Geometry and Algebra in the Algorithms Lab
Lecture Notes HS 2020

Michael Hoffmann

Incomplete Draft R6908 from Wednesday 30th September, 2020

Contents

Chapter 1

Introduction

Numbers are fundamental to represent data, and operations on numbers are fundamental building blocks of many computations and algorithms. When working with numbers on an abstract level, we often consider them as elements of $\mathbb{N}$ (the set of natural numbers), $\mathbb{Z}$ (the set of integers), $\mathbb{Q}$ (the set of rational numbers), $\mathbb{R}$ (the set of real numbers), or $\mathbb{C}$ (the set of complex numbers). All of these sets are infinite, which is very convenient in an abstract setting.

But the downside of such powerful abstractions is that we cannot directly transfer them to practice because real-world computers work with a finite amount of resources. To begin with, using a finite number of states only a finite number of different numbers can be represented. But in addition to storage limitations also the computation time can be affected because explicitly computing with larger numbers requires computing with more digits, which requires more time.

Therefore, in somewhat realistic models of computation[1] we assume that basic arithmetic operations can be done in constant time (unit cost model), but only for numbers within a certain fixed range. For instance, considering a typical computer in today's world, a reasonable choice would be numbers that can be represented with up to 64 bits.

There are many scenarios where it is easy to see a priori that all numbers that we can possibly encounter during the computations fit into such a fixed range. Then there is no real need to worry about the differences between infinite abstraction and finite practice. Not incidentally, most—if not all—of the problems you have encountered in the course up to this point are of that nature.

However, there are also many scenarios where these differences do matter and disregarding them can lead to spectacular failures. To cite just one very prominent example: On June 4, 1996, the first flight of an Ariane 5 type spacecraft ended after less than a minute with an explosion, destroying material worth several hundred million euros (it is an unmanned vehicle, so luckily nobody was injured or killed in the accident). According to the inquiry board report [8], the main reason was a "... data conversion from 64-bit floating point to 16-bit signed integer value. The floating point number which was con-

[1] such as the popular Word RAM model (see, for instance, the survey by Hagerup [5])

verted had a value greater than what could be represented by a 16-bit signed integer." While it seems obvious that such a conversion cannot work in general, apparently it did happen to work for the predecessor Ariane 4 where the code originated from. But Ariane 5 operates with a much higher velocity and so the numbers were larger than before and did not fit into the target data type anymore.

The Ariane example is just one out of several similar incidents and probably many more that are not publicly known because the consequences were not as dramatic and also because failures are not necessarily advertized if it can be avoided[2]. It just goes to illustrate that in every implementation of an algorithm we need to take care

- that the computations that are done internally are executed correctly,
- that the results of these computations are stored and interpreted correctly, and
- that all assumptions and restrictions are documented clearly.

As a Master of Computer Science, you are expected to have internalized the three items listed above and, of course, to be able to put them into practice. How to do this is the first main focus of this week.

To illustrate, recognize and then overcome possible challenges with computations, we need to consider some nontrivial computations that go beyond addition or comparison of two integers. At the same time, we want to keep things reasonably simple, as our focus here is not on numerical or algebraic computation. Therefore we will use blackbox tools to handle the underlying algebra and numerics for us: exact algebraic number types. Your job is to recognize and decide when to use a tool and when you are better off without—because as is often the case with powerful tools: they come with a cost.

To gain access to a large variety of problems that involve simple but nontrivial arithmetic computations, we consider Euclidean geometry in the plane $\mathbb{R}^2$. Euclidean spaces are a very important model for all kinds of aspects of our physical world, and the two-dimensional setting has the additional advantage that it is easy to think about and draw examples. Furthermore, everybody has ample experience with two-dimensional geometry from highschool. The underlying linear algebra is also familiar from highschool and from the first year course here at ETH. Finally, geometric problems often encompass combinatorial algorithmic aspects (such as sorting, searching and geometrically defined graphs) as well. One particular example of a geometric problem that is also familiar from the first year course "Algorithmen & Wahrscheinlichkeit" here at ETH is the two-dimensional convex hull problem, which we will get back to soon.

In this course, our focus is not on the details of how to implement geometric algorithms, such as computing convex hulls or intersections of geometric objects. Hence, we will use a blackbox tool that provides these algorithms. Our second main focus of this week is to get familiar with this tool, the CGAL library, and how to use it to solve certain geometric problems.

[2]The latter observation is somewhat unfortunate because often failures provide much better opportunities to learn from than success stories.

These notes comprise two chapters, which correspond to our two main foci this week, as described above. In the first chapter, we discuss representations of numbers and how to ensure correct results of computations, both using limited precision arithmetic and using exact algebraic computation. In the second chapter, we introduce some basic data types, operations and algorithms for two-dimensional Euclidean geometry provided by CGAL and discuss how to use them.

1.1 Outline

When working with any kind of data (here we work with numbers mostly), we need to choose an appropriate datatype for representation and computation. To begin with, all possible inputs must be representable. Also whatever computations we want to do on the data must be possible in an efficient and ideally convenient manner, and the results of these computations must be representable (to the extent they are needed). In this context, we advise you to not routinely and blindly trust limited precision arithmetic. In scenarios where limited precision arithmetic does not suffice to guarantee correct results, use exact symbolic algebra instead—in our case, as provided by CGAL and GMP [4]. Finally, we discuss the conceptual difference between predicates and constructions and their impact on the performance of the implementation of an algorithm. This impact is very noticeable in some scenarios and, therefore, should be taken into account in the modeling and algorithm design phases already.

1.2 Goals: What do we expect from you after this week?

You are acutely aware of challenges that arise from the differences between an abstract model that allows computation with arbitrary numbers and a real-world computer, which can represent a limited range of numbers only. These challenges apply to both correctness and efficiency of code.

Specifically, whenever reading input data, you take care to choose an appropriate data type so that all possible (according to the specification) input values can be represented and the input handling is sufficiently efficient. Similarly, whenever writing output data you take care that the (possibly complicated) internal representation is efficiently converted to the correct limited precision output, according to the specification. Whenever there are nontrivial computations inside the algorithm, you ensure that all these computations are done correctly and efficiently, by working with an appropriate datatype.

For reading input, the choices for an "appropriate" datatype are among the builtin C++ types `char`, `int`, and `long` only. For internal computations and output, also the field types of the three CGAL default kernels, as well as the types `CGAL::GMPz` and `CGAL::GMPq` are to be considered.

To make these decisions, you know some properties—in particular, the value range—of the datatypes listed above. You also understand the concept of floating point representations and are fluent in the corresponding terminology (mantissa, exponent, nor-

malization, binary representation)—both abstractly and specifically for IEEE 754 double precision. For simple computations (involving the basic arithmetic operations addition, subtraction, multiplication, and division) you know how to obtain simple bounds on the size (in terms of number of bits) of the numbers that arise during the computation.

