

ACM/ICPC: Competitive Programming Notebook

Lucas Mattioli, Marlon Mendes, Simião Carvalho

Contents

Points	4
Comparing floating point values	4
Lines	4
General equation of a line	4
General equation of a line normalized	4
Point on a line	5
Equal and parallel lines	5
Orthogonal	5
Intersection	5
Angle between lines	6
Distance to point	6
Bisector / Mediatriz	7
Orientation between point and line	7
Line segments	7
Contains point	7
Closest point	8
Intersectin with segment	8
Vectors	9
Angle between vector and X-axis	9
Translation	9
Rotation around origin	9
Rotation around another point	9
Rotation around origin 3D	10
Scale	10
Normalization	10
Dot product	10
Angle between vectors	11
Cross product	11
Circles	11
Definition	11
Perimeter, Area	11
From 2 points	12
From 3 points	12
Intersection between 2 circles	13
Intersection between circle and line	14
Intersection between circle and point	14
Triangles	15
Perimeter	15
Area	15
Side classification	16
By sides	16
By angles	16
Important points	17

Barycenter	17
Incenter	17
Orthocenter	17
Circumcircle	18
Quadrilaterals	19
Area	19
Trapezium	19
Quadrilateral	19
Intersection between rectangles	20

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 }
```

Bisector / 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

```
1 Point scale(double sx, double sy)
2 {
3     return Point(sx * P.x, sy * P.y);
4 }
```

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

Normalization

```
1 Vector normalize(const Vector& v)
2 {
3     auto len = v.length();
4     auto u = Vector(v.x / len, v.y / len);
5
6     return u;
7 }
```

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)
2 {
3     return u.x * v.x + u.y * v.y;
4 }
```

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;
    }
8
    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 };
38

```

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
19         return Circle { Point(x, y), r };
20     }
21 }

```

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    }  
};
```

Triangles

Perimeter

```

1 template<typename T>
2 struct Triangle {
3     Point<T> A, B, C;
4
5     double perimeter() const
6     {
7         auto a = dist(A, B);
8         auto b = dist(B, C);
9         auto c = dist(C, A);
10
11         return a + b + c;
12     }
13 };

```

Area

```

1 // Definição das estruturas Point e Line
2
3 template<typename T>
4 struct Triangle {
5     Point<T> A, B, C;
6
7     double area() const
8     {
9         Line<T> r(A, B);
10
11         auto b = dist(A, B);
12         auto h = r.distance(C);
13
14         return (b * h)/2;
15     }
16 };

```

```

1 // Definição da estrutura Point
2
3 template<typename T>
4 struct Triangle {
5     Point<T> A, B, C;
6
7     double area() const
8     {
9         auto a = dist(A, B);
10        auto b = dist(B, C);
11        auto c = dist(C, A);
12
13        auto s = (a + b + c)/2
14
15        return sqrt(s)*sqrt(s - a)*sqrt(s - b)*sqrt(s - c);
16    }
17 };

```

```

1 // Definição da estrutura Point
2
3 template<typename T>

```

```
struct Triangle {
5   Point<T> A, B, C;

7   double area() const
    {
9       double det = (A.x*B.y + A.y*C.x + B.x*C.y)
                    - (C.x*B.y + C.y*A.x + B.x*A.y);

11      return 0.5 * fabs(det);
13  }
};
```

Side classification

By sides

```
template<typename T>
2 struct Triangle {
    Point<T> A, B, C;

4     enum Sides { EQUILATERAL, ISOSCELES, SCALENE };

6     Sides classification_by_sides() const
    {
8         auto a = dist(A, B);
        auto b = dist(B, C);
        auto c = dist(C, A);

12        if (equals(a, b) and equals(b, c))
            return EQUILATERAL;

14        if (equals(a, b) or equals(a, c) or equals(b, c))
            return ISOSCELES;

16        return SCALENE;
18    }
20 };
```

By angles

```
1 // Defini o da classe Point, da fun o de compara o equals() e
  // da fun o de dist ncia entre pontos dist()
3
4 template<typename T>
5 struct Triangle {
    Point<T> A, B, C;

7     enum Angles { RIGHT, ACUTE, OBTUSE };

9     Angles classification_by_angles() const
    {
11        auto a = dist(A, B);
        auto b = dist(B, C);
        auto c = dist(C, A);

13        auto alpha = acos((a*a - b*b - c*c)/(-2*b*c));
        auto beta = acos((b*b - a*a - c*c)/(-2*a*c));
        auto gamma = acos((c*c - a*a - b*b)/(-2*a*b));
15
17
19    }
```



```

    auto right = PI / 2.0;
21
    if (equals(alpha, right) || equals(beta, right)
23        || equals(gamma, right))
        return RIGHT;
25
    if (alpha > right || beta > right || gamma > right)
27        return OBTUSE;
29
    return ACUTE;
    }
31};

```

Important points

Barycenter

```

1 // Definição da estrutura Point
3
4 template<typename T>
5 struct Triangle {
6     Point<T> A, B, C;
7
8     Point<T> barycenter() const
9     {
10         auto x = (A.x + B.x + C.x) / 3.0;
11         auto y = (A.y + B.y + C.y) / 3.0;
12
13         return Point<T> { x, y };
14     }
15 };

```

Incenter

