Handbook of Geometry for Competitive Programming

Victor Lecomte

Draft April 26, 2018

Contents

Chapter 1

Precision issues and epsilons

Computational geometry very often means working with floating-point values. Even when the input points are all integers, as soon as intermediate steps require things like line intersections, orthogonal projections or circle tangents, we have no choice but to use floating-point numbers to represent coordinates.

Using floating-point numbers comes at a cost: loss of precision. The number of distinct values that can be represented by a data type is limited by its number of bits, and therefore many "simple" values like 0.1 or $\sqrt{2}$ cannot be exactly represented. Worse, even if a and b are exact, there is no guarantee that simple operations like $a + b$, $a - b$ or ab will give an exact result.

Though many people are well aware that those issues exist, they will most often argue that they only cause small imprecisions in the answer in the end, and do not have any major consequence or the behavior of algorithms. In the rest of this chapter, we will show how both those assumptions can sometimes be false, then present some ways in which we *can* make accurate statements about how precision loss affects algorithms, go on with a few practical examples, and finally give some general advice to problem solvers and setters.

1.1 Small imprecisions can become big imprecisions

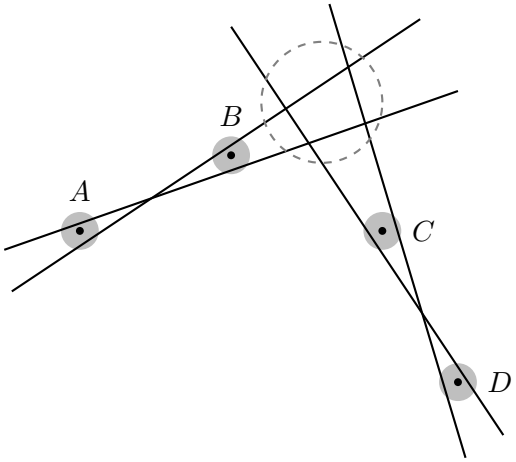
In this section we explore two ways in which very small starting imprecisions can become very large imprecisions in the final output of a program.

1.1.1 When doing numerically unstable computations

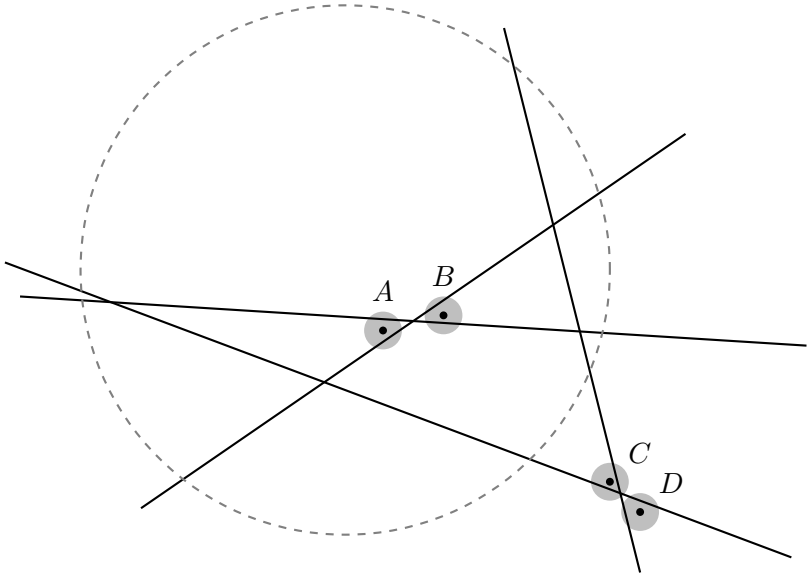
There are some types of computations which can transform small imprecisions into catastrophically large ones, and line intersection is one of them. Imagine you have four points A, B, C, D which were obtained through an previous imprecise process (for example, their position can vary by a distance of at most $r = 10^{-6}$), and you have to compute the intersection of lines AB and CD.

For illustration, we represent the imprecisions on the points with small disk of radius r: the exact position is the black dot, while the small gray disk contains the positions it could take because of imprecisions. The dashed circle gives an idea of where point I might lie.

In the best case, when A, B and C, D aren't too close together, and not too far from the intersection point I, then the imprecision on I isn't too big.



But if those conditions are not respected, the intersection I might vary in a very wide range or even fail to exist, if given the imprecision lines AB and CD end up being parallel, or if A and B (or C and D) end up coinciding.



This shows that finding the intersection of two lines defined by imprecise points is a task that is inherently problematic for floating-point arithmetic, as it can produce wildly incorrect results even if the starting imprecision is quite small.

1.1.2 With large values and accumulation

Another way in which small imprecisions can become big is by accumulation. Problem "Keeping the Dogs Apart", which we treat in more detail in a case study in section 1.4.1, is a very good example of this. In this problem, two dogs run along two polylines at equal speed and you have to find out the minimum distance between them at any point in time.

Even though the problem seems quite easy and the computations do not have anything dangerous for precision (mostly just additions, subtractions and distance computations), it turns out to be a huge precision trap, at least in the most direct implementation.

Let's say we maintain the current distance from the start for both dogs. There are 10^5 polyline segments of length up to $\sqrt{2} \times 10^4$, so this distance can reach $\sqrt{2} \times 10^9$. Besides, to compute the sum, we perform 10^5 sum operations which can all bring a $2^{-53} \approx 1.11 \times 10^{-16}$ relative error if we're using `double`. So in fact the error might reach

$$\left(\sqrt{2} \times 10^9\right) \times 10^5 \times 2^{-53} \approx 0.016$$

Although this is a theoretical computation, the error does actually get quite close to this in practice, and since the tolerance on the answer is 10^{-4} this method actually gives a WA verdict.

This shows that even when only very small precision mistakes are made ($\approx 1.11 \times 10^{-16}$), the overal loss of precision can get very big, and carefully checking the maximal imprecision of your program is very important.

1.2 Small imprecisions can break algorithms

In this section, we explore ways in which small imprecisions can modify the behavior of an algorithm in ways other than just causing further imprecisions.

1.2.1 When making binary decisions

The first scenario we will explore is when we have to make clear-cut decisions, such as deciding if two objects touch.

Let's say we have a line l and a point P computed imprecisely, and we want to figure out if the point lies on the line. Obviously, we cannot simply check if the point we have computed lies on the line, as it might be just slightly off due to imprecision. So the usual approach is to compute the distance from P to l and then figure out if that distance is less than some small value like $\epsilon_{\text{cutoff}} = 10^{-9}$.

While this approach tends to works pretty well in practice, to be sure that this solution works in every case and choose ϵ_{cutoff} properly,[1] we need to know two things. First, we need to know ϵ_{error}, the biggest imprecision that we might make while computing the distance. Secondly, and more critically, we need to know ϵ_{chance}, the smallest distance that point P might be from l while not being on it, in other words, the closest distance that it might be from l "by coincidence".

Only once we have found those two values, and made sure that $\epsilon_{\text{error}} < \epsilon_{\text{chance}}$, can we then choose the value of ϵ_{cutoff} within $[\epsilon_{\text{error}}, \epsilon_{\text{chance}})$.[2] Indeed, if $\epsilon_{\text{cutoff}} < \epsilon_{\text{error}}$, there is a risk that P is on l but we say it is not, while if $\epsilon_{\text{cutoff}} \geq \epsilon_{\text{chance}}$ there is a risk that P is not on l but we say it is.

Even though ϵ_{error} can be easily found with some basic knowledge of floating-point arithmetic and a few multiplications (see next section), finding ϵ_{chance} is often very difficult. It depends directly on which geometric operations were done to find P (intersections, tangents, etc.), and in most cases where ϵ_{chance} can be estimated, it is in fact possible to make the comparison entirely with integers, which is of course the preferred solution.

[1] And motivated problem setters *do* tend to find the worst cases.

[2] In practice you should try to choose ϵ_{cutoff} so that there is a factor of safety on both sides, in case you made mistakes while computing ϵ_{error} or ϵ_{chance}.

1.2.2 By violating basic assumptions

Many algorithms rely on basic geometric axioms in order to provide their results, even though those assumptions are not always easy to track down. This is especially the case for incremental algorithms, like algorithms for building convex hulls. And when those assumptions are violated by using floating-point numbers, this can make algorithms break down in big ways.

Problems of this type typically happen in situation when points are very close together, or are nearly collinear/coplanar. The ways to solve the problem depend a lot on what the algorithm, but tricks like eliminating points that are too close together, or adding random noise to the coordinates to avoid collinearity/coplanarity can be very useful.

For concrete examples of robustness problems and a look into the weird small-scale behavior of some geometric functions, see [1].

1.3 Modelling precision

In this section, we try to build a basic model of precision errors that we can use to obtain a rough but reliable estimate of a program's precision.

1.3.1 The issue with "absolute or relative" error

When the output of a problem is some real value like a distance or an area, the problem statement often specifies a constraint such as: "The answer should be accurate to an absolute or relative error of at most 10^{-5}." While considering the relative accuracy of an answer can be a useful and convenient way to specify the required precision of an answer in some cases (for example in tasks where only addition and multiplication of positive values are performed), we think that for most geometry problems it is unsuitable.

The reason for this is the need to subtract[3] large values of similar magnitude. For example, suppose that we are able to compute two values with relative precision 10^{-6}, such as $A = 1000 \pm 10^{-3}$ and $B = 999 \pm 10^{-3}$. If we compute their difference, we obtain $A - B = 1 \pm 2 \times 10^{-3}$. The absolute error remains of a comparable size, being only multiplied by 2, but on the other hand relative error increases drastically from 10^{-6} to 2×10^{-3} because of the decrease in magnitude. This phenomenon is called *catastrophic cancellation*.

In fact, whenever a certain relative error can affect big numbers, catastrophic cancellation can cause the corresponding absolute error to appear on very small values. The consequence is that if a problem statement has a

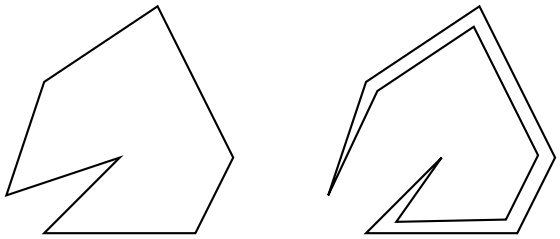
[3]Strictly speaking, we mean both subtraction of values of the same sign, and addition of values of opposite signs.

certain tolerance on the relative error of the answer, and a correct solution has an error close to it for the biggest possible values, then the problem statement also needs to specify a tolerance on the corresponding absolute error in case catastrophic cancellation happens. And since that tolerance on absolute error is at least as tolerant as the tolerance on relative error for all possible values, it makes it redundant. This is why we think that tolerance on "absolute or relative error" is misleading at best.

Catastrophic cancellation shows that relative precision is not a reliable way to think about precision whenever subtractions are involved — and that includes the wide majority of geometry problems. In fact, the most common geometric operations (distances, intersections, even dot/cross products) all involve subtractions of values which could be very similar in magnitude.

Examples of this appear in two of the case studies of section 1.4: in problem "Keeping the Dogs Apart" and when finding the solution of a quadratic equation.

Another example occurs when computing areas of polygons made of imprecise points. Even when the area ends up being small, the imprecision on it can be large if there were computations on large values in intermediate steps, which is the case when the coordinates have large magnitudes.



area	$\gg$	area
absolute error	$\approx$	absolute error
relative error	$\ll$	relative error

Because of this, we advise against trying to use relative error to build precision guarantees on the global scale of a whole algorithm, and we recommend to reason about those based on absolute error instead, as we describe below.

1.3.2 Precision guarantees from IEEE 754

Nearly all implementations of floating-point numbers obey the specifications of the IEEE 754 standard. This includes `float` and `double` in Java and C++, and `long double` in C++. The IEEE 754 standard gives strong guarantees

that ensure floating-point numbers will have similar behavior even in different languages and over different platforms, and gives users a basis to build guarantees on the precision of their computations.

The basic guarantees given by the standard are:

1. decimal values entered in the source code or a file input are represented by the closest representable value;
2. the five basic operations $(+, -, \times, /, \sqrt{x})$ are performed as if they were performed with infinite precision and then rounded to the closest representable value.

There are several implications. First, this means that integers are represented exactly, and basic operations on them $(+, -, \times)$ will have exact results, as long as they are small enough to fit within the significant digits of the type: $\geq 9 \times 10^{15}$ for `double`, and $\geq 1.8 \times 10^{19}$ for `long double`. In particular, `long double` can perform exactly all the operations that a 64-bit integer type can perform.

Secondly, if the inputs are exact, the relative error on the result of any of those five operations $(+, -, \times, /, \sqrt{x})$ will be bounded by a small constant that depends on the number of significant digits in the type.[4] This constant is $< 1.2 \times 10^{-16}$ for `double` and $< 5.5 \times 10^{-20}$ for `long double`. It is called the *machine epsilon* and we will often write it ϵ.

1.3.3 Considering the biggest possible magnitude

We explained earlier why we need to work with absolute error, but since IEEE 754 gives us guarantees in terms of relative errors, we need to consider the biggest magnitude that will be reached during the computations. In other words, if all computations are precise up to a relative error of ϵ, and the magnitude of the values never goes over M, then the absolute error of an operation is at most $M\epsilon$.

This allows us to give good guarantees for numbers obtained after a certain number of $+$ and $-$ operations: a value that is computed in n operations[5] will have an absolute error of at most $nM\epsilon$ compared to the theoretical result.

We can prove the guarantee by induction: let's imagine we have two intermediate results a and b who were computed in n_a and n_b operations

[4] This assumes the magnitudes do not go outside the allowable range ($\approx 10^{\pm 308}$ for `double` and $\approx 10^{\pm 4932}$ for `long double`) which almost never happens for geometry problems.

[5] Note that when we say a value is "computed in n operations" we mean that it is computed by a single formula that contains n operations, and not that n operations are necessary to actually compute it. For example $(a+b)+(a+b)$ is considered to be "computed in 3 operations" even though we can implement this with only 2 additions.

respectively. By the inductive hypothesis their imprecise computed values a' and b' respect the following conditions.

$$|a' - a| \leq n_a M\epsilon \qquad |b' - b| \leq n_b M\epsilon$$

The result of the floating-point addition of a' and b' is $\text{round}(a'+b')$ where round() is the function that rounds a real value to the closest representable floating-point value. We know that $|\,\text{round}(x) - x| \leq M\epsilon$, so we can find a bound on the error of the addition:

$$\begin{aligned}
&|\,\text{round}(a' + b') - (a + b)| \\
&\qquad = \big|\big[\text{round}(a' + b') - (a' + b')\big] + \big[(a' + b') - (a + b)\big]\big| \\
&\qquad \leq |\,\text{round}(a' + b') - (a' + b')| + |(a' + b') - (a + b)| \\
&\qquad \leq M\epsilon + |(a' - a) + (b' - b)| \\
&\qquad \leq M\epsilon + |a' - a| + |b' - b| \\
&\qquad \leq M\epsilon + n_a M\epsilon + n_a M\epsilon \\
&\qquad = (n_a + n_b + 1)M\epsilon
\end{aligned}$$

where the first two steps follow from the triangle inequality. Since the sum is "computed in $n_a + n_b + 1$ operations", the bound of $(n_a + n_b + 1)M\epsilon$ that is obtained is small enough. The proof for subtraction is very similar.

1.3.4 Incorporating multiplication

The model above gives good guarantees but is very limited: it only works for computations that use only addition and subtraction. Multiplication does not give guarantees of the form $nM\epsilon$. However, we can still say interesting things if we take a closer look the different types of values we use in geometry:

- Adimensional "0D" values: e.g. angles, constant factors;
- 1D values: e.g. coordinates, lengths, distances, radii;
- 2D values: e.g. areas, dot products, cross products;
- 3D values: e.g. volumes, mixed products.

Usually, the problem statement gives guarantees on the magnitude of coordinates, so we can find some constant M so that all 1D values that will be computed in the code have a magnitude less than M. And since 2D and 3D values are usually created by products of 1D values, we can usually say that 2D values are bounded in magnitude by M^2 and 3D values by M^3 (we may need to multiply M by a constant factor).

It turns out that computations made of $+, -, \times$ and in which all d-dimensional values are bounded in magnitude by M^d have good precision guarantees. In fact, we can prove that the absolute error of a d-dimensional

number computed in n operations is at most $M^d\left((1+\epsilon)^n - 1\right)$, which assuming $n\epsilon \ll 1$ is about $nM^d\epsilon$.

The proof is similar in spirit to what we did with only + and − earlier. Since it is a bit long, we will not detail it here, but it can be found in section A.1, along with a more precise definition of the precision guarantees and its underlying assumptions.

Note that this does *not* cover multiplication by an adimensional factor bigger than 1: this makes sense, since for example successive multiplication by 2 of a small value could make the absolute error grow out of control even if the magnitude remains under M^d for a while.

