;

}

else if (dcmp(d + C[i].r - C[j].r) <= 0)

flag = false;

}

if (flag)

tC[ttot++] = C[i];

}

copy(tC, tC+ttot, C);

tot = ttot;

double area = 0;

for (int i = 0; i < tot; i++)

{

vector<Rad> vec;

vec.push\_back(Rad(0, 0));

vec.push\_back(Rad(2\*pi, 0));

int k = 0;

for (int j = 0; j < tot; j++) if (i != j)

{

double a1, a2;

if (GetCircleCircleIntersection(C[i], C[j], a1, a2))

{

vec.push\_back(Rad(a1, 1));

vec.push\_back(Rad(a2, -1));

if (a1 > a2)

k++;

}

}

for (int j = 0; j < 4; j++)

{

double t1, t2;

Line L = Line(border[j], border[(j+1)%4]);

if (GetCircleSegmentIntersection(C[i], L, t1, t2))

{

double a1 = adjust(Angle(L.point(t1) - C[i].o)),

a2 = adjust(Angle(L.point(t2) - C[i].o));

vec.push\_back(Rad(a1, 1));

vec.push\_back(Rad(a2, -1));

if (a1 > a2)

k++;

}

}

sort(vec.begin(), vec.end());

for (int j = 0; j+1 < vec.size(); j++)

{

k += vec[j].second;

if (!k)

{

double a = vec[j+1].first - vec[j].first;

area += C[i].r \* C[i].r \* (a - sin(a)) / 2;

area += Cross(C[i].point(vec[j].first), C[i].point(vec[j+1].first)) / 2;

}

}

}

for (int i = 0; i < 4; i++)

{

Line L = Line(border[i], border[(i+1)%4]);

vector<Rad> vec;

vec.push\_back(Rad(0, 0));

vec.push\_back(Rad(1, 0));

for (int j = 0; j < tot; j++)

{

double t1, t2;

if (GetCircleSegmentIntersection(C[j], L, t1, t2))

{

vec.push\_back(Rad(t1, 1));

vec.push\_back(Rad(t2, -1));

}

}

sort(vec.begin(), vec.end());

int k = 0;

for (int j = 0; j+1 < vec.size(); j++)

{

k += vec[j].second;

if (k)

area += Cross(L.point(vec[j].first), L.point(vec[j+1].first)) / 2;

}

}

return area;

}

int GetTangents(Circle A, Circle B, Point \*a, Point \*b)

{

int cnt = 0;

if (A.r < B.r) swap(A, B), swap(a, b);

double d2 = Length2(A.o-B.o);

double rdiff = A.r - B.r;

double rsum = A.r + B.r;

if (dcmp(d2 - rdiff \* rdiff) < 0) return 0; // contain

// B.o - A.o !!!!

double base = Angle(B.o - A.o);

if (dcmp(d2) == 0 && dcmp(rdiff) == 0) return -1; // coincide

if (dcmp(d2 - rdiff \* rdiff) == 0) // inscibe

{

a[cnt] = A.point(base);

b[cnt] = B.point(base);

cnt++;

return 1;

}

// two external common tangent

double ang = acos(rdiff / sqrt(d2));

a[cnt] = A.point(base + ang);

b[cnt] = B.point(base + ang);

cnt++;

a[cnt] = A.point(base - ang);

b[cnt] = B.point(base - ang);

cnt++;

if (dcmp(d2 - rsum \* rsum) == 0) // one internal common tangent

{

a[cnt] = A.point(base);

b[cnt] = B.point(pi + base);

cnt++;

}

else if (dcmp(d2 - rsum \* rsum) > 0) // two

{

ang = acos(rsum / sqrt(d2));

a[cnt] = A.point(base + ang);

b[cnt] = B.point(pi + base + ang);

cnt++;

a[cnt] = A.point(base - ang);

b[cnt] = B.point(pi + base - ang);

cnt++;

}

return cnt;

}

int GetTangents(const Circle &C, CP p, vector<double> &rad)

{

Vector u = p - C.o;

double base = Angle(u);

double dist = Length(u);

if (dcmp(dist-C.r) < 0) return 0; // in circle

if (dcmp(dist-C.r) == 0)

{

rad.push\_back(base);

return 1;

}

double ang = acos(C.r / dist);

rad.push\_back(base - ang);

rad.push\_back(base + ang);

return 2;

}