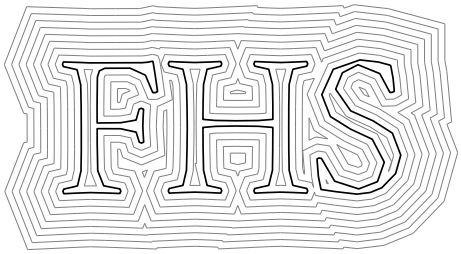
Algorithms in industry

Selected topics in algorithms, optimization, numerics and geometry

Stefan Huber

Dpt. for Information Technologies and Digitalisation
Salzburg University of Applied Sciences

Summer 2026



Contents

Abbreviations

ABI	application binary interface
ACPI	advanced configuration and power interface
AI	artificial intelligence
APCI-PM	advanced power management
AVX	advanced vector extensions
BCD	binary coded decimal
BFS	breadth-first search
ccw	counter-clockwise
CPU	central processing unit
cw	clockwise
DAG	directed acyclic graph
DCEL	doubly-connected edge list
DFS	depth-first search
DSP	digital signal processor
EGC	exact geometric computation
EMST	Euclidean minimum spanning tree
ETSP	Euclidean traveling salesperson problem
FFT	fast fourier transform
FIFO	first in, first out
FPU	floating point unit
GPU	graphics processing unit
HPET	high precision event timer
ILP	integer linear programming
KKT	Karush-Kuhn-Tucker
LIFO	last in, first out
LLC	last level cache
LoRA	Low Rank Adapation
LRU	least recently used
MAT	medial axis transform

MDP	Markov decision process
MIPS	million instructions per second
ML	machine learning
MLP	multi-layer perceptron
MMX	multi-media extensions
MPI	message passing interface
MST	minimum spanning tree
NTP	network time protocol
OS	operating system
PCA	principal component analysis
POSIX	portable operating system interface
PPO	proximal policy optimization
PSLG	planar straight-line graph
PTAS	polynomial time approximation scheme
RAM	random access machine (computing model)
RAM	random access memory (memory type)
RL	reinforcement learning
RSS	resident set size
SAT	boolean satisfiability problem
SGD	stochastic gradient descent
SIMD	single instruction, multiple data
SSE	streaming SIMD extensions
SVM	support vector machine
TAI	international atomic time
TD	temporal difference
TSC	time stamp counter
TSP	traveling salesperson problem
UTC	coordinated universal time
VCR	video cassette recorder

Symbols

A^+	(Moore-Penrose) pseudoinverse of matrix A
A^{-1}	Inverse of matrix A
C_n	Cycle graph with n vertices
$E(X)$	Expected value of random variable X
$H_f(x)$	Hessian matrix of function f at point x
$J_f(x)$	Jacobian matrix of function f at point x
K_n	Complete graph with n vertices
$K_{m,n}$	Complete bipartite graph with m and n vertices
$O(f)$	Big-O notation for function f, see table 1.1
$Q_b^{m,n}$	Fixed-point numbers with m integral and n fractional digits to basis b
$Qm.n$	Q format: numbers with m integral and n fractional bits
Qn	Q format: numbers with n fractional bits
S_n	Star graph with n vertices
W_n	Wheel graph with n vertices
$\Omega(f)$	Big-Ω notation for function f, see table 1.1
$\Theta(f)$	Big-Θ notation for function f, see table 1.1
$\alpha(n)$	Inverse Ackermann function
$\approx$	Approximately equal
$\operatorname{bd} A$	Boundary of set A
$\binom{n}{k}$	Binomial coefficient of n over k
$\operatorname{conv} A$	Convex hull of set A
gcd	Greatest common divisor
$\operatorname{im} A$	Image of linear function given by matrix A
$\operatorname{int} A$	Interior of set A
κ_{abs}	Absolute condition number
κ_{rel}	Relative condition number
C^∞	Smooth functions
C^k	k-times continuously differentiable functions
$\mathbb{N}$	Natural numbers, i.e., positive integers
$\mathbb{R}$	Real numbers
$\mathbb{Z}$	Integer numbers
$\nabla f(x)$	Gradient of function f at point x
$\omega(f)$	Little-ω notation for function f, see table 1.1
$\overline{pq}$	Line segment between points p and q
$\operatorname{rd}_{s,b}(z)$	Rounding real number z to mantissa length s and basis b
$\|\|A\|\|$	Norm of matrix A
$\|\|A\|\|_p$	p-norm of matrix A
$\|\|f\|\|$	Norm of function f
$\|\|v\|\|$	Norm of vector v

$\|v\|_p$	p-norm of vector v
$f \sim g$	Asymptotic equivalence of functions f and g, see table 1.1
$f^{(n)}$	n-th derivative of function f
$o(f)$	Little-o notation for function f, see table 1.1
$u \rightsquigarrow v$	Walk, path or tour from vertex u to v in a graph
$v^\intercal$	Transpose of vector v
$x \ll y$	x is much less than y
$x \otimes y$	Outer product of vectors x and y
$x \sim p$	Sample x from a probability distribution p
$\|A\|$	Cardinality of set A
$\|x\|$	Length of string x

Introduction

A technocratic introduction. At the Department for Information Technologies and Digitalisation we offer a handful of of Master studies, and two of them are *Industrial Informatics and Robotics* (IIR) and *Cyber Security* (CSY). These lectures notes serve two courses in these curricula: The course *Selected Algorithms and Optimization* is shared by IIR and CSY students and the course *Numerics and Industrial Algorithms* is for IIR students only. The aim of both courses is to bridge theoretical computer science with practical engineering in the respective domains, which is why we always attempt to combine theoretical concepts with the practical instantiation from an engineer's point of view.

A personal introduction. My personal career was a dual one between academics and industry, as a student of computer science and mathematics, a PhD in algorithms and geometry, a PostDoc in computational topology and in cooperate R&D in the industrial automation domain. Especially during my time in industry I experienced two things: First, the academic education alone is insufficient to become an effective engineer in industry and the personal experience gained as a developer from a teenager's age on was extremely important on daily basis. Secondly, the industrial engineering practice would benefit a lot from the theoretical concepts in academy. In other words, at least when it comes to computer science, I feel like academy and industry should grow closer together for the sake of both.

Since the two master courses mentioned above allow for a lot of freedom in the selection of topics, I took the opportunity to pick topics I found of great value during my own time in industry and I attempted to integrate them into a common academic story line, while still respecting the needs of both curricula.

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 and the like. On the other hand, physics or chemistry are natural sciences. They investigate phenomena that we observe in our reality.[1]

The typical subfields of mathematics used in computer science are discrete, like graphs, abstract algebra, or number theory. 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, describing phenomena in mechanics, electromagnetism, thermodynamics and so on. This is why we tend to find integers in computer science but reals[2] in physics.

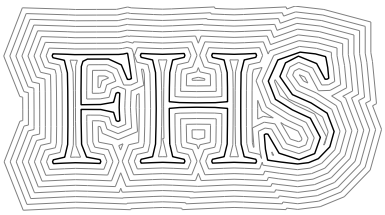
[1] The natural sciences use mathematics for modeling and as a language on one hand and as a tool to draw conclusions and extrapolations on the other hand. The classical engineering sciences sit on top of the natural sciences and leverage insights on our real world for technical mechanisms and constructions. Computer science is in some sense different to the classical engineering sciences as it builds upon mathematics directly. Brooks reflects on the nature of programming in his seminal work in the section called "The Joys of the Craft", when he refers to programming as "slightly removed from pure thought-stuff" and "building castles in the air, from air, creating by exertion of the imagination" [39, p. 7], which highlights the mathematical nature of programming.

[2] We also find complex numbers, which are just pairs of reals in a sense. More precisely, we find discrete numbers in computer science and numbers from a continuum in physics.

In the first chapters of these lecture notes, we build up basics in the discrete world of computer science. But then we more and more enter topics that effectively arise from the application of computer science to a continuous mathematical world, as they emerge from problems of the natural sciences, in particular in physics, and engineering. More precisely, a great deal of 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 in an exact manner – without numerical loss – by a (digital) computer, but dealing with real numbers is fundamentally impossible for a computer.[3] We are left with approximations of real numbers and each operation introduces loss of precision. The field of numerical analysis investigates the challenges that arise from this fact.[4]

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.[5] 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.



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?[6] How do we compute them in a computationally 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. These lecture notes are split into three parts: Basics, numerical programming and computational geometry.

- **Basics.** The first part deals with basic concepts from computer science and mathematics, which we need for the later parts. This includes algorithm analysis, performance optimization, graph theory, and convexity.

[3] 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. In particular, there are more real numbers than computer programs or Turing machines. So in this sense we really leave the natural habitat of computer science.

[4] Another yet very different example where digital computers face a challenge, but nature does not, is randomness. As far as we know true randomness – the absent of full determinism, at least the lack of full predictability – occurs in natural processes, like thermal motion of molecules or radioactive decay. But it does not occur in a digital computer in the sense of a deterministic Turing machine. To have true randomness in computers, we need additional hardware that uses natural randomness. This observation is essential for cyber security, when an element of unpredictability is required, e.g., in key generation.

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

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

- **Numerical mathematics.** The second part deals with the foundation for numerical computations and numerical algorithms within mathematics, in particular linear algebra and calculus.

- **Computational geometry.** The last part deals with algorithms and data structures in a geometric context, like basics on geometric computations, convex hulls, range searching, and spatial structures like Voronoi diagrams and Delaunay triangulations.

There is large body of literature covering the basics covered in the chapters of part one. Here is a selection of them:

- A classic book on algorithm theory is Robert Sedgewick's book on algorithms. In the past couple of decades it appeared in many forms and also for different programming languages (although the language is not of primary concern). One of the later versions is this:

  [45] Robert Sedgewick and Kevin Wayne. *Algorithms in C++*. 4th ed. Addison-Wesley, 2011. ISBN: 978-0-321-87758-3

- A classic book on algorithms is by Thomas Cormen, Charles Leiserson, Ronald Rivest and Clifford Stein. It is often referred to as "CLRS" after the initials of the authors.

  [11] Thomas H. Cormen et al. *Introduction to Algorithms*. 4th ed. The MIT Press, 2022. ISBN: 9780262046305

- A standard textbook concerning computer architecture is by David Patterson and John Hennessy. It explains the inner workings of computers and processors, based on which we can understand the performance of algorithms on a deeper level.

  [26] John L. Hennessy and David A. Patterson. *Computer Architecture*. 5th ed. Morgan Kaufmann, 2012. ISBN: 978-0-12-383872-8

- Two books that cover aspects of writing efficient programs are the following:

  [41] Alexander Pikus. *The Art of Writing Efficient Programs*. Packt Publishing, 2021, p. 464. ISBN: 978-1800208117

  [8] Randal E. Bryant and David R. O'Hallaron. *Computer Systems: A Programmer's Perspective*. 3rd ed. Pearson, 2018, p. 1128. ISBN: 978-0134092669

- A comprehensive book on discrete and convex geometry is by Peter Gruber:

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

The following literature is recommended for further reading concerning topics in part two:

- 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.

  [37] 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.)

  [42] 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 [21] that discusses many details of floating-point units.

  [46] 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.

  [6] 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.

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

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

  [19] 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.

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

Acknowledgments. I am thankful for the following people to have provided comprehensive feedback on the lecture notes: Simon Schindler, Martin Mayr.

Part I

Basics

Chapter 1

Algorithm analysis

1.1 Algorithms and models of computing

1.1.1 What is an algorithm?

The term "algorithm" has its origin in name of the arabic mathematician[1] "al-Khwarizmi", who lived about 1200 years ago and is considered as the father of algebra. The term "algebra" stems from "al-ğabr" and has its origin in a very influential book of al-Khwarizmi, in which he describes his method of solving quadratic equations by sequences of transformations of equations. This nature of a step-by-step procedure to solve given problems is precisely what we mean by *algorithm*.[2]

Historically, the first example of an algorithm is probably Euclid's algorithm for computing the *greatest common divisor* $\gcd(a, b)$ of two natural numbers a and b, which is defined as the largest natural number that divides both, a and b. It has many applications, like cryptography, simplification of fractions and offline computations of real-time schedules. The algorithm is given in algorithm 1.

Algorithm 1 Euclid's algorithm

```
procedure EUCLID(a, b)
    while b ≠ 0 do
        t ← b
        b ← a mod b
        a ← t
    return a
```

On this example we can illustrate two basic questions always to be asked about algorithms: First, is it correct? Second, what resources does it take?

1.1.2 Correctness and loop invariants

In order to speak of an algorithm, we need to specify what problem P the algorithm is supposed to solve. The problem P must be precisely stated in the sense that for every input x it must be

[1] He was a polymath contributing in various disciplines, including geography and astronomy.

[2] As a personal remark of the author, in a certain sense, "algorithm" and "algebra" do not only share this step-by-step paradigm, but also in the following way: many problems only become tangible to computers after they have been algebraized. For instance, to do geometry on computers, we need algebraized geometry, e.g., by using Cartesian coordinates or, more generally, algebraic geometry. In a certain sense, when Galileo said that "book of nature is written in the language of mathematics", it seems that the language of computer science is algebra.

clear what the output y should be, see fig. 1.1. For some problems the output may not be unique, but in any case it must be precisely defined when an output y is considered to solve the problem P.[3] For instance, if the problem is "sorting numbers" then the input x is a finite sequence of numbers and the output y is correct if and only if a permutation of x is returned such that the numbers in x are sorted. That is, we have a formalization of the problem P.

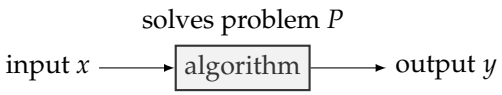


Figure 1.1: An algorithm turns an input x into an output y according to a problem P.

If the problem is not precisely stated then we have troubles even talking about algorithms for the problem, let alone correctness. Instead, for these type of problems we rather talk about heuristics.[4]

Typically, to argue for the correctness of an algorithm, we require some insight in the problem structure. For instance, we know that $\gcd(a, b) = \gcd(b, a)$. More relevant, however, is the insight that the result of $\gcd(a, b)$ does not change if we remove multiples of b from a. In fact, we could reduce a to $a \bmod b$ without changing the outcome of gcd. We can summarize this as follows:

$$\gcd(a, b) = \gcd(b, a \bmod b) \tag{1.1}$$

And this is all what Euclid's algorithm does: It repeatedly replaces a, b by $b, a \bmod b$. In Python, we can rephrase algorithm 1 by literally only looping over eq. (1.1), even getting rid of the temporary variable t:

```
1 def gcd(a, b):
2     """Returns the greatest common divisor of ‘a‘ and ‘b‘."""
3     while b != 0:
4         a, b = b, a % b
5     return a
```

A powerful technique to prove the correctness of algorithms is to consider *loop invariants*: A loop invariant is a property that holds before and after a loop iteration, i.e., the loop does not change this property. In case of algorithm 1, the $\gcd(a, b)$ is an invariant, since each iteration of the loop replaces a, b by $b, a \bmod b$.

Another loop invariant is that $a > b$, except possibly for the first iteration. If in the first iteration, $a = b$ then a, b turns into $b, 0$, and we are done with correctly returning b. If in the first iteration $a < b$ then the first loop iteration effectively swaps a and b.

To argue for the correctness of algorithm 1, we have to show two things: (i) the algorithm reaches the return statement and so finishes in finite time and (ii) it actually returns the correct result. The first part can be seen by observing that a and b become smaller in each iteration and hence $b > 0$ cannot hold indefinitely. (Recall the second loop invariant from above.) How fast they become slower tells us something about the speed of the algorithm, but this is to be discussed later. The second part can be shown by using the first loop invariant from above.

[3] For some problems it is even the very essence that the output is not unique, like yielding a random number according to some distribution. Violations of this required randomness property can be fatal for cryptographic applications.

[4] Watch out when somebody talks about "machine learning algorithms" and what is meant by it. Formalizing a problem can be quite challenging. For instance, if we investigate algorithms for random number generation – how do we precisely define the problem to capture our intuition of what we mean? An algorithm that returns 42 should be excluded from the problem definition, but when is an output y considered to be a random number?

Note that $\gcd(x, 0) = x$. Since every iteration of the loop keeps the gcd invariant and we end up with $b = 0$, a contains the gcd.[5]

Let us do a different example, concerning sorting numbers. The problem is as follows: We are given a sequence of numbers, $a_1, \ldots, a_n$, and we want to sort them in ascending order. A simple algorithm for this problem is *insertion sort*, as given in algorithm 2.

Algorithm 2 In-place insertion sort

```
1: procedure INSERTION_SORT(a_1, ..., a_n)
2:     i ← 2
3:     while i ≤ n do
4:         j ← i
5:         while j > 1 and a_{j-1} > a_j do
6:             swap a_j and a_{j-1}
7:             j ← j − 1
8:         i ← i + 1
```

We remark that algorithm 2 is an *in-place* algorithm, which means that it does not return a sorted copy of the input sequence, but sorts the input sequence itself. We again ask whether this algorithm is correct. First of all, the algorithm terminates in finite time. In order to argue that the algorithm actually sorts the sequence, we again can phrase the following loop invariant concerning the for-loop: The sequence $a_1, \ldots, a_{i-1}$ is sorted. After i reaches n, we therefore have that $a_1, \ldots, a_n$ is sorted. To check the correctness of the loop invariant note that the inner loop puts a_i into the right position of the sorted sequence $a_1, \ldots, a_{i-1}$, effectively leaving a sorted sequence $a_1, \ldots, a_i$.

A note on generative AI. In the 2020s there was a rapid development of powerful generative pretrained large language models, such as OpenAI's GPT models first launched in November 2022. Such models are the engines behind code generation assistants, like GitHub's Copilot, first release in October 2021. Even from a reluctant viewpoint, it seems evident that they lead to far-reaching consequences on the programming practices. In particular, one might speculate that the ability to verify the correctness of generated code becomes of increasing importance for software engineers.

Note that the underlying technique of those systems is based on next-token prediction based on massive machine learning models, which is inherently of statistical nature. Note that there are methods of artificial intelligence (AI) that provide formal guarantees of correctness, however those are logic-based, which face scaling issues due to their computational complexity. In any case, theoretical computer sciences provides fundamental limits of code generation and verification, like the halting problem and the P-versus-NP problem.[6]

[5] This argument may sound a bit like cheating by hiding the argument behind definition of $\gcd(x, 0)$. A more verbose argument would be to unroll the loop and give the a's and b's indices for each loop iteration, like $a_1, b_1, a_2, b_2, \ldots$. Then the loop invariant says $\gcd(a_1, b_1) = \gcd(a_2, b_2) = \ldots = \gcd(a_n, 0) = a_n$, which is finally returned. Behind the "..." we actually hide a proof by induction, which is another typical proving tool in algorithm theory, especially when it comes to recursions.

[6] As a personal remark I would like to add that there is a certain chance that history of software engineering repeats given the following quote of Dijkstra [16] in 2000: "The required techniques of effective reasoning are pretty formal, but as long as programming is done by people that don't master them, the software crisis will remain with us and will be considered an incurable disease. And you know what incurable diseases do: they invite the quacks and charlatans in, who in this case take the form of Software Engineering gurus."

1.1.3 Resources and models of computing

Algorithm versus implementation. Our first question on algorithms concerned the correctness, the second was on the resources taken by algorithms. In algorithm theory, the two resources considered are time and space.[7] A first difficulty resides in the question what we actually mean by time and space.

For instance, we could take algorithm 2 and implement it, in some programming language on some computer, and run it. We could then measure the physical time taken, i.e., the *wall clock time*. However, this time does not depend solely on the algorithm, but also on the compiler, the computer, the operating system, caching effects, memory fragmentation, and many more. Same goes for space, by which mean the memory used by the algorithm. It is not even clear how to technically measure the memory consumption of the procedure itself, leaving alone that this again depends on the compiler, memory fragmentation, operating system, processor architecture and more.

Computing models. Therefore, if we want to compare algorithms per se – and not their implementations or systems on which they are run – then we need a more abstract notion of time and space taken by algorithms. To this end, we take algorithm 2 and think of it being run on a theoretical model of a computer that provides certain operations at *unit cost*, like basic arithmetic operations, comparison of numbers, access of a memory cell, and so on. We then count the number of operations taken to measure *time* and the number of memory cells taken to measure *space*. We call this the *model of computing*.

The time and space taken by an algorithm of course depends on this model. From theoretical computer science, the model of a *Turing machine* is a common model. This model is important to investigate fundamental questions of theoretical computer science, like the so-called *halting problem* or questions concerning *computability*.

Using the Turing machine for our purposes would be very tedious, since even simple problems, like adding two numbers, quickly becomes very involved. Instead we use more convenient models like the so-called *random access machine (RAM)* model, which we can compare to an assembly like language. In this model we have memory organized in cells that can hold integers and every assignment, comparison, logical and arithmetic operation, the access of a single memory cell or control flow instructions are all at unit cost. This makes it much easier to analyze the time and memory taken by algorithms.

For some application domains, the Turing machine model can actually not be applied. For instance, when we deal with real numbers like in numerical mathematics or computational geometry. Here the *Real RAM* model is used, which is essentially the RAM model in which a memory cell can store a real number.

Analyzing insertion sort. Let us demonstrate the RAM model for insertion sort as in algorithm 2. The space taken is 2 for the two variables i and j. An analysis of the time spent, however, is a bit more involved. The outer loop is iterated exactly $\sum_{i=2}^{n} 1 = (n-1)$ times. The inner loop is iterated up to $\sum_{i=2}^{n} \sum_{j=2}^{i} 1 = \frac{n(n-1)}{2}$ times. But if $a_{j-1} \leq a_j$ holds right away for all j initialized to i then the inner loop is iterated zero times, which is the case if and only if $a_1, \ldots, a_n$ was already sorted. So the time spent by insertion sort depends on the input data – not only its size, but also its content. For our analysis of insertion sort, we therefore distinguish between the *best case* and the *worst case*.

[7] Another resource would be "energy", which receives more attention in recent times. Other measures of qualities of algorithms concern numerical properties, as we will see in section 5.4.

We can summarize our analysis of the time spent by insertion sort in the RAM model as below, where the annotations refer to the lines in algorithm 2 and the inner sum is added for the worst case and discarded for the best case:

$$1 + 4 + \sum_{i=2}^{n} \Bigg(4 + 2 + 7 + \overbrace{\sum_{j=2}^{i} \Big(7 + 2 + 1 + 1 \Big)}^{\text{for worst case}} + 1 + 1 \Bigg) \tag{1.2}$$

l2, l3, l4, l5, l6, l7, l8

In eq. (1.2) each loop contributes at three locations of the loop: (i) the condition has to be checked even if the loop is then not entered, (ii) the condition has to be checked in each iteration, and (iii) we have to jump back to the beginning of the loop. We account for each memory access, so the cost of l2 is 1 and the cost of l4 is 2 and the cost of evaluating $a_{j-1} > a_j$ is 3. The cost of l3 in the first location is 4, because we have to evaluate $i \leq n$ at the cost of 3 and then jump to the according line at the cost of 1. By the same argument we have a cost of 4 at the second occurrence of l3. Likewise for l4 we have to evaluate the two inequalities, the logical "and" and perform a jump to the according line, so we have a cost of 7.

However, the exact details depend on how we exactly define the RAM model. For instance, we take the costs of l6 to be 2 by assuming our RAM model has a builtin swap operation so all it takes is the access of the two memory cells involved. Furthermore, we assume that l7 and l8 are done at unit costs by assuming our RAM model can increment and decrement directly.[8]

We can now simplify eq. (1.2) as follows to obtain the time spent on the RAM model:

$$5 + \sum_{i=2}^{n} \left(15 + \overbrace{\sum_{j=2}^{i} 11}^{\text{for worst case}} \right) = 5 + 15(n-1) + \overbrace{11 \sum_{i=2}^{n} (i-1)}^{\text{for worst case}} = 15n - 10 + \overbrace{11 \frac{n(n-1)}{2}}^{\text{for worst case}}$$

$$= \begin{cases} 15n - 10 & \text{for the best case} \\ \frac{11}{2} n^2 + \frac{19}{2} n - 10 & \text{for the worst case} \end{cases} \tag{1.3}$$

Best, worst and average case. As we have already seen, for some algorithms the resources taken depends on the input data and we distinguish between the best and the worst case. If we want to say something about the "typical" case then we could analyze the *average case*. More precisely, we would think of the time (or any other resource) as a random variable whose expectation value we want to analyze.

However, in order to do so, we have to define a probability distribution of the input data. Here we can incorporate a priori knowledge we have on the input data occurring in our application. And if we do not have such knowledge then we resort to the uniform distribution, given that it exists.[9]

[8] We refrain from a complete definition of computing model as it would take as a whole chapter on its own and the value it would add for our discussions to follow is negligible. However, Donald Knuth did invent a complete computing model – or rather a hypothetical computer – in his seminal work *The Art of Computer Programming* and he called it the MIX, which later was modernized to become the MMIX, see [36, sec. 1.3].

[9] For instance, the uniform distribution exists if we draw the input from a finite domain, like it is the case if our a_i in

Randomization. For many algorithms the average case is better than the wort case. Still, in practice the worst case may occur naturally, simply because input is not drawn randomly. For instance, quicksort with the pivot chosen as the first element displays a quadratic growth of time in the worst case if the input sequence is sorted in reverse order. For many such algorithms we can use a technique called *randomization*, in which we randomly permute the input. This way the worst case becomes unlikely by chance and the analysis of the average case is often simplified.

A note on security. If we use algorithms in security-critical applications that display a difference in execute time depending on the input data then this can potentially be exploited to reveal information about the input data. Those attacks are called *timing attacks* and we speak of a timing *side channel* over which information is revealed. From the perspective of algorithm design we attempt to make the time spent independent of the data. In cryptography, we speak of *constant-time algorithm*. We cannot hope for sorting algorithms to be of constant time in the sense that the time spent is independent of the length n of the input sequence, but we can ask for every execution taking the same time for every input permutation, e.g., by applying randomization as mentioned above.

Note that it is insufficient to have the same execution time on an abstract computing model, like RAM, but we need to have this property also on actual computing hardware, as timing attacks were also demonstrated against caching and branch prediction mechanisms of processors, like the Spectre and Meltdown attacks. Hence, the constant-time property does not only depend on the algorithm (on an abstract computing model) but on actual implementations (on concrete computing systems).

1.2 Growth and complexity

1.2.1 Classifying functions by growth

In the previous section we performed algorithm analysis on abstract computing models to get rid of implementation effects. We already saw that the exact time spent on a those models still depends on details of the computing model. This is undesireable for a couple of reasons:

First, when we compare algorithms then typically we are interested in the "dominating behavior" of the growth of time as the input size grows. In terms of eq. (1.3), our conclusion is that insertion sort takes linear time in the best case and quadratic time in the worst case. The coefficients 15 and $\frac{11}{2}$ in eq. (1.3) are irrelevant for this judgment and depend on details of the RAM model anyhow.[10]

Secondly, if we want to say something about the complexity of the problem per se – like what are the theoretical bounds of how fast sorting can be done – then arguing on the level of detail of computing models is becoming too complicated. However, from a scientific point of view we would like to be able to say how fast sorting can be done and whether a specific algorithm matches this bound is therefore considered *optimal*. As another example, in security research

insertion sort are integers of finite size. In general, the uniform distribution exists over domains of finite measure in the sense of measure theory. However, even if the uniform distribution exists, the analysis can quickly become involved. For instance, assume we would like compute the convex hull of n points distributed in a square or, more generally, in a convex polygon with r vertices, as we will do in chapter 9. The time spent by some algorithms may depend on the size of the output. What is the expected number of points of the convex hull in this case? There is a seminal work by Rènyi and Sulanke [43] using tools from integral geometry to answer this question.

[10] Only if n is relatively small and the constants are large then it could happen that we prefer quadratic over linear, for instance, we prefer n^2 over $1000n$ if $n < 1000$, on this specific computing model.

we would like to prove that certain computational problems are impossible to solve efficiently when we erect cryptographic methods on them.

Asymptotic analysis. What we need is a mathematical notion of the "order of growth" of functions $\mathbb{N} \to \mathbb{R}$, like $15n - 10$ or $7n \log n - n$ as functions in n, which display the growth of a resource spent by an algorithm by the size n of the input. The notion of *asymptotic analysis* provides us with a tool to compare how $f(n)$ and $g(n)$ evolve as n becomes larger and larger, such that the dominating terms actually become dominant. Formally, the *asymptotic equivalence* of two functions $f, g \colon \mathbb{N} \to \mathbb{R}$ is denoted by $f \sim g$ and defined as

$$f \sim g \quad \text{iff} \quad \lim_{n\to\infty} \frac{f(n)}{g(n)} = 1. \tag{1.4}$$

This is mathematically an equivalence relation, which means that the following three properties hold: Reflexivity as $f \sim f$, symmetry as $f \sim g$ implies $g \sim f$, and transitivity as $f \sim g$ and $g \sim h$ implies $f \sim h$. As a result, asymptotic equivalence can be used to classify functions. Also it allows us to write entire chains of equivalences like $f \sim g \sim h$, meaning that they are all equivalent by the law of transitivity.

