ACM/ICPC: Competitive Programming Notebook

Lucas Mattioli, Marlon Mendes, Simião Carvalho

Contents

Points

Comparing floating point values

Returns true if double values a and b are equal

```
const double EPS { 1e-9 };
bool equals(double a, double b)
{
    return fabs(a - b) < EPS;
}
```

Listing 1: equals

Lines

General equation of a line

Non-normalized form: ax + by + c = 0

```
class Line {
public:
    double a;
    double b;
    double c;

    Line(double av, double bv, double cv) : a(av), b(bv), c(cv) {}

    Line(const Point& p, const Point& q)
    {
        a = p.y - q.y;
        b = q.x - p.x;
        c = p.x * q.y - p.y * q.x;
    }
};
```

Listing 2: General equation of a line

General equation of a line normalized

```
class Line {
public:
    double a;
    double b;
    double c;

    Line(double av, double bv, double cv) : a(av), b(bv), c(cv) {}

    Line(const Point& p, const Point& q)
    {
        a = p.y - q.y;
        b = q.x - p.x;
        c = p.x * q.y - p.y * q.x;

        auto k = a ? a : b;

        a /= k;
        b /= k;
```

```
        c /= k;
    }
};
```

Listing 3: General equation of a line

Point on a line

Is the given point located on the given Line?

```
template<typename T>
struct Line {
    bool contains(const Point<T>& P) const
    {
        return equals(a*P.x + b*P.y + c, 0);
    }
};
```

Listing 4: Point on line

Equal and parallel lines

```
template<typename T>
struct Line {
    bool operator==(const Line& r) const
    {
        auto k = a ? a : b;
        auto s = r.a ? r.a : r.b;

        return equals(a*s, r.a*k) && equals(b*s, r.b*k)
            && equals(c*s, r.c*k);
    }

    bool parallel(const Line& r) const
    {
        auto det = a*r.b - b*r.a;
        return det == 0 and !(*this == r);
    }
};
```

Orthogonal

```
template<typename T>
struct Line
{
    bool orthogonal(const Line& r) const
    {
        return equals(a * r.a + b * r.b, 0);
    }
};
```

Intersection

```
const int INF { -1 };
template<typename T>
std::pair<int, Point<T>> intersections(const Line<T>& r, const Line<T>& s)
```

```
{
    auto det = r.a * s.b - r.b * s.a;

    if (equals(det, 0))      // Coincidentes ou paralelas
    {
        int qtd = (r == s) ? INF : 0;
        return std::pair<int, Point<T>>(qtd, Point());
    } else                   // Concorrentes
    {
        auto x = (-r.c * s.b + s.c * r.b) / det;
        auto y = (-s.c * r.a + r.c * s.a) / det;

        return std::pair<int, Point<T>>(1, Point<T>(x, y));
    }
}
```

Angle between lines

```
template<typename T>
double angle(const Point<T>& P, const Point<T>& Q,
             const Point<T>& R, const Point<T>& S)
{
    auto ux = P.x - Q.x;
    auto uy = P.y - Q.y;

    auto vx = R.x - S.x;
    auto vy = R.y - S.y;

    auto num = ux * vx + uy * vy;
    auto den = hypot(ux, uy) * hypot(vx, vy);
    return acos(num / den);
}
```

Distance to point

```
#include <cmath>
#include <iostream>

template<typename T>
struct Point {
    T x, y;
};

template<typename T>
struct Line {
    T a, b, c;

    double distance(const Point<T>& p) const
    {
        return fabs(a*p.x + b*p.y + c)/hypot(a, b);
    }

    Point<T> closest(const Point<T>& p) const
    {
        auto den = (a*a + b*b);

        auto x = (b*(b*p.x - a*p.y) - a*c)/den;
        auto y = (a*(-b*p.x + a*p.y) - b*c)/den;
```

```
        return Point<T> { x, y };
    }
};

int main()
{
    Point<double> P { 1.0, 4.0 };
    Line<double> r { 1.0, -1.0, 0 };

    std::cout << "Distance: " << r.distance(P) << '\n';

    auto Q = r.closest(P);

    std::cout << "Closest: Q = (" << Q.x << ", " << Q.y << ")\n";

    return 0;
}
```

