

ACM/ICPC: Competitive Programming Notebook

Lucas Mattioli, Marlon Mendes, Simião Carvalho

Contents

| | |
|---|-----------|
| Points | 3 |
| Comparing floating point values | 3 |
| Lines | 3 |
| General equation of a line | 3 |
| General equation of a line normalized | 3 |
| Point on a line | 4 |
| Equal and parallel lines | 4 |
| Orthogonal | 4 |
| Intersection | 4 |
| Angle between lines | 5 |
| Distance to point | 5 |
| Bisector / Mediatriz | 6 |
| Orientation between point and line | 6 |
| Line segments | 6 |
| Contains point | 6 |
| Closest point | 7 |
| Intersectin with segment | 7 |
| Vectors | 8 |
| Angle between vector and X-axis | 8 |
| Translation | 8 |
| Rotation around origin | 8 |
| Rotation around another point | 8 |
| Rotation around origin 3D | 9 |
| Scale | 9 |
| Normalization | 9 |
| Dot product | 9 |
| Angle between vectors | 10 |
| Cross product | 10 |
| Circles | 10 |
| Definition | 10 |
| Perimeter, Area | 10 |
| From 2 points | 11 |
| From 3 points | 11 |
| Intersection between 2 circles | 12 |
| Intersection between circle and line | 13 |
| Intersection between circle and point | 13 |

Points

Comparing floating point values

Returns true if double values a and b are equal

```
1  const double EPS { 1e-9 };
2  bool equals(double a, double b)
3  {
4      return fabs(a - b) < EPS;
5  }
```

Listing 1: equals

Lines

General equation of a line

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

```
1  class Line {
2  public:
3      double a;
4      double b;
5      double c;
6
7      Line(double av, double bv, double cv) : a(av), b(bv), c(cv) {}
8
9      Line(const Point& p, const Point& q)
10     {
11         a = p.y - q.y;
12         b = q.x - p.x;
13         c = p.x * q.y - p.y * q.x;
14     }
15 };
```

Listing 2: General equation of a line

General equation of a line normalized

```
1  class Line {
2  public:
3      double a;
4      double b;
5      double c;
6
7      Line(double av, double bv, double cv) : a(av), b(bv), c(cv) {}
8
9      Line(const Point& p, const Point& q)
10     {
11         a = p.y - q.y;
12         b = q.x - p.x;
13         c = p.x * q.y - p.y * q.x;
14
15         auto k = a ? a : b;
16
17         a /= k;
18         b /= k;
```

```
19         c /= k;  
20     }  
21 };
```

Listing 3: General equation of a line

Point on a line

Is the given point located on the given Line?

```
1 template<typename T>  
2 struct Line {  
3     bool contains(const Point<T>& P) const  
4     {  
5         return equals(a*P.x + b*P.y + c, 0);  
6     }  
7 };
```

Listing 4: Point on line

Equal and parallel lines

```
1 template<typename T>  
2 struct Line {  
3     bool operator==(const Line& r) const  
4     {  
5         auto k = a ? a : b;  
6         auto s = r.a ? r.a : r.b;  
7  
8         return equals(a*s, r.a*k) && equals(b*s, r.b*k)  
9             && equals(c*s, r.c*k);  
10    }  
11  
12    bool parallel(const Line& r) const  
13    {  
14        auto det = a*r.b - b*r.a;  
15        return det == 0 and !(*this == r);  
16    }  
17 };
```

Orthogonal

```
1 template<typename T>  
2 struct Line  
3 {  
4     bool orthogonal(const Line& r) const  
5     {  
6         return equals(a * r.a + b * r.b, 0);  
7     }  
8 };
```

Intersection

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

```
4 {  
6     auto det = r.a * s.b - r.b * s.a;  
8     if (equals(det, 0)) // Coincidentes ou paralelas  
9     {  
10        int qtd = (r == s) ? INF : 0;  
11        return std::pair<int, Point<T>>(qtd, Point());  
12    } else // Concorrentes  
13    {  
14        auto x = (-r.c * s.b + s.c * r.b) / det;  
15        auto y = (-s.c * r.a + r.c * s.a) / det;  
16  
17        return std::pair<int, Point<T>>(1, Point<T>(x, y));  
18    }
```

Angle between lines

```
1 template<typename T>  
2 double angle(const Point<T>& P, const Point<T>& Q,  
3             const Point<T>& R, const Point<T>& S)  
4 {  
5     auto ux = P.x - Q.x;  
6     auto uy = P.y - Q.y;  
7  
8     auto vx = R.x - S.x;  
9     auto vy = R.y - S.y;  
10  
11     auto num = ux * vx + uy * vy;  
12     auto den = hypot(ux, uy) * hypot(vx, vy);  
13     return acos(num / den);  
14 }
```