In other cases, this formula $nM^d\epsilon$ gives us a quick and reliable way to estimate precision errors.

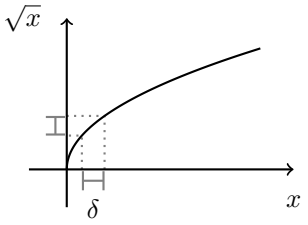
1.3.5 Why other operations do not work as well

Now that we have precision guarantees for $+, -, \times$ operations, one might be tempted to try and include division as well. However, if that was possible, then it would be possible to give strong precision guarantees for line intersection, and we saw in subsection 1.1.1 that this is not the case.

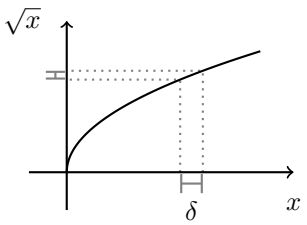
The core of the problem is: if some value x is very close to zero, then a small absolute error on x will create a large absolute error on $1/x$. In fact, if x is smaller than its absolute error, the computed value $1/x$ might be arbitrarily big, both in the positive or negative direction, and might not exist. This is why it is hard to give guarantees on the results of a division whose operands are already imprecise.

An operation that also has some problematic behavior is $\sqrt{x}$. If x is smaller than its absolute error, then $\sqrt{x}$ might or might not be defined in the reals. However, if we ignore the issue of existence by assuming that the theoretical and actual value of x are both nonnegative, then we *can* say some things on the precision.

Because $\sqrt{x}$ is a concave increasing function, a small imprecision on x will have the most impact on $\sqrt{x}$ near 0.



close to 0 ⇒ big effect



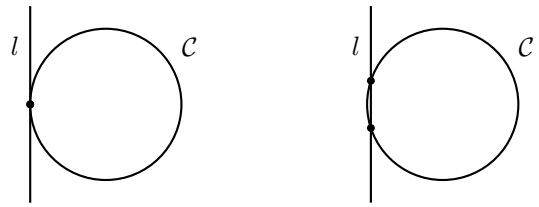
far from 0 ⇒ small effect

Therefore for a given imprecision δ, the biggest imprecision on $\sqrt{x}$ it might cause is $\sqrt{\delta}$. This is usually pretty bad: if the argument of the square root had an imprecision of $nM^2\epsilon$ then in the worst case the result will have an imprecision of $\sqrt{n}M\sqrt{\epsilon}$, instead of the $nM\epsilon$ bound that we have for $+, -, \times$ operations.

For example let us consider a circle $\mathcal{C}$ of radius tangent to a line l. If $\mathcal{C}$ gets closer to l by 10^{-6}, then the intersection points will move by about

$$\sqrt{1^2 - (1 - 10^{-6})^2} \approx \sqrt{2 \times 10^{-6}} = \sqrt{2} \times 10^{-3}$$

away from the tangency point, as pictured below.



Note that here we have only shown that $1/x$ and $\sqrt{x}$ perform poorly on imprecise inputs. Please bear in mind that on exact inputs, the IEEE 754 guarantees that the result is the closest represented floating-point number. So when the lines and circles are defined by integers, line intersections and circle-line intersections have a relative precision error proportional to ϵ and thus an absolute error proportional to $M\epsilon$.

1.4 Case studies

In this section, we explore some practical cases in which the imprecisions of floating-point numbers can cause problems and give some possible solutions.

1.4.1 Problem "Keeping the Dogs Apart"

We will first talk about problem "Keeping the Dogs Apart", which we mentioned before, because it is a good example of accumulation of error and how to deal with it. It was written by Markus Fanebust Dregi for NCPC 2016. You can read the full statement and submit it at `https://open.kattis.com/problems/dogs`.

Here is a summarized problem statement: There are two dogs A and B, walking at the same speed along different polylines $A_0 \dots A_{n-1}$ and $B_0 \dots B_{m-1}$, made of 2D integer points with coordinates in $[0, 10^4]$. They

start at the same time from A_0 and B_0 respectively. What is the closest distance they will ever be from each other before one of them reaches the end of its polyline? The relative/absolute error tolerance is 10^{-4}, and $n, m \leq 10^5$.

The idea of the solution is to divide the time into intervals where both dogs stay on a single segment of their polyline. Then the problem reduces to the simpler task of finding the closest distance that get when one walks on $[PQ]$ and the other on $[RS]$, with $|PQ| = |RS|$. This division into time intervals can be done with the two-pointers technique: if we remember for each dog how many segments it has completely walked and their combined length, we can work out when is the next time on of the dogs will switch segments.

The main part of the code looks like this. We assume that `moveBy(a,b,t)` gives the point on a segment $[AB]$ at a certain distance t from A, while `minDist(p,q,r,s)` gives the minimum distance described above for P, Q, R, S.

```
int i = 0, j = 0; // current segment of A and B
double ans = abs(a[0]-b[0]), // closest distance so far
      sumA = 0, // total length of segments fully walked by A
      sumB = 0; // total length of segments fully walked by B

// While both dogs are still walking
while (i+1 < n && j+1 < m) {
    double start = max(sumA, sumB), // start of time interval
              dA = abs(a[i+1]-a[i]), // length of current segment of A
              dB = abs(b[j+1]-b[j]), // length of current segment of B
            endA = sumA + dA, // time at which A will end this segment
            endB = sumB + dB, // time at which B will end this segment
             end = min(endA, endB); // end of time interval

    // Compute start and end positions of both dogs
    pt p = moveBy(a[i], a[i+1], start-sumA),
       q = moveBy(a[i], a[i+1], end-sumA),
       r = moveBy(b[j], b[j+1], start-sumB),
       s = moveBy(b[j], b[j+1], end-sumB);

    // Compute closest distance for this time interval
    ans = min(ans, minDist(p,q,r,s));

    // We get to the end of the segment for one dog or the other,
    // so move to the next and update the sum of lengths
    if (endA < endB) {
        i++;
        sumA += dA;
    } else {
        j++;
        sumB += dB;
    }
}
// output ans
```

As we said in section 1.1.2, the sums `sumA` and `sumB` accumulate very large errors. Indeed, they can both theoretically reach $M = \sqrt{2} \times 10^9$, and are based on up to $k = 10^5$ additions. With `double`, $\epsilon = 2^{-53}$, so we could reach up to $kM\epsilon \approx 0.016$ in absolute error in both `sumA` and `sumB`. Since this error directly translates into errors in P, Q, R, S and is bigger than the tolerance of 10^{-4}, this causes WA.

In the rest of this section, we will look at two ways we can avoid this large accumulation of error in `sumA` and `sumB`. Since this is currently much bigger than what could have been caused by the initial length computations, `moveBy()` and `minDist()`, we will consider those errors to be negligible for the rest of the discussion.

Limiting the magnitude involved

The first way we can limit the accumulation of error in `sumA` and `sumB` is to realize that in fact, we only care about the difference between them: if we add a certain constant to both variables, this doesn't change the value of `start-sumA`, `end-sumA`, `start-sumB` or `end-sumB`, so the value of `p`, `q`, `r`, `s` is unchanged.

So we can adapt the code by adding these lines at the end of the `while` loop:

```
double minSum = min(sumA, sumB);
sumA -= minSum;
sumB -= minSum;
```

After this, one of `sumA` and `sumB` becomes zero, while the other carries the error on both. In total, at most $n + m$ additions and $n + m$ subtractions are performed on them, for a total of $k \leq 4 \times 10^5$. But since the difference between `sumA` and `sumB` never exceeds the length of one segment, that is, $M = \sqrt{2} \times 10^4$, the error is much lower than before:

$$kM\epsilon = \left(4 \times 10^5\right) \times \left(\sqrt{2} \times 10^4\right) \times 2^{-53} \approx 6.3 \times 10^{-7}$$

so it gives an AC verdict.

So here we managed to reduce the precision mistakes on our results by reducing the magnitude of the numbers that we manipulate. Of course, this is only possible if the problem allows it.

Summing positive numbers more precisely

Now we present different way to reduce the precision mistake, based on the fact that all the terms in the sum we're considering are positive. This is a good thing, because it avoids catastrophic cancellation (see section 1.3.1).

In fact, addition of nonnegative numbers conserves relative precision: if you sum two nonnegative numbers a and b with relative errors of $k_a\epsilon$ and $k_b\epsilon$ respectively, the worst-case relative error on $a+b$ is about[6] $(\max(k_a, k_b)+1)\epsilon$.

Let's say we need to compute the sum of n nonnegative numbers $a_1, \ldots, a_n$. We suppose they are exact. If we perform the addition in the conventional order, like this:

$$\Big(\cdots\Big(\big((a_1 + a_2) + a_3\big) + a_4\Big) + \cdots\Big) + a_n$$

then

- $a_1 + a_2$ will have a relative error of $(\max(0, 0) + 1)\epsilon = \epsilon$;
- $(a_1 + a_2) + a_3$ will have a relative error of $(\max(1, 0) + 1)\epsilon = 2\epsilon$;
- $\big((a_1 + a_2) + a_3\big) + a_4$ will have a relative error of $(\max(2, 0) + 1)\epsilon = 3\epsilon$;
- ... and so on.

So the complete sum will have an error of $(n - 1)\epsilon$, not better than what we had before.

But what if we computed the additions in another order? For example, with $n = 8$, we could do this:

$$\big((a_1 + a_2) + (a_3 + a_4)\big) + \big((a_5 + a_6) + (a_7 + a_8)\big)$$

then all additions of two numbers have error ϵ, all additions of 4 numbers have error $(\max(1, 1) + 1)\epsilon = 2\epsilon$, and the complete addition has error $(\max(2, 2) + 1)\epsilon = 3\epsilon$, which is much better than $(n - 1)\epsilon = 7\epsilon$. In general, for $n = 2^k$, we can reach a relative precision of $k\epsilon$.

We can use this grouping technique to create an accumulator such that the relative error after adding n numbers is at most $2\log_2(k)\epsilon$.[7] Here is an $O(n)$ implementation:

```
struct stableSum {
    int cnt = 0;
    vector<double> v, pref{0};
    void operator+=(double a) {
        assert(a >= 0);
        int s = ++cnt;
        while (s % 2 == 0) {
            a += v.back();
            v.pop_back(), pref.pop_back();
            s /= 2;
        }
        v.push_back(a);
```

[6] It could in fact go up to $(\max(k_a, k_b)(1 + \epsilon) + 1)\epsilon$ but the difference is negligible for our purposes.

[7] We could even get $(\log_2 k+1)\epsilon$ but we don't know a way to do it faster than $O(n \log n)$.

```
        pref.push_back(pref.back() + a);
    }
    double val() {return pref.back();}
};
```

Let's break this code down. This structure provides two methods: `add(a)` to add a number a to the sum, and `val()` to get the current value of the sum. Array `v` contains the segment sums that currently form the complete sum, similar to Fenwick trees: for example, if we have added 11 elements, `v` would contain three elements:

$$v = \{a_1 + \cdots + a_8,\ a_9 + a_{10},\ a_{11}\}$$

while `pref` contains the prefix sums of `v`: `pref[i]` contains the sum of the i first elements of `v`.

Function `add()` performs the grouping: when adding a new element `a`, it will merge it with the last element of `v` while they contain the same number of terms, then `a` is added to the end of `v`. For example, if we add the 12^{th} element a_{12}, the following steps will happen:

$$\begin{aligned}
v &= \{a_1 + \cdots + a_8,\ a_9 + a_{10},\ a_{11}\} & a &= a_{12} \\
v &= \{a_1 + \cdots + a_8,\ a_9 + a_{10}\} & a &= (a_{11}) + a_{12} \\
v &= \{a_1 + \cdots + a_8\} & a &= (a_9 + a_{10}) + a_{11} + a_{12} \\
v &= \{a_1 + \cdots + a_8,\ a_9 + a_{10} + a_{11} + a_{12}\}
\end{aligned}$$

The number of additions we have to make for the i^{th} number is the number of times it is divisible by 2. Since we only add one element to `v` when adding an element to the sum, this is amortized constant time.

By simply changing the types of `sumA` and `sumB` to `stableSum` and adding `.val()` whenever the value is read in the initial code, we can get down to an error of about

$$2\log_2\left(10^5\right)M\epsilon = \left(2\log_2\left(10^5\right)\right) \times \left(\sqrt{2} \times 10^9\right) \times 2^{-53} \approx 5.2 \times 10^{-6}$$

which also gives an AC verdict.[8]

This is not as good as the precision obtained with the previous method, but that method was specific to the problem, while this one can be applied whenever we need to compute sums of nonnegative numbers.

[8]Theoretically we can't really be sure though, since both `sumA` and `sumB` could have that error, and we still have to take into account the other operations performed.

1.4.2 Quadratic equation

As another example, we will study the precision problems that can occur when computing the roots of an equation of the type $ax^2 + bx + c = 0$ with $a \neq 0$. We will see how some precision problems are unavoidable, while others can be circumvented by

When working with full-precision reals, we can solve quadratic equations in the following way. First we compute the discriminant $\Delta = b^2 - 4ac$. If $\Delta < 0$, there is no solution, while if $\Delta \geq 0$ there is are 1 or 2 solutions, given by

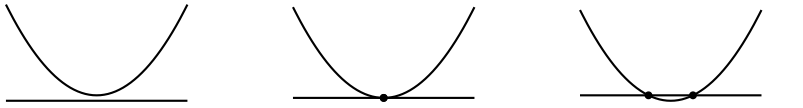
$$x = \frac{-b \pm \sqrt{\Delta}}{2a}$$

The first difficulty when working with floating-point numbers is the computation of Δ: if $\Delta \approx 0$, that is $b^2 \approx 4ac$, then the imprecisions can change the sign of Δ, therefore changing the number of solutions.

Even if that does not happen, since we have to perform a square root, the problems that we illustrated with line-circle intersection in section 1.3.5 can also happen here.[9] Take the example of equation $x^2 - 2x + 1 = 0$, which is a single root $x = 0$. If there is a small error on c, it can translate into a large error on the roots. For example, if $c = 1 - 10^{-6}$, then the roots become

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} = \frac{2 \pm \sqrt{4 - 4(1 - 10^{-6})}}{2} = \frac{2 \pm 2\sqrt{10^{-6}}}{2} = 1 \pm 10^{-3}.$$

where the error 10^{-3} is much bigger than the initial error on c.



$x^2 - 2x + (1 + \epsilon) = 0$ no solution

$x^2 - 2x + 1 = 0$ $x = 0$

$x^2 - 2x + (1 - \epsilon) = 0$ $x = 1 \pm \sqrt{\epsilon}$

Even if the computation of $\sqrt{\Delta}$ is very precise, a second problem can occur. If b and $\sqrt{\Delta}$ have a similar magnitude, in other words when $b^2 \gg ac$, then catastrophic cancellation will occur for one of the roots. For example if $a = 1, b = 10^4, c = 1$, then the roots will be:

$$x_1 = \frac{-10^4 - \sqrt{10^8 - 4}}{2} \approx -10^4 \qquad x_2 = \frac{-10^4 + \sqrt{10^8 - 4}}{2} \approx 10^{-4}$$

The computation of x_1 goes fine because $-b$ and $-\sqrt{\Delta}$ have the same sign. But because the magnitude of $-b + \sqrt{\Delta}$ is 10^8 times smaller than the

[9] Which is not surprising, since the bottom of a parabola looks a lot like a circle.

magnitude of b and $\sqrt{\Delta}$, the relative error on x_2 will be 10^8 times bigger than the relative error on b and $\sqrt{\Delta}$.

Fortunately, in this case we can avoid catastrophic cancellation entirely by rearranging the expression:

$$\begin{aligned}
\frac{-b \pm \sqrt{b^2-4ac}}{2a} &= \frac{-b \pm \sqrt{b^2-4ac}}{2a} \times \frac{-b \mp \sqrt{b^2-4ac}}{-b \mp \sqrt{b^2-4ac}} \\
&= \frac{(-b^2) - \left(\sqrt{b^2-4ac}\right)^2}{2a\left(-b \mp \sqrt{b^2-4ac}\right)} \\
&= \frac{4ac}{2a\left(-b \mp \sqrt{b^2-4ac}\right)} \\
&= \frac{2c}{-b \mp \sqrt{b^2-4ac}}
\end{aligned}$$

In this new expression, since the sign of the operation is opposite from the sign in the original expression, catastrophic cancellation happens in only one of the two.

So if $b \geq 0$, we can use $\frac{-b-\sqrt{\Delta}}{2a}$ for the first solution and $\frac{2c}{-b-\sqrt{\Delta}}$ for the second solution, while if $b \leq 0$, we can use $\frac{2c}{-b+\sqrt{\Delta}}$ for the first solution and $\frac{-b+\sqrt{\Delta}}{2a}$ for the second solution. We only need to be careful that the denominator is never zero.

This gives a safer way to find the roots of a quadratic equation. This function returns the number of solutions, and places them in `out` in no particular order.