Concerning the example in eq. (1.3), we can say for the time $T(n)$ spent by insertion sort it holds that

$$T(n) \sim \begin{cases} 15n & \text{for the best case} \\ \frac{11}{2}n^2 & \text{for the worst case} \end{cases} \tag{1.5}$$

Other examples are $n^2 \sim n^2 + n$, $n \sim n + \log n$ and $n \sim n + \sqrt{n}$. However, $2n + 1 \not\sim n$, $n \not\sim n \log n$ and $e^n \not\sim n^x$ for any $x \in \mathbb{R}$. An important yet less trivial example is

$$n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n. \tag{1.6}$$

It is a result from Stirling's formula. Since in algorithm theory we sometimes consider the number $n!$ of permutations of n numbers, like when we analyze the problem of sorting, this formula is of importance. In particular, from this formula it follows that

$$\log n! \sim \log \sqrt{2\pi n} + n \log n - n \log e \sim n \log n \tag{1.7}$$

In order to show this it helps to know that if $f \sim g$, $a \sim b$ and $x \in \mathbb{R}$ then

$$f^x \sim g^x \quad \text{and} \quad f \cdot a \sim g \cdot b \quad \text{and} \quad \frac{f}{a} \sim \frac{g}{b} \quad \text{and} \quad \log f \sim \log g. \tag{1.8}$$

(We only need to take care that a and b are not zero if we divide by them and the right hand sides do not become zero, which is the case for $\log g$ when g goes to 1.) Those rules basically say that we can exchange $\sim$ with the operations above.

Big-O notation and order of growth. The asymptotic equivalence is still a very fine-grained notion. Often we need something more coarse to compare algorithms, and even more so to investigate the complexity of problems. As we already mentioned above, we would also like to become ignorant of the constant factors in our analysis, such that $2n \log n$ and $n \log n$ are considered equivalent, although they are not asymptotically equivalent. Hence, in eq. (1.4) we do not ask for $f(n)/g(n)$ to tend to 1, but any constant. This leads to the so-called *big-O notation*,

where the "O" stands for "order" in the sense of "order of"[11] and was popularized in computer science by Knuth [34].

Like we can compare two real numbers by $<, \leq, =, \geq$ and $>$, we introduce for the order of growth of functions the notations o, O, Θ, Ω and ω. Let $f, g\colon \mathbb{N} \to \mathbb{R}$ then we mean by $f \in O(g)$ that f grows at most as fast as g. Note that formally $O(g)$ is the whole set of functions that grow at most as fast as g, in an asymptotic sense and up to constant factors.[12] Using this notation we can make the following true statements:

$$n \in O(n) \qquad 2n \in O(n) \qquad n \in O(n \log n)$$

We formally define $f \in O(g)$ as follows: There is a constant c and a $n_0 \in \mathbb{N}$ such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$. This is what we exactly mean by f grows at most as fast as g up to constant factors.[13] Alternatively, we could have also defined $\limsup_{n\to\infty} f(n)/g(n) < \infty$ in an analogy to eq. (1.4).[14] In a similar fashion we can also define o, Θ, Ω and ω, giving rise to the list of definitions in table 1.1.

Table 1.1: Definition of Bachmann-Landau symbols.

Notation	Definition	Alternative definition
$f \in o(g)$	$\forall c > 0 \exists n_0 \forall n > n_0 \quad f(n) < c \cdot g(n)$	$\lim_{n\to\infty} f(n)/g(n) = 0$
$f \in O(g)$	$\exists c > 0 \exists n_0 \forall n > n_0 \quad f(n) \leq c \cdot g(n)$	$\limsup_{n\to\infty} f(n)/g(n) < \infty$
$f \in \Theta(g)$	$\exists c, c' > 0 \exists n_0 \forall n > n_0 \quad c' \cdot g(n) \leq f(n) \leq c \cdot g(n)$	$f \in O(g) \cap \Omega(g)$
$f \in \Omega(g)$	$\exists c > 0 \exists n_0 \forall n > n_0 \quad f(n) \geq c \cdot g(n)$	$\liminf_{n\to\infty} f(n)/g(n) > 0$
$f \in \omega(g)$	$\forall c > 0 \exists n_0 \forall n > n_0 \quad f(n) > c \cdot g(n)$	$\lim_{n\to\infty} f(n)/g(n) = \infty$

1.2.2 Complexity of algorithms and problems

Complexity of loops. We can now use the big-O notation to quickly judge the complexity of algorithms. We essentially have two means of repetition in algorithms, namely loops and recursion. We will first focus on loops and start with this example to compute sum of squares:

$s \leftarrow 0$
for $i = 1$ to n **do**
  $s \leftarrow s + i^2$

We can quickly conclude that the time complexity is in $\Theta(n)$ simply by the fact that its costs in the RAM model are of *some* form of $c_1 n + c_2$, without analyzing in detail what c_1 and c_2 would be. They depend on the exact number of operations involved in a single loop iteration and the constant costs outside the loop, but we do not care for the big-O notation anyways. All that matters for the big-O notation is the number of loop iterations.

[11] This notation was introduced by Bachmann and Landau around 1900. In fact, the letter "O" stands for "Ordnung" in German.

[12] Many authors actually write "$f = O(g)$" instead, in which Knuth calls "=" a "one-way equivalence" [34].

[13] Historically, Knuth required $|f(n)| \leq cg(n)$, but modern textbooks like [11] do not. First, we could anyways write $|f| \in O(g)$ to express the stricter notion of Knuth and secondly, typically our functions of interest are actually non-negative.

[14] Note that the sequence $f(n)/g(n)$ may not converge but possess multiple accumulation points. Then lim does not exist, but the limes superior, denoted by lim sup, does and denotes the largest accumulation point. If the sequence converges then lim sup and lim are the same.

Note that we could have improved the above algorithm to a constant time complexity, which is $\Theta(1)$, by simply implementing the formula $\sum_{i=1}^{n} i^2 = {}^{n(n+1)(2n+1)}/_{6}$. Let us do another example:

```
i ← 1
while i ≤ n do
    i ← 2i
```

This loop only is performed *roughly* $\log_2 n$ times, because this is the number of times we double the 1 until we reach n. Hence, the complexity is $\Theta(\log n)$. Well, to be precise, the loop is executed exactly $1 + \lfloor \log_2 n \rfloor$ times, assuming $n \geq 1$. But for the big-O notation we can be ignorant of the +1 and the floor function $\lfloor\rfloor$. We can even be ignorant of the base of $\log n$, because $\log_b n = {}^{\ln n}/_{\ln b}$. This allows us to just write $\Theta(\log n)$, because the base of log has no influence on $\Theta(\log n)$. Let us do a third loop example:

```
i ← 1
while i ≤ n do
    j ← 1
    while j ≤ i do
        j ← j + 1
    i ← i + 1
```

The inner loop is executed i times for i going from 1 to n in the outer loop. So the inner loop is executed $\sum_{i=1}^{n} i = {}^{n(n+1)}/_{2}$ times in total. We conclude that the time complexity is $\Theta(n^2)$. Note that on the RAM model, the time $T(n)$ taken is not asymptotically equivalent to n^2, nor to ${}^{n(n+1)}/_{2}$, because in the above argument we did not take into account the number of operations involved during the loop iterations. It does not matter for the big-O notation – we can still conclude $T \in \Theta(n^2)$.

Lower and upper bounds. In terms of the big-O notation, insertion sort has a time complexity of $O(n^2)$. More precisely, the worst case is in $\Theta(n^2)$ and the best case is in $\Theta(n)$, but in *any* case it is in $O(n^2)$. We can either argue by eq. (1.5) or skip the asymptotic analysis and have look at the loops in algorithm 2.

In fact, insertion sort also runs in $O(n^3)$, but this would be a weaker statement then saying that it is in $O(n^2)$, because the latter is a worse (i.e., higher) upper bound than the former. In fact, $O(n^2)$ is a *tight bound* on the time complexity of insertion sort; it cannot be improved since the worst case is in $\Theta(n^2)$. It is also true that insertion sort takes $\Omega(n)$ time, i.e., insertion sort takes *at least* linear time in any case. This, again, is a tight lower bound on the time complexity of insertion sort.

Complexity of recursions. The question is whether we can do better than $O(n^2)$ for the sorting problem. Let us consider a different sorting algorithm, namely *merge sort*, which is given in algorithm 3. We first note that the merge routine in algorithm 3 takes $\Theta(n_1 + n_2)$ time, or $\Theta(n)$ time when n denotes the size of the *output*. When the time complexity depends on the size of the output then we call this the *output-sensitive* time complexity. To analyze merge sort we investigate the tree of recursive calls as illustrated in fig. 1.2. We have $\Theta(\log n)$ levels as we split each subsequence in equal halves from level to level. Further note that the time spent by merge is per level in total $\Theta(n)$. So summing up the times per level gives a total time complexity of $\Theta(n \log n)$ for merge sort.

A mathematical tool to analyze the time complexity of all kinds of recursive algorithms is the co-called *Master theorem*. For details see [11, pp. 102].

Algorithm 3 Merge sort

```
1:  procedure MERGE_SORT(seq)
2:      n ← LENGTH(seq)
3:      if n ≤ 1 then
4:          return seq
5:      m ← ⌊n/2⌋                                  ▷ split the sequence and recursively sort
6:      seq1 ← MERGE_SORT(seq[0 : m])
7:      seq2 ← MERGE_SORT(seq[m : n])
8:      return MERGE(seq1, seq2)                   ▷ re-merge the sorted subsequences

9:  procedure MERGE(seq1, seq2)                    ▷ seq1, seq2 are already sorted
10:     n1 ← LENGTH(seq1), n2 ← LENGTH(seq2)
11:     i1 ← 0, i2 ← 0, k ← 0
12:     seq ← ALLOCATE_ARRAY(n1 + n2)
13:     while i1 < n1 and i2 < n2 do
14:         if i2 = n2 or (i1 < n1 and seq1[i1] ≤ seq2[i2]) then   ▷ Take an element from seq1
15:             seq[k] ← seq1[i1]
16:             i1 ← i1 + 1
17:         else                                   ▷ Take an element from seq2
18:             seq[k] ← seq2[i2]
19:             i2 ← i2 + 1
20:         k ← k + 1
21:     return seq
```

Complexity of problems and optimality. So merge sort is in the worst case strictly better than insertion sort, since $n \log n \in o(n^2)$. However, the best case of insertion sort is strictly better than merge sort since $n \in o(n \log n)$. The question is now whether we can improve on the worst case of merge sort, i.e., is there a sorting algorithm that does better than $O(n \log n)$. In other words, can sorting be done in $o(n \log n)$ in the worst case?

Actually, the answer is "no". More precisely, every comparison-based sorting algorithm takes in the worst case $\Omega(n \log n)$ time. The proof goes as follows: Sorting means that we have to find the one permutation out of $n!$ possible permutations that is sorted. By each comparison between two elements of a sequence *any* algorithm can at best eliminate halve of the potential permutations as not containing the sorted one. So by repeatedly comparing elements we need at least $\Omega(\log n!)$ comparisons to find the one sorted permutation out of $n!$ possible. Recall (1.7), which implies that $\log n! \in \Theta(n \log n)$. We say that sorting has a *lower bound* of $\Omega(n \log n)$. Note

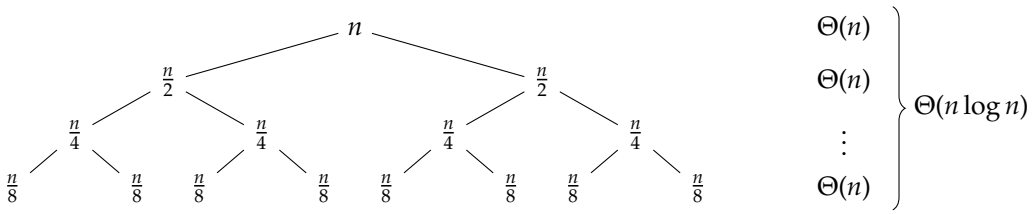


Figure 1.2: Sizes of subsequences in each recursion of merge sort. Gives $\Theta(n)$ time per level and $\Theta(n \log n)$ time in total.

that we just proved a property of the sorting problem as such!

Merge sort is therefore a worst-case *optimal* sorting algorithm (based on element comparison) since its worst-case complexity $O(n \log n)$ matches the lower bound $\Omega(n \log n)$ of the problem. Still we shall not forget that this notion of optimality is based on the big-O notation. A sorting algorithm can still be faster than merge sort for small enough n. Furthermore, there are other worst-case optimal sorting algorithms in terms of the big-O notation. Then a closer investigation of the constants by means of asymptotic analysis may be able distinguish between them. At some point we simply have to take runtime measurements of implementations.

1.2.3 Complexity classes

In the previous section, we introduced the big-O notation to classify the complexity of algorithms, say, in terms of time or space. We learned that the asymptotic equivalence gives a equivalence relation. The same is also true for Θ: It is reflexive as $f \in \Theta(f)$, it is symmetric as $f \in \Theta(g)$ implies $g \in \Theta(f)$ and it is transitive as $f \in \Theta(g)$ and $g \in \Theta(h)$ implies $f \in \Theta(h)$. In other words, by means of Θ we obtain whole classes of complexities that are distinct from each other.[15] In table 1.2 we list some common classes often occurring in practice. Some of the examples refer to the promised complexities of C++ algorithms of the STL on cppreference.com.

Table 1.2: Complexity classes by means of Θ in increasing order.

Θ-class	description	examples
1	constant	array access, amortized `std::vector::push_back()`
$\alpha(n)$	inverse Ackermann	amortized time for search in union-find
$\log n$	logarithmic	`std::binary_search()`, `std::map::insert(value)`
$\log^c n$	poly-logarithmic	
n	linear	linear search, merge, `std::unique()`
$n \log n$	linearithmic	`std::sort()`
n^2	quadratic	pairwise checks
n^3	cubic	naive matrix multiplication
2^n	exponential	brute-force on all subsets
$n!$	factorial	brute-force on all permutations

P and NP. In the theory of computing, we often consider even more coarser classes of complexity than Θ. For instance, we have the class P of problems that can be solved in polynomial time, i.e., in $O(n^k)$ time for some $k \in \mathbb{N}$. The class NP is the class of problems for which a solution can be verified in polynomial time.[16] Many classical problems are in P, like sorting, finding shortest paths in graphs, searching in trees or lists, solving linear equation systems, graph coloring, or string matching. Since 2002 we know that testing whether a number is prime is in P due to the *AKS primality test* [2] algorithm. Problems in NP are, for instance, the *traveling salesperson problem (TSP)*, the knapsack problem, the *boolean satisfiability problem (SAT)*, or the *graph isomorphism* problem. The SAT problem asks whether a given Boolean formula with n variables can be made true by assigning truth values to its variables. Assume one provides a

[15] Mathematically speaking, these are the equivalence classes of the equivalence relation.

[16] We strongly simplify matters here. To be more precise, P is the class of problems that can be solved in polynomial time on a deterministic Turing machine, where NP is the class of problems that can be solved in polynomial time on a non-deterministic Turing machine. However, it is known that solving a problem in polynomial time on a non-deterministic Turing machine corresponds to verifying a solution in polynomial time on a deterministic Turing machine.

solution – i.e., a variable assignment satisfying the formula – then verifying this solution can be done in P time.

P versus NP. Probably the most famous open question in computer science is whether $P = NP$. For sure, $P \subseteq NP$, i.e., every problem in P is also in NP. And although most researchers believe that $P \neq NP$, this has not been proven yet.[17] If it would actually turn out that $P = NP$ then this would have a huge impact. In particular, many cryptographic methods are based on the assumption that certain problems are hard to solve, e.g., being in NP. If $P = NP$ then this would mean that those problems could actually be solved in polynomial time. For instance, the RSA cryptosystem is based on the assumption that factoring large numbers is hard. It is known that integer factorization is in NP. If $P = NP$ then this would mean that factoring large numbers is not hard after all and the RSA cryptosystem would be broken.[18]

We do not know whether $P = NP$, but we still want give the "hardest" problems in NP a name, in some sense. We call those the *NP-complete* problems. To make this more precise, we first need the concept of a *reduction*: Assume we have a problem A and a problem B. We say that B can be reduced to A if we can take an input of B, transform it to an input of A, obtain the output through an algorithm for A, and transform it back to a result of B. If this forth- and back-transformation takes f time then we speak of an f-reduction. This allows us to transfer lower bounds of problems: If B is in $\Omega(f)$ and B can be $o(f)$-reduced to A then A must also be in $\Omega(f)$, otherwise $A \in o(f)$ and we could solve B in $o(f)$ time via A, violating the lower bound of B, see fig. 1.3. We will use this technique in chapter 9 to argue that that the convex hull is in $\Omega(n \log n)$ by a reduction of sorting to convex hulls.

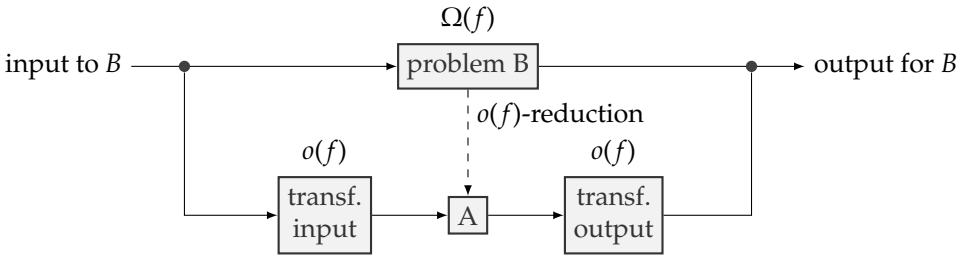


Figure 1.3: The lower bound $\Omega(f)$ of B applies also to A by the $o(f)$-reduction of B to A.

Now in order to define the "hardest" problems in NP we follow exactly this idea: We define as NP-complete the set of problems $A \in NP$ such that *every* problem $B \in NP$ can be P-reduced to A. This means that if we could solve a single NP-complete problem in polynomial time then we could solve *all* problems in NP in polynomial time, and in fact prove $P = NP$. The first problem that was shown to be NP-complete is the *boolean satisfiability problem (SAT)* problem, based on which in 1972 the famous *Karp's 21 NP-complete problems* were published, many of them being graph problems. We can summarize this as follows:

> No NP-complete problem can be solved in polynomial time, unless a single one can (meaning $P = NP$).

[17] The Clay Mathematical Institute defined seven *Millennium Prize Problems* in 2000 and put up a million dollar prize for each of them. The question whether $P = NP$ is one of them. Only one was solved so far: the Poincaré conjecture.

[18] Though, in practice the question would still be how large the exponents of the polynomial-time algorithms for integer factorization then would be. This is related to the *time hierarchy theorem* and *Cobham's thesis*.

Complexity zoo and decidability. Now there are actually many more complexity classes than just P and NP. For instance, $PSPACE$ refers to the class of problems requiring polynomial space on a deterministic Turing machine. EXP is the class of problems that can be solved in exponential time.[19] BQP is the class of problems that can be solved in polynomial time on a quantum computer. A famous problem in BQP is integer factorization and Shor's algorithm is an example. NC is the class of problems that can be solved in poly-logarithmic time on a parallel computer with a polynomial number of processors. The *complexity zoo* by Scott Aaronson is a website at complexityzoo.net that gives an overview over all those classes.

A problem beyond complexity is the *halting problem*. It takes as input an encoded program and an input to for it and asks whether it would halt on this input, i.e., finish in finite time. If we could solve the halting problem then we could tell in general whether a loop is an endless loop, for instance. This would be a sub-task of verifying whether some code is correct, i.e., whether it produces the correct output. It is a famous result by Alan Turing that the halting problem is undecidable, i.e., there is no algorithm that can solve the halting problem in general. This is a consequence of the *incompleteness theorem* on first-order logic by Kurt Gödel.

1.3 Summary

In this chapter we started from the very beginning of algorithm analysis, namely with algorithms and their basic properties: correctness and efficiency. We learned how to analyze the correctness using loop invariants. We discussed the two main resources taken by an algorithm, namely time and space. We introduced models of computing, in particular the RAM model, and analyzed insertion sort against it. Then we introduced the mathematical notion of asymptotic equivalence and the big-O notation to classify the complexity of algorithms and algorithmic problems. This allowed us to speak of optimality of algorithms. Finally, we touched a few topics of theoretical computer science, like complexity classes, the $P = NP$ problem and the halting problem.

1.4 Exercises

Exercise 1.1 (⋆⋆)**.** Below is an algorithm for a binary search on a sorted array `seq`, in which we search for an element `key`. If `key` is found the index of the element in `seq` is returned, and otherwise -1.

```
1:  procedure BINARY_SEARCH(seq, key)
2:      l ← 0, r ← LENGTH(seq)
3:      while l < r do
4:          m ← ⌊(l + r)/2⌋
5:          if seq[m] = key then
6:              return m
7:          else if seq[m] < key then
8:              l ← m + 1
9:          else
10:             r ← m
11:     return −1
```

Prove the correctness by (i) showing that the algorithm terminates and (ii) by giving an adequate loop invariant.

[19] It is unknown whether $NP \subsetneq EXP$. This is related to the *exponential time hypothesis*.

Furthermore, analyze time and space taken by the algorithm in the best and worst case on the RAM model. For simplicity, we may assume that LENGTH(seq) is a power of two. (It is not so important that you get the constants right, but you should be able to defend your constants.)

Exercise 1.2 (⋆). In the big-O notation we simply write $\log n$ without specifying the base of the logarithm. To investigate whether this is an issue, check the following statements:

$$\log_{10} n < \log_2 n \quad \text{for all } n > 1$$
$$\log_2 n \sim \log_{10} n$$
$$\log_{10} n \in o(\log_2 n)$$
$$\log_2 n \in \Theta(\log_{10} n)$$

Hint: Note that $\log_b n = \frac{\ln n}{\ln b}$ for $b > 0$. Bonus exercise: Prove this formula.

Exercise 1.3 (⋆). Give the simplest asymptotic equivalent expression for

$$\sum_{k=1}^{n} k$$
$$n\sqrt{n} + 3n^2$$
$$2n \log n + \log n!$$
$$\frac{\log 2n}{\log n}$$

Exercise 1.4 (⋆⋆). Consider the first column in table 1.2 as functions $f\colon \mathbb{N} \to \mathbb{R}\colon n \mapsto f(n)$. Plot them to illustrate the growth of the complexity classes.

Note: Consider making two plots, one for the sub-exponential functions and another with the rest. Choose a suitable range for values for n. Also consider using logarithmic scales adequately. Make a good figure. You may skip the inverse Ackermann function.

Exercise 1.5 (⋆). Let us say we have a computer that that performs 10^9 instructions per seconds, kind of a 1 GHz computer. Assume we have a couple of algorithms for different problems that take $f(n)$ instructions for input size n:

Search	Sort I	Sort II	matrix mult.	password cracker	naive TSP
$\log n$	$n \log n$	n^2	n^3	2^n	$n!$

Compute the runtimes of the algorithms for $n \in \{10, 20, 30, 40, 50, 10^2, 10^5, 10^7, 10^9\}$. Use illustrative units from nano seconds to years. Skip values once the age of the universe is clearly exceeded. (Hint: You may write a program. But we really want to have a table as output.)

Exercise 1.6 (⋆). Let $f, g\colon \mathbb{N} \to [0, \infty)$ be two functions. Show that exactly one of three cases is true:

$$f \in o(g) \quad \text{or} \quad f \in \Theta(g) \quad \text{or} \quad f \in \omega(g).$$

For simplicity we assume that $\lim_{n\to\infty} f(n)/g(n)$ exists.

Exercise 1.7 (⋆). Give a simpler asymptotic equivalent expression for the following expressions:

$$\sum_{k=1}^{n} k^2 \qquad \frac{n^2 + n}{2^n} \qquad \frac{2^n}{e^n}$$

Exercise 1.8 (⋆). Show that $2^n \in o(n!)$.

Exercise 1.9 (⋆). Show that $\binom{n}{2} \in \Theta(n^2)$ and $\binom{n}{k} \in \Theta(n^k)$ for a fixed k.

Exercise 1.10 (⋆). What is the space complexity of insertion sort and merge sort in big-O notation? Explain your answer.

Exercise 1.11 (⋆). What is the time complexity of selection sort in big-O notation below? This algorithm has a beneficial property from a security perspective. What is it?

Algorithm 4 Selection sort

```
1: procedure SELECTION_SORT(seq)
2:     n ← LENGTH(seq)
3:     for i = 0 to n − 1 do
4:         k ← i
5:         for j = i + 1 to n do
6:             if seq[j] < seq[k] then
7:                 k ← j
8:         SWAP(seq[i], seq[k])
```

Exercise 1.12 (⋆). Prove the correctness of selection sort in exercise 1.11. What is the invariant in the outer loop that helps to prove correctness?

Exercise 1.13 (⋆). What is the time complexity of each of the three blocks of Python code in Θ-notation in dependence of n?

```python
s = 0
for i in range(n):
    for j in range(n):
        s += 1

s = 0
for i in range(n):
    for j in range(i):
        s += 1

s = 0
for i in range(n):
    j = 1
    while j <= n:
        s += 1
        j *= 2
```

As a bonus question: What is the complexity of the following loop?

```python
s = 0
for i in range(n):
    j = 1
    while j < i:
        s += 1
        j *= 2
```

Exercise 1.14 (⋆). We add an element into a sorted linked list such that it stays sorted. What is the time complexity?

Exercise 1.15 (⋆). What is the time complexity of appending a single element to a dynamic array with doubling strategy?

Exercise 1.16 (⋆). As above, but say we start with an empty dynamic array and append n elements. We call the average complexity of adding one element the *amortized* complexity. What is it?

Exercise 1.17 (⋆). GitHub's Copilot auto-completed on 2024-08-06 the following implementation after the comment.

```
def combine(a, b):
    """Combines the sorted lists ‘a‘ and ‘b‘ into a single sorted list in
    linear time."""
    return sorted(a + b)
```

What is wrong with this implementation? How can we fix it?

Exercise 1.18 (⋆⋆). GitHub's Copilot auto-completed on 2024-08-06 the following implementation after the comment:

```
def kill_repeated_elements(li):
    """The following loop removes repeated elements in a list in linear time."""
    i = 0
    while i < len(li):
        if li[i] in li[:i]:
            li.pop(i)
        else:
            i += 1
    return li
```

We assume that `li` is a dynamic array of length n.

- Check the correctness of the code.
- What is the time complexity in big-O notation?
- How does it compare with the C++ implementation `std::unique()` in the STL?
- Bonus: How does it compare to `return list(set(li))`? (It was given by Copilot when the comment said "Removes repeated elements in a list in linear time.".)

Exercise 1.19 (⋆). Consider the following problem: Given a sequence $p_1, \ldots, p_n$ of points on a circle in the plane, compute the polygon formed by the n points. (The polygon shall be encoded by some sequence $q_1, \ldots, q_n, q_1$ of vertices along the polygon boundary.)

Give a lower bound of this problem and an optimal algorithm in terms of time complexity.

Chapter 2

Performance optimization

2.1 Introduction

In the previous chapter we learned about algorithm analysis. We distinguished between algorithms and implementations, between models of computing and computers, see fig. 2.1. The advantage of the algorithm perspective is that we can make theoretical statements about algorithms (upper bounds on complexity) and also problems (lower bound on complexity). We can speak of optimality when the upper bound of an algorithm matches the lower bound of a problem. We can make predictions about the resource consumption of algorithms, without even requiring to implement any algorithm.

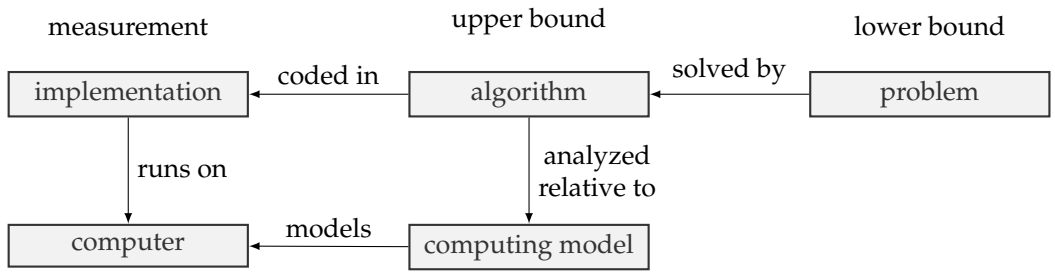


Figure 2.1: An algorithm is coded into an implementation, running on a computer, which is modeled by a computing model, relative to which the algorithm is analyzed, which solves a specific problem.

