Moscow International Workshop ACM ICPC 2015
Geometry and All, All, All
Lecture notes

Lector: Roman Andreev

Date: November 12, 2015

Created at Thursday 12th November, 2015, 02:16

Contents

Chapter 1: Simple geometric objects

1.1. Point

Any point on a Cartesian plane could be represented as a pair (x, y).

- **Storing points**

```
using coordinate = long double;
struct Point {
  coordinate x, y;
};
```

You can ask, why do we store coordinates in **long double**? What if all coordinates were integer and we want to calculate everything precisely? Maybe it is better to use **int**? Lets note that all **int** (**long long**) numbers are stored in **double** (**long double**) precisely and everything will be fine. Everywhere in code we will be using **coordinate** and will be writing code, thinking about real numbers. If in the end we will understand that we need only integer type, then we will change type to **int** and change eps value to 0.

How can we store vector $\overrightarrow{v}$? Lets associate it with point (v_x, v_y). So now we have bijection between points and vectors.

Lets introduce basic vector operations so we can work with them easier.

- **Vector operations**

```
Point operator + (const Point& a, const Point& b) {
  return {a.x + b.x, a.y + b.y};
}
Point operator - (const Point& a, const Point& b) {
  return {a.x - b.x, a.y - b.y};
}
Point operator * (const Point& a, coordinate c) {
  return {a.x * c, a.y * c};
}
Point operator * (coordinate c, const Point& a) {
  return {a.x * c, a.y * c};
}
Point operator / (const Point& a, coordinate c) {
  return {a.x / c, a.y / c};
}
```

1.2. Scalar and vector product

Definition 1.2.1. $(\overrightarrow{u}, \overrightarrow{v}) = |\overrightarrow{u}||\overrightarrow{v}|\cos\alpha$ — *scalar product.*

Definition 1.2.2. $\overrightarrow{u} \times \overrightarrow{v} = |\overrightarrow{u}||\overrightarrow{v}|\sin\alpha$ — *vector (pseudovector) product.*

We can calculate them with this formulas:

```
// scalar product
coordinate sp(const Point& a, const Point& b) {
  return a.x * b.x + a.y * b.y;
}
// vector product
coordinate vp(const Point& a, const Point& b) {
  return a.x * b.y - a.y * b.x;
}
```

Definition 1.2.3. *Length of a vector is equal to* $|\overrightarrow{a}| = \sqrt{(\overrightarrow{a}, \overrightarrow{a})}$.

```
coordinate len(const Point& a) {
  return sqrt(sp(a, a));
}
// Distance between two points
coordinate dst(const Point& a, const Point& b) {
  return len(a - b);
}
```

Property 1.2.4. $(\overrightarrow{u}, \overrightarrow{v}) > 0$ — vectors are looking in the same direction, < 0 — different directions.

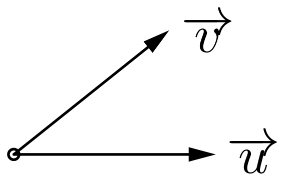


Figure 1: $(\overrightarrow{u}, \overrightarrow{v}) > 0$

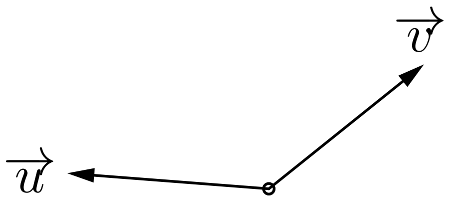


Figure 2: $(\overrightarrow{u}, \overrightarrow{v}) < 0$

Property 1.2.5. $\overrightarrow{u} \times \overrightarrow{v} > 0$ — we go from first vector to second counterclockwise, < 0 — clockwise.