Distance to point

```
1 #include <cmath>  
2 #include <iostream>  
3  
4 template<typename T>  
5 struct Point {  
6     T x, y;  
7 };  
8  
9 template<typename T>  
10 struct Line {  
11     T a, b, c;  
12  
13     double distance(const Point<T>& p) const  
14     {  
15         return fabs(a*p.x + b*p.y + c)/hypot(a, b);  
16     }  
17  
18     Point<T> closest(const Point<T>& p) const  
19     {  
20         auto den = (a*a + b*b);  
21  
22         auto x = (b*(b*p.x - a*p.y) - a*c)/den;  
23         auto y = (a*(-b*p.x + a*p.y) - b*c)/den;
```

```
25         return Point<T> { x, y };
26     }
27 };
28
29 int main()
30 {
31     Point<double> P { 1.0, 4.0 };
32     Line<double> r { 1.0, -1.0, 0 };
33
34     std::cout << "Distance: " << r.distance(P) << '\n';
35
36     auto Q = r.closest(P);
37
38     std::cout << "Closest: Q = (" << Q.x << ", " << Q.y << ")\n";
39
40     return 0;
41 }
```

Bissector / Mediatriz

```
typename<template T>
2 Line<T> perpendicular_bisector(const Point<T>& P, const Point<T>& Q)
3 {
4     auto a = 2*(Q.x - P.x);
5     auto b = 2*(Q.y - P.y);
6     auto c = (P.x * P.x + P.y * P.y) - (Q.x * Q.x + Q.y * Q.y);
7
8     return Line<T>(a, b, c);
9 }
```

Orientation between point and line

```
typedef pair<long long, long long> ii;
2
3 // D = 0: R lies on line PQ
4 // D > 0: R is to the left of line PQ
5 // D < 0: R is to the right of line PQ
6 long long D(const ii &a, const ii &b, const ii &c) {
7     return (a.first * b.second + a.second * c.first + b.first * c.second)
8           - (c.first * b.second + c.second * a.first + b.first * a.second);
9 }
```

Line segments

Contains point

```
1 template<typename T>
2 bool contains(const Point<T>& A, const Point<T>& B, const Point<T>& P)
3 {
4     if (P == A || P == B)
5         return true;
6
7     auto xmin = min(A.x, B.x);
8     auto xmax = max(A.x, B.x);
9     auto ymin = min(A.y, B.y);
10    auto ymax = max(A.y, B.y);
```

```
11 |  
12 |     if (P.x < xmin || P.x > xmax || P.y < ymin || P.y > ymax)  
13 |         return false;  
  
14 |  
15 |     return equals((P.y - A.y)*(B.x - A.x), (P.x - A.x)*(B.y - A.y));  
16 | }
```

Closest point

```
1 | template<typename T>  
2 | struct Segment {  
3 |     Point<T> A, B;  
  
4 |  
5 |     bool contains(const Point<T>& P) const  
6 |     {  
7 |         if (equals(A.x, B.x))  
8 |             return min(A.y, B.y) <= P.y and P.y <= max(A.y, B.y);  
9 |         else  
10 |             return min(A.x, B.x) <= P.x and P.x <= max(A.x, B.x);  
11 |     }  
  
12 |  
13 |     Point<T> closest(const Point<T>& P)  
14 |     {  
15 |         Line<T> r(A, B);  
16 |         auto Q = r.closest(P);  
  
17 |         if (this->contains(Q))  
18 |             return Q;  
  
19 |         auto distA = P.distanceTo(A);  
20 |         auto distB = P.distanceTo(B);  
  
21 |         if (distA <= distB)  
22 |             return A;  
23 |         else  
24 |             return B;  
25 |     }  
26 | }  
27 |  
28 |  
29 | }
```

Intersectin with segment

```
1 | template<typename T>  
2 | class Segment {  
3 | public:  
4 |     Point<T> A, B;  
  
5 |  
6 |     bool intersect(const Segment& s) const  
7 |     {  
8 |         auto d1 = D(A, B, s.A);  
9 |         auto d2 = D(A, B, s.B);  
  
10 |         if ((equals(d1, 0) && contains(s.A)) ||  
11 |             (equals(d2, 0) && contains(s.B)))  
12 |             return true;  
  
13 |         auto d3 = D(s.A, s.B, A);  
14 |         auto d4 = D(s.A, s.B, B);  
  
15 |         if ((equals(d3, 0) && s.contains(A)) ||  
16 |             (equals(d4, 0) && s.contains(B)))  
17 |             return true;  
18 |     }  
19 | }
```

```
        (equals(d4, 0) && s.contains(B)))  
20        return true;  
  
22        return (d1 * d2 < 0) && (d3 * d4 < 0);  
24    }
```

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.

```
1 inline double angle(double x, double y) {  
    return atan2(y, x);  
3 }
```