But there is no free lunch: The downside of this perspective is that we had to go through certain simplifications and abstractions that are sometimes to crude. Yet, if we undo the simplifications – like going from big-O notation to the exact costs w.r.t. the RAM model of computing – then the application of the theory becomes very difficult.

We then proceed by investigating the real implementation on real computers, which is one topic of this chapter. But then we switch from the analysis of algorithms to the measurement of implementations; from time and space complexity to runtime and memory footprint; from algorithm design to code optimization. Yet the goal stays the same: We want the best performing solution to a problem. However, there are three general widsoms to this game:

Wisdom 1. Optimizing an implementation cannot compensate for a bad algorithm.

The rationale behind this wisdom is rooted in the fact that code optimization only improves some constants in the time taken. For large enough input size n, this can never compensate for

a bad algorithm with bad time or space complexity.

For instance, let us compare merge sort versus selection sort (see exercise 1.11), which is $\Theta(n \log n)$ versus $\Theta(n^2)$. Taking constants into consideration, we have merge sort running in about $c_1 n \log_2 n$ time and selection sort in $c_2 n^2$ for some unknown constants c_1, c_2. For a moderately small input size of $n = 1000$, we get for merge sort about $c_1 10^4$ and for selection sort about $c_2 10^6$. In order for selection sort to beat merge sort, we need $c_1 \geq 100 c_2$. In other words, the constants of merge sort would need be a hundred times the constants of selection sort in order for selection sort to make up the inferior time complexity! And this gap between the two algorithms increases with increasing input size n, namely at the speed of $\Theta(n/\log_2 n)$.

The moral of this wisdom is: First choose the right algorithm then go for optimizing its implementation, not the other way round.

Wisdom 2. Premature optimization is the root of all evil ...

... (or at least most of it) in programming, said Donald Knuth in [35]. Always be aware that code optimization comes at costs in various form. Development time is maybe the least of them. Not seldomly it happens that optimizing code performance reduces readability and maintainability of code.

At the same time, we need to measure in order to find out whether this particular piece of code to be optimized actually contributes much to the overall runtime, i.e., whether the costs of optimization pay off or whether those costs are better invested somewhere else.

Wisdom 3. Never guess the effectiveness of an optimization, always measure it.

Say we have identified a piece of code to be optimized. Not seldomly it happens that a seemingly – sometimes obvious – faster alternative for this piece of code actually performs equal or even worse due to all kind of hard-to-predict effects, like cache misses, or wrong assumptions on how code is compiled or executed by processors. So any promised improvement by a code optimization needs to be verified by proper measurements. Otherwise we may pay the price of a code change that is useless. And the price of changing code is again manifold, like the risk to introduce bugs, the time paid for development, review and testing, the organizational overhead in distributed development, version control systems or continuous integration.

So the worst scenario of code optimization would be something like starting with a bad algorithm and then perform ineffective micro optimizations, at irrelevant places, while introducing bugs, increasing technical debt and binding the resources of other developers.

In the next sections of this chapter we will learn techniques to measure code and techniques to make code perform better.

2.2 Measuring

We learn about two different approaches to measure performance of code, namely internal versus external. For the internal approach we measure runtime (or memory footprint) within the code to be tested. In the external approach, we use external tools to make these measurements. The internal approach gives us more fine grain possibilities, but it requires to have the source code. The external approach benefits to have the source code, but could also be applied to binaries. The internal approach is about measuring techniques, the external approach about profiling.

2.2.1 Measuring techniques

Notions of time. In the following we will first concentrate on measuring time. But let us first differentiate between different notions of time in this context: We can differentiate between

- the *physical time*,
- the number of *clock ticks* of the processor, or
- the *instructions* executed.

The relation between the physical time and the number of clock ticks may change, e.g., due to throttling mechanisms of the processor, in particular frequency scaling. The relation between clock ticks and instructions may vary, as contemporary[1] processors are superscalar processors with *data-dependent* instruction-level parallelism, where multiple instructions are in parallel dispatched to different execution units, effectively being able to execute more than one instruction per clock cycle.

Measuring physical time. Let us concentrate on the physical time first. Yet again we have to distinguish between *wall clock time* and the *CPU time*. The wall clock time is the elapsed real time. The CPU time is only the time the processors actually executed our code. The accounting of the CPU time is done by the operating system (OS). The wall clock time can be much larger than the CPU time when our code is interrupted during execution, e.g., by a preemptive multi-tasking OS or simply because the computer went into sleep mode. The wall clock time can also be less than the CPU time if our code is executed in parallel on multiple processors[2].

On POSIX-compatible OSs[3], we can use the libc function `clock_gettime()` to measure the physical time.[4] It gets as first argument the clock to be used. Some of them are:

- `CLOCK_REALTIME` is the wall clock time. This clock is affected by system time changes, and in fact it can go backwards.
- `CLOCK_TAI` is Linux-specific. It derives from `CLOCK_REALTIME`, but it ignores leap seconds and it does not experience (forward or backwards) jumps.[5]
- `CLOCK_MONOTONIC` is a monotonically increasing time since[6] the system was booted. It is not affected by jumps or by system suspends, but by incremental adjustments, e.g., through NTP[7] synchronization.
- `CLOCK_THREAD_CPUTIME_ID` measures the CPU time of the thread. `CLOCK_PROCESS_CPUTIME_ID` measures the CPU time of the process, meaning the total over all its threads.

[1] For desktop PCs, the Intel Pentium processors (1993) were the first superscalar processors. Mainframe computers already had superscalar processors in the 1960s, like Cray's CDC 6600 or IBM's System/360.

[2] We only say "processors", but we also mean multiple cores or hyper-threading technologies such that the OS's CPU time accounting.

[3] POSIX stands for portable operating system interface. The following OSs are POSIX-compatible (or mostly compatible): Linux, macOS, various BSDs, VxWorks, various RTOSs, AIX, HP-UX, and many more. The following OSs are not: Windows. But you can gain POSIX for Windows through MinGW, Windows Subsystem for Linux and other technologies.

[4] The man page [10] gives all details. You may view man pages also online, like on `https://www.mankier.com/2/clock_getres`.

[5] The *international atomic time (TAI)* is an international time standard, formed by an average over hundreds of atomic clocks. It differs form *coordinated universal time (UTC)* notably by not having leap seconds.

[6] POSIX only defines "some unspecific point in the past", which Linux interprets as boot time.

[7] The *network time protocol (NTP)* is the prevalent standard to synchronize clocks of computers.

In listing 2.1 we see a complete program that illustrates (i) how to measure time of a code section and (ii) prints the resolution of the used clock. We use CLOCK_PROCESS_CPUTIME_ID as we are interested in the CPU time rather than wall clock time for this measurement. Also, if the code to be measured is multi-threaded then we would would like to have the accumulated time used on all threads. If, however, computational load would have been offloaded from the CPU – like GPU computing or distributed computing – then CLOCK_MONOTONIC would be more reasonable.

Listing 2.1: Measuring time with `clock_gettime()`

```c
#include <stdint.h>
#include <stdio.h>
#include <time.h>

void timespec_sub(struct timespec *x, struct timespec *d) {
  x->tv_sec -= d->tv_sec;
  x->tv_nsec -= d->tv_nsec;
  if (x->tv_nsec < 0) {
    x->tv_nsec += 1000000000;
    x->tv_sec -= 1;
  }
}

int main() {
  struct timespec start, end;

  clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &start);
  /* [...] code to be measured */
  clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &end);

  timespec_sub(&end, &start);
  printf("Timespan: %jd.%09ld s\n", end.tv_sec, end.tv_nsec);

  /* Resolution of this clock: */
  struct timespec res;
  clock_getres(CLOCK_PROCESS_CPUTIME_ID, &res);
  printf("Resolution: %jd.%09ld s\n", (intmax_t)res.tv_sec, res.tv_nsec);
  return 0;
}

/* $ make clock_gettime_demo && ./clock_gettime_demo
 *   Timespan: 0.000002706 s
 *   Resolution: 0.000000001 s
 * $ uname -r -s -p
 *   Linux 6.10.3-gentoo Intel(R) Core(TM) i5-6300U CPU @ 2.40GHz
 * $ cat /sys/devices/system/clocksource/clocksource0/available_clocksource
 *   tsc acpi_pm
 * $ cat /sys/devices/system/clocksource/clocksource0/current_clocksource
 *   tsc */
```

This code is closely oriented to the demo code in the man page of `clock_gettime()` on Linux [10]. Note that every code between the `clock_gettime()` calls will be measure, so do not add code not be included to the measurement, like `print()` for debugging. Also note that this particular demo does not use floating-point operations but is integer only, so we do not have to discuss numerical accuracy here. However, we shall discuss the accuracy of the clock: The *accuracy* tells us by how much the true time can differ from the measured time. That is, the accuracy depends on how accurately the clocks have been set, say, through time synchronization protocols like NTP.[8] But since we are interested in the relative time of code sections, the accuracy of our clock

[8] On modern Linux systems, in the output of `timedatectl timesync-status`, the value of `offset` is some estimate

is not that important.

Resolution of time measurement. Listing 2.1 prints the *resolution*, which tells us the smallest time unit that can be represented, and is 1 ns on our system. The resolution is also called *precision*. To get a better understanding of the actual resolution we can expect, we need to know more about the actual clock sources. In Linux, we can list the available clock sources as follows as given in the last comment of listing 2.1. The *time stamp counter (TSC)* is a register on x86 processors that counts the number of cycles since the last processor reset[9]. However, this timer would then be affected by power saving mechanisms. This is why contemporary Intel processors also provide a so-called *constant rate TSC*, which runs at the nominal (i.e., maximal) frequency of the processor and which is actually used by Linux.[10] The *advanced configuration and power interface (ACPI)* is a standard for system and power management[11], where *advanced power management (APCI-PM)* is the power management timer. Other systems – like a Raspberry Pi, which is ARM-based – will have different output.[12]

Understanding the measurement of physical time towards performance evaluation of code took us to discussions on the level of OSs and hardware. This typically leads to code examples in the C programming language. However, other programming languages provide similar function to measure time in an OS-agnostic fashion. For instance, in C++11 we have the "data and time library" with classes like `std::chrono::clock` and `std::chrono::duration`. Since C++20, we also have clocks like `std::chrono::tai_clock` and various convenience functions. In order to understand what is happening behind these abstractions, we have to consult the documentation. For instance, according to cppreference.com, `std::chrono::clock` effectively gives the result of `clock_gettime` on `CLOCK_PROCESS_CPUTIME_ID` on POSIX systems, however, possibly with a lower resolution[13].

Measuring memory. Measuring memory is on the one hand simpler than measuring time, because memory is a concept on the level of the OS, i.e., we do not need measurement hardware as for timers. On the other hand, we cannot easily measure the memory taken by a piece of code. By memory we typically mean heap memory. In fact, the stack memory has actually a fixed size for most OSs.[14]

For POSIX-compatible OSs, we can us the libc function `getrusage()`, which gives us various resource usage statistics.[15] It reports the maximum resident set size (RSS), which is the total size of the process in the main memory. Furthermore, the glibc reports by `mallinfo2()` statistics concerning `malloc()` and `free()`.

Another option would be to override the `malloc()` and `new()` calls to do bookkeeping of the memory usage.[16] This is a bit more involved, but it gives us the most fine-grained control over memory usage. Similarly, one could theoretically read out the stack pointer to get an idea of the

---

of the accuracy.

[9] Which is typically since it was powered up.

[10] Have a look for the flag `constant_tsc` in *cat/proc/cpuinfo* to check whether your processor supports this and the output of `journalctl -b -0 | grep -i tsc` for related boot messages from the Linux kernel.

[11] Configuring keyboard hotkeys, display brightness or suspend modes.

[12] Until end of 2010s, x86 systems typically also listed the *high precision event timer (HPET)*, which is a hardware timer standard [28] by Intel. But it is less efficient than TSC and caused various problems, leading to the removal of HPET from the list of available clock sources.

[13] The constant `CLOCKS_PER_SEC` is 10^6 on our system, giving a resolution of 1 µs.

[14] In Linux, the stack size is typically 8 MB and is reported by `ulimit -s`.

[15] However, POSIX only defines user and system CPU time to be returned. The memory usage is not standardized, though reported by Linux.

[16] Various tools use the `LD_PRELOAD` mechanism to hook into as memory profiler ord debugger. There are also illegitimate use cases for this mechanisms, like rootkits.

stack usage. Anyhow, the preferred techniques to measure memory is actually through profiling tools like Valgrind.

2.2.2 Profiling

Profiling and tracing. *Profiling* is the process of measuring the resource consumption of a program. But instead of manually adding measurement code to the program, we use external tools to measure the program, which use different techniques. Profiling tools can measure various resources, like time, memory, or disk and network I/O. We will concentrate on time and memory profiling.

A related term is *tracing*. Tracing is the process of recording the execution of a program. We will not go further to into details of tracing, but only mention *strace* and *ltrace* as two examples that trace system calls and library calls on Linux, respectively. However, often the borderline between profiling and tracing is blurred, as many profiling tools also trace the execution of the investigated program.

Assume we would like to identify the *hotspots* of a program, i.e., the parts of the code that consume most of the time. There are a couple of profiler techniques that achieve this:

- A *sampling profiler* periodically interrupts the program and records the current instruction pointer. This gives a statistical view on which parts of the code are executed how often. Some processors[17] provide hardware *performance counters*, to allow for profiling with low or virtually no overhead. These are instrumented by *perf* [40], which is the most popular sampling profiler on Linux.

- A different profiler technique is *instruction set simulation*. The program is run on a virtual machine, which allows for in-depth runtime, cache and memory debugging, tracing and profiling. This technique is very precise and powerful, but also much slower. This technique is used by *Valgrind* [50], which is a very popular tool on Linux and a few other OSs.

- An *instrumenting profiler* transparently adds measurement code to the program. This gives a more precise view on the resource consumption, but may also change the program behavior.

In the following we will give a brief overview over two powerful and popular profiling tools on Linux, namely perf and Valgrind. However, these tools are easier to demonstrate than to explain in theory, which we will do in section 2.2.3.

Perf. Perf is a performance analysis tool for Linux that is capable to analyzing the entire OS – kernel and user space – and collects data from various event sources, like

- hardware events (performance counters), e.g., cache misses, branch mispredictions, or instructions completed[18],

- software events, e.g., central processing unit (CPU) migrations, page faults, or context switches, or

[17] Like Intel Pentium III, AMD Athlon, ARM Cortex-A5 and newer.

[18] Superscalar processors with techniques like *speculative execution* actually execute more instructions than might turn out to be needed. So "instructions completed" is actually not the official term, but rather the term *instruction retired* is used to denote the number of completed instructions that actually were proven to be on the control flow.

- tracepoints, which are static probes in the kernel placed by kernel maintainers to measure the kernel behavior.

The complete list of events can be printed by `perf list` as shown in listing 2.2; it produces an output of over 3600 lines on this machine. The most basic command of perf is `perf stat`, which gives a summary of the performance counters. A simple demo is given in listing 2.2. We can also use perf for profiling. This is done in two steps: (i) we need to record data, which is stored in a file, (ii) we can then analyze the data. In listing 2.2 we see a brief demo.

Listing 2.2: A demo of perf showing the list of events, collecting statistics and profiling programs.

```
$ perf list
  branch-instructions OR branches                    [Hardware event]
  branch-misses                                      [Hardware event]
[...]
  task-clock                                         [Software event]
  duration_time                                      [Tool event]
[...]
cpu:
  L1-dcache-loads OR cpu/L1-dcache-loads/
  L1-dcache-load-misses OR cpu/L1-dcache-load-misses/
[...]
floating point:
  fp_arith_inst_retired.128b_packed_double
[...]
pipeline:
  arith.divider_active
[...]

# System-wide collection of statistics. Abort command with Ctrl-C after some
# seconds.
$ perf stat
 Performance counter stats for 'system wide':

         17,468.61 msec cpu-clock                        #    4.002 CPUs utilized
            21,977      context-switches                 #    1.258 K/sec
             1,545      cpu-migrations                   #   88.444 /sec
[...]
# Collection of statistics for a specific command, e.g., ls.
$ perf stat -- ls
# Attach to process with PID 2262 and collect number of CPU cycles for 2 seconds
$ perf stat -e cycles -p 2262 sleep 2

# Profiling and analyzing using perf
$ perf record -- ls
$ perf report
$ perf annotate
```

Valgrind. Valgrind provides a whole battery of tools for the *dynamic analysis* of programs, i.e., at execution rather than compile time. While it is a simulation-based profiler, different tools of Valgrind add different amount of instrumentation code. Valgrind might be best known for its memory checker *memcheck*, which detects various memory bugs, e.g., leaks, double frees, or use after free.

Note that we want to compile the program with debugging information using the compiler flag `-g`. This enables various tools of Valgrind to report on the level of line numbers, functions and source files. Valgrind comes with a battery of profiling and analysis tools:

- *Cachegrind* is profiler that precisely collects the numbers of every instruction executed by the program. This makes Cachegrind slow, but gives highly reproducible results.[19]
- *Callgrind* is a call graph profiler and is closely related to Cachegrind. However, while Cachegrind collects flat data, Callgrind collects hierarchical data. That means, if function `f()` calls `g()` then the costs of `f()` includes the execution costs of `g()`.
- *Massif* is a heap profiler. It records data on the allocated space, which code allocated the space and the extra space necessary for the heap organization. In addition, Massif can also be enabled to measure the stack usage.
- And there are number of other tools, like the thread-error detectors *Helgrind* and *DRD*, or the dynamic heap analysis tool *DHAT*.

2.2.3 Case study: Profiling merge sort

Let us do some practice-minded demonstration of perf and Valgrind. In listing 2.3 we have an implementation of merge sort in C, or C99 to be specific, as given in algorithm 3. The `main()` routine demonstrates `mergesort()` on a random integer array of some size, which we are going to profile to gain insight where optimizations might be promising. Recall wisdom 2? Mind to have a guess how much time is spent where?

We compiled the code with the `-g` compiler flag to have debugging information, which will make it easier to interpret the profiling results.

Perf. We first run the sampling profiler perf to get a first and easy impression:

```
$ CFLAGS="-g" make mergesort-demo
$ perf record -- ./mergesort-demo
$ perf report
[...]
# Overhead  Command          Shared Object          Symbol
# ........  ...............  .....................  ................................
#
    22.16%  mergesort-demo   mergesort-demo         [.] merge_sort
    20.85%  mergesort-demo   [unknown]              [k] 0xffffffff8d000fde
    20.27%  mergesort-demo   libc.so.6              [.] cfree
    20.06%  mergesort-demo   libc.so.6              [.] random
    10.10%  mergesort-demo   ld-linux-x86-64.so.2   [.] intel_check_word.constprop.0
     4.94%  mergesort-demo   ld-linux-x86-64.so.2   [.] __GI___tunables_init
     1.62%  mergesort-demo   ld-linux-x86-64.so.2   [.] _dl_start
```

We see that most of the time is spent in the `merge_sort()` function. Interestingly, we miss the `merge()` function, which presumably should take a non-negligible amount of time. If the compiler optimization level would allow for inlining then this could be an explanation. We could check the disassembly to prove this explanation wrong. Let us repeat the measurement:

```
$ perf record -- ./mergesort-demo
$ perf report
[...]
# Overhead  Command          Shared Object          Symbol
# ........  ...............  .....................  ................................
#
```

[19] Cachegrind also comes with a rather basic simulation for the cache hierarchy and branch predictions, which is by default turned off. But for this use case perf is the preferred tool, which uses performance counters of the actual hardware.

```
    45.27%  mergesort-demo  mergesort-demo         [.] merge
    19.62%  mergesort-demo  mergesort-demo         [.] merge_sort
    19.45%  mergesort-demo  ld-linux-x86-64.so.2   [.] _dl_relocate_object
    10.14%  mergesort-demo  ld-linux-x86-64.so.2   [.] intel_check_word.constprop.0
     4.51%  mergesort-demo  ld-linux-x86-64.so.2   [.] __GI___tunables_init
     1.02%  mergesort-demo  [unknown]              [k] 0xffffffff8d000fde
```

Now we see both, `merge()` and `merge_sort()`. Recall that a sampling profiler gives us a statistical view on the execution of the program. There is a chance that perf never saw the instruction pointer having resided in `merge()`. Or `merge_sort()`. And our demo runs for a very short time, so this can be an issue. We can verify this explanation by printing the number of samples using the option `-n`:

```
$ perf report -n
[...]
# Overhead  Samples  Command  Shared Object          Symbol
# ........  .......  .......  .....................  ................................
#
    45.27%        2  mergeso  mergesort-demo         [.] merge
    19.62%        1  mergeso  mergesort-demo         [.] merge_sort
    19.45%        1  mergeso  ld-linux-x86-64.so.2   [.] _dl_relocate_object
    10.14%        1  mergeso  ld-linux-x86-64.so.2   [.] intel_check_word.constprop.0
     4.51%        1  mergeso  ld-linux-x86-64.so.2   [.] __GI___tunables_init
     1.02%        6  mergeso  [unknown]              [k] 0xffffffff8d000fde
```

Note that the percentages on the left are quite worthless given the absolute numbers of samples. We could change our program and simply sort over a larger array. But this alters our program to be measured. We can also increase the sampling frequency, which is by default 4 kHz. We can actually specify `max`, which then instructs perf to use the highest frequency allowed by the kernel:

```
$ perf record -F max -- ./mergesort-demo
$ perf report -n
# Overhead  Samples  Command          Shared Object          Symbol
# ........  .......  ...............  .....................  .................
#
    27.25%        8  mergesort-demo   mergesort-demo         [.] merge
    16.89%        5  mergesort-demo   mergesort-demo         [.] merge_sort
     7.18%        4  mergesort-demo   ld-linux-x86-64.so.2   [.] do_lookup_x
     6.58%        2  mergesort-demo   libc.so.6              [.] cfree
     6.16%        2  mergesort-demo   libc.so.6              [.] malloc
[...]
```

This run now gave us a bit more reliable numbers. Nevertheless, rerunning the record step will give us different numbers by the nature of sampling profilers. What do we learn from this run about our merge sort implementation?

- We spend about the same time in `merge()` as in `merge_sort()`.
- We pay quite some overhead for memory allocation and deallocation.

Valgrind. Let us switch now to Valgrind, which is a more fine-grained, simulation-based profiler. We first check that we have no memory bug in our demo:

```
$ valgrind ./mergesort-demo
[...]
==536531== HEAP SUMMARY:
==536531==     in use at exit: 0 bytes in 0 blocks
```

Listing 2.3: An implementation `mergesort-demo.c` of merge sort in C99, mostly generated by GitHub's Copilot on 2024-08-13.

```c
#include <stdlib.h>

/** Merge the sorted arrays 'left' and 'right' into the output array 'arr'. */
void merge(int *arr, int *left, int left_size, int *right, int right_size) {
  int i = 0, j = 0, k = 0;
  while (i < left_size && j < right_size) {
    if (left[i] <= right[j])
      arr[k++] = left[i++];
    else
      arr[k++] = right[j++];
  }
  while (i < left_size)
    arr[k++] = left[i++];
  while (j < right_size)
    arr[k++] = right[j++];
}

/** A merge sort on the array 'arr' of given size n. */
void merge_sort(int *arr, int n) {
  if (n <= 1)
    return;

  int mid = n / 2;
  int *left = malloc(mid * sizeof(int));
  int *right = malloc((n - mid) * sizeof(int));

  for (int i = 0; i < mid; i++)
    left[i] = arr[i];
  for (int i = mid; i < n; i++)
    right[i - mid] = arr[i];

  merge_sort(left, mid);
  merge_sort(right, n - mid);
  merge(arr, left, mid, right, n - mid);

  free(left);
  free(right);
}

int main(int argc, char *argv[]) {
  /* Constructs an integer array initialized with reproducible random numbers. */
  const size_t n = 1024;
  int arr[n];
  srand(0);
  for (int i = 0; i < n; i++)
    arr[i] = rand();

  merge_sort(arr, n);
  return 0;
}
```

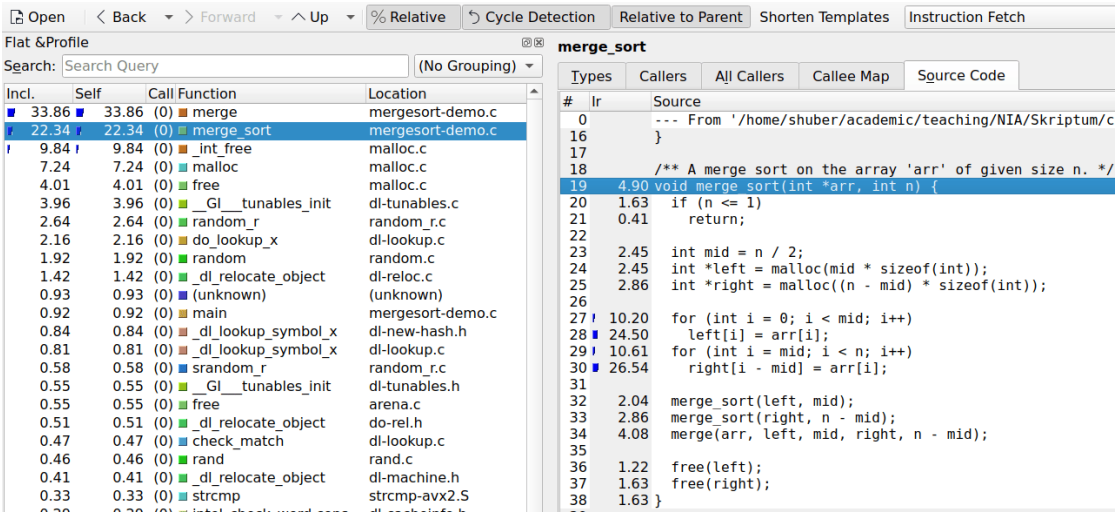


Figure 2.2: The output of cachegrind visualized in kcachegrind.

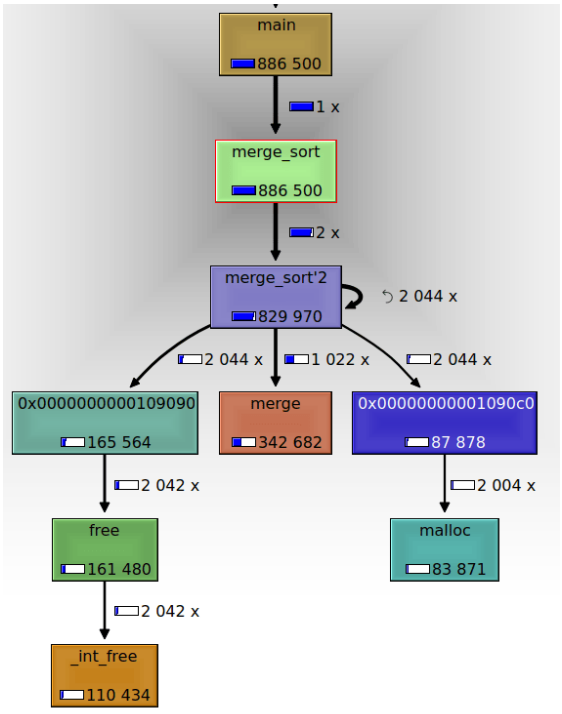


Figure 2.3: The callgraph generated by callgrind and visualized in kcachegrind.

```
==536531==   total heap usage: 2,046 allocs, 2,046 frees, 40,960 bytes allocated
==536531==
==536531== All heap blocks were freed -- no leaks are possible
==536531==
==536531== For lists of detected and suppressed errors, rerun with: -s
==536531== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 2 from 2)
```

We see that we have no memory bugs. Now we run the cachegrind profiler to do an experiment similar to perf.

```
$ valgrind --tool=cachegrind ./mergesort-demo
$ kcachegrind cachegrind.out.*
```

The output of cachegrind will be written in a `cachegrind.out.PID` file. This can then be analyzed, e.g., with the textual tool `cg_annotate`. However, for this kind of analysis we rather prefer an interactive graphical tool like kcachegrind. In fig. 2.2 we see a screenshot of the result of cachegrind. Form the left pane we learn that 34 % of instructions we are in `merge()` and 22 % in `merge_sort()`. Also memory management – namely `_int_free()`, `malloc()` and `free()` – sum up to 20 %. In the right pane we can zoom into individual calls from the left. Here we see that in `merge_sort()` we pay most of the time in copying over to the two sub-arrays.