```

1 template<typename T>
2 struct Triangle {
3     Point<T> A, B, C;
4
5     // Definição dos métodos area() e perimeter()
6
7     double inradius() const
8     {
9         return (2 * area()) / perimeter();
10    }
11
12    Point<T> incenter() const
13    {
14        auto P = perimeter();
15        auto x = (a*A.x + b*B.x + c*C.x)/P;
16        auto y = (a*A.y + b*B.y + c*C.y)/P;
17
18        return { x, y };
19    }
20};

```

Orthocenter

```

1 #include <iostream>
2
3 using namespace std;
4
5 template<typename T>
6 struct Point {
7     T x, y;
8 };
9
10 template<typename T>
11 struct Line {
12     T a, b, c;
13
14     Line(T av, T bv, T cv) : a(av), b(bv), c(cv) {}
15
16     Line(const Point<T>& P, const Point<T>& Q)
17         : a(P.y - Q.y), b(Q.x - P.x), c(P.x * Q.y - Q.x * P.y)
18     {
19     }
20 };
21
22 template<typename T>
23 struct Triangle {
24     Point<T> A, B, C;
25
26     Point<T> orthocenter() const
27     {
28         Line<T> r(A, B), s(A, C);
29
30         Line<T> u { r.b, -r.a, -(C.x*r.b - C.y*r.a) };
31         Line<T> v { s.b, -s.a, -(B.x*s.b - B.y*s.a) };
32
33         auto det = u.a * v.b - u.b * v.a;
34         auto x = (-u.c * v.b + v.c * u.b) / det;
35         auto y = (-v.c * u.a + u.c * v.a) / det;
36
37         return { x, y };
38     }
39 };
40
41 int main()
42 {
43     Point<double> A { 0, 0 }, B { 3, 6 }, C { 9, 1 };
44     Triangle<double> T { A, B, C };
45
46     auto O = T.orthocenter();
47
48     cout << "(" << O.x << ", " << O.y << ")\n";
49
50     return 0;
51 }

```

Circumcircle

```

1 // Definição da estrutura Point e da função de distância
2 // entre pontos dist()
3
4 template<typename T>
5 struct Triangle {
6     Point<T> A, B, C;

```

```
7 // Definição do método area()
9
11 double circumradius() const
12 {
13     auto a = dist(B, C);
14     auto b = dist(A, C);
15     auto c = dist(A, B);
16
17     return (a * b * c) / (4 * area());
18 }
19
20 Point<T> circumcenter() const
21 {
22     auto D = 2 * (A.x * (B.y - C.y) + B.x * (C.y - A.y) + C.x * (A.y - B.y));
23
24     auto A2 = A.x * A.x + A.y * A.y;
25     auto B2 = B.x * B.x + B.y * B.y;
26     auto C2 = C.x * C.x + C.y * C.y;
27
28     auto x = (A2 * (B.y - C.y) + B2 * (C.y - A.y) + C2 * (A.y - B.y)) / D;
29     auto y = (A2 * (C.x - B.x) + B2 * (A.x - C.x) + C2 * (B.x - A.x)) / D;
30
31     return { x, y };
32 };
```

Quadrilaterals

Area

Trapezium

```
template<typename T>
2 struct Trapezium {
3     T b, B, h;
4
5     T area() const
6     {
7         return (b + B) * h / 2;
8     }
9 };
```

Quadrilateral

```
1 // Definição das estruturas Point e Line
2
3 template<typename T>
4 struct Triangle {
5     Point<T> A, B, C;
6
7     double area() const
8     {
9         Line<T> r(A, B);
10
11         auto b = dist(A, B);
12         auto h = r.distance(C);
13     }
```

```
        return (b * h) / 2;
15    }
};
```

Rectangles

From 2 points

```
// Definição da estrutura Point
2
template<typename T>
4 class Rectangle {
public:
6     Point<T> P, Q;
    T b, h;

8     Rectangle(const Point<T>& p, const Point<T>& q) : P(p), Q(q)
10    {
        b = max(P.x, Q.x) - min(P.x, Q.x);
12        h = max(P.y, Q.y) - min(P.y, Q.y);
    }

14    Rectangle(const T& base, const T& height)
16        : P(0, 0), Q(base, height), b(base), h(height) {}
};
```

Intersection between rectangles

```
1 // Definição da classe Point
template<typename T>
3 struct Rectangle {
    // Membros e construtores
5
    Rectangle intersection(const Rectangle& r) const
7    {
        using interval = pair<T, T>;
9
        auto I = interval(min(P.x, Q.x), max(P.x, Q.x));
11        auto U = interval(min(r.P.x, r.Q.x), max(r.P.x, r.Q.x));

13        auto a = max(I.first, U.first);
        auto b = min(I.second, U.second);
15

17        if (b < a)
            return { {-1, -1}, {-1, -1} };

19        I = interval(min(P.y, Q.y), max(P.y, Q.y));
        U = interval(min(r.P.y, r.Q.y), max(r.P.y, r.Q.y));
21

        auto c = max(I.first, U.first);
23        auto d = min(I.second, U.second);

25        if (d < c)
            return { {-1, -1}, {-1, -1} };
27

        inter = Rectangle(Point(a, c), Point(b, d));
29
        return { {a, c}, {b, d} };
31    }
```

|};