Bissector / Mediatriz

```
typename<template T>
Line<T> perpendicular_bisector(const Point<T>& P, const  Point<T>& Q)
{
    auto a = 2*(Q.x - P.x);
    auto b = 2*(Q.y - P.y);
    auto c = (P.x * P.x + P.y * P.y) - (Q.x * Q.x + Q.y * Q.y);

    return Line<T>(a, b, c);
}
```

Orientation between point and line

```
// D = 0: R pertence a reta PQ
// D > 0: R a esquerda da reta PQ
// D < 0: R a direita da reta PQ
template<typename T>
T D(const Point<T>& P, const Point<T>& Q, const Point<T>& R)
{
    return (P.x * Q.y + P.y * R.x + Q.x * R.y) -
           (R.x * Q.y + R.y * P.x + Q.x * P.y);
}
```

Line segments

Contains point

```
template<typename T>
bool contains(const Point<T>& A, const Point<T>& B, const Point<T>& P)
{
    if (P == A || P == B)
        return true;

    auto xmin = min(A.x, B.x);
    auto xmax = max(A.x, B.x);
    auto ymin = min(A.y, B.y);
    auto ymax = max(A.y, B.y);
```

```
    if (P.x < xmin || P.x > xmax || P.y < ymin || P.y > ymax)
        return false;

    return equals((P.y - A.y)*(B.x - A.x), (P.x - A.x)*(B.y - A.y));
}
```

Closest point

```
template<typename T>
struct Segment {
    Point<T> A, B;

    bool contains(const Point<T>& P) const
    {
        if (equals(A.x, B.x))
            return min(A.y, B.y) <= P.y and P.y <= max(A.y, B.y);
        else
            return min(A.x, B.x) <= P.x and P.x <= max(A.x, B.x);
    }

    Point<T> closest(const Point<T>& P)
    {
        Line<T> r(A, B);
        auto Q = r.closest(P);

        if (this->contains(Q))
            return Q;

        auto distA = P.distanceTo(A);
        auto distB = P.distanceTo(B);

        if (distA <= distB)
            return A;
        else
            return B;
    }
}
```

Intersectin with segment

```
template<typename T>
class Segment {
public:
    Point<T> A, B;

    bool intersect(const Segment& s) const
    {
        auto d1 = D(A, B, s.A);
        auto d2 = D(A, B, s.B);

        if ((equals(d1, 0) && contains(s.A)) ||
           (equals(d2, 0) && contains(s.B)))
            return true;

        auto d3 = D(s.A, s.B, A);
        auto d4 = D(s.A, s.B, B);

        if ((equals(d3, 0) && s.contains(A)) ||
```

```
            (equals(d4, 0) && s.contains(B)))
             return true;

        return (d1 * d2 < 0) && (d3 * d4 < 0);
    }
}
```

Vectors

Angle between vector and X-axis

Returns an angle in radians in the interval $[-\pi, +\pi]$. A positive angle means in the COUNTER-clockwise direction. A negative angle is measured in the clockwise direction. Note that the atan2 swaped the parameters.

```
inline double angle(double x, double y) {
    return atan2(y, x);
}
```

Listing 5: angle between X-axis and vectorx, y

Translation

```
Point translate(const Point& P, double dx, double dy)
{
    return Point { P.x + dx, P.y + dy };
}
```

Listing 6: Translate point

Rotation around origin

```
Point rotate(const Point& P, double angle)
{
    auto x = cos(angle) * P.x - sin(angle) * P.y;
    auto y = sin(angle) * P.x + cos(angle) * P.y;

    return Point { x, y };
}
```