You are also aware that a unit cost assumption for (arithmetic) operations is only realistic if the operands are from a small fixed range (basically if the operands fit into a machine word). As a consequence, you aim to formulate computations so as to stay within this range, without loosing accuracy, wherever possible. In particular, you try to avoid unnecessary divisions and squareroot computations.

You understand and can explain the conceptual difference between predicates and constructions, and what are the benefits of avoiding constructions. As a consequence, you actively look for ways to solve problems using predicates only wherever possible—even if it involves, for instance, a reformulation or tranformation of the input or output or doing the internal calculations in a slightly different way.

Chapter 2

Numbers: Representation & Computation

2.1 Choosing input datatypes

Let us start with the easiest item, which has been mentioned in "A Short Introduction to C++ for the Algorithms Lab" already. All input numbers in this course are integral, and our guideline is really easy.

Guideline 1 (Reading numbers). If it fits into an `int`, read it as in `int`; otherwise read it as a `long`.

To make the decision, you need to know what is the value range of `int`. Maybe surprisingly, this is not standardized but depends on the platform. However, what really matters for you is what is the value range on Code Expert. So let us simply repeat the corresponding information from "A Short Introduction to C++ for the Algorithms Lab".

Type specifier	Standard	Code Expert	Min Integer	Max Integer
`int`	$\geqslant 16$ bits	32 bits	-2^{31}	$2^{31}-1$
`long`	$\geqslant 32$ bits	64 bits	-2^{63}	$2^{63}-1$

Table 2.1: *Value range of the builtin C++ number types `int` and `long`.*

The reason to even distinguish these two cases is that—at least on Code Expert—reading an `int` is slightly faster than reading a `long`. To check, for instance, if `int` can represent at least 2^{25}, you can use the following code.

```
#include <limits>
#include <stdexcept>

if (std::numeric_limits<int>::max() < 33554432.0)
  throw std::runtime_error("max(int) < 2^(25)");
```

2.2 Layers of algorithms

Almost all algorithms require some computation. For instance, an algorithm to sort a given sequence of numbers may compare two numbers. Technically such a comparison is a very simple computation: It amounts to a subtraction followed by a test against zero, and there is not much that can go wrong with it. But there are other problems and corresponding algorithms—in this course and even more so in real-world problems—that call for more involved computations, for example,

- computing the Euclidean distance of two points,
- solving a linear system of equations (in two dimensions this corresponds to computing the intersection of two lines), or
- computing the orientation of a simplex (that is, of $d + 1$ points in $\mathbb{R}^d$; in two dimensions this corresponds to the orientation of a point triple).

If we want to argue that an implementation of an algorithm is correct, we better also argue that all internal computations are done correctly. However, the more involved these computations are and the more they are spread out over the algorithm, the more difficult it is to establish and verify such a claim.

A common way to address this problem is to take all algebraic/arithmetic computations out of the algorithm and collect them in a layer underneath. As an interface between both layers acts a collection of *elementary operations*. The algorithm on the top layer interacts with the input data only in terms of these elementary operations. The bottom layer in turn provides and implements them. As a result, correctness can be verified for both layers independently. With such a separation, most—if not all of the—differences between the theoretical model of computation and the practical implementation using limited resources are encapsulated into the layer of elementary operations.

Finding an ideal—small and simple on the one hand and complete on the other hand—set of elementary operations may be a challenge in itself and is tightly connected to the design of the algorithm. For instance, for the problem of sorting a sequence of numbers a suitable and well established elementary operation is the pairwise comparison already mentioned above. Many sorting algorithms are formulated based on comparisons. But there are also sorting algorithms, such as Counting Sort or Bucket Sort, that work in a different model of computation and are not formulated in terms of pairwise comparisons. For more details concerning everything related to sorting mentioned in this section, refer, for example, to the textbook by Cormen et al. [2, Chapter 8].

2.3 Decision trees

Intuitively it seems clear that a correct realization of the elementary operations is necessary for the overall correctness of the algorithm. To make this observation a bit more

explicit, let us combine the two-layer model described above with the well-known decision tree model. This model should be familiar to you from lower bound arguments for comparison-based sorting, which you must have encountered at some point during your Bachelor studies; see Mehlhorn [9, Chapter II.1.6] for details[1].

In the decision tree model, we regard an algorithm as a rooted tree whose leaves correspond to the possible results. Any execution of the algorithm for a specific input traces a root-to-leaf path in this tree. Each internal vertex v of the tree corresponds to a branch (`if` in C++ terminology) with an associated computation that determines in which of the children of v the execution continues. In our combined model here, each such computation is based on an elementary operation on the input.

For many interesting algorithms such a tree would be huge and therefore very messy to draw explicitly. To illustrate the concept, let us consider a very simple algorithm to compute a minimum of three numbers, as shown in Figure 2.1.

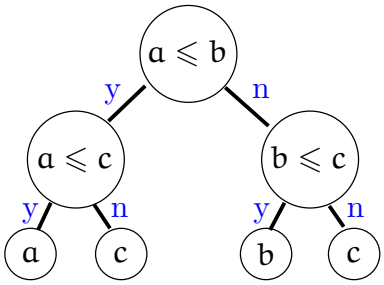


Figure 2.1: *The computation tree corresponding to an algorithm to compute a minimum of three input numbers* a, b, c.

Exercise 1. *Draw a computation tree for an algorithm to compute both minimum and maximum of four input numbers* a, b, c, d. *So at each leaf there is a pair of two numbers, the first of which is a minimum of all four and the second is a maximum of all four.*

In a similar fashion we can also visualize and analyze algorithms for comparison-based sorting of n numbers by a binary tree. Each internal vertex corresponds to a comparison of two input numbers, and each leaf contains a (sorted) sequence of the n input numbers. (This is the structure used to prove lower bounds for sorting.) With growing n the corresponding computation tree gets huge because it has $n!$ (the number of different total orders/permutations of n elements) leaves.

The main point for us here is that the computations (elementary operations) at those internal vertices drive the algorithm. If all these computations are done correctly, then all decisions are made correctly, and the algorithm is correct overall. If, however, these computations may give incorrect results that lead to wrong decisions, then all hell breaks

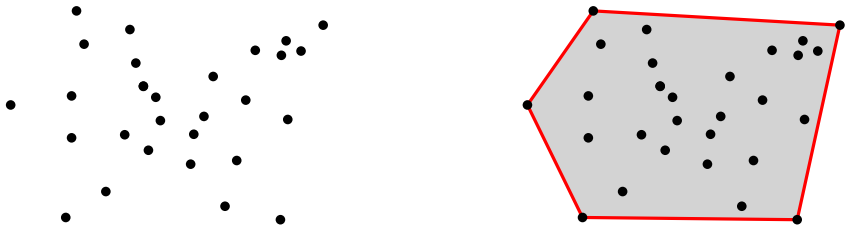
[1] If you are interested, you will also find some information here about the more general *algebraic computation tree model*, which we will not touch upon in this course

loose—even if the rest of the algorithm (that is, the top layer) is entirely correct. Already in the simple example from Figure 2.1 you can easily see that the overall result may be invalidated by any single wrong comparison.