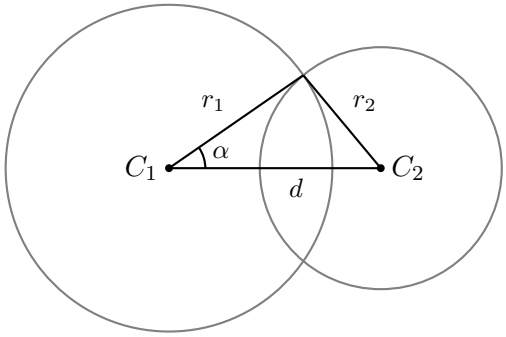
```
int quadRoots(double a, double b, double c, pair<double,double> &out) {
    assert(a != 0);
    double disc = b*b - 4*a*c;
    if (disc < 0) return 0;
    double sum = (b >= 0) ? -b-sqrt(disc) : -b+sqrt(disc);
    out = {sum/(2*a), sum == 0 ? 0 : (2*c)/sum};
    return 1 + (disc > 0);
}
```

In many cases, there are several ways to write an expression, and they can have very different behaviors when used with floating-point numbers. So if you realize that the expression you are using can cause precision problems in some cases, it can be a good idea to rearrange the expression to handle them, as we did here.

1.4.3 Circle-circle intersection

This last case study will study one possible implementation for the intersection of two circles. It will show us why we shouldn't rely too much on mathematical truths when building our programs.

We want to know whether two circles of centers C_1, C_2 and radii r_1, r_2 touch, and if they do what are the intersection points. Here, we will solve this problem with triangle inequalities and the cosine rule.[10] Let $d = |C_1C_2|$. The question of whether the circles touch is equivalent to the question of whether there exists a (possibly degenerate) triangle with edge lengths d, r_1, r_2.



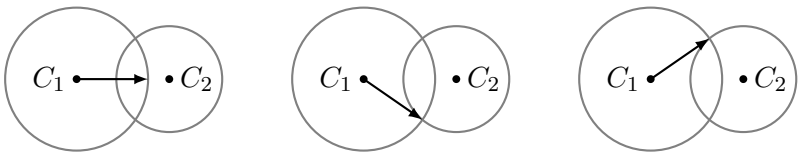
We know that such a triangle exists iff the triangle inequalities are respected, that is:

$$|r_2 - r_1| \leq d \leq r_1 + r_2$$

If this is true, then we can find the angle at C_1, which we'll call α, thanks to the cosine rule:

$$\cos\alpha = \frac{d^2 + r_1^2 - r_2^2}{2dr_1}$$

Once we have α, we can find the intersection points in the following way: if we take vector $\overrightarrow{C_1C_2}$, resize it to have length r_1, then rotate by α in either direction, this gives the vectors from C_1 to either intersection points.



resized rotated right rotated left

[10] The way we actually implement it in this book is completely different.

This gives the following code. It uses a function `abs()` to compute the length of a vector (see section 2.1.2) and a function `rot()` to rotate a vector by a given angle (see section 2.2.3).

```
bool circleCircle(pt c1, double r1, pt c2, double r2, pair<pt,pt> &out) {
    double d = abs(c2-c1);
    if (d < abs(r2-r1) || d > r1+r2) // triangle inequalities
        return false;
    double alpha = acos((d*d + r1*r1 - r2*r2)/(2*d*r1));
    pt rad = (c2-c1)/d*r1; // vector C1C2 resized to have length d
    out = {c1 + rot(rad, -alpha), c1 + rot(rad, alpha)};
    return true;
}
```

This implementation is quite nice, but unfortunately it will sometimes output `nan` values. In particular, if

$$r_1 = 0.625 \qquad r_2 = 0.3750000000000004 \qquad d = 1.0000000000000004$$

then the triangle inequalities are respected, so the function returns **`true`**, but the program computes

$$\frac{d^2 + r_1^2 - r_2^2}{2dr_1} > 1$$

In fact, this is mathematically impossible! The cosine rule should give values in $[-1, 1]$ as long as the edge lengths respect the triangle inequality. To make sure, we can compute:

$$\begin{aligned} \frac{d^2 + r_1^2 - r_2^2}{2dr_1} > 1 \;&\Rightarrow\; d^2 + r_1^2 - r_2^2 > 2dr_1 \\ &\Leftrightarrow\; (d - r_1)^2 > r_2^2 \\ &\Leftrightarrow\; |d - r_1| > r_2 \\ &\Leftrightarrow\; d > r_2 + r_1 \quad \text{or} \quad r_1 > d + r_2 \end{aligned}$$

Indeed, both are impossible because of the triangle inequalities. So this must be the result of a few unfortunate roundings made while computing the expression.

There are two possible solutions to this. The first solution would be to just treat the symptoms: make sure the cosine is never outside $[-1, 1]$ by either returning **`false`** or by moving it inside:

```
double co = (d*d + r1*r1 - r2*r2)/(2*d*r1);
if (abs(co) > 1) {
    return false; // option 1
    co /= abs(co); // option 2
}
double alpha = cos(co);
```

The second solution, which we recommend, is based on the principles that we should always try to minimize the number of comparisons we make, and that if we have to do some computation that might fail (giving a result of `nan` or infinity), then we should test the input of that computation *directly*.

So instead of testing the triangle inequalities, we test the value of $\cos\alpha$ directly, because it turns out that it will be in $[-1, 1]$ iff the triangle inequalities are verified. This gives the following code, which is a bit simpler and safer.

```
bool circleCircle(pt c1, double r1, pt c2, double r2, pair<pt,pt> &out) {
    double d = abs(c2-c1), co = (d*d + r1*r1 - r2*r2)/(2*d*r1);
    if (abs(co) > 1) return false;
    double alpha = acos(co);
    pt rad = (c2-c1)/d*r1; // vector C1C2 resized to have length d
    out = {c1 + rot(rad, -alpha), c1 + rot(rad, alpha)};
    return true;
}
```

1.5 Some advice

In this last section, we present some general advice about precision issues when solving or setting a problem.

1.5.1 For problem solvers

One of the keys to success in geometry problems is to develop a reliable implementation methodology as you practise. Here are some basics to get you started.

As you have seen in this chapter, using floating-point numbers can cause many problems and betray you in countless ways. Therefore the first and most important piece of advice is to avoid using them altogether. Surprisingly many geometric computations can be done with integers, and you should always aim to perform important comparisons with integers, by first figuring out the formula on paper and then implementing it without division or square root.

When you are forced to use floating-point numbers, you should minimize the risks you take. Indeed, thinking about everything that could go wrong in an algorithm is very hard and tedious, so if you take many inconsiderate risks, the time you will need to spend too much time on verification (or not spend it and suffer the consequences). In particular:

- Minimize the number of dangerous operations you make, such as divisions, square roots, and trigonometric functions. Some of these functions can amplify precision mistakes, and many are defined on re-

stricted domains. Make sure you do not go out of the domains by considering every single one of them carefully.

- Separate cases sparingly. Many geometry problems require some casework, making comparisons to separate them can be unsafe, and every case adds more code and more reasons for failures. When possible, try to write code that handles many situations at once.
- Do not rely too much on mathematical truths. Things that are true for reals are not necessarily true for floating-point numbers. For example, $r^2 - d^2$ and $(r + d)(r - d)$ are not always exactly the same value. Be extra careful when those values are then used in an operation that is not defined everywhere (like $\sqrt{x}$, $\arccos(x)$, $\tan(x)$, $\frac{x}{y}$ etc.).

In general, try to build programs that are resistant to the oddities of floating-point numbers. Imagine that some evil demon is slightly modifying every result you compute in the way that is most likely to make your program fail. And try to write clean code that is *clearly correct* at first glance. If you need long explanations to justify why your program will not fail, then it is more likely that your program will in fact fail.

1.5.2 For problem setters

Finally, here is some general advice about precision issues when creating a geometry problem and its datasets.

- Never use floating-point numbers as inputs, as this will already cause imprecisions when first reading the input numbers, and completely exclude the use of integers make it impossible to determine some things with certainty, like whether two segmentns touch, whether some points are collinear, etc.
- Make the magnitude of the input coordinates as small as possible to avoid causing overflows or big imprecisions in the contestant's codes.
- Favor problems where the important comparisons can be made entirely with integers.
- Avoid situations in which imprecise points are used for numerically unstable operations such as finding the intersection of two lines.
- In most cases, you should specify the tolerance in terms of absolute error only (see subsection 1.3.1).
- Make sure to prove that all correct algorithm are able to reach the precision that you require, and be careful about operations like circle-line intersection which can greatly amplify imprecisions. Since error analysis is more complicated than it seems at first sight and requires a bit of expertise, you may want to ask a friend for a second opinion.

Chapter 2

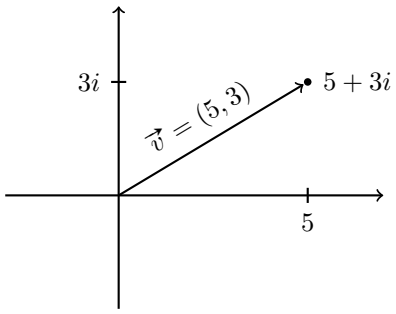
Basics

2.1 Points and vectors

In this section, we will first introduce complex numbers, because they are a useful way to think about and represent 2D points, especially when rotations are involved. We will then present two different ways to represent points in code: one by creating our own structure, the other by using the C++ built-in `complex` type. Either can be used to run the code samples in this book, though `complex` requires less typing.

2.1.1 Complex numbers

Complex numbers are an extension of the real numbers with a new unit, the *imaginary unit*, noted i. A complex number is usually written as $a + bi$ (for $a, b \in \mathbb{R}$) and we can interpret it geometrically as point (a, b) in the two-dimensional plane, or as a vector with components $\vec{v} = (a, b)$. We will sometimes use all these notations interchangeably. The set of complex numbers is written as $\mathbb{C}$.

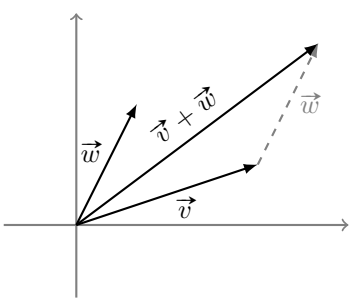


Basic operations

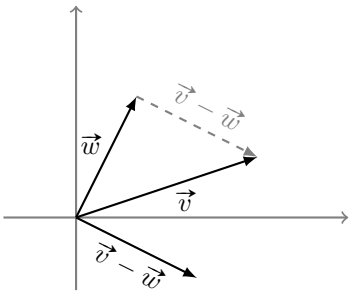
Complex numbers are added, subtracted and multiplied by scalars as if i were an unknown variable. Those operations are equivalent to the same operations on vectors.

$$(a+bi)+(c+di)=(a+c)+(b+d)i \qquad \text{(addition)}$$
$$(a+bi)-(c+di)=(a-c)+(b-d)i \qquad \text{(subtraction)}$$
$$k(a+bi)=(ka)+(kb)i \qquad \text{(multiplication by scalar)}$$

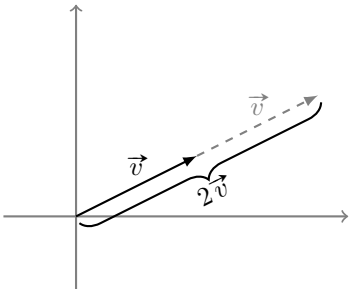
Geometrically, adding or subtracting two complex numbers $\vec{v}=(a,b)$ and $\vec{w}=(c,d)$ corresponds to making $\vec{w}$ or its opposite start at the end of $\vec{v}$, while multiplying $\vec{v}$ by a positive real k corresponds to multiplying its length by k but keeping the same direction.



addition



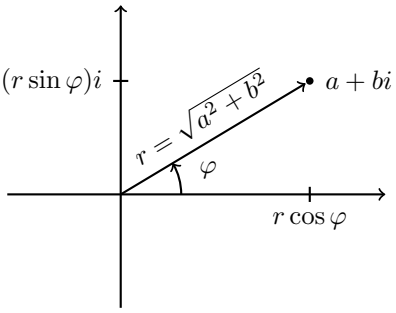
subtraction



multiplication by scalar

Polar form

The polar form is another way to represent complex numbers. To denote a complex $\vec{v}=a+bi$, instead of looking at the real and complex parts, we look at the *absolute value* r, the distance from the origin (the length of vector $\vec{v}$), and the *argument* φ, the amplitude of the angle that $\vec{v}$ forms with the positive real axis.



For a given complex number $a + bi$, we can compute its polar form as

$$r = |a + bi| = \sqrt{a^2 + b^2}$$
$$\varphi = \arg(a + bi) = \operatorname{atan2}(b, a)$$

and conversely, a complex number with polar coordinates r, φ can be written

$$r \cos \varphi + (r \sin \varphi)i = r(\cos \varphi + i \sin \varphi) =: r \operatorname{cis} \varphi$$

where $\operatorname{cis} \varphi = \cos \varphi + i \sin \varphi$ is the unit vector that forms an angle of amplitude φ with the positive real axis.

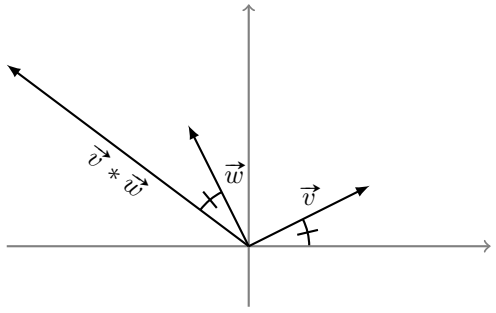
Note that this is not a one-to-one mapping. Firstly, adding or subtracting 2π from φ doesn't change the point being represented; to solve this problem, φ is generally taken in $(-\pi, \pi]$. Secondly, when $r = 0$, all values of φ represent the same point.

Multiplication

Complex multiplication is easiest to understand using the polar form. When multiplying two complex numbers, their absolute values are multiplies, while their arguments are added. In other words,

$$(r_1 \operatorname{cis} \varphi_1) * (r_2 \operatorname{cis} \varphi_2) = (r_1 r_2) \operatorname{cis}(\varphi_1 + \varphi_2).$$

In the illustration below, $|\vec{v} * \vec{w}| = |\vec{v}||\vec{w}|$ and the angle between the x-axis and $\vec{v}$ is the same as the angle between $\vec{w}$ and $\vec{v} * \vec{w}$.



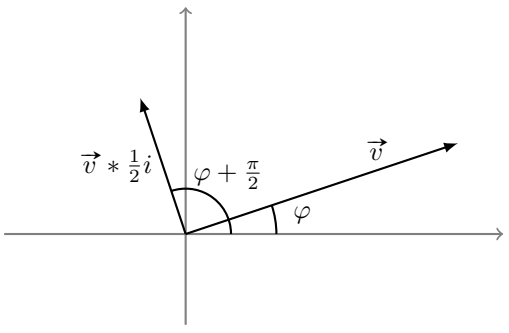
Remarkably, multiplication is also very simple to compute from the coordinates: it works a bit like polynomial multiplication, except that we transform i^2 into -1.

$$\begin{aligned}(a+bi)*(c+di) &= ac + a(di) + (bi)c + (bi)(di)\\ &= ac + adi + bci + (bd)i^2\\ &= ac + (ad+bc)i + (bd)(-1)\\ &= (ac-bd) + (ad+bc)i\end{aligned}$$

Exercise

Prove that $(r_1 \operatorname{cis}\varphi_1) * (r_2 \operatorname{cis}\varphi_2) = (r_1 r_2)\operatorname{cis}(\varphi_1+\varphi_2)$ using this new definition of product.
Solution: (select to reveal)

Another way to explain complex multiplication is to say that multiplying a number by $r \operatorname{cis}\varphi$ will scale it by r and rotate it by φ counterclockwise. For example, multiplying a number by $\frac{1}{2}i = \frac{1}{2}\operatorname{cis}\frac{\pi}{2}$ will divide its length by 2 and rotate it 90° counterclockwise.



2.1.2 Point representation

In this section we explain how to implement the point structure that we will use throughout the rest of the book. The code is only available in C++

at the moment, but should be easy to translate in most languages. Our long-term plan is to offer code snippets in other languages as well.

With a custom structure

Let us first declare the basic operations: addition, subtraction, and multiplication/division by a scalar.

```
typedef double T;
struct pt {
    T x,y;
    pt operator+(pt p) {return {x+p.x, y+p.y};}
    pt operator-(pt p) {return {x-p.x, y-p.y};}
    pt operator*(T d) {return {x*d, y*d};}
    pt operator/(T d) {return {x/d, y/d};} // only for floating-point
};
```

For generality, we declare type `T`: the type of the the coordinates. Generally, either `double` or `long long` (for exact computations with integers) is appropriate. `long double` can also be very useful if extra precision is required. The cases where integers cannot be used are often quite clear (e.g. division by scalar, rotation by arbitrary angle).

