École polytechnique de Louvain (EPL)

Geometry in competitive programming

Dissertation presented by
Victor Lecomte

for obtaining the master's degree in
Computer Science and Engineering

Supervisor
Yves Deville

Co-supervisor
François Aubry

Readers
François Aubry, Yves Deville, Jean-François Remacle

Academic year 2017-2018

Abstract

Competitive programming consists in the organization of programming contests, which see the world's most talented coders compete to solve problems of an algorithmic nature by implementing efficient programs as quickly as possible. Recently, geometry problems have had an increasing importance in those contests, but many contestants dislike them for being too technical or causing too precision issues linked to the use of floating-point numbers.

We believe that those complaints are for the largest part due to a lack of specific training on the basics of computational geometry. We therefore introduce a new handbook of computational geometry, aimed primarily at the competitive programming public. It treats the subject in a practice-oriented way, providing code snippets for every operation, and explaining them in intuitive terms with the help of a large amount of illustrative figures. Its contents focus on the foundations of both 2D and 3D computational geometry, as well as the management of floating-point imprecisions, where we present a model of error propagation based on magnitude conditions and dimensionality of values.

In addition, we present an original geometry contest whose problems showcase innovative notions and techniques, especially in 3D geometry. We hope that in the future, problem setters can reuse and adapt those ideas in order to foster variety and expand the scope of geometry as a competitive programming subfield.

Contents

Part I

Main contents

Chapter 1

Introduction

In this chapter, we give a short introduction to competitive programming and the role of geometry in it, then outline the goals and structure of this thesis.

1.1 About competitive programming

Competitive programming is the participation in programming contests, which test the participants' ability to write programs that solve problems of an algorithmic or mathematical nature in an efficient way.

A typical problem in a programming contest describes the problem to be solved, specifies how the input data is provided and how the output should be written. The participant then has to write a program that will read the input data (typically in `stdin`), compute the result, then write the output (typically in `stdout`).

When the participant submits a program, it will be compiled by the judging system then executed on a batch of tests, which is designed to comprehensively test the program in many situations and detect any flaw. If for each test case the program runs successfully within the time and memory limits, and the judging system evaluates the answer as correct, then the solution is *Accepted* and awarded points. Other verdicts include *Wrong answer* if the output is incorrect, *Time limit exceeded* if the program took too much time, or *Runtime error* if the program crashed.

Besides this standard "read input, compute result, write output" scheme, there are also interactive tasks, in which the participant's solution has to interact directly with the judging system, exchanging information back and forth. Also, in most tasks, finding a fast enough solution is the main challenge, there are some tasks in which the memory limit might be quite limiting, or some other resource determined by the problem statement (like the number of queries the solution may ask the judging system, in interactive tasks).

Contestants are free to implement their solutions in a variety of languages, but C++ is the most popular, because of its execution speed and relatively rich standard library. To succeed in contests, good implementation skills and a mastery of the language used are as important as talent in algorithm design and problem solving.

For more details about competitive programming, see [4].

1.2 Types of programming contests

Programming contests can be mostly classified under four categories. The type of problems is generally the same across the board, but the contest formats vary a lot.

Olympiads in Informatics Often called OIs, olympiads in informatics are contests for secondary school students. The most famous OI is the IOI (International Olympiad in Informatics), held every year for about a week in the summer. National OIs organized in most countries, usually as part of the selection process for IOI, and there are also many regional OIs, like APIO (Asia-Pacific), Balkan OI, CEOI (Central Europe), etc. OIs are individual contests and are characterized by relatively few tasks (at IOI, 3 tasks for 5 hours), each containing many subtasks that give partial score to partial or suboptimal solutions.

ACM-ICPC Composed of the World Finals, dozens of regionals serving as qualifiers, and many subregionals for local practice, the ACM-ICPC contests are for teams of 3 university students. Teams have 5 hours to solve usually 9-13 problems. All problems are worth 1 point even though their difficulty varies greatly, and ties are broken with submission times, so speed is key. There is no partial score.

Regular online contests A handful of websites hold regular individual contests all through the year, at a rate of about one contest per week. Famous contest platforms include Codeforces, AtCoder, CodeChef and CS Academy. Those contests are usually about 2 hours in length, with 4-5 problems whose score varies according to the difficulty. Although the contests are mostly for fun, after the contest, the contestants' *rating*, which is a reflection of their recent performance, is updated.

Corporate contests Several IT companies organize their own annual programming competitions to advertize themselves and recruit talented programmers. Those generally consist of one or more online qualifiers and one onsite final with prizes. The format of the contests is very similar to regular online contests. Well-known examples include Google Code Jam, Facebook Hacker Cup and Yandex Algorithm.

1.3 Geometry in competitive programming

Computational geometry problems commonly appear in all contest types mentioned in section 1.2, except in OIs, where it is mostly avoided. Indeed, there are strong arguments against the use of problems which might cause numerical difficulties in pre-university competitions [2], and this is reflected in the IOI syllabus [5].

In recent years, geometry has been flourishing, especially in ACM-ICPC contests, where problemsets nearly always include at least one geometry problem. They are often among the hardest problems of the contest, and thus have an important role in determining the winners. For this, and because special training on geometry problems can greatly increase the speed and accuracy at which a contestant is able to solve a problem, mastery of geometry has been called a key to success at the World Finals.

But geometry is also a controversial subfield of competitive programming, as many people get frustrated when their code fails due to precision issues or forgetting to handle some edge cases, which are more frequent than in other types of problems. It is also often the case that contestants stumble on simple geometry subroutines if they are not used to juggling with geometry primitives and have never formally studied the subject.

Currently, the vast majority of geometry problems asked in contests are 2D problems, though occasional forays into 3D can also be seen, for example at NWERC 2015[1] or Google Code Jam 2018[2].

1.4 Goals and structure of this thesis

We believe that the main reason why many contestants feel a certain dislike towards geometry problems is that they base most of their knowledge of computational geometry from their basic euclidean geometry knowledge, and from occasionally browsing the web to read about specific algorithms or

[1] Northwestern Europe Regional Contest 2015, Problem F "Flight Plan Evaluation" by Per Austrin. `https://open.kattis.com/problems/flightplan`

[2] Google Code Jam 2018, Round 3, Problem "Raise the Roof". `https://codejam.withgoogle.com/2018/challenges/0000000000007707/dashboard/000000000004b90d`

primitives and to seek code snippets. As such, they never studied the subject formally and aren't always aware of the best practices to follow, and all the issues that can happen if they do it wrong. Also, if they are inexperienced, they may often get stuck on basic tasks that they never learned to do.

In order to remedy this situation, our first goal was to *write a comprehensive and practice-oriented reference on the basic tools of computational geometry*, to help raise the general knowledge of contestants, and give problem setters a strong basis to create problems from. We realized this as a freely available handbook of geometry for competitive programmers, which is given in part II. It is still being worked on, and more information on the approach, the current content covered and the plans for the future are outlined in chapter 2.

Because we feel that geometry in programming contests has the potential to be a wider and more varied subject than it is at the moment, in particular by incorporating more 3D problems, our second goal was to *showcase some innovative notions and elements that could be used in more contests in the future*. We realized this by organizing an online contest composed entirely of geometry problems, each introducing some geometric elements and notions that are rarely seen in current contests, and focusing on 3D geometry. The organization of the contest is presented in chapter 3.

The overall structure of this thesis is as follows:

- Part I contains the main contents of the thesis, namely this introduction (chapter 1), a presentation of the handbook and the writing approach (chapter 2), a report on the organization of the online contest (chapter 3), and some concluding notes on the work in this thesis (chapter 4).
- Part II contains the handbook itself, as described in chapter 2. Because of the page limit on the main contents of a Master's thesis, it should be considered to be an appendix.

Chapter 2

Writing a handbook of geometry

In this chapter, we explain the approach that was taken while writing the handbook in part II, outline some points of interest in the current contents, and talk about future plans for it.

2.1 Approach and writing style

In order to fulfill its mission as a standard and accessible reference for geometry in programming contests, and to adequately adapt the material to its target audience, we worked in line with the following the following principles. Of course, satisfying all of them at the same often proved challenging, so in many cases we had to try out many different explanations and implementations until we could reach a good compromise.

Low prerequisites Because the main target audience is undergraduate students or younger, and not all of them have a very advanced level in math, special care was taken to start from the very basics whenever possible, and when necessary not assume a level in mathematics higher than what is taught in high school.

Practice-oriented To accomodate for an audience that likely just wants to know how to implement the problem they are interested in, even in more theoretical considerations we are careful to go straight to the point and always complement the new notions that are introduced with their implementation and practical uses.

Abundance of figures In an effort to make the discussion simple and clear, we privileged explaining our ideas and the different situations that might occur using figures whenever possible. Though creating all those figures seriously increased the writing time, we believe that they are one of the best features of this handbook.

Short and consistent code A defining feature of this handbook is that it provides sample code for all algorithms explained. Because contestants in many contests such as the ACM-ICPC, all code has to be written from scratch during the contest, the implementations are designed to be as short as possible while remaining clear, efficient and easy to use. The code snippets taken together are meant to form a complete library, and effort was put into its practical reusability and modifiability.

Managing precision issues As explained in the next section, we have a whole chapter dedicated to ways in which precision issues can arise and how to manage them. In addition, despite the length constraints that we tried to stick to, we made sure to make all of our code snippets as robust to precision issues as they can be, and we highlighted the cases in which the reader needs to pay extra attention to them.

2.2 Current contents

Because we believe that the foundations of computational geometry in 2D and 3D are the areas that have the most urgent need for such a manual, rather than the more complex algorithms that are built on it, that is what we started with, while the next section describes our future plans. There are three large chapters in the current state of the handbook.

Precision issues and epsilons This chapter aims to show readers ways in which geometric computations can go wrong because of floating-point imprecisions, and how to make them go right. In a crucial part, we present a model that can provide guarantees on absolute error for some operations, based on constraints of magnitude and dimensionality. We believe this model will be useful in practice for both contestants and problem solvers to prove the correctness of their solutions.

Basics of 2D geometry This chapter provides a complete view of the basic operations of computational geometry in 2D, exploring topics such as segment intersection, tangents to two circles, areas, winding number, and various manipulations on lines. It builds all of this from a blank slate, starting with vectors and dot/cross product and building up from there, providing intuitive explanations and easy implementations all along the way.

3D geometry This chapter takes much the same approach as the its 2D equivalent, building up from the dot and cross products and showing how to do many operations on planes, lines, polyhedrons, as well as a reasonable introduction to spherical geometry, always trying to foster an intuitive understanding in the reader. To our knowledge, this is one of the first references that provide a comprehensive overview of the foundations of 3D computational geometry in such a practical manner with accompanying code snippets.

2.3 Publication and future plans

The handbook is currently hosted on the author's personal website at `https://vlecomte.github.io/cp-geo.pdf`, and has already been shared on the competitive programming community Codeforces, where it has received an overwhelmingly positive reception.

While we are already proud of what has been accomplished, it is far from done, and we will continue to expand it and make it freely available under a CC-BY-SA license. We are planning to make it a complete reference on all topics in computational geometry that are relevant in a competitive context. Here are the planned next steps.

Sweep algorithms An overview of famous sweep-line and sweep-angle algorithms, including algorithms for convex hull, the segment intersection problem, closest pair problem, etc.

Applications of convex hulls An exposition of techniques linked to convex hulls, such as the $O(\log n)$ point-in-polygon query, an $O(\log n)$ way to cut a convex polygon by a line, $O(\log n)$ point-to-polygon distance, rotating calipers, etc.

Practice problems We are planning to link to collect practice problems for each section whenever possible, so that readers can directly put what they learn into practice as they are reading the book.

Chapter 3

Organizing a geometry contest

In this chapter we present the content and results of the experimental geometry contest that we organized.

3.1 Introduction

The goal of organizing this contest was threefold:

- First, to gauge how prepared the competitive programming public is to problems which involve notions or techniques that are not often seen in regular contests.
- Secondly, to bring those notions and techniques into the light in the hope that they would be more easily used in later contests in order to widen the scope of geometry in programming contests.
- Thirdly, to study the reception of those new types of problems by the participants: would they appreciate their originality or be discouraged by the work needed to solve them?

In the rest of this chapter, we present the general process of contest preparation, present the problems that were prepared for the contest, give their full problem statements and solutions, and draw our conclusions from the organization and the results of the contest.

3.2 Introduction to contest preparation

Newcomers to problem setting are almost always surprised by the quantity of work that goes into preparing a contest task, when they do it for the first time. Although finding a good problem idea and the intended solution is the most important part of the process, it's only a small part of the preparation time. Here we present the typical steps that a problem setter has to go through to use their task in a contest.

Finding the idea Of course, the first step is to imagine the problem. There are two main schools for creating problems: some have a clear idea from the start of what algorithm they would like the intended solution to use and build the problem and context around that, while some start from everyday situations or interesting question and from there figure out if they're solvable and interesting.

Fixing the details and constraints Often, a single idea can lead to several possible variants that can be reached by adding or removing elements from the problem, or adjusting the input limits to accept slower solutions or not. It is the work of the problem setter to determine which version will be the most interesting and fun to solve for the contest that is targeted. This step is particularly crucial when the contest awards partial marks for partial or suboptimal solutions. An overview of methods to find problem ideas and refining them is given in [1].

Writing the problem statement This is usually not the most time-consuming step, especially if the problem is described in a concise way (which is in general appreciated by the participants). But creating figures to illustrate the problem and the sample inputs/outputs can take some time. If the statements are available in multiple languages, then translation and keeping the different versions synchronized requires some work as well.

Implementing correct and incorrect solutions In order to verify the strength of the test data, we need both correct and intentionally incorrect solutions (because they give the wrong answer, are too slow, crash, use too much memory, etc.). It is often useful to have several people implement the correct solution, and to make many solutions based on the correct solution but with very small mistakes. It is typical that quite a few solutions that were intended to be correct turn out to be incorrect as more tests are added.

Generating the test data This is usually the most time-consuming step. As a general rule, generating input data that reach all the desired limits and make all incorrect solutions fail is a tedious task, and the algorithmic challenges that arise when generating the test data often turn out to be more challenging than the actual task. The input is generated by an independent script, and besides tests described imperatively, a reasonable use of randomness can be useful to make incorrect approaches produce wrong answers. The output is typically generated using the official solution.

Validating the test data Because generating the test data is a challenging task, problem setters often make mistakes, and some inputs end up not respecting the constraints given in the problem statement. So an important step is to make another script that will read the input data and verify that it has all the properties it should have. To avoid repeated mistakes, this is sometimes done by a person other than the person who generated the test data. Depending on the problem, this step can range from trivial to very hard (tasks "Pen spinning accident" and "Gold paint optimization" in this contest are good examples).

Implementing output checkers (optional) In some tasks, there is a unique answer, so the output can be compared to the official output by some `diff` (sometimes ignoring whitespace). But in others, there are multiple possible answers, or the answers are floating-point value which are accepted with some tolerance. Those cases require a checker program, which given the input, the official output, and the contestant's output will determine whether the contestant's output should be accepted.

Implementing interactors (optional) Some cases are based on an interaction between the contestant's program and the judging system, which requires a programs that will handle this interaction and evaluate the program.

Testing that solutions get the right verdict Once all the previous steps have been completed, it is time to verify that all solutions are given the expected verdicts and the correct number of points. Besides, to make sure that this will still hold if the implementation is slightly different or in a different languages, margins should be taken. For example, if the time limit is T seconds, all correct solutions should pass in less than T/k seconds, and all slow solutions should take at least kT seconds on some tests, for some $k \geq 2$. Also, solutions that give wrong answers should do so on more than 1 test. Reaching those standards is not always easy, and often requires increasing the input constraints and the time limits, at the expense of a higher load on the judging system during the contest. This testing step can be made easier by tools that automate this process to various degrees.

Beta-testing the tasks When the task is in the late stages of preparation, they are usually sent out to several people for testing and comments. This phase is very useful to determine the difficulty of the task, improve some aspects of it, and strengthen the test data by testing it against more incorrect solutions. Sometimes, a task has to be dropped because an unexpected simpler solution appears and reduces the interest of the problem. Another reason to drop a problem is if many heuristic solutions happen to solve the problem correctly in the average case, and it is too hard to increase the test data to make all of them fail (as well as all those which you might not have thought of). We would like to warmly thank Nicolas Radu who accepted to beta-test the present contest and provided crucial feedback.

Setting up and announcing the contest The tasks should be set up and tested on the judging platform that will be used during the contest. In case this is an online contest, it should be announced and advertized in advance.

Monitoring the contest Despite all the precautions taken while preparing the tasks, it often happens that some unexpected solutions get more or less points than they should. In this unfortunate situation, the authors have to figure out if the issue can be fixed and either add more test cases or change the constraints as soon as possible, then rejudge all solutions in hope that this will solve the problem. Of course, this requires monitoring the solutions that are submitted and understanding them. An other important activity during the contest is to answer the requests for clarification that are sent about the problem statements.

Writing the editorials Once the contest is over, the problem authors generally publish *editorials*, that is, descriptions of the intended solution for each problem. This allows participants to find out how to solve the problems that they couldn't solve during the contest. Most platforms allow users to submit solutions unofficially after the contest for practice, an activity called *upsolving*.

3.3 Presentation of the problems

Here we present the problems that were prepared for the contest, and the new notions or techniques they included.

Problem A: Sledding down the mountain You are given a polygonal mesh and a starting point, and have to compute the total area that can be reached by travelling along the mesh with nonincreasing z. This is essentially a reachability graph problem, but on an unusual setting: a polygonal mesh in space. The aim of this problem was to test if contestants can see through the setting and discover the graph problem that lies behind, and also to be a model for future problems using the same object.

Problem B: Pen spinning accident You are given two rotating lines in space and asked whether those two lines will eventually cross, and if so when. To come up with the intended solution, there were two difficulties: knowing some criterion that determines whether two lines are skew, and manipulating a mathematical expression with enough skill to make the implementation easy. It is our hope that the handbook has solved the first difficulty from now on.

Problem C: Polar map You are given a country on Earth determined by successive points on its border, linked consecutively by great-circle paths, and are asked to determine its area in some map projection. This problem was meant to showcase a technique to compute the area of a region based on a piecewise parametric representation of its boundary. Although in this case, the area computation is the whole problem, in the future, this technique is intended to be used as a subroutine in a larger problem.

Problem D: Gold paint optimization This problem introduces solid angles, a notion probably new to most contestants, and asks them to maximize a solid angle given some limited resource. Although it is challenging in its own right, the aim of this problem was to give publicity to this new geometric notion that could be used in future tasks.

3.4 Problem statements

Hereafter we include the problem statements of the five tasks, in their original form. The judge solutions, input generators, input validators and output checkers are available as an attachment to this thesis and also at the following url: `https://vlecomte.github.io/set/geo18/geo18.zip`.

Sledding down the mountain

Writer: Victor Lecomte
Time limit: 12 s Memory limit: 256 MB

You are on vacation in the mountains, and are going on a crazy sled ride through hilly terrain. You can easily go down a slope and you can also travel while staying at constant height, but you can't go up, that is, your z-coordinate is nonincreasing.

The surface of the mountain is formed of triangle faces linked together at their edges. There are no vertical slopes and no overhangs. You should always stay on those triangle faces, but you are allowed to move on their edges and to move from one triangle to another *if they share an edge.*

Given a starting point on the mountain, what is the total surface occupied by the points that you could reach on the mountain without going up?

Input

The first line of the input contains one integer n, the number of triangles in the mountain. Then follow n groups of 3 lines. The i^{th} group contains three 3D points $P_{i,1}, P_{i,2}, P_{i,3}$, the vertices of the triangle, each one one line. Each vertex is described by 3 space-separated integers $x_{i,j}, y_{i,j}, z_{i,j}$, its coordinates. The z-coordinate represents the height, while the x- and y-coordinates represent the position of the point on a map of the mountain.

Your starting point is $P_{1,1}$.

Output

Output a floating-point value on a single line: the total area of the mountain that you can reach on your sled without going up. The answer should be accurate to an absolute or relative error of 10^{-6}.

Limits

- $1 \leq n \leq 10^5$;
- $-10^5 \leq x_{i,j}, y_{i,j}, z_{i,j} \leq 10^5$;
- the vertices of a triangle are given in counter-clockwise order, that is, the z-coordinate of $\overrightarrow{P_{i,1}P_{i,2}} \times \overrightarrow{P_{i,1}P_{i,3}}$ is positive;
- the mountain is connected, that is, you can go from any triangle to any other triangle by passing through common edges;

- if two triangles touch, they share either an entire edge, or one vertex;
- no point in a triangle is directly above or below another triangle.

Example 1

stdin

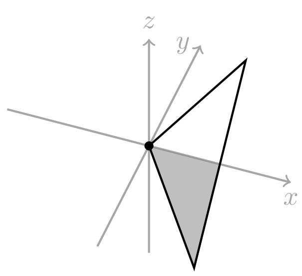
```
1
0 0 0
1 -1 -1
1 1 1
```

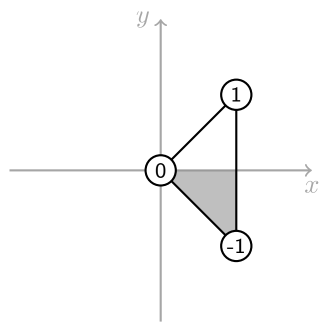
stdout

```
0.707106781187
```

This mountain has only one triangle. You start at point $(0,0,0)$ and you can ride your sled to the whole lower half of the triangle, that is, you can ride to any point on the triangle of vertices $(0,0,0)$, $(1,-1,-1)$ and $(1,0,0)$ (in gray). This is a right triangle with sides 1 and $\sqrt{2}$, so its area is $\frac{\sqrt{2}}{2}$.

Below on the left is a 3D view of the mountain, and on the right a view from above, with the height marked at every vertex. The starting point is marked with a dot in the 3D view, and the reachable region is highlighted.



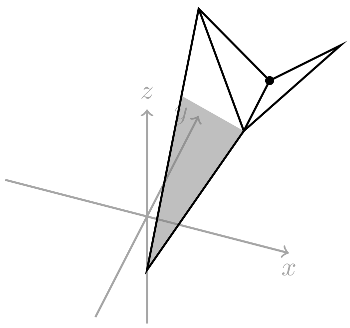


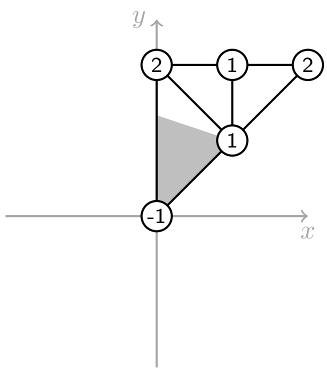
Example 2

stdin
```
3
1 2 1
0 2 2
1 1 1
1 1 1
2 2 2
1 2 1
0 0 -1
1 1 1
0 2 2
```

stdout
```
1.247219128925
```

This mountain has three triangles. Triangle 1 has the edge from $(1, 2, 1)$ to $(1, 1, 1)$ in common with triangle 2, and the edge from $(1, 1, 1)$ to $(0, 2, 2)$ in common with triangle 3. You start at point $(1, 2, 1)$, at height 1. You cannot climb up triangles 1 or 2, so the area that you cover on them is zero. But you can travel on the edge from $(1, 2, 1)$ to $(1, 1, 1)$ at constant height to get to triangle 3. From there, you can reach $\frac{2}{3}$ of the area of triangle 3, which gives the answer $\frac{2}{3} \times \frac{\sqrt{14}}{2}$.

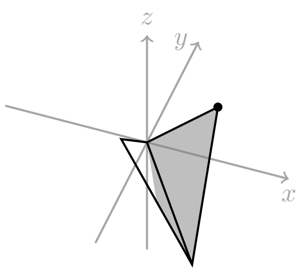


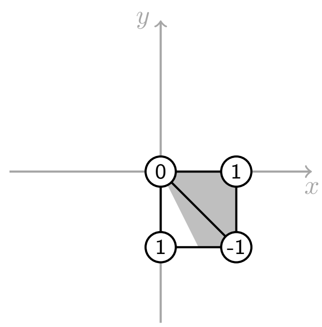


Example 3

stdin
```
2
1 0 1
0 0 0
1 -1 -1
1 -1 -1
0 0 0
0 -1 1
```

stdout
```
1.837117307087
```



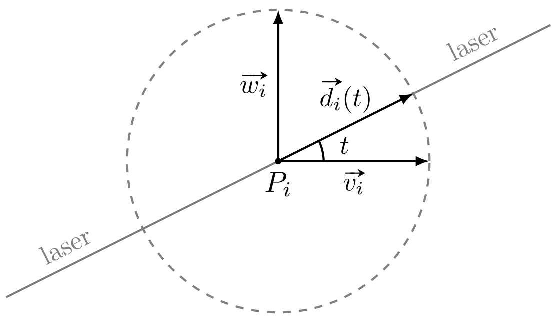


Pen spinning accident

Writer: Victor Lecomte
Time limit: 1 s Memory limit: 256 MB

You and your friend are huge pen spinning experts, and since it was getting too easy you decided to take it to the next level by mounting deadly lasers on both ends of your pens. While you don't worry much about accidentally hurting people, there is one safety guideline that you must respect: much like in a famous '80s movie, you should never cross the laser beams, because it would be... bad. So you would like to know when the beams are going to cross, if ever, so that you can safely stop before it happens.