So you may ask: "Why should such a computation not be correct? It should be possible to implement it correctly, shouldn't it?" Indeed, it should, and we expect no less of you. However, let us see why this is not always as straightforward as it may seem at a first glance. Of course, the issue is what we have alluded to in the introduction of this chapter: limited precision arithmetic. For the minimum computation or the sorting algorithms discussed above this is not much of a problem because the computation that drives these algorithms is a simple comparison. However, there are other algorithms where more involved computations are used on the lower layer to implement the elementary operations. We will discuss such an example in the next section.

2.4 Example: Planar Convex Hull

In the planar convex hull problem we are given a finite set P of points in $\mathbb{R}^2$ and want to compute the smallest convex[2] polygon that contains P. Intuitively, think of the points as nails that stick out a piece of wood (the plane); the convex hull you get be putting a rubber band loosely around all nails and then pulling the rubber band tight; see Figure 2.2 for an example. For more details about convex hulls, refer, for instance, to de Berg et al. [1, Chapter 11] or Cormen et al. [2, Chapter 33.3].



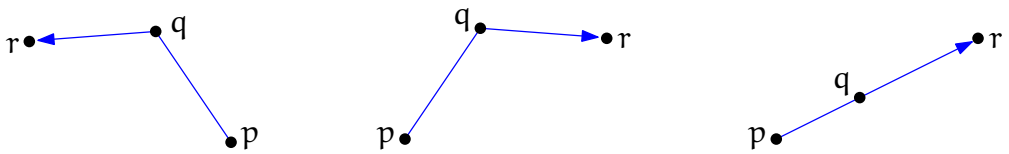
(a) Input set P **(b)** Convex hull of P

Figure 2.2: *Planar convex hull.*

There are many different algorithms to compute convex hulls. Many of them can be formulated in terms of the following two elementary operations.

- *lexicographic comparison* of two points: given two points $p = (p_x, p_y)$ and $q = (q_x, q_y)$, is $p_x < q_x$ or both $p_x = q_x$ and $p_y < q_y$?
- *orientation test*: given a triple p, q, r of points, does it form a leftturn, a rightturn, or are the points collinear? (The triple p, q, r forms a leftturn if $p \neq q$ and r is strictly to the left of the oriented line pq; see Figure 2.3 for illustration.)

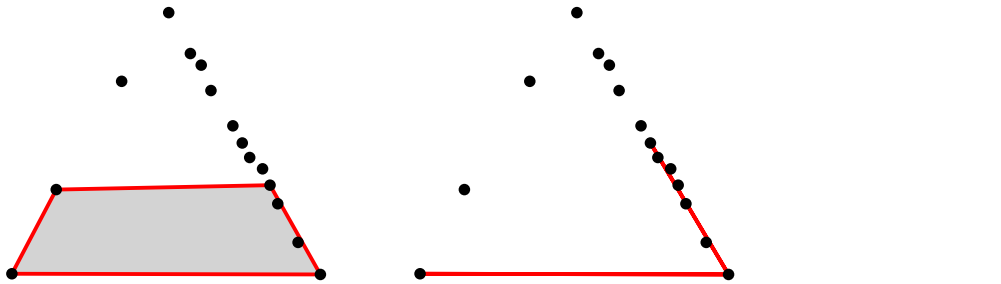
[2] A set C is convex if for any $a, b \in C$ also the whole line segment ab is contained in C.



(a) leftturn (b) rightturn (c) collinear

Figure 2.3: *Orientation test for a point triple* p, q, r *in* $\mathbb{R}^2$.

The first operation, lexicographic comparison, can be implemented using a standard comparison of two numbers and therefore is mostly foolproof. However, a straightforward implementation of the second operation, orientation test, using limited precision floating point arithmetic does not guarantee correct results in general. In fact, it is quite easy to run into outputs as depicted in Figure 2.4, with an otherwise completely correct implementation of the top layer algorithm.



(a) (b) (c)

Figure 2.4: *Possible results of a convex hull algorithm with a naively implemented orientation test.*

Maybe you are wondering what the examples in Figure 2.4 are supposed to show? There is not much to see in Figure 2.4c, for instance. Indeed, this "empty figure" stands for a situation where the program runs into an infinite loop and therefore does not produce any output. This should not happen with an *algorithm*, of course, but it is not an algorithm anymore if the elementary operations are not guaranteed to give correct results. In Figure 2.4b the output polygon is formed by a cyclic sequence of four vertices p, q, r, q, which does not even form a convex polygon, let alone one that contains all points. Finally, in Figure 2.4a we see a convex polygon, at least, but there are many points far outside of it. To sum it up,

> a naive implementation using limited precision arithmetic may lead to a complete failure rather than to a reasonable approximation of the desired solution.

Note the contrast to a typical situation for a (reasonably stable) numerical computation where the solution is—if not exact—always *some* approximation of the actual

result. The reason for such an extreme "instability" here is that we use a numerical computation to obtain a discrete (that is, finite) combinatorial structure. If the inevitable rounding goes the wrong way, the result is incorrect, period. True is not a reasonable approximation of false and neither the other way around...

2.5 Orientation Test

Now you may still be wondering what is the deal with this orientation test and how something can go wrong with it, even if implemented naively using limited precision arithmetic. So let us look at what computations are required on the arithmetic level for such a test.

