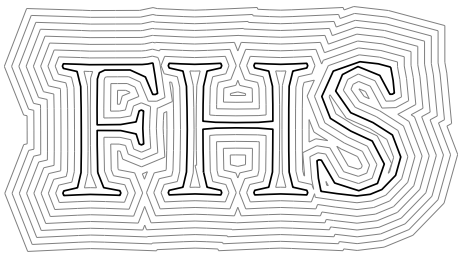
Numerical programming and industrial algorithms

Stefan Huber

Information Technology and System Management
Salzburg University of Applied Sciences

Winter 2021



Contents

Chapter 0

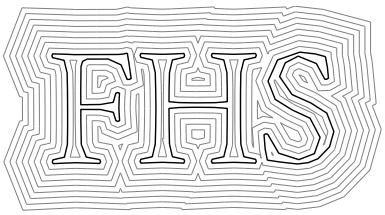
Introduction

A philosophical introduction. Computer science and mathematics both belong to the structural sciences. The structural sciences investigate formal systems and abstract structures, like sets, functions, mathematical spaces, algorithms, programs, databases, software architectures. Physics or chemistry are natural sciences. They investigate phenomena that we observe in our reality.

The typical subfields of mathematics used in computer science are discrete, like graphs, abstract algebra, number theory and the like. The typical subfields of mathematics used in physics – at least for the purpose of engineering – are continuous, like the Euclidean geometry in $\mathbb{R}^3$ or differential equations, which describing phenomena in mechanics, electromagnetism, thermodynamics and so on. This is why we tend to find integers in computer science but real numbers in physics.

This course is about selected topics that arise from the application of computer science to a continuous mathematical world, as they arise when we deal with problems of the natural sciences, in particular in physics, and engineering. More precisely, this course is about computer science applied to the disciplines of linear algebra, calculus, geometry and topology. When we apply computer science to the real, continuous world, in some sense we leave the "natural habitat" of computer science. Operations on integers are performed exact – without numerical loss – by a (digital) computer, but dealing with real numbers is impossible for a computer.[1] We are left with approximations of real numbers and each operation introduces loss of precision. The field of numerical analysis investigates the problems that arise from this fact.

The figure at the title page of these lecture notes exemplifies the type of problems we discuss here. It has been computed by Stalgo, a software package to compute so-called straight skeletons.[2] We are given a shape forming the letters "FHS". Suppose we want to mill out the interior or exterior with a CNC milling machine. In order to do so, we have to compute tool paths for the machine.



[1] There are uncountably many real numbers, but there are only countably many strings over a finite alphabet, so we cannot even represent all possible real numbers even if we would have unlimited memory, like a Turing machine with its infinite memory tape. So in this sense we really leave the natural habitat of computer science.

[2] `https://www.sthu.org/code/stalgo/`

One strategy is to compute a family of so-called offset curves that are parallel to the shape. But how do we precisely define what an offset curve is?[3] How do we compute them in a computational efficient and numerically stable way? How do we actually test whether the CNC tool center is currently located within the letter shapes or outside? How can we approximate parts of the tool paths with a smoother representation?

The organization of these lecture notes. This course is split into two parts: Numerical programming and computational geometry. The first part deals with numerical analysis, linear algebra and calculus. This branch of mathematics is often called *numerical mathematics*, but since we put some emphasis also on technical details and programming, we instead call it *numerical programming*. The following literature is recommended for further reading:

- The lecture notes of Johann Linhart are an excellent and dense compilation on the main topics of numerical mathematics. Johann Linhart was professor at the math department of the University of Salzburg and his lecture notes put an emphasis on the mathematical point of view.

  [17] Johann Linhart. *Numerische Mathematik*. WS 2004/05. URL: https://www.uni-salzburg.at/fileadmin/multimedia/Mathematik/documents/Num.Mathematik_WS2004.pdf

- The *Numerical Recipes* belongs to the standard literature of every engineer that produces code and every computer scientist with applications in the real world. This book ships code with a strong emphasis on numerical stability and computational speed. (However, it has not so much emphasis on readable and clean code.)

  [19] William H. Press et al. *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. 3rd ed. New York, NY, USA: Cambridge University Press, 2007. ISBN: 0521880688, 9780521880688

- A comprehensive guide on numerical computation, scientific computing and floating-point arithmetics has been published by Sun microsystems. It contains an article by Goldberg [10] that discusses many details of floating-point units.

  [20] Sun One Studio. *Numerical Computation Guide*. 2003

The field of computational geometry is widely considered of being a part of theoretical computer science, i.e., algorithm theory in the context of geometry. In recent years a development started that expended its scope to also encompass computational topology with major applications in data analysis. The following literature is recommended for further reading:

- This book is largely considered as being the standard book on computational geometry. It contains all classical topics of the field starting in the 1970s.

  [4] Mark de Berg et al. *Computational Geometry: Algorithms and Applications*. 3rd ed. Santa Clara, CA, USA: Springer-Verlag TELOS, 2008. ISBN: 3540779736, 9783540779735

- The lecture notes of Martin Held are one of the few that also cover the topics of numerical computation for computational geometry. Martin Held is a professor at the computer science department at the University of Salzburg.

  [13] Martin Held. *Computational Geometry*. lecture notes. SS 2018. URL: https://www.cosy.sbg.ac.at/~held/teaching/compgeo/comp_geo.html

[3] If we cannot tell that then we cannot judge whether an algorithm is correct or not. Then we we leave the terrain of science.

- The standard book on computational topology is by Herbert Edelsbrunner and John Harer. Herbert Edelsbrunner is a driving figures of both fields, computational geometry and computational topology, and was professor at University of Urbana-Champaign and Duke University and moved 2008 to IST Austria.

  [8] Herbert Edelsbrunner and John Harer. *Computational Topology - an Introduction*. American Mathematical Society, 2010. ISBN: 978-0-8218-4925-5

- This is a more recent book that puts more emphasis to discrete geometry.

  [6] Satyan L. Devadoss and Joseph O'Rourke. *Discrete and Computational Geometry*. Princeton University Press, 2011. ISBN: 9781400838981

Part I

Numerical programming

Chapter 1

Basics of numerical programming

1.1 Representation of numbers

1.1.1 The b-adic expansion

A suitable representation of numbers plays an important role when operating with numbers in a computational fashion. The representation of forty-two in the decimal system[1] as "42" is by far superior to the roman numeral "XLII" for the execution of arithmetic operations, such as addition and multiplication. The following quote is by Whitehead[2] [22, p. 59]:

> By relieving the brain of all unnecessary work, a good notation sets it free to concentrate on more advanced problems, and in effect increases the mental power of the race. Before the introduction of the Arabic notation, multiplication was difficult, and the division even of integers called into play the highest mathematical faculties. Probably nothing in the modern world would have more astonished a Greek mathematician than to learn that, under the influence of compulsory education, a large proportion of the population of Western Europe could perform the operation of division for the largest numbers. This fact would have seemed to him a sheer impossibility. The consequential extension of the notation to decimal fractions was not accomplished till the seventeenth century. Our modern power of easy reckoning with decimal fractions is the almost miraculous result of the gradual discovery of a perfect notation.

Let us restrict ourselves to non-negative real numbers for now. The representation of forty-two as $42 = 4 \cdot 10^1 + 2 \cdot 10^0$ allows us to encode this number by the sequence $(2, 4)$ of its digits. In fact, every non-negative real number can be represented in the decimal system as a sequence of digits. More precisely, for each real number $z \geq 0$ there is a doubly infinite sequence $(\dots, a_{-2}, a_{-1}, a_0, a_1, \dots)$ of digits $a_i \in \{0, \dots, 9\}$, such that

$$z = \cdots + a_1 \cdot 10^1 + a_0 \cdot 10^0 + a_{-1} \cdot 10^{-1} + a_{-2} \cdot 10^{-2} \cdots = \sum_{i=-\infty}^{\infty} a_i \cdot 10^i.$$

There is nothing magical about the 10 in the decimal system: The decimal system is just a special case of the *b-adic number expansion*. That is, for every integer $b > 1$ and every real $z \geq 0$

[1] That is, in the base-ten positional number system.

[2] Whitehead was a mathematician and philosopher. Together with his student Bertrand Russel he wrote the seminal book *Principia Mathematica*.

there is a doubly infinite sequence (a_i) of *digit* $a_i \in \{0, \dots, b-1\}$ such that

$$z = \sum_{i=-\infty}^{\infty} a_i \cdot b^i. \tag{1.1}$$

For instance, the number π – which is not rational, not even algebraic – would be represented by $(\dots, 5, 1, 4, 1, 3, 0, 0, 0, \dots)$ where $a_0 = 3$ is the digit at position 0.

The number b is called the *basis* of the b-adic number expansion. When calculating with ten fingers then typically the decimal system is easier, for a digital computer with two-valued bits we favor the binary system – the 2-adic expansion –, and for other applications the basis 8 (octal) or the basis 16 (hexadecimal) are more favorable. It is a common notation to write the basis as subindex if there might be confusion. By this notation we have $11 = 11_{10} = 13_8 = 1011_2$.

If there is an index m such that the digits $a_i = 0$ for all $i > m$ then the b-adic expansion can be reversed, i.e., for each such sequence there is a non-negative real z that fulfills eq. (1.1).[3] Moreover, the representation is also unique if there is no index k such that $a_i = b - 1$ for all $i < k$. For instance, in base 10 the real number 1 is both represented as 1.0 by the sequence $(\dots, 0, 0, 0, 1, 0, \dots)$ and the periodic number $0.\overline{9}$ represented by $(\dots, 9, 9, 9, 0, 0, \dots)$.

1.1.2 b-adic fixed-point and floating-point numbers

A real-world computer can only handle a finite number of digits, so we have to restrict the sequence (a_i) of digits to a finite number of non-zero digits. For instance, the data type `uint32_t` in the programming language C holds only 32 binary digits and is therefore restricted to finitely many digits $a_0, \dots, a_{31}$.

Fixed-point numbers. In general, however, we could simply choose a fixed, finite number m of *integral digit* and a finite number n of *fractional digits* and obtain a so-called *fixed-point number*

$$z = \underbrace{a_{m-1}b^{m-1} + \cdots + a_1 b^1 + a_0 b^0}_{\text{integral part}} + \underbrace{a_{-1}b^{-1} + \cdots + a_{-n}b^{-n}}_{\text{fractional part}} = \sum_{i=-n}^{m-1} a_i b^i. \tag{1.2}$$

This fixed-point number is represented by $m + n$ digits $a_{m-1}, \dots, a_0, \dots, a_{-n}$, and all others are zero. That is, the (decimal) point is at a fixed position. For the boundary case $n = 0$ we have ordinary integers and for $n > 0$ we can also represent non-integer numbers. If we choose $m = 0$ then we obtain numbers in the interval $[0, 1)$.

As an example, the number π would be approximated by a fixed-point number to base 10 with $m = 2$ integral digits and $n = 5$ fractional digits as

$$\pi \approx 03.14159,$$

where we intentionally also print the leading digit $a_{m-1} = a_1 = 0$. Let us denote by $Q_b^{m,n}$ the set of fixed-point numbers with m integral and n fractional digits. We can rephrase eq. (1.2) such that

$$z = \left(\sum_{i=0}^{m+n-1} a_{i-n} b^i \right) \cdot b^{-n} \tag{1.3}$$

[3] If there is no such index then the series in eq. (1.1) goes to infinity. However, if there is such an index then there is a smallest such index and we have $a_m \neq 0$ or the sequence is all zero, i.e., $z = 0$.

and therefore $z = k \cdot b^{-n}$ for some integer k. From eq. (1.3) we see that a fixed-point number is just a $(m+n)$-bit integer scaled by b^{-n}. Another conclusion we can draw from this point of view is that the set $Q_b^{m,n}$ has the particular property of being equidistantly distributed on the interval $[0, b^m)$ with a step size of b^{-n} between neighboring numbers. We will see that floating-point numbers lack this property.

Floating-point numbers. The floating-point number representation extends the fixed-point number representation by the information which index domain $-n, \dots, m$ is used.

Assume we are given a real number $z > 0$ and its b-adic expansion

$$z = a_m b^m + a_{m-1} b^{m-1} + \cdots$$

with $a_m \neq 0$. That is, m is the largest index of a non-zero digit a_m. We obtain z' from z by *chopping*[4] to $p = m - k + 1$ digits by setting

$$z' = \underbrace{a_m b^m + a_{m-1} b^{m-1} + \cdots + a_k b^k}_{p \text{ summands}} \approx z. \tag{1.4}$$

We can factor out b^{m+1} and obtain the so-called *normalized floating-point representation*

$$z' = \underbrace{(a_m b^{-1} + \cdots + a_k b^{-p})}_{a} \cdot b^{m+1} = a \cdot b^{m+1}. \tag{1.5}$$

The number a is called *mantissa*[5], the number p is the *mantissa length* and the number $m+1$ is the *exponent*. Note that the mantissa is in the interval $[0, 1)$. Following these definitions the number π has as normalized floating-point representation to base 10 and chopped to 3 digits

$$\pi \approx 0.314 \cdot 10^1.$$

Its mantissa is 0.314 and its exponent is 1.

For technical-didactical reasons we assumed so far that $z > 0$, otherwise the index m in eq. (1.4) would not exist. However, once we fix m in eq. (1.5), we can of course set all digits $a_i = 0$ to represent $z' = 0$. But of course, there is no normalized floating-point representation of zero as we cannot "normalize zero".

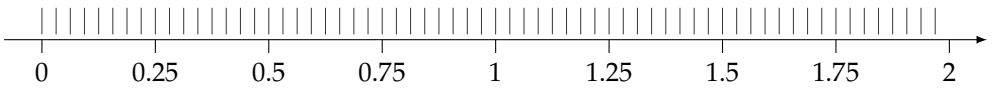
Distribution of numbers. In fig. 1.1 we see the distribution of fixed- and floating-point numbers. With loss of generality, we chose two as basis. For reasons of comparison, we use 6 digits to encode the numbers and cover the number range $[0, 2)$. For fixed-point numbers we chose 1 integral digit and 5 fractional digits, which leads to an equidistant distribution with a step size of $2^{-5} = 0.03125$. For the floating-point numbers we chose 4 digits for the mantissa and 2 digits for the exponent. We chose the exponent range as $-2, \dots, 1$. That is, each interval $[0, 2^e)$ for $e \in \{-2, \dots, 1\}$ contains $2^4 = 16$ equally spaced numbers.

As we can clearly see, floating-point numbers do not possess a "uniform resolution", but it rather depends on the interval $[0, 2^e)$ in which a value falls into. This simple fact has profound consequences. For instance, in general we cannot represent the sum of two floating-point numbers in the same representation when we move to a coarser interval. In contrast, fixed-point numbers can be precisely added.[6]

[4] Dt. Abschneiden

[5] Dt. Mantisse

[6] The sum may exceed the range covered and cause an overflow, but this issue is of an essentially different nature.



(a) Fixed-point numbers with 1 integral digit and 5 fractional bits.

(b) Floating-point numbers with 4 digits for the mantissa and 2 bits for the exponent.

Figure 1.1: Distribution of fixed- and floating-point numbers to cover the range $[0, 2)$. In both cases the basis is two and in total 6 digits were used for their representation.

Note that there are $2^6 = 64$ distinct fixed-point numbers, but only 40 distinct floating-point numbers. For instance, the number zero is represented four times, namely as $0.0000_2 \cdot 2^k$ for each $k \in \{-2, 1, 0, 1\}$. But also the identity $0.1100_2 \cdot 2^{-1} = 0.0110_2 \cdot 2^0 = 0.0011_2 \cdot 2^1$ leads to multiple representations of the same number; the latter two representations are called *denormalized* – in contrast to normalized – since their *most-significant digit*[7] in the mantissa is zero.

If we add one further digit to fixed-point representation as fractional digit then we place between each two neighboring numbers in fig. 1.1a an additional number. The same is true for fig. 1.1b if we add a digit to the mantissa. If, however, we add a digit to the exponent then we can extend the range of the exponent to $-6, \ldots, 1$ and we effectively add 16 numbers for each of the four intervals $[0, 2^k)$ with $k \in \{-6, -5, -4, -3\}$. The smallest positive number we can then represent is $0.0001_2 \cdot 2^{-6} = 2^{-10} = 0.0009765625$, while in $Q_2^{1,6}$ the smallest positive number is $2^{-6} = 0.015625$.

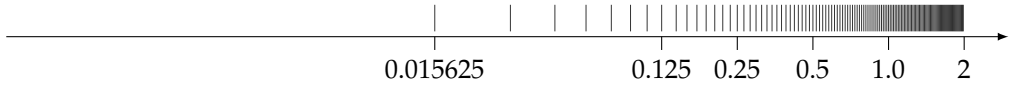
So the floating-point representation allows us to cover a much larger range in terms of order of magnitude. In other words, on a logarithmic scale, as shown in fig. 1.2, the floating-point numbers are more evenly spaced than fixed-point numbers. Any 32-bit fixed-point number format (like the integers) spans less than 10 decimal orders of magnitude, while a 32-bit floating-point number in IEEE 754 spans 77 decades. The diameter of the observable universe is probably less than 10^{27} m and the Planck length is more than 10^{-35} m, so we could express the entire range of 63 decades with 32-bit floating-point numbers of IEEE 754. Floating-point numbers can even express the wealth of Bill Gates, but 32-bit integers cannot. Any 64-bit fixed-point number format spans about 19 decades, while the 64-bit floating-point numbers of IEEE 754 spans more than 616 decimal orders of magnitude.

1.1.3 Hardware number formats

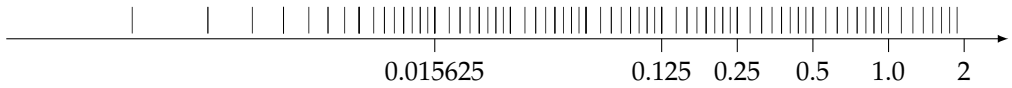
The previous introduction of fixed- and floating-point numbers based on the b-adic expansion is suitable for mathematical analysis; we will make use it in section 1.2. In real-world applications, however, the basis b is essentially always two[8], but we also need to deal with negative numbers, which we ignored so far. So let us assume $b = 2$ as the basis for the remainder of this section. A digit is therefore called a *bit* (binary digit).

[7] The most-significant digit a_i is the one with the largest index i. It has the most influence on the number.

[8] Indeed, the binary coded decimal (BCD) format uses bits to encode numbers, but only to represent decimal digits. That is, the arithmetic is actually based on the decimal system. Legal requirements that financial software must be free of rounding errors essentially implies to do the math in the decimal system. For instance, 0.1_{10} has infinitely many non-zero digits in the 2-adic expansion.



(a) Fixed-point numbers with 1 integral digit and 6 fractional digits.



(b) Floating-point numbers with 4 digits for the mantissa and 3 digits for the exponent.

Figure 1.2: Distribution of fixed- and floating-point numbers on a logarithmic scale. In both cases the basis is two and 7 digits were used for their representation.

Integers

Non-negative integers can directly be treated as fixed-point numbers with $n = 0$ fractal bits and $m > 0$ integer bits. So an n-bit *unsigned integer* with the bits $a_{n-1}, \dots, a_0$ is an element of $Q_2^{n,0}$ and encodes the number

$$z = \sum_{i=0}^{n-1} a_i 2^i. \tag{1.6}$$

By eq. (1.6), the number z can attain any integer number in the range $[0, 2^n - 1]$.

To also accommodate negative integers, one could simply let a_{n-1} encode the sign, which would give the one's complement format.[9] In this format we would actually distinguish between a $+0$ and -0, however arithmetic with one's complement is more complicated. Instead, the *two's complement*, which has been proposed by John von Neumann, became the dominant format. In two's complement an n-bit *signed integer* with the bits $a_{n-1}, \dots, a_0$ encodes the number

$$z = -a_{n-1} 2^{n-1} + \sum_{i=0}^{n-2} a_i 2^i. \tag{1.7}$$

By eq. (1.7), the number z can attain any integer number in the range $[-2^{n-1}, 2^{n-1} - 1]$.

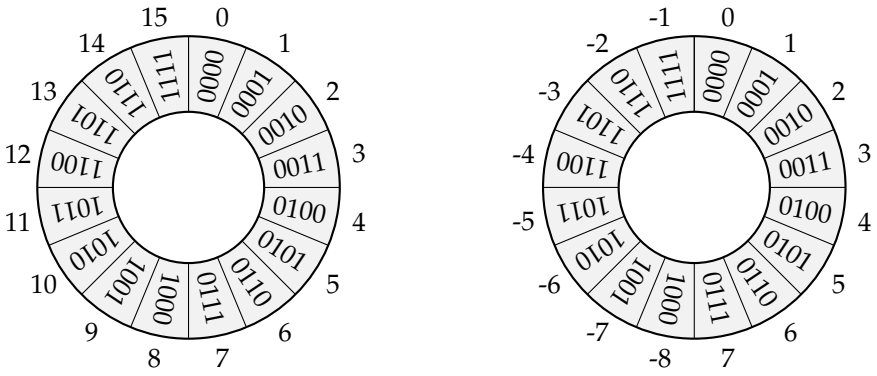
Figure 1.3 illustrates eq. (1.6) and eq. (1.7). It also illustrates that arithmetic with signed numbers in two's complement is essentially the same as arithmetic with unsigned numbers on a bit level: If we want to add the number 5 to a number then we simply go 5 steps in positive (clockwise) direction. In fig. 1.3 we have 4-bit unsigned and signed integers. In case of signed integers, adding 5 to -2 (1110) gives us 3 (0011). Likewise, for unsigned integers adding 5 to 14 (1110) gives 3 (0011) modulo 2^4. This example illustrates a key aspect: On a bit level we executed the same calculation and it can be trivially implemented in hardware.[10]

IEEE 754 floating-point numbers

The prevalent floating-point number format in real-world processors is given by the IEEE 754 standard [1] of the year 1985. The most common binary floating-point data types from this

[9] To be precise, in the *one's complement* we negate a number by flipping all bits, so -1 is encoded by all bits set except the least-significant bit.

[10] A cascade of n full adders realizes an n-bit adder digital circuit.



(a) 4-bit unsigned integers (b) 4-bit signed integers

Figure 1.3: Unsigned and signed 4-bit integers. Adding numbers works the same way on a bit level.

standard are *single precision* (32 bit) and *double precision* (64 bit), but there are others[11]. The binary formats of those two data types are illustrated in fig. 1.4. The mantissa a is obtained by

$$a = 1.M_2 = 1 + M \cdot 2^{-p},$$

where we put the mantissa bits M as fractional part behind "1." in the first interpretation, while in the second interpretation we see M as an unsigned integer.

This definition differs slightly from b-adic floating-point numbers: Note that for $b = 2$ the leading non-zero digit is always 1, so we do not need to explicitly store it. (The attentive reader may have noticed that this argument does not work for denormalized numbers.[12]) The real number z, which is encoded as a binary pattern according to fig. 1.4, is then given by

$$z = (-1)^S \cdot \underbrace{(1 + M \cdot 2^{-p})}_{a} \cdot 2^{E-B}, \tag{1.8}$$

where p is the mantissa length (23 for single precision, 52 for double precision) and B is the so-called *bias* (127 for single, 1023 for double).[13]

In addition to eq. (1.8) certain binary patterns have a distinguished meaning: A zero is represented by $E = M = 0$ and we distinguish between $+0$ and -0 depending on S. The largest possible exponent, when $E = 11\cdots1_2$, is used to represent $+\infty$ and $-\infty$, but also NaN (not a

[11] The Intel x87 floating-point unit also knows a 80 bit format with 64 bits of mantissa. The rise of machine learning in recent years led to the `bfloat16` format with 8 bits of exponent and 7 bits mantissa.

[12] For b-adic floating-point numbers we called any representation denormalized, where the most-significant digit of the mantissa was non-zero. In IEEE 754, the *denormalized* numbers are those where $E = 0$ and $M \neq 0$. That is, if $E = 0$ then we interpret the number as $0.M_2$. Handling denormalized numbers can lead to a performance penalty and often we can instruct compilers to be less strict in handling such numbers according the IEEE 754 standard.

[13] The exponent is therefore not encoded by two's complement!

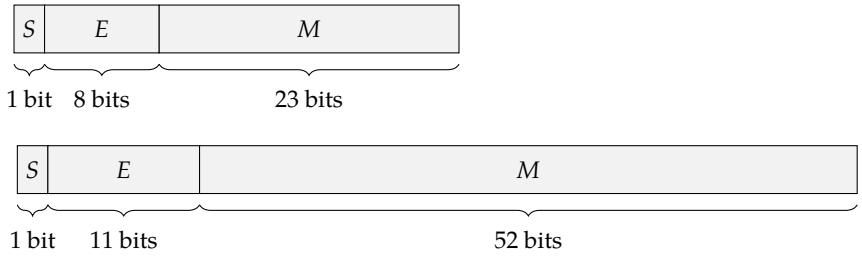


Figure 1.4: The binary layouts of IEEE 754 foating-point numbers with single and double precision. The sign bit is given by S, the exponent bits by E and the mantissa bits by M.

number).[14] This allows for the following arithmetic laws in IEEE 754:

$$\begin{aligned} 1.0 \cdot \pm 0.0 &= \pm 0.0 & -1.0 \cdot \pm 0.0 &= \mp 0.0 \\ 1.0 \cdot \pm \infty &= \pm \infty & -1.0 \cdot \pm \infty &= \mp \infty \\ 1.0 \div \pm 0.0 &= \pm \infty & -1.0 \div \pm 0.0 &= \mp \infty \\ 1.0 \div \pm \infty &= \pm 0.0 & -1.0 \div \pm \infty &= \mp 0.0 \\ \pm 0.0 \div \pm 0.0 &= \text{NaN} & \pm \infty \div \pm \infty &= \text{NaN} \\ \pm 0.0 \cdot \pm \infty &= \text{NaN} \end{aligned}$$

The IEEE 754 standard is extensive and the above presentation gives only a very limited excerpt. In fact, a fully featured implementation of IEEE 754 in a processor is far from being trivial.[15] Goldberg [10] provides many more details regarding programming with IEEE 754 floating-point numbers.

Fixed-point formats

Fixed-point numbers typically span a significantly smaller range than floating-point numbers, as we learned in section 1.1.2. However, fixed-point numbers still possess a couple of significant advantages:

Fixed-point arithmetic is exact and arithmetic operations do not introduce numerical errors. This is why fixed-point arithmetic is preferred over floating-point arithmetic in financial software, like in GNU Cash.

IEEE 754 compatible FPUs are hard to implement but fixed-point arithmetic is essentially as simple as integer arithmetic. This makes them favorable for small or cheap embedded processors, signal processors, or dedicated graphics hardware, which may lack a FPU. For instance, the infamous 3D computer game *Doom*[16] used 32 bit signed fixed-point numbers with $m = 15, n = 16$, where an increment was 2^{-16}. This allowed Doom to be run on Intel 386 machines.

A typical application in digital signal processing is as follows: An analog-digital converter produces a signal that is further processed, say, by a FIR filter. The domain of the signal is often considered to be in a unit interval, like $[0, 1]$ or $[-1, 1]$. This is why a *digital signal processor* (DSP) often provides native data types and instructions for fixed-point arithmetic.

[14] Actually, different cases of NaN are distinguished.

[15] Processor bugs in the FPU (floating-poin unit) are not entirely unusual. The best known example is probably the infamous FDIV bug of early Intel Pentium processors [18]. Also, in the embedded domain some processors only provide a limited feature set of the IEEE 754.

[16] Doom was released in 1993 and constitutes a milestone in games development not only for its 3D graphics.

Q format. For binary fixed-point number formats there are a couple of notations to define them. As a reference to the rational numbers $\mathbb{Q}$, Texas instruments popularized the so-called *Q format*:

- Q$m.n$ refers to signed fixed-point numbers with m integer bits and n fractional bits.
- Qn refers to the set of signed fixed-point numbers with n fractional bits and the number m of integral bits is implicitly given: It is either the number of remaining bits from the register size or is meant to be zero.

Unsigned integers are indicated by prefxing U to Q, so the set of UQ$m.n$ number equals the set $Q_2^{m,n}$. There is an ambiguity on whether to count the sign bit for signed numbers. One convention says the sign bit is included in m, so Doom would have used numbers in the format $Q16.16$. An alternative convention says that the sign bit is not part of m, and hence Doom would have used $Q15.16$. However, by considering $m + n$ and comparing with the register size, it is possible to resolve the ambiguity in practice.

If we have a binary representation of a number in Q$m.n$ or UQ$m.n$ format as illustrated in fig. 1.5 then we simply read the bit pattern as a signed or unsigned integer and think of it being scaled by 2^{-n}, compare also eq. (1.3).

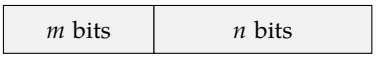


Figure 1.5: A number in Q$m.n$ or UQ$m.n$ format is interpreted as an integer scaled by 2^{-n}.

