```cpp
#include <bits/stdc++.h>
#define FOR(i,a,b) for(int i=(a),_b=(b); i<=_b; ++i)
#define FORD(i,a,b) for(int i=(a),_b=(b); i>=_b; --i)
#define REP(i,a) for(int i=0,_a=(a); i < _a; ++i)

#define DEBUG(X) { cout << #X << " = " << X << endl; }
#define PR(A,n)  { cout << #A << " = "; FOR(_,1,n) cout << A[_] << ' '; cout << endl; }
#define PR0(A,n) { cout << #A << " = "; REP(_,n) cout << A[_] << ' '; cout << endl; }

#define ll long long
#define SZ(x) ((int) (x).size())
#define sqr(x) ((x) * (x))
#define double long double
using namespace std;

const double PI = acos((double) -1.0);
const double EPS = 1e-10;

struct Point {
    double x, y;
    Point() {}
    Point(double x, double y) : x(x), y(y) {}
    Point operator - (const Point& a) const {
        return Point(x-a.x, y-a.y);
    }
    double len() {
        return hypot(x, y);
    }
};

struct Circle : Point {
    double r;
};

double cir_area_solve(double a, double b, double c) {
    return acos((a*a + b*b - c*c) / 2 / a / b);
}
double cir_area_cut(double a, double r) {
    double s1 = a * r * r / 2;
    double s2 = sin(a) * r * r / 2;
    return s1 - s2;
}

double commonCircleArea(Circle c1, Circle c2) { //return the common area of two circle
    if (c1.r < c2.r) swap(c1, c2);
    double d = (c1 - c2).len();
        if (d + c2.r <= c1.r + EPS) return c2.r*c2.r*acos(-1.0);
        if (d >= c1.r + c2.r - EPS) return 0.0;
    double a1 = cir_area_solve(d, c1.r, c2.r);
    double a2 = cir_area_solve(d, c2.r, c1.r);
    return cir_area_cut(a1*2, c1.r) + cir_area_cut(a2*2, c2.r);
}

int main() {
    cout << (fixed) << setprecision(12);
    ll x1, y1, r1, x2, y2, r2;
```

```
    while (cin >> x1 >> y1 >> r1 >> x2 >> y2 >> r2) {
        Circle c1; c1.x = x1; c1.y = y1; c1.r = r1;
        Circle c2; c2.x = x2; c2.y = y2; c2.r = r2;
        cout << commonCircleArea(c1, c2) << endl;
    }
}
```

```
#include <cstdio>
#include <cstdlib>
#include <cstring>
#include <cassert>
#include <ctime>
#include <cmath>
#include <algorithm>
#include <string>
#include <vector>
#include <deque>
#include <queue>
#include <list>
#include <set>
#include <map>

#define pb push_back
#define mp make_pair
#define TASKNAME ""

#ifdef LOCAL
#define eprintf(...) fprintf(stderr,__VA_ARGS__)
#else
#define eprintf(...)
#endif

#define TIMESTAMP(x) eprintf("[" #x "] Time = %.3lfs\n",clock()*1.0/
CLOCKS_PER_SEC)

#ifdef linux
#define LLD "%lld"
#else
#define LLD "%I64d"
#endif

#define sz(x) ((int)(x).size())

using namespace std;

typedef double ld;
typedef long long ll;
typedef vector<ll> vll;
typedef vector<int> vi;
typedef vector<vi> vvi;
typedef vector<bool> vb;
typedef vector<vb> vvb;
typedef pair<int, int> pii;

const int inf = 1e9;
const double eps = 1e-9;
const double INF = inf;
const double EPS = eps;

struct point{
  ld x,y,z;
  point(ld x,ld y, ld z):x(x),y(y),z(z){}
  point(){ x = y = z = 0;}
  bool operator==(const point& a) const{
    return fabs(x - a.x) < eps && fabs(y - a.y) < eps && fabs(z - a.z) < eps;
  }
```

```
  bool load(){
    double _x,_y,_z;
    if (scanf("%lf%lf%lf",&_x,&_y,&_z) != 3) return 0;
    x = _x, y = _y, z = _z;
    return 1;
  }
  void print(){
    printf("%lf %lf %lf\n",(double)x, (double)y, (double)z);
  }
  void eprint(){
    eprintf("%lf %lf %lf\n",(double)x, (double)y, (double)z);
  }
  ld dist2() const {
    return x*x+y*y+z*z;
  }
  ld dist() const{
    return sqrt(dist2());
  }
};

inline point operator+(const point& a,const point& b){
  return point(a.x+b.x,a.y+b.y,a.z+b.z);
}

inline point operator-(const point& a,const point& b){
  return point(a.x-b.x,a.y-b.y,a.z-b.z);
}

inline point operator*(const point& a,ld t){
  return point(a.x*t, a.y*t, a.z*t);
}

inline ld det(ld a,ld b,ld c,ld d){
  return a*d - b*c;
}

inline ld det(ld a11,ld a12,ld a13,ld a21,ld a22, ld a23, ld a31, ld a32,ld a33){
  return a11*det(a22,a23,a32,a33) - a12*det(a21,a23,a31,a33) + a13*det(a21,a22,a31,a32);
}

int sgn(ld x){
  return (x > eps) - (x < -eps);
}

inline ld det(const point& a,const point& b,const point& c){
  return det(a.x,a.y,a.z,b.x,b.y,b.z,c.x,c.y,c.z);
}

point vp(const point& a,const point& b){
  return point(det(a.y,a.z,b.y,b.z),-det(a.x,a.z,b.x,b.z),det(a.x,a.y,b.x,b.y));
}

ld sp(const point& a,const point& b){
  return a.x * b.x + a.y * b.y + a.z * b.z;
}