In the rest of the statement, you will be called "person 1" and your friend "person 2". Person i is currently spinning their pen around point P_i in space at a constant rate of 1 radian per second. At time t their pen is pointing in direction $\vec{d_i}(t) = (\cos t)\vec{v_i} + (\sin t)\vec{w_i}$, that is, the laser beams on both ends run along the line containing points P_i and $P_i + \vec{d_i}(t)$. You can consider that the length of the pen is negligibly small, so that the whole line is covered by the laser beams.



Input

The output is made of 2 groups of 3 lines, separated by a blank line. The ith group contains the values $P_i, \vec{v_i}, \vec{w_i}$ on separate lines. Each of them consists of three space-separated integers: their x-, y- and z-coordinates.

Output

The output is a single line. Print the earliest time $t > 0$ at which the beams cross, if it exists. Otherwise, print the word `never`. The time should be accurate to an absolute error of 10^{-7}.

Limits

- all coordinates are integers in $[-100, 100]$;
- $P_1 \neq P_2$ (you are at distinct locations);
- $\vec{v_i} \cdot \vec{w_i} = 0$ and $\|\vec{v_i}\| = \|\vec{w_i}\| \neq 0$ (so direction $\vec{d_i}(t)$ rotates on a circle);
- $\vec{d_1}(t) \times \vec{d_2}(t) \neq \vec{0}$ for all t (the beams are never parallel);
- the laser beams are not touching at time 0;
- you will never shoot each other with your lasers, that is, P_1 is not on the line containing P_2 and $P_2 + \vec{d_2}(t)$ for any t, and vice versa.

Example 1

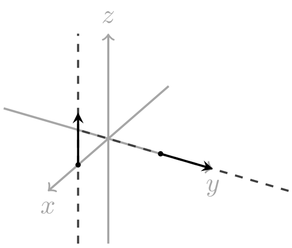
stdin

```
1 0 0
0 0 1
1 0 0

0 1 0
0 1 0
0 0 1
```

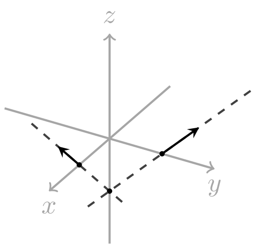
stdout

```
0.785398163397
```

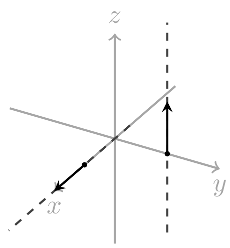
In this example, the starting directions of the pens are $(0, 0, 1)$ and $(0, 1, 0)$, so the lines formed by their lasers are at $x = 1, y = 0$ and $x = 0, z = 0$ respectively. After $\pi/4$ seconds, pen 1 points in direction $\frac{1}{\sqrt{2}}(1, 0, 1)$ while pen 2 points in direction $\frac{1}{\sqrt{2}}(0, 1, 1)$, so that both lines contain point $(0, 0, -1)$. This is the first time they touch.



$t = 0$



$t = \frac{\pi}{4}$



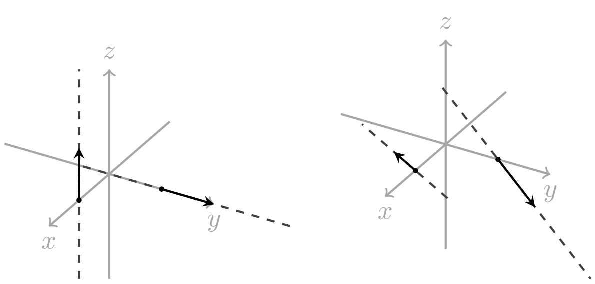
$t = \frac{\pi}{2}$

Example 2

stdin
```
1 0 0
0 0 1
1 0 0

0 1 0
0 1 0
0 0 -1
```

stdout
```
never
```

This example is identical to the first, except that the second pen rotates in the opposite direction.



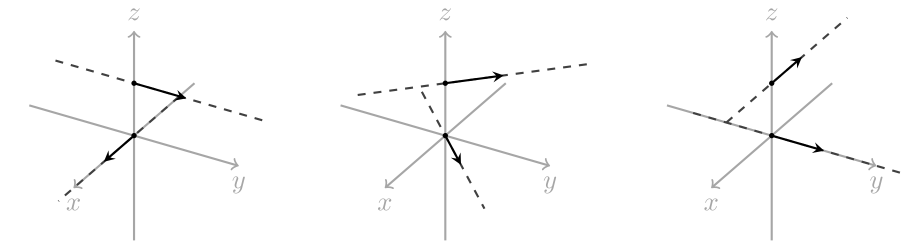
$t = 0$ $\qquad$ $t = \frac{\pi}{4}$ $\qquad$ $t = \frac{\pi}{2}$

Example 3

stdin
```
0 0 0
1 0 0
0 1 0

0 0 1
0 1 0
-1 0 0
```

stdout
```
never
```

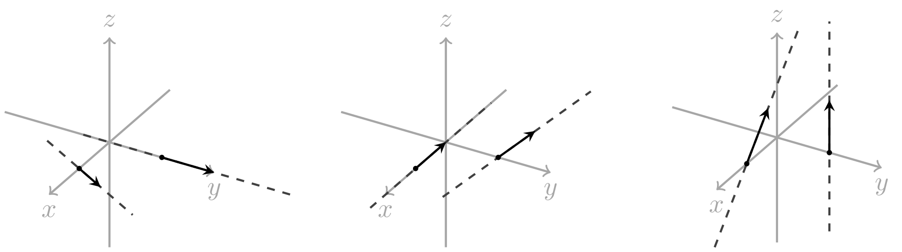
In this example, the lasers of the first pen only reach points for which $z = 0$, while the lasers of the second pen only reach points for which $z = 1$, so they clearly never touch.



$t = 0$ $t = \frac{\pi}{4}$ $t = \frac{\pi}{2}$

Example 4

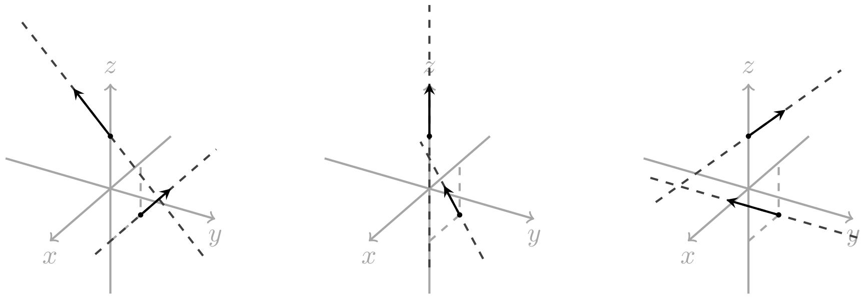
stdin	stdout
1 0 0 -1 0 -1 -1 0 1 0 1 0 0 1 0 0 0 1	0.392699081699



$t = 0$ $t = \frac{\pi}{4}$ $t = \frac{\pi}{2}$

Example 5

stdin	stdout
0 0 1 0 -1 1 0 1 1 -1 0 -1 -1 0 0 0 -1 0	0.511669644114



$t = 0$ $t = \frac{\pi}{4}$ $t = \frac{\pi}{2}$

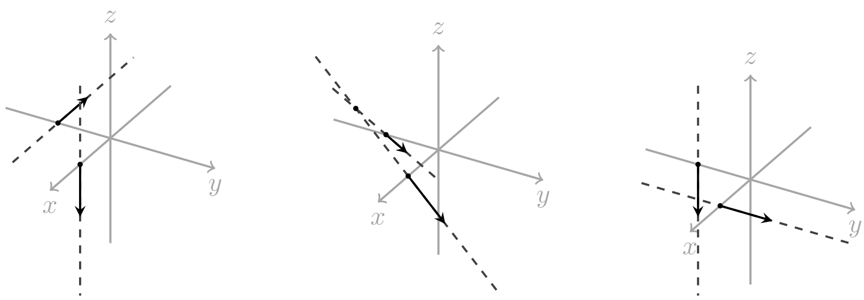
Example 6

stdin

```
0 -1 0
-1 0 0
0 0 -1

1 0 0
0 0 -1
0 1 0
```

stdout

```
0.785398163397
```



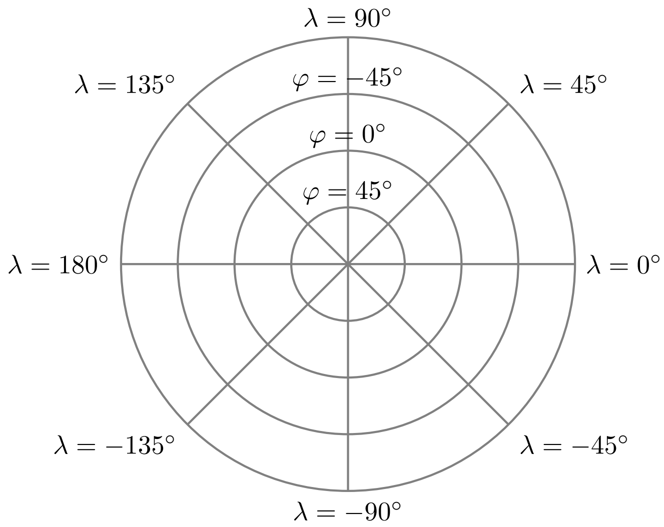
$t = 0$ $t = \frac{\pi}{4}$ $t = \frac{\pi}{2}$

Polar map

Writer: Victor Lecomte
Time limit: 8 s   Memory limit: 256 MB

You live on the North Pole, and are frustrated that your house is located on the edge (or even outside) of most world maps. So you have decided to create your own map, with two criteria: the North Pole should be in the center, and distances from it should be conserved. The *azimuthal equidistant projection*[1] is perfect for this.

In the azimuthal equidistant projection, the map is a circle of radius 1, and a point of latitude φ and longitude λ (given in degrees) will be represented at position $(d \cos \lambda, d \sin \lambda)$, where $d = \frac{90^\circ - \varphi}{180^\circ}$.



Now, consider a country, whose border is determined by some border points, where each border point is linked to the next by the shortest path along the Earth's surface (the *great-circle distance*[2]). What is the area occupied by that country on your map?

Input

The first line of the input contains one integer n, the number of border points of the country. Then follow n lines, the i^{th} of which contains two integers φ_i and λ_i, the latitude and longitude of the border point, in degrees. The points are given in counter-clockwise order, that is, when going from point i to point $i+1$, the inside of the country is on the left side.

[1] See `https://en.wikipedia.org/wiki/Azimuthal_equidistant_projection` for a more in-depth introduction.

[2] See `https://en.wikipedia.org/wiki/Great-circle_distance` for a more in-depth introduction.

Output

Output a floating-point value on a single line: the area occupied by the country on your map. The answer should be accurate to an absolute error of 10^{-7}.

Limits

- $3 \le n \le 10^3$;
- $-85 \le \varphi_i \le 85$, $-180 < \lambda_i \le 180$;
- the Earth is a perfect sphere (sorry, flat earthers);
- the country does not self-intersect;
- no two consecutive points on the border of the country are exactly opposite each other;
- the country contains neither the North Pole, nor the South Pole;
- no border of the country will come closer than 2×10^{-4} to the center of the map (North Pole) or to the border of the map (South Pole).

Example 1

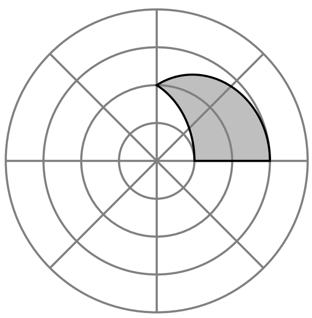
stdin

```
3
45 0
-45 0
0 90
```

stdout

```
0.269064434316
```

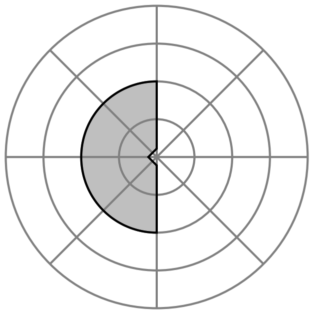
The border points of the country are located at $(0.25, 0)$, $(0.75, 0)$ and $(0, 0.5)$ on the map. However, the area is not $1/8$ because the borders of the country are not straight lines on the map.



Example 2

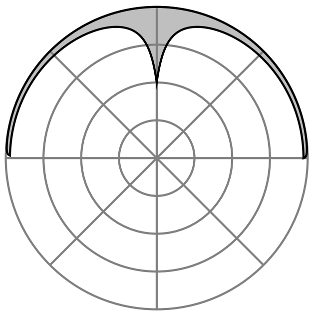
stdin	stdout
6 0 90 0 180 0 -90 80 -90 80 180 80 90	0.389591701512

Between the first three points, the border runs along the equator, which is a circular arc on the map. The area is a bit less than $\pi/8$.



Example 3

stdin	stdout
3 -85 0 -85 179 0 90	0.207400294880



Gold paint optimization

Writer: Victor Lecomte
Time limit: 10 s   Memory limit: 256 MB

You are working on a modern art piece consisting of n cubes of side $1\,\mathrm{m}$ in space. The cubes are currently unpainted, and that's a bit boring, so you have decided to buy some gold paint to decorate them. Since the gold paint is made of real gold, it's very expensive, so you only have l liters. One liter can paint a surface of $1\,\mathrm{m}^2$. You are allowed to distribute the paint completely as you wish, painting only some of the cubes, only some of the faces of a given cube, and even only part of a given face.

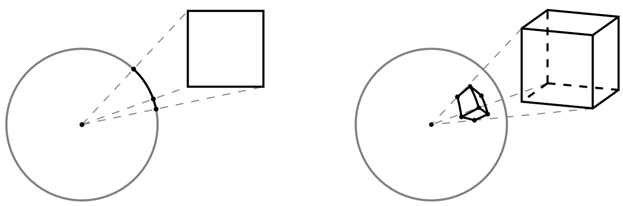
The visitors will come to see your work at point $(0,0,0)$, so you would like to optimize your use of the paint so that the part of their field of view that is covered by gold paint is as large as possible, that is, the total *solid angle* subtended at $(0,0,0)$ by the visible painted parts of the cubes is maximized.

Paint does not count if it is placed on faces that are not visible from the origin because they are on the wrong side of the cube.

To achieve this goal, how much paint should you use on each of the cubes?

Solid angle

Just like the planar angle subtended by an object is the length of the object once it is projected onto a unit circle around the point, the solid angle subtended by an object is the area of the object once it is projected onto a unit sphere around the point.
See `https://en.wikipedia.org/wiki/Solid_angle` for a complete definition.



planar angle   solid angle

Input

The first line of the input contains two integers n and l, the number of cubes and the quantity of paint in liters. Then follow n lines, the i^{th} of which contains three integers x_i, y_i, z_i, meaning that there is an axis-aligned unit cube at $[x_i, x_i+1] \times [y_i, y_i+1] \times [z_i, z_i+1]$.

Output

The output consists of n lines, the i^{th} of which contains a floating-point value l_i, the number of liters of paint that should be used on cube i. The numbers should be accurate up to a relative or absolute error of 10^{-4}. It can be proven that the solution is unique under the constraints of this problem.

Limits

- $1 \leq n \leq 10^4$;
- $0 \leq l < 3n$;
- $0 \leq x_i, y_i, z_i < 100$, which means that the faces visible from the origin are in planes $x = x_i$ if $x_i \neq 0$, $y = y_i$ if $y_i \neq 0$ and $z = z_i$ if $z_i \neq 0$;
- no cube touches the origin;
- no cube hides any part of another cube, that is, no ray from the origin touches more than one cube;
- you don't have enough paint to cover all visible faces, that is, if you had more paint, you could strictly increase the solid angle subtended at the origin by the visible painted parts.

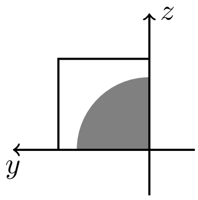
Example 1

stdin
```
2 1
2 0 0
0 2 0
```

stdout
```
0.50000000
0.50000000
```

You have 1 liter to paint two cubes at $[2, 3] \times [0, 1] \times [0, 1]$ and $[0, 1] \times [2, 3] \times [0, 1]$. They are at equal distances from $(0, 0, 0)$ and have only one visible face from there, at $x = 2$ and $y = 2$, respectively. It is optimal to use half of the paint on each face.

Here is what the visible face of the first cube looks like from the origin, with the painted part highlighted:



Example 2

stdin	stdout
2 2 2 0 0 10 10 10	1.00000000 1.00000000

You have 2 liters to paint two cubes at $[2, 3] \times [0, 1] \times [0, 1]$ (one visible face) and $[10, 11] \times [10, 11] \times [10, 11]$ (three visible faces). It is optimal to paint the visible face of the first cube completely, then to use the remaining liter to paint part of the three faces of the second cube.

Example 3

stdin	stdout
2 2 2 0 0 0 1 1	0.98985432 1.01014568

3.5 Editorial

This section contains the editorial of the contest, that is, a description of the main ideas of the intended solution, for each of the four problems that were used in the contest. It was published on the contest site right after the contest.

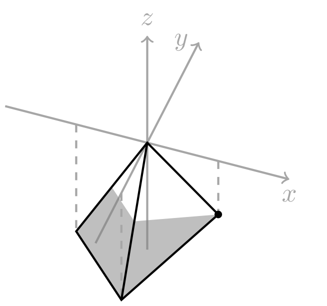
3.5.1 Sledding down the mountain

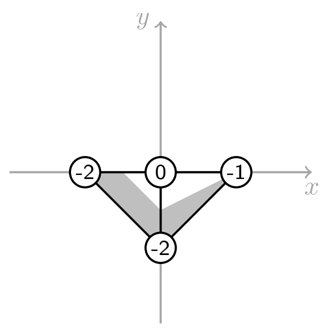
In this problem I wanted to experiment with graph-like problems on a polyhedral mesh. My aim was to show that the added dimension of height could make interesting ideas possible.

Idea

The first observation to make is that if we can reach a triangle at height h, then we can reach all points of the triangle at height $\leq h$. So for each triangle i, we want to find the highest height h_i at which it can reached from $P_{1,1}$.

A first idea would be to build a graph from the vertices of the triangles and determine for each of them whether it is reachable, with some kind of flood fill. However, this is insufficient, as sometimes the critical height h_i is not the height of any vertex of the triangle, like in the left triangle below.





A better idea is to build a graph on the triangles of mountain, where two triangles are linked if they share an edge. Let's figure how to do the relaxations between two triangles. Suppose triangles i and j share an edge, such that the lowest point of the edge has height h_{lo} and the highest point of the edge has height h_{hi}. If triangle i can be reached at h_i, what height can we reach in j using that edge?

There are three cases:

- if $h_i < h_{\text{lo}}$, triangle j cannot be reached from triangle i;

- if $h_{\text{lo}} \leq h_i \leq h_{\text{hi}}$, it can be reached with a height of h_i;
- if $h_i > h_{\text{hi}}$, it can be reached with a height of h_{hi}.

Since in all cases the height that is reached is smaller or equal to h_i, we can use Dijkstra's algorithm to efficiently compute h_i for all i. We maintain the best known height for every triangle, and always select the unvisited triangle that has the highest height.

Cutting the triangles

Now that we found h_i for all triangles i, we need to find the part of the triangle that is under plane $z = h_i$. This is a similar exercise to cutting a polygon by a line in 2D: we need to keep the points for which $z \leq h_i$ and add points whenever a edge goes through the plane. This can be simply implemented this way:

```
// Assume p3 is a 3D point structure with basic operators

p3 planeInter(p3 a, p3 b, int h) {
    return (a*(b.z-h) - b*(a.z-h))/(b.z-a.z);
}

vector<p3> polygonUnder(vector<p3> ps, int h) {
    vector<p3> under;
    for (int i=0, n=ps.size(); i<n; i++) {
        p3 a = ps[i], b = ps[(i+1)%n];

        // Keep points under h
        if (a.z <= h)
            under.push_back(a);

        // Add intersections
        if ((a.z-h)*(b.z-h) < 0) // if a and b on either side
            under.push_back(planeInter(a,b,h));
    }
    return under;
}
```

Finding the area

It turns out finding the area of a polygon in 3D is pretty much the same as in 2D. In 2D, we can find the area of a polygon $P_1 \cdots P_n$ as

$$\frac{|P_1 \times P_2 + P_2 \times P_3 + \cdots + P_n \times P_1|}{2}$$

where $\times$ is the 2D cross product (a scalar). In 3D, if $P_1, \ldots, P_n$ are coplanar, the vector

$$\vec{S} = P_1 \times P_2 + P_2 \times P_3 + \cdots + P_n \times P_1$$

is a vector perpendicular to their plane of polygon $P_1 \cdots P_n$ and whose norm is twice the area of the polygon. So we can similarly compute the area as $\frac{1}{2}\|\vec{S}\|$.

So we just need to use this formula and for every triangle sum up the area of the part that is under h_i.

3.5.2 Pen spinning accident

There were two ways to approach this problem (that I know of), with wildly different difficulties of implementation. The first is interesting in that we transform a 3D problem into a 1D problem. The second shows that good knowledge of geometric primitives can help you simplify a problem and avoid a lot of edge cases.

Approach 1: Position on the intersection line

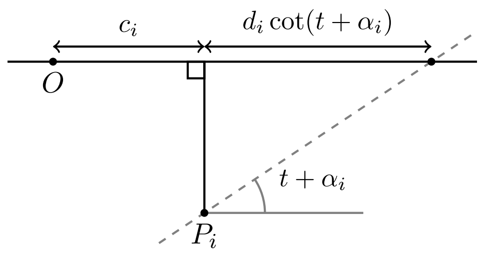
We can observe that when the pens spin around, they each sweep a distinct plane. So if the pens touch at some point, it must happen on the intersection of the planes. If it doesn't exist, the answer is `never`, otherwise because of the constraints of the problem, it must be a line l that touches neither P_1 nor P_2.

Since we're only interested in positions on l, let's set an arbitrary starting point O on l, and figure out how far from O the lasers of the i^{th} pen touch l. We'll call this $f_i(t)$.

We can easily convince ourselves that $f_i(t)$ is of the form

$$f_i(t) = c_i + d_i \cot(t + \alpha_i)$$

where c_i is the position of the projection of P_i on l, d_i is the distance from P_i to l, and α_i depends on the angle between l and $\vec{v_i}$.



Instead of trying to find a root straight away (which would be very tricky), we can use the summation formula:

$$\cot(t+\alpha_i) = \frac{\cot t \cot \alpha_i - 1}{\cot t + \cot \alpha_i}$$

By substituting and eliminating the denominators gives this (rather lengthy) quadratic equation in $\cot t$ which can be used to find t more easily:

$$\begin{aligned} 0 &= (\cot t)^2(c_2 - c_1 + d_2 \cot \alpha_2 - d_1 \cot \alpha_1) \\ &+ (\cot t)[(c_2 - c_1)(\cot \alpha_1 + \cot \alpha_2) + (d_2 - d_1)(\cot \alpha_1 \cot \alpha_2 - 1)] \\ &+ [(c_2 - c_1)\cot \alpha_1 \cot \alpha_2 - (d_2 \cot \alpha_1 - d_1 \cot \alpha_2)] \end{aligned}$$

Some caveats:

- Since the values of c_i, d_i and $\cot \alpha_i$ are most likely imprecise, we need to do an epsilon comparison to test the value of the discriminant and determine the existence of solutions.
- We also need to handle the cases when $\cot \alpha_1$ or $\cot \alpha_2$ don't exist. One trick is to multiply the whole equation by $\sin \alpha_1 \sin \alpha_2$, which replaces $\cot \alpha_i$ with $\cos \alpha_i$ and adds some $\sin \alpha_i$ in the mix.
- Note that we don't care about the times when $\cot t$ doesn't exist, that is, $t = k\pi$, since the lasers are guaranteed not to touch at the start.

Approach 2: Vector magic

Given 4 points A, B, C, D in space, we can define the expression

$$\operatorname{orient}(A,B,C,D) = \left(\overrightarrow{AB} \times \overrightarrow{AC}\right) \cdot \overrightarrow{AD}$$

which has many properties similar to its 2D counterpart

$$\operatorname{orient}(A,B,C) = \overrightarrow{AB} \times \overrightarrow{AC}$$

For example, $\operatorname{orient}(A,B,C,D) = 0$ iff A, B, C, D are coplanar, its sign determines on which side of plane ABC point D lies, and its norm is 6 times the volume of tetrahedron $ABCD$. The property we are interested in here is that it is zero iff lines AB and CD are either parallel or intersecting (remember that in space, lines can be neither). Since in our case the lasers are never parallel, we can use this to determine easily whether the lines intersect.