Embedded C. Having fixed-point arithmetic hardware is one thing, but we also require according mechanisms on the software side. For instance, the C programming language knows the data types `float` and `double`, which are the IEEE 754 floating-point number types in single- and double-precision. However, standard C – like C11 – does not know fixed-point number types and so often vendor-specific C compilers shipped extensions to accommodate fixed-point number types.

Since 2004 there is an ISO/IEC standard, briefly called *Embedded C*, which is based on DSP-C [7]. The current and second version from 2008 is called *ISO/IEC TR 18037:2008* [16].This standard adds fixed-point arithmetic data types and more.[17] The GCC C-compiler supports this standard since 2007 [9]. The Clang compiler of LLVM has some preliminary implementation since 2018 [5]. However, neither fully supports this standard yet.[18]

Embedded C defines to new type specifiers `_Fract` and `_Accum`, which can be used in combination with the type specifiers `short`, `long`, `signed`[19] and `unsigned` and gives rise to the 12 types in table 1.1. The specifier `_Fract` refers to a fixed-point data type with no integral bits, so the number range is within $[-1, 1)$ for signed types and $[0, 1)$ for unsigned types. In contrast, `_Accum` also defines integral bits and is, for instance, used when building sums of `_Frac` numbers, i.e., accumulating them. This is why the number of fractional bits of the `_Accum` types match the number of bits of the corresponding `_Fract` types.

[17] Often, embedded processors follow a Harvard architecture, where program and data memory do not reside in the same address space. Hence, there is a need to specify the address space to which a pointer in C refers to and Embedded C adds support for that.

[18] For instance, clang version 9.0 does not yet support conversion from fixed-point numbers to floating-point numbers or provides the corresponding standard header files. On the other hand, GCC only supports certain targets, for instance x86 is not supported for GCC version 9, but clang does.

[19] As usual for C, `signed` does not need to be explicitly specified.

In addition, there is a type specifier `_Sat` that makes a type a *saturating* fixed-point type. Adding two large (positive or negative) number of a saturating type does not cause an overflow, as with ordinary integer numbers, but results in the largest (positive or negative) number in the respective number range. This is often the intended behavior in signal processing, e.g., think of mixing audio signals. The header file `stdfix.h` defines `sat`, `fract`, and `accum` as the natural spelling of `_Sat`, `_Fract`, and `_Accum`. The following code listing illustrates the usage of these fixed-point number types:

```
sat fract x = -0.7r;                        // Suffix r for fract literals
sat fract y = -0.7r;
printf("x + y = %f\n", (double) (x+y));     // Prints -1.0 due to sat
sat accum a = 0.7k;                         // Suffix k for accum literals
sat accum b = 0.7k;
printf("a + b = %f\n", (double) (a+b));     // Prints 1.4 due to accum
```

As usual, the C language does not define the sizes of data types but minimum sizes, which are given in table 1.1. The actual sizes are implementation dependent and may differ between compilers, operating systems or processors. However, there are macros like `FRACT_FBIT`, which gives the number of fractional bits of `_Fract`, or `ULACCUM_IBIT` which gives the number of integral bits of `unsigned long _Accum`.

Type	Suffix	Min.	Type	Suffix	Min.
`short _Fract`	`hr`	Q7	`short _Accum`	`hk`	Q4.7
`_Fract`	`r`	Q15	`_Accum`	`k`	Q4.15
`long _Fract`	`lr`	Q23	`long _Accum`	`lk`	Q4.23
`unsigned short _Fract`	`uhr`	UQ7	`unsigned short _Accum`	`uhk`	UQ4.7
`unsigned _Fract`	`ur`	UQ15	`unsigned _Accum`	`uk`	UQ4.15
`unsigned long _Fract`	`ulr`	UQ23	`unsigned long _Accum`	`ulk`	UQ4.23

Table 1.1: ISO/IEC TR 18027 fixed-point data types with literal suffix and minimum sizes. Here the Q-format does not add the sign bit to the integral bits.

1.2 Floating-point arithmetic

1.2.1 Rounding

In order to obtain from a real number z a floating-point number $\overline{z}$ with a given mantissa length, we could simply perform the b-adic expansion and apply chopping. A more accurate result, however, can be obtained if we apply the "usual" rounding: We replace in eq. (1.4) the digit a_k by $a_k + 1$ if $a_{k-1} \geq b/2$. However, if this results in $a_k = b$ then we set $a_k = 0$ and increment a_{k+1} by one (carry-over), and so forth.

We denote the result by $\mathrm{rd}_{s,b}(z)$ and call it the *machine number* obtained by rounding to mantissa length s to the basis b. For example

$$\mathrm{rd}_{2,10}(0.134) = 0.13 \qquad \mathrm{rd}_{2,10}(0.135) = 0.14 \qquad \mathrm{rd}_{3,10}(0.1996) = 0.2,$$

where in the last example chopping would have resulted in 0.199, which is less accurate. Among all possible machine numbers of length s to the basis b the machine number $\mathrm{rd}_{s,b}(z)$ is closest to

the real number z. Also note that rounding and chopping yields a rational number from a real number. All machine numbers are rational numbers.[20]

The above method ignores the size of the resulting exponent and the issue of *overflow* for the sake of simplicity. Of course, a real-world implementation of IEEE 754 needs to deal with these cases. If the exponent becomes too small then we could simply set the resulting number to $+0$ or -0. Similarly, if the exponent becomes too large then we can set the resulting number to $+\infty$ for $-\infty$. Besides that IEEE 754 also knows various kinds of rounding modes. The one explained above is known as *round to nearest*,[21] but there are also *round toward* 0, *round toward* $+\infty$, and *round toward* $-\infty$. The rounding mode then again influences how overflow is handled.[22]

1.2.2 Error and accuracy

Let $\tilde{z}$ denote some approximation of the number z. Then $|z - \tilde{z}|$ is called the *absolute error* of the approximation and

$$\left|\frac{z - \tilde{z}}{z}\right|$$

is called the *relative error*. Note that the relative error is only defined for $z \neq 0$. For instance, let $\tilde{\pi} = 3.14$ be an approximation of $\pi = 3.14159\ldots$ then the absolute error is $0.00159\ldots$ and the relative error is $\frac{0.00159\ldots}{\pi} \approx 5 \cdot 10^{-4}$, which is about 0.05 %. Relative errors are often given as percentages when adequate.

If $\tilde{z}$ was obtained by rounding, i.e., $\tilde{z} = \mathrm{rd}_{s,b}(z)$, then we also speak of an *absolute* resp. *relative rounding error*. For the rounding procedure given in section 1.2.1 the bounds

$$|z - \tilde{z}| \leq \frac{b^k}{2} \quad \text{and} \quad \left|\frac{z - \tilde{z}}{z}\right| \leq \frac{b^{k-m}}{2} = \frac{b^{1-s}}{2},$$

hold, where the latter is only defined for $z \neq 0$. The bound for the relative rounding error is also called *relative machine accuracy* or *machine epsilon*. Hence, for IEEE 754 single-precision numbers we obtain a machine accuracy of $2^{1-24}/2 = 2^{-24} \approx 5.96 \cdot 10^{-8}$ and for double-precision numbers we obtain $2^{1-53}2/=2^{-53} \approx 1.11 \cdot 10^{-16}$. That is, a single-precision number is accurate for up to about 7 decimal digits and a double-precision number for about 16 decimal digits.[23]

1.2.3 Machine operations

Machine operations result in machine numbers. That is, the result of machine operations on machine numbers are rounded again in order to again obtain machine numbers. Instead of the usual basic arithmetic operations $+, -, \ldots$ a processor therefore performs the following machine

[20] In addition, IEEE 754 knows special "numbers" like $+0$, -0, $+\infty$, $-\infty$ or NaN.

[21] Actually, IEEE 754 round to nearest applies *round half to even*, where the tie break is not rounding up but rounding to the nearest even integer. So 23.5 and 24.5 are both round to 24. This is also known as the *banker's rounding*. It reduces the accumulation of rounding errors, whereas tie breaking by rounding up has a bias towards $+\infty$.

[22] For instance, if we choose round toward $-\infty$ or round toward 0 and there is a overflow in positive direction then the result is not $+\infty$ but the largest finite positive machine number.

[23] The IEEE 754 standard does not define the term *machine accuracy* which unfortunately led to different definitions of this term. Another common definition is: The machine accuracy is b^{1-s}, which is the difference between 1 and the next larger machine number. This definition leads to a machine accuracy of $1.19 \cdot 10^{-7}$ resp. $2.22 \cdot 10^{-16}$ for single resp. double precision. This definition is used for the ISO C standard, Python, Mathematica, MATLAB, or Octavia.

operations $\oplus, \ominus, \ldots$ that involves rounding:

$$\begin{aligned} x \oplus y &= \mathrm{rd}_{s,b}(x + y) \\ x \ominus y &= \mathrm{rd}_{s,b}(x - y) \\ x \odot y &= \mathrm{rd}_{s,b}(x \cdot y) \\ x \oslash y &= \mathrm{rd}_{s,b}(x / y) \end{aligned}$$

Since a change of sign is not influenced by rounding, we could reduce the definition of $x \ominus y$ to addition, i.e., $x \ominus y = \mathrm{rd}_{s,b}(x - y) = \mathrm{rd}_{s,b}(x + (-y)) = x \oplus (-y)$. That is, the following arithmetic law holds for machine operations:

$$x \ominus y = x \oplus (-y)$$

However, other than the above rule, rounding has typically far-reaching consequences! For instance, the order of operations matters because the usual associative law does not hold anymore. For example, let $b = 10$ and $s = 2$ then

$$\begin{aligned} (0.50 \oplus 0.54) \oplus (-0.53) &= 1.0 \oplus (-0.53) = 0.47 \\ 0.50 \oplus (0.54 \oplus (-0.53)) &= 0.50 \oplus 0.01 = 0.51 \end{aligned}$$

In a Python interpreter we can easily demonstrate this with IEEE 754 floating point numbers as well:

```
>>> e = 2**(-53)
>>> (1.0 + e) + e == 1.0
True
>>> 1.0 + (e + e) == 1.0
False
```

The operations also do not adhere to the distributive law as $((x \oplus y) \odot z)$ is not necessarily $(x \odot z) \oplus (y \odot z)$. Hence, different compilers, different compiler versions, different optimization levels or seemingly trivial changes in the source code[24] can influence the order of machine operations and therefore alter the computational results, even though nothing has changed in a "usual mathematical interpretation" based on real number arithmetic. We need to impute an error corresponding to the machine accuracy to each single operation.

As a consequence the comparison of machine numbers can (virtually) never be done exactly but need to be performed by means of thresholds. Those so-called *threshold-based* or *epsilon-based comparisons* introduce a fixed application-dependent $\epsilon > 0$ in order to define the following comparison operators:

$$\begin{aligned} x \doteq y &\Leftrightarrow |x - y| \leq \epsilon \\ x \dot{\neq} y &\Leftrightarrow \neg(x \doteq y) \\ x \dot{<} y &\Leftrightarrow x < y \wedge x \dot{\neq} y \\ x \dot{>} y &\Leftrightarrow x > y \wedge x \dot{\neq} y \\ x \dot{\leq} y &\Leftrightarrow x \dot{<} y \vee x \doteq y \\ x \dot{\geq} y &\Leftrightarrow x \dot{>} y \vee x \doteq y \end{aligned}$$

However, also for these comparison operators the usual laws do not holds, for instance the transitive law: It can be that $x \doteq y$ and $y \doteq z$, but still $x \dot{\neq} z$.

[24] Such as switching from a debug build to a release build.

Finally, it should be mentioned that the arithmetic operations defined in this section do not always correspond to the implementation of some FPUs. For instance, the x87 FPU implements an 80 bit *extended double-precision* floating-point number data type with a 64 bit mantissa. Floating-point operations are done with this 80 bit data type and rounding to a 23 resp. 52 bit mantissa for single or double precision is only done when the result is written into the CPU registers. The more modern SSE floating-point unit does not possess the extended double precision registers and performs operations in single or double precision, which may lead to different results.

1.3 Numerical analysis

1.3.1 Numerical algorithms

An *algorithm* transforms input into output in order to solve a specific problem. The problem *sorting* considers a list of numbers[25] as input and asks for a sorted permutation of the list as output. A well known algorithm to the problem *sorting* is for instance *merge sort*. While in algorithm theory we typically ask for the time and space complexity of algorithm, which captures speed and memory footprint, in numerical analysis we also ask for the "numerical quality" of algorithms.

Many problems in the field of numerical mathematics can be phrased as computing a mathematical map that takes a real n-tuple $(x_1, \dots, x_n)$ as input and produces a real m-tuple $(y_1, \dots, y_m)$ as output. That is, the task is to compute a given function

$$\varphi\colon \mathbb{R}^n \to \mathbb{R}^m\colon x \mapsto \varphi(x),$$

where $x = (x_1, \dots, x_n)$ is the input tuple and $\varphi(x)$ is the output tuple. For instance, let us consider the problem of computing the roots of a quadratic polynomial $az^2 + bz + c$ in the real variable z. Here the input tuple would be $(a, b, c) \in \mathbb{R}^3$ and the output tuple would be $(z_1, z_2) \in \mathbb{R}^2$ of the two roots.[26] A concrete algorithm for this problem could follow the well known formula

$$z_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

in order to compute the roots. Each of the operations $+, \cdot, \sqrt{}, \dots$we interpret as a computational step in the algorithm, which are composed together to form a map $\tilde{\varphi}$ that constitutes the algorithm.

Some problems φ possess the property that little changes in x lead to large changes in $\varphi(x)$. If the input contains errors – rounding errors, measurement errors, et cetera – and we obtain therefore an approximation $\tilde{x}$ of the original input x then this can have large impact on the result. The so-called *condition* of a problem captures this property of a problem. The condition is independent of the specific algorithm but is intrinsic to the problem, it lies in the nature of the specific problem. More precisely, the condition considers the term

$$\|\varphi(\tilde{x}) - \varphi(x)\|,$$

where $\|.\|$ denotes the norm of a vector.

Let us now consider for a given problem φ an arbitrary algorithm $\tilde{\varphi}$. Different algorithms may solve the same problem φ in different ways. The *stability* of an algorithm tells us how sensitive a

[25] In general a list of elements from a partially ordered set (poset). This not only includes numbers with the ordinary order, $\geq$, but for instance also strings with the lexicographical order.

[26] We known from calculus that a polynomial of degree two has zero real roots or two (which might be equal). For the sake of simplicity, we assume that roots exist.

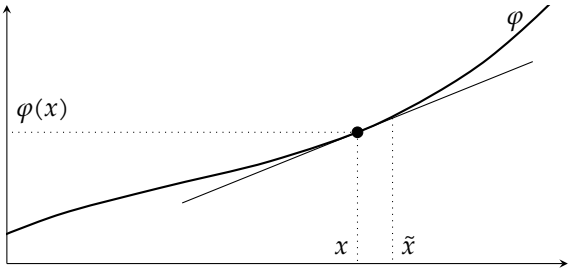


Figure 1.6: The tangent on the function graph of φ at position x has the slope $\varphi'(x)$. At a nearby position $\tilde{x}$ the function evaluates to $\varphi(\tilde{x})$, which can be approximated by the tangent.

given algorithm is to changes in the input data. The stability is a property of the algorithm and considers the term

$$\|\tilde{\varphi}(\tilde{x}) - \tilde{\varphi}(x)\|.$$

Some algorithms $\tilde{\varphi}$ for continuous problems consider certain methods of discretization. For instance in order to differentiate a function $f\colon \mathbb{R} \to \mathbb{R}$ at a position x we could consider $(f(x + h) - f(x))/h$, but of course the result depends on our choice of the *discretization step size* h. The *consistency* of a numerical algorithm tells us to what extent the algorithm introduces numerical errors independent of errors in the input data. More precisely, the consistency of an algorithm considers the term

$$\|\tilde{\varphi}(x) - \varphi(x)\|.$$

1.3.2 Condition of a problem

Let us consider a problem $\varphi\colon \mathbb{R}^n \to \mathbb{R}^m$, which takes as input $x \in \mathbb{R}^n$ and produces an output $\varphi(x) \in \mathbb{R}^m$ and let us assume that φ can be differentiated.[27]

For the simple case where $n = m = 1$ we have a function $\mathbb{R} \to \mathbb{R}$ and we can consider its function graph, see fig. 1.6. The function φ can be approximated by a linear function at any position $x \in \mathbb{R}$. That is, if we consider a $\tilde{x} \in \mathbb{R}$ close to x then

$$\varphi(\tilde{x}) \doteq \varphi(x) + \varphi'(x) \cdot (\tilde{x} - x),$$

where $\doteq$ means that the $=$ holds only approximately.[28] Similarly, we denote by $\dot{\leq}$ that $\leq$ holds only approximately. Then we have

$$\|\varphi(\tilde{x}) - \varphi(x)\| \dot{\leq} \|\varphi'(x)\| \cdot \|\tilde{x} - x\|. \tag{1.9}$$

The above lines were motivated for the case $n = m = 1$. However, the beauty of mathematics unfolds here in a way such that ineq. (1.9) also holds for $n, m \geq 1$ if $\|\varphi'\|$ is suitably interpreted for

[27] If φ cannot be differentiated at position x then there is an $\epsilon > 0$ such that $\|\varphi(x) - \varphi(\tilde{x})\| > \epsilon$ for some $\tilde{x} \in [x-\delta, x+\delta]$, no matter how small $\delta > 0$ is. That is, we cannot make the output error smaller, no matter how small we make the input error. We will see later that this means that the relative condition is essentially infinite, which means that the problem is in some sense infinitely ill-conditioned.

[28] We can interpret the right-hand side as a Taylor polynomial of degree 1. So the remainder of the Taylor series is in $O(\|\tilde{x} - x\|^2)$, i.e., the error of $\doteq$ converges to zero at order $\|\tilde{x} - x\|^2$.

higher dimensions: Let $\varphi(x) = (\varphi_1(x), \dots, \varphi_m(x))$ then we denote by $\varphi'(x)$ the Jacobian matrix

$$\varphi'(x) = \begin{pmatrix} \frac{\partial \varphi_1}{\partial x_1}(x) & \dots & \frac{\partial \varphi_1}{\partial x_n}(x) \\ \vdots & & \vdots \\ \frac{\partial \varphi_m}{\partial x_1}(x) & \dots & \frac{\partial \varphi_m}{\partial x_n}(x) \end{pmatrix}.$$

The Jacobian of $\varphi\colon \mathbb{R}^n \to \mathbb{R}^m$ contains the partial derivatives of all m elements of $(\varphi_1(x), \dots, \varphi_m(x))$ by all n dimensions of $x = (x_1, \dots, x_n)$. Hence, it generalizes the tangent slope in fig. 1.6 and if we would have $n = m = 1$ then the Jacobian is just the ordinary derivative of $\varphi\colon \mathbb{R} \to \mathbb{R}$ at a position x. Next, for a matrix $A = (a_{ij})$ we denote by $\|A\| = \sqrt{\sum_i \sum_j a_{ij}^2}$ the matrix norm of A, or more precisely, the Euclidean matrix norm.[29] Hence, we have

$$\|\varphi'(x)\| = \sqrt{\sum_{i=1}^{n} \sum_{j=1}^{m} \left(\frac{\partial \varphi_j}{\partial x_i}(x) \right)^2}.$$

Using this interpretation for $\|\varphi'\|$ generalizes ineq. (1.9) to higher dimensions.[30] Note that ineq. (1.9) tells us by how much the output of φ may change at most if we change the input. This immediately motivates the definition of the *absolute condition* of φ by

$$\kappa_{\text{abs}} = \|\varphi'(x)\|$$

and we obtain

$$\|\varphi(\tilde{x}) - \varphi(x)\| \dot{\leq} \kappa_{\text{abs}} \cdot \|\tilde{x} - x\|.$$

We can rearrange this inequality to

$$\frac{\|\varphi(\tilde{x}) - \varphi(x)\|}{\|\varphi(x)\|} \dot{\leq} \frac{\|x\|}{\|\varphi(x)\|} \kappa_{\text{abs}} \cdot \frac{\|\tilde{x} - x\|}{\|x\|}.$$

This inequality motivates the definition of the *relative condition* of φ as

$$\kappa_{\text{rel}} = \frac{\|x\|}{\|\varphi(x)\|} \kappa_{\text{abs}}$$

and we obtain

$$\frac{\|\varphi(\tilde{x}) - \varphi(x)\|}{\|\varphi(x)\|} \dot{\leq} \kappa_{\text{rel}} \cdot \frac{\|\tilde{x} - x\|}{\|x\|}.$$

We call a problem being *ill-conditioned*[31] if κ_{abs} or κ_{rel} are significantly greater than 1 and otherwise *well-conditioned*[32]. However, there is no general rule to what "significantly" actually means, so it depends on the specific application at hand.

[29] We can interpret the set of real $m \times n$ matrices as a vector space V of dimension $m \cdot n$. In general, for a vector space V a norm $\|.\|$ is any map $V \to \mathbb{R}$ with the following properties: $\|x\| \geq 0, \|x\| = 0 \Leftrightarrow x = 0, \|\lambda x\| = |\lambda| \|x\|, \|x + y\| \leq \|x\| + \|y\|$ for all $\lambda \in \mathbb{R}$ and $x \in V$. The Euclidean norm is also known as the 2-norm, which is a special case of the p-norm. For $V = \mathbb{R}^d$ and $x = (x_i) \in \mathbb{R}^d$ the p-norm of x is defined by $\|x\|_p = \sqrt[p]{\sum_i |x_i|^p}$. The 1-norm is also known as sum norm as $\|x\|_1 = \sum_i |x_i|$ and the ∞-norm is also known as maximum norm $\|x\|_\infty$ because $\lim_{p \to \infty} \|x\|_p = \max_i |x_i|$.

[30] Note that for $n = m = 1$ actually equality holds in ineq. (1.9). However, for higher dimensions this is not true any longer.

[31] Dt. schlecht konditioniert

[32] Dt. gut konditioniert

As a simple example we consider the calculation of square roots, so our problem is given by $\varphi(x) = \sqrt{x}$. We obtain $\kappa_{\text{abs}} = |\varphi'(x)| = |\frac{1}{2\sqrt{x}}|$ and $\kappa_{\text{rel}} = |\frac{x}{\sqrt{x}} \cdot \frac{1}{2\sqrt{x}}| = \frac{1}{2}$. The relative condition is unconditionally good, however the absolute condition is bad when x is close to zero. Also note that the absolute condition, but also the relative condition, is not defined at $x = 0$.

Condition of addition and subtraction. We would like to determine the condition of the addition resp. subtraction of two real numbers. So we consider $\varphi\colon \mathbb{R}^2 \to \mathbb{R}$ with $\varphi(x_1, x_2) = x_1 + x_2$. The partial derivatives of φ with respect to x_1 is 1 and the same holds for x_2. Hence, by definition

$$\varphi' = \begin{pmatrix} \frac{\partial \varphi}{\partial x_1} & \frac{\partial \varphi}{\partial x_2} \end{pmatrix} = \begin{pmatrix} 1 & 1 \end{pmatrix}.$$

The norm of this matrix gives the absolute condition:

$$\kappa_{\text{abs}} = \|\varphi'\| = \sqrt{1^2 + 1^2} = \sqrt{2}.$$

The relative condition[33] is therefore

$$\kappa_{\text{rel}} = \sqrt{2} \cdot \frac{\sqrt{x_1^2 + x_2^2}}{|x_1 + x_2|}.$$

Note that when $x_1 = -x_2$ then the relative condition is undefined. But even if $x_1 \approx -x_2$ then κ_{rel} is becoming very large and hence addition becomes very ill-conditioned. This leads to a very important observation:

> Adding two numbers with similar absolute values but different sign – or subtracting two similar numbers – is very ill-conditioned!

This phenomenon is called *cancellation*: When subtracting two similar numbers the digits cancel each other out and in the floating-point representation the remaining digits constitute the remaining significance resp. accuracy.

In the following example we calculate the difference $\pi - \sqrt[3]{31}$ using 4-digit floating-point to the base 10. We obtain $\text{rd}_{4,10}(\pi) = 0.3142 \cdot 10^1$ and $\text{rd}_{4,10}(\sqrt[3]{31}) = 0.3141 \cdot 10^1$ from which we calculate the difference $0.1 \cdot 10^{-2}$. The digits $3, 1, 4$ canceled each other out. The exact result would have been $\pi - \sqrt[3]{31} = 0.0212\ldots \cdot 10^{-2}$. The absolute error is just $7.88\ldots \cdot 10^{-4}$. However, the relative error is about 370 %, although the relative rounding errors are no larger than about 0.05 %!

1.3.3 Stability of an algorithm

Many numerical algorithms can be described as a sequence of elementary calculation steps. Each step solves an elementary problem $\psi_1, \ldots, \psi_n$ and altogether they solve the original problem φ and hence $\varphi(x) = \psi_n(\ldots \psi_2(\psi_1(x)) \ldots)$ or more briefly $\varphi = \psi_n \circ \cdots \circ \psi_1$. The stability[34] of an algorithm results from the conditions of the individual steps ψ_i: If one step is ill-conditioned then the entire algorithm is *instable*.

[33] If we would have used the 1-norm for the definitions of conditions then we would have $\kappa_{\text{rel}} = \frac{|x_1|+|x_2|}{|x_1+x_2|}$.

[34] In [17] the stability is also called *condition of an algorithm*.

As an example we consider the problem of solving the quadratic equation $x^2+2px+q = 0$. We are only interested in the larger of possibly two solutions and we simply assume that a solution actually exists.[35] Hence, our problem $\varphi\colon \mathbb{R}^2 \to \mathbb{R}$ is given by

$$\varphi(p,q) = -p + \sqrt{p^2 - q}.$$

A first algorithm could simply perform the stepwise calculations obtained from the formula above. We then obtain the following sequence of elementary operations:

procedure QuadEquation(p,q)
  $s \leftarrow p^2$
  $t \leftarrow s - q$
  $u \leftarrow \sqrt{t}$ ▷ u is $\sqrt{p^2-q}$
  $r \leftarrow -p + u$
  return r
end procedure

If $p^2 \gg q$ then $u \approx \sqrt{p^2} = |p|$. In other words, if $p > 0$ and $p^2 \gg q$ then the fourth step is ill-conditioned because $p \approx u$ and we suffer cancellation.

However, it is easy to show that

$$-p + \sqrt{p^2 - q} = \frac{-q}{p + \sqrt{p^2 - q}}.$$

The left and right hand sides are mathematically identical. In some sense, we just rephrased the problem, but it is still the same. However, the right hand side gives us a hint to a different algorithm:

procedure QuadEquationAlt(p,q)
  $s \leftarrow p^2$
  $t \leftarrow s - q$
  $u \leftarrow \sqrt{t}$ ▷ u is $\sqrt{p^2-q}$
  $v \leftarrow p + u$
  $r \leftarrow -q/v$
  return r
end procedure

This algorithm is stable for $p > 0$. However, conversely to the first algorithm, it is instable for $p < 0$ and $p^2 \gg q$. So depending on p and q we would rather choose the first algorithm or the second algorithm from a numerical analysis point of view.

Nevertheless, both algorithms are instable when $s \approx q$, meaning $p^2 \approx q$, due to the second step. However, in this case the equation has either no solution or the two solutions are very close to each other as the extreme value of the parabolic function graph is close the x-axis. Hence, little changes in p or q have big impacts to the solutions. In other words, our intuition says that the problem is already ill-conditioned for $p^2 \approx q$, which we can confirm:

$$\kappa_{\text{abs}} = \|\varphi'(p,q)\| = \left\| \begin{pmatrix} -1 + \frac{p}{\sqrt{p^2-q}} & \frac{-1}{2\sqrt{p^2-q}} \end{pmatrix} \right\| \geq \frac{1}{4|p^2-q|}.$$

[35] A real solution exists in $\mathbb{R}$ if $p^2 \geq q$.

Chapter 2

Systems of linear equations

Systems of linear equations play an essential role not only for all technical disciplines, but also in physics, chemistry, economics, and essentially in all fields where mathematics in general plays a role. In computer science alone, we have applications in computer vision, machine learning, computational geometry, and so on. In industrial automation a prime example of systems of linear equations is the forward and backward kinematic in robotics.

2.1 Introduction

In the following we consider a system

$$\begin{aligned} a_{11}x_1 + \cdots + a_{1n}x_n &= b_1 \\ &\vdots \\ a_{n1}x_1 + \cdots + a_{nn}x_n &= b_n \end{aligned}$$

of linear equations with variables $x_j \in \mathbb{R}$ and real coefficients $a_{ij} \in \mathbb{R}$ at the left-hand side and constants $b_i \in \mathbb{R}$ at the right-hand side. Using matrices we can write more concisely

$$A \cdot x = b,$$

and we assume that the real $n \times n$ matrix $A = (a_{ij})$ is regular[1], $x = (x_1, \ldots, x_n)$ and $b = (b_1, \ldots, b_n)$. From linear algebra we know that there is a unique $x \in \mathbb{R}^n$ such that $A \cdot x = b$. If we would know the inverse Matrix A^{-1} of A already then we could simply calculate $x = A^{-1} \cdot b$.