```

```cpp
// BEGIN ALGO

// vp is vector product (point)
// sp is scalar product (ld)

struct line{
  point p,v;
  line(){}; /*BOXNEXT*/
  line(const point& p,const point& v):p(p),v(v){
    assert(!(v == point()));
  }
  bool on(const point& pt) const{
    return vp(pt - p, v) == point();
  }
};
struct plane {
  point n;
  ld d;
  plane() : d(0) {}
  plane(const point &p1, const point &p2,
        const point &p3) {
    n = vp(p2 - p1, p3 - p1);
    d = -sp(n, p1);
    assert(side(p1) == 0);
    assert(side(p2) == 0);
    assert(side(p3) == 0);
  }
  int side(const point &p) const {
    return sgn(sp(n, p) + d);
  }
};

int intersec(const line& l1, const line& l2,
             point& res){
  assert(!(l1.v == point()));
  assert(!(l2.v == point()));
  if (vp(l1.v,l2.v) == point()){
    if (vp(l1.v, l1.p - l2.p) == point())
      return 2; // same
    return 0; // parallel
  }
  point n = vp(l1.v,l2.v);
  point p = l2.p - l1.p;
  if (sgn(sp(n,p)))
    return 0; // skew
  ld t;
  if (sgn(n.x))
    t = (p.y * l2.v.z - p.z * l2.v.y) / n.x;
  else if (sgn(n.y))
    t = (p.z * l2.v.x - p.x * l2.v.z) / n.y;
  else if (sgn(n.z))
    t = (p.x * l2.v.y - p.y * l2.v.x) / n.z;
  else
    assert(false);
  res = l1.p + l1.v * t;
  assert(l1.on(res)); assert(l2.on(res));
  return 1; // intersects
}
```

```
ld dist(const line& l1,const line& l2){
  point ret = l1.p - l2.p; /*BOXNEXT*/
  ret = ret - l1.v * (sp(l1.v,ret) / l1.v.dist2());
  point tmp = l2.v - l1.v *
              (sp(l1.v,l2.v) / l1.v.dist2());
  if (sgn(tmp.dist2()))  /*BOXNEXT*/
    ret = ret - tmp * (sp(tmp,ret) / tmp.dist2());
  assert(fabs(sp(ret,l1.v)) < eps);
  assert(fabs(sp(ret,tmp)) < eps);
  assert(fabs(sp(ret,l2.v)) < eps);
  return ret.dist();
}

void closest(const line& l1,const line& l2,
             point& p1,point& p2){
  if (vp(l1.v,l2.v) == point()){
    p1 = l1.p;
    p2 = l2.p - l1.v * /*BOXNEXT*/
         (sp(l1.v,l2.p - l1.p) / l1.v.dist2());
    return;
  }
  point p = l2.p   - l1.p;
  ld t1 = (
           sp(l1.v,p) * l2.v.dist2() -
           sp(l1.v,l2.v) * sp(l2.v,p)
          ) / vp(l1.v,l2.v).dist2();
  ld t2 = (
             sp(l2.v,l1.v) * sp(l1.v,p) -
             sp(l2.v,p) * l1.v.dist2()
          ) / vp(l2.v,l1.v).dist2();
  p1 = l1.p + l1.v * t1;
  p2 = l2.p + l2.v * t2;
  assert(l1.on(p1));
  assert(l2.on(p2));
}

int cross(const line &l, const plane &pl,
          point &res) {
  ld d = sp(pl.n, l.v);
  if (sgn(d) == 0) {
    return (pl.side(l.p) == 0) ? 2 : 0;
  }
  ld t = (-sp(pl.n, l.p) - pl.d) / d;
  res = l.p + l.v * t;
  #ifdef DEBUG
  assert(pl.side(res) == 0);
  #endif
  return 1;
}

bool cross(const plane& p1,const plane& p2,
           const plane& p3, point& res){
  ld d = det(p1.n,p2.n,p3.n);
  if (sgn(d) == 0) {
     return false;
  }
  point px(p1.n.x, p2.n.x, p3.n.x);
  point py(p1.n.y, p2.n.y, p3.n.y);