Now let us use the callgrind tool, which gives us a hierarchical view on the execution of the program:

```
$ valgrind --tool=callgrind ./mergesort-demo
$ kcachegrind callgrind.out.*
```

In fig. 2.3 we see a screenshot of the callgraph as visalized by kcachegrind. In particular, we see how the most relevant functions are called from each other, how often they are called and what their absolute costs are.

2.3 Making code faster

We now have the tools to profile and measure the performance of our code. In a next step we will discuss a couple of techniques to make code faster.

2.3.1 Algorithm techniques

Following wisdom 1, we first concentrate on the underlying algorithm. There is no general silver bullet that makes algorithms faster. Designing algorithms is not unlike proving theorems in mathematics – it is mostly the result of creativity and experience. However, in both cases, there are a few standard techniques that reoccur.

Divide and conquer. This is powerful technique to devise efficient algorithms. The abstract idea is the following: Divide your problem of size n into subproblems of smaller size, like $\frac{n}{2}$, solve those smaller problem instances recursively, and then resemble the solutions of of the subproblems to form the solution of the original problem instance. So the general pattern looks like this:

1: **procedure** DIVIDE_AND_CONQUER(problem)
2: $p_1, \ldots, p_k \leftarrow$ DIVIDE(problem)
3: $s_1, \ldots, s_k \leftarrow$ DIVIDE_AND_CONQUER(p_1), …, DIVIDE_AND_CONQUER(p_k)
4: **return** MERGE($s_1, \ldots, s_k$)

Often the division step or the merge step are both are simple and can be done efficiently. If division and merge takes in total $O(n)$ time and we have two subproblems of size $\frac{n}{2}$ then we end up with a time complexity of $O(n \log n)$. Merge sort, see algorithm 3, is one example. The fast fourier transform (FFT) algorithm is another one. Quick sort is another example, but if the pivot is chosen poorly then the subproblems are not of equal size, and then quick sort can degrade to a $\Theta(n^2)$ complexity. The binary search algorithm, see exercise 1.1, is another interesting example. Here we divide the problem only into one sub-problem of size $\frac{n}{2}$ and there is no real merge step, leading to an $O(\log n)$ complexity.

Due to their recursive reduction of the problem size, divide and conquer strategies also go naturally with parallelization and cache friendliness.

Choosing the right data structure. Many algorithms use various data structures to solve a specific task. For instance, the Dijsktra algorithm for shortest paths in graphs uses a priority queue to determine the next node to visit in the graph, see algorithm 9. The time complexity of this algorithm therefore depends on the time complexities of the operations on the priority queue.

A frequently reoccurring abstract data structure is a *key-value map*, also known as *dictionary* or *associative array*. Many programming languages have them also built-in, like `dict` in Python. It allows to store a value v associated with a key k and then lookup and remove elements by their key k. Mathematically, it resembles a map $K \to V$ with a key set K and a value set V. They have a large number of applications, like indices for databases, or the symbol table of a compiler or all kind of associations, caches or bookkeeping in a program. As a more or less random example, the popular web framework Django has 148 occurrences of `dict`.[20]

The two prominent data structures for key-value maps are *hash table* and *balanced binary tree*. Each one giving different average case and worst case time complexities for the different operations, see table 2.1.

Table 2.1: Time complexities of operations on hash tables and binary search trees.

Data structure	Insert		Lookup		Delete		Ordered
	avg	worst	avg	worst	avg	worst	
hash table	$O(1)$	$O(n)$	$O(1)$	$O(n)$	$O(1)$	$O(n)$	no
balanced binary tree	$O(\log n)$		$O(\log n)$		$O(\log n)$		yes

In practice, we typically go for a hash table, unless we have specific reasons, e.g., leveraging the benefit of binary trees to have keys in order. In C++ this means, we choose `std::unordered_map` rather than `std::map`. In Python, a `dict` is implemented by hash table.[21] The worst case for hash tables occurs when the hash function is poorly chosen and leads to collisions. Note that if keys are defined over your own data type then you may have provided the hash function, so choose it wisely.

There is also a security remark about this: After 2000, so-called *algorithmic complexity attacks* became popular. These are denial-of-service attacks that exploit the worst case complexities of algorithms and data structures, e.g., by constructing input that provokes hash collisions in hash tables [12] in widely known implementations, like associative arrays in Perl. Those attacks have been demonstrated against Perl-based web applications or the HTTP proxy server Squid.

[20] `git clone --depth=1 https://github.com/django/django; grep "<dict(" django/**/*.py | wc -l`.

[21] Inofficially since Python 3.6 and officially since Python 3.7, a `dict` preserves order while still using a hash table. To be precise, it has a hash table to store the lookup index in a list of the key-value pairs.

However, the fastest key-value map is often the one that can be avoided. They are easily used through standard libraries or language features and it is easy to forget the hidden complexity. By giving the one or other usage a second thought, we may realize that we can store some key-value association by other means than an explicit map, like directly with the key. For instance, we can store weights of edges in a network graph by a map $E \to \mathbb{R}$ on the edge set E, or directly in the edge data structure without any map. In any case, recall wisdom 2 and profile your code.

Memoization and dynamic programming. For many algorithmic problems, we observe that the problem structure comprises subproblems of smaller size. The Dijkstra algorithm is such an example: If $v_0, \ldots, v_k$ is the shortest path from v_0 to v_k then $v_0, \ldots, v_j$ is the shortest path from v_0 to v_j for all $0 \leq j \leq k$. This sub-problem insight is exploited by *Dijkstra's algorithm*. We will discuss this in more detail in section 4.2.3. It sounds unimpressively simple, but it has powerful consequences and is exploited by an algorithm technique called *dynamic programming*.[22]

Let us consider another example, namely computing the Fibonacci numbers F_n, which are recursively defined as $F_0 = 0$, $F_1 = 1$ and $F_n = F_{n-1} + F_{n-2}$ for $n \geq 2$. The Fibonacci sequence $F_0, \ldots$ then results in $0, 1, 1, 2, 3, 5, 8, 13, \ldots$. The naive recursive implementation is given in algorithm 5.

Algorithm 5 Naive recursive implementation of the Fibonacci numbers.

```
procedure FIB(n)
    if n ≤ 1 then
        return n
    return FIB(n − 1) + FIB(n − 2)
```

Note that the time complexity $T(n)$ of `fib(n)` follows the same recursion as F_n itself, namely $T(n) = T(n-1) + T(n-2) + O(n)$, leading to a time complexity of $\Theta(F_n)$. Since $F_n \sim g^n/\sqrt{5}$, where $g = {}^{1+\sqrt{5}}/_2 \approx 1.618$ denotes the golden ratio, the time complexity is exponential. This is also visualized in the tree of recursions in fig. 2.4.

Following the idea of dynamic programming, we observe that fig. 2.4 is highly redundant in terms of subproblems: We keep computing the same `fib(n)` over and over again. We can avoid this by applying a technique called *memoization*: Instead of recomputing numbers, we store and reuse already computed results of subproblems. As a result, we effectively prune the recursion tree, leaving only $\Theta(n)$ recursive calls, as shown in fig. 2.4.

In Python there is actually a very handy decorator mechanism to turn precedures into memoized versions of the procedure. Applied to algorithm 5, this would look like listing 2.4. The `@cache` decorator uses a `dict` to cache the results. So the worst case complexity is $O(n^2)$ in the unlikely case of hash collisions. We can do better by realizing that we do not need this key-value map to cache the intermediates, but we can simply compute `fib(0)`, ... `fib(n)` in this order in $O(n)$ time, leading to algorithm 6.

Note that we now turned an exponential-time algorithm into a linear-time algorithm, and got rid of the recursion, too! Often this comes at a certain price, namely the space we pay for caching the intermediate results. In fact, it points to another pattern we sometimes observe in code optimization, namely the *time-space tradeoff*: We pay space to gain time, or vice versa. Only in the case of algorithm 6 and algorithm 5 we pay $\Theta(n)$ space for both.

[22] In fact, dynamic programming is also a technique to solve (dynamic) optimization problems, and the field of mathematical optimization is also called *mathematical proramming*, hence the term *dynamical programming*. A key concept here is the *Bellman equation*, which is also important in *control theory* and *reinforcment learning*. The Bellman equation, again, is the source of the concept known *curse of dimensionality*, which again is a often cited phrase concerning the immense volume of training data needed to train large neural nets, and "training" here means weight optimization. This illustrates how fundamental this concept is in mathematics, computer science and engineering.

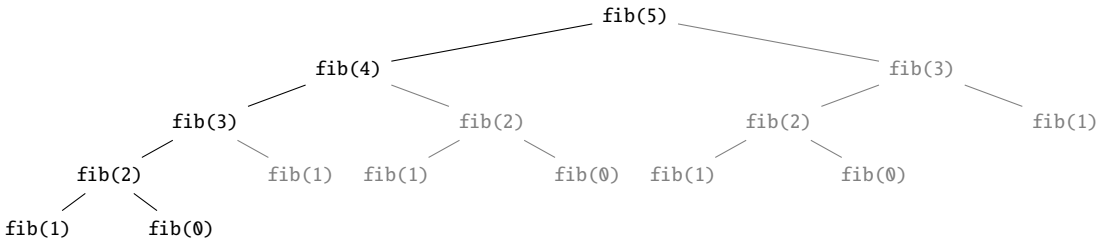


Figure 2.4: `fib(n)` causes $\Theta(F_n) = \Theta(2^n)$ calls of `fib()`. Memoization prunes the shaded calls, leaving only $\Theta(n)$ of them.

Listing 2.4: Memoized Fibonacci implementation Python taking $\Theta(n)$ time on average.

```
from functools import cache

@cache
def fib(n):
    if n < 2:
        return n
    return fib(n-1) + fib(n-2)
```

As a final remark, every computer scientist shall have heard about *Bloom filters*, or more modern variants like *cuckoo filters*. It is a probabilistic data structure for sets, but more space efficient than hash tables or binary trees. The lookup can have false positives, but no false negatives. They can be used to speed up key-value maps to make lookups for inexistent keys faster. Applications would be spell checkers or caching systems (e.g., for content delivery networks).

2.3.2 Memory management

Dynamic memory management is expensive. The C functions `malloc()` and `free()` or the C++ operators `new` and `delete` are costly. While they should in general be considered banned for real-time critical or embedded code[23], not only for real-time reasons[24], there is a significant price to pay. Also keep in mind that if your programming language provides *garbage collection*

[23] You may use them in the setup phase of your program, but not in the cyclic operation of your application.

[24] The TLSF [49] library provides memory management with $O(1)$ costs, yet the constants are significant. On x86, a `malloc()` costs up to 168 processor instructions.

Algorithm 6 Memoized Fibonacci algorithm with worst case $O(n)$ time complexity.

procedure FIB(n)
    $V \leftarrow$ array of length $n + 1$
    $V[0] \leftarrow 1$, $V[1] \leftarrow 1$
    for $i = 2$ to n **do**
        $V[i] \leftarrow V[i-1] + V[i-2]$
    return $V[n]$

than there is still memory management, albeit hidden behind abstractions. Furthermore, the allocated memory is allocated somewhere on the heap, which typically does not facility *cache locality*.

Often we can achieve good performance improvements by getting rid of memory allocations. We can demonstrate this on our merge sort implementation from listing 2.3. In our profiling excursion in section 2.2.3 we learned that `malloc()` and `free()` took 20 % of the instructions according to callgrind. Compare the original version in listing 2.5 and an improved version in listing 2.6, in which we (i) halved the number of allocations (and deallocations) and (ii) changed the manual array copying to a call to `memcpy()`, which can be highly optimized[25].

Callgrind reported 886 500 instructions for the original version and 603 340 instructions for the improved version. This is a speedup[26] of about 1.47, i.e., the new version is 47 % faster. In fact, the new version is faster, shorter and easier to maintain than the original[27] one.

Listing 2.5: Orginal from listing 2.3.

```
/** A merge sort on the array 'arr' of
    given size n. */
void merge_sort(int *arr, int n) {
  if (n <= 1)
    return;

  int mid = n / 2;
  int *left = malloc(mid * sizeof(int));
  int *right = malloc((n - mid) * sizeof(
      int));

  for (int i = 0; i < mid; i++)
    left[i] = arr[i];
  for (int i = mid; i < n; i++)
    right[i - mid] = arr[i];

  merge_sort(left, mid);
  merge_sort(right, n - mid);
  merge(arr, left, mid, right, n - mid);

  free(left);
  free(right);
}
```

Listing 2.6: Less memory allocations.

```
/** A merge sort on the array 'arr' of
    given size n. */
void merge_sort(int *arr, int n) {
  if (n <= 1)
    return;

  int mid = n / 2;
  int *copy = malloc(n * sizeof(int));
  memcpy(copy, arr, n * sizeof(int));

  int *const left = copy;
  int *const right = copy + mid;
  merge_sort(left, mid);
  merge_sort(right, n - mid);
  merge(arr, left, mid, right, n - mid);

  free(copy);
}
```

But we can even do better and eliminate the memory allocations completely. We observe that in the recursion tree in fig. 1.2, we could do the merge sort from bottom up rather than recursively top down. That is, we first merge all the blocks of size 1 on the bottom level, then the blocks of size 2 on the next, then blocks of size 4, 8, 16 and so on in powers of two. This allows us to get rid of the memory allocation, except for one temporary array, which we fill up levelwise. See listing 2.7 for the implementation. Callgrind reports 400 578 instructions for this version. We summarized our improvements in table 2.2.

Function call costs. There is another performance benefit in the bottom-up mergesort: We got rid of recursion and hence eliminated the costs of function calls (including the stack management). The costs of a function call, however, can vary largely. It depends on the hardware

[25] For instance, fast implementations leverage vectorized SIMD operations. The compiler may recognize it as a built-in which enables certain optimizations, like inlining.

[26] The *speedup* is defined by the quotient of the old to the new runtime, which here is $\frac{886500}{603340}$.

[27] Which was mostly generated by GitHub's Copilot.

Table 2.2: Speedup of different merge sort implementations.

implementation	instructions	speedup
original	886 500	
less malloc	603 340	1.47
bottom-up	400 578	2.21

platform, the application binary interface (ABI) (concerning argument passing), the programming language and the compiler. Typically the costs comprise register saving, argument and return value passing, the actual call to the function, stack frame setup in the function preamble, and maybe exception handling. But modern processors and compilers optimized a lot for these costs. In C++ and since C99[28] we may make use of function inlining, however doing it too aggressively can have the opposite effect. Polymorphic functions, on the other hand, can increase the costs further, e.g., in C++ the vtable is involved to determine the address of the virtual function. In C++, Java and since Python, we have zero-cost exception handling, yet in C there is no concept of exceptions.

Security remark. A little security remark on the way: All of our merge sort implementations have security issues. The `malloc()` function can possibly be attacked by integer overflow attacks. We could have used `calloc()` instead, but in any case we would need to check the return value of those. Beware of code in textbooks for production use. It might have been written in a way to improve a didactical aspect and not in terms of the best way for industrial usage.

2.3.3 Cache optimization

In reference to fig. 2.1, in order to design fast implementations of algorithms, we need to know a few things about the architecture of computers. One inconvenient truth is that fast memory is expensive and therefore small, and, vice versa, large memory is slow. We cannot have both. The fastest memory we have are the registers of the processors, which we can accesss in a single clock tick. The slowest is the main memory (aka, random access memory (RAM)), ignoring the even slower hard disk.

Memory hierarchy and cache organization. The idea is to avoid the costs of higher access times to RAM by adding caches with lower access times. By doing so, we exploit the *principle of locality*, which says that if a memory location is accessed then it is likely that nearby memory locations will be accessed soon.[29] And if the gap between access times is very large, we may add multiple levels of caches.

This leads to the so-called *memory hierarchy*, which is a hierarchy of access times. Contemporary multi-core processors typically comprise a three-level cache hierarchy, as illustrated in fig. 2.5, with L1 and L2 caches local to a single core and a L3 cache shared by all cores. [30] Also in a *von Neumann architecture* we may suffer from the *von Neumann bottleneck*: We lack concurrent access to data and instruction memory on the same memory bus. Splitting up the L1 cache in

[28] Non-standard extensions supported it earlier, like gnu89, but details in semantics vary.

[29] Same holds true for all communication systems, including distributed systems, and therefore we have caches in web browsers, DNS servers, content delivery systems, hard disks, virtual file systems, and so on.

[30] The Linux command `lscpu` prints details on the processor caches.

Listing 2.7: Bottom-up merge sort without memory allocations.

```
/** A bottom-up merge sort on the array 'arr' of given size n. */
void merge_sort(int *arr, int n) {
  int *tmp = malloc(n * sizeof(int));
  /* Merge in blocks of given width, in increasing powers of two */
  for (int width = 1; width < n; width *= 2) {
    /* Merge arr[start:mid] and arr[mid:end] and store at tmp[start:end] */
    for (int i = 0; i < n; i += 2 * width) {
      const int start = i;
      const int mid = MIN(i + width, n);
      const int end = MIN(i + 2 * width, n);

      int left = start;
      int right = mid;
      int k = i;
      while (left < mid && right < end) {
        if (arr[left] < arr[right])
          tmp[k++] = arr[left++];
        else
          tmp[k++] = arr[right++];
      }
      while (left < mid)
        tmp[k++] = arr[left++];
      while (right < end)
        tmp[k++] = arr[right++];
    }
    /* This level in the "recursion tree" is done, copy over for next level. */
    memcpy(arr, tmp, n * sizeof(int));
  }
  free(tmp);
}
```

an *instruction cache* L1i and *data cache* L1d is a Harvard-like[31] mitigation.

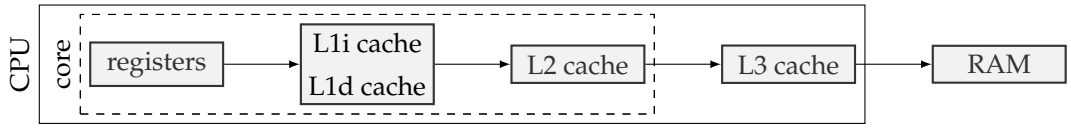
The caches are organized in *cache lines* of fixed size[32], like 64 B. A *cache hit* means that we attempt to access a memory location contained in a cache line, otherwise we speak of a *cache miss*. The *replacement policy* determines when a cache line is "evicted" from the cache and replaced by another one upon a cache miss. A popular strategy following the principle of locality is least recently used (LRU), which evicts the least recently used cache line. This policy also facilitates another optimization principle, namely putting *focus on the common case*.

The entire hierarchy then works as follows: Say we want to access a certain memory location. If it is in the L1 cache we have a cache hit in L1 and then we access the respective cache line rather than paying the costs down the memory hierarchy. Otherwise we may have a cache hit in the larger L2, which is only a bit slower. And so on. Only if we also have a cache miss in the last level cache (LLC), we have to pay the access time to RAM. This is a very brief and simplified summary. For a more detailed discussion, we refer to [27].

Cache friendliness. Memory access is fast if we have many cache hits. Vice versa, an implementation will be slow if the principle of locality is frequently violated. For instance, if

[31] A Harvard architecture has separate memory (busses) to program and data, while in a von Neumann architecture the differentiation of memory between program and data is a dynamic interpretation. This forms a security hazard when an attacker can inject code and alter the control flow, e.g., through stack-overflow attacks.

[32] In Linux, the command `getconf -a` gives information about the cache line sizes of the different caches.



	300 ps 1 kB	1 ns 2× 64 kB	3 ns 512 kB	10 ns 4 MB	100 ns 32 GB
Intel Core i5-6300:		2× 64 kB	512 kB	3 MB	
Intel Xeon E-2176G:		2× 192 kB	1.5 MB	12 MB	
Cortex-A72 (RPi4):		48 kB 32 kB	1 MB	none	

Figure 2.5: Typical memory hierarchy of a computer with a multi-core processor. At the bottom access times and typical sizes are displayed, along with two concrete Intel processors and the ARM processor on the Raspberry Pi 4 Model B.

we perform access to memory locations at random positions then we provoke cache misses with high probability in all cache levels. Something similar happens when we access memory locations on the heap, with addresses organized by `malloc()`, say, when we traverse the nodes of a linked list or a binary tree. In contrast, if we access memory locations in an array then we linearily scan through the memory, which facilitates the principle of locality. We say that the array is *cache friendly*.

Note that the factors between access times of L1 and RAM can be in the order of 100. As a result, in practice an array might outperform other data structures up to a significant input size, even if the time complexity might be worse. Keep this in mind when reciting wisdom 1. Sometimes we can combine cache friendliness and good time complexity. For instance, B-trees generalize binary trees to many children, where we can choose the size of nodes to fit natuarlly to the size of cache lines. There are even data structures that fit to different cache hierarchies, so-called *cache-oblivious data structures*, like cache-oblivious B-trees [5].

Case study: Matrix multiplication. Let us do a simple case study on cache friendliness by multiplying two $n \times n$ matrices. In listing 2.8 we have a basic implementation which literally implements the definition of matrix multiplication. Here a matrix is encoded as a one-dimensional array of size n^2 as a row-major matrix, i.e., the concatenation of rows. We can make an important observation already at this point: Iterating over a column in a row-major matrix is cache unfriendly; the entire matrix would need to fit into the cache! The macro `INDEX(i, j, n)` translates the two-dimensional index (i, j) to the one-dimensional array index $i \cdot n + j$.

Listing 2.8: Basic matrix multiplication.

```
/** Performs matrix multiplication c = a*b on n-by-n matrices a, b and c. */
void matmult_basic(double *c, double *a, double *b, size_t n) {
  for (size_t i = 0; i < n; i++) {
    for (size_t j = 0; j < n; j++) {
      c[INDEX(i, j, n)] = 0.0;
      for (size_t k = 0; k < n; k++)
        c[INDEX(i, j, n)] += a[INDEX(i, k, n)] * b[INDEX(k, j, n)];
    }
```

```
    }
  }
```

We test this implementation for different n and on two different processors. Time has been measured with `clock_gettime()` on `CLOCK_PROCESS_CPUTIME_ID`. The results are given in table 2.3 and in the following we will present two improvements over the basic version.[33]

First, we can identify a quick win in listing 2.8: In the inner loop over `k` we iteratively access the same memory location `c[INDEX(i, j, n)]`, which we could factor out from the loop. This is done in listing 2.9. Obviously this code must be faster. But remember wisdom 3! So we have a look at table 2.3. Indeed, this optimization only proves effective until $n = 512$ and then the effect vanishes, and on the Intel Core i5-6300U for $n = 2048$ a speedup of 0.96 is actually a slow down.

Listing 2.9: Matrix multiplication with less memory access.

```
/** matmult() with reduced memory access. */
void matmult_lessmemaccess(double *c, double *a, double *b, size_t n) {
  for (size_t i = 0; i < n; i++) {
    for (size_t j = 0; j < n; j++) {
      double sum = 0.0;
      for (size_t k = 0; k < n; k++)
        sum += a[INDEX(i, k, n)] * b[INDEX(k, j, n)];

      c[INDEX(i, j, n)] = sum;
    }
  }
}
```

In the next step we systematically exploit our knowledge of the caching hierarchy. If we consider the access patterns in the matrices then we observe that we scan along entire rows and along entire columns of matrices. This is not very cache friendly when a row does not fit into a cache line. And a column of a matrix in a row-major format only fits into a cache line if the entire matrix would fit. In listing 2.10 we address both issues as follows:

- The column-wise scanning in the inner loop happens for matrix `b`. But if we transpose `b` then we switch from column-wise scanning to a row-wise scanning. So we first create a transposed copy `bt` and hope[34] that the costs pay off.
- We do not scan entire rows but only parts that would fit into a cache line. So the original three loops in listing 2.8 are replaced by corresponding loops that jump in steps of the block size and then in inner loops we only iterate within a block, leading to six nested loops. Figure 2.6 illustrates the memory scanning pattern.

The runtime experiments in table 2.3 tell us that these optimizations clearly pay off, with speedups varying 2.77 and 15.67. The high variance in the speedup actually deserves some closer look. Algorithm analysis on listings 2.8 to 2.10 tells us that the complexity is clearly $\Theta(n^3)$. Hence, when doubling the input size n, we expect the runtime to be eightfold. And indeed, the progression of the runtime for the Intel Core i5-6300U for the "cache friendly"

[33] It is also important to note that we compiled with `-O2`, because effectively leveraging the memory hierarchy only becomes dominant when the processor pipeline is well utilized and the memory bandwidth starts to become a bottleneck rather than the number crunching power. In general, the dynamics of the memory hierarchy are difficult to predict and the numbers difficult to reproduce.

[34] Of course not, we measure and compare to check for our hypothesis!

Table 2.3: Absolute times and speedups of different matrix multiplication implementations.

n	implementation	Intel Core i5-6300U		Intel Xeon E-2176G	
		time in s	speedup	time in s	speedup
256	basic	0.054		0.025	
	less mem access	0.034	1.58	0.020	1.25
	cache friendly	0.012	4.50	0.009	2.77
512	basic	0.365		0.258	
	less mem access	0.273	1.33	0.167	1.54
	cache friendly	0.101	3.61	0.068	3.79
1024	basic	12.257		1.954	
	less mem access	12.272	0.99	1.889	1.03
	cache friendly	0.782	15.67	0.517	3.77
2048	basic	98.802		51.365	
	less mem access	102.498	0.96	49.920	1.02
	cache friendly	6.322	15.62	4.217	12.18

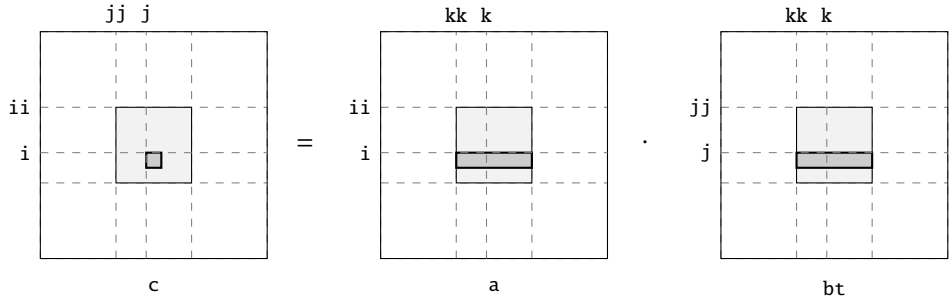


Figure 2.6: The blockwise memory scanning scheme in the matrix multiplication implemented in listing 2.10. The blocks are shaded in light gray. In dark gray we shaded the elements involved in the inner loop performing the calculation $c_{i,j} = \sum_k a_{i,k} \cdot b_{k,j} = \sum_k a_{i,k} \cdot b^\dagger_{j,k}$. This inner loop iterates over rows in a sub-block of a and $b^\dagger$, each fitting into a cache line.

Listing 2.10: A cache friendly matrix multiplication by row-major scanning and memory access fitting to cache lines as illustrated in fig. 2.6.

```
#define CACHE_LINE_SIZE 64
#define BLOCK_SIZE (CACHE_LINE_SIZE / sizeof(double))

/** Reimplementing matmult() with blockwise execution and row-major scanning. */
void matmult_cachefriendly(double *c, double *a, double *b, size_t n) {
  memset(c, 0, n * n * sizeof(double));

  /* Transpose matrix b*/
  double *bt = malloc(n * n * sizeof(double));
  for (size_t i = 0; i < n; i++)
    for (size_t j = 0; j < n; j++)
      bt[INDEX(i, j, n)] = b[INDEX(j, i, n)];

  /* c[i, j] = sum_k a[i, j] * bt[j, k] with 0 <= i, j, k <= n, iterating in
   * blocks of size BLOCK_SIZE. */
  for (size_t ii = 0; ii < n; ii += BLOCK_SIZE) {
    const size_t maxi = MIN(ii + BLOCK_SIZE, n);
    for (size_t jj = 0; jj < n; jj += BLOCK_SIZE) {
      const size_t maxj = MIN(jj + BLOCK_SIZE, n);
      for (size_t kk = 0; kk < n; kk += BLOCK_SIZE) {
        const size_t maxk = MIN(kk + BLOCK_SIZE, n);

        for (size_t i = ii; i < maxi; i++) {
          for (size_t j = jj; j < maxj; j++) {
            double sum = 0.0;
            for (size_t k = kk; k < maxk; k++)
              sum += a[INDEX(i, k, n)] * bt[INDEX(j, k, n)];
            c[INDEX(i, j, n)] += sum;
          }
        }
      }
    }
  }

  free(bt);
}
```

implementation is 0.012, 0.101, 0.782, 6.322, and the factors between them are 8.4, 7.7, 8.1, which are reasonably close to eight. But for the "basic" implementation the runtime progression is is 0.054, 0.365, 12.257, 98.802, so the factors in between are 6.8, 33.6, 8.1. We see that the cache friendly version sticks much better to the prediction according to the algorithm analysis.