On the algebraic level the orientation test for the triple p, q, r with $p = (p_x, p_y)$, $q = (q_x, q_y)$, and $r = (r_x, r_y)$ boils down to[3] computing the sign of the homogeneous 3×3-determinant

$$\begin{vmatrix} p_x & p_y & 1 \\ q_x & q_y & 1 \\ r_x & r_y & 1 \end{vmatrix} = \begin{vmatrix} p_x & p_y & 1 \\ q_x - p_x & q_y - p_y & 0 \\ r_x - p_x & r_y - p_y & 0 \end{vmatrix} = (q_x - p_x)(r_y - p_y) - (q_y - p_y)(r_x - p_x).$$

Computing the test in this fashion requires two multiplications and five subtractions. This seems simple enough as a computation. But—in contrast to a simple comparison of two numbers—there is a multiplication involved, which may drastically increase the size of the representation of the numbers involved. If these intermediate results do not fit into the data type used for the computation, then a loss of precision occurs and the overall result may be incorrect. In particular, this problem is relevant if the overall result of the computation is small, that is, close to zero, for two reasons:

1. The range around zero is the most interesting range for the test because here the sign changes and hence the result flips between left, collinear, and right.
2. If both products yield comparatively large numbers (of the same sign, given that the overall result is close to zero; see (2.1) below), then even a relatively small loss of precision for these numbers may affect the sign of the overall result.

$$|\underbrace{(q_x - p_x)(r_y - p_y)}_{\text{large}} - \underbrace{(q_y - p_y)(r_x - p_x)}_{\text{large}}| \leqslant \text{small} \tag{2.1}$$

In summary we conclude that

> already a tiniest loss of precision at any step of the computation may change the result of an orientation test (and thus make this result incorrect).

[3] This claim may not be immediately obvious. If not, do not worry and just accept it. This is not hard to see, using that the determinant describes twice the signed area of the triangle pqr. But this type of reasoning is not our focus here.

Conversely, the most straightforward way to obtain correct results is to ensure that *no* loss of precision occurs, that is,

Exact (Geometric) Computation Paradigm (Yap [10])
ensure that all computations are done exactly (and therefore correctly).

Now the big question is: How can we compute exactly with the limited resources of a real-world computer? We will see a few possible answers to this question in the remainder of this chapter.

2.6 Simple bounds on the size of integral numbers

In computations with an integral data type it is very clear when a loss of precision occurs: whenever a number is larger or smaller than the largest or smallest, respectively, representable integer of this data type. A simple strategy to ensure that this does not happen goes as follows:

- Assume that all input numbers are bounded, that is, for every input number x we have $|x| < 2^w$, for some fixed $w \in \mathbb{N}$ to be determined.
- Analyze the computation(s) made and derive worst-case bounds in terms of w for all numbers that arise during the computation.
- Based on the bounds obtained in the previous step and the characteristics of the data type to be used, set an upper bound on w and thereby on the input numbers.

As we are concerned with data types on a real-world computer, where numbers are usually represented in a binary format, we formulate all bounds in terms of bits. Hence our assumption from above states that the input numbers are w-bit unsigned integers or, if we want to account for the sign, $(w+1)$-bit signed integers. Then for any algebraic expression that uses additions, subtractions, and multiplications only, it is very easy to obtain simple worst-case bounds using the following observation.

Observation 2.2. *For $a, b \in \mathbb{Z}$ with $|a| < 2^v$ and $|b| < 2^w$, we have $|a \pm b| < 2^{\max\{v,w\}+1}$ and $|ab| < 2^{v+w}$.*

Let us apply this scheme to the orientation test, computed as

$$\underbrace{\underbrace{\underbrace{(q_x - p_x)}_{|\cdot| < 2^{w+1}} \underbrace{(r_y - p_y)}_{|\cdot| < 2^{w+1}}}_{|\cdot| < 2^{2w+2}} - \underbrace{\underbrace{(q_y - p_y)}_{|\cdot| < 2^{w+1}} \underbrace{(r_x - p_x)}_{|\cdot| < 2^{w+1}}}_{|\cdot| < 2^{2w+2}}}_{|\cdot| < 2^{2w+3}}$$

The worst bound we get for the overall result, and so we conclude that all numbers in the computation are $< 2^{2w+3}$ in absolute. Therefore, in order to be able to compute exactly with the type `int` we get a sufficient condition of the form $2w+3 \leqslant 31$. That is, setting $w = 14$ suffices to guarantee that all numbers in the computation can be represented. Similarly, for the type `long` we get $2w+3 \leqslant 63$, that is, setting $w = 30$ and in this way requiring all input coordinates to be $< 2^{30}$ in absolute is enough here.

This type of analysis with these simple computations we expect you to do by yourself whenever you need to check whether the (final and intermediate) results of a computation fit into a particular limited precision data type.

Exercise 2. *The* squared Euclidean distance *of two points* $p = (p_x, p_y)$ *and* $q = (q_x, q_y)$ *is computed by* $(p_x - q_x)^2 + (p_y - q_y)^2$. *If we want to compute it exactly using the type* `int` *for all input coordinates bounded by* 2^w *in absolute, how large can we allow* w *to be? What if we use the type* `long` *instead?*

Exercise 3. *The* incircle test *for a quadruple of points* p, q, r, s *determines if* s *is inside, on, or outside the disk bounded by* p, q, r. *For* $p = (p_x, p_y)$, $q = (q_x, q_y)$, $r = (r_x, r_y)$, *and* $s = (s_x, s_y)$ *it is computed as the sign of the determinant*

$$\begin{vmatrix} p_x & p_y & p_x^2 + p_y^2 & 1 \\ q_x & q_y & q_x^2 + q_y^2 & 1 \\ r_x & r_y & r_x^2 + r_y^2 & 1 \\ s_x & s_y & s_x^2 + s_y^2 & 1 \end{vmatrix}.$$

If we want to compute it exactly using the type `int` *for all input coordinates bounded by* 2^w *in absolute, how large can we allow* w *to be? What if we use the type* `long` *instead?*

2.7 Computing with floating point numbers

In practice most computations are done with floating point numbers rather than with integers. Floating point data types allow to represent a much broader range of numbers. For instance, using 64 bits the type `double` can represent positive numbers as small as 2^{-1074} and as large as $\approx 2^{1024}$; compare this to a 64-bit integer, where the smallest representable positive number is 1 and the largest one is $\approx 2^{63}$. Of course, `double` cannot represent *all* numbers is this range exactly, but it can represent them with a relative error of at most 2^{-53}.

Although we talk about the C++ type `double` here, the properties of a binary 64-bit floating point representation are by no means restricted to C++. These properties are regulated the IEEE Standard for Floating-Point Arithmetic (IEEE Std 754 [6]), which most programming languages and computers—if high-end multiprocessor devices or simple pocket calculators—adhere to by and large.

So, how does this standard binary 64-bit floating point representation look like? It consists of three parts, see Figure 2.5 for illustration:

- the sign $\pm$ (using 1 bit),
- the exponent x (using 11 bits, with $-1022 \leqslant x \leqslant 1023$), and
- the mantissa m (using 53 bits, with $1 \leqslant m \leqslant 2 - 2^{-52}$),

to represent numbers of the form $\pm m \cdot 2^x$.

$\pm$	exponent	mantissa (significand)
1 bit	11 bits	53 bits

Figure 2.5: *IEEE 754 binary 64-bit floating point representation.*

Looking at these parameters you may notice a few curiosities, some of which can be explained by the normalization, which is implicit in the condition $1 \leqslant m \leqslant 2 - 2^{-52}$.