2.2 Gaussian elimination

2.2.1 Right triangular matrix and back substitution

The well known Gaussian elimination is able to solve the system $A \cdot x = b$, but can also be used to find the inverse A^{-1}. The Gaussian elimination method is also known as *row reduction*, which essentially already describes how it works:

[1]Regular means invertible. That is, a matrix A^{-1} of the same dimension exists such that $A \cdot A^{-1} = A^{-1} \cdot A = I$, where I is the identity matrix. Another common notation is $I = (\delta_{ij})$, where δ_{ij} denotes the Kronecker-delta, which is 1 when $i = j$ and 0 otherwise. A square matrix A is regular iff its determinate $\det A \neq 0$.

In a first step we subtract from the i-th equation the a_{i1}/a_{11}-multiple of the first equation for all $2 \leq i \leq n$. In matrix notation we receive the as a new i-th row in the matrix A:

$$\begin{pmatrix} a_{i1} & a_{i2} & a_{i3} & \dots & a_{in} \end{pmatrix} - \frac{a_{i1}}{a_{11}} \begin{pmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} \end{pmatrix} = \begin{pmatrix} 0 & a_{i2}^{(1)} & a_{i3}^{(1)} & \dots & a_{in}^{(1)} \end{pmatrix}.$$

By doing so, we do not alter the solution of the original system, but we introduced leading zeros (row reduction) to all equations below the first one. After the first step we receive this new system:

$$\left(\begin{array}{c|ccc} a_{11} & a_{12} & \dots & a_{1n} \\ \hline 0 & a_{22}^{(1)} & \dots & a_{2n}^{(1)} \\ 0 & a_{32}^{(1)} & \dots & a_{3n}^{(1)} \\ & & \vdots & \\ 0 & a_{n2}^{(1)} & \dots & a_{nn}^{(1)} \end{array}\right) \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{pmatrix} = \left(\begin{array}{c} b_1 \\ \hline b_2^{(1)} \\ b_3^{(1)} \\ \vdots \\ b_n^{(1)} \end{array}\right), \tag{2.1}$$

where $a_{ij}^{(1)} = a_{ij} - a_{1j} \cdot a_{i1}/a_{11}, b_i^{(1)} = b_i - b_1 \cdot a_{i1}/a_{11}$ for all $1 \leq i, j \leq n$. We now repeat this row reduction in a way that we subtract from the i-th row the $a_{i2}^{(1)}/a_{22}^{(1)}$-multiple of the second row for all $3 \leq i \leq n$. We now introduced leading zeros in the second column of row 3 to n.

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} \\ 0 & a_{22}^{(1)} & a_{23}^{(1)} & \dots & a_{2n}^{(1)} \\ 0 & 0 & a_{32}^{(2)} & \dots & a_{3n}^{(2)} \\ & & & \vdots & \\ 0 & 0 & a_{n2}^{(2)} & \dots & a_{nn}^{(2)} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2^{(1)} \\ b_3^{(2)} \\ \vdots \\ b_n^{(2)} \end{pmatrix}. \tag{2.2}$$

We keep doing row reduction and successively eliminate all elements below the diagonal of the matrix. After the $(n-1)$-th step we therefore obtain a so-called *right (upper) triangular matrix*, which is typically denoted by the letter R. Hence, with $R = (r_{ij})$ we end up with the system

$$\begin{aligned} r_{11} \cdot x_1 + r_{12} x_2 + \dots + r_{1n} x_n &= b_1' \\ r_{22} x_2 + \dots + r_{2n} x_n &= b_2' \\ &\vdots \\ r_{nn} x_n &= b_n', \end{aligned} \tag{2.3}$$

which has the identical the solution space than the original system of linear equations. From the last equation we conclude $x_n = b_n'/r_{nn}$, and we can *back-substitute* x_n into the other equations. Now we can determine x_{n-1} and so forth in order to obtain the solution vector $(x_1, \dots, x_n)$. We can summarize the Gaussian elimination as follows:

> A system of linear equations with a right triangular matrix is easily solved using back-substitution. The Gaussian elimination method simply translates the original system into a right triangular matrix form.

2.2.2 Pivoting

In the first step of the Gaussian elimination we divided by a_{11} and in the subsequent steps we divided by $a_{ss}^{(s-1)}$. Of course, this only works if these elements are not zero. If this would be the

case then at least one the elements below in the same column must be non-zero[2] and we can simply exchange the two rows in order to make $a_{ss}^{(s-1)}$ non-zero.

This procedure is called *pivoting* and the element $a_{ss}^{(s-1)}$ is called the *pivot*. Regarding the numerical condition of division operation, however, we do not only require $a_{ss}^{(s-1)} \neq 0$ but we want its absolute value to be as large as possible. The reason for this is that the absolute condition of the operation $\varphi(x) = a/x$ is

$$\kappa_{\text{abs}} = \frac{|a|}{x^2},$$

which is the smaller the larger x is. Hence, in the s-th step we first look for the largest value of $|a_{ks}^{(s-1)}|$ among $|a_{ss}^{(s-1)}|, \ldots, |a_{ns}^{(s-1)}|$ and then swap the s-th and k-th row.

From a numerical point of view we could also swap columns in order to make the pivot even larger. However, then we would need to do bookkeeping on the column transpositions because each of them also swaps elements in the solution vector, which have to be undone at the very end of the Gaussian elimination. We do not go into further details here as Gaussian elimination is anyhow not the first choice in practice.

However, we would like to mention that the Gaussian elimination can also be used with singular coefficient matrices A and it computes the solution space. The solution space is either empty or an affine-linear subspace with zero, one or infinitely many elements.

2.2.3 Time complexity

In the s-th row reduction step we modify $(n-s)^2$ elements of A and $(n-s)$ elements of b. Each modification involves a constant number of elementary[3] operations. Hence, altogether we have

$$\sum_{s=1}^{n-1} (n-s)^2 + (n-s) = \sum_{s=0}^{n-1} s^2 + s = \underbrace{\frac{(n-1)n(2n-1)}{6}}_{\text{in } A} + \underbrace{\frac{(n-1)n}{2}}_{\text{in } b} \sim \frac{n^3}{3} \tag{2.4}$$

modifications, where $\sim$ means *asymptotically equivalent*.[4] Hence, in $O(n^3)$ time we obtain the right triangular form.

The back-substitution of x_s requires us to modify $s-1$ entries in the vector b: We keep modifying b by plugging in known x_s in eq. (2.3) on the left-hand side of the equation and subtracting those terms onto b on the right-hand side. That is, for x_n we have modify $n-1$ elements of b, for x_{n-1} we modify $n-2$ elements in b, and so on. This is repeated until b is the solution vector. This leads to

$$\sum_{s=2}^{n} (s-1) = \sum_{s=1}^{n-1} s = \frac{(n-1)n}{2} \sim \frac{n^2}{2} \tag{2.5}$$

element modifications of the vector b until we have the solution vector x. Hence, the back-subsitution takes $O(n^2)$ time. Altogether we can solve $A \cdot x = b$ in $O(n^3)$ time, which is spent in asymptotically $n^3/3$ element modifications, as the row-reduction dominates.

[2]Otherwise the $(s-1)$-th and the $(s-2)$-th column would be linearly dependent and the original matrix A could not have been regular and $\det A = 0$.

[3]An elementary operation – such as add and multiply – can be done in constant time. For instance, a processor has machine instructions to add and multiply elements.

[4]The equivalence relation $f(n) \sim g(n)$ is defined as $\lim_{n\to\infty} f(n)/g(n) = 1$.

2.2.4 Multiple right-hand sides

From the analysis of the time complexity we learned that the back-substitution takes $O(n^2)$ time, while the row reduction itself takes $O(n^3)$ time. In some sense, we could perform the back-substitution a linear number of times without compromising the overall time complexity of $O(n^3)$. In particular, we could consider n right-hand sides $b_1, \dots, b_n$ and solve all the systems $A \cdot x = b_i$.

Let us therefore consider r right-hand sides $b_1, \dots, b_r$ and join them column-wise to a matrix $B = (b_1, \dots, b_r)$. We then look for a $n \times r$ matrix X that solves

$$A \cdot X = B.$$

The row reduction and the back-substitution work the same way, we just have to modify all r columns of B instead of a single column vector b. Following eq. (2.4) and eq. (2.5) we therefore have asymptotically

$$\frac{n^3}{3} + r\frac{n^2}{2} + r\frac{n^2}{2} \quad = \quad \frac{n^3}{3} + rn^2 \tag{2.6}$$

element modifications until we obtain the solution matrix X. Hence, solving $r \in O(n)$ linear systems with the same coefficient matrix A costs the same as solving a single system in terms of the O-notation, namely $O(n^3)$ time.

Inverse of a matrix. For the special case where B is the identity matrix we obtain a matrix X such that $A \cdot X = I$. This is just the definition of the inverse matrix A^{-1} of A. The Gaussian elimination can therefore be used to compute the matrix inverse and it requires asymptotically

$$\frac{4}{3}n^3$$

element modifications reps. $O(n^3)$ time by eq. (2.6). If we look more closely, we can exploit the fact that B contains mostly zeros, which allows for improvements to reduce the number of operations, but it does not improve the $O(n^3)$ bound. See [19] for details.

2.3 Linear regression

2.3.1 Overdetermined system of equations

In the introduction to this chapter we started with a system of linear equations $A \cdot x = b$, where the square matrix A was regular. In other words, we had exactly as many equations as variables and the rank of A was maximal, i.e, the rows of A were linearly independent.

In this section we study an *overdetermined* system of linear equations, which possesses more equations than variables. That is, we consider a system $A \cdot x = b$ of m equations in n variables, where $m > n$ and A is therefore an $m \times n$ matrix. Such a system cannot be solved exactly in general, which means that there is no $x \in \mathbb{R}^n$ such that $A \cdot x = b$.

However, we can ask which $x \in \mathbb{R}^n$ minimizes the error $\|A \cdot x - b\|$. Minimizing $\|A \cdot x - b\|$ is equivalent to minimizing $\|A \cdot x - b\|^2$, hence this problem can be rephrased into finding $x \in \mathbb{R}^n$ such that

$$\sum_{i=1}^{m} (a_{i1}x_1 + \dots a_{in}x_n - b_i)^2$$

is minimized. This leads to the method of *least squares*, which dates back to Carl Friedrich Gauss. First applications of least squares were in geodesy and astronomy.[5] Least squares are a standard tool in regression analysis, they are a fundamental tool in data fitting[6] and often are used as an introduction into machine learning of linear models.

2.3.2 Normal equations

We are given a real $m \times n$-matrix A and a vector $b \in \mathbb{R}^m$ and we seek for a $x \in \mathbb{R}^n$ that minimizes $\|A \cdot x - b\|$. The term $A \cdot x$ expresses the transformation of a point $x \in \mathbb{R}^n$ into a point $A \cdot x \in \mathbb{R}^m$ and the set of all such transformed points form $\operatorname{im} A$, the image of A:

$$\operatorname{im} A = \{Ax \colon x \in \mathbb{R}^n\}.$$

Note that $\operatorname{im} A$ lives in $\mathbb{R}^m$, and from linear algebra we know that it forms in fact a linear subspace[7] of $\mathbb{R}^m$. So our goal is now to find a point in $\operatorname{im} A$ that is closest to b. In case that $b \in \operatorname{im} A$ then $A \cdot x = b$ can be solved exactly. In the general case, however, we look for the orthogonal projection of b onto the linear subspace $\operatorname{im} A$ within $\mathbb{R}^m$, see fig. 2.1.

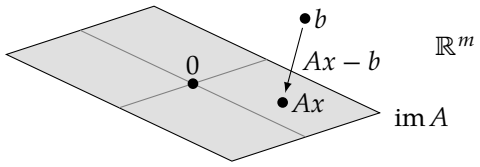


Figure 2.1: When we orthogonally project b onto the linear subspace $\operatorname{im} A$ of $\mathbb{R}^m$, we hit a certain Ax which minimizes $\|Ax' - b\|$ among all $x' \in \mathbb{R}^n$.

Assume that x fulfills the property that Ax is the orthogonal projection of b onto $\operatorname{im} A$. Then any vector in $\operatorname{im} A$ is orthogonal to $(Ax - b)$, which means that their inner product is zero. Hence, for any $x' \in \mathbb{R}^n$ it holds that

$$(Ax') \cdot (Ax - b) = 0.$$

Let us denote by $A^\dagger$ the transpose of the matrix A. We recall that $(AB)^\dagger = B^\dagger A^\dagger$ for matrices A and B, and therefore $A \cdot x' = x' \cdot A^\dagger$, where x' first denotes a column vector and then a row vector. We conclude that for any $x' \in \mathbb{R}^n$

$$x' \cdot A^\dagger (Ax - b) = 0$$

This is indeed the case if $A^\dagger(Ax - b)$ is zero, which means $A^\dagger Ax = A^\dagger b$. In other words, we look for a solution x of the so-called *normal equation system*

$$(A^\dagger A) \cdot x = (A^\dagger b). \tag{2.7}$$

Hence, any $x \in \mathbb{R}^n$ that solves eq. (2.7) minimizes $\|Ax - b\|$. In general the solution x is not unique. However, if A has maximal rank – so the columns are linearly independent – then it

[5] In Jan 01, 1801 the asteroid Ceres was discovered and tracked for 40 days, until it vanished behind the sun. It was the 24-year old Gauss who could predict its future position using least squares, which allowed to relocate Ceres again.

[6] In fact, for a linear model of data points with Gaussian distributed errors the least square method yields the maximum likelihood estimator.

[7] Dt. Untervektorraum

can be shown that $A^\dagger A$ is regular and the solution x is unique. (This follows from $\operatorname{rank} A = \operatorname{rank} A^\dagger A$), see [17] for details.)

Note that if A has maximal rank such that $A^\dagger A$ is regular then we can directly calculate

$$x = (A^\dagger A)^{-1} A^\dagger \cdot b \tag{2.8}$$

as the "best solution" of the overdetermined system $Ax = b$ in the sense that the error $\|Ax - b\|$ is minimized. Hence, $(A^\dagger A)^{-1} A^\dagger$ plays the role of an "inverse" of A although A is not necessarily regular, not even square. In fact, if A has maximal rank then $(A^\dagger A)^{-1} A^\dagger$ goes by the name *pseudoinverse* or *Moore-Penrose inverse*[8] of A. The pseudoinverse is closely linked to the *singular value decomposition*, see [19] for details.

One of numerous applications of the pseudoinverse is the computation of the inverse of a Jacobian matrix that arises in the multi-dimensional Newton-Raphson method when searching for roots of functions. This is, for instance, the basis of a numerical method for the backward kinematics in robotics. Another example is presented in the following section.

2.3.3 Fitting functions

The term "regression" in *linear regression* has a historical background: Francis Galton studied the heights of adults as a function of their parent's height and found a tendency, namely *regression to the mean*. Galton essentially got a plot of points $(x_1, y_1), \ldots, (x_m, y_m) \in \mathbb{R}^2$ as in fig. 2.2 and asked for the best fitting linear function.

By "best fitting" we mean that we assume a linear model $f(x) = k \cdot x + d$ for the data and we want to find the parameters $k, d \in \mathbb{R}$ such that the (sum of squares) error $\sum_i (f(x_i) - y_i)^2$ is minimized. Phrasing the problem like this makes it fit into the framework of overdetermined linear equation systems, namely $f(x_i) = y_i$ for all $1 \leq i \leq m$, or in matrix notion:

$$\underbrace{\begin{pmatrix} 1 & x_1 \\ \vdots & \vdots \\ 1 & x_m \end{pmatrix}}_{A} \cdot \begin{pmatrix} d \\ k \end{pmatrix} = \begin{pmatrix} y_1 \\ \vdots \\ y_m \end{pmatrix} \tag{2.9}$$

If not all x_i are equal then A has maximal rank 2 and by virtue of eq. (2.8) we can directly calculate the parameter vector

$$\begin{pmatrix} d \\ k \end{pmatrix} = (A^\dagger A)^{-1} A^\dagger \cdot \begin{pmatrix} y_1 \\ \vdots \\ y_m \end{pmatrix}$$

from eq. (2.8) for the function $f(x) = kx + d$ plotted in fig. 2.2. If $m = 2$ and $x_1 \neq x_2$ then A is regular. As a consequence $(A^\dagger A)^{-1} A^\dagger = A^{-1}$ and we obtain the one linear function that passes through the two points. If all x_i are equal then $A^\dagger A$ is singular[9]. Then eq. (2.7) has infinitely many solutions and there are infinitely many best fitting functions.[10]

We can generalize this idea of fitting functions as follows. If we recap eq. (2.9) then we can interpret this equation as follows: We have a linear combination of two base functions, the

[8] Note that we only considered overdetermined systems here. However, the Moore-Penrose inverse can also be defined for underdetermined systems, in fact it uniquely exists for any matrix A.

[9] Not regular, not invertible.

[10] If all x_i are equal then every best fitting linear function passes through a point (x_i, y), and every linear function that passes through that point is best fitting as it has the same error. This point (x_i, y) therefore minimizes $\sum_i (y_i - y)^2$, which gives $y = 1/m \sum_i y_i$. So the point (x_i, y) is the center of gravity of the data points.

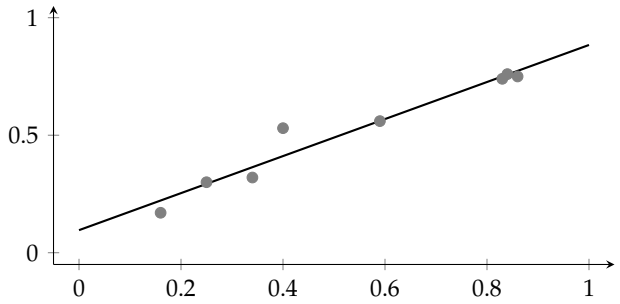


Figure 2.2: Eight data points give rise to an overdetermined linear equation system for a fitting linear function.

constant function $g_1(x) = 1$ and the function $g_2(x) = x$, and we want to find the coefficients k and d of this linear combination that minimizes the least square error.

The general idea is to consider n base functions $g_1, \dots, g_n$ and ask for the linear combination $f = \sum_{i=1}^{n} \alpha_i g_i$ for the data points $(x_1, y_1), \dots, (x_m, y_m)$ such that the error

$$\sum_{i=1}^{m} (f(x_i) - y_i)^2 = \sum_{i=1}^{m} \left(\sum_{j=1}^{n} \alpha_j g_j(x_i) - y_i \right)^2$$

is minimized. This again is the solution of the overdetermined linear equation system

$$\underbrace{\begin{pmatrix} g_1(x_1) & \cdots & g_n(x_1) \\ & \vdots & \\ g_1(x_m) & \cdots & g_n(x_m) \end{pmatrix}}_{A} \cdot \begin{pmatrix} \alpha_1 \\ \vdots \\ \alpha_n \end{pmatrix} = \begin{pmatrix} y_1 \\ \vdots \\ y_m \end{pmatrix}. \tag{2.10}$$

The set of linear combinations of the g_i span a vector space of functions and it makes sense that the g_i form a *basis*[11] of this vector space, i.e., they shall be linearly independent. Otherwise the columns of A in eq. (2.10) will not be linearly independent and A cannot have maximal rank.

Note that the g_i do not need to be linear functions. We could choose the base functions $1, \sin(x), \cos(x), \dots, \sin(nx), \cos(nx)$ and span the vector space of all trigonometric polynomials up to degree n. This way we create a conceptional bridge between the Fourier transform and fitting of functions.

A very practical choice of base functions is $1, x, x^2, \dots, x^n$ to form polynomials of the form $\sum_{i=0}^{n} \alpha_i x^i$. These base functions span the vector space of polynomial functions up to degree n. In this case eq. (2.10) yields

$$\begin{pmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^n \\ & & & \vdots & \\ 1 & x_m & x_m^2 & \cdots & x_m^n \end{pmatrix} \cdot \begin{pmatrix} \alpha_0 \\ \vdots \\ \alpha_n \end{pmatrix} = \begin{pmatrix} y_1 \\ \vdots \\ y_m \end{pmatrix}. \tag{2.11}$$

and the solution leads to the best approximating polynomial function up to degree n. The initial example of fitting a linear function is a polynomial fit with $n = 1$, see eq. (2.9) versus eq. (2.11).

[11] The vectors $v_1, \dots, v_d$ of a vector space V (of finite dimension) form a *basis* if they span V – the set of linear combinations fills V out – and they are linearly independent. So a basis is a smallest set of linearly independent vectors that span V. Every basis of V has the same number of elements, which is called the *dimension* of v. Also functions can form vector spaces, e.g., the set of functions $\mathbb{R} \to \mathbb{R}$ forms a vector space V, where $(f + g)(x) = f(x) + g(x)$ and $(\lambda f)(x) = \lambda \cdot f(x)$ for all $f, g \in V$ and $\lambda \in \mathbb{R}$.

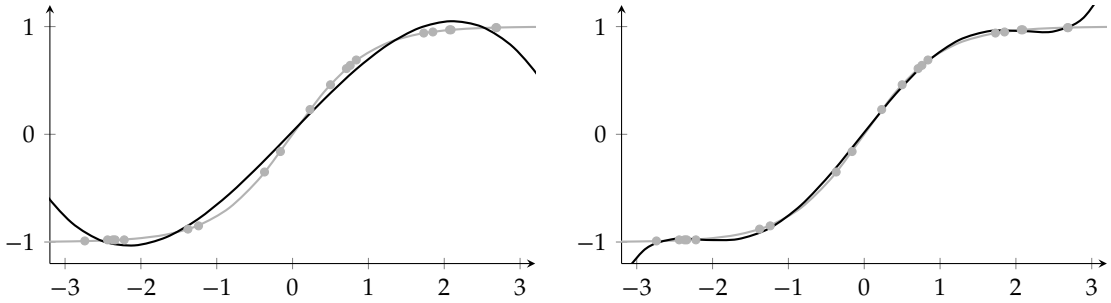


Figure 2.3: Polynomial fit of data points that sample tanh at 20 uniformly random points on $[-3, 3]$. Left: The polynomial of degree 3. Right: The polynomial of degree 5.

In fig. 2.3 two examples of approximating polynomials are given. They have both been computed by eq. (2.8) applied to eq. (2.11).

Software packages like numpy for Python or MATLAB provide library functions that implement least square polynomial fit. The following lines are from a Python interpreter shell:

```
>>> import numpy as np
>>> x = np.array([0.0, 1.0, 2.0, 3.0,  4.0,  5.0])
>>> y = np.array([0.0, 0.8, 0.9, 0.1, -0.8, -1.0])
>>> z = np.polyfit(x, y, 3)
>>> z
array([ 0.08703704, -0.81349206,  1.69312169, -0.03968254])
>>> print(np.poly1d(z))
         3          2
0.08704 x - 0.8135 x + 1.693 x - 0.03968
```

2.3.4 QR decomposition

The QR decomposition is a method that can solve regular linear equation systems and overdetermined linear equation systems. It can be shown that the condition of the QR decomposition is significantly better than the one of the normal equations. It also introduces less rounding errors than Gaussian elimination.

We again consider a linear equation system $Ax = b$ with a $m \times n$ matrix A, where $m \geq n$. It can be shown that there is always an $m \times m$ orthogonal matrix Q such that

$$QA = \begin{pmatrix} R \\ 0 \end{pmatrix}, \tag{2.12}$$

where R is a $n \times n$ right triangular matrix and 0 is a $(m - n) \times n$ zero matrix.

Recall that a square matrix is called orthogonal if its columns are orthogonal unit vectors, i.e., they form a orthonormal basis of the $\mathbb{R}^m$. From that it follows that $Q^\dagger Q = QQ^\dagger = I$ which means that Q can be trivially inverted as $Q^\dagger = Q^{-1}$. An orthogonal matrix Q also has the nice property that it does not change the length of vectors, i.e., $\|Qx\| = \|x\|$. In some sense, Q is only performing combinations of rotations and reflections.

Once we know Q we can also compute

$$Qb = \begin{pmatrix} c \\ d \end{pmatrix}$$

with a vector $c \in \mathbb{R}^n$ and $d \in \mathbb{R}^{m-n}$. Our goal is again that to find the $x \in \mathbb{R}^n$ that minimizes $\|Ax - b\|$. We observe that for any $x \in \mathbb{R}^n$

$$\|Ax - b\|^2 = \|Q(Ax - b)\|^2 = \|QAx - Qb\|^2 = \left\| \begin{pmatrix} Rx - c \\ -d \end{pmatrix} \right\|^2 = \|Rx - c\|^2 + \|d\|^2 \geq \|d\|^2,$$

where the last step is actually an equality when $Rx = c$. Hence, we look for the solution x of $Rx = c$ as it minimizes $\|Ax - b\|$. But remember that R is an right triangular matrix and hence $Rx = c$ is easy to solve via back substitution. This method is called *QR method* because A can be expressed as

$$A = Q^\dagger \begin{pmatrix} R \\ 0 \end{pmatrix}. \tag{2.13}$$

The essential step in the QR method is the computation of the matrix Q. A numerically stable algorithm to compute Q is based on Householder reflections. Reflection matrices $Q_1, \ldots, Q_n$ are applied to A in a way such that the Q_k basically generates the k-th column of the right-hand side in eq. (2.12). The product $Q_n \cdots Q_1$ is then Q in eq. (2.12). The other algorithm uses Givens rotations rather than Housholder reflections.

Software packages like numpy for Python or MATLAB provide library functions that implement QR decomposition. The following lines are from a Python interpreter shell. (Note that `np.linalg.qr()` returns the two factors in the right-hand side of eq. (2.13).)

```
>>> import numpy as np
>>> A = np.random.randn(9, 6)            # Random 9x6 matrix
>>> Q, R = np.linalg.qr(A)
>>> np.linalg.norm(A - Q @ R) <= 1e-14   # A == QR
True
```

Besides the QR decomposition there are a couple of other common matrix decompositions. For instance, any square matrix A admits an *LU decomposition*

$$A = P \cdot L \cdot U,$$

where P is a permutation matrix, L is a left (lower) triangular matrix and U is an upper (right) triangular matrix. This decomposition is typically used to solve regular linear equation systems, like the function `np.linalg.solve()` does in numpy. In MATLAB the function `linsolv()` uses LU decomposition if the coefficient matrix A is square and QR decomposition in all other cases.

2.3.5 Equilibration and regularization

In the row reduction steps of the Gaussian elimination method we multiplied single equations without changing the set of solutions. The solution of an overdetermined system of equations, however, is only an approximate solution that minimizes the least square error. Hence, if we scale a single equation then we change the expression for the error and we obtain a different solution.

More precisely, let $A \cdot x = b$ denote an overdetermined system and let x be a solution. We learned from section 2.3.1 that x therefore minimizes the error expression

$$\|Ax - b\|^2 = \sum_{i=1}^{m} \left(\sum_{j=1}^{n} a_{ij} x_j - b_i \right)^2,$$

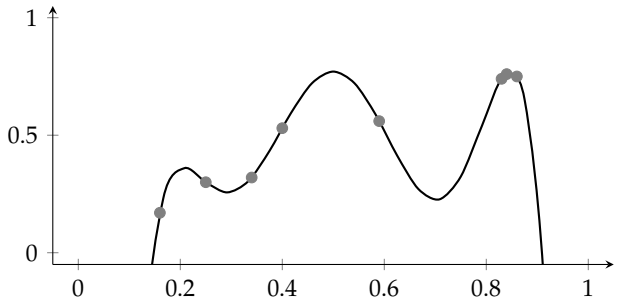


Figure 2.4: The polynomial of degree 6 fitting 8 data points shows strong oscillations.

where $A = (a_{ij})$ and $b = (b_i)$. Assume now that we multiply the k-th equation by a factor $\lambda \neq 0$. We therefore obtain a new matrix $A' = (a'_{ij})$ and a new right-hand side b'_i with

$$a'_{ij} = \begin{cases} a_{ij} & \text{for } i \neq k \\ \lambda a_{ij} & \text{for } i = k \end{cases} \qquad \text{and} \qquad b'_i = \begin{cases} b_i & \text{for } i \neq k \\ \lambda b_i & \text{for } i = k \end{cases}.$$

The solution x' to this new system now minimizes the error expression

$$\sum_{i \neq k} \left(\sum_j a_{ij} x_j - b_i \right)^2 + \lambda^2 \left(\sum_j a_{kj} x_j - b_k \right)^2.$$

In other words, the solution x' now puts a *weight* λ on the k-th equation for the error minimization. By adjusting λ we can control how important or significant an equation is to us.