  point pz(p1.n.z, p2.n.z, p3.n.z);
```

```
  point p(-p1.d,-p2.d,-p3.d);
  res = point(
              det(p,py,pz)/d,
              det(px,p,pz)/d,
              det(px,py,p)/d
             );
  #ifdef DEBUG
  assert(p1.side(res) == 0);
  assert(p2.side(res) == 0);
  assert(p3.side(res) == 0);
  #endif
  return true;
}

int cross(const plane &p1, const plane &p2,
          line &res) {
  res.v = vp(p1.n, p2.n);
  if (res.v == point()) {
    if ( (p1.n * (p1.d / p1.n.dist2())) ==
         (p2.n * (p2.d / p2.n.dist2())) )
      return 2;
    else
      return 0;
  }
  plane p3;
  p3.n = res.v;
  p3.d = 0;
  bool ret = cross(p1, p2, p3, res.p);
  assert(ret);
  assert(p1.side(res.p) == 0);
  assert(p2.side(res.p) == 0);
  return 1;
}
// END ALGO



int main(){
  freopen(TASKNAME".in","r",stdin);
  #ifdef LOCAL
  freopen(TASKNAME".out","w",stdout);
  #endif

  {
    line l;
    l.p = point(1, 1, 1);
    l.v = point(1, 0, -1);
    plane p(point(10, 11, 12), point(9, 8, 7), point(1, 3, 2));
    point res;
    assert(cross(l, p, res) == 1);
  }
  {
    plane p1(point(1, 2, 3), point(4, 5, 6), point(-1, 5, -4));
    plane p2(point(3, 2, 1), point(6, 5, 4), point(239, 17, -42));
    line l;
    assert(cross(p1, p2, l) == 1);
  }
  {
    plane p1(point(1, 2, 3), point(4, 5, 6), point(-1, 5, -4));
```

```
    plane p2(point(1, 2, 3), point(7, 8, 9), point(3, -1, 10));
    line l;
    assert(cross(p1, p2, l) == 2);
  }
  {
    plane p1(point(1, 2, 3), point(4, 5, 6), point(-1, 5, -4));
    plane p2(point(1, 2, 4), point(4, 5, 7), point(-1, 5, -3));
    line l;
    assert(cross(p1, p2, l) == 0);
  }

  line l1,l2;
  while (l1.p.load()){
    l1.v.load(); l1.v = l1.v - l1.p;
    l2.p.load();
    l2.v.load(); l2.v = l2.v - l2.p;
    if (l1.v == point() || l2.v == point()) continue;
    point res;
    int cnt = intersec(l1,l2,res);
    ld d = dist(l1,l2);
    if (fabs(d) < eps)
      assert(cnt >= 1);
    else
      assert(cnt == 0);
    point p1,p2;
    closest(l1,l2,p1,p2);
    assert(fabs((p1-p2).dist() - d) < eps);
  }
  plane a(point(1,0,0),point(0,1,0),point(0,0,1));
  plane b(point(-1,0,0),point(0,-1,0),point(0,0,-1));
  line l;
  assert((cross(a,b,l))==0);
  TIMESTAMP(end);
  return 0;
}
```

```cpp
// Two-phase simplex algorithm for solving linear programs of the form
//
//     maximize     c^T x
//     subject to   Ax <= b
//                  x >= 0
//
// INPUT: A -- an m x n matrix
//        b -- an m-dimensional vector
//        c -- an n-dimensional vector
//        x -- a vector where the optimal solution will be stored
//
// OUTPUT: value of the optimal solution (infinity if unbounded
//         above, nan if infeasible)
//
// To use this code, create an LPSolver object with A, b, and c as
// arguments.  Then, call Solve(x).

#include <iostream>
#include <iomanip>
#include <vector>
#include <cmath>