Why do we normalize? With a "floating point" and an exponent parameter to move it, there are many different ways to express the same number, as in

$$1 \cdot 2^0 = \frac{1}{2} \cdot 2^1 = 2 \cdot 2^{-1} = \frac{1}{4} \cdot 2^{-2} = 4 \cdot 2^{-2} = \cdots .$$

Such an ambiguity would be both confusing and inefficient for a representation. Fixing the mantissa to always lie between two specific consecutive powers of two—here $1 = 2^0$ and $2 = 2^1$—resolves this issue and yields a unique representation.

Normalization also explains why the representation seems to use $1 + 11 + 53 = 65$ bits rather than the 64 bits expected: By requiring $1 \leqslant m \leqslant 2 - 2^{-52}$, we fix the binary digit before the floating point to be one. So there is no need to actually store this bit. Denoting by $d_1, \ldots, d_{52}$ the 52 binary digits of m explicitly stored in the representation, we can write m as a binary number of the form $(1.d_1 \cdots d_{52})_2$, where the subscript two indicates a binary representation.

However, there is a rather important integer that cannot be normalized in this fashion because it does not have any nonzero digits: the number zero. Indeed, the number zero has a special representation, using an all-zero mantissa (one, with the implicit leading bit) and a reserved exponent.[4]

As for the *reserved exponent*, you may have noted that two numbers are "missing" from the exponent: For a signed 11-bit integer we would expect a range from -1024 to 1023 (compare with the types `int` and `long` in Table 2.1). These two reserved exponents are used to encode several special limit and error states, including the number zero, and also so-called subnormal numbers to fill the range between zero and 2^{-1022}, which is the smallest normal positive number representable.[5] But we will not further concern us with such details here. If you want to read more about floating point numbers and

[4] both signs are allowed; yes, zero is actually signed here...

[5] The abovementioned number 2^{-1074} is such a subnormal number, the smallest positive `double` value.

computations, check out the report "What Every Computer Scientist Should Know About Floating-Point Arithmetic" by Goldberg [3].

In order to get an intuition about how the numbers that are representable as a floating point number distribute over the reals, let us look at the graphical representation in Figure 2.6. Every tick along the real line corresponds to a floating point number. For `double` there would be so many ticks that it would be hard to see anything. Therefore, we visualize a similar but simpler and much sparser floating point system, with

- two bits for the mantissa, which is normalized so that the representable numbers are $(1.d_1d_2)_2$, where d_1 and d_2 denote the two bits in the representation, and
- three bits for the exponent, which ranges from -3 to 2, so that there is room for one reserved value to represent zero and subnormal numbers (shown as red ticks in the figure); no other special values are represented[6].

As the picture for negative numbers would be symmetric, we do not use a sign bit and consider positive numbers only. The largest number representable is $(1.11)_2 \cdot 2^2 = 7$, and the smallest normal number is $(1.00)_2 \cdot 2^{-3} = 1/8$. For a mantissa of $(1.00)_2$ we get powers of two, and the remaining three values $(1.01)_2$, $(1.10)_2$, and $(1.11)_2$ correspond to the three uniformly spread ticks between any two representable powers of two.

Using normal numbers only there would be a gap between the smallest representable positive number (here $1/8$) and zero. Moreover, this gap would be quite large compared to the gap between the smallest and the second smallest positive number (here $1/32$). That is why there exist subnormal numbers, to fill this gap with the same number of ticks (here three) as there are between any consecutive powers of two.

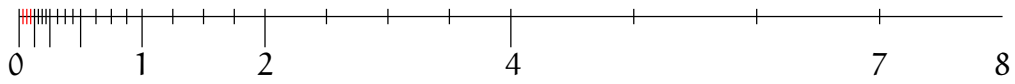


Figure 2.6: *Graphical representation of a simple floating point systen with two bit mantissa and three bit exponent.*

So for `double` we would have $2^{52} - 1 > 4.5 \cdot 10^{15}$ ticks between any two representable consecutive powers of two, which is why we made the figure for such a hypothetical toy system. In particular, `double` can represent a large contiguous range of integers exactly. Let us see what is the smallest positive integer that cannot be represented as a `double`. With an exponent of 52, we use the full bit pattern of the mantissa. The largest number of this form consists of 53 consecutive binary digits of one, in decimal $2^{53} - 1$. The next larger integer 2^{53} is a power of two (with exponent $\leqslant 1023$) and therefore also exactly representable as a `double`. But the next larger integer

$$2^{53} + 1 = (1\underbrace{0 \cdots\cdots\cdots 0}_{52 \text{ times}}1)_2$$

[6] in contrast to `double` where another exponent is reserved for these

cannot be represented anymore because its two digits of one in binary format are separated by 52 zero digits. Nevertheless, we get a fairly large range of integers that can all be represented exactly.

Using `double` as an integer data type. All integers between -2^{53} and 2^{53} can be represented exactly as a `double`.

Exercise 4. *What if you we use the type `double` instead of `int` in Exercise 2 and 3? What bound do you get for the orientation test with `double`?*

Exercise 5. *Which of the following numbers can be represented exactly as a `double`? (a)* $2^{52}+2$ *(b)* $2^{52}+1/2$ *(c)* $\sqrt{2}$ *(d)* $1/10$ *(e)* $1,234,567,890$

So `double` can represent a large range of numbers with a fairly decent precision. That is why computing with floating point numbers works so amazingly well in so many cases. However, we should not be lured into the trap of thinking that we can take floating point computations as a faithful realization of algebra over the reals. It is easy to run into a number that cannot be represented exactly, in which case it is represented by rounding to the nearest representable number. This is where a loss of precision occurs. Let us consider the following program as an example.

```
#include <iostream>

int main()
{
    double x = 1.1;
    x -= 1;
    x -= 0.1;
    std::cout << x << "\n";
}
```

Naively, we would expect the output to be zero. But the actual output is `8.32667e-17`. Why? Let us see how the number 1.1 (in decimal) is represented as a `double`. Of course, 1.1 is not the same as $(1.1)_2 = 1+2^{-1} = 1.5$. In fact, $1.1 = (1.0\overline{0011})_2$ has a periodic binary expansion and therefore no exact explicit representation using finitely many binary digits, similar to, for instance, $1/3 = 0.\overline{3}$ in decimal. So in a representation as a `double`, which is an explicit representation with finitely many digits, all binary digits beyond the 52^{nd} digit after the floating point are lost. As the first lost digit happens to be a one, the result is rounded up. A similar error is made when representing 0.1 as a `double`, but here the cutoff happens four binary digits further to the right so that the overall error by rounding up 1.1 dominates the result and makes it a small positive number $< 2^{-53}$.