We define some comparators for convenience:

```
bool operator==(pt a, pt b) {return a.x == b.x && a.y == b.y;}
bool operator!=(pt a, pt b) {return !(a == b);}
```

Note that there is no obvious way to define a $<$ operator on 2D points, so we will only define it as needed.

Here are some functions linked to the absolute value:

```
T sq(pt p) {return p.x*p.x + p.y*p.y;}
double abs(pt p) {return sqrt(sq(p));}
```

The squared absolute value `sq()` can be used to compute and compare distances quickly and exactly if the coordinates are integers. We use `double` for `abs()` because it will return floating-point values even for integer coordinates (if you are using `long double` you should probably change it to `long double`).

We also declare a way to print out points, for debugging purposes:

```
ostream& operator<<(ostream& os, pt p) {
    return os << "("<< p.x << "," << p.y << ")";
}
```

Some example usage:

```
pt a{3,4}, b{2,-1};
cout << a+b << " " << a-b << "\n"; // (5,3) (1,5)
cout << a*-1 << " " << b/2 << "\n"; // (-3,-4) (1.5,2)
```

With the C++ `complex` structure

Using the `complex` type in C++ can be a very practical choice in contests such as ACM-ICPC where everything must be typed from scratch, as many of the operations we need are already implemented and ready to use.

The code below defines a `pt` with similar functionality.

```
typedef double T;
typedef complex<T> pt;
#define x real()
#define y imag()
```

Warning

As with the custom structure, you should choose the appropriate coordinate type for `T`. However, be warned that if you define it as an integral type like `long long`, some functions which should always return floating-point numbers (like `abs()` and `arg()`) will be truncated to integers.

The macros `x` and `y` are shortcuts for accessing the real and imaginary parts of a number, which are used as x- and y-coordinates:

```
pt p{3,-4};
cout << p.x << " " << p.y << "\n"; // 3 -4
// Can be printed out of the box
cout << p << "\n"; // (3,-4)
```

Note that the coordinates can't be modified individually:

```
pt p{-3,2};
p.x = 1; // doesn't compile
p = {1,2}; // correct
```

We can perform all the operations that we have with the custom structure and then some more. Of course, we can also use complex multiplication and division. Note however that we can only multiply/divide by scalars of type `T` (so if `T` is `double`, then `int` will not work).

```
pt a{3,1}, b{1,-2};
a += 2.0*b; // a = (5,-3)
cout << a*b << " " << a/-b << "\n"; // (-1,-13) (-2.2,-1.4)
```

There are also useful methods for dealing with polar coordinates:

```
pt p{4,3};
// Get the absolute value and argument of point (in [-pi,pi])
cout << abs(p) << " " << arg(p) << "\n"; // 5 0.643501
// Make a point from polar coordinates
cout << polar(2.0, -M_PI/2) << "\n"; // (1.41421,-1.41421)
```

Warning

The `complex` library provides function `norm`, which is mostly equivalent to the `sq` that we defined earlier. However, it is not guaranteed to be exact for `double`: for example, the following expression evaluates to `false`.

```
norm(complex<double>(2.0,1.0)) == 5.0
```

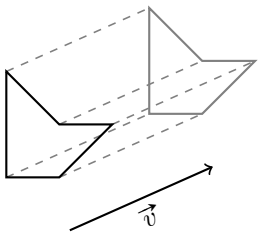
Therefore, to be safe you should implement a separate `sq()` function as for the custom structure (or you can wait use function `dot()` that we will define later).

2.2 Transformations

In this section we will show how to implement three transformations of the plane, in increasing difficulty. We will see that they all correspond to linear transformations on complex numbers, that is, functions of the form $f(p) = a * p + b$ for $a, b, p \in \mathbb{C}$, and deduce a way to compute a general transformation that combines all three.

2.2.1 Translation

To translate an object by a vector $\vec{v}$, we simply need to add $\vec{v}$ to every point in the object. The corresponding function is $f(p) = p + \vec{v}$ with $\vec{v} \in \mathbb{C}$.

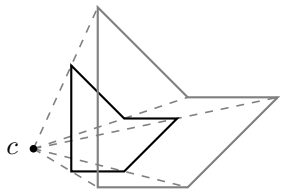


The implementation is self-explanatory:

```
pt translate(pt v, pt p) {return p+v;}
```

2.2.2 Scaling

To scale an object by a certain ratio α around a center c, we need to shorten or lengthen the vector from c to every point by a factor α, while conserving the direction. The corresponding function is $f(p) = c + \alpha(p - c)$ (α is a real here, so this is a scalar multiplication).

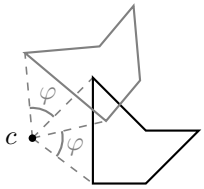


Again, the implementation is just a translation of the expression into code:

```
pt scale(pt c, double factor, pt p) {
    return c + (p-c)*factor;
}
```

2.2.3 Rotation

To rotate an object by a certain angle φ around center c, we need to rotate the vector from c to every point by φ. From our study of polar coordinates in 2.1.1 we know this is equivalent to multiplying by $\operatorname{cis}\varphi$, so the corresponding function is $f(p) = c + \operatorname{cis}\varphi * (p - c)$.



In particular, we will often use the (counter-clockwise) rotation centered on the origin. We use complex multiplication to figure out the formula:

$$\begin{aligned}(x + yi) * \operatorname{cis}\varphi &= (x + yi) * (\cos\varphi + i\sin\varphi) \\ &= (x\cos\varphi - y\sin\varphi) + (x\sin\varphi + y\cos\varphi)i\end{aligned}$$

which gives the following implementation:

```
pt rot(pt p, double a) {
    return {p.x*cos(a) - p.y*sin(a), p.x*sin(a) + p.y*cos(a)};
}
```

which if using `complex` can be simplified to just

```
pt rot(pt p, double a) {return p * polar(1.0, a);}
```

And among those, we will use the rotation by $90°$ quite often:

$$\begin{aligned}(x + yi) * \operatorname{cis}(90°) &= (x + yi) * (\cos(90°) + i\sin(90°)) \\ &= (x + yi) * i = -y + xi\end{aligned}$$

It works fine with integer coordinates, which is very useful:

```
pt perp(pt p) {return {-p.y, p.x};}
```

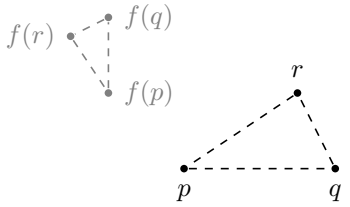
2.2.4 General linear transformation

It is easy to check that all those transformations are of the form $f(p) = a*p+b$ as claimed in the beginning of this section. In fact, all transformations of this type can be obtained as combinations of translations, scalings and rotations.[1]

Just like for real numbers, to determine a linear transformation such as this one, we only need to know the image of two points to know the complete function. Indeed, if we know $f(p) = a * p + b$ and $f(q) = a * q + b$, then we can find a as $\frac{f(q)-f(p)}{q-p}$, and then b as $f(p) - a * p$.

And thus if we want to know a new point $f(r)$ of that transformation, we can then compute it as:

$$f(r) = f(p) + (r - p) * \frac{f(q) - f(p)}{q - p}$$



This is easy to implement using `complex`:

```
pt linearTransfo(pt p, pt q, pt r, pt fp, pt fq) {
    return fp + (r-p) * (fq-fp) / (q-p);
}
```

Otherwise, you can use the cryptic but surprisingly short solution from [2] (see the next sections for `dot()` and `cross()`):

```
pt linearTransfo(pt p, pt q, pt r, pt fp, pt fq) {
    pt pq = q-p, num(cross(pq, fq-fp), dot(pq, fq-fp));
    return fp + pt(cross(r-p, num), dot(r-p, num)) / sq(pq);
}
```

[1] Actually, if $a = 1$ it is just a translation, and if $a \neq 1$ it is the combination of a scaling and a rotation combination from a well-chosen center.

2.3 Products and angles

Besides complex multiplication, which is nice to have but is not useful so often, there are two products involving vectors that are of critical importance: dot product and cross product. In this section, we'll look at their definition, properties and some basic use cases.

2.3.1 Dot product

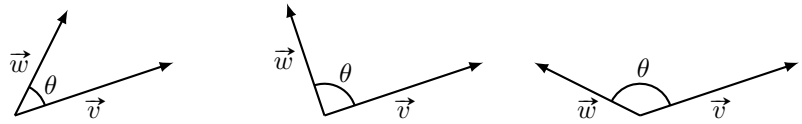
The dot product $\vec{v} \cdot \vec{w}$ of two vectors $\vec{v}$ and $\vec{w}$ can be seen as a measure of how similar their directions are. It is defined as

$$\vec{v} \cdot \vec{w} = \|\vec{v}\|\|\vec{w}\| \cos\theta$$

where $\|\vec{v}\|$ and $\|\vec{w}\|$ are the lengths of the vectors and θ is amplitude of the angle between $\vec{v}$ and $\vec{w}$.

Since $\cos(-\theta) = \cos(\theta)$, the sign of the angle does not matter, and the dot product is symmetric: $\vec{v} \cdot \vec{w} = \vec{w} \cdot \vec{v}$.

In general we will take θ in $[0, \pi]$, so that dot product is positive if $\theta < \pi/2$, negative if $\theta > \pi/2$, and zero if $\theta = \pi/2$, that is, if $\vec{v}$ and $\vec{w}$ are perpendicular.



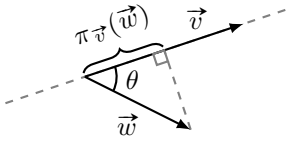
$\theta < \pi/2$ $\quad$ $\theta = \pi/2$ $\quad$ $\theta > \pi/2$

$\vec{v} \cdot \vec{w} = 5$ $\quad$ $\vec{v} \cdot \vec{w} = 0$ $\quad$ $\vec{v} \cdot \vec{w} = -5$

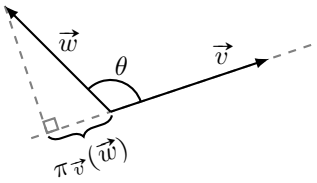
If we fix $\|\vec{v}\|$ and $\|\vec{w}\|$ as above, the dot product is maximal when the vectors point in the same direction, because $\cos\theta = \cos(0) = 1$, and minimal when they point in opposite directions, because $\cos\theta = \cos(\pi) = -1$.

Math insight

Because of the definition of cosine in right triangles, dot product can be interpreted in an interesting way (assuming $\vec{v} \neq 0$): $\vec{v} \cdot \vec{w} = \|\vec{v}\| \pi_{\vec{v}}(\vec{w})$ where $\pi_{\vec{v}}(\vec{w}) := \|\vec{w}\| \cos\theta$ is the signed length of the projection of $\vec{w}$ onto the line that contains $\vec{v}$ (see examples below). In particular, this means that the dot product does not change if one of the vectors moves perpendicular to the other.



$\pi_{\vec{v}}(\vec{w}) = 1.58$



$\pi_{\vec{v}}(\vec{w}) = -1.26$

Remarkably, the dot product can be computed by a very simple expression: if $\vec{v} = (v_x, v_y)$ and $\vec{w} = (w_x, w_y)$, then $\vec{v} \cdot \vec{w} = v_x w_x + v_y w_y$. We can implement it like this:

```
T dot(pt v, pt w) {return v.x*w.x + v.y*w.y;}
```

Dot product is often used for testing if two vectors are perpendicular, since we just need to test whether $\vec{v} \cdot \vec{w} = 0$:

```
bool isPerp(pt v, pt w) {return dot(v,w) == 0;}
```

It can also be used for finding the angle between two vectors, in $[0, \pi]$. Because of precision errors, we need to be careful not to call `acos` with a value that is out of the allowable range $[-1, 1]$.

```
double smallAngle(pt v, pt w) {
    double cosTheta = dot(v,w) / abs(v) / abs(w);
    if (cosTheta < -1) cosTheta = -1;
    if (cosTheta > 1) cosTheta = 1;
    return acos(cosTheta);
}
```

Since C++17, this can be simplified to:

```
double smallAngle(pt v, pt w) {
    return acos(clamp(dot(v,w) / abs(v) / abs(w), -1, 1));
}
```

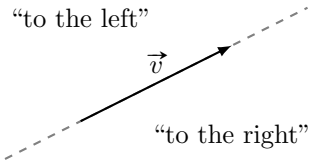
2.3.2 Cross product

The cross product $\vec{v} \times \vec{w}$ of two vectors $\vec{v}$ and $\vec{w}$ can be seen as a measure of how perpendicular they are. It is defined in 2D as

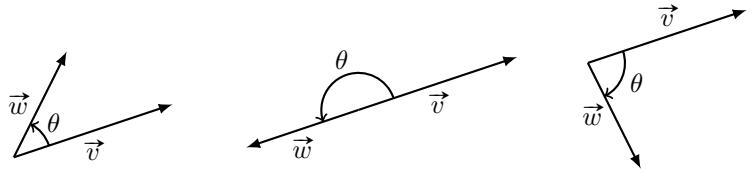
$$\vec{v} \times \vec{w} = \|\vec{v}\|\|\vec{w}\| \sin\theta$$

where $\|\vec{v}\|$ and $\|\vec{w}\|$ are the lengths of the vectors and θ is amplitude of the *oriented* angle from $\vec{v}$ to $\vec{w}$.

Since $\sin(-\theta) = -\sin(\theta)$, the sign of the angle matters, the cross product changes sign when the vectors are swapped: $\vec{w} \times \vec{v} = -\vec{v} \times \vec{w}$. It is positive if $\vec{w}$ is "to the left" of $\vec{v}$, and negative is $\vec{w}$ is "to the right" of $\vec{v}$.



In general, we take θ in $(-\pi, \pi]$, so that the dot product is positive if $0 < \theta < \pi$, negative if $-\pi < \theta < 0$ and zero if $\theta = 0$ or $\theta = \pi$, that is, if $\vec{v}$ and $\vec{w}$ are aligned.

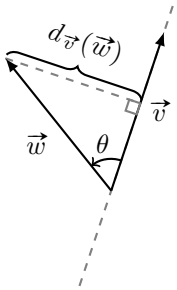


$0 < \theta < \pi$	$\theta = \pi$	$-\pi < \theta < 0$
$\vec{v} \times \vec{w} = 5$	$\vec{v} \times \vec{w} = 0$	$\vec{v} \times \vec{w} = -7$

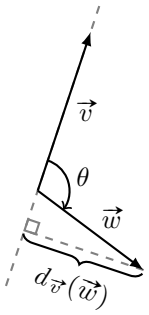
If we fix $\|\vec{v}\|$ and $\|\vec{w}\|$ as above, the cross product is maximal when the vectors are perpendicular with $\vec{w}$ on the left, because $\sin\theta = \sin(\pi/2) = 1$, and minimal when they are perpendicular with $\vec{w}$ on the right, because $\sin\theta = \sin(-\pi/2) = -1$.

Math insight

Because of the definition of sine in right triangles, cross product can also be interpreted in an interesting way (assuming $v \neq 0$): $\vec{v} \times \vec{w} = \|\vec{v}\| d_{\vec{v}}(\vec{w})$, where $d_{\vec{v}}(\vec{w}) = \|\vec{w}\| \sin\theta$ is the *signed* distance from the line that contains $\vec{v}$, with positive values on the left side of $\vec{v}$. In particular, this means that the cross product doesn't change if one of the vectors moves parallel to the other.



$d_{\vec{v}}(\vec{w}) = 2.69$



$d_{\vec{v}}(\vec{w}) = -2.37$

Like dot product, cross product has a very simple expression in cartesian coordinates: if $\vec{v} = (v_x, v_y)$ and $\vec{w} = (w_x, w_y)$, then $\vec{v} \times \vec{w} = v_x w_y - v_y w_x$:

```
T cross(pt v, pt w) {return v.x*w.y - v.y*w.x;}
```

Implementation trick

When using `complex`, we can implement both `dot()` and `cross()` with this trick, which is admittedly quite cryptic, but requires less typing and is less prone to typos:

```
T   dot(pt v, pt w) {return (conj(v)*w).x;}
T cross(pt v, pt w) {return (conj(v)*w).y;}
```

Here `conj()` is the complex conjugate: the conjugate of a complex number $a + bi$ is defined as $a - bi$. To verify that the implementation is correct, we can compute `conj(v)*w` as

$$(v_x - v_y i) * (w_x + w_y i) = (v_x w_x + v_y w_y) + (v_x w_y - v_y w_x) i$$

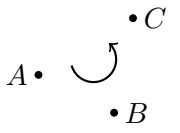
and see that the real and imaginary parts are indeed the dot product and the cross product.