#include <limits>

using namespace std;

typedef long double DOUBLE;
typedef vector<DOUBLE> VD;
typedef vector<VD> VVD;
typedef vector<int> VI;

const DOUBLE EPS = 1e-9;

struct LPSolver {
	int m, n;
	VI B, N;
	VVD D;

	LPSolver(const VVD &A, const VD &b, const VD &c): m(b.size()), n(c.size()), N(n + 1), B(m), D(m + 2, VD(n + 2))
	{
		for (int i = 0; i < m; i++)
			for (int j = 0; j < n; j++)
				D[i][j] = A[i][j];
		for (int i = 0; i < m; i++)
		{
			B[i] = n + i;
			D[i][n] = -1;
			D[i][n + 1] = b[i];
		}
		for (int j = 0; j < n; j++)
		{
			N[j] = j;
			D[m][j] = -c[j];
		}
		N[n] = -1;
		D[m + 1][n] = 1;
	}

	void Pivot(int r, int s)
```

```
	{
		for (int i = 0; i < m + 2; i++)
			if (i != r)
				for (int j = 0; j < n + 2; j++)
					if (j != s)
						D[i][j] -= D[r][j] * D[i][s] / D[r][s];
		for (int j = 0; j < n + 2; j++)
			if (j != s)
				D[r][j] /= D[r][s];
		for (int i = 0; i < m + 2; i++)
			if (i != r)
				D[i][s] /= -D[r][s];
		D[r][s] = 1.0 / D[r][s];
		swap(B[r], N[s]);
	}

	bool Simplex(int phase)
	{
		int x = phase == 1 ? m + 1 : m;
		while (true)
		{
			int s = -1;
			for (int j = 0; j <= n; j++)
			{
				if (phase == 2 && N[j] == -1)
					continue;
				if (s == -1 || D[x][j] < D[x][s] || D[x][j] == D[x][s] && N[j] < N[s])
					s = j;
			}
			if (D[x][s] > -EPS)
				return true;
			int r = -1;
			for (int i = 0; i < m; i++)
			{
				if (D[i][s] < EPS)
					continue;
				if (r == -1 || D[i][n + 1] / D[i][s] < D[r][n + 1] / D[r][s] ||
					(D[i][n + 1] / D[i][s]) == (D[r][n + 1] / D[r][s]) && B[i] < B[r])
					r = i;
			}
			if (r == -1)
				return false;
			Pivot(r, s);
		}
	}

	DOUBLE Solve(VD &x)
	{
		int r = 0;
		for (int i = 1; i < m; i++)
			if (D[i][n + 1] < D[r][n + 1])
				r = i;
		if (D[r][n + 1] < -EPS)
		{
			Pivot(r, n);
```

```
			if (!Simplex(1) || D[m + 1][n + 1] < -EPS)
				return -numeric_limits<DOUBLE>::infinity();
			for (int i = 0; i < m; i++)
				if (B[i] == -1)
				{
					int s = -1;
					for (int j = 0; j <= n; j++)
						if (s == -1 || D[i][j] < D[i][s] || D[i][j] == D[i][s] && N[j] < N[s])
							s = j;
					Pivot(i, s);
				}
		}
		if (!Simplex(2))
			return numeric_limits<DOUBLE>::infinity();
		x = VD(n);
		for (int i = 0; i < m; i++)
			if (B[i] < n)
				x[B[i]] = D[i][n + 1];
		return D[m][n + 1];
	}
};

int main()
{
	//	Example:
	//	maximize	x1 - x2
	//	subject to
	//		6x1 -  x2      <= 10
	//		-x1 - 5x2      <= -4
	//		 x1 + 5x2 + x3 <= 5
	//		-x1 - 5x2 - x3 <= -5
	//
	//		x1 >= 0
	//		x2 >= 0
	//		x3 >= 0

	const int n = 4;
	const int m = 3;
	DOUBLE _A[n][m] =
	{
		{6, -1, 0},
		{-1, -5, 0},
		{1, 5, 1},
		{-1, -5, -1}
	};
	DOUBLE _b[n] = {10, -4, 5, -5};
	DOUBLE _c[m] = {1, -1, 0};

	VVD A(n);
	VD b(_b, _b + n);
	VD c(_c, _c + m);
	for (int i = 0; i < n; i++)
		A[i] = VD(_A[i], _A[i] + m);

	LPSolver solver(A, b, c);
	VD x;
	DOUBLE value = solver.Solve(x);