This is a rather small error, of course, which in isolation would not be a big deal. This is true in general for the effect of rounding a single number to the nearest representable `double` because by construction of the floating point system (placing the "ticks" uniformly) the relative error is small. However, if further computations are done with

this number, the error may be amplified up to a point where it suddenly makes a difference. When writing code, it is *your job* as a programmer to keep this in mind and consciously investigate and decide if everything is fine, or if there may be a problem with the precision of computations.

2.8 Anomalies in floating point geometry

As an example of how a small loss of precision can ruin an algorithm let us go back to the orientation predicate and look at some experiments by Kettner et al. [7]. They evaluated the orientation predicate for a collection of point triples $p = (p_x, p_y)$, $q = (q_x, q_y)$, $r = (r_x, r_y)$—as discussed in Section 2.5, by computing the sign of the expression

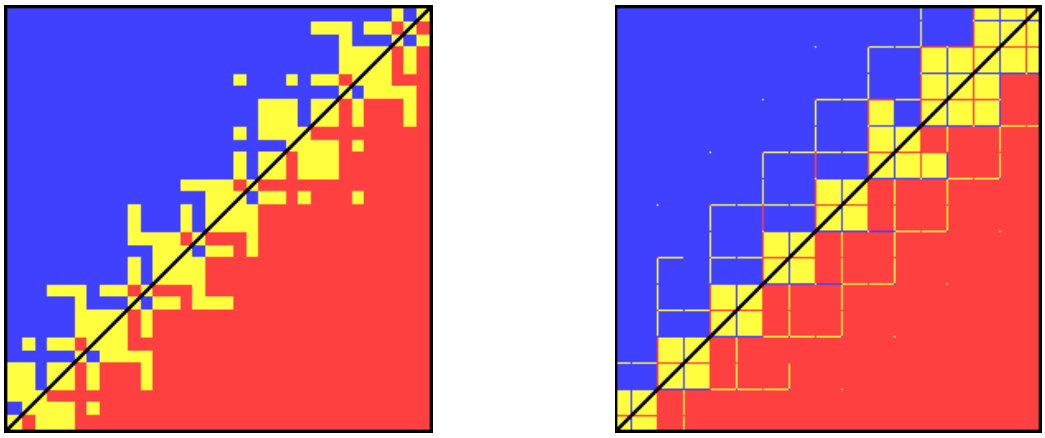
$$(q_x - p_x)(r_y - p_y) - (q_y - p_y)(r_x - p_x)$$

using the number type `double`—where q and r are fixed and p varies in both coordinates. More precisely, they consider the points $p = (p_x + X \cdot 2^{-53}, p_y + Y \cdot 2^{-53})$, where $p_0 = (p_x, p_y)$ is fixed and X and Y range over the integers between 0 and 255 (eight bit integers). For every choice of X and Y they obtain a corresponding orientation, that is, a function $f : \{0, \ldots, 255\}^2 \to \{\text{left}, \text{right}, \text{collinear}\}$. This function can be visualized as a two-dimensional (256×256)-grid, where the grid rectangle (i, j) is filled blue if $f(i, j) = \text{left}$, red if $f(i, j) = \text{right}$, and yellow if $f(i, j) = \text{collinear}$.

They consider several different choices for p_0, q, r and the corresponding family of orientations. Here we will only look at two of those choices. First, let $p_0 = (0.5, 0.5)$, $q = (12, 12)$, and $r = (24, 24)$. This means that p_0, q, r are collinear, on the line ℓ described by $y = x$. So what kind of picture do we expect for the corresponding colorful grid defined above? The diagonal (where $X = Y$) should be yellow because all those points are on ℓ, together with q and r. Above the diagonal we have $Y > X$, and so p, q, r form a leftturn and the corresponding grid cells should be colored blue. Symmetrically, below the diagonal we have $X > Y$, and so p, q, r form a rightturn and the corresponding grid cells should be colored red. Now compare these observations to the actual picture obtained using computations with `double`, as shown in Figure 2.7a.

Obviously, expectation and result do not quite match. The region that corresponds to the supposedly straight line (shown in yellow) is neither straight nor convex, nor even connected. The same holds for the regions above and below the line (shown in blue and red, respectively), which are neither convex nor connected. At least the actual geometric line ℓ is contained in the yellow region, and points that are really far away from ℓ are classified correctly. But many points that are somewhat close to ℓ are misclassified in weird, seemingly unpredictable ways. Even more interesting patterns can be obtained by choosing other coordinates for q and r. For instance, setting $q = (8.8000000000000007, 8.8000000000000007)$ and $r = (12.1, 12.1)$ yields the picture shown in Figure 2.7b.

Even without going into further details, it should be plausible that an algorithm that expects to work with straight lines but instead works with lines such as the yellow regions



(a) (b)

Figure 2.7: *The geometry of lines according to limited precision computation (examples by Kettner et al. [7]). The actual line is shown as a black solid line, the line according to limited precision arithmetic is shown as a yellow region.*

in Figure 2.7 runs into inconsistencies with potentially unpredictable consequences. Also note that the popular "fix" of considering all values smaller than some fixed $\varepsilon > 0$ to be zero does not change the picture at all, except that the solid yellow region around the line grows; but the boundaries between the colors remain just as jagged and disconnected as before.

2.9 How to ensure correctness

On the one hand, the examples discussed in the previous sections illustrate that we need to be careful with limited precision computations and their potential to invalidate an otherwise correct implementation of an algorithm. On the other hand, limited precision computation is all that real-world computers can provide. So there seems to be a problem here: We know of the shortcomings of our tool but as we do not have anything better, we are stuck with using it and just hope for the best? That would be an easy but also sluggish conclusion. Of course, we can do better. Let us mention at least two more conscious, satisfying and ultimately correct ways to deal with the differences between abstract model and practice.

The first approach is to know your inputs and your computations and verify that the available limited precision arithmetic suffices to guarantee correct results. In some cases, like sorting integers, or more generally when only trivial computations are required, such a guarantee is easy to provide. In other cases, when nontrivial computations are required, this can be more challenging. But, for instance, using simple bounds such as those discussed in Section 2.6 one can relatively easily determine a sufficient precision depending

on the size of the input parameters, and then select the data type for the computations accordingly. See, for example, the bounds in Exercise 4 regarding computations with `double` for several elementary operations. A few questions remain, though.

For once, we did not discuss any simple bounds for divisions and root operations, neither for other standard mathematical operations, such as trigonometric functions. Indeed, we did not, and there is a reason: We recommend you to avoid these operations in implementations if possible. Within the scope of this course, they can avoided in most cases, as we will explain in the following paragraphs.

Guideline 2 (Avoid (square)roots). In comparisons you can often omit squareroots because $\sqrt{x} \leqslant \sqrt{y} \iff x \leqslant y$, for all $x, y \geqslant 0$.