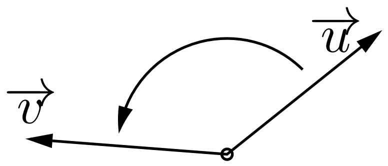


Figure 3: $\overrightarrow{u} \times \overrightarrow{v} > 0$

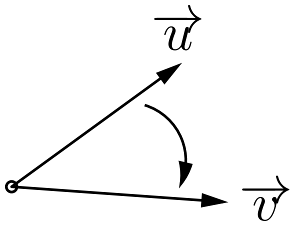


Figure 4: $\overrightarrow{u} \times \overrightarrow{v} < 0$

Property 1.2.6. $|\overrightarrow{u} \times \overrightarrow{v}|$ — area of a parallelogram constructed with our vectors as its sides.

```
// Triangle area
coordinate Area(const Point& a, const Point& b, const Point& c) {
  return fabs(vp(b - a, c - a)) / 2;
}
```

Corollary 1.2.7. $(\overrightarrow{u}, \overrightarrow{v}) = 0$ iff vectors are orthogonal (perpendicular).

Corollary 1.2.8. $\overrightarrow{u} \times \overrightarrow{v} = 0$ iff vectors are collinear (parallel).

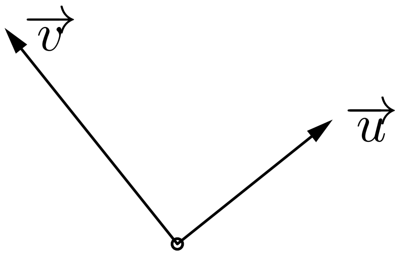


Figure 5: $\overrightarrow{u} \times \overrightarrow{v} = 0$

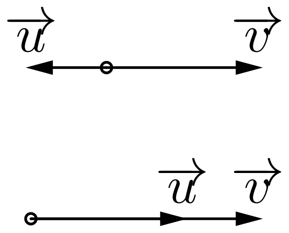


Figure 6: $\overrightarrow{u} \times \overrightarrow{v} = 0$

Note 1.2.9. We have vector $\overrightarrow{v} = (v_x, v_y)$. Lets look at the vector $\overrightarrow{v}^R = (-v_y, v_x)$.

- $|\overrightarrow{v}| = |\overrightarrow{v}^R|$
- $(\overrightarrow{v}, \overrightarrow{v}^R) = v_x(-v_y) + v_x v_y = 0$
- $\overrightarrow{v} \times \overrightarrow{v}^R = v_x^2 + v_y^2 > 0$

This means that v^R is the rotation of v by $\frac{\pi}{2}$ (90 degrees) counterclockwise!

And how can we rotate vector by an arbitrary angle α?

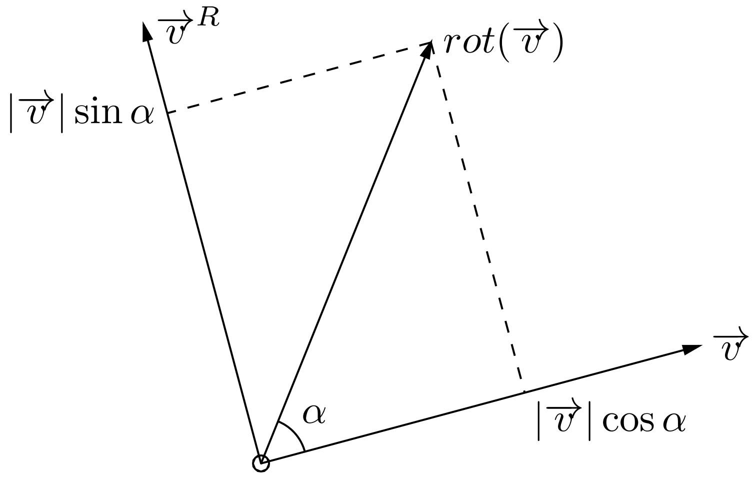

$\overrightarrow{v}^R$
$rot(\overrightarrow{v})$
$|\overrightarrow{v}| \sin\alpha$
$\overrightarrow{v}$
α
$|\overrightarrow{v}| \cos\alpha$

Figure 7: $\mathrm{rot}(\overrightarrow{v}, \alpha) = \overrightarrow{v} \cdot \cos\alpha + \overrightarrow{v}^R \cdot \sin\alpha$

1.3. Line

Line is defined by any two of its points a and b. We want to write line equation in form $f(x, y)$, so for any point $c = (x, y)$ on this line such equation would be true $f(c) = f(x, y) = 0$, and for any other point $f(c) = f(x, y) \neq 0$.