The "basic" implementation suffers from large jump in runtime from $n = 512$ to $n = 1024$ for the Intel Core i5-6300U with a runtime factor of 33.6. The same happens for Intel Xeon E-2176G, but from $n = 1024$ to $n = 2048$, where the factor is 26.3. The memory hierarchy in fig. 2.5 can explain this when looking at the LLC: The Intel Core i5-6300U has a L3 cache of 3 MB and the Intel Xeon E-2176G has a L3 cache of 12 MB. A single matrix has a size of `n*n*sizeof(double)` bytes, which for $n = 256, 512, 1024, 2048$ gives 0.5 MB, 2 MB, 8 MB, 32 MB, respectively. In the "basic" implementation, the memory is primarily scanned over a single row of matrix `a` and over columns of matrix `b` in the inner loop. So a single row and a whole matrix should fit into the LLC for the inner loop to perform well. This is the case until $n = 512$ for the Intel Core i5-6300U and until $n = 1024$ for the Intel Xeon E-2176G.

We can confirm this hypothesis by measuring the number of cache hits and misses using perf, as illustrated in the below listing. First and second experiment are run with the "basic" implementation with $n = 512$ and $n = 1024$. As expected, the number of instructions becomes eightfold in the second experiment. However, the number of cycles increases from 1.1 Million to 17.9 Million, which is 16.3-fold. The explanation lies in the ratio between LLC-misses to LLC-loads: In the first experiment is is 0.97 %, so almost all memory accesses are cache hits. In the second experiment it is 87.54 %, so the vast majority of all memory accesses are cache misses, and we pay the full price of RAM access.

```
$ uname -p
Intel(R) Core(TM) i5-6300U CPU @ 2.40GHz

$ perf stat -e task-clock,cycles,instructions,LLC-loads,LLC-load-misses ./matrix-
    mult -n 512 -m BASIC
[...]
            405.30 msec task-clock:u              #    0.998 CPUs utilized
     1,128,363,203      cycles:u                  #    2.784 GHz
     1,110,336,833      instructions:u            #    0.98  insn per cycle
       142,960,281      LLC-loads:u               #  352.724 M/sec
         1,381,923      LLC-load-misses:u         #    0.97% of all L1-icache accesses

$ perf stat -e task-clock,cycles,instructions,LLC-loads,LLC-load-misses ./matrix-
    mult -n 1024 -m BASIC
[...]
         11,697.05 msec task-clock:u              #    0.997 CPUs utilized
    17,853,348,693      cycles:u                  #    1.526 GHz
     8,735,810,296      instructions:u            #    0.49  insn per cycle
     1,151,753,045      LLC-loads:u               #   98.465 M/sec
     1,008,276,730      LLC-load-misses:          #   87.54% of all L1-icache accesses

$ perf stat -e task-clock,cycles,instructions,LLC-loads,LLC-load-misses ./matrix-
    mult -n 1024 -m CACHEFRIENDLY
[...]
            844.88 msec task-clock:u              #    0.999 CPUs utilized
     2,391,123,626      cycles:u                  #    2.830 GHz
     8,264,332,712      instructions:u            #    3.46  insn per cycle
         1,515,549      LLC-loads:u               #    1.794 M/sec
         1,248,162      LLC-load-misses:          #   82.36% of all L1-icache accesses
```

In the third experiment with the "cache friendly" implementation the ratio is actually also quite low, but the absolute numbers of loads is only 1.5 Million, which is a factor of 760 less than in the second experiment! This is the effect of the cache friendly implementation and leads

to 3.46 instructions per CPU cycle in the cache friendly version opposed to the 0.49 instructions per CPU cycle in the basic version. That is, we leverage the superscalar processor design only in the cache friendly version.

As a final remark, there is a monograph by Ulrich Drepper[35] that every computer scientist should know, namely *What Every Programmer Should Know About Memory* [17].

2.3.4 Vectorization and parallelization

In order to reduce the wall clock time of a program, we may also improve the ratio of data processed per CPU cycle. This can in general be done in three different ways:

- In *instruction-level parallelism*, we exploit the fact that modern superscalar processors can execute multiple instructions in parallel by issuing them to different execution unit in the processor pipeline. Different techniques attempt to increase the number of instructions per cycles, like *pipelining*, *out-of-order execution*, *speculative execution*, and *branch prediction*.

- In *data-level parallelism*, we exploit the fact that modern processors have vector units, which can execute the same instruction on multiple data in parallel, using so-called *single instruction, multiple data (SIMD)* instructions. The masters of this kind of parallelism are graphics processing units (GPUs), which is a key ingredient for the success of deep learning.

- In *thread-level parallelism*, we exploit the fact that modern processors have multiple cores, which can execute different instructions in parallel.

We briefly touched instruction-level parallelism in the previous section. In the following, we will briefly discuss data-level and thread-level parallelism.

SIMD and vectorization. Many, many engineering problems can be phrased by means of linear algebra.[36] This is why computer hardware is optimized to perform linear algebra operations fast, like multiplying a matrix with a matrix or a matrix with a vector. A prominent building block of all these operations is the evaluation of sum expressions of the form $\sum_{i=1}^{n} a_i \cdot b_i$. SIMD operations would evaluate such expressions for some fixed n – like $a_1b_1 + a_2b_2 + a_3b_3 + a_4b_4$ – in a single instruction. We we do not only use SIMD for linear algebra, but exploit the data-parallelism for all kind of tasks, like speeding up `memcpy()` or sorting.

For x86 processors the era of SIMD instructions started with the multi-media extensions (MMX) in 1996, followed by streaming SIMD extensions (SSE) in 1999 and advanced vector extensions (AVX) in 2011, in various versions.[37] While MMX was integer only, SSE would also support floating point numbers.

The following "hello world" example illustrates the basic idea. On the left we start with a classical implementation of a dot-product of two vectors `a` and `b`. Below that we perform a fourfold *loop unrolling*. This is an optimization technique that reduces the loop overhead, which may also be performed by the compiler, but we pay with four times the code size of the body. In the right example we then replace the four multiplications by single SIMD operation `_mm_mul_ps()`. We applied a so-called *vectorization*.

[35] Drepper was the maintainer of glibc between 2001 and 2012 and is affiliated at Red Hat.
[36] This is reflected by the name of the famous software Matlab, which stands for "matrix laboratory".
[37] The Intel Core i5-6300U supports mmx, sse, sse2, ssse3, sse4.1, sse4.2, avx and avx2 according to `lscpu`

```
/* Basic */
for (int i = 0; i < n; ++i)
  c[i] = a[i] * b[i];
```

```
/* Loop unrolled */
for (int i = 0; i < n - 3; i += 4) {
  c[i+0] = a[i+0] * b[i+0];
  c[i+1] = a[i+1] * b[i+1];
  c[i+2] = a[i+2] * b[i+2];
  c[i+3] = a[i+3] * b[i+3];
}
```

```
/* Vectorized */
for (int i = 0; i < n - 3; i += 4) {
  /* Loading vectors from memory.
     Needs to be 16-byte aligned.*/
  __m128 va = _mm_load_ps(a+i);
  __m128 vb = _mm_load_ps(b+i);
  /* Apply the mm (multimedia)
     mul (multiplication) on
     ps (packed single-precision) */
  __m128 vc = _mm_mul_ps(va, vb);
  _mm_store_ps(c+i, vc);
}
```

In this example, the `ps` means packed single-precision float. A float has 32 bit, so four of them require 128 bit. Hence, SSE comes with 128 bit registers, called `__m128`. If we would have used double-precision floating-point numbers, we would have needed 256 bit registers, which were introduced with SSE2. Those type and function definitions come from a header file called `immintrin.h`.

In a benchmark we measured the time for the three implementations with $n = 2^{17}$ and compiler optimization `-O2`. So one vector would have 512 kB and three vectors would fit into the LLC on the test machine. Hence, the memory access should have little impact on the benchmark. In table 2.4 we summarized the results. In order to obtain more reliable runtime measurements, the dot product was computed 1000 times and time was averaged. Time was measured by `clock_gettime()` on `CLOCK_PROCESS_CPUTIME_ID`.[38]

Table 2.4: Speedup of different dot-product implementations.

implementation	time in s	speedup
basic	146.4	
loop unrolled	90.7	1.6
vectorized	44.7	3.3

Parallel computing. Since 2005, multi-core processors are also wide spread in consumer hardware. We can leverage parallel computing hardware by usual multi-threadig programming techniques, like POSIX threads. However, on a more fine-grained level we may also parallelize single loops by technologies like OpenMP. This is as easy as putting `#pragma omp parallel for` in front of a loop and compiling the code with the flag `-fopenmp`.

We can simplify illustrate this on our matrix multiplication example by listing 2.11. Likewise we can parallelize the two outer loops in listing 2.10. In table 2.5 we report on the runtime for $n = 512$, using the "basic" and "cache friendly" method with and without OpenMP. Time has now been measured with two clocks, `CLOCK_PROCESS_CPUTIME_ID` for the CPU time and `CLOCK_MONOTONIC` for the wallclock time on two different machines.

Listing 2.11: Basic matrix multiplication with OpenMP.

```
void matmult_basic_openmp(double *c, double *a, double *b, size_t n) {
```

[38] One needs to be quite careful in benchmarking. In order to avoid unintended function inlining or elimination of the 1000 repetitions by the compiler, the implementations have been compiled into a separate object file, so the compilation pass of the demo code could not derive those optimization opportunities.

```
  #pragma omp parallel for
  for (size_t i = 0; i < n; i++) {
    for (size_t j = 0; j < n; j++) {
      c[INDEX(i, j, n)] = 0.0;
      for (size_t k = 0; k < n; k++)
        c[INDEX(i, j, n)] += a[INDEX(i, k, n)] * b[INDEX(k, j, n)];
    }
  }
}
```

Table 2.5: Benchmark of parallelized matrix multiplication on two Intel machines. Speedup is computed on wallclock time.

machine	method	CPU time in s	wallclock in s	speedup
Core i5-6300	basic	0.385	0.360	
	basic w/ OpenMP	0.537	0.146	2.46
	cache friendly	0.102	0.104	
	cache friendly w/ OpenMP	0.174	0.045	2.31
Xeon E-2176G	basic	0.249	0.264	
	basic w/ OpenMP	0.458	0.040	6.60
	cache friendly	0.066	0.067	
	cache friendly w/ OpenMP	0.089	0.013	5.15

We see that the CPU time increases with OpenMP, which is attributed to the overhead due to the parallelization. However, this overhead pays off, because the wallclock time is reduced by OpenMP, with the speedup given in the last column. The Core i5-6300 posses two cores and four threads. The Xeon E-2176G has six cores and twelve threads. For both processors for each core there are two "virtual cores" allowing for the parallel scheduling of two software threads in the execution units of a single core. However, the effectiveness of this parallelization highly depends on the utilization pattern of the cores. We can therefore not expect a speedup of 4 on the Core i5 or 12 on the Xeon processor simply due to hardware limitations.

And then there is *Amdahl's law*. This law states that the speedup of a program is limited by the fraction of the program that cannot be parallelized. More precisely, if p is the fraction of the program that can be parallelized then $1 - p$ cannot be parallelized, so the speedup due to parallelization is limited by $\frac{1}{1-p}$, no matter how many threads can be executed in parallel. For instance, if $p = 0.9$ then the speedup is limited by 10.

In a next step, we can also discuss parallelization beyond a single computer, in a distributed system. The technology of choice is message passing interface (MPI), or OpenMPI, which we will not further discuss here.

Security remark. In 2018, we entered a new era of processor vulnerabilities exploiting many of the optimization techniques discussed in the previous section, such as *Meltdown* and *Spectre*. They are summarized as *transient execution CPU vulnerabilities* and are based on the speculative execution of modern processors with side channels formed in the caching hierarchy.

2.3.5 Optimizing compilers

Finally, the probably easiest way to make code faster is to tweak the compiler, at least if we use a programming language that is compiled to machine code. Modern C and C++ compilers come

with a rich battery of optimization techniques. However, a strong optimization level also comes at costs: Compilation times are increased, the code is harder to debug[39] and very aggressive optimization levels may even produce wrong output. This is way the optimization level can be chosen as typically set by a compiler flag `-O`.

For gcc – and likewise for other compilers – those levels are a follows:

- `-O0` means no optimization. Can be interesting for debugging or when inspecting the disassembly. This is the default for gcc.
- `-Og` means optimization for debugging. It is recommended for the usual edit-compile-debug cycle of software engineering.
- `-O1` means some optimization, but without taking too much compilation time.
- `-O2` means more optimization. It activates basically all optimizations that do not involve a speed-space tradeoff.
- `-O3` means even more optimization. This level is not always recommended, because it may lead to longer compilation times and the performance gain may not be worth it.
- `-Os` means optimize for size. It is like `-O2`, yet without those optimizations that increase size. This is interesting when the code is size critical, like in embedded systems or when the code is frequently loaded from disk. This optimization level may also be interesting when the code is memory bound, because smaller code may fit better into the caches.
- `-Ofast` means optimize for speed. In particular, it may disregard strict standard compliance, like for IEEE 754 floating point numbers.

These optimization levels are shortcuts for a battery of fine-grained optimization options. For instance, `-Ofast` includes `-ffasth-math`, which breaks strict IEEE 754 compliance.

Besides the compilation level, we also need to tell the compiler for what hardware platform it may generate code, e.g., whether it may make use of SSE4.1 or AVX2 instructions. And different machine instructions come at different costs for different processors. This is done by the `-march` flag. A special value here is `native`, which instructs the compiler to auto-detect the architecture of the build computer.

If we compile our dot-product example with `-Ofast -march=native` then we reduce the runtimes as in table 2.6. We see that through compiler optimizations we easily obtain the same performance as our hand-crafted vectorization, which is automatically done by the compiler, on the Intel Core i5-6300U.

Table 2.6: Speedup by tuned compiler flags for the dot-product example.

compiler flags	method	time in s	speedup
`-O2`	basic	94.568	
	vectorized	40.069	
`-Ofast -march=native`	basic	41.431	2.28
	vectorized	40.130	1.00

Compiler optimization primarily concerns the translation of source code files to translation units, i.e.g, the translation of `.c` files to `.o` files. However, the linker may also perform optimizations, which is called *link-time optimization*.

[39] In a debugger the we may observe code reordered for a better pipeline utilization or a better translation to machine instructions or dead code being eliminated.

2.4 Summary

In this chapter we discussed the performance optimization of code. We switched out perspective from algorithm analysis in chapter 1 to measuring and improving the performance of implementations. Three general wisdoms reoccured in this chapter: (i) first optimize the algorithm, (ii) avoid premature optimization but measure to identify the hot spots and (iii) never guess the effectiveness of a seemingly more efficient code. We learned about different techniques and technologies to measure code concerning runtime and memory footprint. We learned about different clocks an when to use which. We learned about profiling techniques, the sampling profiler perf and the simulatioin-based profiler Valgrind and analysed merge sort with them in detail. Finally, we learned about techniques to make code faster. We started with algorithm techniques, namely divide and conquer, the role of the right data structure, and memoization and dynamic programming. Secondly, we discussed the costs of memory management and demonstrated on merge sort how to speed up an implementation by avoiding memory allocations. Then we discussed the role of the memory hierarchy and how to make code cache friendly. We discussed the role of vectorization and parallelization and how to leverage them. Finally, we discussed the role of the compiler and how to tune it.

All the programming techniques combined can yield speedups up to an order of, say, 100, more or less. This effectively makes the difference between a processor from two or three decades ago and today, which is impressive.[40] However, these speedups are only $O(1)$ improvements. But improving the time complexity of an algorithm can make the difference between the blink of an eye and the age of the universe.[41]

2.5 Exercises

Exercise 2.1 (⋆⋆)**.** Implement a C routine `insertion_sort()` that takes a sequence of numbers, runs insertion sort, and returns an `uint64_t` with the nanoseconds spent for the insertion, using the method in listing 2.1.

- Use `CLOCK_PROCESS_CPUTIME_ID` as clock to measure time.
- Plot the runtime in a over the size n of the input sequence.
- Run the above experiment with two types of input arrays: Sequences with some random numbers and sequences with already sorted numbers.

Exercise 2.2 (⋆⋆)**.** Tools like `stress` and `stress-ng` deliberately generate CPU load. Runtime measurements using `clock_gettime()` are influenced under such load. Design an experiment that illustrates the different influence for the clocks `CLOCK_PROCESS_CPUTIME_ID` and `CLOCK_MONOTONIC`.

You may reuse your solution of exercise 2.1 for this exercise.

[40] It is very hard to make this statement precise. Million instructions per second (MIPS) are insufficient for this judgement, since caching, pipelining, et cetera have a major influence. The notion of BogoMips in the Linux kernel tries to approximate these effects by defining for different processors a factor to the clock rate to have a MIPS-like performance indicator. Use `grep bogomips /proc/cpuinfo` to find out the BogoMips on your system. Each core on my Intel Core i5-6300U from 2015 has 5000 BogoMips. An Intel 486 DX4 at 100 MHz from 1993 has 50 BogoMips, which is a factor of 100 less.

[41] The blink of an eye takes 100 ms. The age of the universe is 14 Gyr = 4.4×10^{17} s. The ratio is 4.4×10^{18}. Computing the Fibonacci number F_n with the naive recursive algorithm takes time proportional to 1.618^n, and about as many recursive calls, which for $n = 100$ is 7.9×10^{20}. Dynamic programming makes this linear, so 100 recursive calls; the factor in between is 7.9×10^{18}, which is almost twice as good as ratio of an eye blink to the age of the universe.

Exercise 2.3 (⋆)**.** The measurement technique of listing 2.1 comes with overheads, of course. Design an experiment that measures the mean and standard deviation of this method. Perform the experiment with two different clocks. Is a difference expected?

Exercise 2.4 (⋆)**.** Implement a C++ version demonstrating the measuring technique of listing 2.1 with different clocks.

Exercise 2.5 (⋆)**.** Implement a Python version demonstrating the measuring technique of listing 2.1 with different clocks.

Exercise 2.6 (⋆)**.** Do literature research on how time measurement works on Windows in the sense of listing 2.1. But use only high quality resources and rely on primary references to the best possible. (In particular, do not just take over solutions from the world wide web.)

Exercise 2.7 (⋆)**.** Write a demo in C that allocates memory at different sizes and use `getrusage()` to report the resulting RSS through `ru_maxrss`. Draw a plot of RSS over the total number of bytes allocated.

Exercise 2.8 (⋆)**.** Write a C routine `double* create_transposed(double* x, size_t n)` that takes an $n \times n$ matrix x and returns a transposed copy of x.

Details: The matrix x is given on row-major format, i.e., it is a concatenation of rows. The routine also allocates the required memory. Add demo code that calls the routine and profile it with the profiler perf. Have a look at the cache misses.

Exercise 2.9 (⋆)**.** Same as in exercise 2.8, but profile it with Valgrind using callgrind and cachegrind. (You may also check for the absence of memory leaks.)

Exercise 2.10 (⋆)**.** The precise formula for the Fibonacci number F_n is

$$F_n = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right).$$

Show that F_n is in $\Theta(g^n)$ with $g = \frac{1+\sqrt{5}}{2}$. Is F_n also in $O(2^n)$? Is it in $\Theta(2^n)$? Is it in $\Omega(1.5^n)$?

Exercise 2.11 (⋆)**.** Compare listing 2.4 with memoization against a version without memoization. Compute the sequence $F_1, \ldots, F_{100}$, measure the time for each F_n. You may abort once F_n takes longer than 1 min. Plot the runtime in a log-log plot.

Bonus: Fit a curve $c \cdot g^n$ to it, where $g = \frac{1+\sqrt{5}}{2}$ is the golden ratio.

Exercise 2.12 (⋆⋆)**.** Write a C++ program to compare the efficiency of `std::map` and `std::unordered_map`.

- Insert n elements in each and compare the time it took.
- Run the experiments with sorted numbers and random numbers.
- Plot the results.

Exercise 2.13 (⋆⋆)**.** Implement a recursive algorithm for the computation of binomial numbers with memoization. What is the time and space complexity?

Exercise 2.14 (⋆)**.** What is the space complexity of algorithm 6? Find an improved version.

Exercise 2.15 (⋆)**.** Measure the costs of `malloc()` and `free()` in a C program. (Or `new` and `delete` in C++.) To this end perform 10 000 allocations of random sizes up to 8 kB. And then perform the deallocations. Report the time for allocation and deallocation separately.

Exercise 2.16 (⋆)**.** Build a `std::map` with 1024 integer elements. Then investigate the memory addresses of the elements. What can we conclude regarding cache friendliness of `std::map`?

Exercise 2.17 (⋆)**.** Use perf to analyze L1d cache misses in the matrix multiplication examples listing 2.8 and listing 2.10.

Exercise 2.18 (⋆ ⋆ ⋆)**.** Repeat the cache optimization techniques for matrix multiplication, but exploit the size of all cache levels.

Exercise 2.19 (⋆⋆)**.** Repeat matrix multiplication experiments with `float`.

Exercise 2.20 (⋆⋆)**.** Consider two vectors $x, y \in \mathbb{R}^n$. The outer product $x \otimes y$ is defined as $x \cdot y^\dagger$ and forms the $n \times n$ matrix of all multiplications of elements of the vectors. (We interpret the vectors as column vectors.) Implement a basic version in C and a cache friendly version. Compare the performance.

Exercise 2.21 (⋆)**.** Implement the outer product as in exercise 2.20, but now investigate the influence of compiler optimization levels.

Exercise 2.22 (⋆)**.** Implement the outer product as in exercise 2.20, but now improve the wallclock time by parallelization using OpenMP.

Exercise 2.23 (⋆)**.** Speed up bottom up mergesort from listing 2.7 using parallelization via OpenMP.

Chapter 3

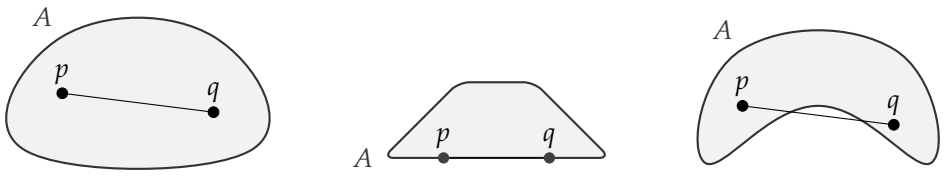
Convexity

Convexity is an important concept which we will encounter a couple of times throughout this book. In particular, it will help us to gain a geometric understanding of planar graphs and optimization, and a geometric understand of matters is typically very powerful. Convexity adds a lot of structure, which has the effect that mathematical properties become stronger if we add convexity, problems become easier when we add convexity and algorithms become faster when we add convexity. This observation has also been articulated by Peter Gruber in a standard book on this matter, see page VI in *Convex and Discrete Geometry* [22]. In this sense, convexity is also an entry point to discrete geometry and even has connections to cryptography, like post-quantum cryptographic schemes based on lattices, where the *shortest vector problem* and the (convex) fundamental polytope of lattices plays a crucial role.

We will give a brief introduction to convexity in this chapter in two steps: We first introduce convex sets and then convex functions.

3.1 Convex sets

Intuitively, a set is called *convex* if it has no "embayments". To make this more precise, we follow this approach: Any light ray enters and leaves a convex set only once. Or in other words, if we take two points within the convex set, then the convex set contains the entire line segment between these points. To make this more formal, we therefore need to first define what we mean by a line segment.



(a) (Strictly) convex set (b) (Non-strictly) convex set (c) Non-convex set

Figure 3.1: 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. If the points between p and q are in the interior of the set then it is even strictly convex.

3.1.1 Line segments and convexity

Let $p, q \in \mathbb{R}^d$ then we denote by $\overline{pq} = \{\lambda p + (1-\lambda)q \colon \lambda \in [0,1]\}$ the line segment between p and q. Here we interpret $\mathbb{R}^d$ as a vector space, i.e., we can add up and scale points in $\mathbb{R}^d$. 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 $\lambda = \frac{1}{2}$. This allows us to define what we mean for a set A to be convex:

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.

These definitions actually works for p, q in any vector space, but we focus on $\mathbb{R}^d$ here. In fig. 3.1 we illustrate three 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. Vaguely speaking, the right set is nonconvex as it has an "embayment". The left and middle sets are convex.

But the left one is in a certain sense particularly convex, in the following sense: In fig. 3.1a, the points in the middle of $\overline{pq}$ are not just in A, they are in the interior of A. In the middle figure fig. 3.1b this is not generally the case as it has a "flat" portion at the boundary, a portion of zero curvature. To make this precise, we first introduce the notion of interior and boundary of a set.

Definition 2. A point $x \in A$ is called an *interior point* of A contains an entire ball centered at x with sufficiently small radius. A point $x \in A$ is called a *boundary point* of A if every ball centered at x contains points in A and points not in A.

We denote by $\operatorname{int} A$ the *interior* of A, i.e., the set of all interior points of A and by $\operatorname{bd} A$ the *boundary* of A, i.e., the set of all boundary points of A. We can now define strictly convex as follows:

Definition 3. A set $A \subseteq \mathbb{R}^d$ is called *strictly convex* if $\overline{pq} \subseteq \operatorname{int} A$ for any two points $p, q \in A$, except possibly for p and q themselves.

A first simple statement on convex sets is the following:

Lemma 1. *The intersection $\bigcap_{k=1}^n A_k$ of (strictly) convex sets $A_1, \ldots, A_n$ is (strictly) convex.*

The proof is quite simple: Pick any p, q in the intersection. So for each A_k we have $p, q \in A_k$ and therefore $\overline{pq} \subseteq A_k$. Hence, $\overline{pq} \subseteq \bigcap_{k=1}^n A_k$. A similar argument holds for strict convexity and observing that $\operatorname{int} \bigcap_k A_k = \bigcap_k \operatorname{int} A_k$.

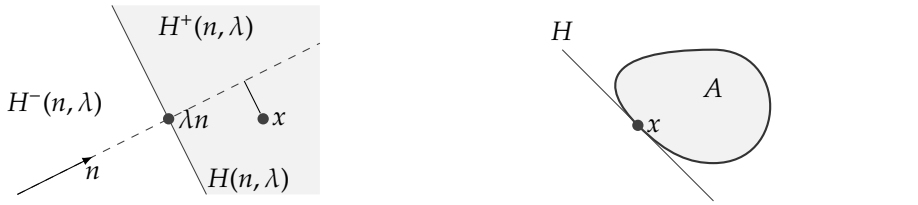
3.1.2 Halfspaces and polytopes

By a *hyperplane* in $\mathbb{R}^d$ we mean set of points $x \in \mathbb{R}^d$ that are orthogonal to a vector $n \in \mathbb{R}^d$ and at distance $\lambda \|n\|$ to the origin. We accordingly define $H(n, \lambda) = \{x \in \mathbb{R}^d \colon n \cdot x = \lambda\}$ as the set of points when orthogonally projected to a line spanned by n then the projection coincides with λn, see fig. 3.2a. A hyperplane in $\mathbb{R}^2$ is a straight line, in $\mathbb{R}^3$ it is a plane and in $\mathbb{R}^1$ it is a point. Furthermore, it is rather a matter of convenience to have $\|n\| = 1$ since $H(\mu n, \mu\lambda) = H(n, \lambda)$ for all $\mu \neq 0$. Only n shall not be the zero vector. But when $\|n\| = 1$ then λ is the distance of the hyperplane to the origin.

A hyperplane divides $\mathbb{R}^d$ into two so-called *halfspaces*. We call the one beyond λn the positive halfspace $H^+(n, \lambda) = \{x \in \mathbb{R}^d \colon n \cdot x \geq \lambda\}$ and the one behind λn the negative halfspace $H^-(n, \lambda) = \{x \in \mathbb{R}^d \colon n \cdot x \leq \lambda\}$, see fig. 3.2a.

We say that $H(n, \lambda)$, $H^+(n, \lambda)$ and $H^-(n, \lambda)$ are *supported* by the point λn. For this notion of a hyperplane and halfspaces, we have the following properties:

Lemma 2. *It holds that* $H(-n, -\lambda) = H(n, \lambda)$, $H^+(-n, -\lambda) = H^-(n, \lambda)$ *for all* $\lambda \in \mathbb{R}$ *and* $n \in \mathbb{R}^d$.