Equilibration. Assume that we actually have $m = m_1 + m_2$ equations in our system, where the first m_1 equations stem from a certain objective to be optimized and a second set of m_2 equations that stem from a different objective. Assume that both objectives are equally important to us, but $m_1 \neq m_2$. We scale the first m_1 equations by $1/m_1$ and the others by $1/m_2$ such that the sum of weights for the equations of each objective is 1, which results in the error expression

$$\frac{1}{m_1} \sum_{i=1}^{m_1} \left(\sum_j a_{ij} x_j - b_i \right)^2 + \frac{1}{m_2} \sum_{i=m_1+1}^{m_1+m_2} \left(\sum_j a_{ij} x_j - b_i \right)^2.$$

Equilibration is the method of scaling equations in a way to equilibrate their weight. Depending on the problem we may, for instance, scale the equations such that the coefficient vectors $(a_{k1}, \dots, a_{kn})$ are all of equal length for $1 \leq k \leq m$.

Regularization. For certain systems the solution may not "behave very well". Consider for instance the polynomoial of degree 6 that is shown in fig. 2.4 and fits the point set from fig. 2.2. What we see is the typical behavior of polynomials of higher degree: They have a strong tendency to oscillate significantly.

In order to counteract against this effect we can "penalize" the oscillation behavior by extending the error term accordingly. By "oscillation" we actually mean high curvature and in fact the *strain energy* of a function $f\colon \mathbb{R} \to \mathbb{R}$ in the interval $[a, b]$ is given by

$$\int_a^b f''(x)^2 \, \mathrm{d}x, \tag{2.14}$$

which we could interpret as the infinitesimal sum (integral) of the square error of the curvature (second derivative). We have a polynomial $f(x) = \sum_{i=1}^{n} \alpha_i x^i$ whose second derivative is

$$f''(x) = \sum_{i=2}^{n} \alpha_i x^{i-2} \cdot i(i-1) \tag{2.15}$$

We would like to minimize the error term given in eq. (2.14) for the α_k, so we set the partial derivatives zero. We see from eq. (2.15) that α_0, α_1 actually do not matter so we only consider the $n-1$ equations for $\alpha_2, \dots, \alpha_n$:

$$\begin{aligned}
0 &= \frac{\partial}{\partial \alpha_k} \int_a^b \left(\sum_{i=2}^{n} \alpha_i x^{i-2} \cdot i(i-1) \right)^2 \, \mathrm{d}x \\
&= \int_a^b 2 \cdot \left(\sum_{i=2}^{n} \alpha_i x^{i-2} \cdot i(i-1) \right) x^{k-2} \cdot k(k-1) \, \mathrm{d}x \\
&= 2 \cdot \sum_{i=2}^{n} \alpha_i \cdot ik(i-1)(k-1) \cdot \int_a^b x^{i+k-4} \, \mathrm{d}x \\
&= 2 \cdot \sum_{i=2}^{n} \alpha_i \cdot ik(i-1)(k-1) \frac{b^{i+k-3} - a^{i+k-3}}{i+k-3}
\end{aligned}$$

We end up with a system of $n-1$ linear equations

$$(c_{ki}) \cdot (\alpha_i) = 0 \qquad \text{where} \qquad c_{ki} = \begin{cases} 0 & \text{for } i \leq 2 \\ ik(i-1)(k-1) \frac{b^{i+k-3} - a^{i+k-3}}{i+k-3} & \text{for } i > 2 \end{cases} \tag{2.16}$$

and k ranging from 2 to n. Here (c_{ki}) and (α_i) denote matrices and vectors, respectively.

However, we want to control the amount of this penalty and therefore we weight them by a weight $\lambda > 0$. Together with the original system of eq. (2.11) for the approximation of the data points, we end up with the following system:

$$\left(\begin{array}{ccccc} 1 & x_1 & x_1^2 & \dots & x_1^n \\ & & & \vdots & \\ 1 & x_m & x_m^2 & \dots & x_m^n \\ \hline 0 & 0 & \lambda c_{22} & \dots & \lambda c_{2n} \\ & & & \vdots & \\ 0 & 0 & \lambda c_{n2} & \dots & \lambda c_{nn} \end{array}\right) \cdot \begin{pmatrix} \alpha_0 \\ \vdots \\ \alpha_n \end{pmatrix} = \left(\begin{array}{c} y_1 \\ \vdots \\ y_m \\ \hline 0 \\ \vdots \\ 0 \end{array}\right). \tag{2.17}$$

The first m equations stem from the approximation task of the data points and the next $n-1$ equations stem from the curvature penalty. The error term that is minimized by the entire system is

$$\sum_{k=1}^{m} (f(x) - y_i)^2 + \lambda^2 \int_a^b f''(x)^2 \, \mathrm{d}x.$$

If we choose $\lambda > 0$ then the solution becomes more *regular* in the sense of "well-behaved". Therefore we call $\lambda \int_a^b f''(x)^2 \, \mathrm{d}x$ the *regularization term* and the method is called *regularization*. If we choose $\lambda = 0$ then we have the original non-regularized version, see fig. 2.5 for an example. Of course, increasing λ reduces the amount of oscillation at the expense of an increased approximation error. Furthermore, we would like to combine this regularization with equilibration, such that the effect of parameter λ becomes somewhat independent of m.

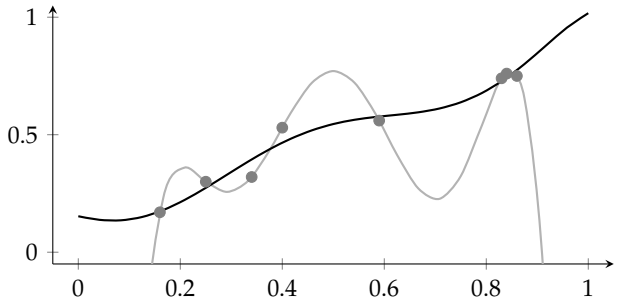


Figure 2.5: Regularized solution for a polynomial of degree 6 as in fig. 2.4. The black graph shows the result for $\lambda = 0.001$ and the gray graph for $\lambda = 0$, which equals therefore the non-regularized version. The strain energy is computed over the interval $[0, 1]$.

Chapter 3

Polynomial interpolation

3.1 Motivation

We often face the situation, where a function f is not given by a closed expression, but at finitely many positions $x_1, \dots, x_n$ and we want to evaluate f at arbitrary positions x. In engineering the pairs $(x_i, f(x_i))$ often stem from measurements. A typical approach to this task is to approximate f by a polynomial p with $p(x_i) = f(x_i)$ for all $1 \leq i \leq n$ and evaluate $p(x)$ as an approximation for $f(x)$. The x_i are called (*interpolation*) *nodes*[1]. Typically x is in the interval between the smallest and the largest node. If x is outside this interval then we also speak of an *extrapolation*.

But interpolation is not only useful when the input stems from measurements: Consider a parallel curve (of small distance) to the function graph of x^2. This curve is again the function graph of a function f, for which it might be difficult to find a simple expression, but we can approximate it using a finite number of samples. Strictly speaking, if we can choose the interpolation points then we actually speak of *function approximation* rather than interpolation.

Let us further assume that we now would like to integrate of differentiate a function f. Even if f is given by as a closed expression, integration or differentiation might be difficult. After all, there is for instance no elementary function for the integral of e^{-x^2}. The integration and differentiation of polynomials is easy, so we could again approximate f by a polynomial and integrate or differentiate the polynomial instead. Interpolation is therefore also the basis for further numerical methods and the question arises what error is introduced by the polynomial approximation. In general, numerical computation is much easier than symbolic computation.

Instead of polynomials we could also use other simple and versatile functions for interpolation, like linear combinations of trigonometric functions. Here we restrict ourselves to polynomials. After all, the Stone-Weierstrass approximation theorem says that every continuous function defined on a closed interval $[a, b]$ actually can be (uniformly) approximated by polynomials, which is all but clear.[2]

3.2 Power series

Polynomials play an important role for approximation because they are able to represent an import class of functions. A function $f\colon \mathbb{R} \to \mathbb{R}$ is called *analytic* on the open set $D \subseteq \mathbb{R}$ if it can

[1] Dt. Stützstelle

[2] More precisely, let $f\colon [a, b] \to \mathbb{R}$ be continuous and let $\varepsilon > 0$ be arbitrarily small. Then there is a polynomial function p such that $\|f - p\| < \varepsilon$, where $\|.\|$ denotes the supremum norm. Put in words of topology: The set of polynomial functions is dense in the set of continuous functions over $[a, b]$.

be represented by a *power series*[3]

$$f(x) = \sum_{k=0}^{\infty} a_k (x - x_0)^k, \tag{3.1}$$

such that x_0 can be chosen arbitrarily from D. All elementary functions are analytic on $\mathbb{R}$: Polynomials, exponential function, logarithms, and the trigonometric functions. Sums, products and compositions of analytic functions are again analytic.

The *Taylor series* tells us how to obtain the coefficients a_n in eq. (3.1):

$$f(x) = \sum_{k=0}^{\infty} \frac{f^{(k)}(x_0)}{k!} (x - x_0)^k.$$

Hence, analytic functions have this incredible property that if we know all derivatives at a single location x_0 then we globally know the function at any x in the domain.

Not all processors provide machine instructions for the exponential, logarithmic or trigonometric functions. But even if they do, they might be inaccurate or we would like to trade accuracy versus speed. Or maybe we would like to implement these functions in an FPGA using addition and multiplication only. In such cases, Taylor series are a common method to approximate these functions by just using the first n summands. For instance

$$\exp(x) \approx \sum_{k=0}^{n} \frac{x^k}{k!} \quad \text{and} \quad \sin(x) \approx \sum_{k=0}^{n} (-1)^k \frac{x^{2k+1}}{(2k+1)!}.$$

Even $\sin(x) \approx x$ is accurate up to a relative error of 1 % in the interval $[-0.25, 0.25]$.

3.3 Interpolation

3.3.1 Existence

A polynomial of degree $n - 1$ can exactly interpolate n points. More precisely, let $x_1, \dots, x_n$ be pairwise distinct real numbers and let $y_1, \dots, y_n$ be real numbers. Then there is exactly one polynomial p of degree at most $n - 1$ such that

$$p(x_i) = y_i \tag{3.2}$$

for all $1 \leq i \leq n$. In particular, a constant function (polynomial of degree 0) can interpolate one point, a linear function (polynomial of degree 1) can interpolate two points, and so on. We can compute p by considering the linear equation system behind eq. (3.2), namely

$$\begin{pmatrix} 1 & x_1 & \dots & x_1^{n-1} \\ & & \vdots & \\ 1 & x_n & \dots & x_n^{n-1} \end{pmatrix} \cdot \begin{pmatrix} a_0 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix}, \tag{3.3}$$

where $p(x) = \sum_{k=0}^{n-1} a_k x^k$. The coefficient matrix is regular if its determinate is nonzero. This determinate is knows as the *Vandermonde determinant*. It has the value

$$\prod_{1 \leq i < k \leq n} (x_k - x_i)$$

and is non-zero for pairwise distinct x_i. We call the x_i the *nodes* and the unique p the *interpolation polynomial*.

[3] Dt. Potenzreihe

3.3.2 Interpolation error

Let us consider a function $f\colon I \to \mathbb{R}$ on an interval $I = [a, b]$ and pairwise distinct interpolation nodes $x_1, \ldots, x_n$. We know that there is a unique interpolation polynomial p of degree at most $n-1$ such that $p(x_i) = f(x_i)$ for all $1 \leq i \leq n$. But what can we say about the error, i.e., the difference $f(x) - p(x)$ for $x \in I$?

First, we define by $\|f\|$ the *maximum norm* of f as

$$\|f\| = \max_{x \in I} |f(x)|.$$

Next we define the *node polynomial* w by

$$w(x) = \prod_{i=1}^{n} (x - x_i).$$

It can be shown that for any x there is some $\xi \in I$ such that

$$f(x) - p(x) = \frac{f^{(n)}(\xi)}{n!} \cdot w(x) \tag{3.4}$$

for all $x \in I$. Even if we do not know how to compute ξ, we can conclude that

$$\|f - p\| \leq \frac{1}{n!} \, \|f^{(n)}\| \, \|w\|. \tag{3.5}$$

The left-hand side of eq. (3.5) is the maximum interpolation error on I, that is $\max_{x \in I} |f(x) - p(x)|$. This term is bound by the right-hand side, which is small if $\|w\|$ is small. If we can choose the interpolation nodes x_i then we can choose them in a clever way such that $\|w\|$ becomes small. If we choose the interpolation nodes uniformly on our interval I then $w(x)$ tends to become large at the boundary of I, see fig. 3.1. More precisely, the extreme values of w get larger towards the boundary of I. This is known as *Runge's phenomenon*.

If we, however, increase the density of the nodes towards the boundary of I then the extreme values of w get balanced. In fact, it can be shown that setting

$$x_i = \cos \frac{(2i-1)\pi}{2n} \tag{3.6}$$

makes the extreme values of w all equal, namely $1/2^{n-1}$, which in this sense is optimal for a polynomial interpolation on $I = [-1, 1]$, see fig. 3.1. The nodes given by eq. (3.6) are called *Chebyshev nodes* and they are the roots of the so-called *Chebyschev polynomial* of n-th degree. For Chebyshev nodes we therefore get an error bound of

$$\|f - p\| \leq \frac{1}{n! \, 2^{n-1}} \, \|f^{(n)}\|. \tag{3.7}$$

In general, increasing the number n of nodes has only limited effect in improving the interpolation error. The reason is that polynomials of higher degree have a strong tendency for oscillation, as we already observed in fig. 2.4 in chapter 2. In fig. 3.2 an interpolation polynomial is shown that also illustrates this behavior.

In fact, it can happen that the interpolation error does not converge to zero when we increase n towards infinity. The reason is that in eq. (3.5) the term $\|f^{(n)}\|$ can grow towards ∞ faster than $\|w\|$ goes to zero.

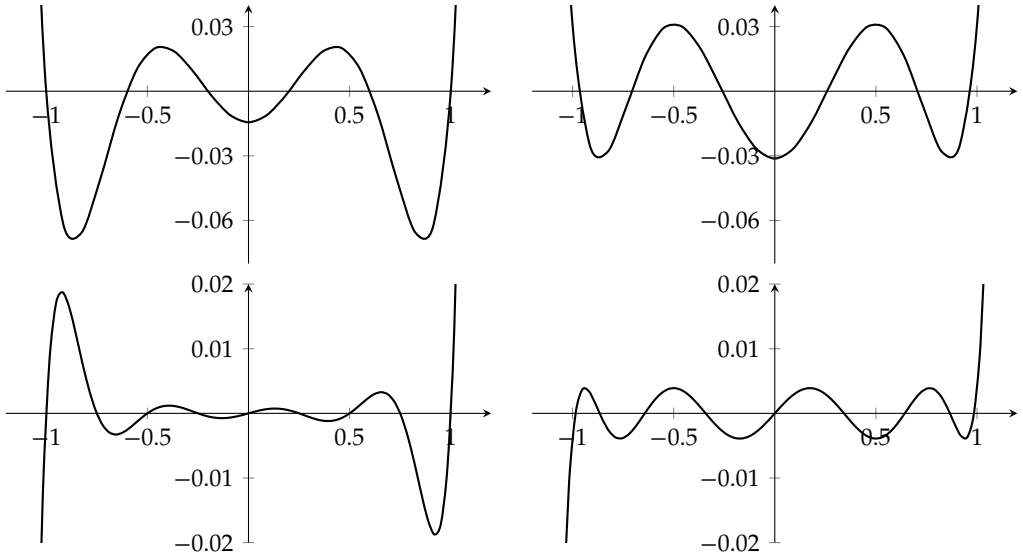


Figure 3.1: The node polynomial $w(x)$. Top: Polynomials of degree 6. Bottom: Polynomials of degree 9. Left: Uniform nodes on $[-1, 1]$. Right: Chebyshev nodes.

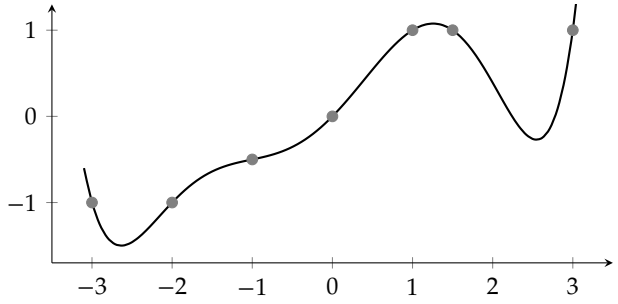


Figure 3.2: An interpolation polynomial of degree 6 that shows typical oscillation behavior.

3.3.3 Computing interpolation polynomials

It is possible to compute the interpolation polynomial by solving the linear system in eq. (3.3), but we can also compute them directly without solving a linear system of equations.

Lagrange's formula

We want to find a polynomial p of degree $n-1$ with $p(x_k) = y_k$ for all $1 \leq k \leq n$. The idea is now that we construct for each k a polynomial L_k of degree $n-1$ that is 1 at x_k, but 0 at all other nodes, i.e., $L_k(x_i) = \delta_{ik}$. Hence,

$$p(x) = \sum_{k=1}^{n} y_k L_k(x) \tag{3.8}$$

is the interpolation polynomial that we look for, because

$$p(x_i) = \sum_{k=1}^{n} y_k \delta_{ik} = y_i \delta_{ii} = y_i.$$

The polynomials L_k are called *Lagrange polynomials* of degree n and the property $L_k(x_i) = \delta_{ik}$ is easily checked for their definition:

$$L_k(x) = \prod_{\substack{i=1 \\ i \neq k}}^{n} \frac{x - x_i}{x_k - x_i}. \tag{3.9}$$

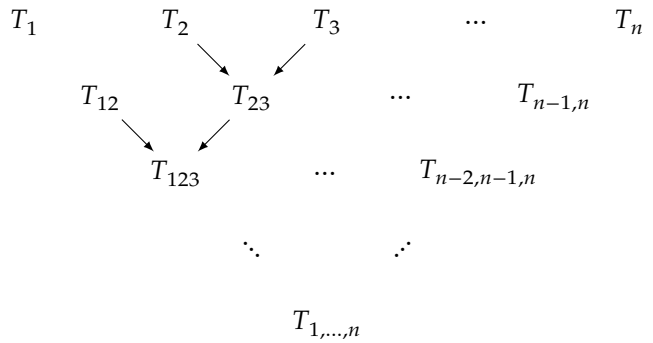
Plugging everything together gives *Lagrange's formula* for the interpolation polynomial:

$$p(x) = \sum_{k=1}^{n} \prod_{\substack{i=1 \\ i \neq k}}^{n} \frac{x - x_i}{x_k - x_i} y_k$$

Neville tableau

The Lagrange formula is easy to understand but the following *Neville algorithm* is better suited for implementation. In particular, it allows to give an error estimate of the result, see [19] for details. It follows a classic approach for algorithm design: We solve the same problem for smaller instances first and then plug the sub-results together to form the solution of larger instances.

More precisely, we start with interpolation polynomials $T_i(x) = y_i$ of degree 0 that only interpolate a single node y_i. We then combine them in a way to form polynomials $T_{i,i+1}$ of degree 1 that interpolate two nodes x_i, x_{i+1}, and so on, until we obtain $T_{1,\dots,n}$ that interpolates all nodes $x_1, \dots, x_n$. We end up with a tableau of this form:



Each element is recursively defined by its two ancestors above, except for the first row:

$$T_i(x) = y_i \tag{3.10}$$

$$T_{i,\dots,i+m}(x) = \frac{(x - x_{i+m}) T_{i,\dots,i+m-1}(x) + (x_i - x) T_{i+1,\dots,i+m}(x)}{x_i - x_{i+m}}. \tag{3.11}$$

Note that in the above recursion each $T_{i,\dots,i+m}$ results from a *linear interpolation* of the two ancestors $T_{i,\dots,i+m-1}$ and $T_{i+1,\dots,i+m}$. In particular, the elements $T_{i,i+1}$ in the second row form a linear interpolation between y_i and y_{i+1}:

$$T_{i,i+1} = \frac{(x - x_{i+1}) y_i + (x_i - x) y_{i+1}}{x_i - x_{i+1}} \tag{3.12}$$

The first row in the tableau, $T_1, \dots, T_n$, does not need to be in any particular order. We could also easily add a new interpolation node and update the tableau without recomputing it entirely. We just add a new (diagonal) column in the tableau. Sometimes it therefore makes sense to sort the $T_1, \dots, T_n$ in increasing distance to the position x and keep adding interpolation points until the changes drop below a certain threshold, instead of computing the entire final tableau.

3.4 Splines

3.4.1 Motivation

We learned in section 3.3 that polynomials of high degree tend to oscillate, as illustrated in fig. 3.2. For certain applications we could, of course, use a piecewise linear function that interpolates between the points. So let $x_1 < \cdots < x_n$ denote n nodes with function values $y_1, \dots, y_n$. As illustrated in fig. 3.3a, for each $1 \le i \le n-1$ we put a linear functions p_i on the interval $[x_i, x_{i+1}]$ that linearly interpolates between the nodes x_i and x_{i+1}:

$$p_i(x) = \lambda\, y_i + \mu\, y_{i+1} \quad \text{with} \quad \lambda = \frac{x - x_{i+1}}{x_i - x_{i+1}} \text{ and } \mu = 1 - \lambda = \frac{x_i - x}{x_i - x_{i+1}}.$$

These are exactly the polynomials $T_{i,i+1}$ of the first row in the Neville tableau, see also eq. (3.12).

Many applications, however, require an interpolation function that is at least twice differentiable, in particular for applications in physics and engineering. For instance, a co-called cam profile tells how a secondary servo drive (slave axis) should move in dependence of a first servo drive (master axis). A cam profile therefore maps one position to another and they are often given in tabulated form $(x_1, y_1), \dots, (x_n, y_n)$ such that we have to compute an interpolation. The second derivative f'' of the interpolating function f relates[4] to the acceleration, which must be finite.

The idea is to generalize piecewise linear functions to piecewise polynomial functions that fulfill certain additional properties, like being twice differentiable. Such functions are called *splines*. In this sense, a piecewise linear function is a spline of degree 1. A step function (piecewise constant) would be a spline of degree 0. By far the most common variant, however, is the cubic spline, which is of degree 3.

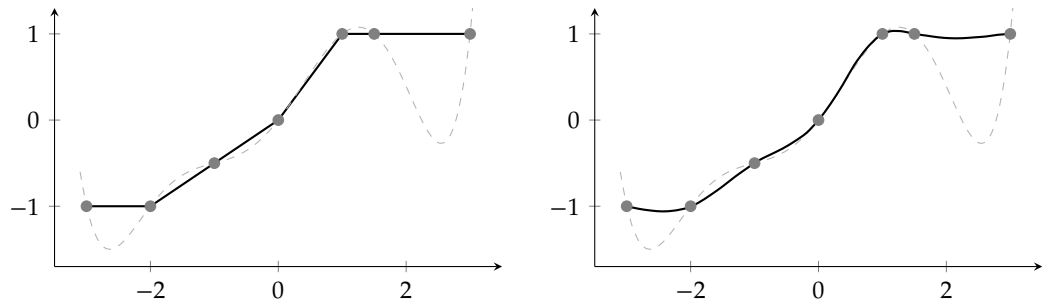
3.4.2 Cubic spline

By a cubic spline f we mean a twice continuously differentiable spline of degree three that interpolates a tabulated function $(x_1, y_1), \dots, (x_n, y_n)$. That is, the second derivative of a cubic spline exists and it is continuous.[5] Assuming $x_1 < \cdots < x_n$, we therefore have these conditions on the polynomial pieces p_i over the intervals $[x_i, x_{i+1}]$:

$$\begin{aligned}
p_i(x_i) &= y_i && \text{for all } 1 \le i \le n-1 \\
p_{n-1}(x_n) &= y_n && \\
p_i(x_{i+1}) &= p_{i+1}(x_{i+1}) && \text{for all } 1 \le i \le n-2 \\
p_i'(x_{i+1}) &= p_{i+1}'(x_{i+1}) && \text{for all } 1 \le i \le n-2 \\
p_i''(x_{i+1}) &= p_{i+1}''(x_{i+1}) && \text{for all } 1 \le i \le n-2
\end{aligned}$$

[4] If the master axis moves at unit speed then the second derivative is exactly the acceleration of the slave axis.

[5] The set of k-times continuously differentiable functions is often denoted by C^k. We then also say that f is C^k-continuous. A C^0-continuous function means that f is simply continuous. In this sense a cubic spline is a C^2-continuous interpolating spline of degree three.



(a) Piecewise linear function (b) Natural cubic spline

Figure 3.3: Spline interpolation of the points in fig. 3.2. The dashed line is the interpolation polynomial of degree six. Left: A piecewise linear function (spline of degree 1). Right: The natural cubic spline.

The first two equation establishes the interpolation property and the next three equations establish C^0-, C^1-, and C^2-continuity. Altogether we have $4n - 6$ equations (conditions) but $4n - 4$ coefficients (degrees of freedom) of the $n - 1$ polynomials. This leaves us with two more conditions that we can impose:

- One common choice is to add $p_1''(x_1) = p_{n-1}''(x_n) = 0$. A cubic spline of this form is called *natural spline*. This might be an attractive choice for the example of cam profiles, because this means that we start and end with zero acceleration.

- Another common choice is to add $p_1'(x_1) = p_{n-1}'(x_n)$ and $p_1''(x_1) = p_{n-1}''(x_n)$. We can therefore plug copies of the splines together end-to-end and the result is still C^2-continuous. If in addition $y_1 = y_n$ then the result is a periodic C^2-continuous function.

  This choice is also attractive for cam profiles as we can periodically repeat the cam profile without jumps in acceleration (or velocity).

Figure 3.3b illustrates a natural cubic spline. Natural cubic splines have the nice property that they minimize strain energy: Consider a thin wooden strip that goes through the interpolation points. The wooden strip will take a form that minimizes the strain energy. The form that we get is exactly the one of the natural spline. This is essentially the reason why natural splines avoid the oscillating behavior of higher-degree interpolation polynomials.

In order to compute a natural cubic spline we could simply solve the linear equation system formed by the $4(n-1)$ conditions. If we take a closer look, however, we see that this system can be significantly simplified and we end up with $n-1$ equations. Moreover, the simplified system actually has a special structure – it is in tridiagonal form – and can be solved in $O(n)$ time by dedicated algorithms. Details can be found in appendix A.

Software packages like scipy for Python or MATLAB provide library functions that implement cubic spline computation. The following lines are from a Python interpreter shell and produced the data for fig. 3.3b:

```
>>> import scipy.interpolate as interp
>>> xs = [-3, -2, -1, 0, 1, 1.5, 3]
>>> ys = [-1, -1, -0.5, 0, 1, 1, 1]
>>> f = interp.CubicSpline(x, y, bc_type='natural')
>>> for x in np.linspace(-3, 3, 33):
```

```
...     print("({}, {})".format(x.round(2), f(x).round(2)))
...
(-3.0, -1.0)
(-2.81, -1.03)
[...]
(3.0, 1.0)
```

3.5 Numerical derivatives

Let f be a differentiable function. We would like to numerically compute $f'(x)$ for some position x. Polynomials are easy to derive, so we could consider an interpolation polynomial p around x and take $p'(x)$ instead. Maybe f is actually given in tabulated form so we cannot symbolically derive it anyway, but we can compute the interpolation polynomial.

This raises the question whether it is justified to take $p'(x)$ as an approximation for $f'(x)$. Let us denote by $x_1 < \cdots < x_n$ the interpolation nodes. Recall eq. (3.4), which said that for any x there is a ξ such that

$$f(x) = p(x) + \frac{f^{(n)}(\xi(x))}{n!} \cdot w(x).$$

We explicitly write $\xi(x)$ to emphasize that ξ depends on x. Assuming that ξ is differentiable, it is easy to show[6] that at all interpolation nodes x_k the following holds:[7]

$$f'(x_k) = p'(x_k) + \frac{f^{(n)}(\xi(x_k))}{n!} \cdot w'(x_k).$$

Note that $w(x_k) = 0$ but $w'(x_k)$ is not zero.[8] In other words, p' only *approximates* f' at interpolation nodes x_k, while p meets f exactly. Also note that if $n = 1$ then $p'(x) = 0$ so only $n \geq 2$ is meaningful.

Two-point formula. In the simplest case we use $n = 2$ interpolation nodes and therefore receive an interpolation polynomial of degree 1. Following eq. (3.12) and setting $h = x_2 - x_1$ we have

$$p(x) = \frac{(x_2 - x)y_1 + (x - x_1)y_2}{h}$$

This immediately yields the so-called *two-point formula*

$$f'(x) \approx p'(x) = \frac{y_2 - y_1}{h} \tag{3.13}$$

Note that $p'(x)$ does not depend on x anymore; it is constant. For any position x we approximate $f'(x)$ by the slope of the secant formed by the interpolation nodes. In other words, we approximate the differential quotient by the difference quotient, see fig. 3.4a.