Definition 1.3.1. $\overrightarrow{ab} = \overrightarrow{v}$ *— tangent vector.*

We already know the answer: $f(c) = \overrightarrow{ab} \times \overrightarrow{ac} = \overrightarrow{v} \times \overrightarrow{ac}$. Lets simplify:

$$f(x, y) = v_x(y - a_y) - v_y(x - a_x) = Ax + By + C$$

$$(A, B) = (-v_y, v_x) = \overrightarrow{v}^R$$

Definition 1.3.2. $\overrightarrow{v}^R = \overrightarrow{n}$ *— normal vector.*

$$C = -(v_x a_y - v_y a_x) = -\overrightarrow{v} \times \overrightarrow{a} = -(\overrightarrow{v}^R, \overrightarrow{a}) = -(\overrightarrow{n}, \overrightarrow{a})$$

$$f(c) = (\overrightarrow{n}, \overrightarrow{c}) - (\overrightarrow{n}, \overrightarrow{a})$$

```
struct Line {
  coordinate a, b, c;
  Line(const Point& p1, const Point& p2) {
    a = -(p2.y - p1.y);
    b = p2.x - p1.x;
    c = -(a * p1.x + b * p1.y);
  }
};
```

Corollary 1.3.3. $Ax + By + C = 0$ is a line equation iff $(A, B) \neq (0, 0)$.

Corollary 1.3.4. $Ax + By + C > 0$ iff our point lies to the left of the line, if we are looking at tangent vector(or the same side where normal (A, B) is looking). < 0 iff point lies to the right (opposite side to the normal).

Corollary 1.3.5. This means that **Line** also describes half-plane!

How can we calculate distance between a point and a line? Lets normalize our line to make life easier.

```
void Normalize(Line& l) {
  coordinate d = sqrt(l.a * l.a + l.b * l.b);
  l.a /= d;
  l.b /= d;
  l.c /= d;
}
```

After normalization normal vector will have unit length. And then $(\overrightarrow{n}, \overrightarrow{c})$ will be the length of the projection of our point on a line, orthogonal to our line. This means that $|f(c)|$ is the distance from c to our line.

```
coordinate dst(const Point& p, const Line& l) {
  return fabs(l.a * p.x + l.b * p.y + l.c);
}
```

Note 1.3.6. How can we construct projection of (x_0, y_0) on the normalized line?

Let $d = Ax_0 + By_0 + C$. Now all we need to do is go along normal vector on distance $-d$:

$$\text{projection}(P, L) = P - (L.a \cdot P.x + L.b \cdot P.y + L.c) \cdot Point(L.a, L.b)$$

And how can we reflect (x_0, y_0) across the line? We need to do the same, but go on doubled distance $-2d$:

$$\text{symm}(P, L) = P - 2(L.a \cdot P.x + L.b \cdot P.y + L.c) \cdot Point(L.a, L.b)$$

Note 1.3.7. Two lines on a plane can:

- be equal
- be parallel
- intersect.

1.4. Segments intersection

Definition 1.4.1. *Segment — set of points* $p_1 + t \cdot (p_2 - p_1)\,(t \in [0,\, 1])$*.*
p_1 *u* p_2 *— segment endpoints.*

```
using Segment = pair<Point, Point>;
```

We want to check if two segments are intersecting (have at least one common point) or not. Lets describe all possible scenarios:

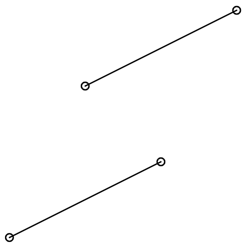


Figure 8: Lines are parallel

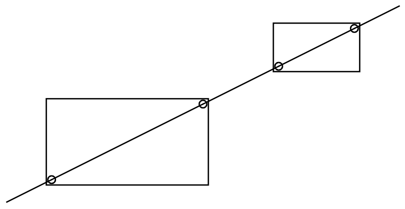


Figure 9: Lines are equal

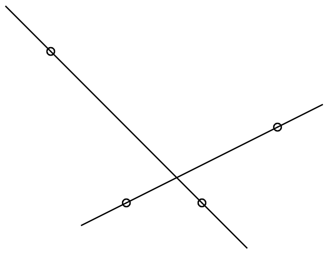


Figure 10: Lines intersect

```
int sign(coordinate x) {
  if (x > eps) return 1;
  if (x < -eps) return -1;
  return 0;
}
// 1-dimensional intersection
using Segment1 = pair<coordinate, coordinate>;
bool Intersects(const Segment1& a, const Segment1& b) {
  // How can two segments NOT intersect?
  return !((max(a.first, a.second) < min(b.first, b.second) - eps) ||
           (max(b.first, b.second) < min(a.first, a.second) - eps));
}
bool Intersects(const Segment& a, const Segment& b) {
  Point v1 = a.second - a.first;
  Point v2 = b.second - b.first;
  if (sign(vp(v1, v2)) == 0) { // Lines are equal or parallel
    if (sign(vp(v1, b.first - a.first)) == 0) { // Lines are equal
      return Intersects({a.first.x, a.second.x}, {b.first.x, b.second.x}) &&
             Intersects({a.first.y, a.second.y}, {b.first.y, b.second.y});
    } else { // Lines are parallel
      return false;
    }
  } else { // Lines are intersecting
    return (sign(vp(v1, b.first - a.first)) *
            sign(vp(v1, b.second - a.first)) <= 0) &&
           (sign(vp(v2, a.first - b.first)) *
            sign(vp(v2, a.second - b.first)) <= 0);
  }
}
```

1.5. Line intersection

We have two line equations and we want to solve this system with two variables:

$$\begin{cases} a_1x + b_1y + c_1 = 0 \\ a_2x + b_2y + c_2 = 0 \end{cases}$$

We can easily write solution using Cramer's rule for matrices 2×2:

$$x = -\frac{\begin{vmatrix} c_1 & b_1 \\ c_2 & b_2 \end{vmatrix}}{\begin{vmatrix} a_1 & b_1 \\ a_2 & b_2 \end{vmatrix}} = -\frac{c_1b_2 - b_1c_2}{a_1b_2 - b_1a_2}$$

$$y = -\frac{\begin{vmatrix} a_1 & c_1 \\ a_2 & c_2 \end{vmatrix}}{\begin{vmatrix} a_1 & b_1 \\ a_2 & b_2 \end{vmatrix}} = -\frac{a_1c_2 - c_1a_2}{a_1b_2 - b_1a_2}$$

Note 1.5.1. In denominator we have vector product of normal vectors, so we are dividing by zero iff lines are parallel.

```
Point Intersect(const Line& a, const Line& b) {
  coordinate d0 = a.a * b.b - a.b * b.a;
  return {-(a.c * b.b - a.b * b.c) / d0,
          -(a.a * b.c - a.c * b.a) / d0};
}
```

1.6. Circle and disc

```
struct Circle {
  Point c;
  coordinate r;
};
```

Circle equation:

$$|p - c| = r$$

$$|p - c|^2 = r^2$$

$$(x - c_x)^2 + (y - c_y)^2 = r^2$$

$$0 = f(x, y) = (x - c_x)^2 + (y - c_y)^2 - r^2$$

1.7. Intersection of a line and a circle

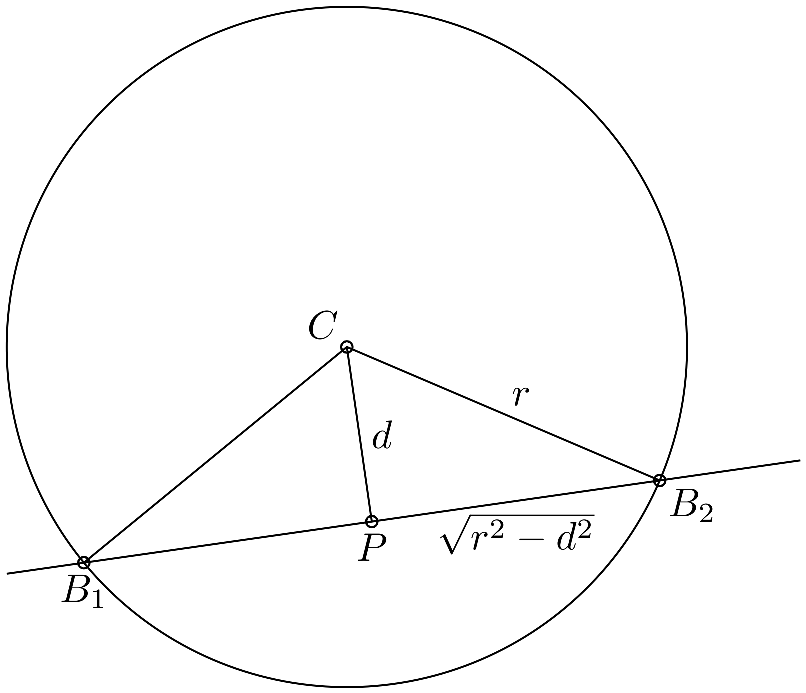


Figure 11: Intersection of a line and a circle

$$P = \text{projection(C, L)}$$
$$\overrightarrow{PB_1} = \text{rot}(Point(L.a, L.b), 90) \cdot \sqrt{r^2 - d^2}$$

1.8. Intersection of two circles

We have two circle equations with two variables and we want to solve this system:

$$\begin{cases} (x - x_0)^2 + (y - y_0)^2 = R_0^2 \\ (x - x_1)^2 + (y - y_1)^2 = R_1^2 \end{cases}$$

Lets subtract second equation from the first one:

$$\begin{cases} (x - x_0)^2 + (y - y_0)^2 = R_0^2 \\ -2xx_0 + x_0^2 - 2yy_0 + y_0^2 - (-2xx_1 + x_1^2 - 2yy_1 + y_1^2) = R_0^2 - R_1^2 \end{cases}$$

Note that now we have line equation instead of the second circle (it is called radical axis). All we have left is to call already written function for intersection a line and a circle.