```

```
        cerr << "VALUE: " << value << endl; // VALUE: 1.29032
        cerr << "SOLUTION:"; // SOLUTION: 1.74194 0.451613 1
        for (size_t i = 0; i < x.size(); i++)
                cerr << " " << x[i];
        cerr << endl;
        return 0;
}
```

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

#include <iostream>
#include <vector>
#include <cmath>
#include <cassert>

using namespace std;

double INF = 1e100;
double EPS = 1e-12;

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 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 << ")";
}

// 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;
}

// compute 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)));
}

// compute distance between point (x,y,z) and plane ax+by+cz=d
```

```cpp
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);
}

// determine if lines from a to b and c to d are parallel or 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> CircleCircleIntersection(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;
}

// This code computes the area or centroid of a (possibly nonconvex)
// polygon, assuming that the coordinates are listed in a clockwise or
// counterclockwise fashion.  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;
}

// tests whether or not a given polygon (in CW or CCW order) 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;
}

int main() {

  // expected: (-5,2)
  cerr << RotateCCW90(PT(2,5)) << endl;

  // expected: (5,-2)
  cerr << RotateCW90(PT(2,5)) << endl;

  // expected: (-5,2)
  cerr << RotateCCW(PT(2,5),M_PI/2) << endl;

  // expected: (5,2)
  cerr << ProjectPointLine(PT(-5,-2), PT(10,4), PT(3,7)) << endl;

  // expected: (5,2) (7.5,3) (2.5,1)
  cerr << ProjectPointSegment(PT(-5,-2), PT(10,4), PT(3,7)) << " "
       << ProjectPointSegment(PT(7.5,3), PT(10,4), PT(3,7)) << " "
       << ProjectPointSegment(PT(-5,-2), PT(2.5,1), PT(3,7)) << endl;

  // expected: 6.78903
  cerr << DistancePointPlane(4,-4,3,2,-2,5,-8) << endl;

  // expected: 1 0 1
  cerr << LinesParallel(PT(1,1), PT(3,5), PT(2,1), PT(4,5)) << " "
       << LinesParallel(PT(1,1), PT(3,5), PT(2,0), PT(4,5)) << " "
       << LinesParallel(PT(1,1), PT(3,5), PT(5,9), PT(7,13)) << endl;