We can reshuffle it this way:

$$\operatorname{orient}(A,B,C,D) = \left(\overrightarrow{AB} \times \overrightarrow{AD}\right) \cdot \overrightarrow{AC} = \left(\overrightarrow{AB} \times \overrightarrow{CD}\right) \cdot \overrightarrow{AC}$$

so that in our case, the lasers touch iff

$$\text{orient}\left(P_1, P_1 + \overrightarrow{d_1}(t), P_2, P_2 + \overrightarrow{d_2}\right) = \left(\overrightarrow{d_1}(t) \times \overrightarrow{d_2}(t)\right) \cdot \overrightarrow{P_1P_2} = 0$$

If we plug in the definition of $\overrightarrow{d_i}(t)$, we obtain

$$\begin{array}{rcll} 0 & = & \cos^2 t & (\overrightarrow{v_1} \times \overrightarrow{v_2}) \cdot \overrightarrow{P_1P_1} \\ & + & \cos t \sin t & (\overrightarrow{v_1} \times \overrightarrow{w_2} + \overrightarrow{w_1} \times \overrightarrow{v_2}) \cdot \overrightarrow{P_1P_2} \\ & + & \sin^2 t & (\overrightarrow{w_1} \times \overrightarrow{w_2}) \cdot \overrightarrow{P_1P_2} \end{array}$$

If we divide the equation by $\sin^2 t$, we obtain (again) a quadratic equation in $\cot t$, but this time with no special cases, and all computations can be done with integers.

3.5.3 Polar map

In this problem, we are interested in finding the area of objects of unusual shapes. So first we need to find mathematical tools that can help us do that. The objective was mostly to show this way to compute areas.

Area surrounded by a curve

Given a curve $\mathcal{C}$, how can we find the area it encloses? One way is to compute the following integral:

$$\frac{1}{2}\int_{\mathcal{C}} x\,dy - y\,dx$$

This can be seen as an infinitesimal version of the shoelace formula, and computes the surface sweeped by the segment from the origin O to a moving point on $\mathcal{C}$.

If we know that the curve never goes through O, then we can consider polar coordinates $(r\cos\theta, r\sin\theta)$ and the integral becomes

$$\frac{1}{2}\int_{\mathcal{C}} r^2\,d\theta$$

Indeed, $\frac{r^2}{2}\,d\theta$ is the area of circular sector of radius r and angle $d\theta$.

This means that if we describe $\mathcal{C}$ in a parametric way, that is, if we find functions $r(t)$ and $\theta(t)$ that describe the coordinates of points on $\mathcal{C}$ as parameter t increases in $[a, b]$, then we can compute the area as

$$\frac{1}{2}\int_a^b r^2(t)\,\theta'(t)\,dt$$

where $\theta'(t)$ is the derivative of $\theta(t)$.

For example, let's say we want to find the area of a circle of radius 1.[1]

[1] Groundbreaking, right?

We can define the points of the circle with

$$\begin{cases} r(t) & = & 1 \\ \theta(t) & = & t \end{cases}$$

for $t \in [0, 2\pi]$.

Then we can find

$$\frac{1}{2}\int_{\mathcal{C}} r^2\, d\theta = \frac{1}{2}\int_0^{2\pi} r^2(t)\,\theta'(t)\,dt = \frac{1}{2}\int_0^{2\pi} 1^2\,1\,dt = \pi$$

Note that the integral can be computed in several parts to handle different parametrizations: for example, this allows us to easily compute the area of shapes formed by line segments and circular arcs.

Parametrization

Now that we have a way to compute the area of any parametric curve, let's try to find a parametrization for the country borders on our map.

Let's say the Earth has radius 1 and we want to find the path between two consecutive points A and B on the border of the country. First, we observe that the "straight line" between A and B actually gives the correct direction: if we take point $P(t) = A + t\overrightarrow{AB}$ ($t \in [0, 1]$) and scale it so that it has a norm of 1, then it will lie on the great circle containing A and B. And since the position on our map only depends on the latitude φ and longitude λ, not on the altitude, we will use this point $P(t)$ directly.

Let's assume that the axis $z = 0$ is the line going between the poles. First, we find the longitude and latitude of $P(t)$:

$$\varphi(t) = \operatorname{atan2}\left(P_z(t), \sqrt{P_x^2(t) + P_y^2(t)}\right)$$
$$\lambda(t) = \operatorname{atan2}\left(P_y(t), P_x(t)\right)$$

This gives the following polar coordinates on the map:

$$r(t) = \frac{\pi/2 - \varphi(t)}{\pi}$$
$$\theta(t) = \lambda(t)$$

We need to find the derivative of $\theta(t)$. Here we are using formulas for the partial derivatives of atan2, see `https://en.wikipedia.org/wiki/Atan2#`

`Derivative` for more info.

$$
\begin{aligned}
\theta'(t) = \lambda'(t) &= \left(\operatorname{atan2}\left(P_y(t), P_x(t)\right)\right)' \\
&= \frac{P_y'(t)P_x(t) - P_x'(t)P_y(t)}{P_x^2(t) + P_y^2(t)} \\
&= \frac{(B_y - A_y)P_x(t) - (B_x - A_x)P_y(t)}{P_x^2(t) + P_y^2(t)} \\
&= \frac{A_xB_y - A_yB_x}{P_x^2(t) + P_y^2(t)}
\end{aligned}
$$

Now we can find the part of the integral for this part of the border as

$$\frac{1}{2}\int_0^1 r^2(t)\,\theta'(t)\,dt$$

Integration

We didn't manage to find an antiderivative for the given function to integrate. However, in general numerical integration works very well, so you can just use that.

For the best precision/time ratio, we advise using something adaptative like adaptative Simpson's method (see `https://en.wikipedia.org/wiki/Adaptive_Simpson%27s_method`). But time limits were lenient enough that regular Simpson's rule and even trapezoidal or midpoint rule worked as well, as long as the step is small enough to detect the brusque changes when the border passes close to the South Pole as in the third sample.

3.5.4 Gold paint optimization

Some of the difficulty of this problem was to get used to the notion of solid angle. But actually, an intuitive understanding of it was enough to solve the problem, since the solution doesn't require actually computing a solid angle at any point.

Idea

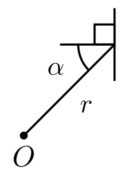
The main insight was thinking in terms of return on investment: given a location, if I paint a small area there, by how much does it increase the solid angle. Thus we consider the ratio

$$\rho = \frac{\text{solid angle added}}{\text{paint used}}$$

It is clear that surfaces with higher ratios should be painted before surfaces with lower ratios. So we perform a binary search on this ratio ρ, computing the total area of parts that have a ratio $\geq \rho$. If this total area is smaller than l, we should decrease ρ, otherwise we should increase it. The distribution of paint after convergence is the optimal distribution of paint.

Computing the ratio

Intuitively, the ratio is inversely proportional to the square of the distance from $O = (0, 0, 0)$, and it will be lower if the surface is seen at an angle. More precisely, if the surface is seen from distance r and the angle between the direction of observation and the normal of the surface is α, then the ratio is $\rho = \cos\alpha/r^2$.

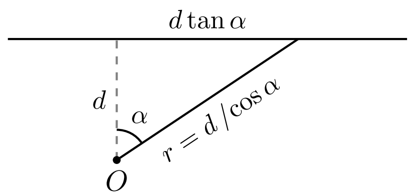


This can be seen from a quick drawing or taken from Wikipedia:

$$\Omega = \int_S \frac{\hat{r} \cdot \hat{n} \, d\Sigma}{r^2}$$

where Ω is the solid angle, S the surface observed, $\hat{r}$ the direction of observation, $\hat{n}$ the normal of the surface and $d\Sigma$ is an infinitesimal area.

Now, given a ratio ρ and a cube face, how do we find the part of the face that has ratio at least ρ? Suppose without loss of generality that the square is on plane $x = d$.



Clearly, on that plane, the closer to $O = (0, 0, 0)$, the better the ratio. In particular, if the observed point is at angle α from the perpendicular, then $r = d/\cos\alpha$, so the ratio is

$$\rho = \frac{\cos\alpha}{r^2} = \frac{\cos^3\alpha}{d^2}$$

and we can find α as

$$\alpha = \arccos\left(\sqrt[3]{\rho d^2}\right)$$

The time limits were lenient enough to allow solutions to find this value by some binary search instead.

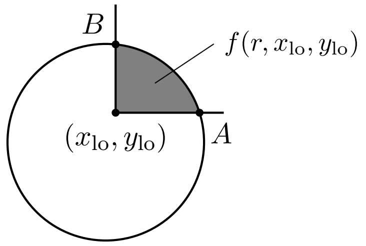
Thus, for the value of α we fond, all points on the plane $x = d$ that are within a disk of radius $d \tan \alpha$ around $(d, 0, 0)$ will have ratio $\geq \rho$. So the relevant points on the cube face are the intersection of that disk and the square of the cube face (a 2D problem).

Circle-square intersection

So the last thing we need to figure out is how to compute the area of the intersection of a disk and a square. This is feasible but tedious in the general case.

Here we are helped by the fact that the square will always be fully within one of the quadrants of the disk. So we can find it easily by inclusion-exclusion. Let's say we have a function $f(r, x_{\text{lo}}, y_{\text{lo}})$ that finds the intersection between the circle of radius r and center $(0, 0)$, and the quadrant $x \geq x_{\text{lo}}, y \geq y_{\text{lo}}$, for $x_{\text{lo}}, y_{\text{lo}} \geq 0$. Then the area of intersection with square $[x, x + 1] \times [y, y + 1]$ can be found as

$$f(r, x, y) - f(r, x + 1, y) - f(r, x, y + 1) + f(r, x + 1, y + 1)$$



Function f can be easily implemented this way, starting by computing intersection points A and B if they exist:

```
double quadrantDisk(double r, double x, double y) {
    if (x*x + y*y >= r*r)
        return 0;
    pt p(x,y), a(sqrt(r*r-y*y), y), b(x, sqrt(r*r-x*x));
    return (cross(p,a) // triangle OPA
        + r*r*asin(min(cross(a,b)/r/r, 1.0)) // circular sector OAB
        + cross(b,p) // triangle OBP
        )/2;
}
```

3.6 Results and conclusion

The contest was organized on the contest platform Codeforces, and is available at the following address: `http://codeforces.com/gym/101793`. It took place on May 6, lasting 4 hours. It seemed likely that no participant would solve all problems, given their difficulty and the knowledge required to solve them.

Over 30 people registered for the contest, but only 8 submitted any code. It is expected that those participants who did not submit either found the problems too difficult and gave up, or had issues while implementing and never reached a working solution.

The problems were sorted according to their estimated difficulty (we presented them in that same order). Two participants solved the first two problems successfully, while the last two problems were untouched. While this is slightly disappointing, it was not completely unexpected.

In a feedback form shared after the contest,[2] we asked contestants

- their perceived difficulty level of the problems, both while solving them and after reading the editorials;
- how much they liked/disliked the problems;
- where they felt the difficulty mainly lay (finding the right idea, knowing the right theory, etc.).

Although the number of responses (8) is rather small, it allows us to draw some general tendencies.

Our expectation was that contestants would mostly realize the problems were easier than they thought once they had read the editorials, because knowing the theory would make the problem more manageable. The results seem to confirm this since out of 26 difficulty evaluations, in 7 cases contestants thought the problem was easier after reading the editorial, while only in 2 cases they thought it was harder, and in 17 cases their appreciation didn't change.

The difficulty appreciation also seemed to mostly match the expected difficulty, with problem A being most often called "easy", problem B "medium", problem C "hard" and problem D "impossible".

It seems that the contestants mostly enjoyed the problems, as all problems were either "liked" or "loved" by more than 70 % of respondants.

Finally, half of the respondants thought a large part of the difficulty lay in knowing the right theory, as expected, and another half thought it lay in finding the right idea, which is typical of a well-made regular contest. Another popular response was correctly handling edge cases.

[2] The full results of which are given as an attachment.

In conclusion, even though we had hoped to have slightly more participants and submissions, we think that this contest was a success. We hope that those problems were interesting and that they may pave the way for more varied geometry problems in the future, and we hope that our handbook will serve as a solid reference to supplement the contestants' knowledge in the theory points that were not known to all.

Chapter 4

Conclusion

In an effort to make geometry problems more accessible to everyone in programming contests, we have produced a practice-oriented handbook covering the basics of both 2D and 3D geometry, as well as an introduction to the issues that can arise from the use of floating-point data types and how to overcome them. All notions and algorithms are explained from the ground up using intuitive explanations, and accompanied by short and cohesive code snippets and by a large amount of illustrative figures. Notable contributions are in presenting a model of error propagation based on magnitude conditions and dimensionality of values, and in our chapter about the foundations of 3D geometry, a topic which has to our knowledge not been treated in such an implementation-oriented way before.

In addition, to foster variety in geometry problems, we have created, prepared and organized a public programming contest composed entirely of geometry problems, in order to showcase innovative notions and techniques that we hope can be used by other problem setters to build other innovative problems in the future.

Future work

The handbook we created can still be expanded and improved in many ways, and we plan to do continue to develop and share it in the future until it covers most of the computational geometry algorithms relevant to competitive programming, as explained in section 2.3.

In terms of problem setting, we believe that geometry still has a long way to go until it reaches maturity. Problems like those we presented in our contest which are considered difficult today will likely become common knowledge in a few years, and with this shift, we can expect even more interesting problems and ideas to appear in the future. We encourage problem setters to continue raising the bar in originality, and to delve into fields that have so far been neglected such as differential geometry and topology.

Part II

Handbook of geometry for competitive programmers

Chapter 5

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.

5.1 Small imprecisions can become big imprecisions

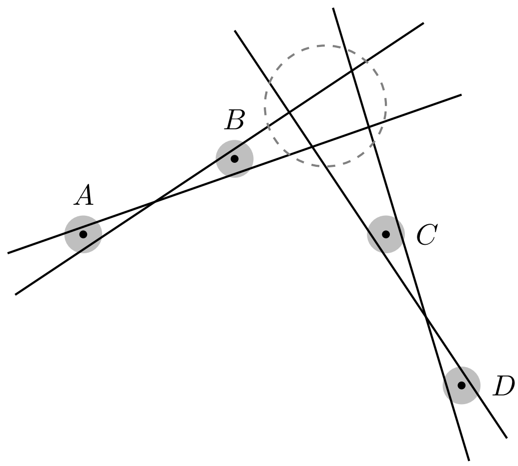
In this section we explore two ways in which very small starting imprecisions can become very large imprecisions in the final output of a program.

5.1.1 When doing numerically unstable computations

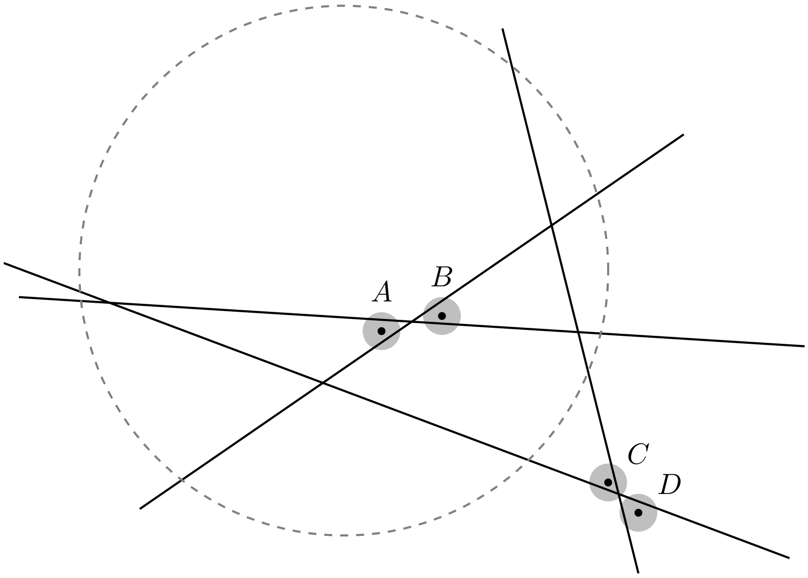
There are some types of computations which can transform small imprecisions into catastrophically large ones, and line intersection is one of them. Imagine you have four points A, B, C, D which were obtained through an previous imprecise process (for example, their position can vary by a distance of 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.

5.1.2 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 5.4.1, 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.

5.2 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.

5.2.1 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}.

5.2.2 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 [3].

5.3 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.

5.3.1 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

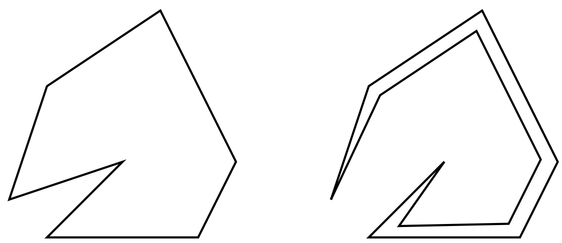
[3] Strictly speaking, we mean both subtraction of values of the same sign, and addition of values of opposite signs.

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.[4]

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 5.4: 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.

[4]In fact, working with relative error tolerances would make sense if this "relative error" was defined based on the magnitude of the input coordinates rather than on the magnitude of the answer, as we will see starting from section 5.3.3. For example, if all input coordinates are bounded by M, it would make sense to require an absolute precision of $M^2 \times 10^{-6}$ on an area. But since the answer can remain very small even if the magnitude of the input grows, requiring a fixed relative precision on it is usually too constraining for test cases with inputs of large magnitude.

5.3.2 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.[5] 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.

5.3.3 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[6] will have an absolute error of at most $nM\epsilon$ compared to the theoretical

[5] 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.

[6] 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

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 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:

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

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.

5.3.4 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

in 3 operations" even though we can implement this with only 2 additions.

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 B.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.

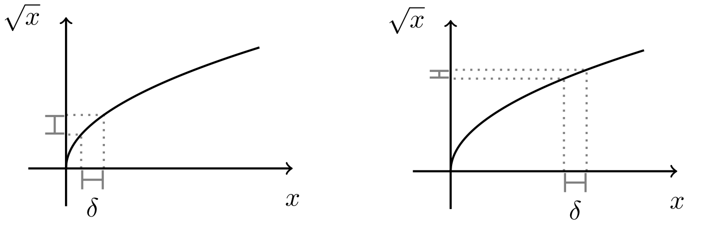
5.3.5 Why other operations do not work as well

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 5.1.1 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.



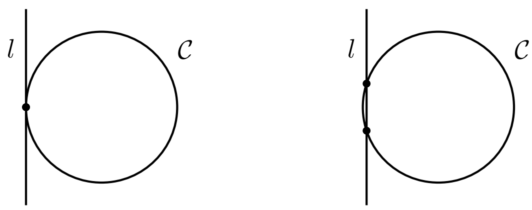
close to 0 $\Rightarrow$ big effect far from 0 $\Rightarrow$ small effect

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 $M\epsilon$.

5.4 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.

5.4.1 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 5.1.2, 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 5.3.1).

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[7] $(\max(k_a, k_b)+1)\epsilon$.

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:

$$\left(\cdots\left(\left((a_1 + a_2) + a_3\right) + a_4\right) + \cdots\right) + a_n$$

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 $(\max(1, 0) + 1)\epsilon = 2\epsilon$;
- $\left((a_1 + a_2) + a_3\right) + a_4$ 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:

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

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

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 $2\log_2(n)\epsilon$.[8] 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);
        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}

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

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.[9]

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.

5.4.2 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 5.3.5 can also happen here.[10] Take the example of equation $x^2 - 2x + 1 = 0$,

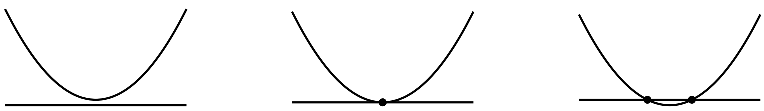
[9] Theoretically we can't really be sure though, since both `sumA` and `sumB` could have that error, and we still have to take into account the other operations performed.

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

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.



$x^2 - 2x + (1 + \epsilon) = 0$ $\quad$ $x^2 - 2x + 1 = 0$ $\quad$ $x^2 - 2x + (1 - \epsilon) = 0$

no solution $\quad$ $x = 1$ $\quad$ $x = 1 \pm \sqrt{\epsilon}$

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 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);
}
```

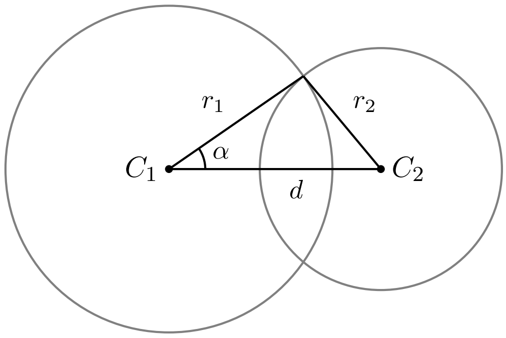
In many cases, there are several ways to write an expression, and they can have very different behaviors when used with floating-point numbers. So if you realize that the expression you are using can cause precision problems in some cases, it can be a good idea to rearrange the expression to handle them,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

5.4.3 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.[11] 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.

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



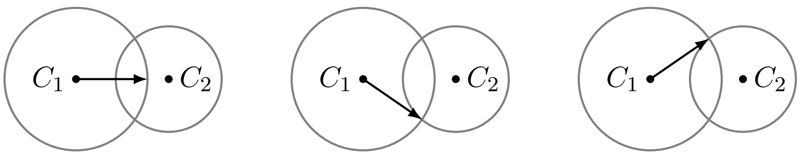
We know that such a triangle exists iff the triangle inequalities are respected, that is:

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

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.



This gives the following code. It uses a function `abs()` to compute the length of a vector (see section 6.1.2) and a function `rot()` to rotate a vector by a given angle (see section 6.2.3).

```
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;
}
```

5.5 Some advice

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

5.5.1 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 restricted 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.

5.5.2 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 5.3.1).
- 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 6

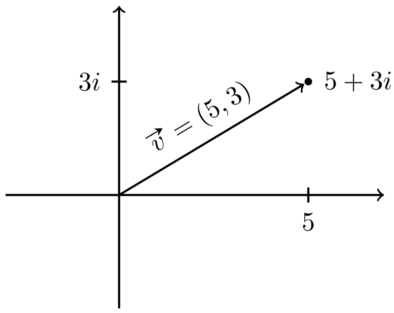
Basics

6.1 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.

6.1.1 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 $\mathbb{C}$.

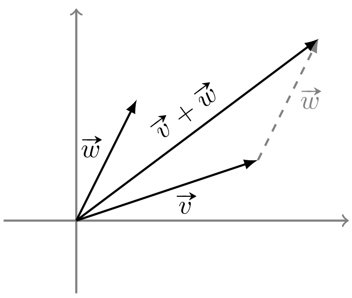


Basic operations

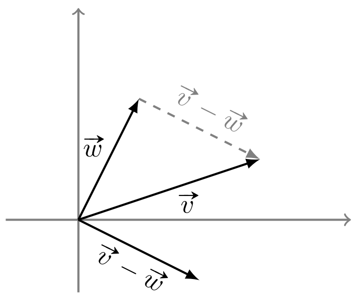
Complex numbers are added, subtracted and multiplied by scalars as if i were an unknown variable. Those operations are equivalent to the same operations on vectors.

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

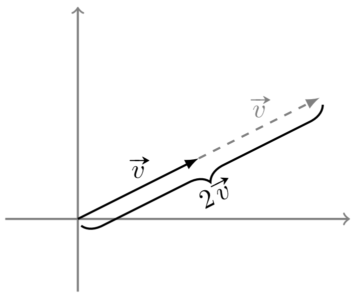
Geometrically, adding or subtracting two complex numbers $\vec{v} = (a, b)$ and $\vec{w} = (c, d)$ corresponds to making $\vec{w}$ or its opposite start at the end of $\vec{v}$, while multiplying $\vec{v}$ by a positive real k corresponds to multiplying its length by k but keeping the same direction.



addition



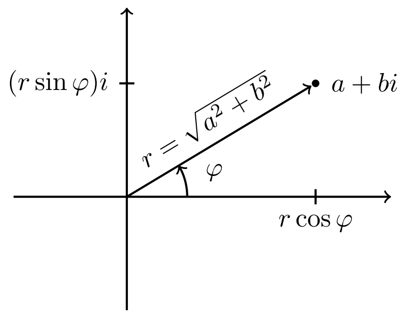
subtraction



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.

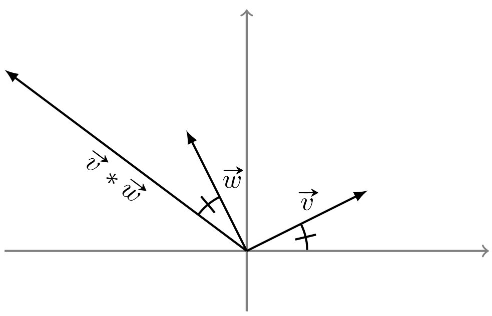
Note that this is not a one-to-one mapping. Firstly, adding or subtracting 2π from φ doesn't change the point being represented; to solve this problem, φ is generally taken in $(-\pi, \pi]$. Secondly, when $r = 0$, all values of φ represent the same point.

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.

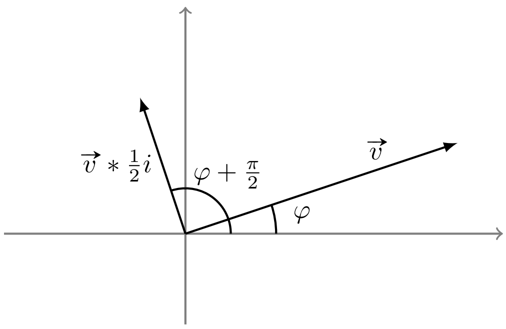
$$\begin{aligned}(a+bi)*(c+di) &= ac + a(di) + (bi)c + (bi)(di)\\ &= ac + adi + bci + (bd)i^2\\ &= ac + (ad+bc)i + (bd)(-1)\\ &= (ac-bd) + (ad+bc)i\end{aligned}$$

Exercise 1

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.

[Go to solution]

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.



6.1.2 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.

With a custom structure

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

```
typedef double T;
```

```cpp
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:

```cpp
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:

```cpp
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:

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

Some example usage:

```cpp
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)
```

We also define a signum function, which will be useful for several applications. It returns `-1` for negative numbers, `0` for zero, and `1` for positive numbers.

```cpp
template <typename T> int sgn(T x) {
    return (T(0) < x) - (x < T(0));
}
```

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.

```cpp
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:

```cpp
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:

```cpp
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
```

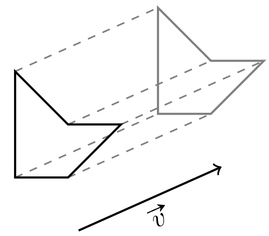
Therefore, to be safe you should implement a separate `sq()` function as for the custom structure (or you can wait use function `dot()` that we will define later).