[6] $f'(x) = p'(x) + \frac{1}{n!}(f^{(n+1)}(\xi(x))\xi'(x)w(x) + f^{(n)}(\xi(x))w'(x))$ and $w(x_k) = 0$.

[7] The equation can also be proven without the assumption that ξ would be differentiable, but this is much harder.

[8] If $w'(x_k) = 0$ would hold then w would have two roots at x_k, and hence two interpolation nodes would be equal.

Three-point formula. Let us increase the number of nodes to $n = 3$. For a fixed $h > 0$ we choose three equidistant interpolation nodes $x_1 = x_2 - h, x_2, x_3 = x_2 + h$. According to Lagrange's formula eq. (3.8) we have

$$p(x) = y_1 L_1(x) + y_2 L_2(x) + y_3 L_3(x) \quad \text{with} \quad L_k(x) = \prod_{\substack{i=1 \\ i \neq k}}^{3} \frac{x - x_i}{x_k - x_i}$$

and therefore

$$p'(x) = y_1 L_1'(x) + y_2 L_2'(x) + y_3 L_3'(x). \tag{3.14}$$

Let us compute:

$$\begin{aligned} L_1(x) &= \frac{x - x_2}{-h} \cdot \frac{x - x_3}{-2h}, & L_1'(x) &= \frac{(x - x_2) + (x - x_3)}{2h^2} \\ L_2(x) &= \frac{x - x_1}{h} \cdot \frac{x - x_3}{-h}, & L_2'(x) &= \frac{(x - x_1) + (x - x_3)}{-h^2} \\ L_3(x) &= \frac{x - x_2}{h} \cdot \frac{x - x_1}{2h}, & L_3'(x) &= \frac{(x - x_1) + (x - x_2)}{2h^2} \end{aligned}$$

In order to compute f' at the middle node x_2 we obtain the so-called *central three-point formula*:

$$f'(x_2) \approx p'(x_2) = \frac{y_1 \cdot (-h) + y_3 \cdot h}{2h^2} = \frac{y_3 - y_1}{2h}. \tag{3.15}$$

Note that $L_2'(x_2) = 0$, so y_2 does not play a role anymore in the central three-point formula! In fact eq. (3.15) looks quite like eq. (3.13) for a simple reason: The central three-point formula gives the slope of the secant between the node x_1 and x_3, see fig. 3.4b. Still, $f'(x_2)$ is much better approximated using the central three-point formula than the two-point formula, because x_2 sits in the middle between x_1 and x_3. More precisely, in fig. 3.4b we obtain the tangent at x_2 of the parabola through the three nodes.

However, if we are given f in a tabulated form and we want to compute f' at the first or last position then we cannot use the *central* three-point formula. But instead of resorting to the two-point formula, we can still receive a better result using eq. (3.14), but we plug in the left resp. right node:

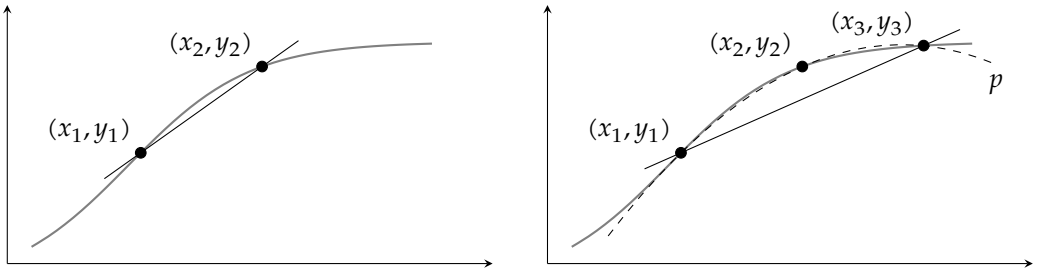
$$f'(x_1) \approx p'(x_1) = y_1 \frac{-3h}{2h^2} + y_2 \frac{-2h}{-h^2} + y_3 \frac{-h}{2h^2} = \frac{-3y_1 + 4y_2 - y_3}{2h}$$

3.6 Numerical integration

3.6.1 Basic integration formulas

Our goal is to numerically compute the integral of a function f over an interval $[a, b]$. We choose $n + 1$ equidistant nodes

$$x_k = a + k \cdot h,$$



(a) Two-point formula (b) Central three-point formula

Figure 3.4: Numerical derivative of $\tanh(x)$ using interpolation polynomials p with two (left) and three (right) nodes.

where $0 \leq k \leq n$ and $h = {}^{b-a}/_{n}$ is called the step size. Similar to numerical derivatives, we compute an interpolation polynomial p for f, which we can integrate. We set

$$y_k = f(x_k).$$

Using Lagrange's formula from eq. (3.8) we obtain

$$\int_a^b f(x)\ \mathrm{d}x \approx \int_a^b p(x)\ \mathrm{d}x = \sum_{k=0}^{n} y_k \int_a^b L_k(x)\ \mathrm{d}x.$$

Hence, the entire problem of numerical integration is essentially reduced to computing the terms

$$A_k = \int_a^b L_k(x)\ \mathrm{d}x. \tag{3.16}$$

We can simplify a bit more by transforming the interval $[a, b]$ to $[0, n]$ via a transformation

$$t = \frac{x - a}{h} \quad \text{resp.} \quad x = a + th.$$

Applying this transformation to the definition of $L_k(x)$ in eq. (3.9) we obtain

$$L_k(x) = \prod_{\substack{i=0 \\ i \neq k}}^{n} \frac{x - x_i}{x_k - x_i} = \prod_{\substack{i=0 \\ i \neq k}}^{n} \frac{t - i}{k - i}.$$

Applying this transformation to eq. (3.16) we then get

$$A_k = \int_a^b L_k(x)\ \mathrm{d}x = h \int_0^n L_k(x)\ \mathrm{d}t = h \int_0^n \prod_{\substack{i=1 \\ i \neq k}}^{n} \frac{t - i}{k - i}\ \mathrm{d}t.$$

With

$$\alpha_k^{(n)} = \int_0^n \prod_{\substack{i=1 \\ i \neq k}}^{n} \frac{t - i}{k - i}\ \mathrm{d}t \tag{3.17}$$

we have

$$A_k = h\alpha_k^{(n)}.$$

This now gives us the so-called (*closed*) *Newton-Cotes formula* of degree n:

$$\int_a^b f(x)\ \mathrm{d}x \approx h \sum_{k=0}^{n} f(x_k)\alpha_k^{(n)}. \tag{3.18}$$

Trapezoidal rule. We start with the case $n = 1$ and solve eq. (3.17):

$$\alpha_0^{(1)} = \int_0^1 \frac{t-1}{0-1}\ \mathrm{d}t = \frac{1}{2}$$
$$\alpha_1^{(1)} = \int_0^1 \frac{t-0}{1-0}\ \mathrm{d}t = \frac{1}{2}$$

Plugging the results in eq. (3.18) gives the *trapezoidal rule*:

$$\int_a^b f(x)\ \mathrm{d}x \approx h\frac{f(a)+f(b)}{2}$$

Simpson's rule. Increasing the number of nodes to $n = 2$ we obtain after some calculations

$$\alpha_0^{(2)} = \frac{1}{3} \quad \alpha_1^{(2)} = \frac{4}{3} \quad \alpha_2^{(2)} = \frac{1}{3}.$$

The resulting formula is known as *Simpson's rule*:

$$\int_a^b f(x)\ \mathrm{d}x \approx \frac{h}{3}\left(f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right)$$

3.6.2 Extended formulas

Instead of increasing the degree of the Newton-Cotes formulas and determining the $\alpha_k^{(n)}$ for large n, we actually use the simpler Trapezoidal or Simpson rule but apply it on parts of $[a, b]$. That is, we divide $[a, b]$ into N parts of equal length

$$H = \frac{b-a}{N}.$$

Each part is no a sub-interval $[z_{j-1}, z_j]$ with

$$z_j = a + jH$$

and we essentially compute

$$\int_a^b f(x)\ \mathrm{d}x = \sum_{j=1}^{N} \underbrace{\int_{z_{j-1}}^{z_j} f(x)\ \mathrm{d}x}_{I_j}$$

Using the trapezoidal rule for I_j gives the *extended trapezoidal rule*:

$$\int_a^b f(x)\ \mathrm{d}x \approx \frac{h}{2}\left(f(z_0) + f(z_N) + 2\sum_{j=1}^{N-1} f(z_j)\right).$$

Using the Simpson's rule for I_j gives the *extended Simpson's rule*. For convenience we set $x_j = a + j\frac{H}{2}$ for $0 \le j \le 2N$. Then we obtain

$$\begin{aligned}\int_a^b f(x)\ \mathrm{d}x &\approx \frac{h}{3}\left(f(x_0) + f(x_{2N}) + 2\sum_{k=1}^{N-1} f(x_{2k}) + 4\sum_{k=1}^{N-1} f(x_{2k-1})\right)\\ &= \frac{h}{3}\left(f(x_0) + 4f(x_1) + 2f(x_2) + 4f(x_3) + 2f(x_4) + \cdots + f(x_{2N})\right).\end{aligned}$$

3.7 Richardson extrapolation

3.7.1 Limit of a sequence

Polynomial interpolation is typically not well suited for extrapolation, i.e., for the computation of $f(x)$ when x is outside the interval of the interpolation nodes. However, one exception is when the nodes $x_1, \dots, x_n$ form the prefix of a sequence (x_i) that converges against x. Prime examples for such a case stem from numerical integration and differentiation.

Assume, for instance that we compute a numerical integral by lower sums with a step size h. Then we can compute the numerical integral for smaller and smaller step sizes and extrapolate the case where the step size h would become zero in the limit. This is a very useful general idea that goes by *Richardson extrapolation*. Richardson extrapolation is therefore a method to accelerate convergence of a sequence. A well known application is *Romberg integration*.

Let us consider a function f and we would like to extrapolate $f(0)$. With a fixed $h > 0$ we choose nodes

$$x_i = \frac{h}{2^{i-1}}.$$

The sequence $(x_i) = (h, h/2, h/4, \dots)$ converges to zero. The Neville recursion in eq. (3.11) for $f(0) = T_{1,\dots,n}$ now becomes a bit simpler:

$$\begin{aligned}T_i &= f(x_i)\\ T_{i,\dots,i+m} &= \frac{2^m T_{i+1,\dots,i+m} - T_{i,\dots,i+m-1}}{2^m - 1}.\end{aligned}$$

For certain applications, such as Romberg integration, it may turn out that f is an even function, which means $f(x) = f(-x)$. In this case we would also use an even polynomial function $p(x) = a_0 + a_2x^2 + a_4x^4 + \dots$ for extrapolation, where the odd terms $x, x^3, \dots$ are removed. Then we can actually substitute $t = x^2$, which gives $p(t) = a_0 + a_2t + a_4t^4 \dots$ and nodes $t_i = x_i^2 = {}^{2h}/_{4^{i-1}}$. This leads finally to an improved Neville recursion

$$T_i = f(t_i) \tag{3.19}$$

$$T_{i,\dots,i+m} = \frac{4^m T_{i+1,\dots,i+m} - T_{i,\dots,i+m-1}}{4^m - 1}. \tag{3.20}$$

3.7.2 Romberg integration

Romberg integration is Richardson extrapolation applied to the extended trapezoidal rule for numerical integration. That is, we consider

$$T(h) = \frac{h}{2}\left(f(x_0) + f(x_N) + 2\sum_{j=1}^{N-1} f(x_j)\right).$$

for $x_j = a + jh$ and $h = {}^{b-a}/_{N}$. Then it holds that

$$\lim_{h\to 0} T(h) = \int_a^b f(x)\ \mathrm{d}x.$$

It now turns out that the Taylor series of T actually only consists of even powers[9] and hence T is an even function. This now allows us to apply eq. (3.20) to extrapolate $T(0)$. This method is known as the *Romberg integration*.

[9]This is a consequence of the Euler-Maclaurin formula.

Part II

Computational Geometry

Chapter 4

Convex hull and range searching

4.1 Introduction

Many algorithmic problems in industrial application domains – GIS[1], computer graphics, CAD/-CAM[2] and robotics, logistics, drug exploration in pharmacy, et cetera – have a strong geometric flavor. For instance, computing the shortest path for a car on a street map, routing a wire on a PCB[3], computing offset paths for CNC machining, intersecting geometric shapes in a graphics design software, shooting and intersecting a ray with objects in a 3D scenery.

In this chapter we deal with an important class of problems of computational geometry, namely convex hulls and range searching. Convex hulls are a very basic concept in this field and often used as a preprocessing step for other algorithms. Range searching addresses problems of the following type: Imagine we are given a geographic map with houses illustrated as polygons. We click with a mouse on the map and ask whether and which house has been hit. Or we process a data set of points and we would like to cleanup the data set by removing points that are very close to others.

4.2 Convex hull

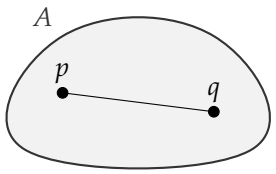
4.2.1 Convexity

Assume we are given n points in the plane that vaguely resemble a shape in the plane, as in fig. 4.3. How do we restore the shape in terms of a polygon? One answer to this could be the *convex hull*.[4] Let us first introduce a few definitions.

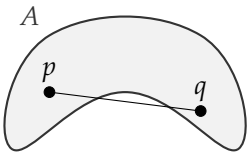
Definition 1. We call a set $A \subseteq \mathbb{R}^d$ *convex* if for any two points $p, q \in A$ the entire line segment $\overline{pq}$ is contained in A.

In fig. 4.1 we illustrate two sets in $\mathbb{R}^2$. The right set is non-convex because we can find two points p and q such the line segment $\overline{pq}$ is not entirely contained in A. So vaguely speaking, a set is convex if it has no "embayment" like the one in fig. 4.1b. As a general rule of thumb in mathematics and computer science, we can say that problems tend to become simpler if they are convex.[5]

[1] Geographic Information Systems
[2] Computer-Aided Design, Computer-Aided Manufacturing
[3] Printed-circuit board
[4] For non-convex shapes the so-called *alpha shape* is a generalization of the convex hull that can be used for this shape restoration task.
[5] Personal remark: Peter Gruber used to say that if mathematical functions and bodies have certain properties then



(a) A convex set in the plane.



(b) A non-convex set in the plane.

Figure 4.1: A set A is called convex if for each pair of points $p, q \in A$ the entire line segment $\overline{pq}$ is contained in A.

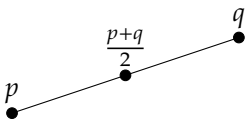
So far, we did not say how we exactly define $\overline{pq}$. Since $\mathbb{R}^d$ is a vector space we can define

$$\overline{pq} = \{\lambda p + (1-\lambda)q \colon \lambda \in [0,1]\}.$$

where we see p and q as vectors. This is a convenient definition from an algorithmic point of view, as we can simply *compute* all points on the line segment $\overline{pq}$ by choosing a $\lambda \in [0,1]$ and computing the vector $\lambda p + (1-\lambda)q$. We get the endpoint q for $\lambda = 0$, the endpoint p for $\lambda = 1$ and all points in between for $0 < \lambda < 1$. For instance, we get the mid point of $\overline{pq}$ for $\lambda = \frac{1}{2}$. The expression $\lambda p + (1-\lambda)q$ with $0 \leq \lambda \leq 1$ is called a *convex combination* of p and q, which is a special case of a linear combination. More generally, for points (or vectors) $p_1, \dots, p_n$ we call the linear combination

$$\lambda_1 p_1 + \lambda_2 p_2 + \cdots + \lambda_n p_n$$

a *convex combination* if $\sum_{i=1}^{n} \lambda_i = 1$ and $\lambda_i \in [0,1]$ for all $0 \leq i \leq n$. The set of convex combinations of two points p and q is the line segment $\overline{pq}$ and the set of convex combinations of three points p, q, r is the triangle formed by the three points. If we set the coefficients λ_i all equal to $1/n$ then we obtain the *center of gravity*, see fig. 4.2.



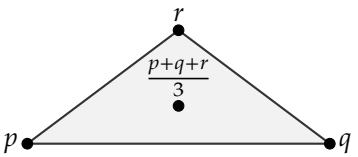


Figure 4.2: The set of convex combinations of two points forms a line segment. The set of convex combinations of three points forms a triangle. By setting the coefficients all equal we obtain the center of gravity, e.g., $(p+q)/2$ for the line segment and $(p+q+r)/3$ for the triangle.

Definition 2. We define the *convex hull* $\operatorname{conv}\{p_1, \dots, p_n\}$ of n points $p_1, \dots, p_n \in \mathbb{R}^d$ as the smallest[6] convex set that contains $p_1, \dots, p_n$. More generally, for a set $A \subseteq \mathbb{R}^d$ we define $\operatorname{conv} A$ as the smallest convex superset of A in $\mathbb{R}^d$.

---

convexity often makes these properties even stronger, see also [11, p. VI]. In this sense, *convexity* is the steroids of mathematical properties, which eventually also makes algorithms simpler and more powerful.

[6] "smaller" than B if $A \subseteq B$. However, it is actually not immediately clear why there would only one smallest convex super set as $\subseteq$ does not yield a total order. Hence, the convex hull may instead be defined as the infinite intersection of all convex supersets and then it is shown that the result is (i) convex and (ii) the smallest such set.

Let us consider the finite point set $p_1, \dots, p_n$ in fig. 4.3. By definition $\text{conv}\{p_1, \dots, p_n\}$ is the smallest convex superset of $\{p_1, \dots, p_n\}$. However, it is also the set of all convex combinations of $p_1, \dots, p_n$. In other words, in fig. 4.2 the line segment is equal $\text{conv}\{p, q\}$ and the triangle is equal $\text{conv}\{p, q, r\}$. The convex hull of a finite point set in the plane is a convex polygon and the convex hull of a finite point set in higher-dimensional space is a convex polyhedron.

An intuitive visualization of the *smallest* convex super set in the plane is the following: We place at each point p_i a nail and put a tight elastic rubber band around the point set. When the rubber band shrinks it attains the shape of $\text{conv}\{p_1, \dots, p_n\}$ as in fig. 4.3.

Adding or removing a nail (point) in the interior does not change the rubber band (convex hull). In particular, adding *any* convex combination of $p_1, \dots, p_n$ to the point set does not change the convex hull. This is a nice insight we can use to speed-up algorithms for convex hulls by reducing the number of input points: We take any three input points from fig. 4.3, form a triangle, and any input point in (the interior of) this triangle can be discarded.[7]

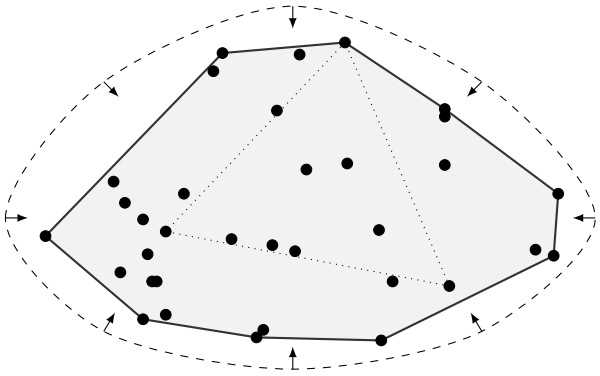


Figure 4.3: The convex hull of n input points in the plane forms a convex polygon. We can think of it as the resulting shape of a tight elastic rubber band (dashed line) around nails placed at the input points. Any triangle (dotted) formed by three input points is entirely contained in the convex hull.

4.2.2 Quickhull

Discarding points in triangles leads us directly to the divide-and-conquer convex hull algorithm *quickhull*, whose name is derived from the quicksort algorithm [3]. The quickhull algorithm is implemented in the well known qhull [2] software library.

The algorithm. The quickhull algorithm separately computes the upper and the lower part of the convex hull by repeatedly discarding points in triangles. It starts by finding the left-most point p and the right-most point q of the point set, see fig. 4.4a. For the upper part it considers the point r with largest orthogonal distance to $\overline{pq}$, but to the left of the ray $\overrightarrow{pq}$. We learned that all points in the triangle pqr can be discarded.

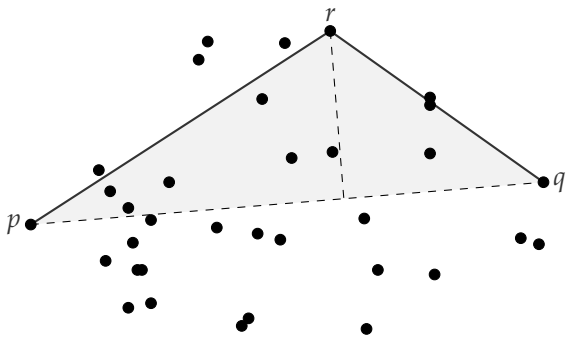
Next it recursively repeats by constructing triangles and discarding points: It considers the line $\overline{pr}$ and finds the point with largest orthogonal distance, again to the left of $\overrightarrow{pr}$. All points in there are discarded. Then it does the same for the ray $\overrightarrow{rq}$: It constructs a triangle with the point

[7] A more practical variant is to remove all points of large axis-aligned rectangles instead of triangles, see [12].

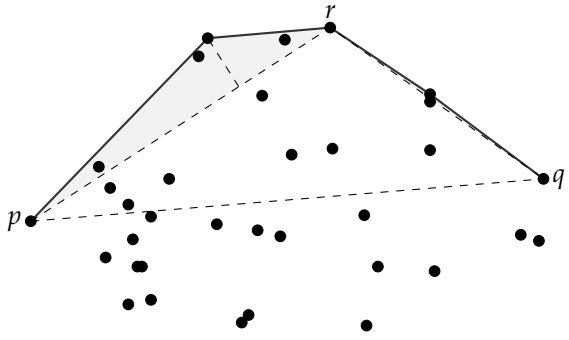
Algorithm 1 The quickhull algorithm to compute conv S.

procedure QUICKHULL(S) ▷ Compute convex hull of point set S
  $p \leftarrow \arg\min_{v \in S} v.x$ ▷ Left-most input point
  $q \leftarrow \arg\max_{v \in S} v.x$ ▷ Right-most input point
  return PARTIALHULL(S, p, q) + PARTIALHULL(S, q, p)
end procedure

procedure PARTIALHULL(S, p, q) ▷ Returns the partial convex hull left to $\overrightarrow{pq}$, including p but excluding q.
  $S \leftarrow \{v \in S : p, q, v \text{ is a left turn}\}$ ▷ Take only points left to $\overrightarrow{pq}$
  if $|S| \leq 1$ **then** ▷ Only at most one point left, we are done
    return $[p] + S$
  end if
  $r \leftarrow \arg\max_{v \in S} d(v, \overrightarrow{pq})$ ▷ Maximum orthogonal distance
  return PARTIALHULL(S, p, r) + PARTIALHULL(S, r, q)
end procedure



(a) Step 1: Find left-most point p and right-most point q. Find r farthest to the left of $\overrightarrow{pq}$. Discard all points in triangle pqr.



(b) Recursive step: Again find point farthest to the left of $\overrightarrow{pr}$ and discard all points in the new triangle. Same for $\overrightarrow{rq}$. Stop if there are no points left.

Figure 4.4: The quickhull algorithm separately computes the upper and the lower part of the convex hull. Here we only illustrate the upper part.

that is left-most to $\overrightarrow{rq}$ and discards points, see fig. 4.4b. When no points are left it is done with the upper part. The entire algorithm is summarized in algorithm 1.

The essential idea can be generalized to convex hulls in $\mathbb{R}^d$. Instead of triangles we have simplices[8] of higher dimension, see [3] for details.

Analysis. Let us consider algorithm 1. If the n input points are distributed in an unfavorable way then $O(n)$ many calls of PARTIALHULL cost $O(n)$ time which leads to a total of $O(n^2)$ worst case time. However, if the recursion is balanced well then the time complexity is $O(n \log n)$.

4.2.3 Graham scan

A simple algorithm that runs in $O(n \log n)$ time in the worst case is Graham's scan. Here we present a practical variant of the original algorithm that separately computes the upper and lower part of the convex hull and avoids sorting the points by angles.

Algorithm. We start by sorting the points lexicographically, i.e., primarily by the x-coordinate and secondarily by the y-coordinate. The left-most point p and right-most point q are the first and last point in this order. To compute the upper convex hull in fig. 4.5 we first remove all points right to $\overrightarrow{pq}$. If we connect the remaining points we obtain a polygonal chain that starts at p, makes a zig-zag line to the right, and ends at q. The upper convex chain only contains right turns, so our goal is to remove all left turns.

In PARTIALHULL in algorithm 2 we compute the upper convex hull iteratively. Assume that our last two points in our result is r and u and we are about to add v. In fig. 4.5 the triple r, u, v forms a right turn, so we simply add v. Next we are about to add w, but u, v, w is a left turn, so we remove v again. Now we consider r, u, w which is still a left turn, so we remove u, too. Now we have p, r, w which is a right turn, and we proceed with the next vertex.

Analysis. The first step of GRAHAMSCAN is a sort operation, which takes $O(n \log n)$ time. All the rest runs in $O(n)$ time, including the double loop in PARTIALHULL(.) The reason for the latter is the following: Each vertex $v \in S$ is appended once and removed at most once. Hence, the POPBACK operation in the inner loop can only be executed $O(n)$ times.

4.2.4 Lower bound on the time complexity

From the above analysis we learned that Graham scan is – from the perspective of time complexity – essentially sorting points. Interestingly, the opposite is also true: We can sort numbers using a convex hull algorithm. Assume we wold like to sort the numbers $x_1, \dots, x_n$. What we do is, we create points $p_i = (x_i, x_i^2)$ on a parabola. When we compute the convex hull with *any* algorithm then every point p_i is part of $\text{conv}\{p_1, \dots, p_n\}$, and we just read off the x_i in sorted order from the convex hull.

We know from algorithm theory that $O(n \log n)$ is a lower bound for sorting n numbers.[9] Assume that there is a convex hull algorithm that is faster than $O(n \log n)$. Computing the points on the parabola, computing the convex hull, and reading off the points from the result is therefore also faster than $O(n \log n)$. Hence, we sorted numbers in time less than $O(n \log n)$, which is a contradiction.

[8] A d-dimensional simplex, or d-simplex, is the convex hull of $d+1$ points in $\mathbb{R}^d$. It is a line segment in $\mathbb{R}^1$, a triangle in $\mathbb{R}^2$, a tetrahedron in $\mathbb{R}^3$.

[9] There is no sorting algorithm based on key comparison that takes less than $O(n \log n)$ time.

Algorithm 2 The Graham scan algorithm to compute conv S.

```
procedure GRAHAMSCAN(S)                          ▷ Compute convex hull of point set S
    S ←SORT(S)                                   ▷ Sort points lexicographically by (x, y)
    p ←S[0]                                      ▷ Left-most input point
    q ←S[−1]                                     ▷ Right-most input point
    return PARTIALHULL(S, p, q) + PARTIALHULL(REVERSE(S), q, p)
end procedure

procedure PARTIALHULL(S, p, q)                   ▷ Returns the partial convex hull left to pq, including p but excluding q.
    S ←{v ∈ S : p, q, v is a left turn}           ▷ Take only points left to pq
    H ←[]
    for v ∈ S do
        while |H| ≥ 2 and H[−2], H[−1], v is left turn do
            H.POPBACK( )                         ▷ Remove last element from H
        end while
        H.APPEND(v)
    end for
    H.POPBACK( )                                 ▷ q is last element in H, remove it
    return H
end procedure
```

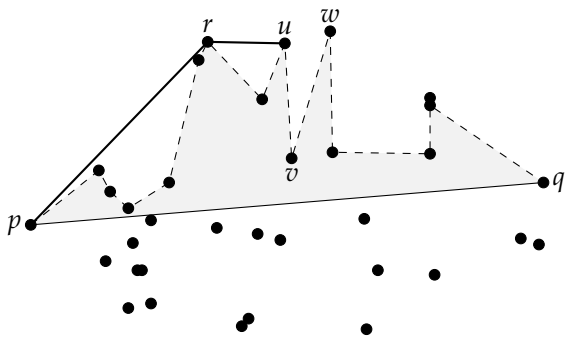


Figure 4.5: The Graham scan algorithm makes sure that while computing the upper part of the convex hull we only make right turns.

We just proved that any convex hull algorithm takes at least as much time as sorting numbers, namely $O(n \log n)$ time. What we also showed is that the Graham scan algorithm is worst-case *time optimal*.

4.2.5 Applications

The convex hull is a basic tool in computational geometry. The reconstruction of a (convex) shape from a set of points is one immediate application, but often we use convex hulls as one step of other geometric algorithms. For instance, the convex hull can be exploited to compute so-called Delaunay triangulations, which again have particularly nice properties for finite element methods, which again are used to solve differential equations, which again are used for numerical simulation of all kind of physical systems.