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

Translation

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

Listing 6: Translate point

Rotation around origin

```
Point rotate(const Point& P, double angle)  
2 {  
    auto x = cos(angle) * P.x - sin(angle) * P.y;  
4    auto y = sin(angle) * P.x + cos(angle) * P.y;  
  
6    return Point { x, y };  
}
```

Rotation around another point

```
1 Point rotate(const Point& P, double angle, const Point& C)  
{  
3     auto Q = translate(P, -C.x, -C.y);  
    Q = rotate(Q, angle);  
5     Q = translate(Q, C.x, C.y);  
  
7     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)
2 {
    return Point(sx * P.x, sy * P.y);
4 }
```

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

Normalization

```
Vector normalize(const Vector& v)
2 {
    auto len = v.length();
    auto u = Vector(v.x / len, v.y / len);
4
    return u;
6
}
```

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

Dot product

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

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

Angle between vectors

```
double angle(const Vector& u, const Vector& v)
2 {
    auto lu = u.length();
    auto lv = v.length();
    auto prod = dot_product(u, v);
    return acos(prod/(lu * lv));
8 }
```

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)
2 {
    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);
8 }
```

Circles

Definition

```
template<typename T>
2 struct Circle {
    Point<T> C;
    T r;
4 };
};
```

Perimeter, Area

```
template<typename T>
2 struct Circle
{
    double perimeter() const
    {
        return 2.0 * PI * r;
    }
    double area() const
    {
10 }
```

```

    return PI * r * r;
12 }

14 double arc(double theta) const
15 {
16     return theta * r;
17 }

18 double chord(double theta) const
19 {
20     return 2 * r * sin(theta/2);
21 }

22 }

24 double sector(double theta) const
25 {
26     return (theta * r * r)/2;
27 }

28 double segment(double a) const
29 {
30     auto c = chord(a);
31     auto s = (r + r + c)/2.0;
32     auto T = sqrt(s*(s - r)*(s - r)*(s - c));
33
34     return sector(a) - T;
35 }
36 }
37 };

```

From 2 points

```

#include <optional>
2
template<typename T>
4 struct Circle {
    static std::optional<Circle>
6     from_2_points_and_r(const Point<T>& P, const Point<T>& Q, T r)
    {
8         double d2 = (P.x - Q.x) * (P.x - Q.x) + (P.y - Q.y) * (P.y - Q.y);
9         double det = r * r / d2 - 0.25;

10
11         if (det < 0.0)
12             return { };

13
14         double h = sqrt(det);

15
16         auto x = (P.x + Q.x) * 0.5 + (P.y - Q.y) * h;
17         auto y = (P.y + Q.y) * 0.5 + (Q.x - P.x) * h;

18         return Circle { Point(x, y), r };
19     }
20 }

```

From 3 points

```

#include <optional>
2
template<typename T>
4 struct Circle {

```

```

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

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

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

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

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

26         return Circle<T>(C, r);
      }
28 };

```

Intersection between 2 circles

```

1  #include <variant>
   #include <vector>
3
   const int oo { 2000000000 };
5
   template<typename T> std::variant<int, std::vector<Point<T>>>
7   intersection(const Circle<T>& c1, const Circle<T>& c2)
   {
9       double d = distance(c1.C, c2.C);

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

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

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

17      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);

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

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

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

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

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

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

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

33

```

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

Intersection between circle and line

```
template<typename T> std::vector<Point<T>>  
2 intersection(const Circle<T>& c, const Point<T>& P, const Point<T>& Q)  
{  
4     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));  
6     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);  
8     auto D = b * b - 4 * a * d;  
  
10    if (D < 0)  
        return { };  
12    else if (equals(D, 0))  
    {  
14        auto u = -b/(2*a);  
        auto x = P.x + u*(Q.x - P.x);  
16        auto y = P.y + u*(Q.y - P.y);  
        return { Point { x, y } };  
18    }  
  
20    auto u = (-b + sqrt(D))/(2*a);  
  
22    auto x = P.x + u*(Q.x - P.x);  
    auto y = P.y + u*(Q.y - P.y);  
24  
    auto P1 = Point { x, y };  
26  
    u = (-b - sqrt(D))/(2*a);  
28  
    x = P.x + u*(Q.x - P.x);  
30    y = P.y + u*(Q.y - P.y);  
32  
    auto P2 = Point { x, y };  
34  
    return { P1, P2 };  
}
```

Intersection between circle and point

```
1 template<typename T>  
struct Circle {  
3     Point<T> C;  
     T r;  
5  
     enum { IN, ON, OUT } PointPosition;  
7  
     PointPosition position(const Point& P) const  
9     {  
         auto d = dist(P, C);  
11  
         return equals(d, r) ? ON : (d < r ? IN : OUT);  
13     }  
};
```