Computing a root is a comparatively expensive operation, which in most cases (except for exact squares) yields a number that cannot be explicitly represented exactly using finite precision. In a comparison as in the box above it is obvious that omitting the roots is better—even in theory. Sometimes it is inevitable to compute roots, though, for instance, to obtain an output number.

Guideline 3 (Avoid divisions). When comparing fractions, you can replace them with multiplications because for $b, d > 0$ we have

$$\frac{a}{b} \leqslant \frac{c}{d} \iff ad \leqslant bc.$$

A typical division results in a number that cannot be explicitly represented using finite precision. Recall the example $0.1 = 1/10$ from Section 2.7. Typical exceptions are divisions by a power of two (which are just shifts or changes to the exponent in a binary representation) or integer divisions where either you know in advance that the divisor divides the dividend without remainder or you want to work with modular arithmetic.

As for many other mathematical functions, we guarantee that you do not need to compute them for any problem in this course. Nonalgebraic functions, such as trigonometric functions or logarithms, are relatively expensive to compute (compared to simple arithmetic operations) and difficult to compute with exactly.

Exercise 6. *Suppose we are given three points in homogeneous coordinates as*

$$p = \left(\frac{p_x}{p_d}, \frac{p_y}{p_d}\right), q = \left(\frac{q_x}{q_d}, \frac{q_y}{q_d}\right), r = \left(\frac{r_x}{r_d}, \frac{r_y}{r_d}\right)$$

and want to test whether p *is at least as close to* r *as* q *(is to* r*), according to the Euclidean distance. Give a corresponding inequality as an algebraic expression in the nine input coordinates* $p_x, p_y, p_d, q_x, q_y, q_d, r_x, r_y, r_d$*, formulated in accordance with the guidelines discussed above. Then analyze what is the largest uniform bound of the form* $|\cdot| < 2^b$ *for the input coordinates so that your test can be computed exactly using* `double`*.*

Another question: What if the analysis of a computation reveals that the result may not fit into any of the builtin number types available? This brings us to the second approach we want to discuss here: Use an *exact* number type. These are data types that can represent a large range of numbers exactly, either explicitly or symbolically, limited only by the available memory and not intrinsically by a fixed maximum precision, as for the builtin types. In this course, we use four different exact number types: `CGAL::Gmpz`, `CGAL::Gmpq`, and the field types of the CGAL default kernels, which will be discussed later.

2.10 The types Gmpz and Gmpq

The types `CGAL::Gmpz` and `CGAL::Gmpq` represent arbitrary precision integer ($\mathbb{Z}$) and rational ($\mathbb{Q}$) numbers, respectively. They provide an explicit representation, which you may think of as a dynamic array of digits[7]. Such an array grows and shrinks as needed so as to represent the result of the computation. You can compute with them, using basic arithmetic operations, as with builtin number types. However, be aware that the complexity of these operations is not necessarily constant but proportional to the size of the numbers involved (that is, the length of the corresponding array).

As an example, suppose that you wanted to compute the factorial 2020!. This number has 5802 decimal digits, which is solidly above what any of the builtin types can represent. As integer multiplication is the only operation needed, we can work with `CGAL::Gmpz` to obtain the result.

```
#include <iostream>
#include <CGAL/Gmpz.h>

int main()
{
  CGAL::Gmpz r = 1;
  for (int i = 2; i <= 2020; ++i) r *= i;
  std::cout << r << "\n";
}
```

As an example for the usage of `CGAL::Gmpq` let us compute a rational approximation for $\sqrt{2}$ up to more than a thousand binary digits using Newton's method.

```
#include <iostream>
#include <cmath>
#include <CGAL/Gmpq.h>

int main()
{
  const double precision = std::ldexp(1.0, -1074); // min. positive double 2^{-1074}
  CGAL::Gmpq s = 1;
  do
```

[7] rather array of words, say, 64-bit numbers, for efficiency reasons, but the principle is the same

```
    s = s/2 + 1/s;
  while (abs(s*s - 2) > precision);
  std::cout << s << "\n";
}
```

Recall that `CGAL::Gmpq` represents a fraction. So in the output of the above code the result is printed as a sequence *numerator* '/' *denominator*. To get a more visible separation between both, you could use the member functions `CGAL::Gmpz numerator()` and `CGAL::Gmpz denominator()` to output both separately.

```
std::cout << s.numerator() << "\n/\n" << s.denominator() << "\n";
```

The examples above are for the purpose of illustration only. We will not deal much with such numerical computations in this course. But it is good to have tools at hand that can go beyond what the builtin types can represent. Also, these two types will be relevant in the context of solving linear programs. So, you should know how to work with them (which is not all that different from working with builtin number types, after all).

Exercise 7. *Adapt the code above to compute a rational approximation for* $\sqrt{2}$ *up to more than ten thousand binary digits.*

Exercise 8. *Write a function* `binomial` *that takes two arguments* n, k *of type* `int` *and computes the binomial coefficient* $\binom{n}{k}$. *Use it to compute* $\binom{2020}{42}$.

Chapter 3

CGAL kernels for geometric computing

A common theme throughout this course is that we encourage you to use tools rather than implementing everything from scratch. Just as there is no need to implement a comparison based sorting algorithm yourself when you can just use `std::sort`, you do not have to implement geometric primitives yourself, either. Instead you can (and probably should) use the elementary operations and algorithms provided by CGAL wherever they come handy.

In CGAL the separation between the two layers of elementary operations and the high-level algorithms and data structures (as discussed in Section 2.2) is very explicit. For a specific algorithm or data structure the corresponding list of required data types and elementary operations is referred to as the algorithm's or data structure's *traits*. This list of requirements is typically described as an abstract *concept*, for which—possibly several different—traits class *models* are available that provide concrete implementations of those data types and operations.

Independently from specific algorithms or data structures there exists a collection of core data types and operations for Euclidean geometry in two, three, and arbitrary dimensions. These collections are referred to as *(geometry) kernels*. Within this course we will only deal with two-dimensional Euclidean geometry and the corresponding CGAL kernels. Such a two-dimensional kernel provides data types to represent points, lines, line segments, triangles, etc. and elementary operations such as an orientation test for point triples.

There are many different kernel models available in CGAL. But for this course only three of them are relevant, the three so-called *default kernels*, which have rather lengthy but descriptive names:

- `CGAL::Exact_predicates_inexact_constructions_kernel` (or *epic* kernel),
- `CGAL::Exact_predicates_exact_constructions_kernel` (or *epec* kernel), and
- `CGAL::Exact_predicates_exact_constructions_kernel_with_sqrt` (or *sqrt* kernel, for short).