As another example, suppose we are given a point set S and a convex polygon P. We would like to test whether S fits into P. The answer is "yes" if and only if conv S fits into P. Hence, we can compute the convex hull of S first and only make the check for the vertices of the convex hull. If we many such tests for different P then the preprocessing step of computing conv S is done only once and can be reused for every new polygon P.

There are two general ideas to learn from the above example: (i) The convex hull can sometimes be used to speed up certain operations and (ii) typically problems are much easier to solve if the input is in some sense "convex". For instance, testing whether a point is in a convex polygon is much simpler than testing whether a point is inside a non-convex polygon. Computing a triangulation of a convex polygon is much simpler than computing a triangulation of a non-convex polygon. Motion planing for a convex robot is much simpler than for a non-convex robot.

So assume we want to test whether two complex three-dimensional objects intersect, e.g., for collision detection in a 3D racing game. Further assume that most of the time the answer is "no". We can speed up the complex task by first testing whether the convex hulls of the complex objects intersect, which is faster and simpler. Only if the answer is "yes" – we assume this happens infrequently – we have to make the slower and more complex test for the original objects.

Software packages like scipy for Python or MATLAB provide convex hull implementations. The following lines are from a Python interpreter shell. The `ConvexHull()` implementation is based on the qhull [2] library:

```
>>> import numpy as np
>>> from scipy.spatial import ConvexHull
>>> points = np.array([[0, 0], [1, 0], [1, 1], [0, 1], [0.5, 0.5]])
>>> hull = ConvexHull(points)
>>> points[hull.vertices]
array([[0., 0.],
       [1., 0.],
       [1., 1.],
       [0., 1.]])
```

4.3 Simple geometric constructions and predicates

Geometric algorithms – like quickhull or Graham's scan – are based on geometric predicates, e.g., given three points, are they forming a left turn or a right turn, or given two line segments, are they intersecting? By *predicate* we mean a property that is either true or false. That is, while

asking whether two line segments intersect defines predicate, computing the intersection point is the *construction* of geometric entities.

Construction of orthogonal vectors. A simple example for a construction is the rotation of a vector by the angle $\varphi = 90°$. This is achieved by simply switching coordinates and changing sign of one coordinate:

$$\begin{pmatrix} x \\ y \end{pmatrix} \cdot \begin{pmatrix} -y \\ x \end{pmatrix} = xy - yx = 0. \tag{4.1}$$

The vectors are orthogonal as the inner product is zero. For rotation by 90° we have two ways to switch sign; the one rotates in counterclockwise direction, the other in clockwise direction. This also evident from the rotation matrix:

$$\begin{pmatrix} \cos\varphi & -\sin\varphi \\ \sin\varphi & \cos\varphi \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} -y \\ x \end{pmatrix}$$

So a rotation by 90° can done very quickly *without* loss in numerical precision.

Orientation of three points. If we look carefully at the Graham scan in algorithm 2 then all we need from a geometric perspective is the test – the predicate – whether three points form a left turn. This is the only numerical part. Actually, there are many ways to phrase this question, see fig. 4.6. Given three points $p, q, r \in \mathbb{R}^2$, we may ask whether the triangle pqr is oriented counterclockwise, or p is left to the ray $\overrightarrow{pq}$, or the polygonal chain pqr forms a left turn.

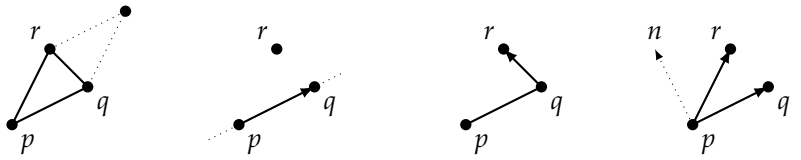


Figure 4.6: Four formulations of the same question: Is triangle pqr ccw oriented, is r left to $\overrightarrow{pq}$, is the polygonal chain pqr a left turn, and is the rotation of $\overrightarrow{pq}$ to $\overrightarrow{pr}$ less than a half turn? All are answered by $\Delta(p, q, r) > 0$.

A *bad* idea would be to actually compute and compare the angles of the vectors $\overrightarrow{pq}$ and $\overrightarrow{pr}$, because this involves trigonometric functions, which are in general inaccurate and slow. Let us instead consider the parallelogram formed by the vectors $\overrightarrow{pq}$ and $\overrightarrow{pr}$ in the left figure of fig. 4.6. There is a formula for the *signed* area of this parallelogram given by the following determinant:

$$\Delta(p, q, r) = \begin{vmatrix} p_x & q_x & r_x \\ p_y & q_y & r_y \\ 1 & 1 & 1 \end{vmatrix} = p_x q_y + q_x r_y + r_x p_y - p_x r_y - q_x p_y - r_x q_y. \tag{4.2}$$

By "signed area" we mean that $|\Delta(p, q, r)|$ is the area, but the $\Delta(p, q, r)$ is positive if pqr is counterclockwise (ccw) oriented, negative if pqr is clockwise (cw) oriented and zero if p, q, r are collinear[10]. In other words, $\Delta(p, q, r)$ is twice the signed area of the triangle pqr. The matrix in

[10]Collinear means that thy reside on a common line.

eq. (4.2) is actually related to homogeneous coordinates, which are heavily used in computer graphics and robotics.[11]

The last figure in eq. (4.2) is somewhat different in the sense that it compares the two vectors $q-p$ and $r-p$ rather than three points. Let us denote by n the vector we obtain by rotating the vector $q-p$ by 90° in ccw direction. We then ask whether $r-p$ projected onto n is positive, i.e., whether the inner product $n \cdot (r-p)$ is positive. Note that $n = (-(q_y-p_y), (q_x-p_x))$ by eq. (4.1) and hence

$$n \cdot (r-p) = \begin{pmatrix} -(q_y-p_y) \\ q_x-p_x \end{pmatrix} \cdot \begin{pmatrix} r_x-p_x \\ r_y-p_y \end{pmatrix} = (q_x-p_x) \cdot (r_y-p_y) - (q_y-p_y)(r_x-p_x) \tag{4.3}$$

On the other hand

$$\Delta(p,q,r) = \begin{vmatrix} p_x & q_x & r_x \\ p_y & q_y & r_y \\ 1 & 1 & 1 \end{vmatrix} = \begin{vmatrix} p_x & q_x-p_x & r_x-p_x \\ p_y & q_y-p_y & r_y-p_y \\ 1 & 0 & 0 \end{vmatrix} = (q_x-p_x) \cdot (r_y-p_y) - (q_y-p_y)(r_x-p_x)$$

and so we just learned that $\Delta(p,q,r) = n \cdot (r-p)$.[12] Note that eq. (4.3) only requires two multiplications, but at the expense of a numerical asymmetry as p_x and p_y are occurring twice.

Algorithm 3 An algorithm that tests wither the points pqr are in ccw orientation.

```
procedure ccw(p, q, r)
    return p_x q_y + q_x r_y + r_x p_y − p_x r_y − q_x p_y − r_x q_y
end procedure
```

We can summarize these insights in algorithm 3. Note that we deliberately do not return a boolean value but a numerical value for two reasons: First, we can distinguish three cases: ccw, collinear, cw. Secondly, the comparison against zero requires in general some epsilon threshold and the epsilon is application dependent, so we leave it to the caller.

Point location in triangles and convex polygons. Assume we are given a convex polygon $P = (p_1, \dots, p_n)$ with its vertices p_i given in ccw order, as illustrated in fig. 4.7. We would like to test whether a given point r is in P or not. In particular, P could be a triangle.

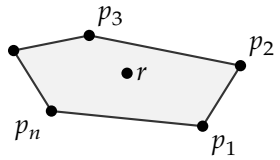


Figure 4.7: A point is in a convex polygon if it lies to the left of all edges in ccw direction.

The convexity of P makes this a very simple task as v is in P if and only if r lies to the left of all edges, i.e., to the left of all rays $\overrightarrow{p_1p_2}, \overrightarrow{p_2p_3}, \dots, \overrightarrow{p_np_1}$. In other words, if r lies outside of P then r lies to the right of some edge of P. So a simple algorithm that runs in $O(n)$ time is the one given in algorithm 4.

If, however, P would not have been convex then a reasonably simple solution could be to triangulate the polygon and make the test for every triangle. There are, however, more efficient methods.

[11]The idea behind homogeneous coordinates is to embed the plane $\mathbb{R}^2$ into three space as $\mathbb{R}^2 \times \{1\}$ by setting the z-coordinate of all points to 1. By this trick the translation of points becomes a linear operation. The vectors p, q, r span

Algorithm 4 An algorithm that tests in linear time whether a point is in a convex polygon.

```
procedure IN_CONVEX_POLYGON(P, r)
    for i ∈ {1, ..., n} do
        if CCW(p_i, p_{(i+1) mod n}, r) < 0 then
            return false
    return true
end procedure
```

Intersection of two line segments. Given two line segments $\overline{ab}$ and $\overline{cd}$, we can use the predicate ccw alone to test whether the two line segments intersect. All we have to do is to check that the endpoints of $\overline{cd}$ lie on different sides of $\overline{ab}$, and vice versa. So four invocations of ccw are sufficient.

4.4 Range searching and nearest neighbors

Consider the following task: We are given a large number n of points in the plane $\mathbb{R}^2$ and we would like to cleanup the dataset by removing duplicate points. A simple algorithm could iteratively construct a clean set C of points by starting with the empty set $\emptyset$ and iteratively adding a new point q to C if the closest point $p \in C$ is farther away than an $\varepsilon > 0$. Denoting by $d(p, q)$ the Euclidean distance between p and q, we have the following algorithm:

```
C ← ∅
for q ∈ P do
    if C = ∅ ∨ min_{p∈C} d(p, q) ≥ ε then
        INSERT(C, q)
```

A core operation in the above algorithm is to compute $\min_{p \in C} d(p, q)$. We call this a *nearest neighbor search* for a query point q. If we do it the naive way it takes time linear in the size of C, which makes the clean up algorithm quadratic time. But we can do much better!

We introduce data structures that can process C in order to allow so-called *range searching*: Given a query region Q the data structure returns all points of C contained in Q. Often Q is an axis-aligned rectangle, in which case we speak of *orthogonal range searching*. For our cleanup algorithm, we now set Q to a square of side length 2ε and center q and instead compute $\min_{p \in Q} d(p, q)$. Assuming that Q contains a constant number of points of C only, the cleanup algorithm runs in linear time now.

4.4.1 Geometric hashing

A simple but in practice often efficient method is geometric hashing. The name is taken from *hash tables* and the idea is similar: Instead of considering the full problem, we "hash" the input data into buckets of smaller size, which allows us to consider the problem only in a single (or few) buckets. We describe the technique in two dimensions, but it is obvious how to extend it to higher dimension.

---

a basis of $\mathbb{R}^3$, if not collinear, and the sign of $\Delta(p, q, r)$ tells us the orientation of the basis. Moreover, $\Delta(p, q, r)$ gives us the signed volume of the "unit cube" in this basis, i.e., the signed volume of $\{\lambda_1 p + \lambda_2 q + \lambda_3 r \colon \lambda_i \in [0, 1]\}$. The plane $\mathbb{R}^2 \times \{1\}$ can also be interpreted as the two-dimensional projective plane.

[12] And now we directly see that $\Delta(p, q, r)$ is the signed area of the parallelogram of the left figure in fig. 4.6.

The data structure. So assume that we have a point set C of size n that fits into some axis-aligned rectangular region R. We now partition R into cells by a regular $k \times m$ grid, see fig. 4.8 So each cell covers a subregion of R and for each cell we maintain a list of objects, namely points of C. That is, each point $p \in C$ is registered at the cell in which it resides.

Range searching. A range query for a rectangular range Q now works as follows: Determine all cells of the geometric hash that intersect with Q and report every point in those cells that is actually contained in Q.

Analysis. The geometric hash is of an advantage if only a significantly smaller fraction of the set C of points needs to be considered. Whether this is the case depends (i) on the distribution of the point set C and (ii) on our choice of m and k.

In a lucky situation the point set C is distributed uniformly over R. Assume further that we choose m and k such that the cells are close to a square and $m \cdot k$ is linear. Hence, on average a cell contains a constant number $n/m \cdot k$ of points and if Q is small such that it covers only a constant number of cells then we can answer the range query in constant time. More generally, the runtime is linear in the size of Q, i.e., in the number of cells intersected by Q.

We do not make $m \cdot k$ super-linear as we would pay super-linear space and time, even though they are mostly empty. If $m \cdot k$ is sub-linear then we still have linear space complexity, simply to store all points. However, each cell now contains super-constant many points and hence it takes super-constant time per cell for a range query.

If the point set is not "nicely" distributed, such that the points concentrate on some cells, we loose the advantage of geometric hashing. In particular, if some cells contain a linear number of points then we are essentially as good as the naive solution in terms of time complexity when we consider those cells. Fortunately, in many data sets of real-world applications points are quite nicely distributed.

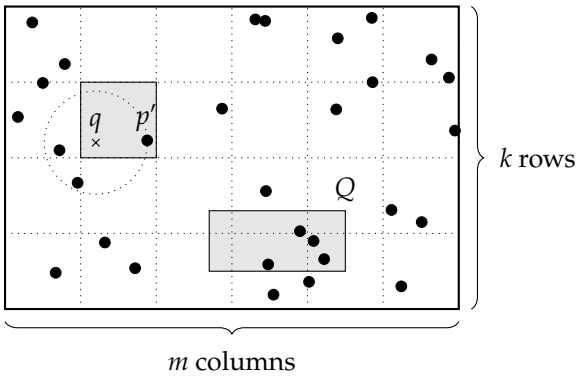


Figure 4.8: Geometric hashing for a point set in a rectangular region. It can be used for range queries for a query range Q or for nearest neighbor searches for a query point q. There is a candidate point p' in the cell of the query point q but the closest point is different to p'.

Nearest neighbor search. For a query point $q \in R$ we would like to find the closest point $p \in C$. First, we find a candidate point p' for being the closest point in C: We determine the cell of the query point q, see the shaded area in fig. 4.8. If this cell contains at least one point then we determine the point p' closest to q within this cell. If the cell was empty, we consider the ring

of neighboring cells for a candidate point. If these are also empty, we successively increase the search radius by adding another ring of cells. At some point we find a candidate p' that is the closest point within the considered cells, unless the entire geometric hash is empty.

Note that p' is not necessarily the closest point to q of all points in C, as fig. 4.8 illustrates. Hence we consider all cells that are intersected by the disk passing p' and centered at q. Within these cells we find the point closest to q among all points in C.

Bounded point query. Some applications, like the initial data cleanup example, are actually a mix of a nearest neighbor search and a range query in the sense that we are interested in the nearest neighbor within a (possibly small) query range Q. When Q is small we can restrict our nearest neighbor search to those cells and do not need to increase search range if no point has been found.

4.4.2 Hierarchical data structures

When the point set C is far from being uniformly distributed, the geometric hash suffers from the fact that grid cells still cover the region R in a uniform fashion. So some cells may take many points while many cells may remain even emtpy. The advantage of geometric hashing is mitigated when cells contain super-constant many points and entirely erased when cells contain a linear number of points.

Hence, we seek for a decomposition scheme that is adaptive. Various hierarchical data structures can do so by refining some tessellation scheme as we go deeper in the hierarchy.

Quadtrees

A quadtree is a tree where a node has either zero or four children. Each node N covers a rectangular region $R(N)$, with the root node covering the original region R. If a node N has children then $R(N)$ is split into four quadrants at a split point in the middle of $R(N)$ and each child node belongs to one of the four sub-rectangles, see fig. 4.9. The point set C is stored in the leaf nodes N of the quadtree, i.e, N stores the points contained in $R(N)$. When a certain region contains more points than other regions then we increase the depth of the hierarchy further and make therefore the subdivision finer. In this sense, a quad tree is like geometric hashing with a locally adaptive resolution.

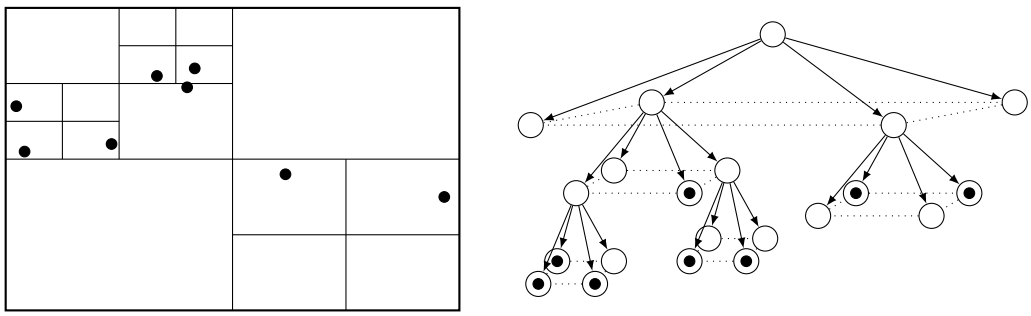


Figure 4.9: A quadtree with bucket capacity 1. Left: the cell decomposition of the quadtree. Right: The hierarchy of nodes, where each node covers a rectangular cell. The leaf nodes that contain a point are marked.

When we insert a new point p to the quadtree we traverse the quadtree from the root node down the hierarchy until we reached the leaf node N with $p \in R(N)$ and we add p to the node N. The question arises when to subdivide a leaf node. Similar to B-trees each (leaf) node has a fixed bucket capacity. If a new point p is to be added to the leaf node N that has reached its capacity then N is subdivided – we add four children – and p is instead inserted into the corresponding child node N' of N. It may happen that all points of N are actually located in $R(N') \subseteq R(N)$, so now N' has to be subdivided, and so forth. In fig. 4.9 we used a bucket size of 1, but in practice we would choose it larger.

When we perform a range query for an axis-aligned rectangular query range Q, we recursively traverse the quadtree top down starting at the root node. When we visit a node N and $R(N)$ intersects Q we have two cases: If N is a leaf node then we report all points of N that are contained in Q. If N is no leaf node then we recursively repeat for every child N' of N for which $R(N')$ intersects Q.

We can generalize a quadtree directly to $\mathbb{R}^3$, but now a node has eight children: One child for each octant at a split point. The resulting data structure is called an *octree*.

k-d trees

If we would not operate on the plane but on the one dimensional space $\mathbb{R}$ then we could use ordinary binary trees for range searching. For points in $\mathbb{R}^d$ with $d \geq 2$ we can use ordinary binary trees only if we choose one particular spatial direction to define a sorting order of the points, but in general this does not help for orthogonal range searching.

A k-d tree organizes the points of C as a binary tree, but we switch the sorting direction levelwise between the coordinate axes. In fig. 4.10 we illustrate an example for $\mathbb{R}^2$. The root node a is in the first level, so it divides the point set into left and right: All points in the first subtree of a are geometrically left to a, which are b, d, e, h. All points in the second subtree are geometrically right to a. The node b is in the second level, so it divides the remaining point set into below and above: All points in the first (second) subtree of b are below (above) b. Similar for node c. The node d is in the third level, so it divides into left and right again: The node h is in the right subtree of d because it is geometrically right of d.

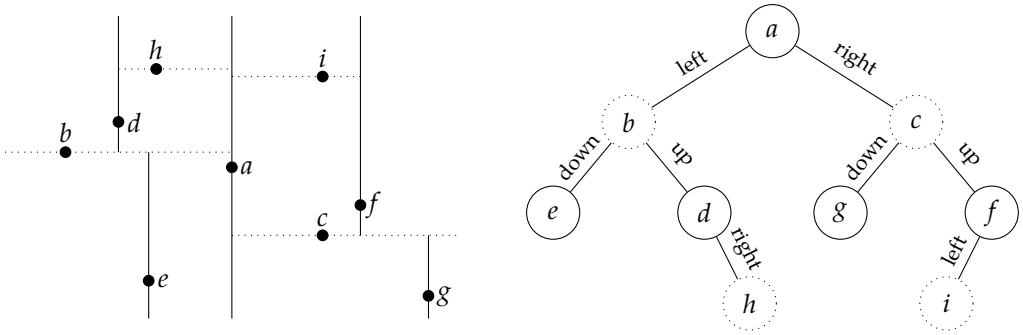


Figure 4.10: A k-d tree for a point set in $\mathbb{R}^2$. Left: The geometric tessellation scheme of the k-d tree. Right: The node hierarchy where every other level corresponds to a split along a different coordinate axis. Dashed lines on the left belong to dashed nodes on the right.

An orthogonal range search for a range Q is done by traversing the tree. In fig. 4.10 this means the following: If a is in Q then we report a. If Q reaches to the left (or right) of a then we recursively repeat for the left (or right) subtree of a. At the next level we do the same, only

left/right is replaced by below/above.

The range search is fast if we can drop large subtrees in the traversal of the k-d tree. For this reason we are interested in balanced k-d trees. If we compute a k-d tree from scratch we therefore start by sorting the point set by x-coordinates and choose the median point as root. We then take the left (right, resp.) subset, sort by y-coordinate, and again take the median as root of the subtree, and so on.

Recall that for quadtrees we have an initial region R in which the points lie, but k-d trees do not need such a thing. Also the location of the splitting is given by R for quadtrees, whereas for k-d trees the splitting lines are determined by the points of the point set.

Chapter 5

Voronoi diagram and Delaunay triangulation

5.1 Voronoi diagram of points

5.1.1 Definition and geometric properties

We would like to add a new power plant in a country to lower the load of the existing ones. To this end we would like to estimate the load of a power plant: We assume that households are uniformly distributed over the country and a household is served by its nearest power plant. So the load of a power plant is proportional to area of households where this plant is nearest to. What we receive is a geometric setup as illustrated in fig. 5.1, where the points are the power plants and the polygonal tessellation of the plane tells which households are served by which power plant. For instance, all households in the shaded, thick polygon are served by p.

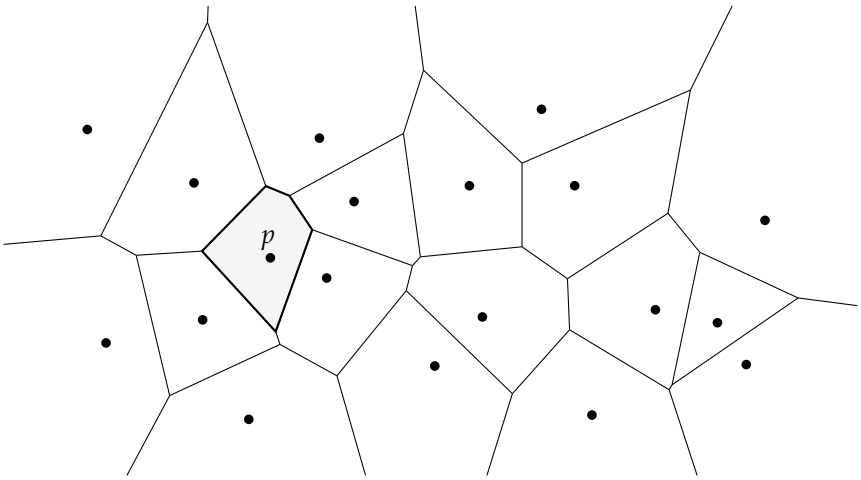


Figure 5.1: Voronoi diagram of points. The Voronoi region of p is shaded in gray and its boundary, the Voronoi polygon, is depicted thick.

A very similar problem formulation is known as the *post office problem*: The points in the plane, which are called *sites*, are post offices. A client posts a letter at its nearest post office, so for a post office p its service region is the set of points where p is the nearest neighbor.

This leads us to the definition of a *Voronoi diagram* of a set S of point sites $s_1, \dots, s_n$ in the Euclidean plane. For a site $s \in S$ we define its *Voronoi region* $\mathrm{VR}(s)$ as the nearest-neighbor region of s among S:

$$\mathrm{VR}(s) = \{p \in \mathbb{R}^2 \colon d(p,s) \leq d(p,s') \ \forall s' \in S\}. \tag{5.1}$$

If the notation is ambiguous on S then we add S as subindex and write $\mathrm{VR}_S(s)$. For the Euclidean plane the distance d is given by

$$d(p,q) = \|p - q\| = \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2}.$$

In an actual implementation we do not compute $d(p,s) \leq d(p,s')$ but the equivalent comparison $d(p,s)^2 \leq d(p,s')^2$ in order to get rid of the square root in the numerical computations. We define the *Voronoi polygon* $\mathrm{VP}(s)$ as the boundary of $\mathrm{VR}(s)$, which is denoted by $\mathrm{VP}(s) = \partial\,\mathrm{VR}(s)$. The *Voronoi diagram* $\mathrm{V}(S)$ of a set $S = \{s_1, \dots, s_n\}$ of sites in $\mathbb{R}^2$ is defined as

$$\mathrm{V}(S) = \bigcup_{s \in S} \mathrm{VP}(s).$$

The Voronoi diagram has many very nice geometric properties. First we note that we can rephrase eq. (5.1) as

$$\mathrm{VR}(s) = \bigcap_{\substack{s' \in S \\ s' \neq s}} H(s,s') \tag{5.2}$$

with

$$H(s,s') = \{p \in \mathbb{R}^2 \colon d(p,s) \leq d(p,s')\}.$$

The point set $H(s,s')$ is a half plane with the bisector between s and s' as boundary, see fig. 5.2. Each point x on the bisector is equidistant to s and s'. From eq. (5.2) we learn that every Voronoi region is the intersection of half planes and therefore a convex polygon. However, some of them might be unbounded, i.e., infinitely large.

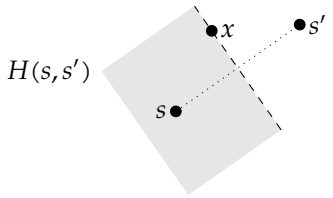
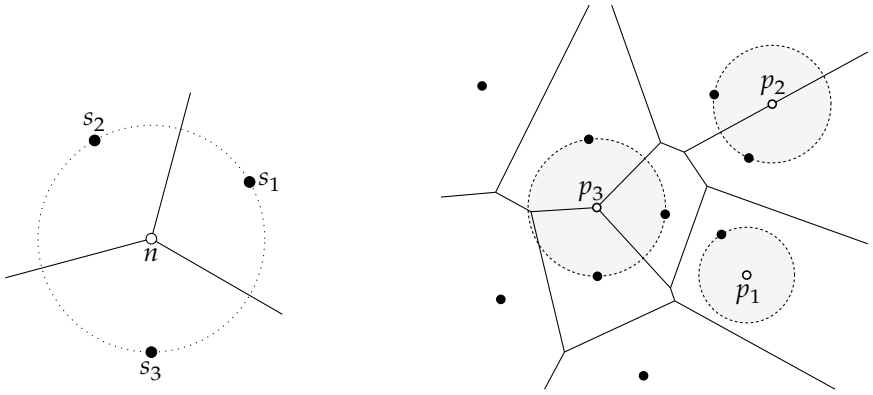


Figure 5.2: The half plane $H(s,s')$ with the bisector between s and s' as boundary.

The Voronoi diagram $\mathrm{V}(S)$ tessellates the plane into convex polygonal cells[1]. We call the edges of $\mathrm{V}(S)$ the *Voronoi edges* and the vertices of $\mathrm{V}(S)$ the *Voronoi nodes*. In fig. 5.3a we illustrate a Voronoi node n, the incident Voronoi edges and the sites s_i of the incident Voronoi regions. Note that n lies on the Voronoi edge and bisector between s_1 and s_2, so n is equidistant to both sites. But this also holds for all other Voronoi edges. Hence, a Voronoi node is equidistant to all sites of incident Voronoi regions.

Every point p in the plane has either one nearest neighbor $s \in S$ or two nearest neighbors $s_1, s_2 \in S$ or more than two. In the first case p lies in the interior of $\mathrm{VR}(s)$, in second case p lies on a Voronoi edge and in the third case p lies on a Voronoi node, see fig. 5.3b.

[1] In some literature the Voronoi region is also called Voronoi cell.



(a) Voronoi nodes. (b) Nearest neighbors.

Figure 5.3: Left: A Voronoi node n has three or more incident Voronoi edges. All sites $s_1, \ldots, s_k$ of the incident Voronoi regions are equidistant to n. Right: The points p_1, p_2, p_3 have one, two or more than two nearest neighbors in S.

5.1.2 Incremental construction

A simple algorithm to compute Voronoi diagrams is to construct it incrementally. If S contains at most three points, the Voronoi diagram is simple and looks like fig. 5.3a. If S contains more than three points, we can start with the first three, and iteratively add the remaining points.

So the problem we have to solve is the following: Given $\mathrm{V}(S)$ of a site set S, how to compute $\mathrm{V}(S \cup \{s\})$ for a new site $s \notin S$? Let us denote by S^+ the new site set $S \cup \{s\}$. So how can we modify $\mathrm{V}(S)$ in order to obtain $\mathrm{V}(S^+)$? At the end $\mathrm{V}(S^+)$ has to contain a new Voronoi polygon $\mathrm{VP}_{S^+}(s)$. The bisectors between the old sites remain unchanged, so many Voronoi edges well remain intact, some may be shortened, and some may be entirely deleted to clear space for $\mathrm{VP}_{S^+}(s)$. In fig. 5.4 we illustrate this situation.