Orientation

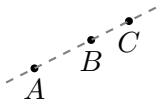
One of the main uses of cross product is in determining the relative position of points and other objects. For this, we define the function orient$(A, B, C) = \overrightarrow{AB} \times \overrightarrow{AC}$. be positive if C is on the left side of $\overrightarrow{AB}$, negative on the right side, and zero if C is on the line containing $\overrightarrow{AB}$. It is straightforward to implement:

```
T orient(pt a, pt b, pt c) {return cross(b-a,c-a);}
```

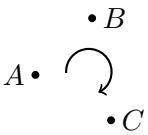
In other words, orient(A, B, C) is positive if when going from A to B to C we turn left, negative if we turn right, and zero if A, B, C are collinear.



left turn
orient$(A, B, C) > 0$



collinear
orient$(A, B, C) = 0$



right turn
orient$(A, B, C) < 0$

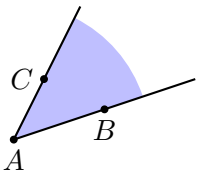
Its value is conserved by cyclic rotation, that is

$$\text{orient}(A, B, C) = \text{orient}(B, C, A) = \text{orient}(C, A, B)$$

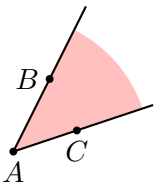
while swapping any two arguments switches the sign.

As an example of use, suppose we want to check if point X lies in the angle formed by lines AB and AC. We can follow this procedure:

1. check that orient$(A, B, C) \neq 0$ (otherwise the question is invalid);
2. if orient$(A, B, C) < 0$, swap B and C;

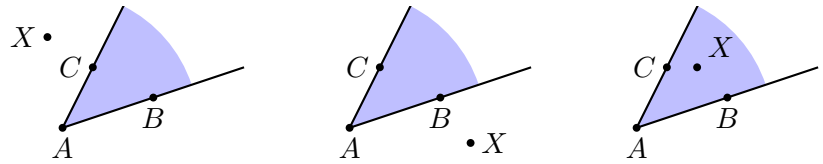


orient$(A, B, C) > 0$
no swap



orient$(A, B, C) < 0$
swap

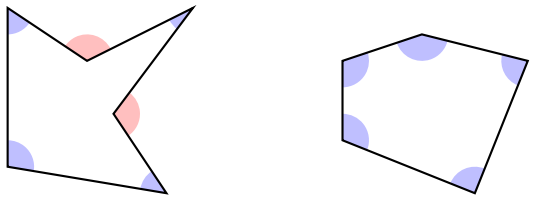
3. X is in the angle iff orient$(A, B, X) \geq 0$ and orient$(A, C, X) \leq 0$.



$\text{orient}(A,B,X) \geq 0$	$\text{orient}(A,B,X) < 0$	$\text{orient}(A,B,X) \geq 0$
$\text{orient}(A,C,X) > 0$	$\text{orient}(A,C,X) \leq 0$	$\text{orient}(A,C,X) \leq 0$
KO	KO	OK

```
bool inAngle(pt a, pt b, pt c, pt x) {
    assert(orient(a,b,c) != 0);
    if (orient(a,b,c) < 0) swap(b,c);
    return orient(a,b,x) >= 0 && orient(a,c,x) <= 0;
}
```

Another usage is checking if a polygon $P_1 \cdots P_n$ is convex: we compute the n orientations of three consecutive vertices $\text{orient}(P_i, P_{i+1}, P_{i+2})$, wrapping around from n to 1 when necessary. The polygon is convex if they are all ≥ 0 or all ≤ 0, depending on the order in which the vertices are given.



different signs
$\Rightarrow$ not convex

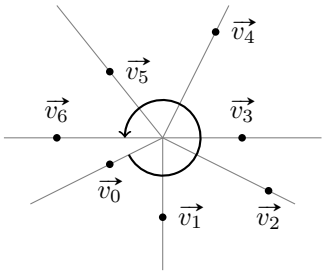
all the same sign
$\Rightarrow$ convex

```
bool isConvex(vector<pt> p) {
    bool hasPos=false, hasNeg=false;
    for (int i=0, n=p.size(); i<n; i++) {
        int o = orient(p[i], p[(i+1)%n], p[(i+2)%n]);
        if (o > 0) hasPos = true;
        if (o < 0) hasNeg = true;
    }
    return !(hasPos && hasNeg);
}
```

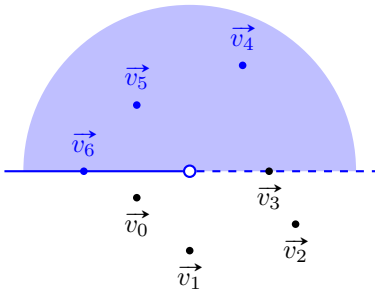
Polar sort

Because it can determine whether a vector points to the left or right of another, a common use of cross product is to sort vectors by direction. This is called polar sort: points are sorted in the order that a rotating ray emanating from the origin would touch them. Here, we will try to use cross

product to safely sort the points by their arguments in $(-\pi, \pi]$, that is the order that would be given by the `arg()` function for `complex`.[2]



In general $\vec{v}$ should go before $\vec{w}$ when $\vec{v} \times \vec{w} > 0$, because that means $\vec{w}$ is to the left of $\vec{v}$ when looking from the origin. This test works well for directions that are sufficiently close: for example, $\vec{v_2} \times \vec{v_3} > 0$. But when they are more than 180° apart in the order, it stops working: for example $\vec{v_1} \times \vec{v_5} < 0$. So we first need to split the points in two halves according to their argument:



If we isolate the points with argument in $(0, \pi]$ (region highlighted in blue) from those with argument in $(-\pi, 0]$, then the cross product always gives the correct order. This gives the following algorithm:

```
bool half(pt p) { // true if in blue half
    assert(p.x != 0 || p.y != 0); // the argument of (0,0) is undefined
    return p.y > 0 || (p.y == 0 && p.x < 0);
}
void polarSort(vector<pt> &v) {
    sort(v.begin(), v.end(), [](pt v, pt w) {
        return make_tuple(half(v), 0) <
               make_tuple(half(w), cross(v,w));
    });
}
```

[2]Sorting by using the `arg()` value would likely be a bad idea: there is no guarantee (that I know of) that vectors which are multiples of each other will have the same argument, because of precision issues. However, I haven't been to find an example where it fails for values small enough to be handled exactly with `long long`.

Indeed, the comparator will return `true` if either $\vec{w}$ is in the blue region and $\vec{v}$ is not, or if they are in the same region and $\vec{v} \times \vec{w} > 0$.

We can extend this algorithm in three ways:

- Right now, points that are in the exact same direction are considered equal, and thus will be sorted arbitrarily. If we want, we can use their magnitude as a tie breaker:

```
void polarSort(vector<pt> &v) {
    sort(v.begin(), v.end(), [](pt v, pt w) {
        return make_tuple(half(v), 0, sq(v)) <
               make_tuple(half(w), cross(v,w), sq(w));
    });
}
```

  With this tweak, if two points are in the same direction, the point that is further from the origin will appear later.

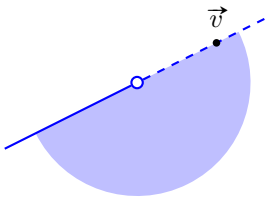
- We can perform a polar sort around some point O other than the origin: we just have to subtract that point O from the vectors $\vec{v}$ and $\vec{w}$ when comparing them. This as if we translated the whole plane so that O is moved to $(0, 0)$:

```
void polarSortAround(pt o, vector<pt> &v) {
    sort(v.begin(), v.end(), [](pt v, pt w) {
        return make_tuple(half(v-o), 0)) <
               make_tuple(half(w-o), cross(v-o, w-o));
    });
}
```

- Finally, the starting angle of the ordering can be modified easily by tweaking function `half()`. For example, if we want some vector $\vec{v}$ to be the first angle in the polar sort, we can write:

```
pt v = {/* whatever you want except 0,0 */};
bool half(pt p) {
    return cross(v,p) < 0 || (cross(v,p) == 0 && dot(v,p) < 0);
}
```

  This places the blue region like this:

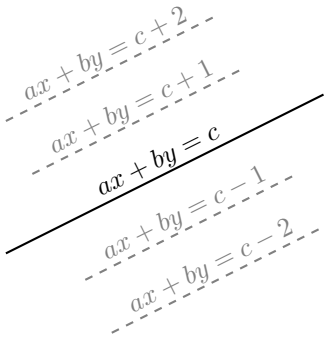


2.4 Lines

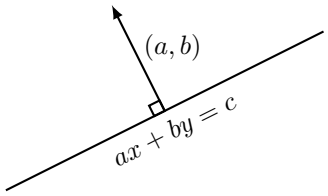
In this section we will discuss how to represent lines and a wide variety of applications.

2.4.1 Line representation

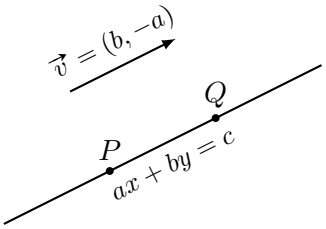
Lines are sets of points (x, y) in the plane which obey an equation of the form $ax + by = c$, with at least one of a and b nonzero. a and b determine the direction of the line, while c determines its position.



Equation $ax+by = c$ can be interpreted geometrically through dot product: if we consider (a, b) as a vector, then the equation becomes $(a, b) \cdot (x, y) = c$. This vector is perpendicular to the line, which makes sense: we saw in 2.3.1 that the dot product remains constant when the second vector moves perpendicular to the first.



The way we'll represent lines in code is based on another interpretation. Let's take vector $(b, -a)$, which is parallel to the line. Then the equation becomes a cross product $(b, -a) \times (x, y) = c$. Indeed, we saw in 2.3.2 that the cross product remains constant when the second vector moves parallel to the first.

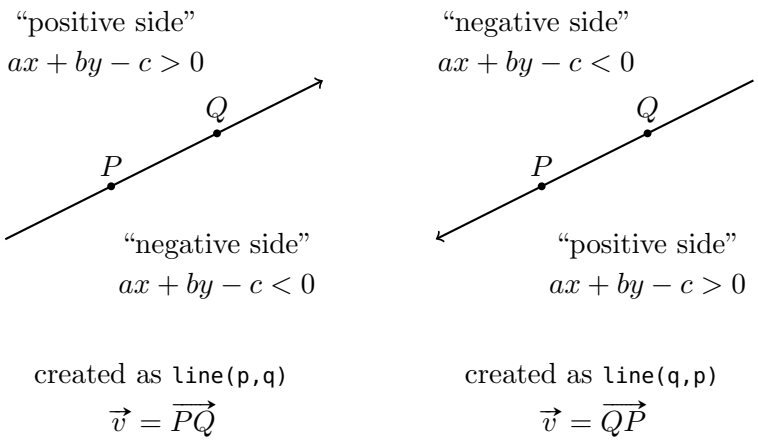


In this way, finding the equation of a line going through two points P and Q is easy: define the direction vector $\vec{v} = (b, -a) = \overrightarrow{PQ}$, then find c as $\vec{v} \times P$.

```
struct line {
    pt v; T c;
    // From points P and Q
    line(pt p, pt q): v(q-p), c(cross(v,p)) {}
    // From equation ax+by=c
    line(T a, T b, T c): v({b,-a}), c(c) {}
    // From direction vector v and offset c
    line(pt v, T c): v(v), c(c) {}

    // Will be defined later:
    // - these work with T = int/double
    T side(pt p);
    double dist(pt p);
    line perpThrough(pt p);
    bool cmpProj(pt p, pt q);
    line translate(pt t);
    // - these require T = double
    void shiftLeft(double dist);
    pt proj(pt p);
    pt sym(pt p);
}
```

A line will always have some implicit orientation, with two sides: the "positive side" of the line ($ax + by - c > 0$) is on the left of $\vec{v}$, while the "negative side" ($ax+by-c < 0$) is on the right of $\vec{v}$. In our implementation, this orientation is determined by the points that were used to create the line. We will represent it by an arrow at the end of the line. The figure below shows the differences that occur when creating a line as `line(p,q)` or `line(q,p)`.

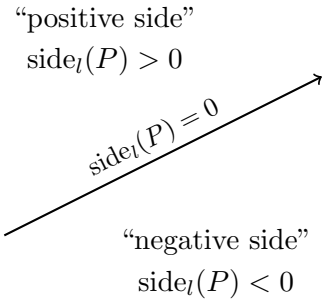


2.4.2 Side and distance

One interesting operation on lines is to find the value of $ax + by - c$ for a given point (x, y). For line l and point $P = (x, y)$, we will denote this operation as

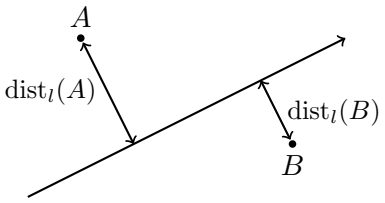
$$\text{side}_l(P) \coloneqq ax + by - c = \vec{v} \times P - c.$$

As we saw above, it can be used to determine which side of the line a certain point is, and $\text{side}_l(P) = 0$ if and only if P is on l (we will use this property a few times). You may notice that $\text{side}_{PQ}(R)$ is actually equal to $\text{orient}(P, Q, R)$.




```
T side(pt p) {return cross(v,p)-c;}
```

The $\text{side}_l(P)$ operation also gives the distance to l, up to a constant factor: the bigger $\text{side}_l(P)$ is, the further from line l. In fact, we can prove that $|\text{side}_l(P)|$ is $\|v\|$ times the distance between P and l (this should make sense if you've read the "mathy insight" in section 2.3.2).



This gives an easy implementation of distance:

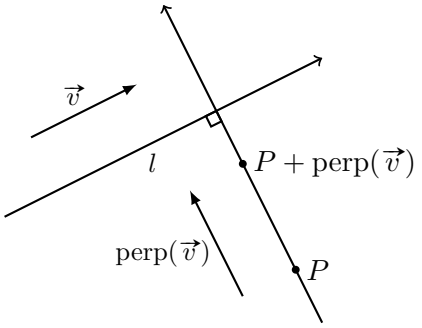
```
double dist(pt p) {return abs(side(p)) / abs(v);}
```

The squared distance can be useful to check if a point is within a certain integer distance of a line, because when using integers the result is exact if it is an integer.

```
double sqDist(pt p) {return side(p)*side(p) / (double)sq(v);}
```

2.4.3 Perpendicular through a point

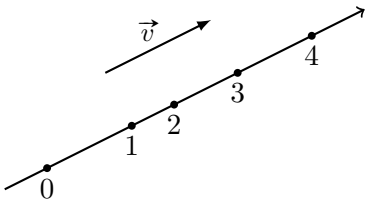
Two lines are perpendicular if and only if their direction vectors are perpendicular. Let's say we have a line l of direction vector $\vec{v}$. To find a line perpendicular to line l and which goes through a certain point P, we could define its direction vector as perp($\vec{v}$) (that is, $\vec{v}$ rotated by 90° counterclockwise, see section 2.2.3) and then try to work out c. However, it's simpler to just compute it as the line from P to $P + \text{perp}(\vec{v})$.




```
line perpThrough(pt p) {return {p, p + perp(v)};}
```

2.4.4 Sorting along a line

One subtask that often needs to be done in geometry problems is, given points on a line l, to sort them in the order they appear on the line, following the direction of $\vec{v}$.



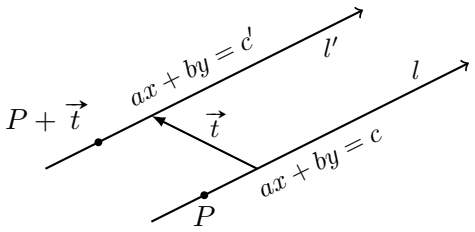
We can use the dot product to figure out the order of two points: a point A comes before a point B if $\vec{v} \cdot A < \vec{v} \cdot B$. So we can a comparator out of it.

```
bool cmpProj(pt p, pt q) {
    return dot(v,p) < dot(v,q);
}
```

In fact, this comparator is more powerful than we need: it is not limited to points on l and can compare two points by their orthogonal projection[3] on l. This should make sense if you have read the "mathy insight" in section 2.3.1.

2.4.5 Translating a line

If we want to translate a line l by vector $\vec{t}$, the direction vector $\vec{v}$ remains the same but we have to adapt c.



To find its new value c', we can see that for some point P on l, then $P + \vec{t}$ must be on the translated line. So we have

$$\text{side}_l(P) = \vec{v} \times P - c = 0$$
$$\text{side}_{l'}\left(P + \vec{t}\right) = \vec{v} \times \left(P + \vec{t}\right) - c' = 0$$

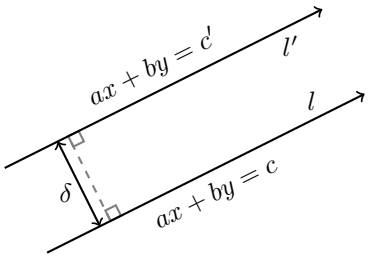
which allows us to find c':

$$c' = \vec{v} \times (P + \vec{t}) = \vec{v} \times P + \vec{v} \times \vec{t} = c + \vec{v} \times \vec{t}$$

```
line translate(pt t) {return {v, c + cross(v,t)};}
```

[3] If you don't know what an orthogonal projection is, read section 2.4.7.