All three kernel provide the same collection of data types and elementary operations. But—as the names suggest—the implementations differ in how they handle nontrivial *constructions* and whether or not the underlying number type supports exact square-roots. In order to understand these differences we need to first understand the difference between predicate and construction.

3.1 Predicates and constructions

Every elementary operation has *some* result. In a comparison of the form $a \leqslant b$ for a sorting algorithm, the result is a Boolean: either true or false. Alternatively one could use a `compare(a,b)` operation that has three possible results: (a is) smaller, equal, or larger (than b). Similarly, we formulated the orientation test with three possible results (leftturn, rightturn, or collinear), but could also phrase it in the form "Do p, q, r form a leftturn?", with a Boolean result. Regardless, for all of the mentioned operations the range of possible results is *finite*. Such operations are called *predicates*.

In contrast, a *construction* is an operation for which the range of possible results is conceptually *infinite*. An example of a construction is the operation "Give me the line through p and q!" for two distinct points p, q. The result is a line, and there are conceptually infinitely many pairwise distinct lines (at least in dimension two and higher). Let us add a few remarks for clarification.

- We write "conceptually" infinite because a real-world computer has only finite memory and therefore can only represent finitely many distinct objects of any particular type (for instance, lines).
- For every *specific* pair p, q with $p \neq q$ there is a unique line through p and q. But *for all possible choices* of the arguments p, q collectively there are infinitely many. In contrast, for all possible choices of arguments a, b to `compare(a,b)` there are three possible results only.
- The term construction a priori has nothing to do with the construction of objects in C++. That said, it is possible to implement a construction using a C++ constructor.

From the perspective of a top-level algorithm that uses an elementary operation, the difference between using a predicate and using a construction can be significant. To see this, consider the decision tree model discussed in Section 2.3: A predicate with three possible results corresponds to a vertex in the tree with three children; a construction, however, is like a vertex with infinitely many children or rather, it adds another possible input to all computations in the subtree below. Moreover, depending on the operation at hand this "new input" may be more complex (for instance, with larger parameters/-coordinates) than the original inputs.

In a similar fashion a support for exact (computation of) constructions introduces complications for the implementation of a corresponding traits class or kernel, and requires more involved internal data structures and computations. While you do not see

the internal details when just using a kernel as a black box, you may notice a resulting runtime overhead.

3.2 Which kernel to choose

As mentioned above, for the purposes of this course your choice is between three kernels: epic, epec, and sqrt. All three provide the same data types and operations, and all three provide exact predicates, that is, all predicates are guaranteed to always give correct results. This is done by making sure that the internal computations are done with sufficient precision.[1]

Each kernel has an associated number type as a *field type*, to model the field of real numbers, which is used for most coordinates and parameters. On the C++ level the field type of a kernel can be obtained as a nested type `FT`. The field types of the three mentioned kernels are pairwise distinct.

For the epic kernel, the field type is `double`. All constructions in the epic kernel are computed with `double`. As a result, constructions may suffer from loss of precision and are *not* guaranteed to be exact and correct.

In contrast, the field types of the epec and the sqrt kernel are special custom types. They allow both of these kernels to perform all constructions exactly and correctly, at the cost of more involved internal representations. Furthermore, the field type of the sqrt kernel supports exact computation with (square)roots, which the field type of the epec kernel does *not*.

The sqrt kernel is the most powerful; it can do everything that the others can and more. So it might look tempting to just always go with this kernel. However, the power comes with a price: As a rule of thumb, the epic kernel is the fastest and the sqrt kernel is the slowest of the three. So, maybe you should rather always go with epic then? Obviously, it depends. If you *need* nontrivial constructions and want to be sure that they are handled correctly, you better choose a kernel that supports it. Similarly, if you really need exact squareroots, the decision for the sqrt kernel is easy.

Guideline 4 (Choosing the kernel). Among the three default kernels epic, epec, and sqrt, choose the first (in the given order) that supports everything you need.

Quantifying the performance differences between these kernels is difficult in general. It really depends on what you do, that is, what is the size of the input numbers, which operations are used how many times, etc. So while in some scenarios the differences are quite noticeable, there may be others where they are marginal. Regardless, you cannnot go wrong by following the above guideline.

So in order to choose the "right" kernel, you need to know what operations you need. And you should try to be a bit flexible when figuring out your needs because

[1] To do this efficiently requires some work. But for the scope of this course let us not worry about the details and just consider the kernels as blackbox tools.

they strongly depend on how you formulate and approach a problem. For instance, at a first glance it may seem that working with Euclidean distances requires computing squareroots. But if all that is needed is comparison of distances, then working with squared Euclidean distances instead works just as well and without squareroots. Or maybe at a first glance it seems natural to go for a solution that uses constructions, even though these constructions could be replaced by one or several predicates.

To be continued...

References

[1] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars, *Computational Geometry: Algorithms and Applications*, Springer, Berlin/Heidelberg, Germany, 3rd edn., 2008.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, *Introduction to Algorithms*, MIT Press, Cambridge, MA, 3rd edn., 2009.

[3] David Goldberg, What Every Computer Scientist Should Know About Floating-Point Arithmetic. *ACM Comput. Surv.*, 23/1, (1991), 5–48.

[4] Torbjörn Granlund and the GMP development team, *The GNU Multiple Precision Arithmetic Library (Manual)*. 6.2.0 edn., 2020.

[5] Torben Hagerup, Sorting and Searching on the Word RAM (Invited Talk). In *Proc. 15th Sympos. Theoret. Aspects Comput. Sci.*, vol. 1373 of *Lecture Notes Comput. Sci.*, pp. 366–398, Springer, 1998.

[6] *IEEE Standard for Floating-Point Arithmetic, IEEE Std 754 – 2019*. New York, NY, 2019.

[7] Lutz Kettner, Kurt Mehlhorn, Sylvain Pion, Stefan Schirra, and Chee-Keng Yap, Classroom examples of robustness problems in geometric computations. *Comput. Geom. Theory Appl.*, 40/1, (2008), 61–78.

[8] Jacques-Louis Lions, Lennart Lübeck, Jean-Luc Fauquembergue, Gilles Kahn, Wolfgang Kubbat, Stefan Levedag, Leonardo Mazzini, Didier Merle, and Colin O'Halloran, Ariane 5—Flight 501 Failure. Report by the Inquiry Board, European Space Agency, 1996.

[9] Kurt Mehlhorn, *Data Structures and Algorithms 1: Sorting and Searching*, vol. 1 of *EATCS Monographs on Theoretical Computer Science*, Springer, Berlin/Heidelberg, Germany, 1984.

[10] Chee-Keng Yap, Towards exact geometric computation. *Comput. Geom. Theory Appl.*, 7/1, (1997), 3–23.