Circular scan. Let $s_1 \in S$ be the[2] nearest neighbor of s among the old site set S. Since $s \in \mathrm{VR}_S(s_1)$, some parts of the old Voronoi region $\mathrm{VR}_S(s_1)$ of s_1 will be part of the new Voronoi region $\mathrm{VR}_{S^+}(s)$ of s and so the old region has to be truncated. More precisely, we consider the bisector between s_1 and s and obtain an edge e_1 of the new Voronoi polygon $\mathrm{VP}_{S^+}(s)$. The edge e_1 ends at an old Voronoi edge between s_1 and a neighboring site s_2. The old Voronoi region $\mathrm{VR}_S(s_2)$ is therefore also truncated by the new Voronoi region of s, and we can again construct a Voronoi edge e_2 on the bisector of s and s_2. We can keep doing that and eventually end up at s_1 again, because we know that $\mathrm{VP}_{S^+}(s)$ is a convex polygon.

We call this traversal scheme of $\mathrm{V}(S)$ a *circular scan* around s. After we circularly constructed the new Voronoi polygon $\mathrm{V}_{S^+}(s)$ and removed the parts of the Voronoi edges enclosed by new new polygon, we obtain the new Voronoi diagram $\mathrm{V}(S^+)$.

Topology-oriented construction. The circular scan involves many geometric constructions that build upon each other: We compute a bisector line, intersect it with a Voronoi edge, construct a new point, and keep doing so in order to construct a whole sequence of new Voronoi edges $e_1, e_2, \ldots, e_k$. In each step we *accumulate* numerical errors due to geometric computations, which

[2]In general, there could be many nearest neighbors if s sits on an edge or node of $\mathrm{V}(S)$. In this case we take any such nearest neighbor and call it s_1.

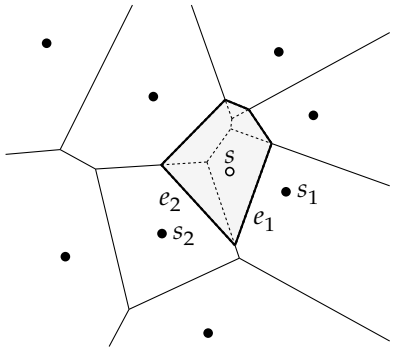


Figure 5.4: Inserting a site s creates a new Voronoi polygon $\mathrm{VP}(s)$ and removes the dashed line structure.

are based on floating-point arithmetic.[3] Due to numerical inaccuracies the last edge e_k will not exactly meet with the first edge e_1. In fact, we may end up in a spiral of new Voronoi edges that actually fails to meet e_1 again.

Sugihara [21] introduced a technique called *topology-oriented computation*. The general, abstract idea is to avoid geometric computations – and the continuous world as a whole – as far as possible and instead rely on discrete, integer-based information. In case of the incremental construction of Voronoi diagrams, we observe that the structure to be removed from $\mathrm{V}(S)$ is always a tree, i.e., connected and free of cycles. Roughly speaking, if the dashed structure in fig. 5.4 would contain a cycle then a whole Voronoi polygon would be removed, which is impossible as every site has a Voronoi region.

We can exploit this fact for a topology-oriented insertion of a new site s: We again consider the site s_1 whose Voronoi region $\mathrm{VR}_S(s_1)$ contains s and find a Voronoi node of $\mathrm{VP}_S(s_1)$ that is closer to s than s_1, which always exist. This node is definitely to be removed. Now we traverse $\mathrm{V}(S)$ and mark all nodes that are closer to s than their defining sites. Voronoi edges with both nodes marked are removed. Voronoi edges with only one marked node are truncated (partially removed). The key is now that if we would remove a whole cycle of Voronoi edges due to some numerical errors then we proactively avert that by breaking this cycle up at reasonable locations. That is, we *enforce* to remove a tree structure only. Therefore we help the implementation to produce topologically correct results; the implementation is *guided by* topological properties of Voronoi diagrams. Sugihara [21] demonstrated this technique for Voronoi diagrams, but the underlying idea is a powerful technique in general:

> Avoid geometric and continuous computations, leverage topological and discrete information.

5.2 Delaunay triangulation

5.2.1 Definition and geometric properties

Every Voronoi edge e of a Voronoi diagram $\mathrm{V}(S)$ belongs to two sites $s, s' \in S$ in the sense that both Voronoi polygons $\mathrm{VP}(s)$ and $\mathrm{VP}(s')$ contain e. If we draw a line segment between s

[3] There are alternatives to floating-point arithmetic, like Exact Geometric Computation (EGC) based on libraries like Core or LEDA.

and s' for every Voronoi edge then we obtain the picture in fig. 5.5. What we obtain is the so-called *Delaunay triangulation* $\mathrm{D}(S)$ of the point set S. We call the line segments of the Delaunay triangulation the (*Delaunay*) *edges*. Note that the Delaunay triangulation contains the convex hull of S as edges.

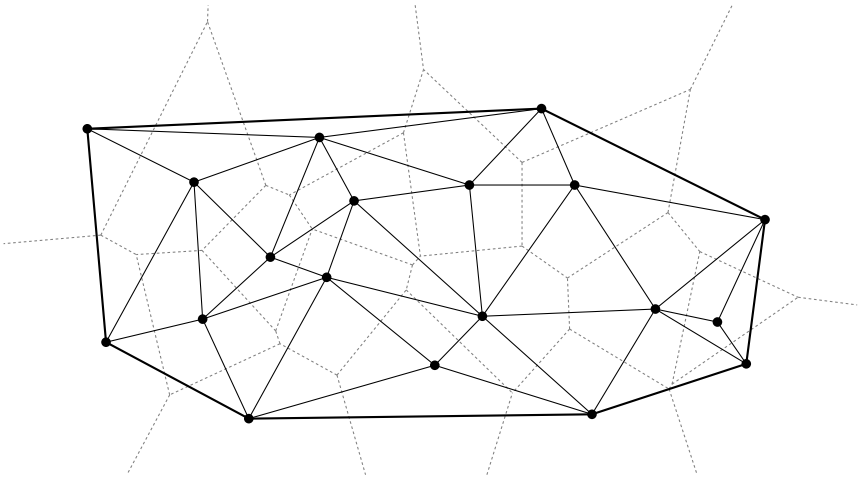


Figure 5.5: The Delaunay triangulation $\mathrm{D}(S)$ of a point set S is the dual graph of the Voronoi diagram $\mathrm{V}(S)$.

In general, a triangulation of S is defined as a *maximal planar subdivision* of S. We call a finite point set S together with some straight edges connecting vertices of S a *planar subdivision* if no edges cross. So *planarity* means that edges may only intersect at their endpoint. A planar subdivision is *maximal* if we cannot add any further edge without destroying planarity.

The Delaunay triangulation is a triangulation with particularly nice properties. If we consider the incident Voronoi edges to a Voronoi node n then each of them gives rise to a Delaunay edge. Hence, every Voronoi node gives rise to a Delaunay triangle, see fig. 5.6a.

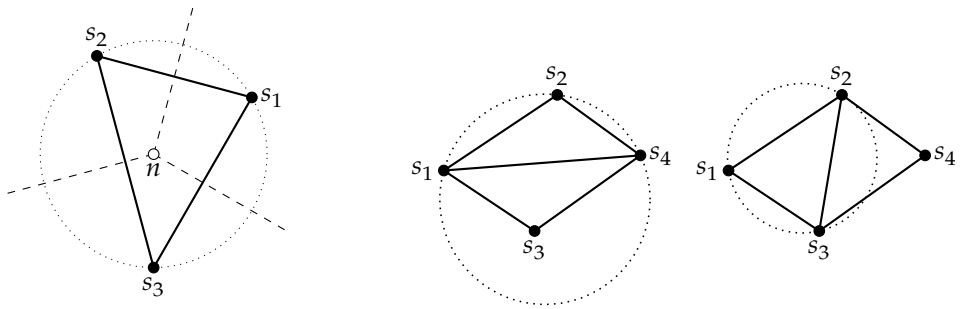
In fact, if a Voronoi node is of higher degree then three – because more than four sites $s_1, \ldots, s_k$ are cocircular – then we do not obtain a triangle but a regular k-gon and we have to triangulate this k-gon. Only in this case the Delaunay triangulation is not unique.

Another nice property is that the circumcircle of a Delaunay triangle does not contain any sites s in its interior. If this would be the case, the Voronoi node n to the Delaunay triangle would not have its defining sites as nearest neighbors but s, see fig. 5.6a. This property leads to a different definition of the Delaunay triangulation: For any three sites s_1, s_2, s_3 whose circumcircle is empty we add a Delaunay triangle.[4]

In fig. 5.6b the left triangulation is not Delaunay for exactly this reason. However, we can perform an *edge flip* operation: We remove an edge, obtain a convex 4-gon, and re-triangulate this 4-gon the other way. If the resulting 4-gon would not be convex then re-triangulating it the other way would destroy planarity, so we can only do an edge flip if the resulting 4-gon is convex. It can be shown that a triangulation of n points can be turned into *any* other triangulation – in particular the Delaunay triangulation – using only $O(n^2)$ edge flip operations.

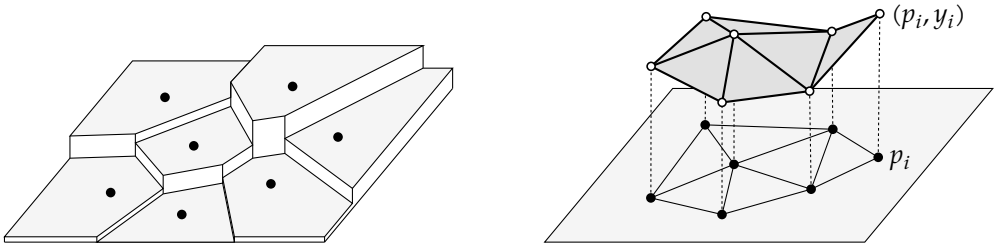
The edge flip operation in fig. 5.6b makes the triangles more acute, closer to equilateral triangles. In fact, it can be shown that the Delaunay triangulation is optimal in the sense that among all triangulations the smallest angle over all its triangles is maximized. This makes the Delaunay

[4]We have a special case if $k \geq 4$ sites are cocircular. In such situations we have to triangulate the resulting k-gon.



(a) A triangle for each node. (b) Edge flip to obtain Delaunay triangles.

Figure 5.6: The circumcircles of Delaunay triangles are empty. Left: For every Voronoi node n there is a Delaunay triangle of the defining sites. Right: The triangle (s_1, s_2, s_4) is not Delaunay, but flipping the edge (s_1, s_4) creates Delaunay triangles.



(a) Piecewise constant interpolation (b) Piecewise linear interpolation

Figure 5.7: Approximating a function $f\colon D \to \mathbb{R}$ over a domain $D \subset \mathbb{R}^2$ given at a point set $P = \{p_1, \dots, p_n\}$ with function values $y_1, \dots, y_n$. Left: A piecewise constant interpolation by computing $\mathrm{V}(P)$ and lifting each Voronoi region $\mathrm{VR}(p_i)$ by y_i. Right: A piecewise linear interpolation by computing $\mathrm{D}(P)$ and lifting each triangle vertex p_i by y_i.

triangle particularly "aesthetic" and, for instance, interesting for finite element methods.

5.2.2 Terrain interpolation

Assume we are interested in some function $f\colon D \to \mathbb{R}$, where $D \subseteq \mathbb{R}^2$ is some domain. We can interpret f is a scalar field over D. Take for example the temperature distribution or the height profile in a geometric map. Let us assume that the function f is not given explicitly, but only at certain points $p_1, \dots, p_n \in D$ and we would like to compute $f(p)$ for some arbitrary $p \in D$. In other words, we would like to interpolate f. In the following we denote by y_i the value associated to p_i.

From section 3.4.1 we recall that a piecewise linear function is a simple solution to this problem. However, in contrast to fig. 3.3a we now we deal with a piecewise linear surface as function graph of f. In fact, an even simpler solution would be a piecewise constant interpolation, a two-dimensional step function in some sense, but here again we would need to associate to each point p_i a neighborhood to which we assign the value y_i. For both approaches, the piecewise constant and the piecewise linear interpolation, we can use the Voronoi diagram resp. the Delaunay triangulation.

Piecewise constant interpolation. For any $p \in D$ we assign some function value of the set $\{y_1, \dots, y_n\}$. The most natural choice is probably to find the nearest neighbor p_i of p as the best representation among $\{p_1, \dots, p_n\}$ and then take y_i as function value of p. So what we effectively do is that we assign the function value y_i to the entire Voronoi region $\mathrm{VR}(p_i)$ and so we receive a function graph as in fig. 5.7a.

This simple scheme also works if the co-domain of f is not $\mathbb{R}$, but any discrete set L, e.g., a set of labels. We therefore want to interpolate a function $f\colon D \to L$ with a domain $D \subseteq \mathbb{R}^2$ and a co-domain L. The idea of "classifying" a point p by its nearest neighbor p_i is exactly what the k-NN (k-nearest neighbor) classification algorithm in machine learning does for $k = 1$. Actually, there are generalizations of Voronoi diagrams to so-called *higher-order Voronoi diagrams* that are in exact correspondence to the k-NN classification algorithm for $k \geq 1$.

Piecewise linear interpolation. The piecewise constant interpolation is not continuous. For many applications, in order to compute an interpolation $f(p)$ at a point p we would probably like to mix multiple function values y_i depending on the distance of p to p_i. So what we can do is to compute $\mathrm{D}(P)$ and lift each triangle vertex p_i by y_i. This gives as a continuous, triangulated surface as function graph, see fig. 5.7b.

Let us take any p in the domain D. We then project p vertically to the function graph to determine its interpolated function value $f(p)$. So if p is in the Delaunay triangle (p_1, p_2, p_3) then the function value of p is a mix of y_1, y_2 and y_3. More precisely, p is then a convex combination

$$p = \lambda_1 p_1 + \lambda_2 p_2 + \lambda_3 p_3.$$

The coefficients $\lambda_1, \lambda_2, \lambda_3$ are called the *Barycentric coordinates* of p (with respect to the affine basis p_1, p_2, p_3). The function value assigned to p is simply

$$f(p) = \lambda_1 f(p_1) + \lambda_2 f(p_2) + \lambda_3 f(p_3) = \lambda_1 y_1 + \lambda_2 y_2 + \lambda_3 y_3,$$

which is now a convex combination of the function values y_1, y_2, y_3. Any triangulation would work for this method of constructing a piecewise linear interpolation. However, as the Delaunay triangulation contains triangles that tend to be more equilateral, the (maximum) distance of a point p to its triangle vertices tend to be smaller. (The circumcenter of acute triangles is within the triangle.) Take for instance a point in the middle but slightly above the edge $\overline{s_1 s_4}$ in fig. 5.6b. Then such a point is better interpolated using the Delaunay triangulation at the right.

5.2.3 Planar graphs

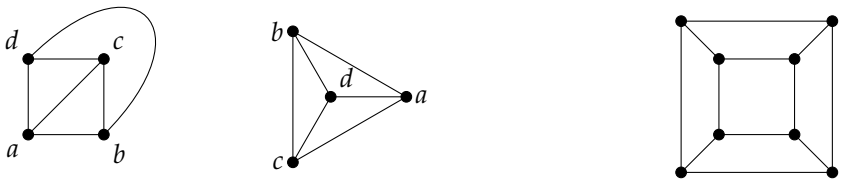
One important question is how many edges does $\mathrm{D}(S)$ posses if S contains n points? If we place an edge between any pair of points then we place a quadratic number of edges, but this does not form a triangulation in general.[5]

We can answer this question by means of graph theory. First, a *graph* $G = (V, E)$ is a pair of a set of vertices V and a set E of edges between vertices. For instance, we would model communication networks as graphs, where the routers are vertices and interlinks are edges. But also the vertices and edges of a convex polyhedron form a graph, the so-called *edge graph* of the polyhedron.

If we can draw a graph G in the plane in a way such that the edges do not cross then we call G a *planar graph*. A planar graph G may possess many planar drawings, which we call a *planar embedding* of the graph. The wire routing of a circuit on a PCB layer is required to form a planar embedding. Also the edge graph of any convex polyhedron[6] is planar. See fig. 5.8 for examples.

[5] Actually, if $n \geq 5$ then it does never form a triangulation.

[6] A convex polyhedron is the convex hull of finitely many points in $\mathbb{R}^3$.



(a) Two planar embeddings (b) Edge graph of the cube

Figure 5.8: Planar graphs have a crossingfree drawing. Left: Two different embeddings of the same graph with four vertices a, b, c, d. It is the edge graph of the tetrahedron. Right: The edge graph of the cube with six quadrilateral faces.

If we consider a planar embedding of a planar graph G then the plane is tessellated into *faces* formed by the edges, see also fig. 5.8. The main theorem on planar graphs is given by *Euler's formula*[7]:

Theorem 1. *For any connected, planar graph G with v vertices, e edges and f faces it holds that*

$$v - e + f = 2. \tag{5.3}$$

Here we also count the unbounded, infinitely large face as an face. It is called the *outer face*. In fig. 5.8a the outer face is a triangle and in fig. 5.8b the outer face is a quadrilateral. As an example, the edge graph of a cube has 8 vertices, 12 edges and 6 faces and $8 - 12 + 6 = 2$ and the edge graph of a tetrahedron has 4 vertices, 4 faces and 6 edges and $4 - 6 + 4 = 2$.

A triangulation is by definition planar. The triangles are the faces. Every edge belongs to two faces and every face has three edges. Draw a little dot on either side of an edge, then you have placed $2e$ dots. On the other hand, each triangle received three dots, one from each edge, so we also placed $3f$ dots and therefore $2e = 3f$. If we plug this into Euler's formula we get

$$6 = 3v - 3e + 3f = 3v - 3e + 2e = 3v - e$$

and therefore

$$e = 3v - 6.$$

Note that this only holds if all faces, including the outer face, form triangles, as in fig. 5.8a. However, since triangulating a non-triangular face only adds edges but no vertices, we obtain by eq. (5.3) for *any* planar graph the inequality

$$e \leq 3v - 6 \tag{5.4}$$

and equality holds if all faces are triangles. In other words, planar graphs possess at most a linear number of edges! Many algorithms for graphs have a time complexity that depends on the number of edges. They perform better on planar graphs than arbitrary graphs. Prime examples are the computation of shortest paths[8] or minimum spanning trees[9].

Since the Voronoi diagram $\mathrm{V}(S)$ and the Delaunay triangulation $\mathrm{D}(S)$ form (embeddings of) planar graphs, this is also true for them: they have only a linear number of Voronoi edges resp.

[7] Dt. Euler's Polyedersatz.
[8] Take for instance Dijkstra's algorithm.
[9] Take for instance Kruskal's algorithm.

Delaunay edges. This insight – and a bit more – can be used to prove that if we construct $\mathrm{V}(S)$ incrementally and choose the points in a random fashion then the *expected* runtime is $O(n \log n)$. However, there are also deterministic $O(n \log n)$ algorithms. Based on $\mathrm{V}(S)$ we can compute $\mathrm{D}(S)$ in $O(n)$ time.

The qhull library – which is used by scipy and MATLAB – can actually compute $\mathrm{V}(S)$ and $\mathrm{D}(s)$ by means of the convex hull in higher dimensions. It can be shown that if we lift the point set S into $\mathbb{R}^3$ by projecting them on a paraboloid then their convex hull forms a polyhedron, whose edges projected back onto the plane gives the $\mathrm{D}(S)$.

5.2.4 Euclidean minimum spanning trees

An *edge-weighted* graph G is a graph where we add a weight to each edge. For our purposes we interpret the "weight" as some kind of length of an edge, but in general it could also be an abstract label. For instance, G could be a street network with vertices being cities and the edge weights are the travel times between the cities.

A famous problem is the so-called *traveling salesman problem* (TSP). It asks for the *shortest* roundtrip (a cyclic tour) within G that visits all vertices (e.g., cities in a street network). If the vertices are embedded in the Euclidean plane with straight edges between any pair of vertices and the edge weights are the Euclidean lengths of the edges then we also call this problem the *Euclidean traveling salesman problem* (ETSP).

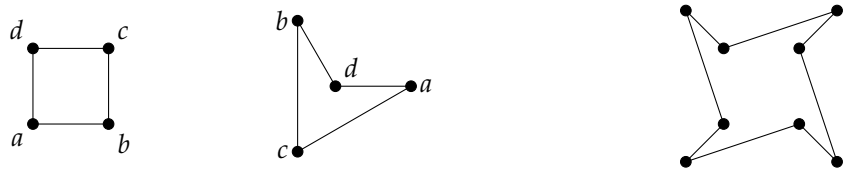


Figure 5.9: Solutions to the Euclidean TSP for the point sets in fig. 5.8.

The Euclidean TSP is very much related to the computation of the *Euclidean minimum spanning tree*. A *spanning tree* of a graph G is a subgraph T we obtain from G by removing edges such that (i) T is still connected but (ii) acyclic. That means that we can walk from any vertex to any other vertex along edges of T, but there are no cycles formed by the edges of T. A *minimum spanning tree* (MST) is a spanning tree with a minimal total weight, i.e., the sum of edge weights is minimal. If the vertices are embedded in the Euclidean plane, with straight edges existing between any pair of vertices, and edge weights given by their Euclidean length, then we call its MST the *Euclidean minimum spanning tree* (EMST), see fig. 5.10.

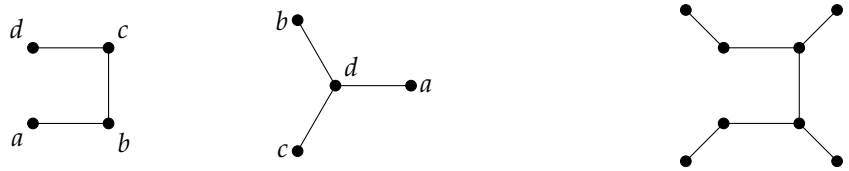


Figure 5.10: The Euclidean minimum spanning trees of the point sets in fig. 5.8.

We could directly compute the EMST using Kruskal's algorithm for the computation of the MST of arbitrary edge-weighted graphs. However, we would need to consider $e = \binom{n}{2} = \frac{n(n-1)}{2}$ edges between all pairs of points and Kruskal's algorithm has a time complexity of $O(e \log e)$.

It can be shown that the EMST is part of the Delaunay triangulation, see fig. 5.11. We already know that the Delaunay triangulation has only a linear number of edges and can be computed in $O(n \log n)$ time. Hence, we can compute the EMST in $O(n \log n)$ time.

The ETSP problem is NP-hard, so there are no polynomial time algorithms assuming $P \neq NP$. However, one can use the EMST to compute so-called constant-factor approximations that are no worse than a factor of $c > 1$ longer than the ETSP solution. Christofides' heuristic, for instance, computes a 1.5-approximation in $O(n^3)$ time. In 2010 Aroa and Mitchell received the Gödel Prize for their independent discovery of (families of) polynomial-time algorithms that compute approximations for arbitrarily small $c > 1$. They discovered so-called *polynomial-time approximation schemes* (PTAS) for Euclidean TSP.

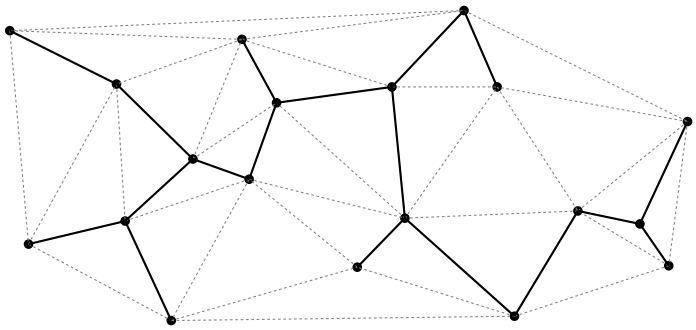


Figure 5.11: The Delaunay triangulation (dotted) contains the Euclidean minimum spanning tree (solid).

Chapter 6

Skeleton structures

6.1 Motivation

We denote by a *polygon with holes* a set $P \subset \mathbb{R}^2$ that results from a polygon after removing polygons (holes) and we assume the holes do not reach the outer boundary of the original polygon. A polygon with holes, P, could model a two-dimensional terrain of a mobile disk-shaped robot V, or P could model a workpiece and V is an NC tool. On such polygons with holes we will introduce the notion of *skeleton structures* that will allow us to solve the following exemplary problems:

- First, we could ask whether V could reach a certain target position $q \in P$ without leaving P. We interpret the boundary of P as walls.
- The diameter of V could be too large to pass through certain corridors of P. So a very related task would be to identify the so-called *bottlenecks* of P.
- Computing paths within P leads us to the desire to represent P as a transportation network, i.e., a graph structure consisting of nodes and edges. So another related goal is to transform the geometric shape P into a network structure.
- When we interpret P also as a workpiece that should be milled out by an NC machine or a garden that should be mowed. In either case we would like to compute a tool path for the CNC machine or the mowing robot.

6.2 Medial axis

The *medial axis* $\mathrm{M}(P)$ of a polygon with holes, P, consists of all points $p \in P$ with the property that the largest disk within P and centered at p touches the boundary of P at two or more points. Put in different words, $\mathrm{M}(P)$ consists of all points $p \in P$ that do not have a unique closest point to the boundary of P, see fig. 6.1.

We can actually interpret $\mathrm{M}(P)$ as a *transformation* that contracts the shape P to a 1-dimensional version while still somehow capturing topological features of P. For instance, every "loop" in P corresponds to "loop" in $\mathrm{M}(P)$. But also if P resembles the shape of a hand with five fingers then $\mathrm{M}(P)$ contains a path for each finger. This is why the medial axis – also known as *medial axis transform* (MAT) – is widely known in the image processing domain for shape description, reconstruction and comparison.

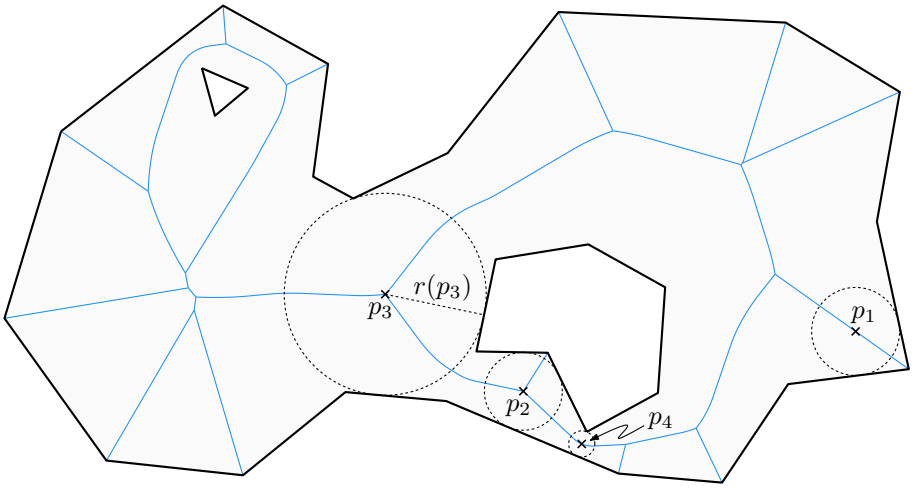


Figure 6.1: The medial axis $\mathrm{M}(P)$ in blue of a polygon with holes P shaded in gray. For some points p_i the disk $D(p_i, r(p_i))$ with clearance radius $r(p_i)$ is shown in dashed lines. The point p_4 constitutes a bottleneck.

Shape reconstruction. The application for *shape reconstruction* stems from the following observation: Assume we do not know P, but we know $\mathrm{M}(P)$ and for each $p \in \mathrm{M}(P)$ we know the distance $r(p)$ to the boundary of P. In fact, we interpret r as a function $r\colon \mathrm{M}(P) \to \mathbb{R}$ and call $r(p)$ the *clearance radius* at p, see fig. 6.1. Then we can reconstruct P by

$$P = \bigcup_{p \in \mathrm{M}(P)} D(p, r(p)), \tag{6.1}$$

where $D(p, r(p))$ denotes the disk centered at p and with radius $r(p)$. That is, if we place at each $p \in \mathrm{M}(P)$ a disk of radius $r(p)$ and build the union of those disks then we obtain P again. In this sense, eq. (6.1) is the inverse transformation of the medial axis transformation. We could exploit eq. (6.1) to modify the shape P by modifying $r(p)$, e.g., making certain "corridors" slimmer or wider by reducing or increasing $r(p)$ at those points $p \in \mathrm{M}(P)$. In some sense we edit the geometry of the shape but we leave the topology as is. Of course, if we change $r(p)$ by too much then the topology of the resulting shape changes, e.g., holes may get filled up or bays may turn into cavities or bridges may disrupt.