1.9. Tangent lines from a point to a circle

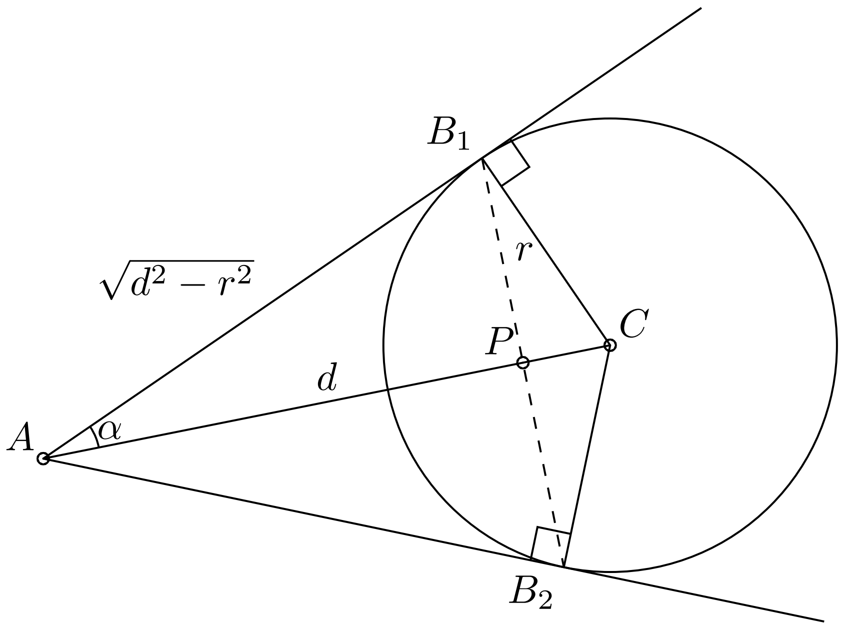


Figure 12: Tangent lines from a point to a circle

$$|AB_1| = |AB_2| = \sqrt{|AC|^2 - |B_1C|^2} = \sqrt{d^2 - r^2}$$

$$\sin\alpha = \frac{r}{d}$$

$$\cos\alpha = \frac{\sqrt{d^2 - r^2}}{d}$$

$$|AP| = |AB_1|\cos\alpha = \sqrt{d^2 - r^2}\frac{\sqrt{d^2 - r^2}}{d} = \frac{d^2 - r^2}{d}$$

$$|B_1P| = |AB_1|\sin\alpha = \sqrt{d^2 - r^2}\frac{r}{d}$$

$$\overrightarrow{AP} = \overrightarrow{AC}\frac{|AP|}{|AC|} = \overrightarrow{AC}\frac{d^2 - r^2}{d^2}$$

$$\overrightarrow{PB_1} = \operatorname{rot}\left(\frac{\overrightarrow{AC}}{|AC|}, 90\right)|B_1P| = \operatorname{rot}\left(\overrightarrow{AC}, 90\right)\frac{r\sqrt{d^2 - r^2}}{d^2}$$

1.10. Common tangent lines of two circles

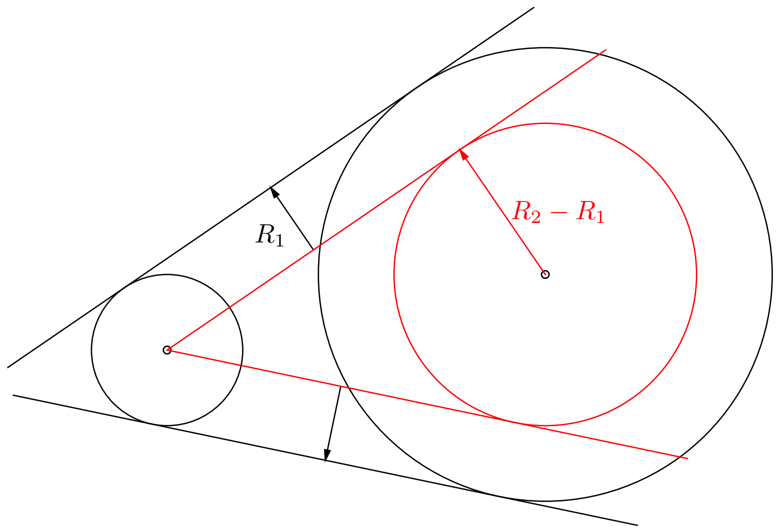


Figure 13: Outside tangent lines

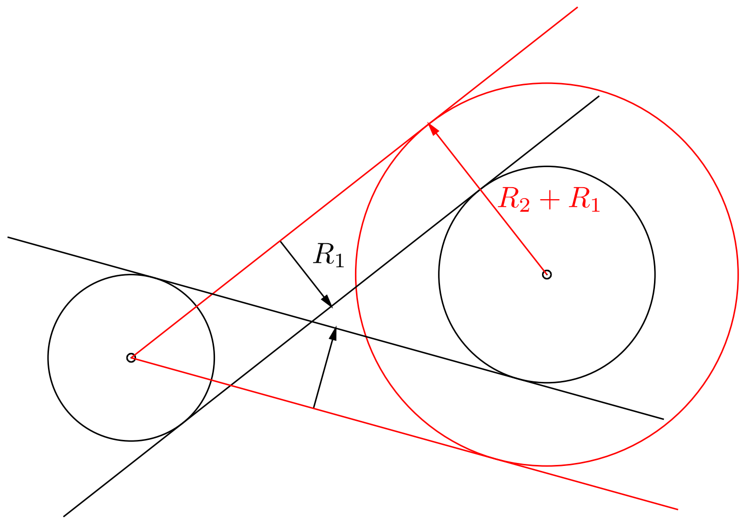


Figure 14: Inside tangent lines

We will construct both tangent types with the same technique — construct tangent line from one of the centers to other circle with changed radius and then move answer in the right direction.