  // expected: 0 0 1
```

```
    cerr << LinesCollinear(PT(1,1), PT(3,5), PT(2,1), PT(4,5)) << " "
         << LinesCollinear(PT(1,1), PT(3,5), PT(2,0), PT(4,5)) << " "
         << LinesCollinear(PT(1,1), PT(3,5), PT(5,9), PT(7,13)) << endl;

    // expected: 1 1 1 0
    cerr << SegmentsIntersect(PT(0,0), PT(2,4), PT(3,1), PT(-1,3)) << " "
         << SegmentsIntersect(PT(0,0), PT(2,4), PT(4,3), PT(0,5)) << " "
         << SegmentsIntersect(PT(0,0), PT(2,4), PT(2,-1), PT(-2,1)) << " "
         << SegmentsIntersect(PT(0,0), PT(2,4), PT(5,5), PT(1,7)) << endl;

    // expected: (1,2)
    cerr << ComputeLineIntersection(PT(0,0), PT(2,4), PT(3,1), PT(-1,3)) << endl;

    // expected: (1,1)
    cerr << ComputeCircleCenter(PT(-3,4), PT(6,1), PT(4,5)) << endl;

    vector<PT> v;
    v.push_back(PT(0,0));
    v.push_back(PT(5,0));
    v.push_back(PT(5,5));
    v.push_back(PT(0,5));

    // expected: 1 1 1 0 0
    cerr << PointInPolygon(v, PT(2,2)) << " "
         << PointInPolygon(v, PT(2,0)) << " "
         << PointInPolygon(v, PT(0,2)) << " "
         << PointInPolygon(v, PT(5,2)) << " "
         << PointInPolygon(v, PT(2,5)) << endl;

    // expected: 0 1 1 1 1
    cerr << PointOnPolygon(v, PT(2,2)) << " "
         << PointOnPolygon(v, PT(2,0)) << " "
         << PointOnPolygon(v, PT(0,2)) << " "
         << PointOnPolygon(v, PT(5,2)) << " "
         << PointOnPolygon(v, PT(2,5)) << endl;

    // expected: (1,6)
    //           (5,4) (4,5)
    //           blank line
    //           (4,5) (5,4)
    //           blank line
    //           (4,5) (5,4)
    vector<PT> u = CircleLineIntersection(PT(0,6), PT(2,6), PT(1,1), 5);
    for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;
    u = CircleLineIntersection(PT(0,9), PT(9,0), PT(1,1), 5);
    for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;
    u = CircleCircleIntersection(PT(1,1), PT(10,10), 5, 5);
    for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;
    u = CircleCircleIntersection(PT(1,1), PT(8,8), 5, 5);
    for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;
    u = CircleCircleIntersection(PT(1,1), PT(4.5,4.5), 10, sqrt(2.0)/2.0);
    for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;
    u = CircleCircleIntersection(PT(1,1), PT(4.5,4.5), 5, sqrt(2.0)/2.0);
    for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;

    // area should be 5.0
    // centroid should be (1.1666666, 1.166666)
    PT pa[] = { PT(0,0), PT(5,0), PT(1,1), PT(0,5) };
    vector<PT> p(pa, pa+4);
```

```
  PT c = ComputeCentroid(p);
  cerr << "Area: " << ComputeArea(p) << endl;
  cerr << "Centroid: " << c << endl;

  return 0;
}
```

```
@ECHO OFF

SET generator=%1
SET checkProgram=%2
SET correctProgram=%3

for /l %%x in (1, 0, 1) do (
%generator% > test.in
if %errorlevel% neq 0 exit
%checkProgram% < test.in > test1.out
if %errorlevel% neq 0 exit
%correctProgram% < test.in > test2.out
if %errorlevel% neq 0 exit
FC test1.out test2.out
if %errorlevel% neq 0 exit
del test.in test1.out test2.out
timeout 1 /NOBREAK
)
```