6.2 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.

6.2.1 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 $\vec{v} \in \mathbb{C}$.

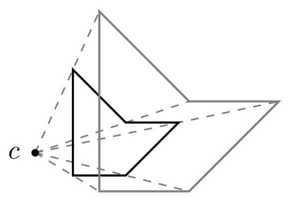


The implementation is self-explanatory:

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

6.2.2 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).

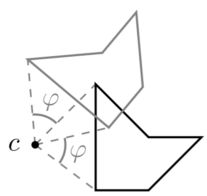


Again, the implementation is just a translation of the expression into code:

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

6.2.3 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 6.1.1 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:

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

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

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

6.2.4 General linear transformation

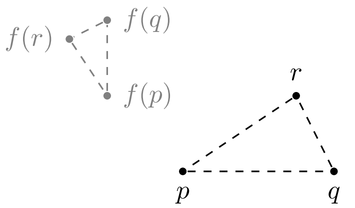
It is easy to check that all those transformations are of the form $f(p) = a*p+b$ as claimed in the beginning of this section. In fact, all transformations of this type can be obtained as combinations of translations, scalings and rotations.[1]

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:

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

[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.



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 [**kactl**] (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);
}
```

6.3 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.

6.3.1 Dot product

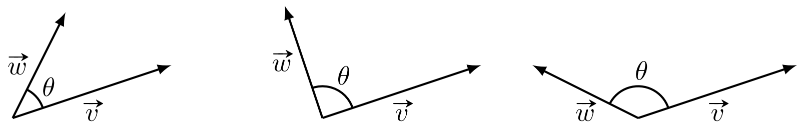
The dot product $\vec{v} \cdot \vec{w}$ of two vectors $\vec{v}$ and $\vec{w}$ can be seen as a measure of how similar their directions are. It is defined as

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

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.



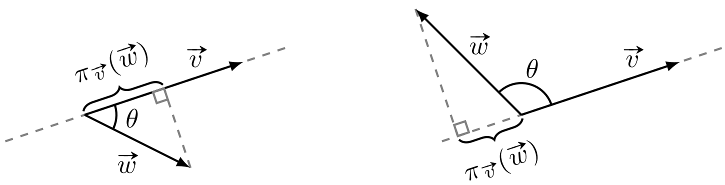
$\theta < \pi/2$ $\quad$ $\theta = \pi/2$ $\quad$ $\theta > \pi/2$

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

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.



$\pi_{\vec{v}}(\vec{w}) = 1.58$ $\quad$ $\pi_{\vec{v}}(\vec{w}) = -1.26$

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 angle(pt v, pt w) {
    double cosTheta = dot(v,w) / abs(v) / abs(w);
    return acos(max(-1.0, min(1.0, cosTheta)));
```

```
}
```

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

```
double angle(pt v, pt w) {
    return acos(clamp(dot(v,w) / abs(v) / abs(w), -1.0, 1.0));
}
```

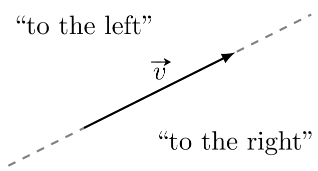
6.3.2 Cross product

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

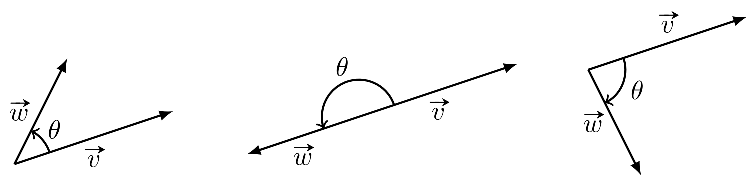
$$\vec{v} \times \vec{w} = \|\vec{v}\|\|\vec{w}\| \sin\theta$$

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.



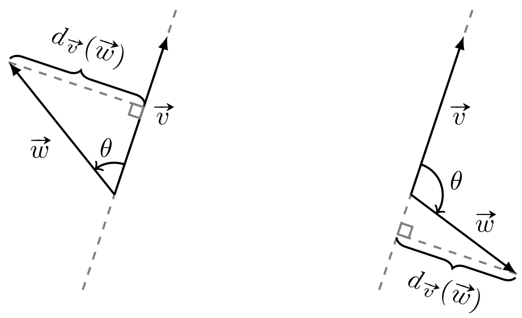
$0 < \theta < \pi$ $\quad$ $\theta = \pi$ $\quad$ $-\pi < \theta < 0$

$\vec{v} \times \vec{w} = 5$ $\quad$ $\vec{v} \times \vec{w} = 0$ $\quad$ $\vec{v} \times \vec{w} = -7$

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.



$d_{\vec{v}}(\vec{w}) = 2.69$ $\qquad$ $d_{\vec{v}}(\vec{w}) = -2.37$

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

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

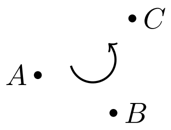
and see that the real and imaginary parts are indeed the dot product and the cross product.

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 $\text{orient}(A, B, C) = \overrightarrow{AB} \times \overrightarrow{AC}$. It is 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:

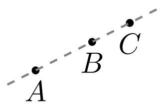
```
T orient(pt a, pt b, pt c) {return cross(b-a,c-a);}
```

In other words, $\text{orient}(A, B, C)$ 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.



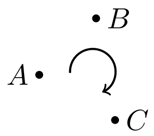
left turn

$\text{orient}(A, B, C) > 0$



collinear

$\text{orient}(A, B, C) = 0$



right turn

$\text{orient}(A, B, C) < 0$

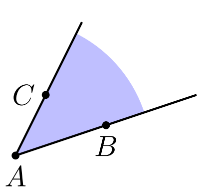
Its value is conserved by cyclic rotation, that is

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

while swapping any two arguments switches the sign.

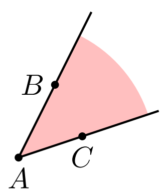
As an example of use, suppose we want to check if point X lies in the angle formed by lines AB and AC. We can follow this procedure:

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



$\text{orient}(A, B, C) > 0$

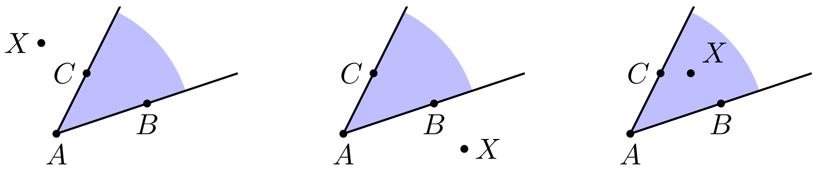
no swap



$\text{orient}(A, B, C) < 0$

swap

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



orient$(A,B,X) \geq 0$ | orient$(A,B,X) < 0$ | orient$(A,B,X) \geq 0$

orient$(A,C,X) > 0$ | orient$(A,C,X) \leq 0$ | 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;
}
```

Using orient() we can also easily compute the amplitude of an *oriented angle* $\widehat{BAC}$, that is, the angle that is covered if we turn from B to C around A counterclockwise.

There are two cases: either orient$(A,B,C) \geq 0$, with an angle in $[0,\pi]$, or orient$(A,B,C) < 0$, with an angle in $(\pi, 2\pi)$. In the first case, we can simply use the `angle()` function we create based on the dot product; in the second case, we should take "the other side", so 2π minus that result.

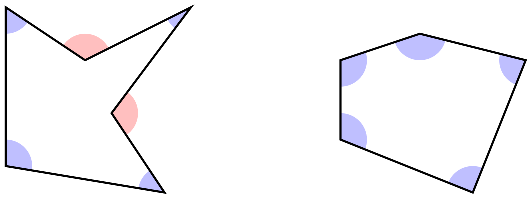
orient$(A,B,C) > 0$ | orient$(A,B,C) < 0$

`angle()` $= 50°$ | `angle()` $= 80°$

`orientedAngle()` $= 50°$ | `orientedAngle()` $= 280°$

```
double orientedAngle(pt a, pt b, pt c) {
    if (orient(a,b,c) >= 0)
        return angle(b-a, c-a);
    else
        return 2*M_PI - angle(b-a, c-a);
}
```

Yet another use case is checking if a polygon $P_1 \cdots P_n$ is convex: we compute the n orientations of three consecutive vertices 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.



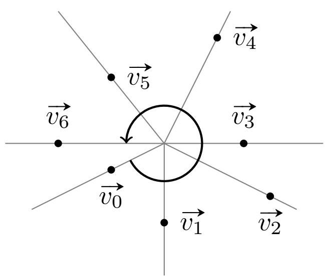
different signs $\Rightarrow$ not convex

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);
}
```

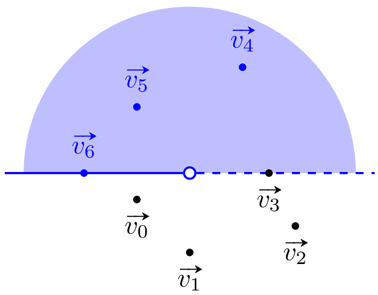
Polar sort

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]



[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`**.

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));
    });
}
```

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.

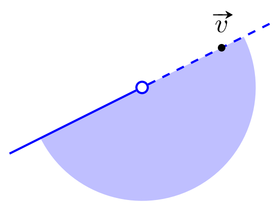
- We can perform a polar sort around some point O other than the origin: we just have to subtract that point O from the vectors $\vec{v}$ and $\vec{w}$ when comparing them. This as if we translated the whole plane so that O is moved to $(0, 0)$:

```
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:

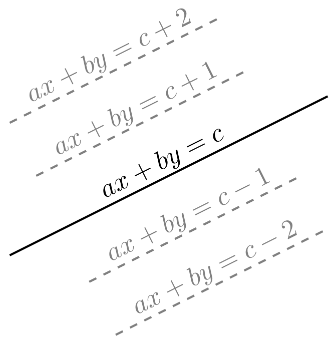


6.4 Lines

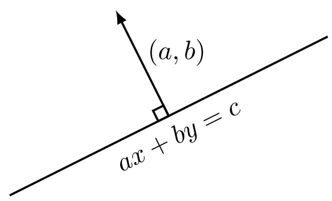
In this section we will discuss how to represent lines and a wide variety of applications.

6.4.1 Line representation

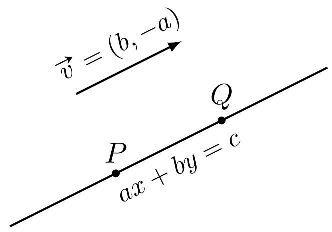
Lines are sets of points (x, y) in the plane which obey an equation of the form $ax + by = c$, with at least one of a and b nonzero. a and b determine the direction of the line, while c determines its position.



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 6.3.1 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 6.3.2 that the cross product remains constant when the second vector moves parallel to the first.

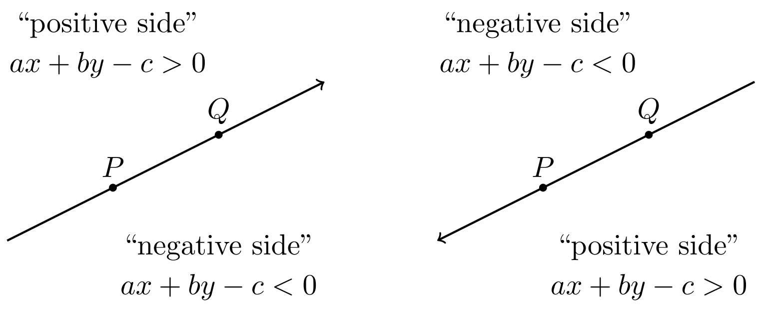


In this way, finding the equation of a line going through two points P and Q is easy: define the direction vector $\vec{v} = (b, -a) = \overrightarrow{PQ}$, then find c as $\vec{v} \times P$.

```
struct line {
    pt v; T c;
    // From direction vector v and offset c
```

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

    // Will be defined later:
    // - these work with T = int
    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 refl(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)`.



created as `line(p,q)`
$\vec{v} = \overrightarrow{PQ}$

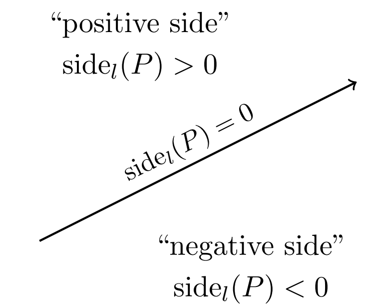
created as `line(q,p)`
$\vec{v} = \overrightarrow{QP}$

6.4.2 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

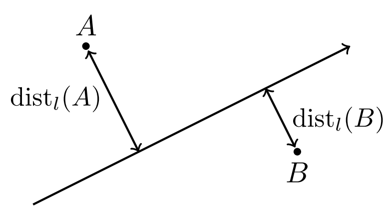
$$\mathrm{side}_l(P) := ax + by - c = \vec{v} \times P - c.$$

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




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

The $\mathrm{side}_l(P)$ operation also gives the distance to l, up to a constant factor: the bigger $\mathrm{side}_l(P)$ is, the further from line l. In fact, we can prove that $|\,\mathrm{side}_l(P)|$ is $\|v\|$ times the distance between P and l (this should make sense if you've read the “mathy insight” in section 6.3.2).



This gives an easy implementation of distance:

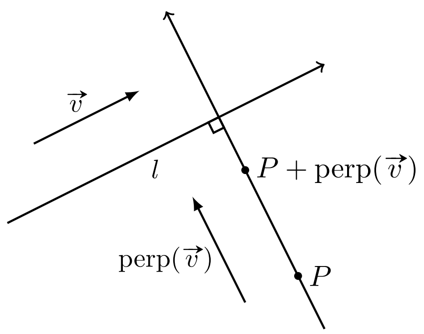
```
double dist(pt p) {return abs(side(p)) / abs(v);}
```

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);}
```

6.4.3 Perpendicular through a point

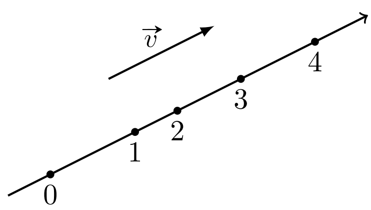
Two lines are perpendicular if and only if their direction vectors are perpendicular. Let's say we have a line l of direction vector $\vec{v}$. To find a line perpendicular to line l and which goes through a certain point P, we could define its direction vector as $\text{perp}(\vec{v})$ (that is, $\vec{v}$ rotated by 90° counterclockwise, see section 6.2.3) 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)};}
```

6.4.4 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 $\vec{v}$.



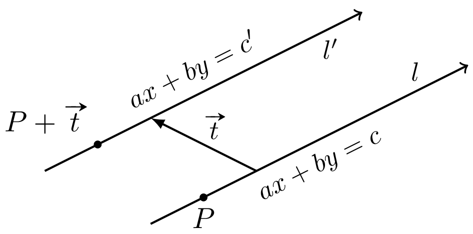
We can use the dot product to figure out the order of two points: a point A comes before a point B if $\vec{v} \cdot A < \vec{v} \cdot B$. So we can a comparator out of it.

```
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 6.3.1.

6.4.5 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

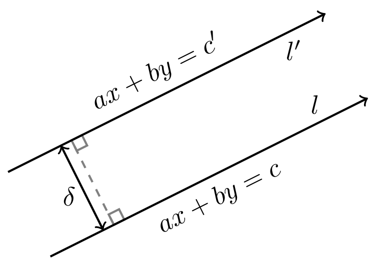
$$\text{side}_l(P) = \vec{v} \times P - c = 0$$
$$\text{side}_{l'}\left(P + \vec{t}\right) = \vec{v} \times \left(P + \vec{t}\right) - c' = 0$$

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)};}
```

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



[3] If you don’t know what an orthogonal projection is, read section 6.4.7.

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

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

so in this case c' becomes

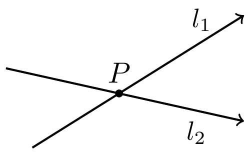
$$\begin{aligned} c' &= c + \vec{v} \times \vec{t} \\ &= c + (\delta/\|\vec{v}\|)(\vec{v} \times \mathrm{perp}(\vec{v})) \\ &= c + (\delta/\|\vec{v}\|)\|\vec{v}\|^2 \\ &= c + \delta\|\vec{v}\| \end{aligned}$$

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

6.4.6 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 $\mathrm{side}_{l_1}(P) = 0$:

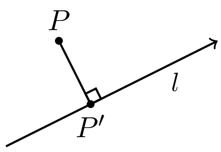
$$\begin{aligned} \mathrm{side}_{l_1}(P) &= \overrightarrow{v_{l_1}} \times \left(\frac{c_{l_1}\overrightarrow{v_{l_2}} - c_{l_2}\overrightarrow{v_{l_1}}}{\overrightarrow{v_{l_1}} \times \overrightarrow{v_{l_2}}}\right) - c_{l_1} \\ &= \frac{c_{l_1}(\overrightarrow{v_{l_1}} \times \overrightarrow{v_{l_2}})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

```
bool inter(line l1, line l2, pt &out) {
    T d = cross(l1.v, l2.v);
    if (d == 0) return false;
```

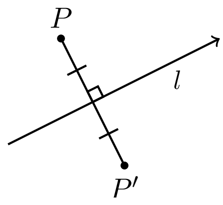
```
    out = (l2.v*l1.c - l1.v*l2.c) / d; // requires floating-point
        coordinates
    return true;
}
```

6.4.7 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 that is at the same distance and 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 $\mathrm{side}_l(P + k\,\mathrm{perp}(\vec{v})) = 0$.

We compute

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

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

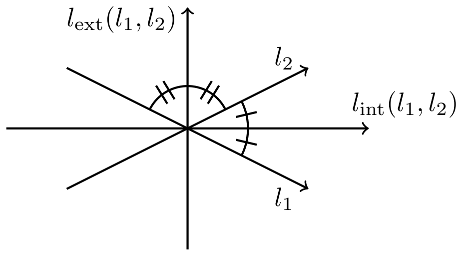
```
pt proj(pt p) {return p - perp(v)*side(p)/sq(v);}
```

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);}
```

6.4.8 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* $l_{\mathrm{int}}(l_1, l_2)$ 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 6.4.2 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

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

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 $l_{\text{int}}(l_1, l_2)$:

$$
\begin{aligned}
\vec{v} &= \frac{\overrightarrow{v_{l_1}}}{\|\overrightarrow{v_{l_1}}\|} + \frac{\overrightarrow{v_{l_2}}}{\|\overrightarrow{v_{l_2}}\|} \\
c &= \frac{c_{l_1}}{\|\overrightarrow{v_{l_1}}\|} + \frac{c_{l_2}}{\|\overrightarrow{v_{l_2}}\|}
\end{aligned}
$$

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)};
}
```

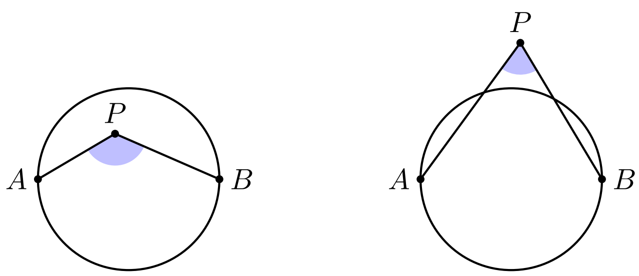
6.5 Segments

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

6.5.1 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

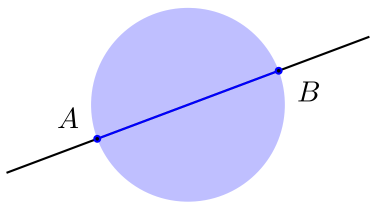
$\overrightarrow{PA} \cdot \overrightarrow{PB} > 0$
out of disk

```
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);
}
```

6.5.2 Segment-segment intersection

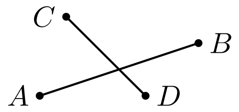
Finding the precise intersection between two segments $[AB]$ and $[CD]$ is quite tricky: many configurations are possible and the intersection itself might be empty, a single point or a whole segment.

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.[4]

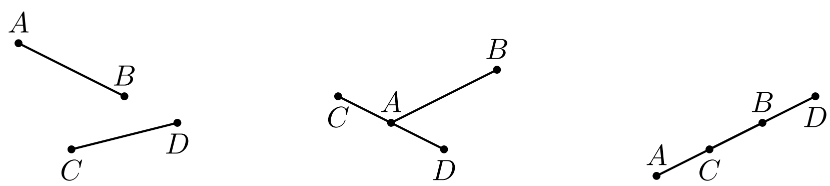
[4] We can understand the formula as the center of gravity of point A with weight $|o_B|$ and point B with weight $|o_A|$, which gives us a point I on $[AB]$ such that $|IA|/|IB| = o_A/o_B$.



proper intersection

```
bool properInter(pt a, pt b, pt c, pt d, pt &out) {
    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) {
        out = (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.

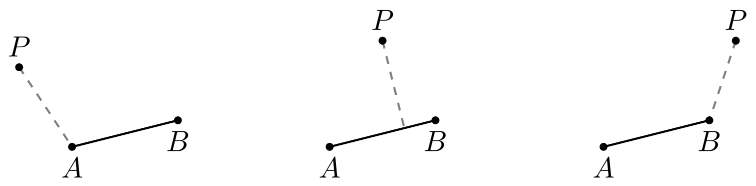


$S = \emptyset$ no intersection — $S = \{A\}$ intersection point — $S = \{B, C\}$ intersection segment

```
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;
}
```

6.5.3 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
}
```

6.5.4 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:

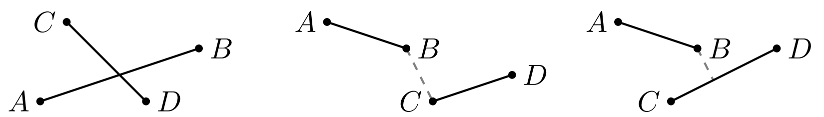
1. Segments $[AB]$ and $[CD]$ intersect properly, in which case the distance is of course 0.
2. In all other cases, the shortest distance between the segments is attained in at least one of the endpoints, so we only need to test the four endpoints and report the minimum.

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.



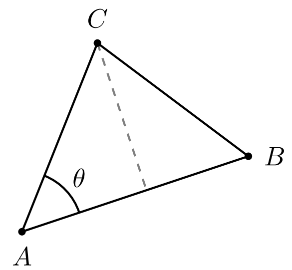
proper intersection | between points | between point and line

6.6 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.

6.6.1 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,

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

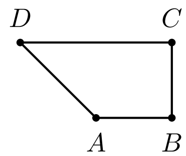
Since O is the origin, it can be implemented simply like this:

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

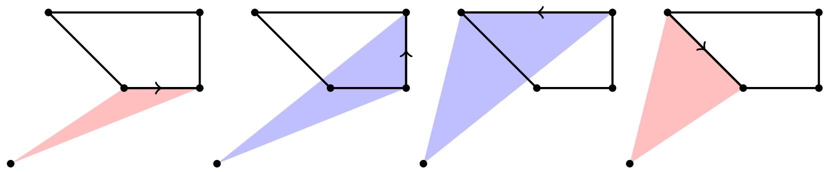
Now that we can compute the area of a triangle, the intuitive way to find the area of a polygon would be to

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).