How can we move normalized line in the direction of its normal vector on distance d?

```
line Shift(Line l, coordinate d) {
  l.c -= d;
  return l;
}
```

Chapter 2: Polygons

2.1. Polygon

```
using Polygon = vector<Point>;
```

2.2. Area of non-convex polygon

Lets look at this code:

```
coordinate Area(const Polygon& p) {
  coordinate res = 0;
  for (int i = 0; i < p.size(); i++) {
    res += vp(p[i], p[(i + 1) % p.size()]);
  }
  return fabs(res) / 2;
}
```

It works for all types of polygons!

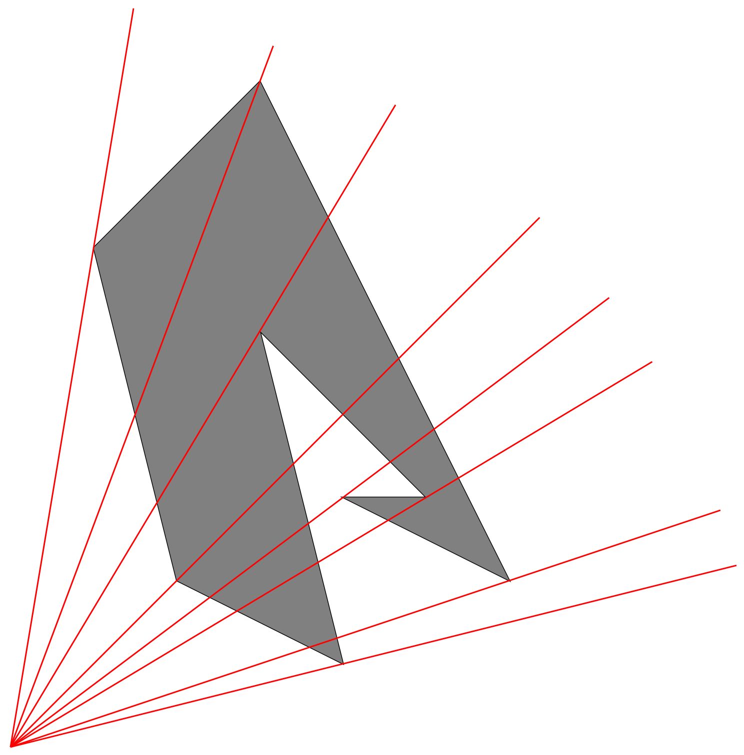


Figure 15: Polygon area

Lets look at one of the angles between rays through vertices as on the picture above. When we will we go through this angle clockwise we will decrease area, when counterclockwise — increase. If we sort segments by the distance from the origin then our adds and decreases will form correct bracket sequence of such structure "()()()()()()". If point $(0, 0)$ lies inside then we need to add event "decrease area by 0" as the first bracket in our sequence. We can see that sum of all this "brackets" will get as exactly what we want — area of intersection of our polygon with the angle between two rays.

2.3. Point location in non-convex polygon in $\mathcal{O}(n)$

Lets look at this code:

```
coordinate Angle(const Point& u, const Point& v) {
  //atan2(x * sin(alpha), x * cos(alpha)) = alpha
  return atan2(vp(u, v), sp(u, v));
}
coordinate SumAngles(const Polygon& p, const Point& a) {
  coordinate res = 0;
  for (int i = 0; i < p.size(); i++) {
    res += Angle(p[i] - a, p[(i + 1) % p.size()] - a);
  }
  return fabs(res);
}
```

We are simply counting number of rotation of our polygon around our point!
What will this function return if our point lies:

- strictly inside polygon? 2π.
- strictly outside polygon? 0.
- on the side? π.
- in the vertex? Angle between sides of this vertex.