Bottlenecks. Intuitively, a bottleneck is a narrow place of a corridor of P, a location where the boundary lines of P come close. The medial axis $\mathrm{M}(P)$ allows us to turn this intuitive description into a mathematical precise definition: A point $p \in \mathrm{M}(P)$ is called a *bottleneck* if p is a local minimum of the function $r\colon \mathrm{M}(P) \to \mathbb{R}$ as shown in fig. 6.1.

Note that a *minimum* of r is a place $p \in \mathrm{M}(P)$ with the following property: There is a small neighborhood around p such that for any point p' in this neighborhood $r(p') \geq r(p)$ holds. In more details, there is a $\varepsilon > 0$ such that for any $p' \in \mathrm{M}(P)$ with $d(p, p') < \varepsilon$ it holds that $r(p') \geq r(p)$. Here d refers to the Euclidean distance[1].

[1] It actually does not matter so much which distance (metric) we take but which topology the metric induces.

6.3 Generalized Voronoi diagrams

6.3.1 Introduction

The medial axis is a useful tool in computational geometry, so the practical question arises how to actually compute it. It turns out that the medial axis $\mathrm{M}(P)$ is actually part of a more general structure, namely the generalized Voronoi diagram $\mathrm{V}(P)$ of a polygon with holes P.

In section 5.1 we introduced the Voronoi diagrams of a point set in the Euclidean plane as a nearest-neighbor cell decomposition. We can generalize this idea from a point set to a more general set S of *sites*. The Voronoi diagram $\mathrm{V}(S)$ of the site set S is then called a *generalized Voronoi diagram*. There are other ways to generalize the Voronoi diagram of points, e.g., by generalizing the metric of the Euclidean plane or by considering cells defined by k nearest neighbors instead of one. But here we focus on generalized sites.

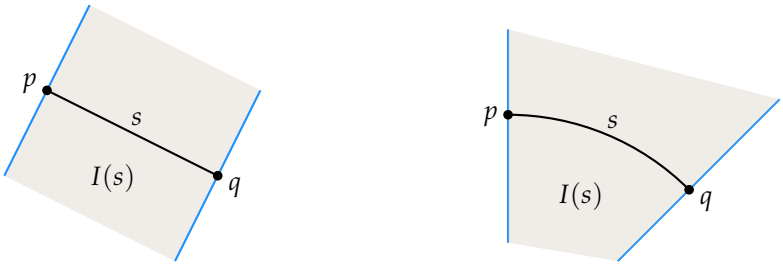
6.3.2 Straight-line segments and circular arcs

In industrial practice, we are particularly interested in points and straight-line segments and circular arcs. So let us denote by S a finite set of sites consisting of points, straight-line segments and circular arcs. To overcome some technical complications, we assume that for each straight-line segment or circular arc $s \in S$ also both its endpoints are individual sites in S.

In fig. 6.2 we illustrate the Voronoi diagram as a nearest-neighbor cell decomposition of the plane that results from a straight-line segment and its endpoint and likewise for the circular arc. Mathematically, a point q in the Voronoi cell in an endpoint of s is just as close to the endpoint as to s itself. So in order to have a line-like Voronoi edge – with zero area – we have to perform a little exercise by introducing the concept of *cone of influence* $\mathrm{I}(s)$ of a site s. The idea is to restrict the Voronoi region $\mathrm{VR}(s)$ of a site s to a certain region, its cone of influence $\mathrm{I}(s)$. The cone of influence is illustrated in fig. 6.2 and defined as follows:

$$\mathrm{I}(s) = \begin{cases} \mathbb{R}^2 & \text{if } s \text{ is a point} \\ \text{orthogonal strip spanned by } s & \text{if } s \text{ is a straight-line segment} \\ \text{cone spanned by } s & \text{if } s \text{ is a circular arc} \end{cases}$$

A different way to define $\mathrm{I}(s)$ for straight-line segments or circular arcs s is by means of an intersection of two half spaces: At each end point of s we place a half space that contains s such that s is locally orthogonal to the boundary of the half space.



(a) Straight-line segment (b) Circular arc

Figure 6.2: The Voronoi diagram of a straight-line segment s and its endpoints p, q and likewise for the circular arc. The Voronoi region $\mathrm{VR}(s)$ of s is restricted to its cone of influence $\mathrm{I}(s)$.

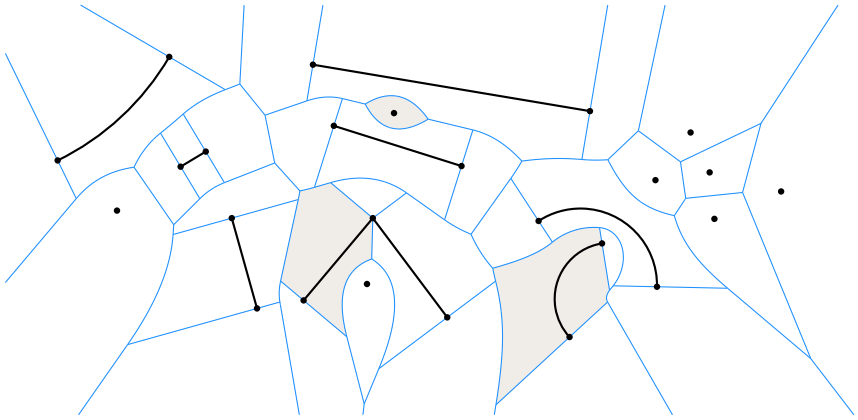


Figure 6.3: The generalized Voronoi diagram of points, straight-line segments and circular arcs. A few Voronoi regions are shaded in gray.

So now we can more or less copy over the definition of Voronoi diagram of points, see section 5.1, to the more general setting:

$$\mathrm{V}(S) = \bigcup_{s \in S} \mathrm{VP}(s) \tag{6.2}$$

where $\mathrm{VP}(s)$ is the Voronoi "polygon" and defined as the boundary $\partial\,\mathrm{VR}(s)$ of the Voronoi region $\mathrm{VR}(s)$, which again is defined as

$$\mathrm{VR}(s) = \{p \in \mathrm{I}(s) : d(p,s) \leq d(p,s') \ \forall s' \in S\}. \tag{6.3}$$

Figure 6.3 shows the generalized Voronoi diagram of a couple of sites. A few Voronoi regions have been shaded in gray and form the nearest-neighbor cells of the respective sites.

A few remarks regarding the definition of the generalized Voronoi diagram are in order. By $d(p,s)$ we mean the infimum distance between p and s, which means that

$$d(p,s) = \inf_{q \in s} d(p,q).$$

In fig. 6.4 we illustrate what this means in general: Roughly speaking, $d(p,A)$ for a point p and a point set A is the smallest distance possible between p and any point $q \in A$. The very same idea could be generalized to the infimum distance $d(A,B)$ between two point sets A and B of the plane.

There is a technical catch in eq. (6.3) which we ignored so far: For circular arcs s the definition in eq. (6.3) can cause irregularities in $\mathrm{VR}(s)$ in form of line-like needles attached to the "region body". For instance, the point p in fig. 6.5 would be member of both Voronoi regions $\mathrm{VR}(s)$ and $\mathrm{VR}(s')$, and so is the entire line segment to the circle center. Hence, the Voronoi polygons are no

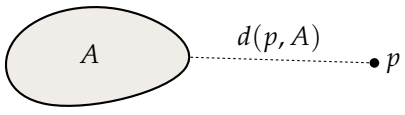


Figure 6.4: The infimum distance $d(p,A)$ between a point p and a point set A.

longer closed curves. In order to remove these irregularities, the definition of $\mathrm{VR}(s)$ is adapted to

$$\mathrm{VR}(s) = \mathrm{cl}\{p \in \mathrm{int}\,\mathrm{I}(s) : d(p,s) \leq d(p,s') \ \forall s' \in S\}, \tag{6.4}$$

which means that we first take the topological interior and then form the topological closure.[2] In simple words, the boundary of $\mathrm{I}(s)$ is first removed, which cuts off the needles, and then the boundary of the resulting set – the intended Voronoi polygon, a closed curve – is re-added. Details can be found in [15, 14].

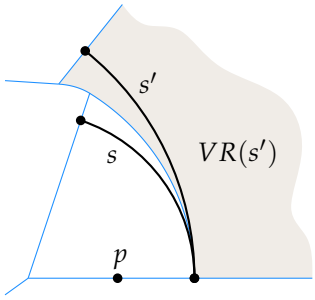


Figure 6.5: The point $p \in \mathrm{VR}(s)$. But since $d(p,s) = d(p,s')$ the point p would also be part of $\mathrm{VR}(s')$ if we would use eq. (6.3) instead of eq. (6.4).

6.3.3 Polygon with holes

Let us consider a polygon with holes P. Its boundary consists of vertices and edges. Let us denote by S the set of vertices and straight-line edges of P. We then define the Voronoi diagram $\mathrm{V}(P)$ of P simply as $\mathrm{V}(S)$. Depending on the application and the context we may restrict the Voronoi diagram to P and ignore everything outside P.

In fig. 6.6 we show the Voronoi diagram $\mathrm{V}(P)$ of the same polygon with holes P as in fig. 6.1. We realize that they are essentially the same. More precisely, $\mathrm{M}(P) \subseteq \mathrm{V}(P)$ and $\mathrm{V}(P) \setminus \mathrm{M}(P)$ only consists of the Voronoi edges incident to reflex[3] vertices of P. This makes sense if we remember that $\mathrm{M}(P)$ consists of all points that have two nearest points on the boundary of P. These points therefore lie on the bisector between vertices and edges of P and no other site can be closer, so they must also lie on Voronoi edges. Hence, one way to compute the medial axis is to compute the Voronoi diagram and drop the Voronoi edges incident to reflex vertices.

Both, the concept of the medial axis and the concept of generalized Voronoi diagrams can be applied to polygons with holes, where the boundary elements are not only straight-line segments but also circular arcs. For industrial applications, especially in the CAD/CAM domain, this is of high relevance.[4]

[2] In topology we call $O(c,r) = \{p \in \mathbb{R}^2 : \|p - c\| < r\}$ an open disk centered at $c \in \mathbb{R}^2$ and with radius r. For a set A we define its interior, $\mathrm{int}\,A$, as the set of points $p \in A$ for which we can find a small enough $r > 0$ such that $D(p,r) \subseteq A$. The closure $\mathrm{cl}\,A$ is defined by $(\mathrm{int}\,A^c)^c$, where $A^c = \mathbb{R}^2 \setminus A$ is the complement of A. The boundary of ∂A is defined as $\mathrm{cl}\,A \setminus \mathrm{int}\,A$.

[3] A reflex vertex is a non-convex vertex.

[4] In theory, the class of shapes P for which $\mathrm{M}(P)$ and $\mathrm{V}(P)$ have meaning can be extended to a significantly more general class.

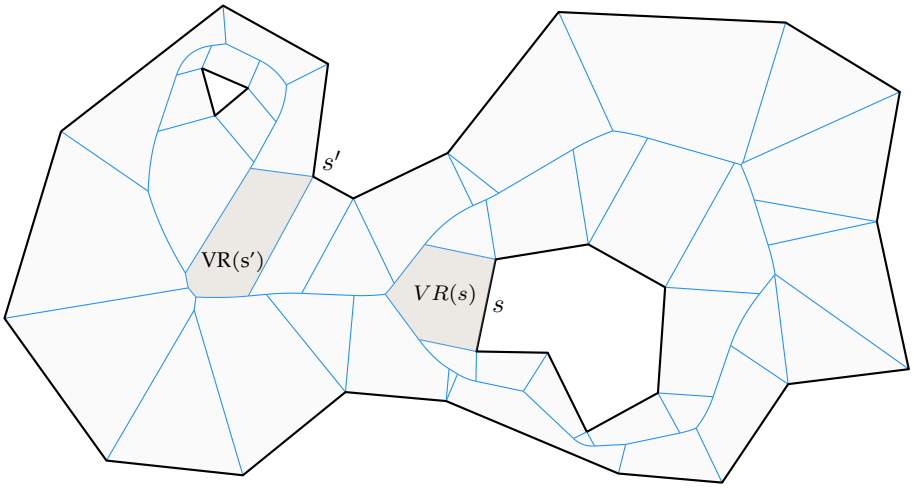


Figure 6.6: The Voronoi diagram $\mathrm{V}(P)$ of a polygon with holes P gives a nearest-neighbor cell decomposition of P. A few cells (Voronoi regions) are shaded in gray. The Voronoi diagram contains the medial axis $\mathrm{M}(P)$, cf. fig. 6.1.

6.3.4 Computing generalized Voronoi diagrams

Geometry. If the site set S consists of points only then the Voronoi polygons $\mathrm{VP}(s)$ form convex polygons. In particular, the Voronoi edges are all straight-line edges and the reason for this is that Voronoi edges are sections of the bisector between two point sites.

For the generalized Voronoi diagram we now have to also take into account the bisector between all combinations of points, straight-line segments and circular arcs. A point can be seen as a circle with zero radius. It turns out that all bisectors are formed by conic sections.

- The bisector between a straight line and a circle is a parabola.
- The bisector between two circles is either a hyperbola or an ellipse.
- The bisector between two straight lines is a straight line line.

The Voronoi nodes are located at the intersection of these bisectors. Directly computing the intersection of conic sections is not recommended from a numerical point of view. However, their location can also be computed by determining the points that are equidistant to three sites. This can actually be done by solving quadratic equations after the problem is transformed in an adequate way, see [15] for details.

Topology. The topology-oriented randomized incremental construction algorithm for Voronoi diagrams of points can be generalized to the generalized Voronoi diagram of points, straight-line segments and circular arcs. It can be proven that the expected runtime of the above algorithm is in $O(n \log n)$, see [14].

However, in order for that to work we require that when we insert a new site and therefore remove parts of the old Voronoi diagram then we only remove a tree structure, without cycles. This is actually not the case if we do not carefully prepare Voronoi edges: We need to split Voronoi edges at their apex (as conic sections). This means the following: If we consider the distance of points on the Voronoi edge to its two sites then this distance, in general, does not

change monotonically when sliding along the Voronoi edge. At the point on the Voronoi edge where this distance attains a minimum, we split the Voronoi edge and add a degree-2 Voronoi node. A detailed discussion on this procedure can be found in [15].

6.4 The grassfire model, offsetting and tool paths

Consider a point set S with n points in the plane and spread a grassfire at each of them. Assume the fire expands with equal speed in each direction, like circular waves that are emanated from the points in S. At certain locations in the plane the "fire waves" emanated from different points of S meet each other and the fire stops. What we receive is the picture in fig. 6.7: The points where the fires meet are exactly given by the Voronoi diagram $\mathrm{V}(S)$.

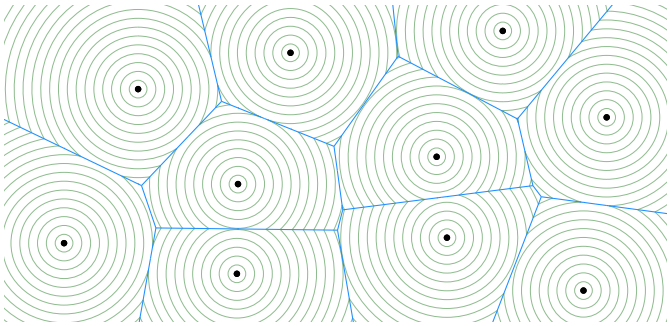


Figure 6.7: Grassfire model for a point set S. It sends out unit-speed offset waves from the sites in S and they interfere on $\mathrm{V}(S)$.

Hence, we could actually define $\mathrm{V}(S)$ also as the "interference pattern" of isotropic unit-speed waves wavefronts. We can apply the very same idea not only to point sets S. In fig. 6.8 we illustrate the grassfire wavefronts emanated by the points and straight-line segments of a polygon with holes P. The interference patterns give as $\mathrm{V}(P)$ again.

In CAD/CAM we call these grassfire wavefronts "offset curves" and in geographic information systems we call this operation "buffering". In NC-machining, wavefront curves are used for tool radius correction and the computation of tool paths, e.g., for NC milling machines or 3D plotters. Once the Voronoi diagram $\mathrm{V}(P)$ has been computed, an offset curve is computed in an easy, fast and numerically stable way by traversing the Voronoi diagram.

6.5 Straight skeletons

We call $\mathrm{V}(P)$ and $\mathrm{M}(P)$ a *skeleton* of P because it encodes topological features of P. In particular, for each hole of P we receive a cycle in the skeleton. They also encode certain geometric features, but a different skeleton would possibly encode different geometric features.

The grassfire model associated to Voronoi diagrams emanates a circular wavefront from non-convex vertices of a polygon with holes P, cf. fig. 6.8. What happens if we would emanate mitered offset curves instead? That is, the wavefront emanated at non-convex vertices would remain a sharp v-shape. So only the edges of P move inwards at unit speed and in parallel. Of course, the skeleton structure associated to this wavefront is different to $\mathrm{V}(P)$. It is called the *straight skeleton* $\mathrm{S}(P)$. Unlike the generalized Voronoi diagram, the straight skeleton consists of straight-line segments only, hence its name.

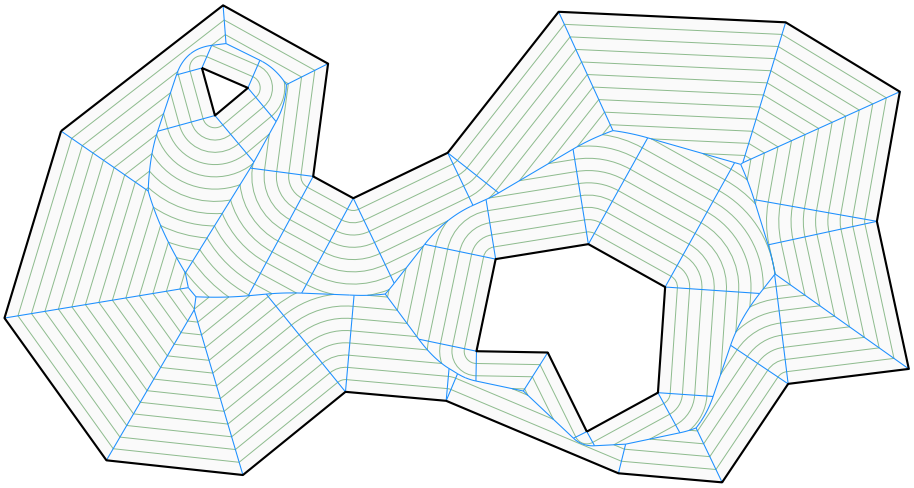


Figure 6.8: The offset curves given by grassfire wavefront curves. Its interference patterns create the Voronoi diagram.

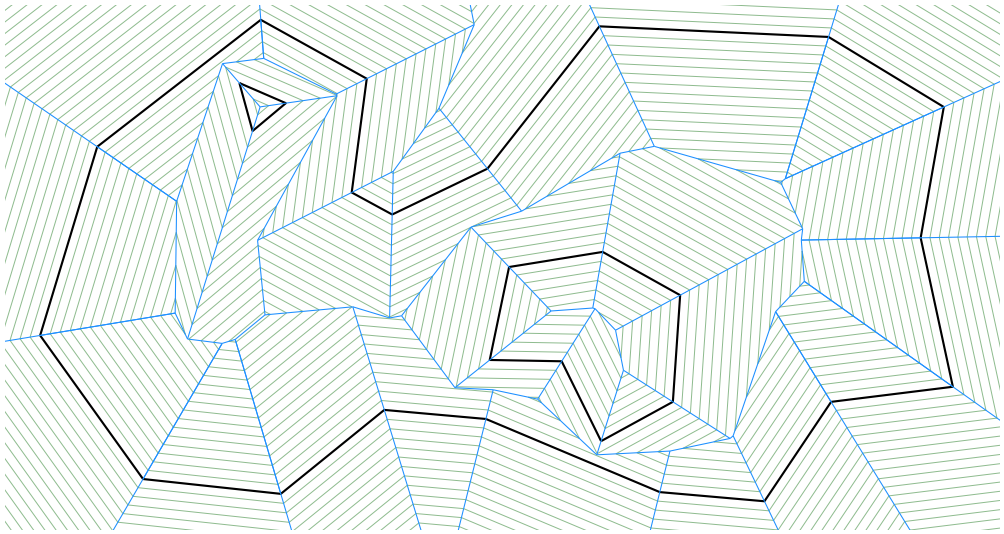


Figure 6.9: Mitered offset curves and the straight skeleton $\mathrm{S}(P)$ of a polygon with holes P.

Part III

Appendices

Appendix A

Computing cubic splines

In order to compute a natural cubic spline we could simply solve the linear equation system formed by the $4(n-1)$ conditions in section 3.4.2.

If we take a closer look, however, we see that this system can be significantly simplified. First of all we apply a parameter substitution that in away shift and stretch the p_i such that they are defined over $[0,1]$ rather than $[x_i, x_{i+1}]$. More precisely, let

$$q_i(x) = p_i\left(\frac{x + x_i}{x_{i+1} - x_i}\right)$$

Hence, $q_i(0) = p_i(x_i)$ and $q_i(1) = p_i(x_{i+1})$. If we know the q_i then we know the p_i, and vice versa.

The q_i are again polynomials

$$q_i(x) = a_i + b_i x + c_i x^2 + d_i x^3$$

Because $a_i = q_i(0) = p_i(x_i) = y_i$ we immediately know that $a_i = y_i$ and therefore

$$\begin{aligned} q_i(x) &= y_i + b_i x + c_i x^2 + d_i x^3 \\ q_i'(x) &= b_i + 2c_i x + 3d_i x^2 \\ q_i''(x) &= 2c_i + 6d_i x \end{aligned}$$

This leaves us with three unknowns b_i, c_i, d_i per polynomial. In the following we will use

$$\begin{aligned} q_i(1) &= y_i + b_i + c_i + d_i \\ q_i'(1) &= b_i + 2c_i + 3d_i \\ q_i'(0) &= b_i \end{aligned}$$

Note that $q_i(1) = y_{i+1}$ and $q_i'(1) = q_{i+1}'(0)$, which gives us the system

$$\begin{aligned} y_i + b_i + c_i + d_i &= y_{i+1} \\ b_i + 2c_i + 3d_i &= b_{i+1} \end{aligned}$$

We can solve for c_i and d_i and receive for $1 \leq i \leq n-2$

$$\begin{aligned} c_i &= 3(y_{i+1} - y_i) - 2b_i - b_{i+1} \\ d_i &= 2(y_i - y_{i+1}) + b_i + b_{i+1} \end{aligned}$$

If we know the b_i then we can compute the c_i and d_i and are done. So all unknowns that remain are $b_1, \ldots, b_{n-1}$. What we did not use so for is that the second derivatives have to match too, i.e., $q_i''(1) = q_{i+1}''(0)$. This gives for $1 \leq i \leq n-3$

$$\begin{aligned} 2c_i + 6d_i &= 2c_{i+1} \\ c_i + 3d_i &= c_{i+1} \\ 3(y_{i+1} - y_i) - 2b_i - b_{i+1} + 6(y_i - y_{i+1}) + 3b_i + 3b_{i+1} &= 3(y_{i+2} - y_{i+1}) - 2b_{i+1} - b_{i+2} \\ b_i + 4b_{i+1} + b_{i+2} &= 3(y_{i+2} - y_i) \end{aligned}$$

For a natural spline we also require $q_1''(0) = q_{n-1}''(1) = 0$. This gives $c_1 = 0$ and $2c_{n-1} + 6d_i = 0$ and results after some calculations in the equations

$$\begin{aligned} 2b_1 + b_2 &= 3(y_2 - y_1) \\ b_{n-2} + 2b_{n-1} &= 3(y_n - y_{n-1}) \end{aligned}$$

Altogether we end up with $n-1$ equations for $b_1, \ldots, b_{n-1}$:

$$\begin{pmatrix} 2 & 1 & & & & & \\ 1 & 4 & 1 & & & & \\ & 1 & 4 & 1 & & & \\ & & 1 & 4 & 1 & & \\ & & & & \ddots & & \\ & & & & 1 & 4 & 1 \\ & & & & & 1 & 2 \end{pmatrix} \cdot \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \\ \vdots \\ b_{n-2} \\ b_{n-1} \end{pmatrix} = \begin{pmatrix} 3(y_2 - y_1) \\ 3(y_3 - y_1) \\ 3(y_4 - y_2) \\ 3(y_5 - y_3) \\ \vdots \\ 3(y_n - y_{n-2}) \\ 3(y_n - y_{n-1}) \end{pmatrix}$$

The coefficient matrix is a so-called *tridiagonal* matrix. Such equation systems can be actually solved in $O(n)$ time by dedicated algorithms. See [19] for details. Solving the linear system give us the b_i, which then yield c_i and d_i, after which we know the polynomials q_i. These can then be transformed back to the p_i if desired.

Bibliography

[1] *754-1985 – IEEE Standard for Binary Floating-Point Arithmetic*. Note: Standard 754–1985. New York: Institute of Electrical and Electronics Engineers, 1985. URL: https://ieeexplore.ieee.org/document/30711.

[2] C. Bradford Barber and Hannu Huhdanpaa. *Qhull*. URL: http://www.qhull.org/.

[3] C. Bradford Barber et al. "The Quickhull Algorithm for Convex Hulls." In: *ACM Transactions of Mathematical Software* 22.4 (1996-12), pp. 469–483. DOI: 10.1145/235815.235821.

[4] Mark de Berg et al. *Computational Geometry: Algorithms and Applications*. 3rd ed. Santa Clara, CA, USA: Springer-Verlag TELOS, 2008. ISBN: 3540779736, 9783540779735.

[5] *Clang Developers: Fixed Point Arithmetic Proposal*. 2018. URL: http://clang-developers.42468.n3.nabble.com/Fixed-Point-Arithmetic-Proposal-td4060468.html (visited on 04/25/2020).

[6] Satyan L. Devadoss and Joseph O'Rourke. *Discrete and Computational Geometry*. Princeton University Press, 2011. ISBN: 9781400838981.

[7] *DSP-C – An extension to ISO/IEC IS 9899:1990*. Standard. ACE Associated Compiler Experts bv, 2008-02. URL: http://www.ace.nl/sites/default/files/paper-dsp-c.pdf (visited on 04/25/2020).

[8] Herbert Edelsbrunner and John Harer. *Computational Topology - an Introduction*. American Mathematical Society, 2010. ISBN: 978-0-8218-4925-5.

[9] *GCC Wiki: Fixed-Point Arithmetic Support*. 2007. URL: https://gcc.gnu.org/wiki/FixedPointArithmetic (visited on 04/25/2020).

[10] David Goldberg. "What Every Computer Scientist Should Know About Floating-point Arithmetic." In: *ACM Comput. Surv.* 23.1 (1991-03), pp. 5–48. ISSN: 0360-0300. DOI: 10.1145/103162.103163.

[11] Peter M. Gruber. *Convex and Discrete Geometry*. Springer-Verlag Berlin Heidelberg, 2007. ISBN: 978-3-540-71132-2.

[12] Martin Held. *Computational Geometry*. 2018-09. URL: https://www.cosy.sbg.ac.at/~held/teaching/compgeo/cg_study.pdf.

[13] Martin Held. *Computational Geometry*. lecture notes. SS 2018. URL: https://www.cosy.sbg.ac.at/~held/teaching/compgeo/comp_geo.html.

[14] Martin Held and Stefan Huber. "Topology-Oriented Incremental Computation of Voronoi Diagrams of Circular Arcs and Straight-Line Segments." In: *Computer Aided Design* 41.5 (2009-05), pp. 327–338. DOI: 10.1016/j.cad.2008.08.004.

[15] Stefan Huber. "Computation of Voronoi Diagrams of Circular Arcs and Straight Lines." MA thesis. Universität Salzburg, Austria, 2008-02.

[16] *Programming languages – C – extensions to support embedded processors*. Standard ISO/IEC TR 18037:2008. International Organization for Standardization, 2008-06. URL: https://www.iso.org/standard/51126.html.

[17] Johann Linhart. *Numerische Mathematik*. WS 2004/05. URL: https://www.uni-salzburg.at/fileadmin/multimedia/Mathematik/documents/Num.Mathematik_WS2004.pdf.

[18] Cleve B. Moler. "A Tale of Two Numbers." In: *SIAM News* 28 (1995-01). Also in *MATLAB News and Notes*, Winter 1995, 10–12, pp. 1, 16. ISSN: 0036-1437.

[19] William H. Press et al. *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. 3rd ed. New York, NY, USA: Cambridge University Press, 2007. ISBN: 0521880688, 9780521880688.

[20] Sun One Studio. *Numerical Computation Guide*. 2003.

[21] Kokichi Sugihara and Masao Iri. "Construction of the Voronoi Diagram for 'One Million' Generators in Single-Precision Arithmetic." In: *Proceedings of the IEEE* 80.9 (1992-09), pp. 1471–1484. DOI: 10.1109/5.163412.

[22] Alfred North Whitehead. *An Introduction to Mathematics*. Oxford University Press, 1958. ISBN: 9780195002119.

Index