Rotation around another point

```
Point rotate(const Point& P, double angle, const Point& C)
{
    auto Q = translate(P, -C.x, -C.y);
    Q = rotate(Q, angle);
    Q = translate(Q, C.x, C.y);

    return Q;
}
```

Rotation around origin 3D

$$R_x = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix}, \quad R_y = \begin{bmatrix} \cos\theta & 0 & -\sin\theta \\ 0 & 1 & 0 \\ \sin\theta & 0 & \cos\theta \end{bmatrix}$$

$$R_z = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Scale

```
Point scale(double sx, double sy)
{
    return Point(sx * P.x, sy * P.y);
}
```

Listing 7: Scale vector by a factor of sx and sy

Normalization

```
Vector normalize(const Vector& v)
{
    auto len = v.length();
    auto u = Vector(v.x / len, v.y / len);

    return u;
}
```

Listing 8: Returns a unit vector with the same direction as the given vector

Dot product

$$< \vec{u}, \vec{v} >= \vec{u} \cdot \vec{v} = u_x v_x + u_y v_y = |\vec{u}||\vec{v}| \cos\theta$$

```
double dot_product(const Vector& u, const Vector& v)
{
    return u.x * v.x + u.y * v.y;
}
```

Angle between vectors

```
double angle(const Vector& u, const Vector& v)
{
    auto lu = u.length();
    auto lv = v.length();
    auto prod = dot_product(u, v);

    return acos(prod/(lu * lv));
}
```

Cross product

$$u \times v = \begin{vmatrix} \vec{i} & \vec{j} & \vec{k} \\ u_x & u_y & u_z \\ v_x & v_y & v_z \end{vmatrix}$$

- $|\vec{u} \times \vec{v}| = |\vec{u}||\vec{v}| \sin\theta$
- where $\vec{i}, \vec{j}, \vec{k}$ are unity vectors on the same direction and orientation as x, y, z, respectively
- the result vector $\vec{w}$ is orthogonal to both $\vec{u}$ and $\vec{v}$
- it is the area of the parallelogram formed by $\vec{u}$ and $\vec{v}$

```
Vector cross_product(const Vector& u, const Vector& v)
{
    auto x = u.y*v.z - v.y*u.z;
    auto y = u.z*v.x - u.x*v.z;
    auto z = u.x*v.y - u.y*v.x;

    return Vector(x, y, z);
}
```

Circles

Definition

```
template<typename T>
struct Circle {
    Point<T> C;
    T r;
};
```

Perimeter, Area

```
template<typename T>
struct Circle
{
    double perimeter() const
    {
        return 2.0 * PI * r;
    }

    double area() const
    {
```

```
        return PI * r * r;
    }

    double arc(double theta) const
    {
        return theta * r;
    }

    double chord(double theta) const
    {
        return 2 * r * sin(theta/2);
    }

  double sector(double theta) const
    {
        return (theta * r * r)/2;
    }

    double segment(double a) const
    {
        auto c = chord(a);
        auto s = (r + r + c)/2.0;
        auto T = sqrt(s*(s - r)*(s - r)*(s - c));

        return sector(a) - T;
    }

};
```

From 2 points

```
#include <optional>

template<typename T>
struct Circle {
    static std::optional<Circle>
    from_2_points_and_r(const Point<T>& P, const Point<T>& Q, T r)
    {
        double d2 = (P.x - Q.x) * (P.x - Q.x) + (P.y - Q.y) * (P.y - Q.y);
        double det = r * r / d2 - 0.25;

        if (det < 0.0)
            return { };

        double h = sqrt(det);

        auto x = (P.x + Q.x) * 0.5 + (P.y - Q.y) * h;
        auto y = (P.y + Q.y) * 0.5 + (Q.x - P.x) * h;

        return Circle { Point(x, y), r };
    }
}
```