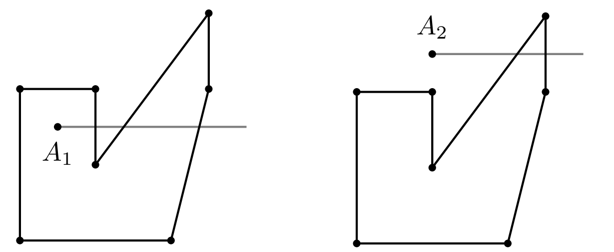
Note that the sign (add or subtract) that we take for the area of OP_1P_2 is exactly the sign that the cross product takes. If we take the origin as reference point O, it gives this simple implementation:

```
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.

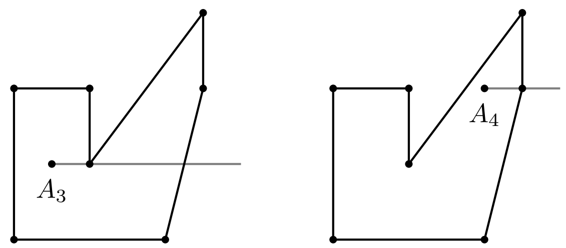
6.6.2 Cutting-ray test

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.

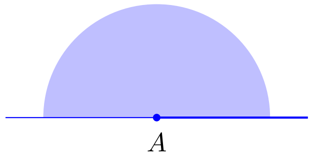


3 inters $\Rightarrow A_1$ inside   2 inters $\Rightarrow A_2$ outside

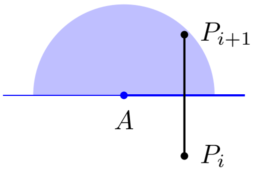
However, sometimes this can go wrong if the ray touches a vertex of the polygon, as below. The ray from A_3 intersects the polygon twice, but A_3 is inside. We can try to solve the issue by counting one intersection per segment touched, which would give three intersections for A_3, but then the ray from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

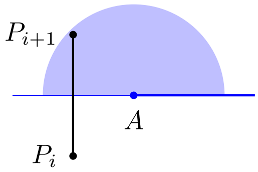


Below we show for some segments whether they are considered to cross the ray or not. We can see in the last two examples that the behavior is different if the segment touches the ray from below or from above.



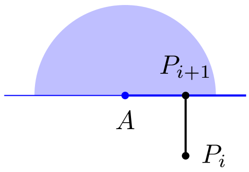
- touches ray: OK
- halves $\neq$: OK

$\Rightarrow$ crossing



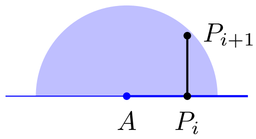
- touches ray: KO
- halves $\neq$: OK

$\Rightarrow$ no crossing



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

$\Rightarrow$ crossing



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

$\Rightarrow$ no crossing

Exercise 2

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;
}
```

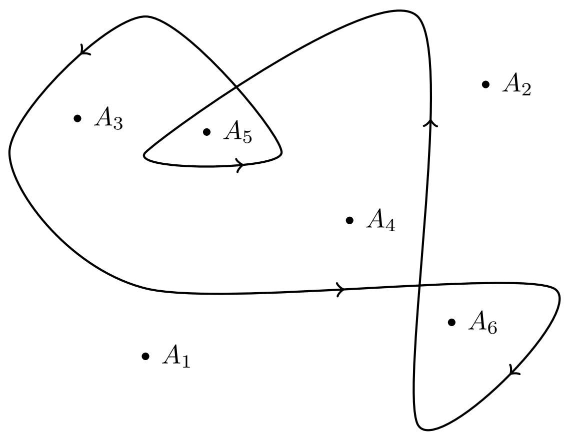
If we now return to the original problem, we still have to check whether A is on the boundary of the polygon. We can do that by using `onSegment()` defined in 6.5.1.

```
// 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
}
```

6.6.3 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.



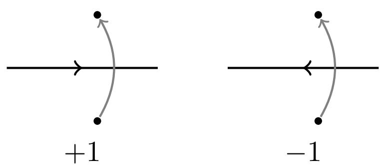
Points A_1 and A_2 are completely out of the curve so the winding number around them is 0 (no turn). Points A_3 and A_4 are inside the main loop, which goes counterclockwise, so the winding number around them is 1. The curve turns twice counterclockwise around A_5, so the winding number is 2. Finally 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 3

What value will `areaPolygon()` (section 6.6.1) 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.

[Go to solution]

To compute the winding number, we need to keep track of the amplitude travelled, positive if counterclockwise, and negative if clockwise. We can use `angle()` from section 6.3.1 to help us.

```
// amplitude travelled around point A, from P to Q
double angleTravelled(pt a, pt p, pt q) {
    double ampli = angle(p-a, q-a);
    if (orient(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.[5] 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.

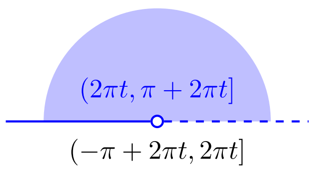
```
struct angle {
    pt d; int t = 0; // direction and number of full turns
    angle t180(); // to be defined later
    angle t360() {return {d, t+1};}
};
```

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 6.3.2, 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) <
           make_tuple(b.t, half(b.d), cross(a.d,b.d));
}
```

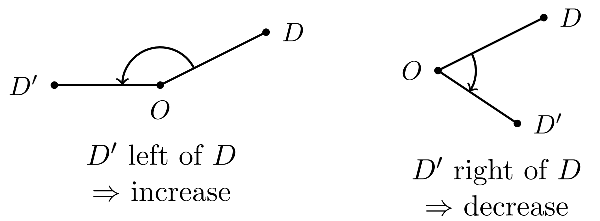
[5]This approach is based on an original idea in [**kactl**], see "Angle.h".

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.



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;
}
```

6.7 Circles

6.7.1 Defining a circle

Circle (O, r) is the set of points at distance exactly r from a point $O = (x_0, y_0)$.

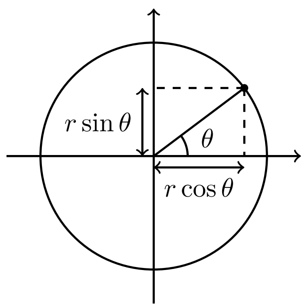
We can also define it by equation

$$(x - x_0)^2 + (y - y_0)^2 = r^2$$

or parametrically as the set of all points

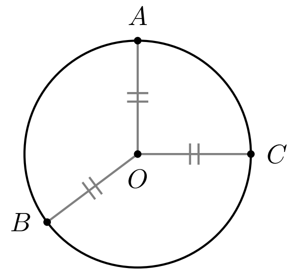
$$(x_0 + r\cos\theta, y_0 + r\sin\theta)$$

with θ in $[0, 2\pi)$, for example.



6.7.2 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 $\overrightarrow{AO}$.

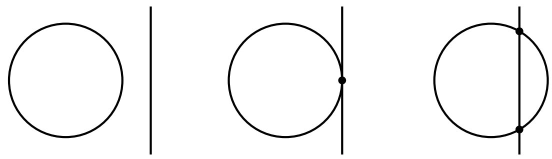
Math insight

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 $\operatorname{perp}(\overrightarrow{AB})$, and it passes through the middle point $\frac{1}{2}\overrightarrow{AB}$, so its constant term (variable `c` in the `line` structure) is $\operatorname{perp}(\overrightarrow{AB}) \times \frac{1}{2}\overrightarrow{AB} = -\frac{1}{2}|AB|^2$. Similarly, the bisector of $[AC]$ is defined by direction vector $\operatorname{perp}(\overrightarrow{AC})$ and constant term $-\frac{1}{2}|AC|^2$. We then just plug those into the formula for line intersection found in section 6.4.6:

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

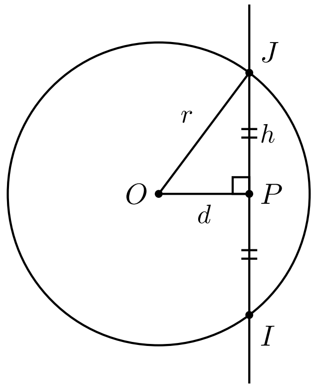
6.7.3 Circle-line intersection

A circle (O, 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 O onto the line l, which we will call P.



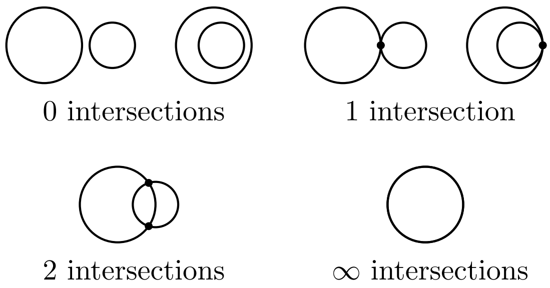
Once we have found P, to find I and J we need to move along the line by a certain distance h. By the Pythagorean theorem, $h = \sqrt{r^2 - d^2}$ where d is the distance from O to l.

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.

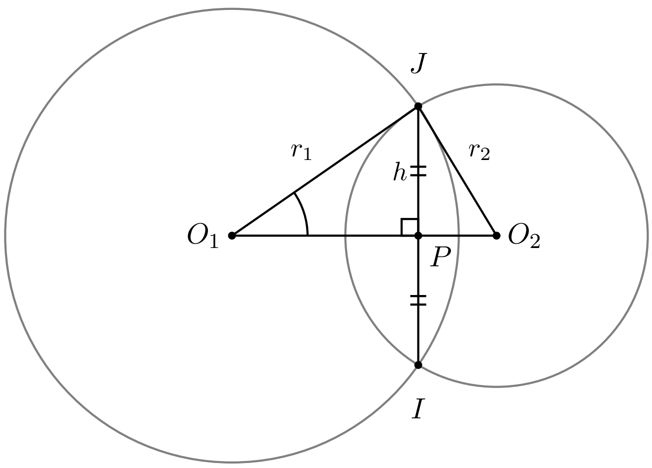
```
int circleLine(pt o, double r, line l, pair<pt,pt> &out) {
    double h2 = r*r - l.sqDist(o);
    if (h2 >= 0) { // the line touches the circle
        pt p = l.proj(o); // point P
        pt h = l.v*sqrt(h2)/abs(l.v); // vector parallel to l, of
            length h
        out = {p-h, p+h};
    }
    return 1 + sgn(h2);
}
```

6.7.4 Circle-circle intersection

Similarly to the previous section, two circles (O_1, r_1) and (O_2, r_2) can have either 0, 1, 2 or an infinity of intersection points (in case the circles are identical).



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 $d = |O_1O_2|$. We know from the law of cosines on O_1O_2J that

$$\cos(\measuredangle O_2O_1J) = \frac{d^2 + r_1^2 - r_2^2}{2dr_1}$$

and since O_1PJ is a right triangle,

$$|O_1P| = r_1 \cos(\measuredangle O_2O_1J) = \frac{d^2 + r_1^2 - r_2^2}{2d}$$

which allows us to find P.

Now to find $h = |PI| = |PJ|$, we apply the Pythagorean theorem on triangle O_1PJ, which gives $h = \sqrt{r_1^2 - |O_1P|^2}$.

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 o1, double r1, pt o2, double r2, pair<pt,pt> &
    out) {
    pt d=o2-o1; double d2=sq(d);
    if (d2 == 0) {assert(r1 != r2); return 0;} // concentric circles
    double pd = (d2 + r1*r1 - r2*r2)/2; // = |O_1P| * d
    double h2 = r1*r1 - pd*pd/d2; // = hˆ2
    if (h2 >= 0) {
        pt p = o1 + d*pd/d2, h = perp(d)*sqrt(h2/d2);
        out = {p-h, p+h};
    }
    return 1 + sgn(h2);
}
```

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

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

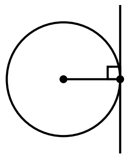
Let's assume this is nonnegative. Thus an even number of those conditions are false:

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

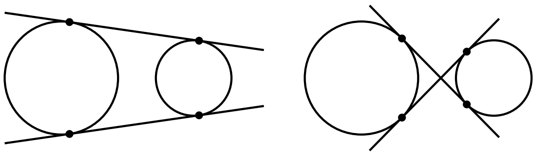
Since $d, r_1, r_2 \geq 0$, no two of those can be simultaneously false, so they must all be true. As a consequence, the triangle inequalities are verified for d, r_1, r_2, showing the existence of a point at distance r_1 from O_1 and distance r_2 from O_2.

6.7.5 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.

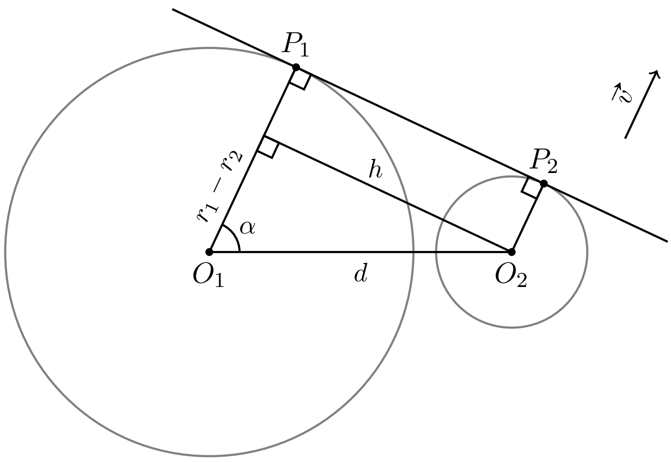


Here we will try and find a line which is tangent to two circles (O_1, r_1) and (O_2, r_2). There are two types of such tangents: outer tangents, for which both circles are on the same side of the line, and inner tangents, for which the circles are on either side.



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 $[O_1P_1]$ and $[O_2P_2]$, in other words, we want to find $\vec{v} = \overrightarrow{O_1P_1}/r_1$. To do this we will try to find angle α marked on the figure.



If we project O_2 onto the ray from O_1, this forms a right triangle with hypothenuse $d = |O_1O_2|$ 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 $\overrightarrow{O_1O_2}$ and $\operatorname{perp}\left(\overrightarrow{O_1O_2}\right)$ as

$$\begin{aligned}\vec{v} &= \cos\alpha\left(\overrightarrow{O_1O_2}/d\right) \pm \sin\alpha\left(\operatorname{perp}\left(\overrightarrow{O_1O_2}\right)/d\right)\\ &= \frac{(r_1 - r_2)\,\overrightarrow{O_1O_2} \pm h\operatorname{perp}\left(\overrightarrow{O_1O_2}\right)}{d^2}\end{aligned}$$

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

$$P_1 = O_1 + r_1\vec{v} \quad \text{and} \quad P_2 = O_2 + r_2\vec{v}$$

Exercise 4

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(o1,p).perpThrough(p)` (see 6.4.3);
- if there are 0 tangents, it does nothing;
- if the circles are identical, it aborts.

```
int tangents(pt o1, double r1, pt o2, double r2, bool inner, V<pair<
    pt,pt>> &out) {
    if (inner) r2 = -r2;
    pt d = o2-o1;
    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({o1 + v*r1, o2 + 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).

Chapter 7

3D geometry

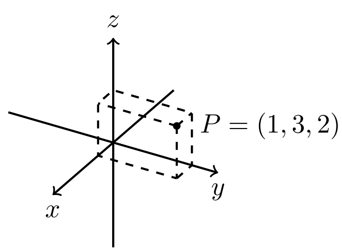
In this chapter we cover the basic objects and concepts of 3D geometry, and the operations we can do on them. Although the additional dimension adds some complexity which can make things more difficult and interesting, a lot of things work the same way as they do in 2D. Therefore, we will make frequent references to chapter 6, which we consider as a prerequisite.

7.1 Points, products and orientation

In this first section, we define our point representation, we explain how the dot and cross products we saw in 2D translate into 3D, and we show how a combination of the two, the *mixed product*, can help us define a 3D analog of the `orient()` function.

7.1.1 Point representation

We will define points and vectors in space by their coordinates (x, y, z): their positions along three perpendicular axes.



As we did in 2D, we start with some basic operators.

```cpp
typedef double T;
struct p3 {
    T x,y,z;

    // Basic vector operations
    p3 operator+(p3 p) {return {x+p.x, y+p.y, z+p.z};}
    p3 operator-(p3 p) {return {x-p.x, y-p.y, z-p.z};}
    p3 operator*(T d) {return {x*d, y*d, z*d};}
    p3 operator/(T d) {return {x/d, y/d, z/d};} // only for floating-point

    // Some comparators
    bool operator==(p3 p) {return tie(x,y,z) == tie(p.x,p.y,p.z);}
    bool operator!=(p3 p) {return !operator==(p);}
};
```

Choosing the scalar type `T` is done the same way as in 2D, see 6.1.2 for our remarks on that.

For convenience, we also define a zero vector $\vec{0} = (0, 0, 0)$:

```cpp
p3 zero{0,0,0};
```

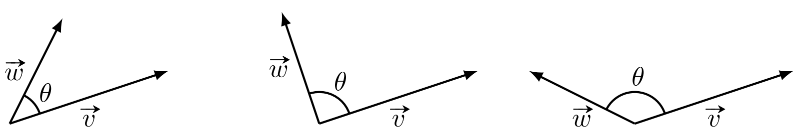
7.1.2 Dot product

The dot product is exactly the same as in 2D. 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 $\vec{w}$. So in other words, the 3D dot product of $\vec{v}$ and $\vec{w}$ is equal to the 2D dot product they would have on a plane that contains them both.

We recall the possible cases for the sign of the dot product:



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

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

We recall some properties of dot product and add a few:

- symmetry: $\vec{v} \cdot \vec{w} = \vec{w} \cdot \vec{v}$;
- linearity: $(\vec{v_1} + \vec{v_2}) \cdot \vec{w} = \vec{v_1} \cdot \vec{w} + \vec{v_2} \cdot \vec{w}$ (also on the right);
- $\vec{v} \cdot \vec{w} = 0$ iff $\vec{v}$ and $\vec{w}$ are perpendicular;
- $\vec{v} \cdot \vec{w}$ stays constant if $\vec{w}$ moves perpendicular to $\vec{v}$.

Like in 2D, dot product is very simple to implement:

```
T operator|(p3 v, p3 w) {return v.x*w.x + v.y*w.y + v.z*w.z;}
```

Since 3D geometry uses dot and cross product a lot, to shorten notations we will be using operator `|` for dot product and operator `*` for cross product. We chose them for these mostly arbitrary reasons:

- `|` has a lower precedence than `*` which is desirable, it kind of looks like a "parallel" operator, and since it most often has to be parenthesized (e.g. `(v|w) == 0`), it can be a bit reminiscent of the inner product notation $\langle v, w\rangle$;
- `*` is the closest thing to a cross in overloadable R++ operators, and the compiler doesn't produce a warning if you don't paranthesize `b*c` in expressions such as `a|b*c`, which we will use a lot.

We define the usual `sq()` and `abs()` based on dot product and add a `unit()` function that makes the norm of a vector equal to 1 while preserving its direction.

```
T sq(p3 v) {return v|v;}
double abs(p3 v) {return sqrt(sq(v));}
p3 unit(p3 v) {return v/abs(v);}
```

Like in 2D, we can also use dot product to find the amplitude in $[0, \pi]$ of the angle between vectors $\vec{v}$ and $\vec{w}$, see secton 6.3.1 for more details.

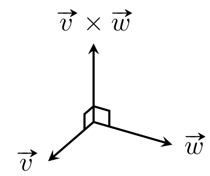
```
double angle(p3 v, p3 w) {
    double cosTheta = (v|w) / abs(v) / abs(w);
    return acos(max(-1.0, min(1.0, cosTheta)));
}
```

7.1.3 Cross product

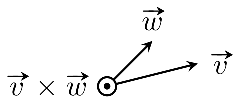
While the cross product in 2D is a scalar, in 3D it is a vector. If $\vec{v}$ and $\vec{w}$ are parallel, $\vec{v} \times \vec{w} = \vec{0}$, and otherwise it is defined as

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

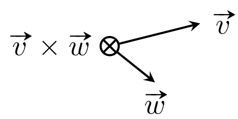
where $\|\vec{v}\|$ and $\|\vec{w}\|$ are the lengths of the vectors, θ is amplitude of the angle between $\vec{v}$ and $\vec{w}$, and $\vec{n}$ is a unit vector perpendicular to both $\vec{v}$ and $\vec{w}$ chosen using the right-hand rule. Note that the norm of the 3D cross product is equal to the absolute value of the 2D cross product.



The right-hand rule says this: if you take your right hand, align your thumb with $\vec{v}$ and your extended index $\vec{w}$, and fold your middle finger at a 90° angle, then it will point in the direction of $\vec{v} \times \vec{w}$. Another way to express it is to say that if you draw $\vec{v}$ and $\vec{w}$ on a sheet of paper and look at the sign of their 2D cross product, if it is positive $\vec{v} \times \vec{w}$ will point up from the sheet, and if it is negative $\vec{v} \times \vec{w}$ will point down through the sheet.



points towards you



points away from you

We summarize some key properties of the cross product:

- anti-symmetry: $\vec{v} \times \vec{w} = -\vec{w} \times \vec{v}$;
- linearity: $(\vec{v_1} + \vec{v_2}) \times \vec{w} = \vec{v_1} \times \vec{w} + \vec{v_2} \times \vec{w}$ (also on the right);
- $\vec{v} \times \vec{w}$ is perpendicular to both $\vec{v}$ and $\vec{w}$;[1]
- $\vec{v} \times \vec{w}$ is perpendicular to the plane containing $\vec{v}$ and $\vec{w}$;[1]
- $\vec{v} \times \vec{w} = \vec{0}$ iff $\vec{v}$ and $\vec{w}$ are perpendicular;
- $\vec{v} \times \vec{w}$ stays constant if $\vec{w}$ moves parallel to $\vec{v}$.

Among those, the one which we will use most often is the fact that it is perpendicular to $\vec{v}$ and $\vec{w}$.

The cross product can be computed this way:

```
p3 operator*(p3 v, p3 w) {
    return {v.y*w.z - v.z*w.y,
            v.z*w.x - v.x*w.z,
            v.x*w.y - v.y*w.x};
}
```

Indeed, it is easy to check that this vector is perpendicular to both $\vec{v}$ and $\vec{w}$ using dot product. Here it is for $\vec{v}$:

$$\begin{aligned}(\vec{v} \times \vec{w}) \cdot \vec{v} &= (v_y w_z - v_z w_y, v_z w_x - v_x w_z, v_x w_y - v_y w_x) \cdot (v_x, v_y, v_z) \\ &= v_x v_y w_z - v_x v_z w_y + v_y v_z w_x - v_y v_x w_z + v_z v_x w_z - v_z v_y w_x \\ &= 0\end{aligned}$$

[1] if $\vec{v} \times \vec{w} \neq \vec{0}$

Note that the z-coordinate is the same expression as the 2D cross product: indeed, it is the value of the 2D cross product if $\vec{v}$ and $\vec{w}$ are projected onto plane $z = 0$.

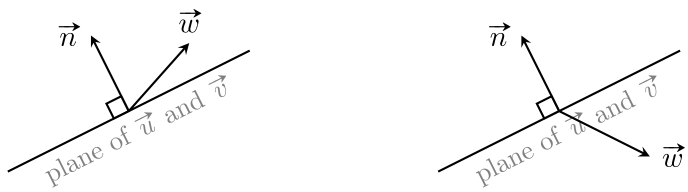
7.1.4 Mixed product and orientation

A very useful combination of dot product and cross product is the *mixed product*. We define the mixed product of three vectors $\vec{u}$, $\vec{v}$ and $\vec{w}$ as

$$(\vec{u} \times \vec{v}) \cdot \vec{w}$$

Let Π be the plane containing $\vec{u}$ and $\vec{v}$. We know that $\vec{n} = \vec{u} \times \vec{v}$ is perpendicular to Π, and $\vec{n} \cdot \vec{w}$ will be positive if the angle between $\vec{n}$ and $\vec{w}$ is less than 90°. This will happen if $\vec{w}$ points to the same side of Π as $\vec{n}$, while when $\vec{n} \cdot \vec{w}$ is negative $\vec{w}$ will point to the opposite side.

The two cases are illustrated in the drawings below. The plane containing $\vec{u}$ and $\vec{v}$ is viewed from the side:

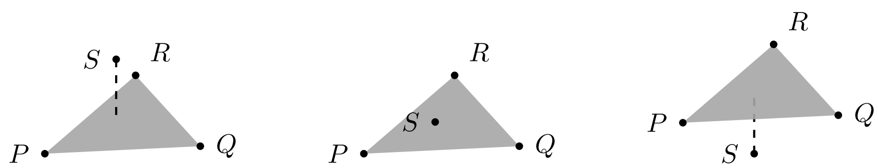


$(\vec{u} \times \vec{v}) \cdot \vec{w} > 0$ $\qquad$ $(\vec{u} \times \vec{v}) \cdot \vec{w} < 0$

Note that this is similar to how the 2D cross product $\vec{v} \times \vec{w}$ tells us to which side of to line containing $\vec{v}$ vector $\vec{w}$ points. So, we similarly define an `orient()` function based on it:

$$\text{orient}(P, Q, R, S) = \left(\overrightarrow{PQ} \times \overrightarrow{PR}\right) \cdot \overrightarrow{PS}$$

It is positive if S is on the side of plane PQR in the direction of $\overrightarrow{PQ} \times \overrightarrow{PR}$, negative if S is on the other side, and zero if S is on the plane.



S above PQR $\qquad$ S on PQR $\qquad$ S below PQR

$\text{orient}(P, Q, R, S) > 0$ $\qquad$ $\text{orient}(P, Q, R, S) = 0$ $\qquad$ $\text{orient}(P, Q, R, S) < 0$

This orient() function has several very nice properties. First, it stays the same if we swap any three arguments in a circular way: for example let's take P, Q, S, then $\operatorname{orient}(P, Q, R, S) = \operatorname{orient}(Q, S, R, P)$. On the other hand, swapping any two arguments changes its sign: for example, $\operatorname{orient}(P, Q, R, S) = -\operatorname{orient}(P, S, R, Q)$.

Exercise 5

Show that those properties also apply to the mixed product. For example, $(\vec{u} \times \vec{v}) \cdot \vec{w} = (\vec{v} \times \vec{w}) \cdot \vec{u}$ and $(\vec{u} \times \vec{v}) \cdot \vec{w} = -(\vec{u} \times \vec{w}) \cdot \vec{v}$.

[Go to solution]

Earlier, we implicitly assumed that P, Q, R were not collinear, but in general $\operatorname{orient}(P, Q, R, S)$ is zero if and only if P, Q, R, S are coplanar, so when any three points are collinear it is always zero. We can also say that it is nonzero if and only if lines PQ and RS are skew, that is, neither intersecting nor parallel.

Finally, $|\operatorname{orient}(P, Q, R, S)|$ is equal to six times the volume of tetrahedron $PQRS$.

It is implemented by simply writing down the definition:

```
T orient(p3 p, p3 q, p3 r, p3 s) {return (q-p)*(r-p)|(s-p);}
```

Exercise 6

A convenient way to check whether two lines PQ and RS are skew is to check whether $\left(\overrightarrow{PQ} \times \overrightarrow{RS}\right) \cdot \overrightarrow{PR} \neq 0$: in fact you can replace PR by any vector going from PQ to RS.
Using the properties of dot product, cross product and orient(), prove that

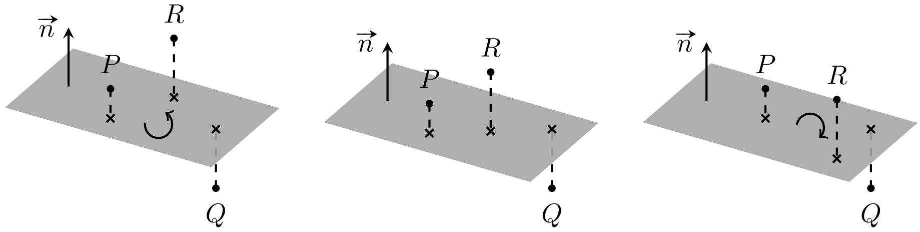
$$\left(\overrightarrow{PQ} \times \overrightarrow{RS}\right) \cdot \overrightarrow{PR} = -\operatorname{orient}(P, Q, R, S)$$

[Go to solution]

Let's say we have a plane Π and a vector $\vec{n}$ perpendicular to it (a *normal* to the plane). Then an interesting variant is to replace $\overrightarrow{PS}$ by $\vec{n}$, giving the expression

$$\left(\overrightarrow{PQ} \times \overrightarrow{PR}\right) \cdot \vec{n}$$

This is equivalent to computing the 2D $\operatorname{orient}(P', Q', R')$ on Π, where P', Q', R' are the projections of P, Q, R on Π.



left turn | aligned | right turn

$\left(\overrightarrow{PQ} \times \overrightarrow{PR}\right) \cdot \vec{n} > 0$ $\quad$ $\left(\overrightarrow{PQ} \times \overrightarrow{PR}\right) \cdot \vec{n} = 0$ $\quad$ $\left(\overrightarrow{PQ} \times \overrightarrow{PR}\right) \cdot \vec{n} < 0$

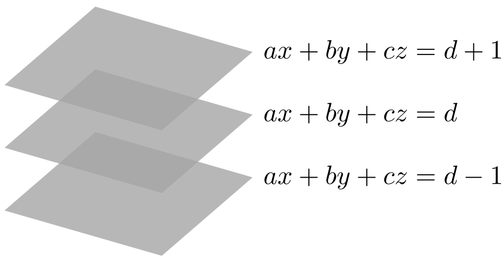
```
T orientByNormal(p3 p, p3 q, p3 r, p3 n) {return (q-p)*(r-p)|n;}
```

7.2 Planes

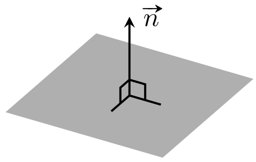
In this section we will discuss how to represent planes, and many ways we can use them. We will see that they play a very similar role to the role lines play in 2D, and many of the operations we defined in section 6.4 have a direct equivalent here. Since the explanations are nearly identical, we make them a bit shorter here; please refer to section 6.4 if you want more details.

7.2.1 Defining planes

Planes are sets of points (x, y, z) which obey an equation of the form $ax + by + cz = d$. Here, a, b, c determine the orientation of the plane, while d determines its position relative to the origin.



Vector $\vec{n} = (a, b, c)$ is perpendicular to the plane and is called a *normal* of the plane. The equation can be rewritten as $\vec{n} \cdot (x, y, z) = d$: that is, the plane is formed by all points whose dot product with $\vec{n}$ is equal to constant d. This makes sense, because we know that the dot product doesn't change when one vector (here, the point (x, y, z)) moves perpendicularly to the other (here, the normal $\vec{n}$).



Here are some other ways to define a plane, and how to find $\vec{n}$ and d in each case:

- if we know the normal $\vec{n}$ and a point P belonging to the plane: we can find d as $\vec{n} \cdot P$;
- if we know a point P and two (non-parallel) vectors $\vec{v}$ and $\vec{w}$ that are parallel to the plane: we can define $\vec{n} = \vec{v} \times \vec{w}$ then find d as above;
- if we know 3 non-collinear points P, Q, R on the plane: we first find two vectors $\vec{v} = \overrightarrow{PQ}$ and $\vec{w} = \overrightarrow{PR}$ that are parallel from the plane, then find $\vec{n}$ and d as above.

We implement this with the following structure and constructors:

```
struct plane {
    p3 n; T d;
    // From normal n and offset d
    plane(p3 n, T d) : n(n), d(d) {}
    // From normal n and point P
    plane(p3 n, p3 p) : n(n), d(n|p) {}
    // From three non-collinear points P,Q,R
    plane(p3 p, p3 q, p3 r) : plane((q-p)*(r-p), p) {}

    // Will be defined later:
    // - these work with T = int
    T side(p3 p);
    double dist(p3 p);
    plane translate(p3 t);
    // - these require T = double
    plane shiftUp(double dist);
    p3 proj(p3 p);
    p3 refl(p3 p);
};
```

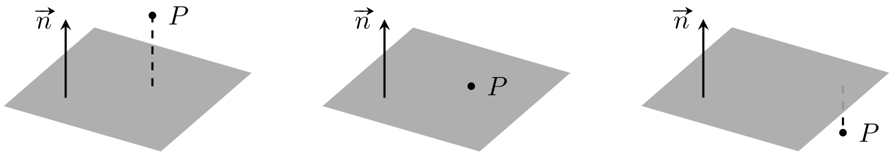
7.2.2 Side and distance

The first thing we are interested for a plane Π in is the value of $ax+by+cz-d$. If it's zero, the point is on Π, and otherwise it tells us which side of Π the point lies. We name it `side()`, like its 2D line equivalent:

```
T side(p3 p) {return (n|p)-d;}
```

which we will denote $\text{side}_\Pi(P)$.

$\text{side}_\Pi(P)$ is positive if P is on the side of Π pointed by $\vec{n}$, and negative for the other side. In fact, for given points P, Q, R, S, `plane(p,q,r).side(s)` gives the same value as `orient(p,q,r,s)`.



P above
$\text{side}_\Pi(P) = 3$

P on plane
$\text{side}_\Pi(P) = 0$

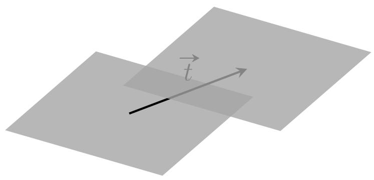
P below
$\text{side}_\Pi(P) = -2$

And just like the `side()` for 2D lines, we can get the distance from it, if we compensate for the norm of $\vec{n}$:

```
double dist(p3 p) {return abs(side(p))/abs(n);}
```

7.2.3 Translating a plane

If we translate a plane by a vector $\vec{t}$, the normal $\vec{n}$ remains unchanged, but the offset d changes.



To find the new value d', we use the same argument as for 2D lines: suppose point P is on the old plane, that is, $\vec{n} \cdot P = d$. Then $P + \vec{t}$ should be in the new plane:

$$d' = \vec{n} \cdot (P + \vec{t}) = \vec{n} \cdot P + \vec{n} \cdot \vec{t} = d + \vec{n} \cdot \vec{t}$$

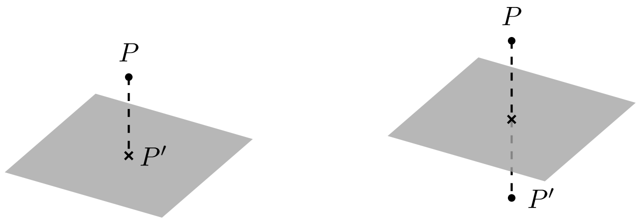
```
plane translate(p3 t) {return {n, d+(n|t)};}
```

And if we want to shift perpendicularly (in the direction of $\vec{n}$) by some distance δ, then $\vec{n} \cdot \vec{t}$ becomes $\delta\|\vec{n}\|$, which gives the following code:

```
plane shiftUp(double dist) {return {n, d + dist*abs(n)};}
```

7.2.4 Orthogonal projection and reflection

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



projection reflection

We use a similar reasoning as in section 6.4.7. Clearly, to go from P to its projection on Π, we need to move perpendicularly to Π, that is, we need to move by $k\vec{n}$ for some k, so that the resulting point $P + k\vec{n}$ is on Π.

From this we find k:

$$\begin{aligned} \vec{n} \cdot (P + k\vec{n}) = d &\Leftrightarrow \vec{n} \cdot P + k(\vec{n} \cdot \vec{n}) = d \\ &\Leftrightarrow k\|\vec{n}\|^2 = -(\vec{n} \cdot P - d) \\ &\Leftrightarrow k = -\frac{\mathrm{side}_\Pi(P)}{\|\vec{n}\|^2} \end{aligned}$$

And to find the reflection, we need to move P twice as far to $P + 2k\vec{n}$, so we get the following implementation for both:

```
p3 proj(p3 p) {return p - n*side(p)/sq(n);}
p3 refl(p3 p) {return p - n*2*side(p)/sq(n);}
```

7.2.5 Coordinate system based on a plane

When we have a plane Π, sometimes we will want to know what are the coordinates of a point in Π. That is, suppose we have a few points that we know are coplanar, and we want to use some 2D algorithm on them. How do we get the 2D coordinates of those points?

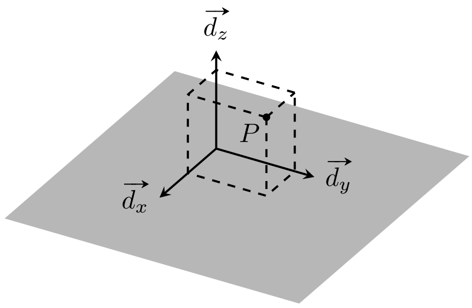
To do this, we need to chose an origin point O on Π and two vectors $\vec{d_x}$ and $\vec{d_y}$ indicating the desired x and y directions. Let's first assume $\vec{d_x}$ and $\vec{d_y}$ are perpendicular and their norms are 1. Then, to find the x- and

y-coordinate of a point P on Π, we simply need to compute

$$x = \overrightarrow{OP} \cdot \overrightarrow{d_x}$$
$$y = \overrightarrow{OP} \cdot \overrightarrow{d_y}$$

and if we also have vector $\overrightarrow{d_z} = \overrightarrow{d_x} \times \overrightarrow{d_y}$ perpendicular to the first two, we can find the "height" of P respective to Π:

$$z = \overrightarrow{OP} \cdot \overrightarrow{d_z}$$



If we have three non-collinear points P, Q, R that form plane Π, how can we choose $\overrightarrow{d_x}$, $\overrightarrow{d_y}$ and $\overrightarrow{d_z}$?

1. First, we choose $\overrightarrow{d_x}$ to be $\overrightarrow{PQ}$, then scale it to have a norm of 1.
2. Then we compute $\overrightarrow{d_z} = \overrightarrow{d_x} \times \overrightarrow{PR}$, wh and scaling it to have a norm of 1. It is perpendicular to Π because it is perpendicular to two non-parallel vectors in it.
3. Finally, we find $\overrightarrow{d_y}$ as $\overrightarrow{d_z} \times \overrightarrow{d_x}$.

This gives the following code. Method `pos2d()` gives the position on the plane as a 2D point, and method `pos3d()` gives the position and height as a 3D point (so in a way this structure represents a change of coordinate system).

```
struct coords {
    p3 o, dx, dy, dz;
    // From three points P,Q,R on the plane:
    // build an orthonormal 3D basis
    coords(p3 p, p3 q, p3 r) : o(p) {
        dx = unit(q-p);
        dz = unit(dx*(r-p));
        dy = dz*dx;
    }
    // From four points P,Q,R,S:
    // take directions PQ, PR, PS as is
    coords(p3 p, p3 q, p3 r, p3 s) :
```

```
        o(p), dx(q-p), dy(r-p), dz(s-p) {}

    pt pos2d(p3 p) {return {(p-o)|dx, (p-o)|dy};}
    p3 pos3d(p3 p) {return {(p-o)|dx, (p-o)|dy, (p-o)|dz};}
};
```

Math insight

The second constructor allows us specify points P, Q, R, S and choose $\overrightarrow{d_x} = \overrightarrow{PQ}$, $\overrightarrow{d_y} = \overrightarrow{PR}$ and $\overrightarrow{d_z} = \overrightarrow{PS}$ directly. This can be useful if we don't care that the 2D coordinate system $(\overrightarrow{d_x}, \overrightarrow{d_y})$ is not orthonormal (perpendicular and of norm 1), because it allows us to keep using integer coordinates.
If $\overrightarrow{d_x}$ and $\overrightarrow{d_y}$ are not perpendicular or do not have a norm of 1, the computed distances and angles will not be correct. But if we are only interested in the relative positions of lines and points, and finding the intersections of lines or segments, then everything works fine. Computing the 2D convex hull of a set of points is an example of such a problem, because it only requires that the sign of orient() is correct.

7.3 Lines

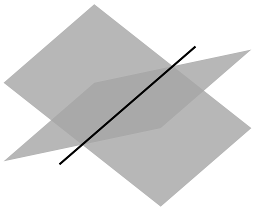
In this section we will discuss how to represent lines in 3D, and some related problems, including finding the line at the intersection of two planes, and finding the intersection point of a plane and a line.

7.3.1 Line representation

Unlike 2D lines and planes, 3D lines don't have a nice representation as a single equation on coordinates like $ax + by = c$ or $ax + by + cz = d$. We could represent them as the intersection of two planes, like

$$\begin{aligned} a_1x + b_1y + c_1z &= d_1 \\ a_2x + b_2y + c_2z &= d_2 \end{aligned}$$

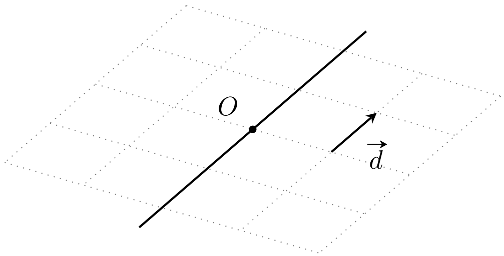
but this representation is not very convenient to work with, and there are many pairs planes we could choose.



Instead, we will work with a parametric representation: we take a point O on the line and a vector $\vec{d}$ parallel to the line and say that the points belonging to the line are all points

$$P = O + k\vec{d}$$

where k is a real parameter.



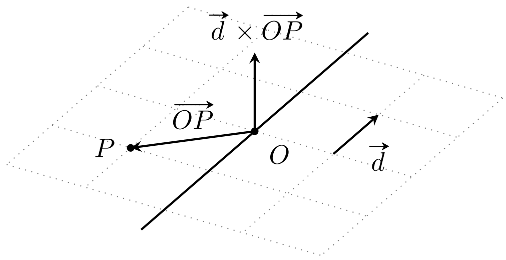
Note that here, $\vec{d}$ plays the same role as $\vec{v}$ did for 2D lines (see section 6.4.1).

If we are given two points P, Q on the line, we can set $O = P$ and $\vec{d} = \overrightarrow{PQ}$, so we implement the structure like this:

```
struct line3d {
    p3 d, o;
    // From two points P, Q
    line3d(p3 p, p3 q) : d(q-p), o(p) {}
    // From two planes p1, p2 (requires T = double)
    line3d(plane p1, plane p2); // will be implemented later

    // Will be defined later:
    // - these work with T = int
    double sqDist(p3 p);
    double dist(p3 p);
    bool cmpProj(p3 p, p3 q);
    // - these require T = double
    p3 proj(p3 p);
    p3 refl(p3 p);
    p3 inter(plane p) {return o - d*p.side(o)/(d|p.n);}
};
```

7.3.2 Distance from a line

A point P is on a line l described by $O, \vec{d}$ if and only if $\overrightarrow{OP}$ is parallel to $\vec{d}$. This is the case when $\vec{d} \times \overrightarrow{OP} = \vec{0}$.



More generally, this product $\vec{d} \times \overrightarrow{OP}$ also gives us information about the distance to the line: the distance from l to P is equal to

$$\frac{\|\vec{d} \times \overrightarrow{OP}\|}{\|\vec{d}\|}$$

```
double sqDist(p3 p) {return sq(d*(p-o))/sq(d);}
double dist(p3 p) {return sqrt(sqDist(p));}
```

7.3.3 Sorting along a line

Just like we did with 2D lines in section 6.4.4, we can sort points according to their position along a line l. To find out if a point P should come before another point Q, we simply need to check whether

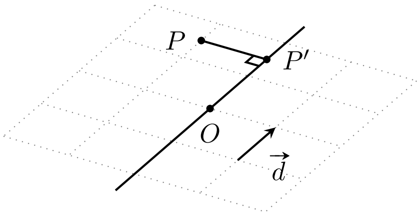
$$\vec{d} \cdot P < \vec{d} \cdot Q$$

so we can use the following comparator:

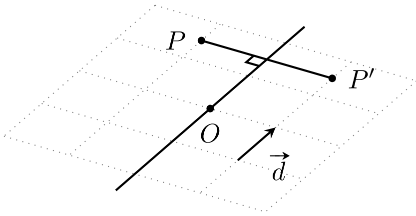
```
bool cmpProj(p3 p, p3 q) {return (d|p) < (d|q);}
```

7.3.4 Orthogonal projection and reflection

Let's say we want to project P on a line l, that is find the closest point to P on l. Our usual approach to projecting things, which is to move P perpendicularly until it touches l, doesn't work as well here: indeed, there are many possible directions that are perpendicular to l.



projection



reflection

Instead we will start from O and move along the line until we reach the projection of P. We have seen in the previous section that taking the dot product with $\vec{d}$ tells us how far some point is along l. In fact, if we compute $\vec{d} \cdot \overrightarrow{OP}$, this tells us the (signed) distance from O to the projection of P, multiplied by $\|\vec{d}\|$. So we can find the projection this way:

```
p3 proj(p3 p) {return o + d*(d|(a-o))/sq(d);}
```

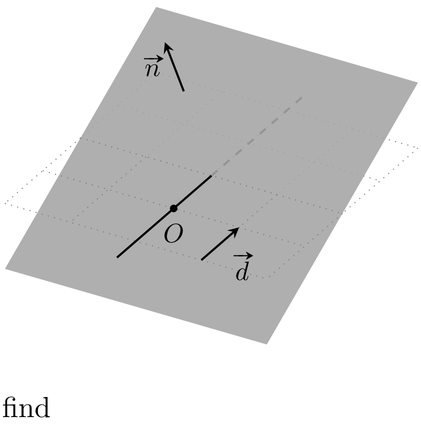
Once we've found the projection P', we can find the reflection P'' easily, since it is twice as far in the same direction: we have $P'' = P' + \overrightarrow{PP'}$, which becomes $2P' - P$ if we allow vector operations on points.

```
p3 refl(p3 p) {return proj(p)*2 - p;}
```

7.3.5 Plane-line intersection

Let's say we have a plane Π, represented by vector $\vec{n}$ and real d, and a line l, represented by point O and $\vec{d}$. To find an intersection between them, we need to find a point $O + k\vec{d}$ that lies on Π, that is, such that

$$\vec{n} \cdot \left(O + k\vec{d}\right) = d$$



Solving for k, we find

$$k = \frac{d - \vec{n} \cdot O}{\vec{n} \cdot \vec{d}} = \frac{-\operatorname{side}_\Pi(O)}{\vec{n} \cdot \vec{d}}$$

which can be implemented directly:

```
p3 inter(plane p) {return o - d*p.side(o)/(p.n|d);}
```

Note that this is undefined when $\vec{n} \cdot \vec{d} = 0$, that is, when Π and l are parallel.[2]

7.3.6 Plane-plane intersection

If we are given two non-parallel planes Π_1 and Π_2 (defined by $\vec{n_1}$, d_1 and $\vec{n_2}$, d_2), how can we find their common line?

First we need to find its direction $\vec{d}$. Clearly, the direction needs to be parallel to both planes, so it must be perpendicular to both $\vec{n_1}$ and $\vec{n_2}$. Thus we take $\vec{d} = \vec{n_1} \times \vec{n_2}$.

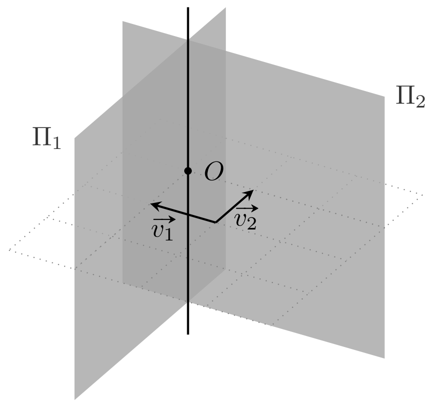
Then we need to find an arbitrary point O that is on both planes. Here, we will actually compute the closest such point to the origin. It is given by

$$O = \frac{\left(d_1\vec{n_2} - d_2\vec{n_1}\right) \times \vec{d}}{\|\vec{d}\|^2}$$

Let's analyze this expression. It is the sum of two vectors,

$$\vec{v_1} = \frac{d_1}{\|\vec{d}\|^2}\left(\vec{n_2} \times \vec{d}\right)$$
$$\vec{v_2} = \frac{d_2}{\|\vec{d}\|^2}\left(\vec{n_1} \times \vec{d}\right)$$

Because $\vec{v_1}$ is perpendicular to $\vec{n_2}$, it is parallel to Π_2, while $\vec{v_2}$ is perpendicular to $\vec{n_1}$ and thus parallel to Π_1. So we can see $\vec{v_1}$ as the vector that leads from the origin to Π_1 while staying parallel to Π_2, and $\vec{v_2}$ as the vector that leads from the origin to Π_2 while staying parallel to Π_1.



[2] We take a closer look at criteria for parallelism and perpendicularity in section 7.4.

Let's verify that O is on Π_1, that is, $\overrightarrow{n_1} \cdot O = \overrightarrow{n_1} \cdot (\overrightarrow{v_1} + \overrightarrow{v_2}) = d_1$. Since $\overrightarrow{v_2}$ is perpendicular to $\overrightarrow{n_1}$, clearly $\overrightarrow{n_1} \cdot \overrightarrow{v_2} = 0$. What remains is

$$\begin{aligned}
\overrightarrow{n_1} \cdot \overrightarrow{v_1} &= \frac{d_1}{\|\vec{d}\|^2} \left(\overrightarrow{n_2} \times \vec{d}\right) \cdot \overrightarrow{n_1} \\
&= \frac{d_1}{\|\vec{d}\|^2} \left(\overrightarrow{n_1} \times \overrightarrow{n_2}\right) \cdot \vec{d} \\
&= \frac{d_1}{\|\vec{d}\|^2} \left(\vec{d} \cdot \vec{d}\right) \\
&= d_1
\end{aligned}$$

where between the first two lines, we used the fact that the mixed product is conserved if we swap its arguments in a circular way (see exercise 5).

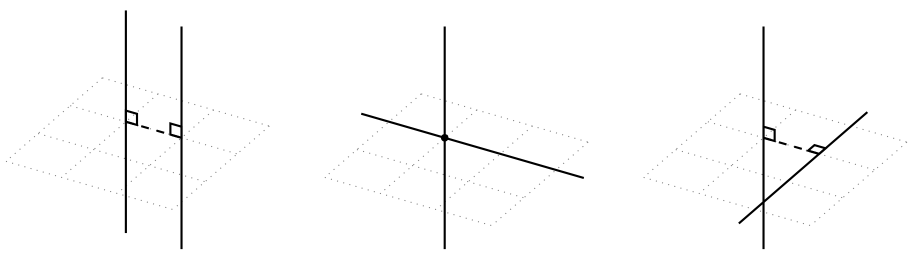
We prove similarly that O is on Π_2. Finally we note that both $\overrightarrow{v_1}$ and $\overrightarrow{v_2}$ are perpendicular to $\vec{d}$, so the vector from the origin to O is perpencular to $\vec{d}$, the direction of the line. Therefore it must necessarily arrive on the line at the closest point to the origin.

We can implement this as the following constructor:

```
line3d(plane p1, plane p2) {
    d = p1.n*p2.n;
    o = (p2.n*p1.d - p1.n*p2.d)*d/sq(d);
}
```

7.3.7 Line-line distance and nearest points

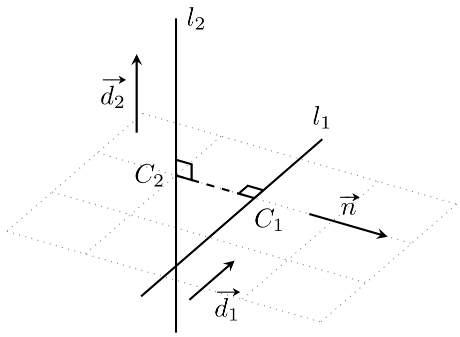
Consider line l_1 defined by $O_1 + k\overrightarrow{d_1}$ and line l_2 defined by $O_2 + k\overrightarrow{d_2}$. If they are parallel, the distance between them is easy to find: just find the distance from l_1 to O_2. Otherwise, they are either intersecting or skew, in which case the question is a bit more complex.



parallel intersecting skew

Let's call C_1 the point of l_1 that is closest to l_2, and C_2 the point on l_2 that is closest to l_1. Direction $\overrightarrow{C_1C_2}$ should be perpendicular to both l_1 and

l_2. Indeed, if it were not, it would be possible to get a smaller distance by moving either C_1 or C_2. So $\overrightarrow{C_1C_2}$ is parallel to $\vec{n} = \vec{v_1} \times \vec{v_2}$.



Because of this, we could compute the distance as

$$|C_1C_2| = \frac{\left|\overrightarrow{C_1C_2} \cdot \vec{n}\right|}{\|\vec{n}\|}$$

We don't know C_1 or C_2 yet, but since the dot product doesn't change when one of the vectors moves perpendicular to the other, we can move C_1 to O_1 and C_2 to O_2, so that we get

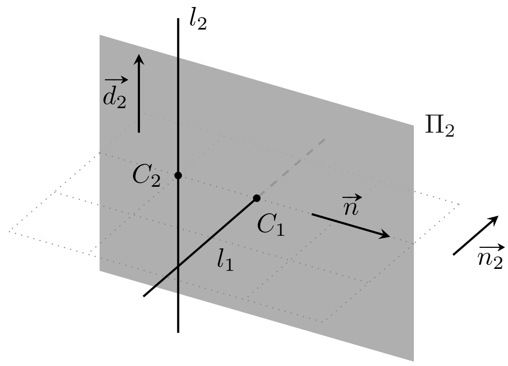
$$\frac{\left|\overrightarrow{C_1C_2} \cdot \vec{n}\right|}{\|\vec{n}\|} = \frac{\left|\overrightarrow{O_1C_2} \cdot \vec{n}\right|}{\|\vec{n}\|} = \frac{\left|\overrightarrow{O_1O_2} \cdot \vec{n}\right|}{\|\vec{n}\|}$$

which gives the following implementation:

```
double dist(line l1, line l2) {
    p3 n = l1.d*l2.d;
    if (n == zero) // parallel
        return l1.dist(l2.o);
    return abs((l2.o-l1.o)|n)/abs(n);
}
```

Now, how can we find C_1 and C_2 themselves? Let's call Π_2 the plane that contains l_2 and is parallel to n_2; its normal is $\vec{n_2} = \vec{d_2} \times \vec{n}$. Then C_1 is the intersection between that plane and line l_1, so we can use the formula obtained in section 7.3.5 to find it:

$$C_1 = O_1 + \frac{\overrightarrow{O_1O_2} \cdot \vec{n_2}}{\vec{d_1} \cdot \vec{n_2}}$$



This is implemented in a straightforward way. If we want C_2 instead we just have to swap l_1 and l_2.

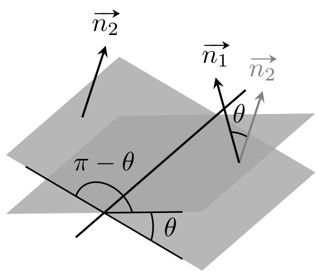
```
p3 closestOnL1(line l1, line l2) {
    p3 n2 = l2.d*(l1.d*l2.d);
    return l1.o + l1.d*((l2.o-l1.o)|n2)/(l1.d|n2);
}
```

7.4 Angles between planes and lines

In this section, we figure out how to check whether lines and planes are parallel or perpendicular, and then how to find the line perpendicular to a plane going through a given point, and vice versa. All those operations are very easy with our representation of planes and lines.

7.4.1 Between two planes

The angle between two planes is equal to the angle between their normals. Since usually two angles of distinct amplitudes θ and $\pi - \theta$ are formed, we take the smaller of the two, in $[0, \frac{\pi}{2}]$.



We can find it with the following code. We take the minimum with 1 to avoid `nan` in case of imprecisions.

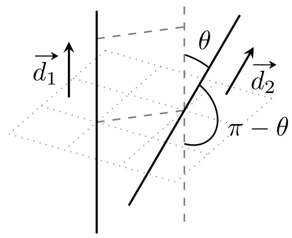
```
double smallAngle(p3 v, p3 w) {
    return acos(min(abs(v|w)/abs(v)/abs(w), 1.0));
}
double angle(plane p1, plane p2) {
    return smallAngle(p1.n, p2.n);
}
```

In particular, we can check whether two planes are parallel/perpendicular by checking if their normals are parallel/perpendicular:

```
bool isParallel(plane p1, plane p2) {
    return p1.n*p2.n == zero;
}
bool isPerpendicular(plane p1, plane p2) {
    return (p1.n|p2.n) == 0;
}
```

7.4.2 Between two lines

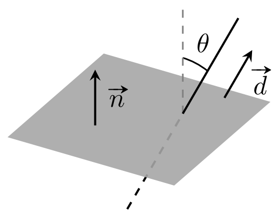
The situation with lines is exactly the same: their angle is equal to the angle between their direction vectors. Note that the lines aren't necessarily in the same plane, so the angle is taken as if they were moved until they touch.




```
double angle(line3d l1, line3d l2) {
    return smallAngle(l1.p, l2.d);
}
bool isParallel(line3d l1, line3d l2) {
    return l1.d*l2.d == zero;
}
bool isPerpendicular(line3d l1, line3d l2) {
    return (l1.d|l2.d) == 0;
}
```

7.4.3 Between a plane and a line

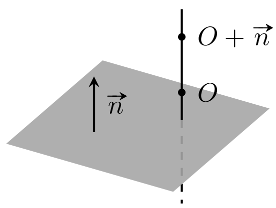
The situation when considering a plane and a line is a bit different. Let's consider a plane Π of normal $\vec{n}$ and a line l of direction vector $\vec{d}$. When they are perpendicular, $\vec{n}$ is parallel to $\vec{d}$, and inversely when they are parallel, $\vec{n}$ is perpendicular to $\vec{d}$. In general, if the angle between $\vec{n}$ and $\vec{d}$ is $\theta \in [0, \frac{\pi}{2}]$, then the angle between the plane and the line is $\frac{\pi}{2} - \theta$.




```
double angle(plane p, line3d l) {
    return M_PI/2 - smallAngle(p.n, l.d);
}
bool isParallel(plane p, line3d l) {
    return (p.n|l.d) == 0;
}
bool isPerpendicular(plane p, line3d l) {
    return p.n*l.d == zero;
}
```

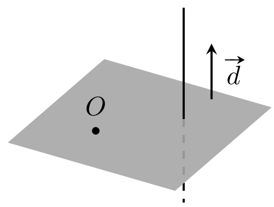
7.4.4 Perpendicular through a point

The line perpendicular to a plane Π of normal $\vec{n}$ and going through a point O is simply the line going through O and whose direction vector is $\vec{n}$, or equivalently the line going through O and $O + \vec{n}$.




```
line3d perpThrough(plane p, p3 o) {return line(o, o+p.n);}
```

The plane perpendicular to a line l of direction vector $\vec{d}$ and going through a point O is simply the plane containing O and whose normal is $\vec{d}$.




```
plane perpThrough(line3d l, p3 o) {return plane(l.d, o);}
```

7.5 Polyhedrons

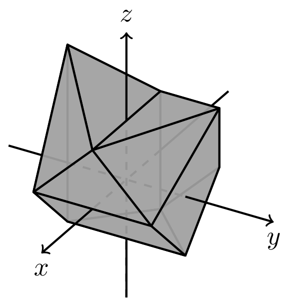
In this section, we give a brief introduction to polyhedrons, and show how to compute their surface area and their volume.

7.5.1 Definition

A polyhedron is a region of space delimited by polygonal faces. Generally, we will describe a polyhedron by listing its faces. Some basic conditions apply:

- all the faces are polygons that don't self-intersect;
- two faces either share a complete edge, or share a single vertex, or have no common point;
- all edges are shared by exactly two faces;
- if we define adjacent faces as faces that share an edge, all faces are connected together.

Here is a polyhedron that respects those conditions:

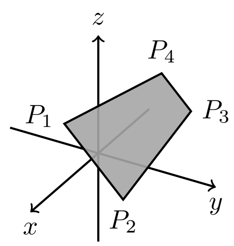


Face type	*Face visibility*
8 triangles	7 visible
4 quadrilaterals	5 hidden

Note that those conditions don't exclude nonconvex polyhedrons (like the example above), but they do exclude self-crossing polyhedrons.

7.5.2 Surface area

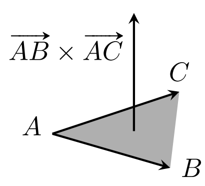
To compute the surface area of a polyhedron, we need to compute the area of their faces. Like we do for polygons, we denote a face by listing its vertices in order, like $P_1P_2P_3P_4$. The vertices must all be coplanar and the edges should not intersect except at their ends.



How do we find the area of a face $P_1 \cdots P_n$? First let's take the most simple case: a triangle ABC. If we compute cross product $\overrightarrow{AB} \times \overrightarrow{AC}$, its direction is perpendicular to the triangle and its norm is

$$|AB|\,|AC|\sin\theta$$

where θ is the amplitude of angle between $\overrightarrow{AB}$ and $\overrightarrow{AC}$. This is twice the area of triangle ABC, as was already noted in section 6.6.1. So the area of triangle ABC is $\|\overrightarrow{AB} \times \overrightarrow{AC}\|$

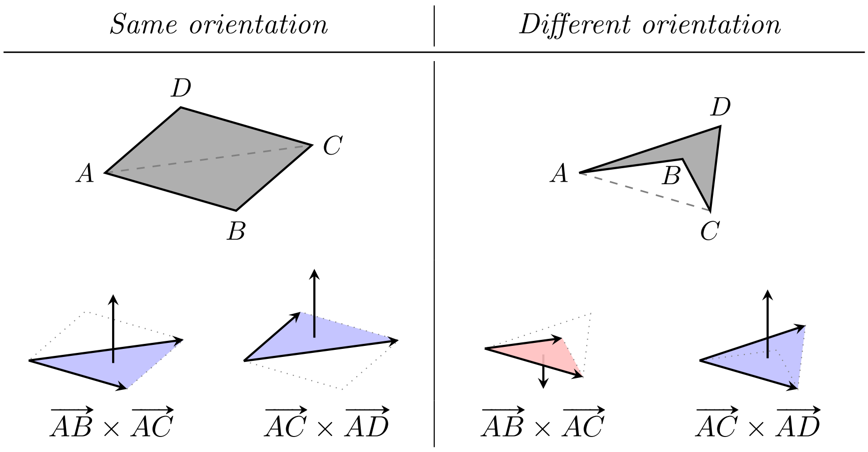


If we rewrite the vectors we can obtain a more symmetric expression:

$$\begin{aligned}\overrightarrow{AB} \times \overrightarrow{AC} &= (B - A) \times (C - A) \\ &= B \times C - B \times A - A \times C + A \times A \\ &= A \times B + B \times C + C \times A\end{aligned}$$

Let's extend this to a quadrilateral $ABCD$. There are two cases:

- Triangles ABC and ACD are oriented in the same way. In this case, vectors $\overrightarrow{AB} \times \overrightarrow{AC}$ and $\overrightarrow{AC} \times \overrightarrow{AD}$ point in the same direction, and the areas should be added together. So we take the vector sum $\overrightarrow{AB} \times \overrightarrow{AC} + \overrightarrow{AC} \times \overrightarrow{AD}$.
- Triangles ABC and ACD are oriented in different ways (the angle at either B or D is concave). In this case, vectors $\overrightarrow{AB} \times \overrightarrow{AC}$ and $\overrightarrow{AC} \times \overrightarrow{AD}$ point in opposite directions, and we should take the difference of the areas. So again, taking the vector sum $\overrightarrow{AB} \times \overrightarrow{AC} + \overrightarrow{AC} \times \overrightarrow{AD}$ will produce the desired effect.



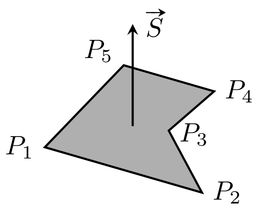
We can reformulate the sum in the same way:

$$\begin{aligned}\overrightarrow{AB} \times \overrightarrow{AC} + \overrightarrow{AC} \times \overrightarrow{AD} &= (A \times B + B \times C + C \times A) \\ &\quad + (A \times C + C \times D + D \times A) \\ &= A \times B + B \times C + C \times D + D \times A\end{aligned}$$

If we continue with more and more vertices, we can make the following general conclusion. Given a polygon $P_1 \cdots P_n$, we can compute the *vector area*

$$\vec{S} = \frac{P_1 \times P_2 + P_2 \times P_3 + \cdots + P_n \times P_1}{2}$$

This vector is perpendicular to the polygon, and $\|\vec{S}\|$ gives its area. It is oriented such that when it points towards the observer, the vertices are numbered in counter-clockwise order.

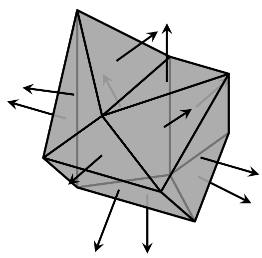


Note that this is very similar to the 2D formula for area we obtained in section 6.6.1. The only thing that changes is that the result is a vector. We can implement this straightforwardly:

```
p3 vectorArea2(vector<p3> p) { // vector area * 2 (to avoid
    divisions)
    p3 S = zero;
    for (int i = 0, n = p.size(); i < n; i++)
        S = S + p[i]*p[(i+1)%n];
    return S;
}
double area(vector<p3> p) {
    return abs(vectorArea2(p)) / 2.0;
}
```

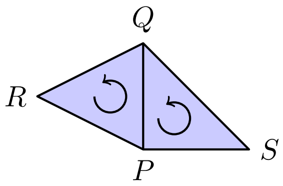
7.5.3 Face orientation

When given an arbitrary polyhedron, an important task is to orient the faces properly, so that we know which side is inside the polyhedron and which side is outside. In particular, we will try to order the vertices of the faces so that their vector areas $\vec{S}$ all points towards the outside of the polyhedron. Note that, depending on the way the polyhedron was obtained, this might already be the case.

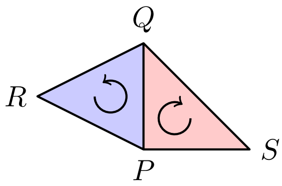


$\vec{S}$ pointing outside (lengths not to scale)

While it's not clear when looking at individual faces which side is inside the polyhedron, it's easy to deduce the correct orientation by looking at the orientation of an adjacent face. If two faces share an edge $[PQ]$, and the first face lists P then Q in this order, then the other face should list them in the other order, so that they "rotate" in the same direction. Note that because of circularity, in $P_1 \cdots P_n$, P_n is considered to come before P_1, not after.



$[P, Q, R]$ and $[S, Q, P]$
$P \to Q$ vs $Q \to P$
OK



$[P, Q, R]$ and $[P, Q, S]$
both $P \to Q$
KO

So to orient the faces in a consistent way, start from an arbitrary face, and perform a graph traversal on faces. Whenever you go from a face to a neighboring face, reverse the new face's vertex order if the common edge is present in the same order on both faces. This will either orient all vector areas towards the outside, or all towards the inside.

Here is a example implementation:

```
// Create arbitrary comparator for map<>
bool operator<(p3 p, p3 q) {
    return tie(p.x, p.y, p.z) < tie(q.x, q.y, q.z);
}
struct edge {
    int v;
    bool same; // = is the common edge in the same order?
};

// Given a series of faces (lists of points), reverse some of them
// so that their orientations are consistent
void reorient(vector<vector<p3>> &fs) {
    int n = fs.size();

    // Find the common edges and create the resulting graph
    vector<vector<edge>> g(n);
    map<pair<p3,p3>,int> es;
    for (int u = 0; u < n; u++) {
        for (int i = 0, m = fs[u].size(); i < m; i++) {
            p3 a = fs[u][i], b = fs[u][(i+1)%m];
            // Let's look at edge [AB]
            if (es.count({a,b})) { // seen in same order
                int v = es[{a,b}];
                g[u].push_back({v,true});
                g[v].push_back({u,true});
            } else if (es.count({b,a})) { // seen in different order
```

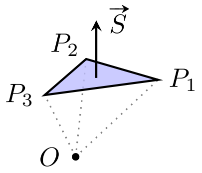
```
                int v = es[{b,a}];
                g[u].push_back({v,false});
                g[v].push_back({u,false});
            } else { // not seen yet
                es[{a,b}] = u;
            }
        }
    }

    // Perform BFS to find which faces should be flipped
    vector<bool> vis(n,false), flip(n);
    flip[0] = false;
    queue<int> q;
    q.push(0);
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        for (edge e : g[u]) {
            if (!vis[e.v]) {
                vis[e.v] = true;
                // If the edge was in the same order,
                // exactly one of the two should be flipped
                flip[e.v] = (flip[u] ^ e.same);
                q.push(e.v);
            }
        }
    }

    // Actually perform the flips
    for (int u = 0; u < n; u++)
        if (flip[u])
            reverse(fs[u].begin(), fs[u].end());
}
```

7.5.4 Volume

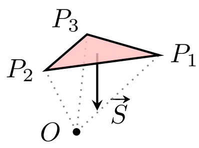
Suppose that the faces are oriented correctly. We will show how to compute the volume of the polyhedron by taking the same approach as in section 6.6.1 for the 2D polygon area.

Let's choose an arbitrary reference point O. We will compute the volume of the polyhedron face by face: for a face $P_1 \cdots P_n$, if the side of the face seen from O is inside the polygon, add the volume of pyramid $OP_1 \cdots P_n$, and otherwise subtract it. That way, by inclusion and exclusion, the final result will be the volume inside the polyhedron.

Since $\vec{S}$ always points towards the outside of the polygon, we just have to check whether $\vec{S}$ points away from O (add) or towards O (subtract).



$\vec{S}$ away from O
$+$ volume of $OP_1P_2P_3$



$\vec{S}$ towards O
$-$ volume of $OP_1P_2P_3$

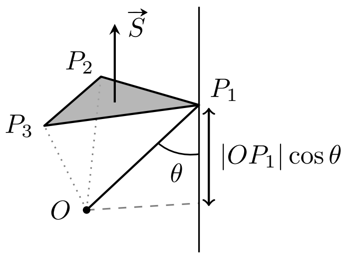
How do we compute the volume of pyramid $OP_1 \cdots P_n$? We can use the formula

$$\text{volume} = \frac{\text{area of base} \times \text{height}}{3}$$

It turns out that dot product $\vec{S} \cdot \overrightarrow{OP_1}$ computes exactly (area of base $\times$ height) in absolute value. Indeed, by definition

$$\vec{S} \cdot \overrightarrow{OP_1} = \left\|\vec{S}\right\| |OP_1| \cos\theta$$

where θ is the angle between $\vec{S}$ and $\overrightarrow{OP_1}$. The norm $\left\|\vec{S}\right\|$ is the area of $P_1 \cdots P_n$, the base of the pyramid, and we can easily see that $|OP_1| \cos\theta$ is the height of the pyramid (up to sign).



So the absolute value of $\vec{S} \cdot \overrightarrow{OP_1}$ is equal to the volume of pyramid $OP_1 \cdots P_n$, and the dot product is positive if $\vec{S}$ points away from O, and negative if $\vec{S}$ points towards O. This is exactly the sign we want.

For the implementation, we take O to be the origin for convenience. We divide by 6 at the end because we need to divide by 2 to get the correct area and then by 3 because of the formula for the volume of a pyramid.

```
double volume(vector<vector<p3>> fs) {
    double vol6 = 0.0;
    for (vector<p3> f : fs)
        vol6 += (vectorArea2(f)|f[0]);
    return abs(vol6) / 6.0;
}
```

In case the vector areas $\vec{S}$ all point towards the inside of the polyhedron (which may happen after applying the face orientation procedure in the previous section), `vol6` will be correct but will have a negative sign. If this happens, flip all the faces so that all $\vec{S}$ now point outside.

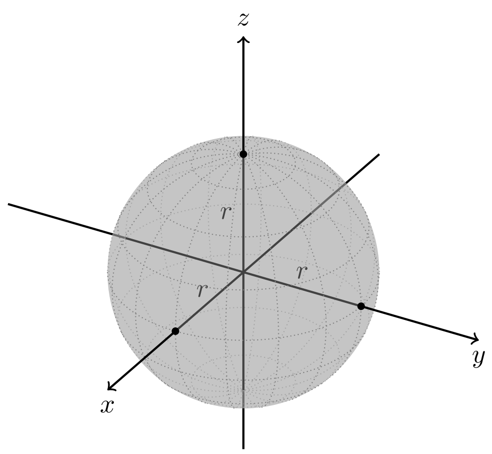
7.6 Spherical geometry

In this section, we look at spheres and how we can define distances, segments, intersections and areas on spheres. We then define the related concept of 3D angles and how we can compute a 3D version of the winding number based on spherical geometry primitives.

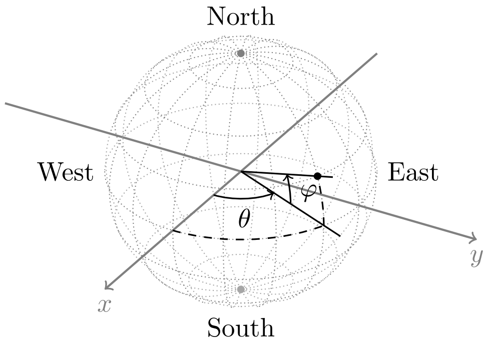
7.6.1 Spherical coordinate system

A sphere with center $O = (x_0, y_0, z_0)$ and radius r is the set of points at distance exactly r from O. We can describe it this by equation

$$(x - x_0)^2 + (y - y_0)^2 + (z - z_0)^2 = r^2$$



To describe a point on a sphere, we can either directly give its coordinates x, y, z or use the spherical coordinate system: this is the system used to position locations on Earth. We use an angle φ, the latitude, which tells us how far North the point is (or South, if $\varphi < 0$), and an angle λ, the longitude, which tells us how far East the point is (or West, if $\lambda < 0$). We usually take $\varphi \in [-\frac{\pi}{2}, \frac{\pi}{2}]$ and $\lambda \in (-\pi, \pi]$.



If the sphere is centered at the origin, the position represented by coordinates (φ, λ) is

$$(r\cos\varphi\cos\lambda, r\cos\varphi\sin\lambda, r\sin\varphi)$$

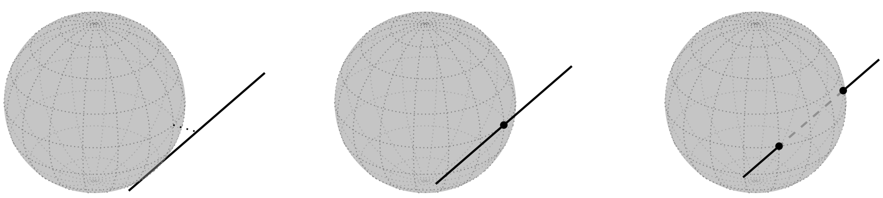
where the z-axis points North and the x-axis points towards meridian $\lambda = 0$ (on Earth, the Greenwich meridian).

The function below finds the position given angles in degrees:

```
p3 sph(double r, double lat, double lon) {
    lat *= M_PI/180, lon *= M_PI/180;
    return {r*cos(lat)*cos(lon), r*cos(lat)*sin(lon), r*sin(lat)};
}
```

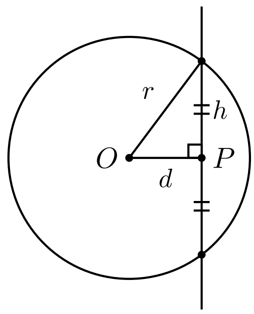
7.6.2 Sphere-line intersection

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



0 intersections　　1 intersection　　2 intersections

Finding them is exactly like circle-line intersection: first compute the projection P of the center onto the line, then find the intersections by moving the appropriate distance forward or backward along l.

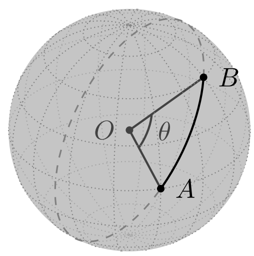


This function returns the number of intersection points and places them in pair `out` if they exist.