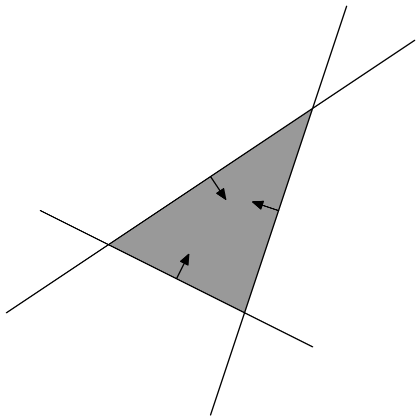
Chapter 3: Half-planes

3.1. Problem description and basic ideas

Lets try to solve classic problem — we have n half-planes and we want to find set of all points lying strictly inside all of this half-planes.

Lemma 3.1.1. Set of all such points is convex.

Proof. Set is called convex if for every two points that lie inside, whole segment between them also lies inside this set. If two points lie in the half-plane then obviously whole segment lies there too. And it is true for all our half-planes. ■



Half-planes are infinite objects and it is not convenient to work with infinity on a computer. So we will add a bounding box — lets say that we are living in a square with very big coordinates, for example $\max(|x|, |y|) < 10^9$. Note that because it is a square we can just add 4 additional half-planes — its sides.

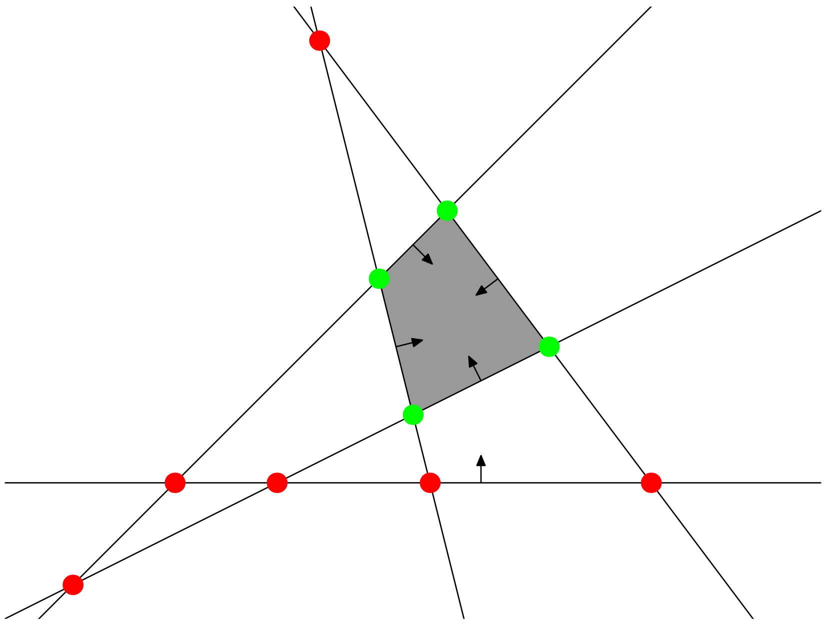
Lemma 3.1.2. Intersection of half-planes is either empty set, or convex polygon (here we are already in a big square).

Proof. Induction by n. Base of induction: for 0 half-planes answer is square — convex polygon. Step of induction — when we add new half-plane we either get empty set or we cut part of a convex polygon so it is still convex. ■

3.2. $\mathcal{O}(n^3)$ algorithm

Lets see what points will be vertices of our answer. Vertex lies in the intersection of two half-planes if we treat them as lines. Lets look at all intersection points. There will be at most $\frac{n(n-1)}{2}$ of them.

Now for every point we will check in $\mathcal{O}(n)$ if it lies inside (not strictly) all of our half-planes and remove wrong ones. In the end we will have either empty set or all vertices of answer polygon in random order. If we want whole polygon then we will sort our points by angle from leftmost point (first step in the convex hull algorithm). But in many real applications we need only one point from intersection. Convenient choice would be the center of mass of all this points, because it will lie strictly inside all of the half-planes (all border points lie not strictly).



In the end we will get $\mathcal{O}(n^3)$. Good optimization: lets shuffle all lines.