(a) Halfspace $H^+(n)$ with boundary $H(n)$ (b) Hyperplane supporting a convex set A at x

Figure 3.2: Halfspaces and supporting hyperplanes.

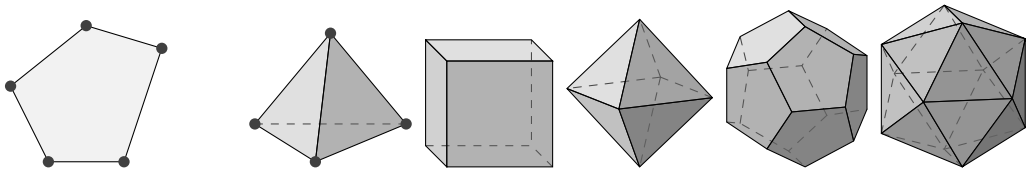
Consider a convex set A and a point x on the boundary $\operatorname{bd} A$ of A as illustrated in fig. 3.2b. Due to the convexity of A there is a hyperplane H that supports A at x, i.e., $x \in H$ and A is entirely on one side of H, meaning that A is contained in a halfspace having H as its boundary. We call these a *supporting hyperplane* respectively *supporting halfspace*. If the boundary $\operatorname{bd} A$ is not smooth at x then there in general many supporting halfspaces at x.

A machine learning technique called *support vector machine* follows roughly this idea or at least this notion, where the goal is to learn hyperplanes that separate two point clouds for the goal of binary classification.

A halfspace is itself a convex set. So the intersection of halfspaces is again a convex set by lemma 1. A triangle or a square in the plane are examples of intersections of halfspaces in $\mathbb{R}^2$. A tetrahedron or cube are examples in $\mathbb{R}^3$. However, note that the intersection of halfspaces may not be bounded, i.e., may be infinitely large. In particular, a single halfspace is a polyhedron. This leads to the following two definitions:

Definition 4. The intersection of finitely many halfspaces in $\mathbb{R}^d$ is called a convex *polyhedron*. A bounded polyhedron is called a *polytope*.

All five platonic solids are convex polytopes in $\mathbb{R}^3$, as shown in fig. 3.3b. The cube, for instance, can be formed by the intersection of six halfspaces. The dodecahedron can be formed by the intersection of 12 halfspaces. Convex polygons are convex polytopes in $\mathbb{R}^2$, see fig. 3.3.



(a) Convex polygon (b) Platonic solids: tetrahedron, cube, octahedron, dodecahedron, icosahedron

Figure 3.3: Convex polytopes in two dimensions (left) and three dimensions (right).

3.1.3 Convex combination and convex hull

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) $p_1, \ldots, p_n$ we call the

linear combination

$$\lambda_1 p_1 + \lambda_2 p_2 + \cdots + \lambda_n p_n = \sum_i \lambda_i p_i$$

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* of the points $p_1, \ldots, p_n$, see fig. 3.4.

Figure 3.4: 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 5. We define the *convex hull* $\text{conv}\{p_1, \ldots, p_n\}$ of n points $p_1, \ldots, p_n \in \mathbb{R}^d$ as the smallest[1] convex set that contains $p_1, \ldots, p_n$. More generally, for a set $A \subseteq \mathbb{R}^d$ we define $\text{conv}\, A$ as the smallest convex superset of A in $\mathbb{R}^d$.

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 $\text{conv}\{p_1, \ldots, p_n\}$ as in fig. 3.5.

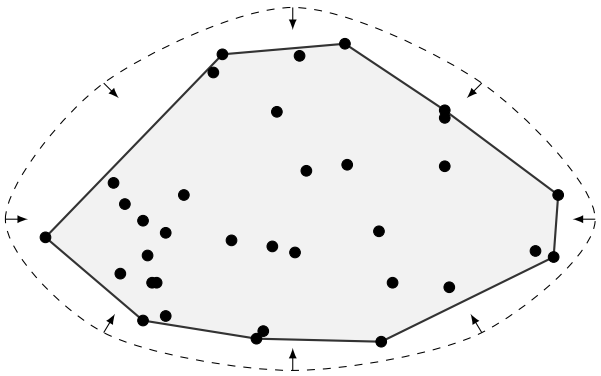


Figure 3.5: The convex hull of n input points in the plane forms a convex polygon. We can think of it as the resulting shape of a tight elastic rubber band (dashed line) around nails placed at the input points. Any triangle (dotted) formed by three input points is entirely contained in the convex hull.

Sometimes we have a set of points $p_1, \ldots, p_n$ that lie on the boundary of a convex set. We say that $p_1, \ldots, p_n$ *lie in convex position*. Using the notion of the convex we could also say that

[1] A set A is "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.

$p_1, \ldots, p_n$ lie in convex position if they are on the boundary of their convex hull. In other words, none of the points lies in the interior of the convex hull.

Let us consider the finite point set $p_1, \ldots, p_n$ in fig. 3.5. By definition $\text{conv}\{p_1, \ldots, p_n\}$ is the smallest convex superset of $\{p_1, \ldots, p_n\}$. However, it is also the set of all convex combinations of $p_1, \ldots, p_n$. In other words, in fig. 3.4 the line segment is equal $\text{conv}\{p, q\}$ and the triangle is equal $\text{conv}\{p, q, r\}$. In chapter 9 we will learn about algorithms to compute convex hulls. They have a ton of applications, where we name two: First, they can be used to accelerate many computational geometry problems, like collision detection. Secondly, convex hulls are used for shape estimation from point clouds, like in robotics or computer vision.

The notion of a convex hull allows us now to give a second definition of a convex polytope:

Lemma 3. *Given finitely many points in $\mathbb{R}^d$ then their convex hull forms a convex polytope. If the points lie in convex positions then they form the vertices of the polytope. Vice versa, given a convex polytope in $\mathbb{R}^d$, the convex hull of the vertices is the polytope itself.*

The polytope formed by convex hulls of finitely many points are sometimes called *V*-polytopes and those formed by the intersection of halfspaces are called *H*-polytopes. The lemma says that these are the same.[2]

When we speak of a convex polytope in $\mathbb{R}^d$ then we often silently assume that it is not contained in a hyperplane. We can make this more explicit by speaking of a n-dimensional convex polytope in $\mathbb{R}^d$ as the convex hull of finitely many points lying in a n-dimensional hyperplane, but not in a $(n-1)$-dimensional hyperplane (i.e., affine-linear subspace).

In the following it will be convenient to consider a convex polytope in $\mathbb{R}^0$ being a point.[3] Then we can observe that a convex polytope in $\mathbb{R}^3$ has a boundary that is made of convex polygons again, which are 2-dimensional convex polytopes. Their boundaries again are made of 1-dimensional convex polytopes, and so on. Like the boundary of a cube comprises squares, whose boundaries comprise straight-line edges, whose boundaries are points, see fig. 3.3b.

In general, the boundary of a d-dimensional convex polytope consists of $(d-1)$-dimensional convex polytopes, whose boundary therefore again consists of convex polytopes one dimension lower, and so forth, until we end up with vertices as polytopes of dimension 0. We will come back to convex polytope in $\mathbb{R}^3$ in chapter 4 when we talk about planar graphs. The concept of convex polytopes whose boundaries are made of convex polytopes of lower dimension will be important for linear optimization in later chapters.

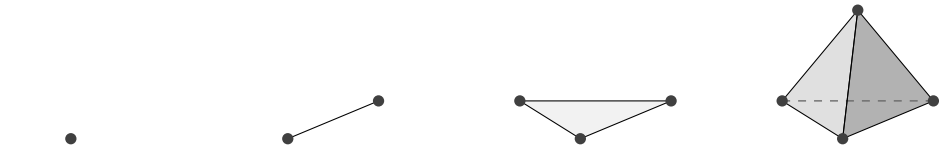
A very special convex polyhedron is a so-called simplex. It is in some sense the simplest convex polyhedron in the sense that it has a minimum number of vertices:

Definition 6. A d-dimensional *simplex* is the convex hull of $d+1$ points not lying on a common $(d-1)$-dimensional hyperplane.

The short name for a d-dimensional simplex is also d-simplex. A 2-simplex is a triangle, a 3-simplex is a tetrahedron, a 1-simplex is a straight edge, a 0-simplex is a point, see fig. 3.6. Simplices are very important for triangulations and meshes, like in geometric software or for all kind of finite-element simulations. Also, simplices correspond to the so-called *complete graphs* in chapter 4. And the so-called simplex algorithm in linear optimization is related to simplices.

[2] See more details in [22, p. 246].

[3] This is formally the case by noting that $\mathbb{R}^0$ is a set that contains the origin as its only point. As a vector space, it has a zero-sequence as basis, and hence 0 as dimension. But all axioms for the vector space hold, the only become trivial. But note that $\mathbb{R}^0$ is not the empty set!



(a) 0-simplex (b) 1-simplex (c) 2-simplex (d) 3-simplex

Figure 3.6: Simplices in various dimensions.

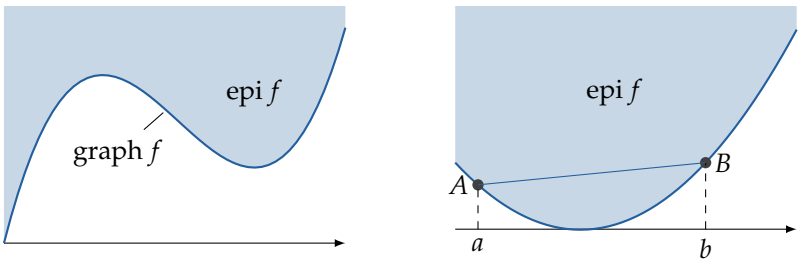
3.2 Convex functions

Besides convex sets, there is a second important notion of convexity, namely the convexity of functions. We can actually define convexity of functions based on convex sets by first introducing the notion of an epipgraph.

Definition 7. Let $X \subset \mathbb{R}^d$ and let $f\colon X \to \mathbb{R}$ be a function. Then we call the set graph $f = \{(x_1, \dots, x_d, y) \in \mathbb{R}^{d+1} \colon (x_1, \dots, x_d) \in X, y = f(x_1, \dots, x_d)\}$ the *graph* of f and we define the *epigraph* of f by epi $f = \{(x_1, \dots, x_d, y) \in \mathbb{R}^{d+1} \colon (x_1, \dots, x_d) \in X, y \geq f(x_1, \dots, x_d)\}$.

So the graph and the epigraph of a function $f\colon \mathbb{R}^d \to \mathbb{R}$ is a set of points in $\mathbb{R}^{d+1}$. The epigraph of f is the set of points on or above graph f, as illustrated in fig. 3.7a. Based on the notion of an epigraph we can now define convex functions as those whose epigraph is a convex set.

Definition 8. Let $C \subset \mathbb{R}^d$ be a convex set and let $f\colon C \to \mathbb{R}$ be a function. Then we call f *(strictly) convex* iff its epigraph epi f is a (strictly) convex set.



(a) Graph and the epigraph (b) A convex function

Figure 3.7: The graph and epigraph of a function $f\colon \mathbb{R} \to \mathbb{R}$ and a function that is strictly convex as its epigraph is strictly convex.

Directly following the definition of a convex function, we observe the following: Let $f\colon C \to \mathbb{R}$ be a convex function and let $a, b \in C$ be two points in C. Then we can consider the points $A = (a, f(a))$ and $B = (b, f(b))$ in epi $f \subset \mathbb{R}^{d+1}$, see fig. 3.7b. We can call $\overline{AB}$ a *secant* of graph f. As f is convex, also epi f is convex and hence the secant $\overline{AB}$ is contained in epi f by convexity. And also the converse is true: If all secants are in epi f then epi f is convex, and so is f. This is illustrated in fig. 3.7b. This gives us the following lemma:

Lemma 4. *A function $f\colon C \to \mathbb{R}$ is (strictly) convex iff for any $a, b \in C$ the respective secant of the function graph is (strictly) contained in* epi f.

Here we mean by "strictly contained" that the secant is in the interior of epi f except for the endpoints. In the special case where f is a differentiable function in one variable, we can even say a bit more about convexity. Consider fig. 3.7b and assume that A and B move closer and closer. As we keep progressing more and more portion of epi f is above the secant (or rather the line supporting the secant). In the limit the secant becomes the tangent at the limit point and the entire epi f is above the tangent. The slope of the tangent at a point x is given by $f'(x)$. In higher dimensions we have a tangent plane supporting epi f at $(x, f(x)$ and the derivative f' generalizes to the *Jacobi matrix*. (See section 8.2.3 for more details.) This is summarized by the following lemma:

Lemma 5. *A differentiable function $f : C \to \mathbb{R}$ is (strictly) convex iff* epi f *is (strictly) above the tangent hyperplane of the function graph at each $x \in C$.*

Here we mean by "strictly above" that the tangent intersects epi f only in one point. If f is twice differentiable then we can say the same by means of the second derivative:

Lemma 6. *A twice differentiable function $f : C \to \mathbb{R}$ in one variable is convex iff $f''(x) \geq 0$ for all $x \in C$ and strictly convex if $f''(x) > 0$ for all $x \in C$.*

Also this statement can be generalized to functions in higher dimensions by means of the Hessian matrix, i.e., $f : C \to \mathbb{R}$ is convex iff the Hessian matrix $H(x)$ is positive semi-definite at any $x \in C$. We will come back to this in chapter E when we talk about optimization.[4]

3.3 Summary

In this chapter we introduced the concept of convexity. We started with the definition of convex sets, for which we introduced the concept of convex combinations of points, where a line segment is a special case. We then discussed halfspaces whose intersections form polyhedra and polytopes. The Platonic solids are examples that display particular regularity and symmetry. Convex combinations of finite points sets form convex hulls, which are of particular importance in algorithm theory. The convex full of $d+1$ points in general position in $\mathbb{R}^d$ are called simplices, which play a fundamental role in optimization, computer-aided design and meshing, algebraic topology and topological data analysis. We concluded this chapter by introducing the concept of convex functions.

3.4 Exercises

Exercise 3.1 (⋆). Consider a quadrilateral polygon P in the plane, where vertices are given in order by $a, b, c, d \in \mathbb{R}^2$. Show that P is convex if $\overline{ac}$ intersects $\overline{bd}$. (Bonus: What if $\overline{ac}$ and $\overline{bd}$ do not intersect?)

Exercise 3.2 (⋆). We consider two convex polygons P and Q in the plane. Show that if $P \cap Q$ has a non-empty interior then $P \cap Q$ is a convex polygon.

Exercise 3.3 (⋆). What shape can $P \cap Q$ in exercise 3.2 have if the interior of the intersection is empty?

Exercise 3.4 (⋆). Let us say that P has n vertices and Q has m vertices in exercise 3.2. How many vertices may $P \cap Q$ have at least and at most?

[4] If the Hessian is always positive definite then f is strictly convex. But not vice versa, as $f(x) = x^4$ illustrates at $x = 0$, and the same holds for higher-dimensional examples $f(x_1, \ldots, x_d) = \sum_i x_i^4$.

Exercise 3.5 (★★). Consider a bounded convex set $C \subset \mathbb{R}^2$. We define the diameter $d_n(C)$ in direction $n \in \mathbb{R}^2$, with $\|n\| = 1$, as follows: Say $H(n, \lambda)$ and $H(n, \mu)$ support C for some $\lambda, \mu \in \mathbb{R}$ and $C \subset H^-(n, \lambda) \cap H^+(n, \mu)$ then $d_n(C) = \lambda - \mu$.

Prove that $d_n(C)$ is constant for all such n if C is a disk. However, is the converse true as well, i.e., if C has the same diameter in all directions is C then a disk?

Exercise 3.6 (★). Consider an axis-parallel rectangle with vertices $p_1 = (0, 1), p_2 = (2, 1), p_3 = (2, 0), p_4 = (0, 0) \in \mathbb{R}^2$ and some point $Q = \sum_{i=1}^{4} \lambda_i p_i$ as a convex combination of the four points. Sketch the point q for the following examples for $\lambda = (\lambda_1, \lambda_2, \lambda_3, \lambda_4)$:

1. $\lambda = (1, 0, 0, 0)$
2. $\lambda = (1/2, 1/2, 0, 0)$
3. $\lambda = (1/3, 1/3, 1/3, 0)$
4. $\lambda = (1/4, 1/4, 1/4, 1/4)$

And the same for the affine combination q with

1. $\lambda = (-0.5, 0, 1.5, 0)$
2. $\lambda = (-1, 0, 2, 0)$

(A linear combination $\sum_i \lambda_i p_i$ is called *affine combination* if $\sum_i \lambda_i = 1$.)

Exercise 3.7 (★). Let $f, g \colon C \to \mathbb{R}$ denote two convex functions on a convex set $C \subseteq \mathbb{R}^d$. Show that $f + g$ is convex.

Exercise 3.8 (★). Consider exercise 3.7 again and assume $f, g \colon \mathbb{R} \to \mathbb{R}$ are twice differentiable. Find a simpler proof.

Exercise 3.9 (★). Let $f \colon \mathbb{R} \to \mathbb{R}$ be a function of the form $f(x) = ax^2 + bx + c$. For which a, b, c is f convex and in which cases is it strictly convex?

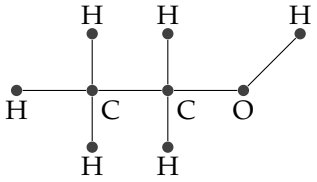
Exercise 3.10 (★). Let $f \colon \mathbb{R} \to \mathbb{R}$ be a function of the form $f(x) = ax^3 + bx^2 + cx + d$. For which a, b, c, d is f convex and in which cases is it strictly convex?

Chapter 4

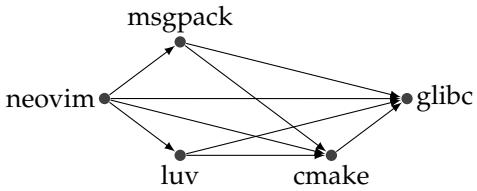
Graphs

Graphs are a fundamental concept in discrete mathematics and computer science. Typically, when we work on a some problem, we first need to translate it into a mathematical language. Graphs are a remarkably versatile tool for this purpose.

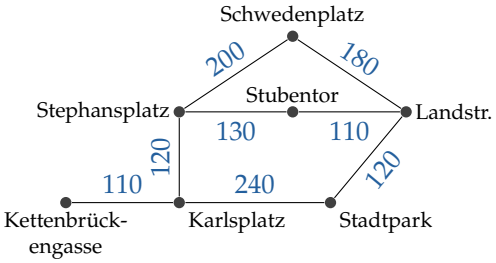
They can model networks, relations, dependencies, geometric objects, data structures, programs, neural nets, and much more. We use them to investigate the critical path in project management, model interactions in a social network, cybersecurity attacks in an attack graph, find shortest routes in road or communication networks, schedule tasks on resources, model the control flow of programs or the chemical structure of molecules, the kinematic chain of a robot, match stellar constellations, and the list goes on indefinitely. Also the callgraph in fig. 2.3 is a graph. Some examples are given in fig. 4.1.



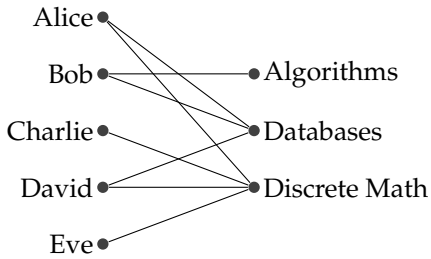
(a) Ethanol molecule C_2H_5OH



(b) Software dependencies of the neovim editor



(c) Subway network in Vienna



(d) Planing with students and courses

Figure 4.1: A couple of graphs related to different applications.

All have in common that they consist of *vertices* and *edges* between them. Sometimes the vertices have colors or labels, edges may have directions or weights, or some other kind of additional properties. The field of graph theory is vast and we will only touch the surface of it. For a more comprehensive treatment of graph theory we refer to the book by Diestel [15].

4.1 Introduction to graph theory

As always, we first need to formally define what a graph is so we can talk about properties of and algorithms and data structures for graphs.

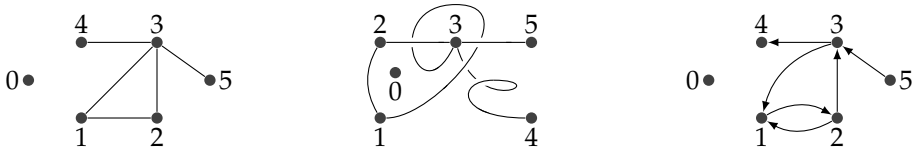
4.1.1 Undirected and directed graphs

A *graph* consists of a finite set V of vertices and a finite set E of edges, where an edge is a connection between two vertices $u, v \in V$. The vertices may also be called *nodes*, the edges are sometimes called *links* and more rarely *arcs*, mostly depending on the field of application. We distinguish between *undirected graphs* and *directed graphs*, depending on whether the edges have a direction or not. A directed graph is often also called *digraph*.

Undirected graphs have undirected edges $\{u, v\}$. Note that $\{u, v\} = \{v, u\}$ as orders of elements in sets are irrelevant. Directed graphs have directed edges (u, v). Note that $(u, v) \neq (v, u)$ when $u \neq v$, because pairs (as 2-tuples) are ordered. We formally define graphs as follows:

Definition 9. An (undirected) graph G is a pair (V, E) of a finite vertex set V and a finite edge set $E \subseteq \{\{u, v\} \colon u, v \in V\}$. A directed graph G is a pair (V, E) with a finite vertex set V and a finite edge set $E \subseteq \{(u, v) \colon u, v \in V\}$. In both cases we require $V \cap E = \emptyset$.

From now on, if we just say "graph", we mean undirected graph, unless we made it clear that we mean both. Graphs are typically illustrated by diagrams as in fig. 4.2, where directed edges are illustrated by arrows and undirected edges by lines. We call this the *drawing* of a graph. Note that the position of the vertices or geometry of the lines play no role for the graph; these are only details of one possible drawing. For the graph per se, we are purely interested in the combinatorial structure of the graph, i.e., whether vertices are connected to each other. In fig. 4.10a we see two different drawings of the same graph.

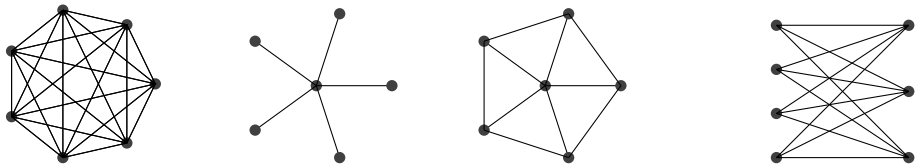


(a) Undirected graph (b) An uglier drawing of (a) (c) Directed graph

Figure 4.2: Drawings of an undirected graph and a directed graph.

Special graphs. Some graphs are frequently reoccurring in life, so we give them names. For instance, the *complete graph* K_n is a graph with n vertices and all possible edges, which are $\frac{n(n-1)}{2}$ many. Figure 4.1d shows a *bipartite graph*: We can partition V into two distinct sets $V = A \dot\cup B$ such that all edges go only between A to B. The *complete bipartite graph* $K_{m,n}$ is a bipartite graph with $|A| = m$ and $|B| = n$ and all possible edges between A and B, which are $m \cdot n$ many.

Other special graphs are regular graphs, the Petersen graph, star graphs, wheel graphs, and many more. There is a whole zoo of special graphs. See fig. 4.3 for some examples. Note that the *star graph* S_{n+1} could be defined as $K_{1,n}$ and the *wheel graph* W_n as a star graph S_n with a "cycle" around the leaves. Observe that $W_4 = K_4$. Note that K_n, S_n, W_n and the cycle graph C_n (see later) are defined to have n vertices and $K_{n,m}$ has $n + m$ vertices.



(a) Complete graph K_7 (b) Star graph S_6 (c) Wheel graph W_6 (d) Complete bipartite graph $K_{4,3}$

Figure 4.3: Some special graphs with their dedicated names and vertex labels omitted.

Graph isomorphism. In fig. 4.3 we omitted the vertex labels, because we were only interested in the structure of the graph. That is, we call *any* graph G with all edges present a complete graph, even if the vertex set is not $\{1, \dots, n\}$. To make this precise, we need a definition when we consider to graphs to be "the equivalent", which is the notion of *graph isomorphism*: We call two graphs $G = (V, E)$ and $G' = (V', E')$ isomorphic if there is a bijection $\phi\colon V \to V'$ such that $\{u, v\} \in E$ if and only if $\{\phi(u), \phi(v)\} \in E'$. We call ϕ the isomorphism. In other words, we can turn G into G' by only changing the vertex "names", as in fig. 4.4. A suitable isomorphism would be $\phi(1) = 1$, $\phi(2) = 3$, $\phi(3) = 5$, $\phi(4) = 2$, $\phi(5) = 4$.

An isomorphism $\varphi\colon G \to G$ is called an *graph automorphism*, and ϕ from before is an example. The algorithmic problem of testing whether G and G' are isomorphic is in NP. But there is a famous heuristic called the *Weisfeiler-Leman algorithm* that is fast in practice[1] yet a one-sided heuristic: It can provide certificates for non-isomorphism, but not for isomorphism. This test plays an important role for so-called *graph neural networks* and *graph kernels* in machine learning.

(a) The cycle graph C_5 (b) The pentagram graph

Figure 4.4: Two isomorophic graphs: We can turn the left into the right by renaming the vertices.

By the way, we call graph in fig. 4.4a a *cycle graph* C_n, where the $n \geq 3$ vertices are connected by n edges to form a single cycle. We will define cycles more formally in the next section. Then we can define $W_{n+1} = S_{n+1} \cup C_n$, where $G \cup G'$ denotes the *graph union* $(V \cup V', E \cup E')$, where $G = (V, E)$ and $G' = (V', E')$, if we define the vertex sets accordingly.

Loops and degrees. A *loop* is an edge from a vertex to itself. Often loops are excluded by definition and we speak of a *simple* graph to stress this point. If we take a digraph and forget about the edge directions then we obtain the *underlying (undirected) graph*. The underlying graph of the digraph in fig. 4.2c is the one in fig. 4.2a. Note that the two directed edges $(1, 2)$ and $(2, 1)$ become the one undirected edge $\{1, 2\} = \{2, 1\}$.

We say that an edge between u and v is *incident* to u and v. Vice versa, vertices are *adjacent* if there is an edge between them. We also say they are *neighbors* and the *neighborhood* $N(v)$ of a

[1] The k-dimensional Weisfeiler-Leman algorithm takes $O(n^{k+1} \log n)$ time. The larger k the stronger the heuristic.

vertex v is the set of all neighbors of v. For an undirected graph we call the number of edges incident to a vertex u the degree $d(u)$ of u. (However, for non-simple graphs, a loop is counted twice for the degree. So in some sense we count the "edge endpoints".) At least for simple graphs we therefore have $d(u) = |N(u)|$. A vertex v of degree $d(v) = 0$ is called *isolated*; it has an empty neighborhood. In fig. 4.2a the vertex 0 is isolated and the vertex 3 has degree 4. We call a graph with every vertex of degree r a *r-regular graph*. The complete graph K_n is a $(n-1)$-regular graph, $K_{n,n}$ is an n-regular graph, and C_n is a 2-regular graph.

The vertex degrees and the number of edges of undirected graphs are in a relationship: When we insert an edge between u and v then the degree of u and v has been incremented. That leads to the following lemma, also known as the *degree formula*:

Lemma 7. *For an undirected graph (V, E) it holds that $\sum_{v \in V} d(v) = 2|E|$.*

In fig. 4.2a the sum is 10 and also in fig. 4.4a. There is also a nice counting argument for this lemma: Consider placing a dot at each "end" of the edge. How many dots did we place? From one point of view, we place $2|E|$ dots because two dots per edge. On the other hand, at each vertex v we placed $d(v)$ dots, so $\sum_{v \in V} d(v)$ dots in total. (That is we a loop edge shall count twice for the definition of the degree.) We will use a similar argument later for triangulations in section 4.1.7.

From lemma 7 we also see that if we remove from $\sum_{v \in V} d(v)$ all vertices of even degree, then the sum is still even. What remains is an even sum of odd degrees. So the number of summands must be even. This is called the *handshaking lemma*:

Lemma 8 (Handshaking lemma)**.** *The number of odd-degree vertices in a graph is even.*

The handshaking lemma says that at a party the number of people, shaking hands an odd number of times, is even. This example also shows a typical way how graphs are used as a modeling language: There are objects that are in relation to each other and we model this as a graph with the objects being the vertices and the relation being an edge. If the relation has a "direction" then we have a digraph, like for dependencies of tasks in a project.