A closely related task is shifting line l to the left by a certain distance δ (or to the right by $-\delta$).



This is equivalent to translating by a vector of norm δ perpendicular to the line, which we can compute as

$$\vec{t} = (\delta / \|\vec{v}\|) \operatorname{perp}(\vec{v})$$

so in this case c' becomes

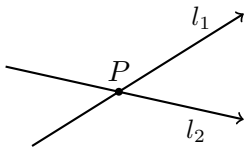
$$\begin{aligned} c' &= c + \vec{v} \times \vec{t} \\ &= c + (\delta / \|\vec{v}\|)(\vec{v} \times \operatorname{perp}(\vec{v})) \\ &= c + (\delta / \|\vec{v}\|)\|\vec{v}\|^2 \\ &= c + \delta\|\vec{v}\| \end{aligned}$$

```
line shiftLeft(double dist) {return {v, c + dist*abs(l.v)};}
```

2.4.6 Line intersection

There is a unique intersection point between two lines l_1 and l_2 if and only if $\overrightarrow{v_{l_1}} \times \overrightarrow{v_{l_2}} \neq 0$. If it exists, we will show that it is equal to

$$P = \frac{c_{l_1}\overrightarrow{v_{l_2}} - c_{l_2}\overrightarrow{v_{l_1}}}{\overrightarrow{v_{l_1}} \times \overrightarrow{v_{l_2}}}$$



We only show that it lies on l_1 (it should be easy to see that the expression

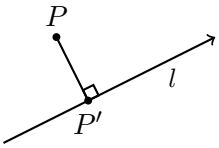
is actually symmetric in l_1 and l_2). It suffices to see that $\text{side}_{l_1}(P) = 0$:

$$\begin{aligned}
\text{side}_{l_1}(P) &= \overrightarrow{v_{l_1}} \times \left(\frac{c_{l_1}\overrightarrow{v_{l_2}} - c_{l_2}\overrightarrow{v_{l_1}}}{\overrightarrow{v_{l_1}} \times \overrightarrow{v_{l_2}}} \right) - c_{l_1} \\
&= \frac{c_{l_1}(\overrightarrow{v_{l_1}} \times \overrightarrow{v_{l_2}}) - c_{l_2}(\overrightarrow{v_{l_1}} \times \overrightarrow{v_{l_1}}) - c_{l_1}(\overrightarrow{v_{l_1}} \times \overrightarrow{v_{l_2}})}{\overrightarrow{v_{l_1}} \times \overrightarrow{v_{l_2}}} \\
&= \frac{-c_{l_2}(\overrightarrow{v_{l_1}} \times \overrightarrow{v_{l_1}})}{\overrightarrow{v_{l_1}} \times \overrightarrow{v_{l_2}}} \\
&= 0
\end{aligned}$$

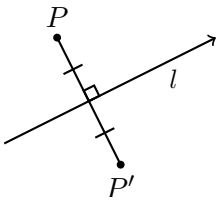
```
bool inter(line l1, line l2, pt &out) {
    T d = cross(l1.v, l2.v);
    if (d == 0) return false;
    out = (l2.v*l1.c - l1.v*l2.c) / d; // requires floating-point
        coordinates
    return true;
}
```

2.4.7 Orthogonal projection and reflection

The orthogonal projection of a point P on a line l is the point on l that is closest to P. The reflection of point P by line l is the point on the other side of l at the same distance and that has the same orthogonal projection.



projection



reflection

To compute the orthogonal projection of P, we need to move P perpendicularly to l until it is on the line. In other words, we need to find the factor k such that $\text{side}_l(P + k\,\text{perp}(\vec{v})) = 0$.

We compute

$$\begin{aligned}
\text{side}_l(P + k\,\text{perp}(\vec{v})) &= \vec{v} \times (P + k\,\text{perp}(\vec{v})) - c \\
&= \vec{v} \times P + \vec{v} \times k\,\text{perp}(\vec{v}) - c \\
&= (\vec{v} \times P - c) + k(\vec{v} \times \text{perp}(\vec{v})) \\
&= \text{side}_l(P) + k\|\vec{v}\|^2
\end{aligned}$$

so we find $k = -\,\text{side}_l(P)/\|\vec{v}\|^2$.

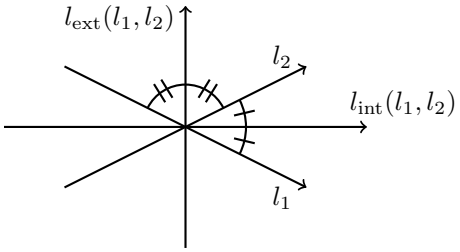
```
pt proj(pt p) {return p - perp(v)*side(p)/sq(v);}
```

To find the reflection, we need to move P in the same direction but twice the distance:

```
pt refl(pt p) {return p - perp(v)*2*side(p)/sq(v);}
```

2.4.8 Angle bisectors

An angle bisector of two (non-parallel) lines l_1 and l_2 is a line that forms equal angles with l_1 and l_2. We define the *internal bisector* $l_{\text{int}}(l_1, l_2)$ as the line whose direction vector points between the direction vectors of l_1 and l_2, and the *external bisector* $l_{\text{ext}}(l_1, l_2)$ as the other one. They are shown in the figure below.



An important property of bisectors is that their points are at equal distances from the original lines l_1 and l_2. In fact, if we give a sign to the distance depending on which side of the line we are on, we can say that $l_{\text{int}}(l_1, l_2)$ is the line whose points are at opposite distances from l_1 and l_2 while $l_{\text{ext}}(l_1, l_2)$ is the line whose points are at equal distances from l_1 and l_2.

For some line l can compute this signed distance as $\text{side}_l(P)/\|\vec{v}\|$ (in section 2.4.2 we used the absolute value of this to compute the distance). So $l_{\text{int}}(l_1, l_2)$ should be the line of all points for which

$$\frac{\text{side}_{l_1}(P)}{\|\overrightarrow{v_{l_1}}\|} = -\frac{\text{side}_{l_2}(P)}{\|\overrightarrow{v_{l_2}}\|}$$

$$\Leftrightarrow \quad \frac{\overrightarrow{v_{l_1}} \times P - c_{l_1}}{\|\overrightarrow{v_{l_1}}\|} = -\frac{\overrightarrow{v_{l_2}} \times P - c_{l_2}}{\|\overrightarrow{v_{l_2}}\|}$$

$$\Leftrightarrow \quad \left(\frac{\overrightarrow{v_{l_1}}}{\|\overrightarrow{v_{l_1}}\|} + \frac{\overrightarrow{v_{l_2}}}{\|\overrightarrow{v_{l_2}}\|}\right) \times P - \left(\frac{c_{l_1}}{\|\overrightarrow{v_{l_1}}\|} + \frac{c_{l_2}}{\|\overrightarrow{v_{l_2}}\|}\right) = 0$$

This is exactly an expression of the form $\text{side}_l(P) = \vec{v} \times P - c = 0$ which defines the points on a line. So it means that we have found the $\vec{v}$ and c that characterize $l_{\text{int}}(l_1, l_2)$:

$$\vec{v} = \frac{\overrightarrow{v_{l_1}}}{\|\overrightarrow{v_{l_1}}\|} + \frac{\overrightarrow{v_{l_2}}}{\|\overrightarrow{v_{l_2}}\|}$$

$$c = \frac{c_{l_1}}{\|\overrightarrow{v_{l_1}}\|} + \frac{c_{l_2}}{\|\overrightarrow{v_{l_2}}\|}$$

The reasoning is very similar for $l_{\text{ext}}(l_1, l_2)$, the only difference being signs. Both can be implemented as follows.

```
line intBisector(line l1, line l2, bool interior) {
    assert(cross(l1.v, l2.v) != 0); // l1 and l2 cannot be parallel!
    double sign = interior ? 1 : -1;
    return {l2.v/abs(l2.v) + sign * l1.v/abs(l1.v),
            l2.c/abs(l2.v) + sign * l1.c/abs(l1.v)};
}
```

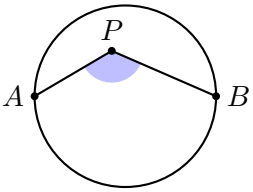
2.5 Segments

In this section we will discuss how to compute intersections and distances involving line segments.

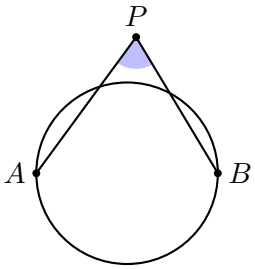
2.5.1 Point on segment

As an introduction, let's first see how to check if a point P lies on segment $[AB]$.

For this we will first define a useful subroutine `inDisk()` that checks if a point P lies on the disk of diameter $[AB]$. We know that the points on a disk are those which form angles $\geq 90°$ with the endpoints of a diameter. This can easily be checked by using dot product: $\widehat{APB} \geq 90°$ is equivalent $\overrightarrow{PA} \cdot \overrightarrow{PB} \leq 0$ (with the exception of $P = A, B$ in which case angle $\widehat{APB}$ is undefined).



$\overrightarrow{PA} \cdot \overrightarrow{PB} \leq 0$
in disk



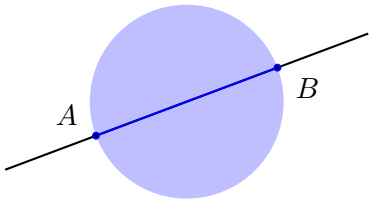
$\overrightarrow{PA} \cdot \overrightarrow{PB} > 0$
out of disk

```
bool inDisk(pt a, pt b, pt p) {
    return dot(a-p, b-p) <= 0;
}
```

Math insight

In fact, we can notice that $\overrightarrow{PA} \cdot \overrightarrow{PB}$ is equal to the power of point P with respect to the circle of diameter $[AB]$: if O is the center of that circle and r its radius, then $\overrightarrow{PA} \cdot \overrightarrow{PB} = |OP|^2 - r^2$. This makes it perfect for our purpose.

With this subroutine in hand, it is easy to check whether P is on segment $[AB]$: this is the case if and only if P is on line AB and also on the disk whose diameter is AB (and thus is in the part the line between A and B).



intersection of line and disk = segment

```
bool onSegment(pt a, pt b, pt p) {
    return orient(a,b,p) == 0 && inDisk(a,b,p);
}
```

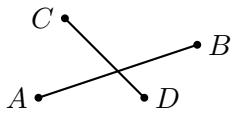
2.5.2 Segment-segment intersection

Finding the precise intersection between two segments $[AB]$ and $[CD]$ is quite tricky: many configurations are possible and the intersection itself might be empty, a single point or a whole segment.

To simplify things, we will separate the problem in two distinct cases:

1. Segments $[AB]$ and $[CD]$ intersect *properly*, that is, their intersection is one single point which is not an endpoint of either segment. This is easy to test with `orient()`.
2. In all other cases, the intersection, if it exists, is determined by the endpoints. If it is a single point, it must be one of A, B, C, D, and if it is a whole segment, it will necessarily start and end with points in A, B, C, D.

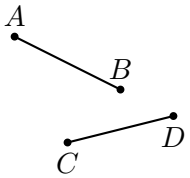
Let's deal with the first case: there is a single proper intersection point I. To test this, it suffices to test that A and B are on either side of line CD, and that C and D are on either side of line AB. If the test is positive, we find I as a weighted average of A and B.



proper intersection

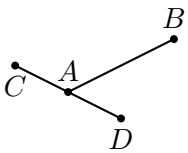
```
bool properInter(pt a, pt b, pt c, pt d, pt &i) {
    double oa = orient(c,d,a),
           ob = orient(c,d,b),
           oc = orient(a,b,c),
           od = orient(a,b,d);
    // Proper intersection exists iff opposite signs
    if (oa*ob < 0 && oc*od < 0) {
        i = (a*ob - b*oa) / (ob-oa);
        return true;
    }
    return false;
}
```

Then to deal with the second case, we will test for every point among A, B, C, D if it is on the other segment. If it is, we add it to a set S. Clearly, an endpoint cannot be in the middle of the intersection segment, so S will always contain 0, 1 or 2 distinct points, describing an empty intersection, a single intersection point or an intersection segment.



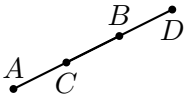
$S = \emptyset$

no intersection



$S = \{A\}$

intersection point



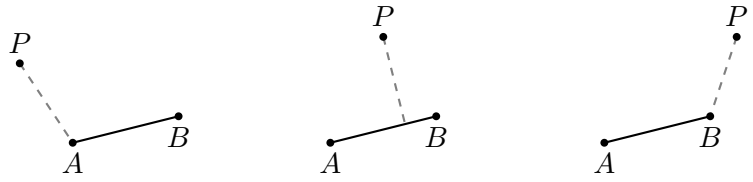
$S = \{B, C\}$

intersection segment

```
set<pt> inters(pt a, pt b, pt c, pt d) {
    pt out;
    if (properInter(a,b,c,d,out)) return {out};
    set<pt> s;
    if (onSegment(c,d,a)) s.insert(a);
    if (onSegment(c,d,b)) s.insert(b);
    if (onSegment(a,b,c)) s.insert(c);
    if (onSegment(a,b,d)) s.insert(d);
    return s;
}
```

2.5.3 Segment-point distance

To find the distance between segment $[AB]$ and point P, there are two cases: either the closest point to P on $[AB]$ is strictly between A and B, or it is one of the endpoints (A or B). The first case happens when the orthogonal projection of P onto AB is between A and B.



closest to A | closest to projection | closest to B

To check this, we can use the `cmpProj()` method in `line`.

```
double segPoint(pt a, pt b, pt p) {
    if (a != b) {
        line l(a,b);
        if (l.cmpProj(a,p) && l.cmpProj(p,b)) // if closest to projection
            return l.dist(p);                 // output distance to line
    }
    return min(abs(p-a), abs(p-b)); // otherwise distance to A or B
}
```

2.5.4 Segment-segment distance

We can find the distance between two segments $[AB]$ and $[CD]$ based on the segment-point distance if we separate into the same two cases as for segment-segment intersection:

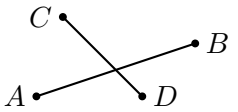
1. Segments $[AB]$ and $[CD]$ intersect properly, in which case the distance is of course 0.

2. In all other cases, the shortest distance between the segments is attained in at least one of the endpoints, so we only need to test the four endpoints and report the minimum.

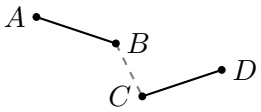
This can be readily implemented with the functions at our disposal.

```
double segSeg(pt a, pt b, pt c, pt d) {
    pt dummy;
    if (properInter(a,b,c,d,dummy))
        return 0;
    return min({segPoint(a,b,c), segPoint(a,b,d),
                segPoint(c,d,a), segPoint(c,d,b)});
}
```

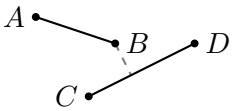
Some possible cases are illustrated below.



proper intersection



between points



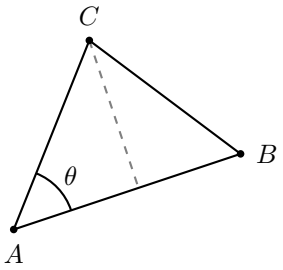
between point and line

2.6 Polygons

In this section we will discuss basic tasks on polygons: how to find their area and two ways to detect if a point is inside or outside them.

2.6.1 Polygon area

To compute the area of a polygon, it is useful to first consider the area of a triangle ABC.



We know that the area of this triangle is $\frac{1}{2}|AB||AC|\sin\theta$, because $|AC|\sin\theta$ is the length of the height coming down from C. This looks a lot like the

definition of cross product: in fact,

$$\frac{1}{2}|AB||AC|\sin\theta = \frac{1}{2}\left|\overrightarrow{AC} \times \overrightarrow{AC}\right|$$

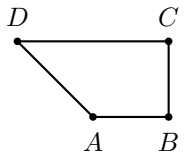
Since O is the origin, it can be implemented simply like this:

```
double areaTriangle(pt a, pt b, pt c) {
    return abs(cross(b-a, c-a)) / 2.0;
}
```

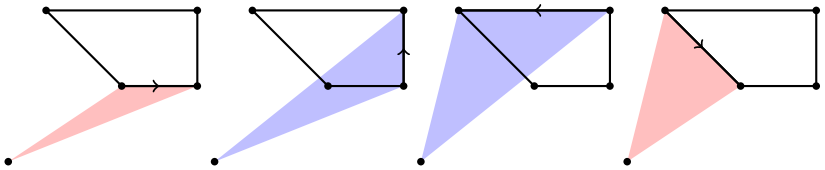
Now that we can compute the area of a triangle, the intuitive way to find the area of a polygon would be to

1. divide the polygon into triangles;
2. add up all the areas.