From 3 points

```
#include <optional>

template<typename T>
struct Circle {
```

```
    static std::optional<Circle>
    from_3_points(const Point<T>& P, const Point<T>& Q, const Point<T>& R)
    {
        auto a = 2*(Q.x - P.x);
        auto b = 2*(Q.y - P.y);
        auto c = 2*(R.x - P.x);
        auto d = 2*(R.y - P.y);

        auto det = a*d - b*c;
        if (equals(det, 0))
            return { };

        auto k1 = (Q.x*Q.x + Q.y*Q.y) - (P.x*P.x + P.y*P.y);
        auto k2 = (R.x*R.x + R.y*R.y) - (P.x*P.x + P.y*P.y);

        auto cx = (k1*d - k2*b)/det;
        auto cy = (a*k2 - c*k1)/det;

        Point<T> C { cx, cy };
        auto r = distance(P, C);

        return Circle<T>(C, r);
    }
};
```

Intersection between 2 circles

```
#include <variant>
#include <vector>

const int oo { 2000000000 };

template<typename T> std::variant<int, std::vector<Point<T>>>
intersection(const Circle<T>& c1, const Circle<T>& c2)
{
    double d = distance(c1.C, c2.C);

    if (d > c1.r + c2.r or d < fabs(c1.r - c2.r))
        return 0;

    if (equals(d, 0.0) and equals(c1.r, c2.r))
        return oo;

    auto a = (c1.r * c1.r - c2.r * c2.r + d * d)/(2 * d);
    auto h = sqrt(c1.r * c1.r - a * a);

    auto x = c1.C.x + (a/d)*(c2.C.x - c1.C.x);
    auto y = c1.C.y + (a/d)*(c2.C.y - c1.C.y);

    auto P = Point<T> { x, y };

    x = P.x + (h/d)*(c2.C.y - c1.C.y);
    y = P.y - (h/d)*(c2.C.x - c1.C.x);

    auto P1 = Point<T> { x, y };

    x = P.x - (h/d)*(c2.C.y - c1.C.y);
    y = P.y + (h/d)*(c2.C.x - c1.C.x);

    auto P2 = Point<T> { x, y };
```

```
    return std::vector<Point<T>> { P1, P2 };
}
```

Intersection between circle and line

```
template<typename T> std::vector<Point<T>>
intersection(const Circle<T>& c, const Point<T>& P, const Point<T>& Q)
{
    auto a = pow(Q.x - P.x, 2.0) + pow(Q.y - P.y, 2.0);
    auto b = 2*((Q.x - P.x) * (P.x - c.C.x) + (Q.y - P.y) * (P.y - c.C.y));
    auto d = pow(c.C.x, 2.0) + pow(c.C.y, 2.0) + pow(P.x, 2.0)
        + pow(P.y, 2.0) + 2*(c.C.x * P.x + c.C.y * P.y);
    auto D = b * b - 4 * a * d;

    if (D < 0)
        return { };
    else if (equals(D, 0))
    {
        auto u = -b/(2*a);
        auto x = P.x + u*(Q.x - P.x);
        auto y = P.y + u*(Q.y - P.y);
        return { Point { x, y } };
    }

    auto u = (-b + sqrt(D))/(2*a);

    auto x = P.x + u*(Q.x - P.x);
    auto y = P.y + u*(Q.y - P.y);

    auto P1 = Point { x, y };

    u = (-b - sqrt(D))/(2*a);

    x = P.x + u*(Q.x - P.x);
    y = P.y + u*(Q.y - P.y);

    auto P2 = Point { x, y };

    return { P1, P2 };
}
```

Intersection between circle and point

```
template<typename T>
struct Circle {
    Point<T> C;
    T r;

    enum { IN, ON, OUT } PointPosition;

    PointPosition position(const Point& P) const
    {
        auto d = dist(P, C);

        return equals(d, r) ? ON : (d < r ? IN : OUT);
    }
};
```