For directed graphs, the *indegree* $d^+(v)$ of a vertex v is the number of edges (u, v) pointing from some u to v and the *outdegree* $d^-(v)$ of a vertex v is the number of edges (v, u) pointing away from v to some u. We can then define $d(v) = d^+(v) + d^-(v)$ and again call a vertex v *isolated* when indegree and outdegree are both zero. In fig. 4.2c the vertex 0 is isolated and $d^+(1) = 2$ and $d^-(1) = 1$. Depending on the application it is sometimes convenient to call a vertex with outdegree zero a *sink* and a vertex with indegree zero a *source*.

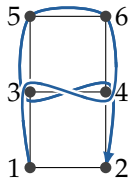
4.1.2 Paths, cycles and connectedness

Many algorithms and problems consider different ways to traverse a graph, i.e., to move from vertex to vertex in specific ways along edges. We will learn about such algorithms in section 4.2.2 and section 4.2.3. The following definitions introduce different notions of "wandering" around:

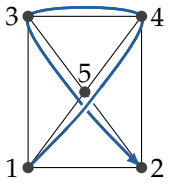
Definition 10. Let $G = (V, E)$ denote a undirected or directed graph. A *walk* from $v_0 \in V$ to $v_n \in V$ is a vertex sequence $(v_0, \ldots, v_n)$ such that for all $0 \leq i < n$ the edge from v_i to v_{i+1} is an edge in E. A *tour* is a walk with all edges being pairwise distinct. A *path* is a walk with all vertices being pairwise distinct, except maybe v_0 and v_n.[2] We call n the length of the walk, tour or path, respectively. A *closed* walk, tour or path goes form a vertex v_0 to the same vertex $v_n = v_0$.

[2] We translate walk, tour and path in German to Wanderung, Weg and Pfad. But unfortunately, usage of terms is not consistent in the literature.

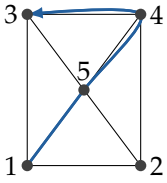
It will sometimes convenient to write $v_0 \rightsquigarrow v_n$ do denote a walk or tour or path. And if this goes via some vertex v_i then we may denote this by $v_0 \rightsquigarrow v_i \rightsquigarrow v_n$. For instance, we can say that a closed path is a path $v \rightsquigarrow v$. The above definition works for directed and undirected graphs. In the following we will discuss undirected graphs only. Remember that a tour does not visit an edge twice, just like a good motorcycle tour. And a path does not visit a place (vertex) twice, in the sense it does not cross itself. See fig. 4.5 for examples.



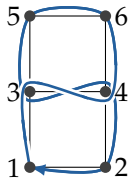
(a) Walk 1, 3, 4, 6, 5, 3, 4, 2



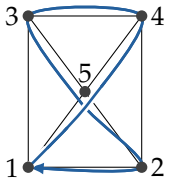
(b) Tour 1, 5, 4, 3, 5, 2



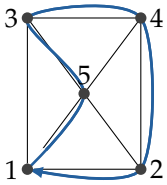
(c) Path, 1, 5, 4, 3



(d) Closed walk 1, 3, 4, 6, 5, 3, 4, 2, 1



(e) Closed tour 1, 5, 4, 3, 5, 2, 1



(f) Closed path 1, 5, 3, 4, 2, 1

Figure 4.5: Walks on a figure-eight graph and tours and paths on W_4.

By means of a walk we can define what we mean when a graph is *connected*: When we can go from any vertex to any other vertex by a walk. Actually, if there is a walk from a vertex u to a vertex v then there is also a path. Whenever the walk crosses itself at a vertex w then we can cut off the detour from w to w and obtain a shorter walk, until there are not any crossings anymore. That is, if there is a walk $u \rightsquigarrow w \rightsquigarrow w \rightsquigarrow v$ then there is also a walk $u \rightsquigarrow w \rightsquigarrow v$. We can repeat this argument until no vertex occurs twice, and then we have a path.

Definition 11. We call a graph *connected* if for all pairs of vertices u, v there is a path $u \rightsquigarrow v$.

The graph in fig. 4.2a is not connected; for instance there is no path $0 \rightsquigarrow 1$. However, if we would remove vertex 0 then the remaining graph is connected. Sometimes we need to refer to parts of a graph, which we call subgraph. Formally, a graph $G' = (V', E')$ is called a *subgraph* of a graph $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$.

A *connected component* of a graph is then a maximal connected subgraph. If we consider fig. 4.3 to show a single graph then it would possess four connected components, illustrated by the four subfigures.

A cycle is also a special subgraph of a graph; one that is "traced out" by a closed path:

Definition 12. A *cycle* of an undirected graph (V, E) is a subgraph (V', E') with a vertex set $V' = \{v_1, \ldots, v_n\}$ of n vertices such that $E' = \{\{v_1, v_2\}, \{v_2, v_3\} \ldots, \{v_{n-1}, v_n\}, \{v_n, v_1\}\}$. We call n the *length* of the cycle.

So a cycle graph C_n can be defined as a graph with n vertices that is itself a cycle in the above sense. Note that a cycle is very similar to but not the same as a closed path: A closed path has a definite start, but a cycle – as defined as a subgraph – has no such thing as a start. The graph in fig. 4.2a has only one cycle (V', E') with $V' = \{1, 2, 3\}$ and $E' = \{\{1, 2\}, \{2, 3\}, \{3, 1\}\}$. But this graph

has three closed paths, all tracing out the same cycle. Also note that we could have defined a cycle as a connected subgraph where every vertex has degree 2.

If the cycle happens to contain all vertices of the graph then we call it a *Hamiltonian cycle*. Testing whether a graph has a Hamiltonian cycle is one of the classical *NP*-complete problems in Karp's list so there is most likely no polynomial algorithm. (See section 1.2.3.)

4.1.3 Trees

Directed and undirected graphs that contain no cycle are called *acyclic*. The directed acyclic graph (DAG) has many important applications, like dependency graphs as in fig. 4.1b. We really would like to have no cycles in dependency graphs, like in project management. For every finite, partially ordered set there is a corresponding DAG and vice versa.

Undirected, acyclic graphs are called *forests*. The connected components of forests are called trees, i.e., they are connected and acyclic:

Definition 13. A *tree* is a connected, acyclic graph. Vertices of degree 1 are called *leafs*.

In fig. 4.6a we see a tree with t(h)ree leafs. A star graph S_n is a tree with $n-1$ leafs. However, the underlying undirected graph of a DAG is not necessarily a tree (or a forest), as fig. 4.1b shows.

(a) A tree (b) A rooted tree

Figure 4.6: Trees are connected acyclic graphs. Rooted trees have directed edges pointing away from the root (or vice versa).

A tree has the property that between any two vertices there is exactly one path: There must be at least one because it is connected, but there cannot be two: If there would be two then at some vertex these paths would diverge and rejoin later, and this would form a cycle. More precisely, assume that there are two different paths $u \rightsquigarrow v$, so there is a first vertex w after they diverge and a first vertex w' at which they rejoin, see fig. 4.7. So we have two paths $u \rightsquigarrow w \rightsquigarrow w' \rightsquigarrow v$. In particular, we have two paths $w \rightsquigarrow w'$ that only share the first and last vertex. So we can form a closed path $w \rightsquigarrow w' \rightsquigarrow w$ by reversing one of the paths, giving rise to a cycle. Then the graph cannot be acyclic.

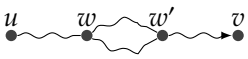


Figure 4.7: If a graph has two different paths $u \rightsquigarrow v$ then there is a cycle.

In a tree the number of edges is also given by the number of vertices:

Lemma 9. *For a tree* (V, E) *holds* $|E| = |V| - 1$.

A first simple proof is based on induction following this idea: We construct the tree iteratively by a sequence of trees $T_1, \ldots, T_n$, where T_k has k vertices. We start with T_1 having 1 vertex and

0 edges, where the property holds. In each step we let the tree T_k grow to T_{k+1} by adding one more vertex and an edge, each time keeping the property $|E| = |V| - 1$ invariant.

Another proof is based on *rooted trees*, where we turn the tree into a directed graph by choosing one vertex as the *root* and then orienting all edges in a way that they point away from the root, see fig. 4.6b. So in this rooted tree there is a single path from the root to any vertex; the paths lead way from the root. Observe that every vertex has in-degree 1, except the root.[3] So we have as many non-root vertex as edges, and hence $|E| = |V| - 1$.

A *binary tree* as data structure is in this sense a rooted tree where every vertex has a maximum out-degree of two, whereas *B-trees* have a higher out-degree.

Actually, depending on the application, we may also orient rooted trees the other way round, so all edges point toward the root.[4] Then the root is the single sink, instead of being the single source. In any case, all we need to specify is the root and therefore we often do not draw the edge directions for rooted trees, like in fig. 4.9.

We can actually turn every connected graph G into a tree by cleverly removing edges without destroying connectedness but achieving acyclicity: As long as there still exists a cycle we remove an arbitrary edge of the cycle, and repeat. Removing an edge of a cycle cannot destroy connectedness. What remains is a tree that "spans" the original G, which we call spanning tree:

Definition 14. We call a subgraph $G' = (V', E')$ of a graph (V, E) a *spanning subgraph* if G' is connected and $V' = V$.

Definition 15. A *spanning tree* of a graph is a spanning subgraph that is a tree.

Following this notation, we could have defined a Hamiltonian cycle as a spanning cycle. By definition, only connected graphs can have a spanning subgraph, let alone a spanning tree or a Hamiltonian cycle. Observe that a spanning tree T of a graph $G = (V, E)$ cannot have more edges than G. The following corollary must therefore hold, otherwise T cannot fulfill lemma 9:

Corollary 1. *Connected graphs* (V, E) *fulfill* $|E| \geq |V| - 1$.

In other words, for connected graphs we have $|E| \in \Omega(|V|)$. Spanning trees have numerous applications. One application is with communication networks and routing of packets, e.g., for the IP protocol. The communication nodes are linked together in a meshing network. To avoid routing loops, many protocols compute a spanning tree, most notably the Spanning Tree Protocol and variants of it. Many applications of spanning trees are actually using minimum spanning trees for weighted graphs, see section 4.1.5.

4.1.4 Euler tours

The discipline of graph theory surfaced through Leonhard Euler investigating the problem of the seven bridges of Königsberg, see fig. 4.8a. He was asked to find a route through Königsberg that would visit each bridge only once.[5] In the modern language of graph theory, we ask for a so-called Euleran tour:

Definition 16. An *Eulerian tour* of a graph is a tour that visits all edges.[6]

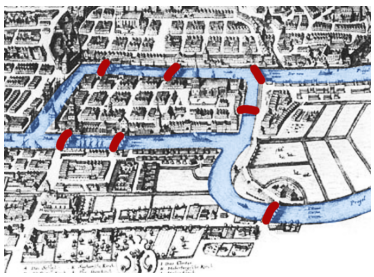
[3] And the leafs have out-degree 0.

[4] The saying "All roads lead to Rome" fits more naturally to this setting.

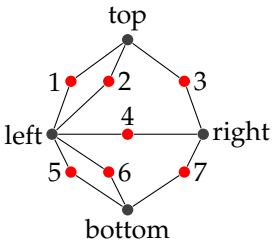
[5] The Königsberg bridge problem also marks the start of topology, which is in some sense geometry without precise lengths and locations of objects. The length of a bridge is irrelevant for this problem.

[6] Eulerian tours are often also called Eulerian paths in literature.

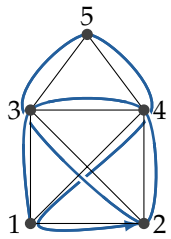
In fig. 4.8b we model the situation as a graph with every island and bridge being a vertex. Is there an Eulerian tour for this graph? The answer is "no". But how can we prove? An algorithmic approach would be to come up with an algorithm to exhaustively search for Eulerian tours and find that the algorithm would yield no solution. A better approach is to learn something about the problem structure itself.



(a) Seven bridges of Königsberg. Based on an image by Merian-Erben, public domain



(b) Graph of Königsberg



(c) A house in one stroke

Figure 4.8: The Königsberg bridge problem and Eulearian tours of graphs.

Say there would be an Eulerian tour $(v_0, \ldots, v_n)$ for a graph G. Then we observe that each vertex $v_1, \ldots, v_{n-1}$ is entered by one edge and exited by another. It may also be revisited later again, but for each visit we need two different edges. That means that the degree must be even. Only the degree of the first vertex v_1 and the last vertex v_n must be odd when the Eulerian tour is not closed, and it must be even if it is closed. This property is necessary.

It is also sufficient, because we can essentially construct an Euler tour using *Hierholzer's algorithm*: Assume all vertices have even degree. We start an arbitrary vertex v and traverse the graph: We arbitrarily follow edges we have not visited again. We cannot get stuck, since there is no vertex having only one unvisited edge incident as the degrees are even. At some point we end up in v again. So we have a tour $v \rightsquigarrow v$. However, there might vertices with yet unvisited edges. If so then such a vertex w must also lie on the tour $v \rightsquigarrow w \rightsquigarrow v$. But then we can recursively start the same procedure again from w, obtaining a tour $w \rightsquigarrow w$, which we can insert into the original, obtaining a larger tour $v \rightsquigarrow w \rightsquigarrow w \rightsquigarrow v$. We repeat this procedure until no vertex remains with unvisited edges incident. This algorithm is a lot like depth-first search (DFS) in section 4.2.2. If there are (two) vertices of odd degree then we have to start the procedure at one of them and the first tour will then end at the other odd-degree vertex.

Lemma 10. *A connected graph has a closed Eulerian tour if and only if all vertices are of even degree and a non-closed Eulerian tour if and only exactly two vertices are of odd degree.*

In fig. 4.8b all vertices corresponding to islands have odd degree, so there is no Eulerian tour. Another commonly known riddle is shown in fig. 4.8c, where we shall draw this house figure in one stroke without tracing a line twice. That is, we again ask for an Eulerian tour, which exists: The vertices 1 and 2 are of odd degree, so we start at either one. The shown solution is the tour $(1, 3, 4, 2, 3, 5, 4, 1, 2)$, but we can find other solutions by starting at any odd-degree vertex and simply continuing with any unvisited edge until we reach the other odd-degree vertex.

Riddles like in fig. 4.8 can be considered to be part of so-called *recreational mathematics*. But Eulerian tours also have industrial applications, like assembling DNA fragments, CMOS circuit design and unfolding surfaces of polytopes [13], e.g., for CAD/CAM applications.

4.1.5 Weighted graphs

The edges of a graph put vertices into relation.[7] Often these relations carry some quantity, i.e., the time to travel between cities, the bandwidth between network nodes, or the probabilities between events in a *Bayesian network*. We call this quantity the *weight* and formalize it by a function $w\colon E \to \mathbb{R}$ on the edge set E.

Definition 17. A *weighted graph* (G, w) is a pair of a graph $G = (V, E)$ and a weight function $w\colon E \to \mathbb{R}$. Likewise we define a *weighted digraph*.

Sometimes we want to stress the point that we put weights on the edges and speak of an *edge-weighted graph*. A weighted digraph is also called *network*. It is also convenient to write $w(u, v)$ instead of $w((u, v))$ when there is an edge from u to v. What follows holds for undirected and directed graphs likewise. We draw weighted graphs by labeling the edges by their weight, like in fig. 4.9. Depending on the application a weight of zero or infinity is often equivalent to an edge being absent.

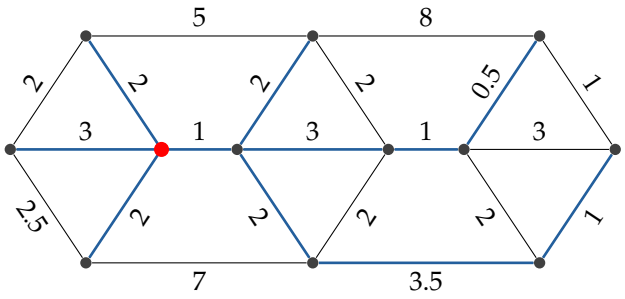


Figure 4.9: The rooted spanning tree of shortest paths created by Dijkstra's algorithm. The source is marked red.

Shortest paths. We can now take different notions we know for undirected and directed graphs and ask for minimum-weight variants of it. Most prominently, we might be interested in a walk from a vertex u to a vertex v of minimum total weight. More formally, for a walk $(v_0, \ldots, v_n)$ we call $\sum_{i=1}^{n} w(v_{i-1}, v_i)$ the *length* of the walk.

For most applications it makes sense to have the weights being non-negative. If weights can be negative then weight-minimizing instances, like walks, may not exist: If there is a cycle of negative total weight, then a weight-minimizing walk would keep spinning within this cycle indefinitely to add up negative lengths.

Let us assume weights are all positive. If there is walk between u and v then there is also a walk of minimum length. This minimum-length walk is actually a path: If the walk would revisit a vertex w then it cannot be of minimum length, because from $u \rightsquigarrow w \rightsquigarrow w \rightsquigarrow v$ we could cut off the detour from $w \rightsquigarrow w$ and reduce the length.[8] Hence, we call a minimum-length walk

[7] There is actually a one-to-one correspondence between diagraphs and relations. Formally, a relation R between a set A and a set B is a subset of the Cartesian product of A and B, i.e., $R \subseteq A \times B$. Hence, for a digraph (V, E) on a vertex set V, we can see $E \subseteq V \times V$ as a relation, and vice versa. Symmetric relations, i.e., $(a, b) \in R \Leftrightarrow (b, a) \in R$ for all $(a, b) \in A \times B$, correspond to undirected graphs.

[8] Note that we assumed positive weights, not just non-negative weights. If weights can be zero then there could be a cycle of zero total weight, and a shortest walk could revisit vertices.

between u and v the *shortest path*.[9] Furthermore, we call largest shortest path between any pair of vertices the *diameter* of the graph.

Shortest paths have many applications, not the least for vehicle routing in street networks. A well-known algorithm for the computation of shortest paths is the *Dijkstra algorithm*, see section 4.2.3.

Minimum spanning trees. For undirected graphs, also spanning trees can be considered in a weight-minimizing fashion: Given a weighted graph (G, w), the *weight* of a subgraph (V', E') of G is defined as its total weight $\sum_{e \in E'} w(e)$. Then we can define a *minimum spanning tree (MST)* of a weighted graph (G, w) as a spanning tree of G of minimum weight. There are many applications of minimum spanning trees, like hierarchical clustering, broadcasting in computer networks, image registration and segmentation and many more. Two well-known algorithms for the computation of minimum spanning trees are *Prim's algorithm* and *Kurskal's algorithm*, see section 4.2.4.

Traveling salesperson problem. A famous optimization problem in computer science is the *traveling salesperson problem (TSP)*.[10] A salesperson wants to visit every vertex in a graph exactly once in a closed path of minimum length. That is, TSP asks for the minimum-weight Hamiltonian cycle, which is again *NP*-complete.[11]

4.1.6 Planar graphs

The concrete drawing of a graph was irrelevant so far. That is, the positions of the vertices and the geometric shapes of the edges were irrelevant. In the following, we will subsequently add more structure concerning the drawing of graphs.

If we can draw a graph in the plane in a way such that edges do not cross then we call this a *planar graph* and such drawing is called a *planar embedding* of the graph into the plane. The complete graphs K_n are planar for $n \leq 4$ and are not planar for $n \geq 5$. Also $K_{3,3}$ is non-planar. In a certain sense K_5 and $K_{3,3}$ are the minimum examples.[12] In fig. 4.10a we see two planar embeddings of the complete graph K_4. Planar graphs and planar embeddings have many applications, e.g., for wire routing of a circuit in PCB layouting. Algorithms for testing for planarity is beyond the scope of this course.

Let us consider a *convex polytope* in $\mathbb{R}^3$, as defined in definition 4. The vertices and edges of the polytope form a graph, the so-called *edge graph* of the convex polytope. In fig. 4.10 we illustrate the edge graphs of the tetrahedron and the cube. One can prove that edge graphs of convex polytopes are planar.[13]

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. 4.10. The main theorem on planar graphs is given by *Euler's formula* for planar graphs[14]:

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

$$v - e + f = 2. \tag{4.1}$$

[9] In literature, often what we call a "walk" is called a "path", so double check the definition of a path.

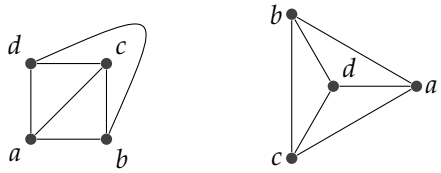
[10] It used to be called *traveling salesman problem*.

[11] Depending on the application, we may also be interested in a closed walk of minimum length rather than a path.

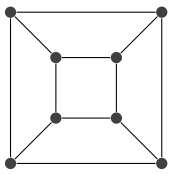
[12] *Wagner's theorem* says that a graph is planar if and only if it does not contain K_5 or $K_{3,3}$ as a minor.

[13] Actually, *Steinitz' theorem* says that a graph is the edge graph of a convex polytopes if and only if it is 3-connected and planar. That is, Steinitz' theorem completely characterizes the edge graphs of convex polytopes. A graph is *k-connected* when the removal of any k vertices keeps the graph connected.

[14] Dt. Euler's Polyedersatz.



(a) Two planar embeddings

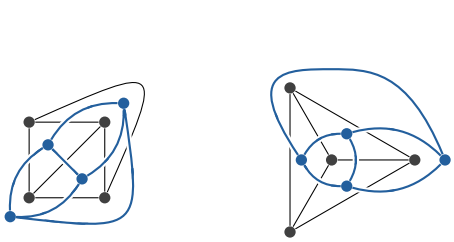


(b) Edge graph of the cube

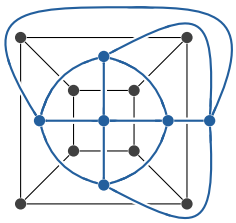
Figure 4.10: Planar graphs have a crossingfree drawing. Left: Two different embeddings of the same graph with four vertices a, b, c, d. It is the edge graph of the tetrahedron. Right: The edge graph of the cube with six quadrilateral faces.

The theorem is simple but deep. First, it holds for any planar drawing of a planar graph. That is, any drawing must have the same number f of faces. We shall also note that we also count the unbounded, infinitely large face as an face. It is called the *outer face*. In fig. 4.10a the outer face is a triangle in both cases and in fig. 4.10b 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 $4 - 6 + 4 = 2$.

In eq. (4.1) we see a certain symmetry concerning v and f. If they would switch roles, the formula would be the same. Indeed, from a planar embedding of a graph G, we can form a *dual graph* G' by turning vertices in faces and vice versa: Each face f of G gives rise to a vertex v' of G' and if two faces f, g in G share an edge then we place an edge in G' between the vertices forming the duals of f and g. In this sense, we can think of "flipping" the edges of G to form the edges of G'. The dual of the dual of G is G again.[15] In fig. 4.11 we illustrated the dual graphs of fig. 4.10. The dual of the cube graph is an octahedron graph. The dual of the tetrahedron graph is a tetrahedron graph; in this sense the tetrahedron is dual to itself.[16]



(a) Tetrahedron



(b) Cube

Figure 4.11: Dual graphs of (planar embeddings of) graphs in fig. 4.10a.

Dual graphs have many applications. One approach to compute unfoldings of the surfaces of polytopes is to search for certain spanning trees of the dual of the edge graph. In fig. 11.4 we have the Delaunay triangulation as the dual of the Voronoi diagram, so we can easily compute the one from the other. But also for a mathematical analysis this duality is very useful. For instance, the regular structure of the Delaunay triangulation translates to regular properties of

[15] This is an essential property of duality. For instance, we consider a point-line duality for the so-called Hough transform. Also here the dual of a dual is the original again.

[16] The dual of the icosahedron would be the dodecahedron. That is all five platonic solids have Platonic solids as duals.

the Voronoi diagram (and vice versa) through duality.[17]

4.1.7 Geometric graphs

For planar graphs it matters that there is a planar drawing, a planar embedding, but the drawing itself is not of a main interest. We can add more geometry to graphs and also fix the way we draw the graph and speak of *geometric graphs*.

Euclidean graph. A natural way to pull in geometry to graphs is by embedding vertices into the plane and for each edge between a vertex u and vertex v we assign a weight equal to the Euclidean distance $d(u, v)$ between the vertices. We call this a *Euclidean graph*. The same would work for an embedding in any metric space. The *complete Euclidean graph* is then the complete, Euclidean graph of a vertex set given by a finite point set $p_1, \ldots, p_n \in \mathbb{R}^2$. As the weights are given by the Euclidean distance, the weights need not be explicitly displayed in a figure but are given implicitly.

The *Euclidean minimum spanning tree (EMST)* of a finite point set $p_1, \ldots, p_n \in \mathbb{R}^n$ is the minimum spanning tree of the complete Euclidean graph. See fig. 4.12 for an example. Likewise the *Euclidean traveling salesperson problem (ETSP)* is then the traveling salesperson problem on the complete Euclidean graph. See fig. 4.13 for an example. We will revisit EMST and ETSP later in section 11.3.2 after we introduced Delaunay triangulations.

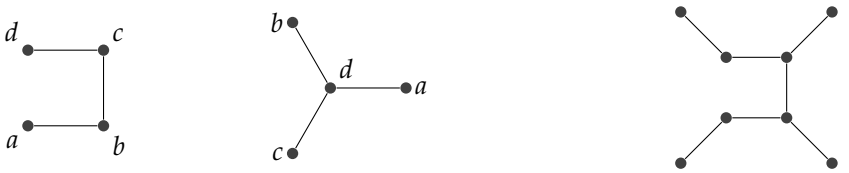


Figure 4.12: The Euclidean minimum spanning trees of the point sets in fig. 4.10.

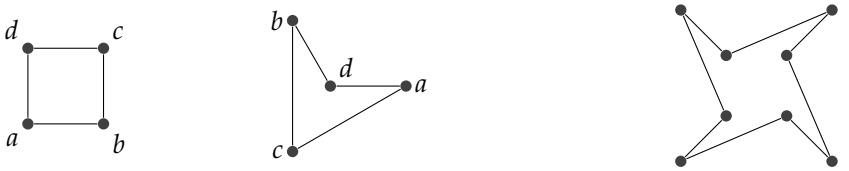


Figure 4.13: Solutions to the Euclidean TSP for the point sets in fig. 4.10.

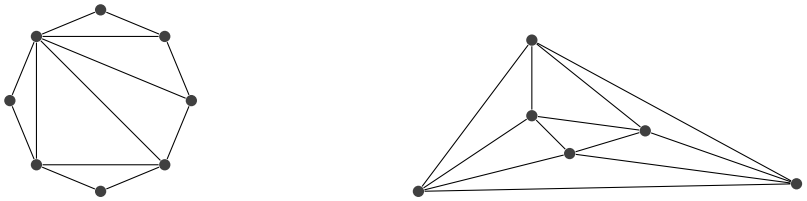
Planar straight-line graph. Another prominent geometric graph is the *planar straight-line graph (PSLG)*: We embed the vertices of the graph in the Euclidean plane, draw the edges as non-crossing straight-line segments. The PSLG as a weighted graph is an Euclidean graph, i.e., we assign the Euclidean lengths of the edges as weights. In fig. 4.10b we see a PSLG.

Given a set of points $S \in \mathbb{R}^2$, we define a *triangulation* of S as a maximal PSLG with the vertex set S. Here "maximal" refers to the edge set of the PSLG.[18] Note that S may possess many

[17] Triangles in the Delaunay triangulation correspond to degree-3 vertices in the Voronoi diagram. By enforcing a triangle as outer face as well, we have a strong combinatorial property through lemma 11.

[18] We could have also defined it as the maximal planar subdivision of the convex hull of S. We introduce the convex hull later in chapter 3.

triangulations.[19] The Delaunay triangulation is a special triangulation and will be introduced in section 11.1.2. See fig. 11.4 for an example.



(a) Tringulation of 8 points in convex position. (b) Triangulation with triangle as outer face.

Figure 4.14: Triangulations are maximal PSLGs. No edge can be added in a planar way.

All faces of a triangulation, except maybe the outer face, are triangles. Assume that also the outer face is a triangle. Then 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 $2e = 3f$. (We used a similar counting argument for the degree formula.) 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. 4.10a. However, since triangulating a non-triangular face only adds edges but no vertices, we obtain by eq. (4.1) for *any* planar graph the following:

Lemma 11. *For any connected, planar graph with $v \geq 3$ vertices and e edges we have $v - 1 \leq e \leq 3v - 6$, with the first equality for a tree and the second equality if all faces are triangles.*

(Recall that $e \geq v - 1$ by corollary 1.) In particular, 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 are faster on graphs with less edges. Hence, they perform better on planar graphs than arbitrary graphs. Prime examples are Dijkstra's algorithm in section 4.2.3 for shortest paths and Kruskal's algorithm in section 4.2.4 for minimum spanning trees.