However, it turns out that reliably dividing a polygon into triangles is a difficult problem in itself. So instead we'll add and subtract triangle areas in a clever way. Let's take this quadrilateral as an example:



Let's take an arbitrary reference point O. Let's consider the vertices of $ABCD$ in order, and for every pair of consecutive points P_1, P_2, we'll add the area of OP_1P_2 to the total if $\overrightarrow{P_1P_2}$ goes counter-clockwise around O, and subtract it otherwise. Additions are marked in blue and subtractions in red.



We can see that this will indeed compute the area of quadrilateral $ABCD$. In fact, it works for any polygon (draw a few more examples to convince yourself).

Note that the sign (add or subtract) that we take for the area of OP_1P_2 is exactly the sign that the cross product takes. If we take the origin as reference point O, it gives this simple implementation:

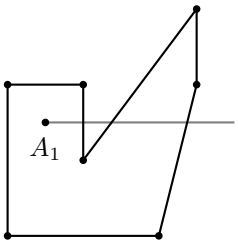
```
double areaPolygon(vector<pt> p) {
    double area = 0.0;
    for (int i = 0, n = p.size(); i < n; i++) {
        area += cross(p[i], p[(i+1)%n]); // wrap back to 0 if i == n-1
```

```
    }
    return abs(area) / 2.0;
}
```

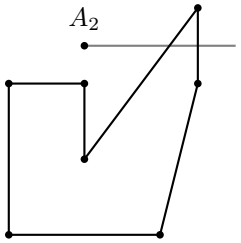
We have to take the absolute value in case the vertices are given in clockwise order. In fact, testing the sign of `area` is a good way to know whether the vertices are in counter-clockwise (positive) or clockwise (negative) order. It is good practice to always put your polygons in counter-clockwise order, by reversing the array of vertices if necessary, because some algorithms on polygons use this property.

2.6.2 Cutting-ray test

Let's say we want to test if a point A is inside a polygon $P_1 \cdots P_n$. Then one way to do it is to draw an imaginary ray from A that extends to infinity, and check how many times this ray intersects $P_1 \cdots P_n$. If the number of intersections is odd, A is inside, and if it is even, A is outside.

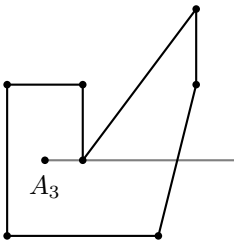


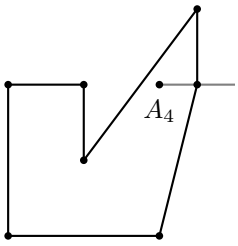
3 inters $\Rightarrow$ A_1 inside



2 inters $\Rightarrow$ A_2 outside

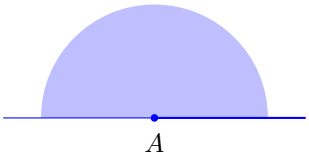
However, sometimes this can go wrong if the ray touches a vertex of the polygon, as below. The ray from A_3 intersects the polygon twice, but A_3 is inside. We can try to solve the issue by counting one intersection per segment touched, which would give three intersections for A_3, but then the ray from A_4 will intersect the polygon twice even though A_4 is inside.



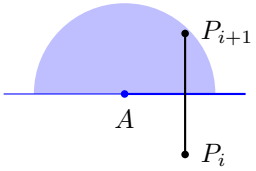


So we need to be more careful in defining what counts as an intersection. We will split the plane into two halves along the ray: the points lower than A, and the points at least as high (blue region). We then say that a segment

$[P_iP_{i+1}]$ crosses the ray right of A if it touches it *and* P_i and P_{i+1} are on opposite halves.

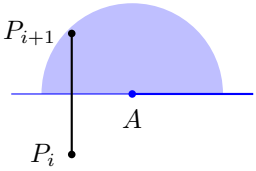


Below we show for some segments whether they are considered to cross the ray or not. We can see in the last two examples that the behavior is different if the segment touches the ray from below or from above.



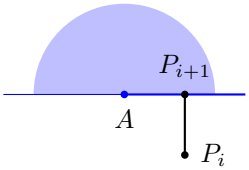
- touches ray: OK
- halves $\neq$: OK

$\Rightarrow$ crossing



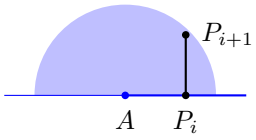
- touches ray: KO
- halves $\neq$: OK

$\Rightarrow$ no crossing



- touches ray: OK
- halves $\neq$: OK

$\Rightarrow$ crossing



- touches ray: OK
- halves $\neq$: KO

$\Rightarrow$ no crossing

Exercise

Verify that, with this new definition of crossing, A_3 and A_4 are correctly detected to be inside the polygon.

Checking the halves to which the points belong is easy, but checking that the segment touches the ray is a bit more tricky. We could check whether the segments $[P_i, P_{i+1}]$ and $[AB]$ intersect for B very far on the ray, but it actually we can do it more simply using orient: if P_i is below and P_{i+1} above, then orient(A, P_i, P_{i+1}) should be positive, and otherwise it should be negative. We can then implement this with the code below:

```
// true if P at least as high as A (blue part)
bool half(pt a, pt p) {
    return p.y >= a.y;
}
// check if [PQ] crosses ray from A
bool crossesRay(pt a, pt p, pt q) {
    return (half(q) - half(p)) * orient(a,p,q) > 0;
}
```

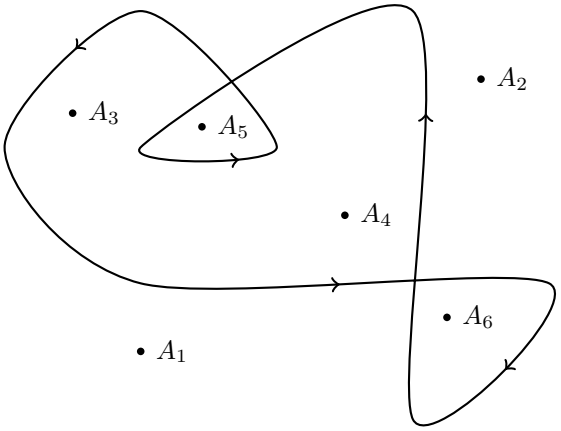
If we now return to the original problem, we still have to check whether A is on the boundary of the polygon. We can do that by using `onSegment()` defined in 2.5.1.

```
// if strict, returns false when A is on the boundary
bool inPolygon(vector<pt> p, pt a, bool strict = true) {
    int numCrossings = 0;
    for (int i = 0, n = p.size(); i < n; i++) {
        if (onSegment(p[i], p[(i+1)%n], a))
            return !strict;
        numCrossings += crossesRay(a, p[i], p[(i+1)%n]);
    }
    return numCrossings & 1; // inside if odd number of crossings
}
```

2.6.3 Winding number

Another way to test if A is inside polygon $P_1 \cdots P_n$ is to think of a string with one end attached at A and the other following the boundary of the polygon, doing one turn. If between the start position and the end position the string has done a full turn, then we are inside the polygon. If however the direction string has simply oscillated around the same position, then we are outside the polygon. Another way to test it is to place one finger on point A while another one follows the boundary of the polygon, and see if the fingers are twisted at the end.

This idea can be generalized to the *winding number*. The winding number of a closed curve around a point is the number of times this curve turns counterclockwise around the point. Here is an example.



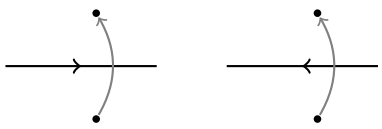
Points A_1 and A_2 are completely out of the curve so the winding number around them is 0 (no turn). Points A_3 and A_4 are inside the main loop, which goes counterclockwise, so the winding number around them is 1. The curve turns twice counterclockwise around A_5, so the winding number is 2. Finally the curve goes clockwise around A_6, for a winding number of -1.

Math insight

In fact, we can move the curve continuously without changing the winding number as long as we don't touch the reference point. Therefore we can "untie" loops which don't contain the point. That's why, when looking at A_3 or A_4, we can completely ignore the loops that contain A_5 and A_6.

Math insight

If we move the reference point while keeping the curve unchanged, the value of the winding number will only change when it crosses the curve. If it crosses the curve from the right (according to its orientation), the winding number increases by 1, and if it crosses it from the left, the winding number decreases by 1.



Exercise

What value will `areaPolygon()` (section 2.6.1) give when applied to a closed polyline that crosses itself, like the curve above, instead of a simple polygon? Assume we don't take the absolute value.
Solution: (select to reveal)

To compute the winding number, we need to keep track of the amplitude travelled, positive if counterclockwise, and negative if clockwise. We can use `smallAngle()` from section 2.3.1 to help us.

```
// amplitude travelled around point A, from P to Q
double angleTravelled(pt a, pt p, pt q) {
    double ampli = smallAngle(p-a, q-a);
    if (cross(a,p,q) > 0) return ampli;
    else return -ampli;
}
```

Another way to implement it uses the arguments of points:

```
double angleTravelled(pt a, pt p, pt q) {
    // remainder ensures the value is in [-pi,pi]
    return remainder(arg(q-a) - arg(p-a), 2*M_PI);
}
```

Then we simply sum it all up and figure out how many turns were made:

```
int windingNumber(vector<pt> p, pt a) {
    double ampli = 0;
    for (int i = 0, n = p.size(); i < n; i++)
        ampli += angleTravelled(a, p[i], p[(i+1)%n]);
    return round(ampli / (2*M_PI));
}
```

Warning

The winding number is not defined if the reference point is on the curve/polyline. If it is the case, this code will give arbitrary results, and potentially `(int)NAN`.

Angles of integer points

While the code above works, its use of floating-point numbers makes it non ideal, and when coordinates are integers, we can do better. We will define a new way to work with angles, as a type `angle`. This type will also be useful for other tasks, such as for sweep angle algorithms.

Instead of working with amplitudes directly, we will represent angles by a point and a certain number of full turns.[4] More precisely, in this case, we will use point (x, y) and number of turns t to represent angle $\text{atan2}(y, x) + 2\pi t$.

We start by defining the new type `angle`. We also define a utility function `t360()` which turns an angle by a full turn.

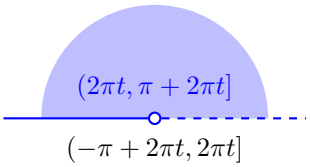
```
struct angle {
    pt d; int t = 0; // direction and number of full turns
    angle t180(); // to be defined later
    angle t360() {return {d, t+1};}
};
```

The range of angles which have the same value for t is $(-\pi + 2\pi t, \pi + 2\pi t]$.

We will now define a comparator between angles. The approach is the same as what we did for the polar sort in section 2.3.2, so we will reuse the function `half()` which separates the plane into two halves so that angles within one half are easily comparable:

```
bool half(pt p) {
    return p.y > 0 || (p.y == 0 && p.x < 0);
}
```

It returns **`true`** for the part highlighted in blue and **`false`** otherwise. Thus, in practice, it allows us to separate each range $(-\pi + 2\pi t, \pi + 2\pi t]$ into the subranges $(-\pi + 2\pi t, 2\pi t]$, for which `half()` returns **`false`**, and $(2\pi t, \pi + 2\pi t]$, for which `half()` returns **`true`**.



We can now write the comparator between angles, which is nearly identical to the one we used for polar sort, except that we first check the number of full turns `t`.

```
bool operator<(angle a, angle b) {
    return make_tuple(a.t, half(a.d), 0) <
```

[4] This approach is based on an original idea in [2], see "Angle.h".

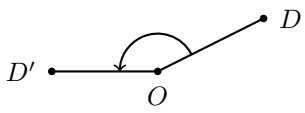
```
            make_tuple(b.t, half(b.d), cross(a.d,b.d));
}
```

We also define the function `t180()` which turns an angle by half a turn counterclockwise. The resulting angle has an opposite direction. To find the number of full turns t, there are two cases:

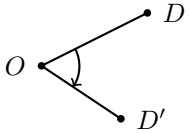
- if `half(d)` is **`false`**, we are in the lower half $(-\pi + 2\pi t, 2\pi t]$, and we will move to the upper half $(2\pi t, \pi + 2\pi t]$, without changing t;
- if `half(d)` is **`true`**, we are in the upper half $(2\pi t, \pi + 2\pi t]$, and we will move to $(-\pi + 2\pi(t+1), 2\pi(t+1)]$, the lower half for $t+1$.

```
angle t180() {return {d*(-1), t + half(d)};}
```

We will now implement the function that will allow us to compute the winding number. Consider an angle with direction point D. Given a new direction D', we would like to move the angle in such a way that if direction D' is to the left of D, the angle increases, and if D' is to the right of D, the angle decreases.



D' left of D
$\Rightarrow$ increase



D' right of D
$\Rightarrow$ decrease

In other words, we want the new angle to be an angle with direction D', and such that the difference between it and the old angle is at most 180°. We will use this formulation to implement the function:

```
angle moveTo(angle a, pt newD) {
    // check that segment [DD'] doesn't go through the origin
    assert(!onSegment(a.d, newD, {0,0}));

    angle b{newD, a.t};
    if (a.t180() < b) // if b more than half a turn bigger
        b.t--;        //     decrease b by a full turn
    if (b.t180() < a) // if b more than half a turn smaller
        b.t++;        //     increase b by a full turn
    return b;
}
```

We know that `b` as it is first defined is less than a full turn away from `a`, so the two conditions are enough to bring it within half a turn of `a`.

We can use this to implement a new version of `windingNumber()` very simply. We start at some vertex of the polygon, move vertex to vertex while

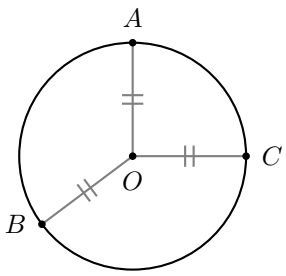
maintaining the angle, then read the number of full turns once we come back to it.

```
int windingNumber(vector<pt> p, pt a) {
    angle a{p.back()}; // start at last vertex
    for (pt d : p)
        a = moveTo(a, d); // move to first vertex, second, etc.
    return a.t;
}
```

2.7 Circles

2.7.1 Circumcircle

The *circumcircle* of a triangle ABC is the circle that passes through all three points A, B and C.



It is undefined if A, B, C are aligned, and unique otherwise. We can compute its center O this way:

```
pt circumCenter(pt a, pt b, pt c) {
    b = b-a, c = c-a; // consider coordinates relative to A
    assert(cross(b,c) != 0); // no circumcircle if A,B,C aligned
    return a + perp(b*sq(c) - c*sq(b))/cross(b,c)/2;
}
```

The radius can then be found by taking the distance to any of the three points, or directly taking the length of `perp(b*sq(c)- c*sq(b))/cross(b,c)/2`, which represents vector $\overrightarrow{AO}$.

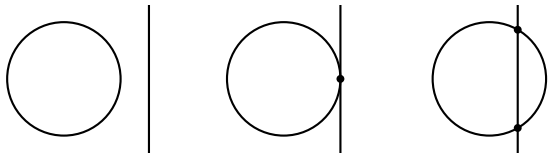
Math insight

The formula can be easily interpreted as the intersection point of the line segment bisectors of segments $[AB]$ and $[AC]$, when considering coordinates relative to A. Consider the bisector of $[AB]$. It is perpendicular to $[AB]$, so its direction vector is $\text{perp}(\overrightarrow{AB})$, and it passes through the middle point $\frac{1}{2}\overrightarrow{AB}$, so its constant term (variable `c` in the `line` structure) is $\text{perp}(\overrightarrow{AB}) \times \frac{1}{2}\overrightarrow{AB} = -\frac{1}{2}|AB|^2$. Similarly, the bisector of $[AC]$ is defined by direction vector $\text{perp}(\overrightarrow{AC})$ and constant term $-\frac{1}{2}|AC|^2$. We then just plug those into the formula for line intersection found in section 2.4.6:

$$\begin{aligned}\overrightarrow{AO} &= \frac{(-\frac{1}{2}|AB|^2)\,\text{perp}(\overrightarrow{AC}) - (-\frac{1}{2}|AC|^2)\,\text{perp}(\overrightarrow{AB})}{\text{perp}(\overrightarrow{AB}) \times \text{perp}(\overrightarrow{AC})} \\ &= \frac{\text{perp}(|AC|^2\overrightarrow{AB} - |AB|^2\overrightarrow{AC})}{2\ \overrightarrow{AB} \times \overrightarrow{AC}}\end{aligned}$$

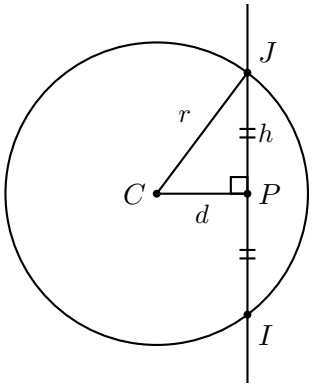
2.7.2 Circle-line intersection

A circle (C, r) and a line l have either 0, 1 or 2 intersection points.