```
int sphereLine(p3 o, double r, line3d l, pair<p3,p3> &out) {
    double h2 = r*r - l.sqDist(o);
    if (h2 < 0) return 0; // the line doesn’t touch the sphere
    p3 p = l.proj(o); // point P
    p3 h = l.d*sqrt(h2)/abs(l.d); // vector parallel to l, of length
        h
    out = {p-h, p+h};
    return 1 + (h2 > 0);
}
```

7.6.3 Great-circle distance

The shortest distance between two points A and B on a sphere (O, r) is given by travelling along plane OAB. It is called the *great-circle distance* because it follows the circumference of one of the big circles of radius r that split the sphere in two.

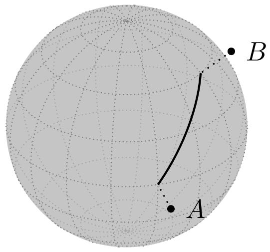


So computing the distance between A and B amounts to finding the length of the circle arc joining them. This arc is subtended by angle θ, the angle between $\overrightarrow{OA}$ and $\overrightarrow{OB}$, so its length is simply $r\theta$.

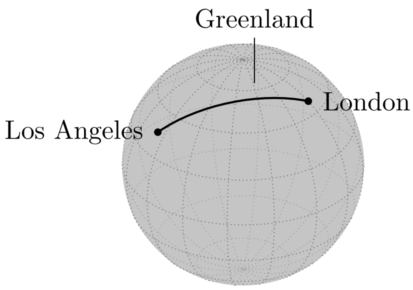
```
double greatCircleDist(p3 o, double r, p3 a, p3 b) {
    return r * angle(a-o, b-o);
```

```
}
```

This code also works if A and B are not actually on the sphere, in which case it will give the distance between their projections on the sphere:



Note that in most 2D projections this great-circle path is not a straight line, and it tends to bend northward in the Northern Hemisphere, and southward in the Southern Hemisphere. This is why, for example, flights going from London to Los Angeles fly over Greenland, although it is much further North than both cities.



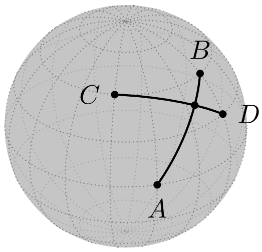
7.6.4 Spherical segment intersection

For points A and B on a sphere, we define spherical segment $[AB]$ as the path drawn by the great-circle distance between A and B on the sphere. This is not well-defined if A and B are directly opposite each other on the sphere, because there would be many possible shortest paths.

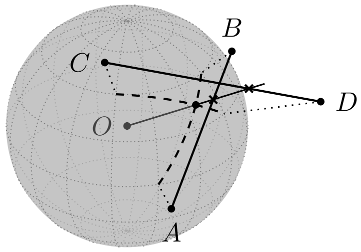
From simplicity, we assume that the sphere is centered at the origin. We can test if a segment is valid using the following function. Note that we accept segments where $A = B$.

```
bool validSegment(p3 p, p3 q) {
    return p*q != zero || (p|q) > 0;
}
```

Given two spherical segments $[AB]$ and $[CD]$, we would like to figure out if they intersect, and what their intersection is. This is part of a more general problem: given two segments $[AB]$ and $[CD]$ in space, if we view them from an observation point O, does one of them hide part of the other, that is, is there a ray from O that touches them both?



common point on sphere



common ray from O

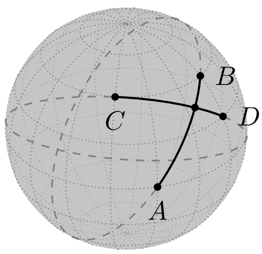
We will solve the general problem, and make sure that our answer is always exact when points A, B, C, D are integer points. To do this, we will separate cases just like we did for 2D segment intersection in section 6.5.2:

1. Segments $[AB]$ and $[CD]$ intersect *properly*, that is, their intersection is one single point which is not an endpoint of either segment. For the general problem this means that there is a single ray from O that touches both $[AB]$ and $[CD]$, and it doesn't touch any of A, B, C, D.
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.

For the rest of explanation, we will consider the case of spherical segment intersection, because it's easier to visualize, but it should be easy to verify that this also applise to the general problem.

Proper intersection

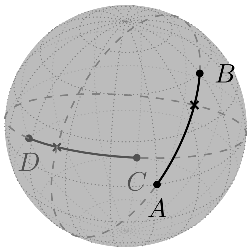
Let's deal with the first case: there is a single proper intersection point I. For this to be the case, A and B must be on either side of plane OCD, and C and D must be on either side of plane OAB. Put another way, A and B must be on either side of the great circle containing C and D, and vice versa.



We can check this with mixed product. We have to verify that

- $o_A = (C \times D) \cdot A$ and $o_B = (C \times D) \cdot B$ have opposite signs;
- $o_C = (A \times B) \cdot C$ and $o_D = (A \times B) \cdot D$ have opposite signs.

However, this time it's not enough. Sometimes, even though the conditions above are verified, there is no intersection, because the segments are on opposite sides of the sphere:



C and D are on the other side of the sphere

To eliminate this kind of case, we also have to check that o_A and o_C have opposite signs (this is clearly not the case here).

Exercise 7

Consider a few more examples and verify that these criteria correctly detect proper intersections in all cases.

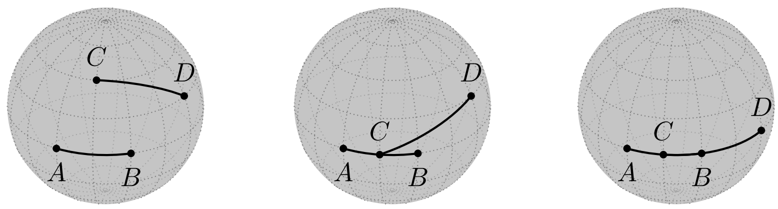
The intersection point I must be in the intersection of planes OAB and OCD. So direction $\overrightarrow{OI}$ must be perpendicular to their normals $A \times B$ and $C \times D$, that is, parallel to $(A \times B) \times (C \times D)$. Multiplying this by the sign of o_D gives the correct direction.

This is implemented by the code below. Note that the result, `out`, only gives the direction of the intersection. If we want to find the intersection on the sphere, we need to scale it to have length r.

```
bool properInter(p3 a, p3 b, p3 c, p3 d, p3 &out) {
    p3 ab = a*b, cd = c*d; // normals of planes OAB and OCD
    int oa = sgn(cd|a),
        ob = sgn(cd|b),
        oc = sgn(ab|c),
        od = sgn(ab|d);
    out = ab*cd*od; // four multiplications => careful with overflow
        !
    return (oa != ob && oc != od && oa != oc);
}
```

Improper intersections

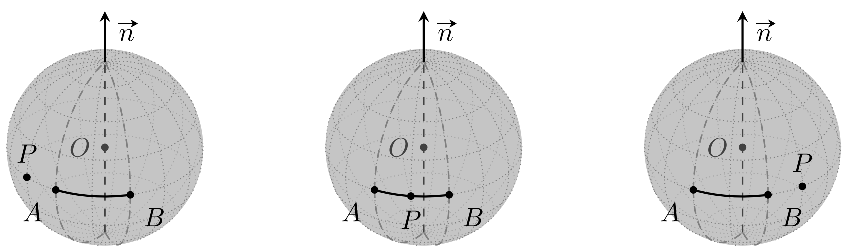
To deal with the second case, we will do as for 2D segments and test for every point among A, B, C, D if it is on the other segment. If it is, we add it to a set S. S will contain 0, 1, or 2 distinct points, describing an empty intersection, a single intersection point or an intersection segment.



$S = \emptyset$ no intersection | $S = \{C\}$ intersection point | $S = \{B, C\}$ intersection segment

To check whether a point P is on spherical segment $[AB]$, we need to check that P is on plane OAB, but also that P is is "between" rays $[OA)$ and $[OB)$ on that plane.

Let $\vec{n} = A \times B$, a normal of plane OAB. Checking that P is on plane OAB is easy: $\vec{n} \cdot P$ should be 0. If P is indeed on plane OAB, then we can check if it is to the "right" of $[OA)$ by looking at cross product $A \times X$. It should be perpendicular to plane OAB. If it is in the same direction as $\vec{n}$, then P is to the right of OA, and if it is in the opposite direction, then P is to the left of OA. So we need to check $\vec{n} \cdot (A \times P) \geq 0$. Similarly, to check that P is to the left of OB, we should check $\vec{n}(B \times X) \leq 0$.



left of $[OA)$ KO | between $[OA)$ and $[OB)$ OK | right of $[OB)$ KO

There remains only one special case: if A and B are the same point on the sphere, then $\vec{n} = \vec{0}$, and then we should just check that P is also that same point.

We arrive at the following implementation. To handle the general problem, instead of directly checking for equality between P and A or B, we check that they are in the same direction with the cross product.

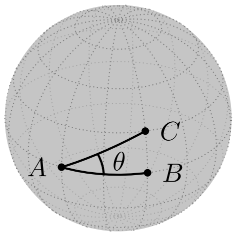
```
bool onSphSegment(p3 a, p3 b, p3 p) {
    p3 n = a*b;
    if (n == zero)
        return a*p == zero && (a|p) > 0;
    return (n|p) == 0 && (n|a*p) >= 0 && (n|b*p) <= 0;
}
```

Now we just have to put all of this together in one function. First we check for a proper intersection, then if there is none we check segment by segment and add the points to set S. Since (as mentioned) we can't check for equality directly, we use a custom set structure that checks if the cross product is zero for every point already in the set.

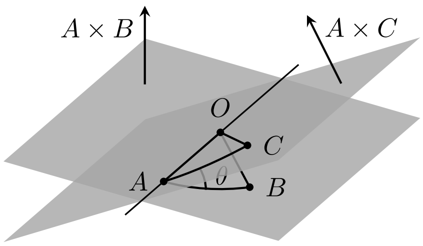
```
struct directionSet : vector<p3> {
    using vector::vector; // import constructors
    void insert(p3 p) {
        for (p3 q : *this) if (p*q == zero) return;
        push_back(p);
    }
};
directionSet intersSph(p3 a, p3 b, p3 c, p3 d) {
    assert(validSegment(a, b) && validSegment(c, d));
    p3 out;
    if (properInter(a, b, c, d, out)) return {out};
    directionSet s;
    if (onSphSegment(c, d, a)) s.insert(a);
    if (onSphSegment(c, d, b)) s.insert(b);
    if (onSphSegment(a, b, c)) s.insert(c);
    if (onSphSegment(a, b, d)) s.insert(d);
    return s;
}
```

7.6.5 Angles on a sphere

Given two spherical segments $[AB]$ and $[AC]$ on a sphere around the origin O, how do we compute the amplitude of the angle they form on the surface of the sphere at A? This angle is equal to the angle between planes OAB and OAC, or more precisely between their normals $A \times B$ and $A \times C$. Thus we can find the angle using `angle()`, which gives values in $[0, \pi]$.



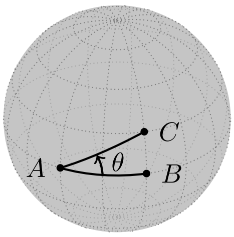
angle ABC



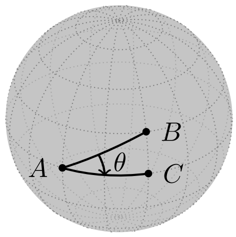
planes OAB, OAC

```
double angleSph(p3 a, p3 b, p3 c) {
    return angle(a*b, a*c);
}
```

If instead of values in $[0, \pi]$, we want to know the oriented angle[3] between $[AB]$ and $[AC]$, that is, how much we rotate if we go from B to C around A counterclockwise, then we need to know on which side of plane OAB point C lies. If C lies "to the left" of plane OAB, the angle given by `angle()` is correct, but if C lies "to the right", we should subtract it from 2π.



`angleSph(a,b,c)` $\approx 30°$
`orientedAngleSph(a,b,c)` $\approx 30°$



`angleSph(a,b,c)` $\approx 30°$
`orientedAngleSph(a,b,c)` $\approx 330°$

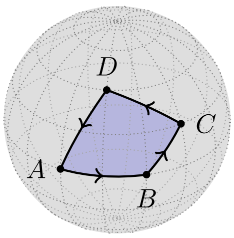
```
double orientedAngleSph(p3 a, p3 b, p3 c) {
    if ((a*b|c) >= 0)
        return angleSph(a, b, c);
    else
        return 2*M_PI - angleSph(a, b, c);
}
```

7.6.6 Spherical polygons and area

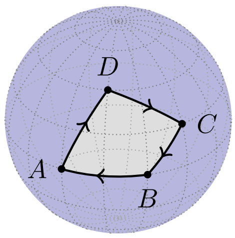
Given points $P_1, \ldots, P_n$ on a sphere, let's call spherical polygon $P_1 \cdots P_n$ the region on the sphere delimited by spherical segments $[P_1P_2]$, $[P_2P_3]$, ...,

[3]We defined a similar notion in 2D in section 6.3.2.

$[P_nP_1]$, and which is on the left when travelling from P_1 to P_2. The counter-clockwise order is important here because both "sides" on the contour are valid candidates.



area of $ABCD$



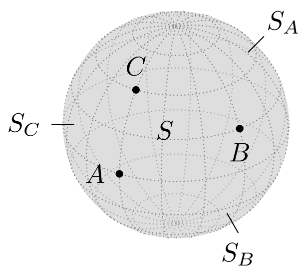
area of $DCBA$

Computing the area of such a spherical polygon is surprisingly simple. First, let's consider the case of a spherical triangle ABC. It's area is given by

$$r^2(\alpha + \beta + \gamma - \pi)$$

where r is the radius of the sphere and α, β, γ are the amplitudes of the three interior angles of ABC. Note that this would be equal to 0 if ABC, but because of the curvature of the sphere, the angles of a triangle actually add up to more than π.

This is actually pretty easy to prove. If we prolong the segments $[AB]$, $[BC]$, and $[CA]$ into their full great-circles, this splits the sphere into 8 parts, of which four are the direct image of each other by point reflection around the center of the sphere. Let's call S the area of triangle ABC, and S_A (resp. S_B, S_C the area of the triangle on the other side of $[BC]$ (resp. $[CA], [AB]$).



Since those four triangles cover half of the sphere together, we have $S + S_A + S_B + S_C = 2\pi r^2$. Besides if we take triangle ABC and the triangle on the other side of $[BC]$, together they form a whole *spherical wedge*[4] of angle α. So their combined area should be $\alpha/2\pi$ of the total area, that is $S + S_A = 2\alpha r^2$. Similarly, we obtain $S + S_B = 2\beta r^2$ and $S + S_C = 2\gamma r^2$.

[4] A section of a sphere determined by two flat cuts going through the center. `https://en.wikipedia.org/wiki/Spherical_wedge`

Combining those four equations, we obtain

$$\begin{aligned}2S &= (S + S_A) + (S + S_B) + (S + S_C) - (S + S_A + S_B + S_C)\\ &= r^2(2\alpha + 2\beta + 2\gamma - 2\pi)\end{aligned}$$

which is the desired result.

The formula can be extended to an arbitrary spherical polygon $P_1 \cdots P_n$. The area is given by

$$r^2 \left[\text{sum of interior angles} - (n-2)\pi\right]$$

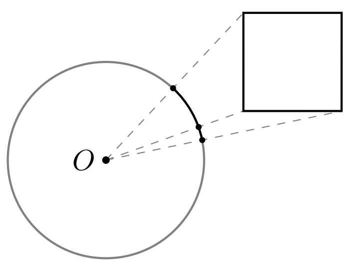
This can be proven by decomposing the n-gon into $n-2$ triangles.

```
double areaOnSphere(double r, vector<p3> p) {
    int n = p.size();
    double sum = -(n-2)*M_PI;
    for (int i = 0; i < n; i++)
        sum += orientedAngleSph(p[(i+1)%n], p[(i+2)%n], p[i]);
    return r*r*sum;
}
```

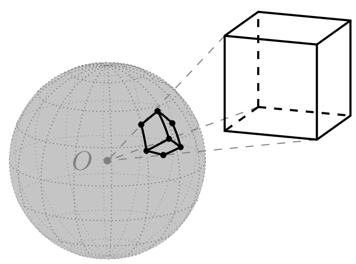
7.6.7 Solid angle

The *solid angle* subtended by an object at an observation point O is the apparent size of the object when looking at it from O. For example, the Sun and the Moon have roughly the same apparent size when watched from the Earth, even though their actual sizes are very different. So we say that they subtend the same solid angle at the observation point that is Earth.

Let's define it more precisely. Just like the planar angle subtended by an object is the length of the object once it is projected onto a unit circle around the point, the solid angle subtended by an object is the area of the object once it is projected onto a unit sphere around the point.



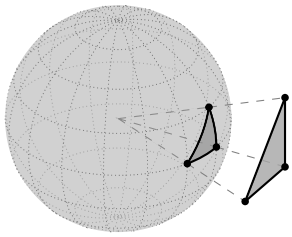
planar angle



solid angle

The unit for solid angles is the steradian (sr), and because the area of a unit sphere is 4π, a solid angle of 4π means that the observation point is completely surrounded, while a solid angle of 2π means that half of the view is covered.

We can easily find the solid angle subtended by a polygon by using the function `areaOnSphere()` we just defined and setting $r = 1$. Indeed, all it does is compute angles, so the distance from the origin O doesn't matter.



This also allows us to find the solid angle subtended by a polyhedron if we know which faces are visible from O.

Math insight

The solid angle subtended by a small surface $d\vec{S}$ at a position $\vec{r}$ is inversely proportional to the square of the distance from the origin, $\|\vec{r}\|^2$, and proportional to the cosine of the angle between $\vec{r}$ and $d\vec{S}$, because a surface seen from sideways occupies less of the view. So we can also find solid angles with the following integral:

$$\Omega = \int \frac{\vec{r} \times d\vec{S}}{\|\vec{r}\|^3}$$

7.6.8 3D winding number

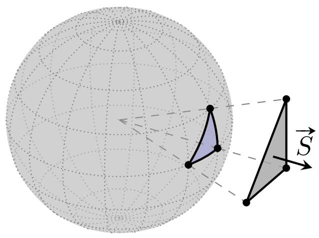
We can use this notion of solid angle to implement a 3D version of winding number that we defined in section 6.6.3. Given an observation point O and a polyhedron, we will compute an integer that is

- 0 if O is outside the polyhedron;
- 1 if O is inside the polyhedron, and the vector areas $\vec{S}$ of the faces are oriented towards the outside;
- -1 if O is inside the polyhedron, and the vector areas $\vec{S}$ of the faces are oriented towards the inside.

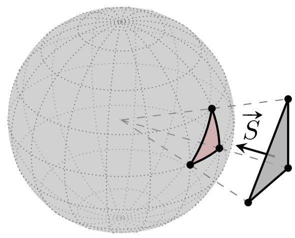
To do this, we will consider the faces one by one. For each one, if its vector area $\vec{S}$ points away from O, we add the solid angle it subtends to the total, and otherwise we subtract it. That way:

- if O is outside of the polyhedron, the solid angles will cancel out;
- if O is inside of the polyhedron and the $\vec{S}$ point towards the outside, there will mostly be additions and the total will add up to 4π;
- if O is inside the polyhedron and the $\vec{S}$ point towards the inside, there will mostly be subtractions and the total will add up to -4π.

Dividing this total angle by 4π gives the desired winding number.



`areaOnSphere()` ≈ 0.079



`areaOnSphere()` ≈ 12.487
`areaOnSphere()` $- 4\pi \approx -0.079$

To find what quantity to add or subtract for each face, we will use function `areaOnSphere()` directly. If $\vec{S}$ points away from O, it will return a value in $(0, 2\pi)$, the area of the projection of the face a unit sphere, which we should keep. If $\vec{S}$ points towards O, it will return a value in $(2\pi, 4\pi)$, the area of the *rest* of the unit sphere, so we should remove 4π to get the subtraction we want.

Since we always want the value to be in $(-2\pi, 2\pi)$ in the end, we can use function `remainder()`, giving the simple implementation below.

```
int windingNumber3D(vector<vector<p3>> fs) {
    double sum = 0;
    for (vector<p3> f : fs)
        sum += remainder(areaOnSphere(1, f), 4*M_PI);
    return round(sum / (4*M_PI));
}
```

Appendix A

Solutions to the exercises

Exercise 1

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.

$$
\begin{aligned}
&(r_1 \operatorname{cis} \varphi_1) * (r_2 \operatorname{cis} \varphi_2) \\
&\qquad = (r_1 \cos \varphi_1 + (r_1 \sin \varphi_1) i) * (r_2 \cos \varphi_2 + (r_2 \sin \varphi_2) i) \\
&\qquad = r_1 r_2 [(\cos \varphi_1 \cos \varphi_2 - \sin \varphi_1 \sin \varphi_2) \\
&\qquad\qquad + (\cos \varphi_1 \sin \varphi_2 + \sin \varphi_1 \cos \varphi_2) i] \\
&\qquad = r_1 r_2 (\cos(\varphi_1 + \varphi_2) + i \sin(\varphi_1 + \varphi_2)) \\
&\qquad = r_1 r_2 \operatorname{cis}(\varphi_1 + \varphi_2)
\end{aligned}
$$

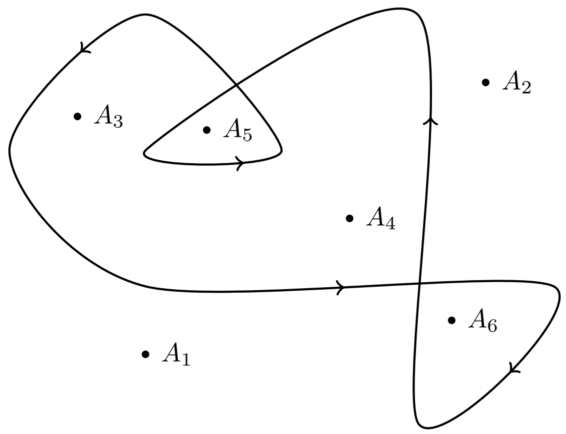
[Back to exercise]

Exercise 3

What value will `areaPolygon()` (section 6.6.1) 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.

It will give the sum of the areas of the parts delimited by the curve, multiplied by their corresponding winding numbers. For the curve below, it will give the sum of:

- $1 \times$ the area of the part containing A_3 and A_4;
- $2 \times$ the area of the part containing A_5;
- $-1 \times$ the area of the part containing A_6.



[Back to exercise]

Exercise 5

Show that those properties also apply to the mixed product. For example, $(\vec{u} \times \vec{v}) \cdot \vec{w} = (\vec{v} \times \vec{w}) \cdot \vec{u}$ and $(\vec{u} \times \vec{v}) \cdot \vec{w} = -(\vec{u} \times \vec{w}) \cdot \vec{v}$.

This can be derived from the properties of orient(P, Q, R, S) by setting $P = \vec{0}$, $Q = \vec{u}$, $R = \vec{v}$ and $S = \vec{w}$. Indeed, in this case

$$\begin{aligned}
\text{orient}(P, Q, R, S) &= \left(\overrightarrow{PQ} \times \overrightarrow{PR}\right) \cdot \overrightarrow{PS} \\
&= \left[\left(\vec{u} - \vec{0}\right) \times \left(\vec{v} - \vec{0}\right)\right] \cdot \left(\vec{w} - \vec{0}\right) \\
&= (\vec{u} \times \vec{v}) \cdot \vec{w}
\end{aligned}$$

[Back to exercise]

Exercise 6

A convenient way to check whether two lines PQ and RS are skew is to check whether $\left(\overrightarrow{PQ} \times \overrightarrow{RS}\right) \cdot \overrightarrow{PR} \neq 0$: in fact you can replace PR by any vector going from PQ to RS.

Using the properties of dot product, cross product and orient(), prove that

$$\left(\overrightarrow{PQ} \times \overrightarrow{RS}\right) \cdot \overrightarrow{PR} = -\,\text{orient}(P, Q, R, S)$$

Note that for any vectors $\vec{v}$ and $\vec{w}$ we have $(\vec{v} \times \vec{w}) \cdot \vec{w} = 0$. This is because $\vec{v} \times \vec{w}$ is perpendicular to $\vec{w}$ and the dot product is zero for perpendicular vectors.

We develop:

$$\begin{aligned}
\left(\overrightarrow{PQ} \times \overrightarrow{RS}\right) \cdot \overrightarrow{PR} &= \left(\overrightarrow{PQ} \times \left(\overrightarrow{PS} - \overrightarrow{PR}\right)\right) \cdot \overrightarrow{PR} \\
&= \left(\overrightarrow{PQ} \times \overrightarrow{PS} - \overrightarrow{PQ} \times \overrightarrow{PR}\right) \cdot \overrightarrow{PR} \\
&= \left(\overrightarrow{PQ} \times \overrightarrow{PS}\right) \cdot \overrightarrow{PR} - \left(\overrightarrow{PQ} \times \overrightarrow{PR}\right) \cdot \overrightarrow{PR} \\
&= \left(\overrightarrow{PQ} \times \overrightarrow{PS}\right) \cdot \overrightarrow{PR} \\
&= \operatorname{orient}(P, Q, S, R) \\
&= -\operatorname{orient}(P, Q, R, S)
\end{aligned}$$

[Back to exercise]

Appendix B

Omitted proofs

B.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

$$\begin{aligned} f(a) &\leq \frac{b}{a+b} f(0) + \frac{a}{a+b} f(a+b) \\ f(b) &\leq \frac{a}{a+b} f(0) + \frac{b}{a+b} f(a+b). \end{aligned}$$

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* $(a * b, \mathrm{round}(a' * b'), d, n_a + n_b + 1)$ *is a valid computation, then Theorem 1 holds for it as well.*

Proof. From the hypotheses know that

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

We find

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

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* $(ab, \mathrm{round}(a'b'), d_a + d_b, n_a + n_b + 1)$ *is a valid computation, then Theorem 1 holds for it as well.*

Proof. From the hypotheses we know that

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

We find

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

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] Benjamin A. Burton and Mathias Hiron. "Creating informatics olympiad tasks: exploring the black art". In: *Olympiads in Informatics* (2008). ISSN: 1822-7732.

[2] Gyula Horváth and Tom Verhoeff. "Numerical difficulties in pre-university informatics education and competitions". In: *Informatics in Education* (2003).

[3] 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`.

[4] Antti Laaksonen. *Guide to Competitive Programming*. Springer International Publishing, 2017. DOI: `10.1007/978-3-319-72547-5`.

[5] "The International Olympiad in Informatics Syllabus". In: (). URL: `https://people.ksp.sk/~misof/ioi-syllabus/`.

Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgium **www.uclouvain.be/epl**