0 intersections 1 intersection 2 intersections

Let's assume there are two intersection points I and J. We first find the midpoint of $[IJ]$. This happens to be the projection of C onto the line l, which we will call P.



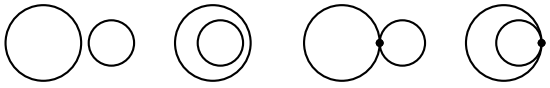
Once we have found P, to find I and J we need to move along the line by a certain distance h. By the Pythagorean theorem, $h = \sqrt{r^2 - d^2}$ where d is the distance from C to l.

This gives the following implementation (note that we have to divide by $\|\vec{v_l}\|$ so that we move by the correct distance). It returns the number of intersections, and places them in `out`. If there is only one intersection, `out.first` and `out.second` are equal.

```
int circleLine(pt c, double r, line l, pair<pt,pt> &out) {
    double h2 = r*r - l.sqDist(c);
    if (h2 < 0) return 0; // the line doesn't touch the circle
    pt p = l.proj(c); // point P
    pt h = l.v*sqrt(h2)/abs(l.v); // vector parallel to l, of length h
    out = {p-h, p+h};
    return 1 + (h2 > 0);
}
```

2.7.3 Circle-circle intersection

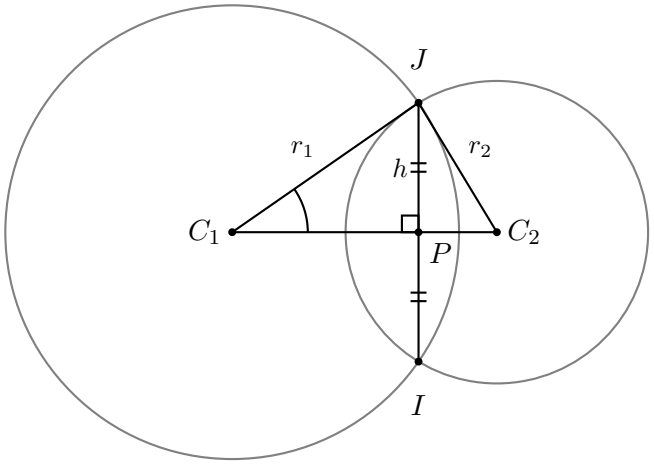
Similarly to the previous section, two circles (C_1, r_1) and (C_2, r_2) can have either 0, 1, 2 or an infinity of intersection points (in case the circles are identical).



0 intersections 1 intersection

2 intersections ∞ intersections

As before, we assume there are two intersection points I and J and we try to find the midpoint of $[IJ]$, which we call P.



Let $d = |C_1C_2|$. We know from the law of cosines on C_1C_2J that

$$\cos(\measuredangle C_2C_1J) = \frac{d^2 + r_1^2 - r_2^2}{2dr_1}$$

and since C_1PJ is a right triangle,

$$|C_1P| = r_1 \cos(\measuredangle C_2C_1J) = \frac{d^2 + r_1^2 - r_2^2}{2d}$$

which allows us to find P.

Now to find $h = |PI| = |PJ|$, we apply the Pythagorean theorem on triangle C_1PJ, which gives $h = \sqrt{r_1^2 - |C_1P|^2}$.

This gives the following implementation, which works in a very similar way to the code in the previous section. It aborts if the circles are identical.

```
int circleCircle(pt c1, double r1, pt c2, double r2, pair<pt,pt> &out) {
    pt d=c2-c1; double d2=sq(d);
    if (d2 == 0) {assert(r1 != r2); return 0;} // concentric circles
    double pd = (d2 + r1*r1 - r2*r2)/2; // = |C_1P| * d
    double h2 = r1*r1 - pd*pd/d2; // = h^2
    if (h2 < 0) return 0;
    pt p = c1 + d*pd/d2, h = perp(d)*sqrt(h2/d2);
    out = {p-h, p+h};
    return 1 + (h2 > 0);
}
```

Math insight

Let's check that if $d \neq 0$ and variable `h2` in the code is nonnegative, there are indeed 1 or 2 intersections (the opposite is clearly true: if `h2` is negative, the length h cannot exist). The value of `h2` is

$$\begin{aligned} & r_1^2 - \frac{(d^2 + r_1^2 - r_2^2)^2}{4d^2} \\ &= \frac{4d^2 r_1^2 - (d^2 + r_1^2 - r_2^2)^2}{4d^2} \\ &= \frac{-d^4 - r_1^4 - r_2^4 + 2d^2 r_1^2 + 2d^2 r_2^2 + 2r_1^2 r_2^2}{4d^2} \\ &= \frac{(d + r_1 + r_2)(d + r_1 - r_2)(d + r_2 - r_1)(r_1 + r_2 - d)}{4d^2} \end{aligned}$$

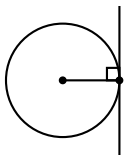
Let's assume this is nonnegative. Thus an even number of those conditions are false:

$$d + r_1 \geq r_2 \qquad d + r_2 \geq r_1 \qquad r_1 + r_2 \geq d$$

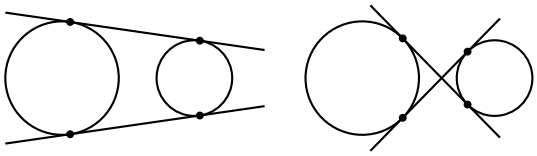
Since $d, r_1, r_2 \geq 0$, no two of those can be simultaneously false, so they must all be true. As a consequence, the triangle inequalities are verified for d, r_1, r_2, showing the existence of a point at distance r_1 from C_1 and distance r_2 from C_2.

2.7.4 Tangent lines

We say that a line is tangent to a circle if the intersection between them is a single point. In this case, the ray going from the center to the intersection point is perpendicular to the line.

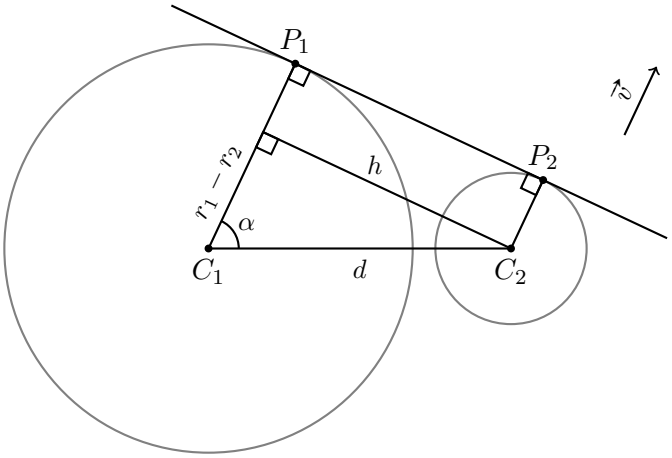


Here we will try and find a line which is tangent to two circles (C_1, r_1) and (C_2, r_2). There are two types of such tangents: outer tangents, for which both circles are on the same side of the line, and inner tangents, for which the circles are on either side.



outer tangents inner tangents

We will study the case of outer tangents. Our first goal is to find a unit vector parallel to the rays $[C_1P_1]$ and $[C_2P_2]$, in other words, we want to find $\vec{v} = \overrightarrow{C_1P_1}/r_1$. To do this we will try to find angle α marked on the figure.



If we project C_2 onto the ray from C_1, this forms a right triangle with hypothenuse $d = |C_1C_2|$ and adjacent side $r_1 - r_2$, which means $\cos\alpha = \frac{r_1-r_2}{d}$. By the Pythagorean theorem, the third side is $h = \sqrt{d^2 - (r_1 - r_2)^2}$, and we can compute $\sin\alpha = \frac{h}{d}$.

From this we find $\vec{v}$ in terms of $\overrightarrow{C_1C_2}$ and $\operatorname{perp}\left(\overrightarrow{C_1C_2}\right)$ as

$$\begin{aligned}\vec{v} &= \cos\alpha\left(\overrightarrow{C_1C_2}/d\right) \pm \sin\alpha\left(\operatorname{perp}\left(\overrightarrow{C_1C_2}\right)/d\right)\\ &= \frac{(r_1 - r_2)\,\overrightarrow{C_1C_2} \pm h\operatorname{perp}\left(\overrightarrow{C_1C_2}\right)}{d^2}\end{aligned}$$

where the $\pm$ depends on which of the two outer tangents we want to find.

We can then compute P_1 and P_2 as

$$P_1 = C_1 + r_1\vec{v} \quad \text{and} \quad P_2 = C_2 + r_2\vec{v}$$

Exercise

Study the case of the inner tangents and show that it corresponds exactly to the case of the outer tangents if r_2 is replaced by $-r_2$. This will allow us to write a function that handles both cases at once with an additional argument `bool inner` and this line:

```
if (inner) r2 = -r2;
```

This gives the following code. It returns the number of tangents of the specified type. Besides,

- if there are 2 tangents, it fills `out` with two pairs of points: the pairs of tangency points on each circle (P_1, P_2), for each of the tangents;
- if there is 1 tangent, the circles are tangent to each other at some point P, `out` just contains P 4 times, and the tangent line can be found as `line(c1,p).perpThrough(p)` (see 2.4.3);
- if there are 0 tangents, it does nothing;
- if the circles are identical, it aborts.

```
int tangents(pt c1, double r1, pt c2, double r2, bool inner, V<pair<pt,pt
    >> &out) {
    if (inner) r2 = -r2;
    pt d = c2-c1;
    double dr = r1-r2, d2 = sq(d), h2 = d2-dr*dr;
    if (d2 == 0 || h2 < 0) {assert(h2 != 0); return 0;}
    for (int sign : {-1,1})
        pt v = (d*dr + perp(d)*sqrt(h2)*sign)/d2;
        out.pb({c1 + v*r1, c2 + v*r2})
    }
    return 1 + (h2 > 0);
}
```

Conveniently, the same code can be used to find the tangent to a circle passing through a point by setting `r2` to 0 (in which case the value of `inner` doesn't matter).

Appendix A

Omitted proofs

A.1 Precision bounds for $+, -, \times$

We first formulate an assumption on the rounding operation round(). In this section, M and ϵ are positive real constants.

Assumption 1. *The rounding of a value x has a relative error of at most ϵ. Therefore, if $|x| \leq M^d$, as we will always assume of a d-dimensional value, then*

$$|\operatorname{round}(x) - x| \leq M^d \epsilon$$

To give a solid formalism to our notions of d-dimensional values and "computed in n operations", we introduce the following recursive definition.

Definition 1. *A quadruplets (x, x', d, n) is a* valid computation *if $|x| \leq M^d$ and one of these holds:*

(a) $x = x'$, $n = 0$;

(b) (a, a', d_a, n_a) *and* (b, b', d_b, n_b) *are valid computations,* $n = n_a + n_b + 1$ *and either:*

(i) $d = d_a = d_b$, $x = a + b$, $x' = \operatorname{round}(a' + b')$ *and* $|a' + b'| \leq M^d$;
(ii) $d = d_a = d_b$, $x = a - b$, $x' = \operatorname{round}(a' - b')$ *and* $|a' - b'| \leq M^d$;
(iii) $d = d_a + d_b$, $x = ab$, $x' = \operatorname{round}(a'b')$ *and* $|a'b'| \leq M^d$.

Note that valid computations are strongly limited by the assumptions we place on the magnitude of the results, both theoretical and actual.

Theorem 1. *If (x, x', d, n) is a valid computation, then*

$$|x' - x| \leq M^d \left((1 + \epsilon)^n - 1\right).$$

We will prove the theorem by induction on the structure of valid computations. We will separate the proof into two lemmas: first addition and subtraction together in Lemma 2, then multiplication in Lemma 3.

Lemma 1. *Let $f(x) = (1+\epsilon)^x - 1$. If $a, b > 0$, then $f(a) + f(b) \leq f(a+b)$.*

Proof. Clearly, f is convex. From convexity we find

$$f(a) \leq \frac{b}{a+b} f(0) + \frac{a}{a+b} f(a+b)$$
$$f(b) \leq \frac{a}{a+b} f(0) + \frac{b}{a+b} f(a+b).$$

Therefore, $f(a) + f(b) \leq f(0) + f(a+b) = f(a+b)$. □

Lemma 2 (addition and subtraction)**.** *Let operator $*$ be either $+$ or $-$. If (a, a', d, n_a) and (b, b', d, n_b) are two valid computations for which Theorem 1 holds and $(a * b, \text{round}(a' * b'), d, n_a + n_b + 1)$ is a valid computation, then Theorem 1 holds for it as well.*

Proof. From the hypotheses know that

$$|a' - a| \leq M^d \left((1+\epsilon)^{n_a} - 1\right)$$
$$|b' - b| \leq M^d \left((1+\epsilon)^{n_b} - 1\right)$$
$$|a' * b'| \leq M^d.$$

We find

$$\begin{aligned}
&|\,\text{round}(a' * b') - (a * b)| \\
&\qquad = |(\text{round}(a' * b') - (a' * b')) + ((a' * b') - (a * b))| \\
&\qquad \leq |\,\text{round}(a' * b') - (a' * b')| + |(a' - a) * (b' - b)| \\
&\qquad \leq M^d \epsilon + |a' - a| + |b' - b| \\
&\qquad \leq M^d \epsilon + M^d \left((1+\epsilon)^{n_a} - 1\right) + M^d \left((1+\epsilon)^{n_b} - 1\right) \\
&\qquad = M^d \left[f(1) + f(n_a) + f(n_b)\right] \\
&\qquad \leq M^d f(n_a + n_b + 1) \\
&\qquad = M^d \left((1+\epsilon)^{n_a + n_b + 1} - 1\right)
\end{aligned}$$

where the step before last follows from two applications of Lemma 1. □

Lemma 3 (multiplication)**.** *If (a, a', d_a, n_a) and (b, b', d_b, n_b) are two valid computations for which Theorem 1 holds and $(ab, \text{round}(a'b'), d_a + d_b, n_a + n_b + 1)$ is a valid computation, then Theorem 1 holds for it as well.*

Proof. From the hypotheses we know that

$$\begin{aligned}
|a| &\le M^d \\
|b| &\le M^d \\
|a' - a| &\le M^{d_a} \left((1+\epsilon)^{n_a} - 1\right) \\
|b' - b| &\le M^{d_a} \left((1+\epsilon)^{n_b} - 1\right) \\
|a'b'| &\le M^{d_a + d_b}.
\end{aligned}$$

We find

$$\begin{aligned}
&|\operatorname{round}(a'b') - ab)| \\
&\quad = |(\operatorname{round}(a'b') - a'b') + (a'b' - ab)| \\
&\quad \le |\operatorname{round}(a'b') - a'b'| + |(a' - a)b + (b' - b)a + (a' - a)(b' - b)| \\
&\quad \le M^{d_a+d_b}\epsilon + |a' - a||b| + |b' - b||a| + |a' - a||b' - b| \\
&\quad \le M^{d_a+d_b}\epsilon + M^{d_a}\left((1+\epsilon)^{n_a} - 1\right) M^{d_b} + M^{d_b}\left((1+\epsilon)^{n_b} - 1\right) M^{d_a} \\
&\qquad\qquad + M^{d_a}\left((1+\epsilon)^{n_a} - 1\right) M^{d_b}\left((1+\epsilon)^{n_b} - 1\right) \\
&\quad = M^{d_a+d_b}\big[\epsilon + \left((1+\epsilon)^{n_a} - 1\right) + \left((1+\epsilon)^{n_b} - 1\right) \\
&\qquad\qquad + \left((1+\epsilon)^{n_a} - 1\right)\left((1+\epsilon)^{n_b} - 1\right)\big] \\
&\quad = M^{d_a+d_b}\left[\epsilon + (1+\epsilon)^{n_a+n_b} - 1\right] \\
&\quad = M^{d_a+d_b}\left[f(1) + f(n_a + n_b)\right] \\
&\quad \le M^{d_a+d_b} f(n_a + n_b + 1) \\
&\quad = M^{d_a+d_b}\left((1+\epsilon)^{n_a+n_b+1} - 1\right)
\end{aligned}$$

where the step before last follows from Lemma 1. □

Proof of Theorem 1. By induction on the recursive structure of valid computations (see Definition 1). Case (a) is trivial because $|x' - x| = 0$. For case (b), the inductive step for (i) and (ii) follows from Lemma 2 while that of (iii) follows from Lemma 3. □

Bibliography

[1] Lutz Kettner et al. "Classroom examples of robustness problems in geometric computations". In: *Computational Geometry* 40.1 (2008), pp. 61–78. URL: https://people.mpi-inf.mpg.de/~kettner/pub/nonrobust_esa_04.pdf.

[2] Simon Lindholm et al. *KTH ACM Contest Template Library*. 2017. URL: https://github.com/kth-competitive-programming/kactl.