Based on the above inequality we can also bound the number of faces in a connected, planar graph, as $f = e - v + 2 \leq 3v - 6 - v + 2 = 2v - 4$.

Corollary 2. *For any connected, planar graph G with $v \geq 3$ vertices and f faces we have $1 \leq f \leq 2v - 4$. Iff G is a tree we have $f = 1$ and iff all faces of G are triangles we have $f = 2v - 4$.*

If G is a tree then there are no cycles and the only face is the outer face. If the connected, planar G is not a tree then G contains a cycle and then we have $2 \leq f \leq 2v - 4$. Then both equalities hold when G is a single triangle, i.e, $2 = f = 2v - 4$. That is, G is a cycle and all faces are triangles.

[19] The number of triangulations is typically difficult to count. In the simple case of $n + 2$ points in convex positions, there are C_n many triangulations, where $C_n = \frac{1}{n+1}\binom{2n}{n}$ is the n-th Catalan number, which quickly becomes huge, i.e., $C_{10} = 16796$, $C_{20} = 6564120420$, $C_{30} = 3814986502092304$.

4.2 Graph algorithms

In the previous section we focused on the mathematical foundations of graphs. In this section we will introduced some basic graph algorithms. As a prerequisite, we also briefly need to introduce some data structures for representing graphs.

4.2.1 Representing graphs

There are two common ways to represent graphs: *adjacency lists* and *adjacency matrices*. For planar graphs, we also have the *doubly-connected edge list (DCEL)*, which requires the notion of faces. The choice of representation has an impact on the time and space complexity of algorithms.

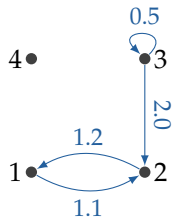
Adjacency matrix. Let us consider a directed or undirected graph $G = (V, E)$ with $V = \{1, \dots, n\}$ Then the *adjacency matrix* $A \in \mathbb{R}^{n \times n}$ is defined as

$$A_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E, \\ 0 & \text{otherwise.} \end{cases} \tag{4.2}$$

That is, the adjacency matrix simply encodes the presence or absence of an edge between all pairs of vertices. If G is undirected then A is symmetric. If we have a weighted graph (G, w) then we can generalize the adjacency matrix to

$$A_{ij} = \begin{cases} w(i, j) & \text{if } (i, j) \in E, \\ 0 & \text{otherwise.} \end{cases} \tag{4.3}$$

That is, the adjacency matrix is simply the matrix of all edge weights, with non-edges being treated as edges of weight zero. See fig. 4.15 for an example. Depending on the application it may also be useful to encode non-edges as ∞, like for the computation of shortest paths or MSTs.



$$\begin{pmatrix} 0 & 1.1 & 0 & 0 \\ 1.2 & 0 & 0 & 0 \\ 0 & 2.0 & 0.5 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

Figure 4.15: A weighted digraph and its adjacency matrix.

Still the question remains how to represent a matrix in a computer. We can of course use a two-dimensional array to do so. In a *row-major representation* we would represent the matrix as a single one-dimensional array by concatenating the rows. Similarly, we can use a *column-major representation* by concatenating the columns. All of those choices require $\Theta(|V|^2)$ space, independent of $|E|$. Note that $|E|$ could even be zero and for planar graphs we know that $|E| \in O(|V|)$.

Adjacency list. If the adjacency matrix A is sparse – i.e., if $|E| \ll |V|^2$ – then other representations are more space efficient, namely so-called *sparse matrix representations*. Take for instance a planar graph where $|E| \in O(|V|)$. A common example is the *compressed sparse row* format, which stores the non-zero entries of the matrix row-wise. Of course, the same can be done column-wise. This will take $\Theta(\max\{|E|, |V|\})$ space, which is more efficient if $|E| \in o(|V|^2)$.

Another name for this is the *adjacency list* representation. Here, for each vertex $v \in V$, we store a list of all adjacent vertices u, which is just what we described above from a matrix perspective. For a directed graph we may distinguish between outgoing edges (v, u) and ingoing edges (u, v), but these are implementation details.

Doubly-connected edge list. The *doubly-connected edge list (DCEL)* is a data structure for planar graphs, or rather for a planar drawing of the graph in the plane. It allows to easily traverse faces of the planar drawing. At each vertex v we consider the edges incident to v and store them in a bidirectional cyclic list in the same order as they appear around v. In particular, for each edge e incident to v, we store the next counter-clockwise (ccw) edge e' and the next clockwise (cw) edge e'' in the cyclic list, as illustrated in fig. 4.16. This allows to traverse entire faces in cw direction by iteratively asking for the next edge in ccw direction as shown in fig. 4.16.

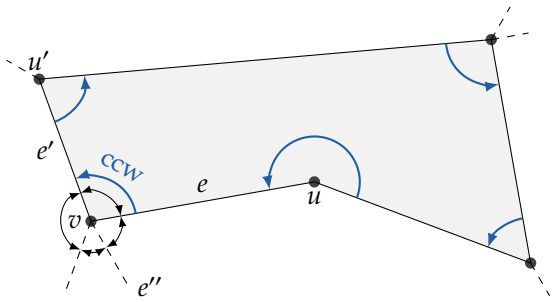


Figure 4.16: A planar drawing and its DCEL. Edges are stored in bidirectional circular lists around each vertex v. We can traverse a shaded face in cw direction by continuously taking the next ccw edge at each successive vertex.

In literature, such as [6], this data structure is also called *half-edge data structure*. The idea is to think of each edge e between two vertices v and u to actually consist of two "directed half-edges" $u \to v$ and $v \to u$. We then can speak of $v \to u'$ being the next ccw edge of $u \to v$ in fig. 4.16. This data structure is used in geometric software like CGAL [7], Vroni [31] or Stalgo [30].

4.2.2 Traversing graphs

The first graph algorithms we will discuss will be about *traversing graphs*. By traversing we mean a systematic way to visit all vertices of the graph. There are two common ways to do so: *depth-first search (DFS)* and *breadth-first search (BFS)*. Both are based on the idea of successively visiting neighbors of already visited vertices, started at some initial vertex, but no vertex is visited twice. BFS and DFS then only depend on the order at which neighbors are visited.

If we consider all the edges of the graph along which we visited the respective vertices, then each graph traversal gives rise to a *spanning tree* of the graph. At least if the graph is connected. If the graph is not connected then the traversal will not visit all vertices. So we can continue at an unvisited vertex and so traverse each connected component individually. In the following we therefore may assume the graph is connected.

Breadth-first search. All we need to define now is the order at which vertices are visited. In BFS we first visit all neighbors of the initial vertex. After that we visit all of their neighbors. And so on. So BFS traverses the graph in "onion layers" around the initial vertex, see fig. 4.17a. Hence, the algorithm is similar to the layer-wise traversal of binary trees: We have a queue of to be visited vertices and when we visit a new vertex we add all its neighbors to the queue, see algorithm 7. And since we do not want to visit a vertex twice, we have an array in which we recall the visited-status of each vertex.

Algorithm 7 Breadth-first search for a connected graph $G = (V, E)$ with source vertex s.

```
procedure BFS(graph (V, E), source vertex s)
    visited ← [False: v ∈ V]          ▷ Initialize all vertices as unvisited
    Q ← {s}                           ▷ Initialize queue with source vertex
    visited[s] ← True                 ▷ Mark source vertex as visited
    while Q not empty do
        u ← Q.pop()                   ▷ Get next vertex from queue
        for neighbors v of u do
            if not visited[v] then    ▷ For unvisited neighbors v
                visited[v] ← True     ▷ Mark v as visited
                Q.push(v)             ▷ Add v to queue
```

Let us analyze the algorithm: Note that Q can contain a vertex at most once, because after the first time its visited status is set. Next, observe that the vertices in Q are in increasing distance to s, which leads to the BFS behavior. The time complexity depends on the data structure we use for the graph. If we use an adjacency matrix then listing all neighbors of a vertex takes $\Theta(|V|)$ time and then on the complete graph algorithm 7 takes $\Theta(|V|^2)$ time. If we use an adjacency list then listing all neighbors of u takes $\Theta(d(u))$ time and the entire algorithm takes $\Theta(\sum_{u \in V} d(u)) = \Theta(|E|)$ time by lemma 7, which might be much less then $\Theta(|V|^2)$.

Figure 4.17a illustrates a BFS traversal on a hexagonal grid. The numbers indicate the order of edges along which the traversal happens. That is, an edge $\{u, v\}$ is highlighted when when the outer loop in algorithm 7 pops u and the inner loop pushes v to Q. This yields a *traversal tree*, since every v is visited by a unique u. The entire tree can be stored by storing the parent vertex for each vertex, which is the predecessor in the traversal. We store the *rooted tree* as introduced in section 4.1.3. The order at which the edges in the traversal tree are traversed depends on the order at which the neighborhood of each vertex u is probed in the inner loop. In the examples of fig. 4.17 the order is assumed to always be cw starting in west direction.

Depth-first search. For depth-first search (DFS) we do not traverse in onion layers but we keep going as deep as possible by continuing with a neighbor of the last visited vertex, if possible, see fig. 4.17b. This is done by replacing the queue by a stack, by replacing the first in, first out (FIFO) property by the last in, first out (LIFO) property. See algorithm 8 for details.

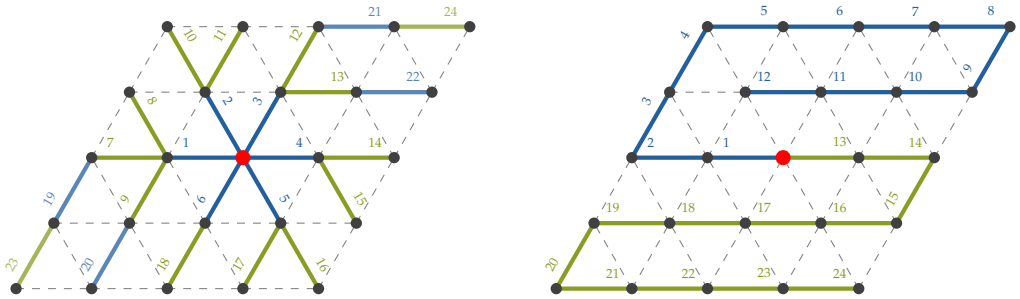
The time complexity of algorithm 8 is the same as for algorithm 7, i.e., $\Theta(|E|)$ if we use an adjacency list to efficiently enumerate the neighbors of a vertex and $\Theta(|V|^2)$ if we use an adjacency matrix.

4.2.3 Shortest paths

Maybe the most prominent class of graph algorithms are those for computing the shortest paths and in particular the *Dijkstra algorithm*, which maybe rooted in the large amount of immediate

Algorithm 8 Depth-first search for a connected graph $G = (V, E)$ with source vertex s.

```
procedure DFS(graph (V, E), source vertex s)
    visited ← [False: v ∈ V]          ▷ Initialize all vertices as unvisited
    S ← {s}                           ▷ Initialize stack with source vertex
    while S not empty do
        u ← S.pop()                   ▷ Get next vertex from stack
        if not visited[u] then        ▷ For unvisited vertex u
            visited[u] ← True         ▷ Mark u as visited
            for neighbors v of u do
                S.push(v)             ▷ Add v to stack
```



(a) BFS (b) DFS

Figure 4.17: BFS and DFS on a graph forming a hexagonal grid with the source vertex marked red. The edges of the traversal trees are numbered in the traversal order. In (a) the color indicates the layers. In (b) the color is changed when a vertex is traversed where traversal cannot continue at a neighbor.

applications, from logistics to networking, from robotics to urban planning, from gaming to social network analysis.

For undirected, unweighted graphs we actually already know how to compute shortest paths, namely through BFS. Its traversal tree contains all shortest paths from a *single source* vertex to *all destination* vertices. This is because BFS traverses the vertices in increasing distance from the source vertex, where "distance" means number of edges.

For weighted graphs, however, it can happen that a shorter path from a vertex v to a vertex u exists that requires more edges, yet with smaller total weight. This makes the problem more interesting. Also, we exclude negative weights, because if there is a cycle with total negative weight then minimizing the length of a path would mean we infinitely traverse the negative-weight cycle. So let us consider a weighted, directed graph (G, w) with positive weights and its adjacency matrix A, where non-edges are encoded as ∞.

Just like BFS, Dijkstra is a single-source all-destination shortest path algorithm. It also leads to a shortest-path tree based on the following observation, which we already mentioned in section 2.3.1 on dynamic programming: Shortest paths contain optimal sub-solutions. That is, if the shortest path $s \rightsquigarrow v$ leads over a vertex u then the shortest path $s \rightsquigarrow u$ is a prefix of $s \rightsquigarrow v$, see fig. 4.18. Otherwise we could have a shortcut from s to v via the alternative $s \rightsquigarrow u$.

Algorithm 9 Dijkstra's algorithm for one-source s all-destinations shortest paths.

```
procedure DIJKSTRA(vertex set V, weights w, source vertex s)
    for v ∈ V do
        dist[v] ← ∞                          ▷ Distance of v to all vertices
        prev[v] ← Null                       ▷ Predecessor in traversal tree
    dist[s] ← 0                              ▷ Distance to source s
    for v ∈ V do                             ▷ Initialize priority queue Q
        Q.insert(v, dist[v])
    while Q not empty do
        u ← Q.pop()                          ▷ Get vertex u closest to current tree
        for neighbors v of u do              ▷ Check for a shortcut to v via u
            if v ∈ Q then                    ▷ For unprocessed vertices v
                d ← dist[u] + w(u, v)
                if d < dist[v] then          ▷ There is a shortcut of distance d
                    dist[v] ← d
                    prev[v] ← u
                    Q.update(v, dist[v])
    return dist, prev
```

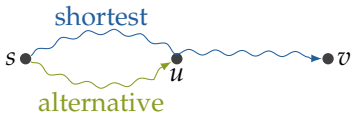


Figure 4.18: The shortest path $s \rightsquigarrow v$ contains optimal sub-solutions $s \rightsquigarrow u$. The alternative $s \rightsquigarrow u$ cannot be shorter, otherwise we would have a shortcut for $s \rightsquigarrow v$.

Dijkstra's algorithm is now very similar to BFS: In a queue we have vertices to be visited. We populate the queue with neighbors of visited vertices and we are done when the queue is empty. However, instead of an ordinary queue we have a *priority queue* Q so that `pop()` returns the vertex with minimum distance to the source vertex. Also, it may happen that when we visit a vertex u that we found a shortcut to some of its neighbors v, and then we have to update the priority of v in Q. As for the traversal tree of BFS, all we need to store is the predecessor information for each vertex to have the entire tree represented. When we then query a shortest path $s \rightsquigarrow v$ we go backwards in the tree from v to s to retrieve the entire path, see fig. 4.9. In algorithm 9 we have the algorithm.

The time complexity of Dijkstra's algorithm depends on the data structure we use for the priority queue. If we use a simple array then the time complexity is $\Theta(|V|^2)$, because we have to scan the entire array to find the vertex of minimum priority. However, a *binary heap* allows to find the minimum and update the priority of an element in $O(\log |V|)$ time. This then leads to a time complexity of $O((|V| + |E|) \log |V|)$ if we encode the graph as an adjacency list.

Finally, we would like to remark that there are also all-source all-destination shortest path algorithms. The most prominent one is the *Floyd-Warshall algorithm*, with a time complexity of $\Theta(|V|^3)$.

Algorithm 10 Prim's algorithm for computing the MST of a connected graph.

```
procedure PRIM(graph (V, E), weights w, source vertex s)
    for v ∈ V do
        dist[v] ← ∞                                  ▷ Distance of v to all vertices
        prev[v] ← Null                               ▷ Predecessor in traversal tree
    dist[s] ← 0                                      ▷ Distance to source s
    for v ∈ V do                                     ▷ Initialize priority queue Q
        Q.insert(v, dist[v])
    while Q not empty do
        u ← Q.pop()                                  ▷ Get vertex u closest to current tree
        for neighbors v of u do                      ▷ Check for a shortcut to v via u
            if v ∈ Q then                            ▷ For unprocessed vertices v
                d ← w(u, v)
                if d < dist[v] then                  ▷ There is a shortcut of distance d
                    dist[v] ← d
                    prev[v] ← u
                    Q.update(v, dist[v])
    return dist, prev
```

4.2.4 Minimum spanning trees

As a final example of graph algorithms we will discuss the *minimum spanning tree (MST)* algorithms. Note that very traversal tree is also a spanning tree of the graph, given it is connected. However, it is general not a MST, also not for the Dijkstra algorithm. In the following we consider a weighted, undirected graph (G, w) with positive weights, and we may assume G is connected. There are two common algorithms to compute the MST: *Kruskal's algorithm* and *Prim's algorithm*. The first merges trees in a forest until we end up with a single spanning tree. The second keeps growing a tree until it spans the entire graph.

Prim's algorithm. The MST contains all vertices of G, so we may start with an arbitrary vertex s. This vertex is considered to be a trivial tree T, which we grow towards the MST. We do so by adding edges incident ot vertices of T (along with the other adjacent vertex), by considering all such edges and picking the one with the minimum weight.

The algorithm is given in algorithm 10. Note that it is almost identical to Dijkstra's algorithm, except for the line when we compute the distance for a possible shortcut: In Prim's algorithm we compute the distance of a vertex v to the current tree, in Dijkstra's algorithm we compute the distance to the source vertex s. The time complexity is therefore the same as for Dijkstra's algorithm, $O((|V| + |E|) \log |V|)$ if we use a binomial heap and an adjacency list.

Kruskal's algorithm. This algorithm is initialized by considering all vertices of G as trivial trees and then iteratively merging them to form larger trees until we en up with the MST, see algorithm 11.

The time complexity of algorithm 11 is in $\Omega(|E| \log |E|)$ since we have to sort E. For an upper bound we need to discuss the data structure for the forest F. Here we would use a so-called *union-find* data structure, also known as *disjoint-set* data structure. It manages a set of of sets, and allows to efficiently find the set by its elements and merge two sets. Creating this data structure takes $O(|V|)$ time. Then for each edge $\{u, v\}$ we have two find operations and at most one union operation, taking $O(\alpha(|V|))$ time, where α is the *inverse Ackermann function*. The inverse

Algorithm 11 Kruskal's algorithm for computing the MST of a connected graph.

```
procedure KRUSKAL(graph (V, E), weights w)
    T ← ∅                                          ▷ Initialize MST as empty set
    F ← {{v} : v ∈ V}                              ▷ Initialize forest with trivial trees
    E ← E sorted by w                              ▷ Sort edges by weight
    for {u, v} ∈ E do                              ▷ Get edges in weight-sorted order
        E ← E \ {{u, v}}                           ▷ Remove edge from E
        if {u, v} connects two different trees in F then
            T ← T ∪ {{u, v}}                       ▷ Add edge to MST
            merge trees in F containing u and v
    return T
```

Ackermann function is a very slowly growing, slower than $\log n$ or $\log \log n$ or any fixed-size composition of log. In practice we can assume it is constant, in fact less than 5.[20]

The time complexity of Kruskal's algorithm is therefore $O(|E| \log |E|)$ for a connected graph. Note that if G is not connected then Kruskal will give a spanning tree for each connected component.

Correctness. So far presented the two MST algorithms and analyzed their time complexity. However, we did not prove their correctness. Let us start with Prim.

Let us start with algorithm 10. Let us denote by T the tree that is grown by the algorithm, i.e., the tree formed by *prev* for all vertices in $V \setminus Q$. It is initially an empty tree, then a tree only consisting of s, and then subsequently new vertices are added. Whenever a new vertex u is added then $prev[u]$ is a vertex already in T. That is, T always remains a tree and when Q is empty, T is a spanning tree as it contains all vertices.

The question is whether T is a MST. Let us compare T with some MST T^* of G. If they are the same, we are done. If not then during the construction of T they must differ by some edge. So assume $\{u, v\}$ is the first edge added during Prim not belonging to T^* and let $U \subset V$ be the set of vertices until $T \subset T^*$. Since $\{u, v\}$ is not in T^* there must be exactly one other edge $\{u', v'\}$ in T^* such that $u' in U$ and $v' \in V \setminus U$. Since Prim has not taken this edge it must be that $w(u, v) \leq w(u', v')$. So if we remove $\{u', v'\}$ from T^* and add $\{u, v\}$ instead then we still have spanning tree T^* and its weight did not increase. (Therefore, in fact, $w(u, v) = w(u', v')$.) Anyhow, now we can repeat our comparison with T and T^*, only now they possibly differ by an edge that was inserted even later. We can repeat this argument until $T = T^*$.

The argument for Kruskal is similar. When we merge trees in the forest F, F always remains a forest, i.e., acyclic, until it is a single tree and as it contains all vertices it is a spanning tree. The argument that Kurskal gives an MST follows by the following lemma: For every forest F produced during Kruskal, there is a MST T that contains F. This is shown by induction over the number of edges of F. We skip the details here.

4.3 Summary

Graph theory and graph algorithms form a main pillar of discrete mathematics and computer science. They serve as versatile and general modeling language, which makes graph algorithms

[20] We have $\alpha(2^{2^{65536}}) = 4$. In Python, already `2**2**2**2**2`, which is 2^{65536}, is a number with 19 729 decimal digits, and leads to an Python exception. But this is only the exponent n to 2^{2^n} in order for α to become 4.

broadly applicable to all kind of problems that can be accessed through this language. This chapter was split into two parts: graph theory and graph algorithms.

The introduction of graph theory laid the language foundations and notions. We introduced some special graphs and the notions of paths, walks, tours, cycles, trees, connectedness, spanning subgraphs, Eulerian tours and Hamiltonian cycles. We then added more structure to graphs, by adding edge weights to receive networks and by drawing them in the plane to obtain notions of planarity and geometric graphs. We also learned about a couple of classic *NP*-hard problems, like graph isomorphism, Hamiltonian cycle and the famuous optimization problem TSP.

Then we discussed graph algorithms and started with the representation of graphs as adjacency matrix, adjacency list and DCEL for planar graphs. We traversed graphs using BFS and DFS, computed shortest paths and MSTs.

But this is only the basic introduction. Further topics would include network flows and the *max-flow min-cut theorem*. Generalization of graphs to vertex colors, the chromatic number of graphs, the famous theorems like the *four color theorem* for planar graphs and their relation to resource planing or register allocation in compilers. Spectral graph theory and the relation to the chromatic number of graphs and the Laplician operator, which is the basis for Google's *page rank* algorithm. Further topics could be generalizations to hyper graphs or topological spaces like simplicial complexes.

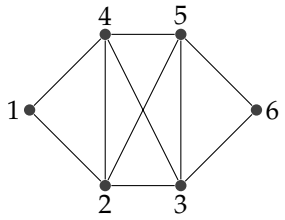
4.4 Exercises

Exercise 4.1 (⋆)**.** Prove that $|E| = \frac{r}{2}|V|$ for any r-regular graph (V, E).

Exercise 4.2 (⋆)**.** For which n and m is $K_n = W_m$? What is the simplest argument you can think of?

Exercise 4.3 (⋆)**.** Find an Eulerian tour or show that there is none:

- For the graph in exercise 4.14.
- For the graph below.



Exercise 4.4 (⋆)**.** How many Hamiltonian cycles does the graph in exercise 4.3 possess? Prove your solution.

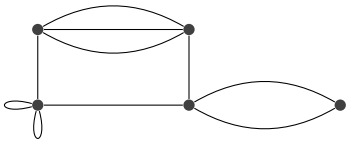
Exercise 4.5 (⋆)**.** We want to test whether a graph has an Eulerian tour and found that three of its vertices have odd degree. So there is no Eulerian tour, whether closed nor non-closed. But something seems odd in the counting. What is it?

Exercise 4.6 (⋆)**.** A graph $G = (V, E)$ with $|V| = v$ and $|E| = e$, which happens to be a forest. How many trees does G contain in terms of v and e?

Exercise 4.7 (⋆). For which $K_{m,n}$ is there a Hamiltonian cycle and an Eularian tour?

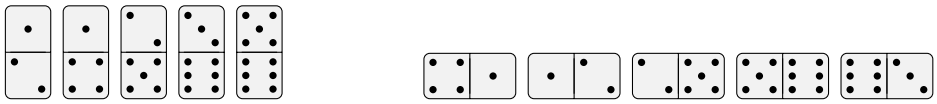
Exercise 4.8 (⋆). Show that all cycles of a bipartite graph have even length.

Exercise 4.9 (⋆). A *multigraph* is an (undirected) graph (V, E) where E is a multiset, i.e., there can be multiple edges $\{u, v\}$ between the same pair of vertices. Below is a drawing of such a multigraph.



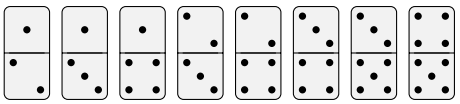
How do we need to generalize the definition of the *degree* of a vertex for lemma 10 to generalize to multigraphs?

Exercise 4.10 (⋆). Domino is a game where we have a set of domino tiles, each with two numbers from 1 to 6. The goal is to place the tiles in a row such that the numbers of adjacent tiles match. For instance, below we have at the left a set of domino tiles and at the right a possible solution to arrange this set of tiles.



How do we phrase this problem as a graph problem? What is the graph (V, E)? Draw the graph. What do we do with a tile that has the same number on both sides? What do we do if a tile is there twice?

Give a solution for this set of tiles:



Exercise 4.11 (⋆). We have a computer network and a routing mechanism that is based on a spanning tree. Unfortunately, the routing mechanism broke down. The graph $G = (V, E)$ that describes the network's topology has $|V| = n$ and $|E| = e$ and happens to form a forest. In how many trees did G broke given n and e?

Exercise 4.12 (⋆). We have a computer network that is modeled as a graph G. Somebody came with a clever idea to efficiently check whether some links (edges of G) might be broken: A token is passed from host to host until every link has been traversed. If the token comes back to the origin then the test passed; otherwise it did not.

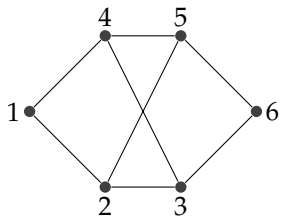
What graph-theoretic problem underpins this idea? Do you have objections or comments regarding this method?

Exercise 4.13 (⋆). Let A be the adjacency matrix for a graph $G = (V, E)$ with $V = \{1, \ldots, n\}$.

- Show that for a digraph G we have $d^+(k) = \sum_{j=1}^{n} A_{kj}$ and $d^-(k) = \sum_{j=1}^{n} A_{jk}$.
- Show that for an unweighted graph G we have $d(k) = \sum_{j=1}^{n} A_{kj} = \sum_{j=1}^{n} A_{jk}$.

- What is the $\sum_{i,j} A_{ij}$?
- How to interpret the trace of A, i.e., $\sum_{i=1}^{n} A_{ii}$?

Exercise 4.14 (⋆⋆). We are given the graph $G = (V, E)$ below with $V = \{1, \ldots, 6\}$.



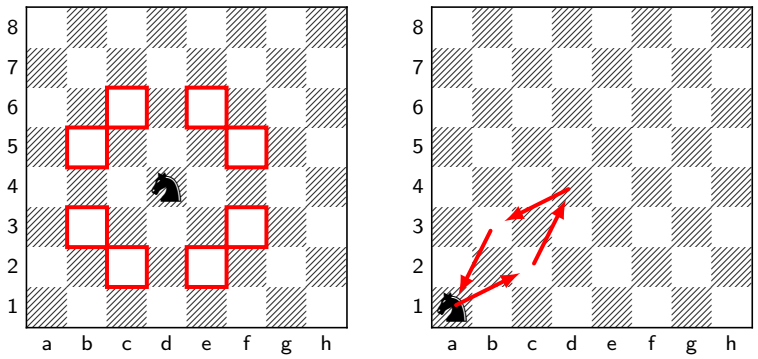
- What is the adjacency matrix A for the given graph?
- Compute with Python or Matlab the matrices A^3, A^4, A^5. The matrix A^k contains the number of walks $i \rightsquigarrow j$ of length k as A^k_{ij}. Check this claim for a few examples.
- Find all walks $1 \rightsquigarrow 6$ of length 3 and length 4.
- Is the graph bipartite?

Exercise 4.15 (⋆). Let A be the adjacency matrix of an undirected graph $G = (V, E)$. Prove that $A^k_{i,j}$ is the number of walks $i \rightsquigarrow j$ of length k between vertices i and j.

Exercise 4.16 (⋆). The adjecency matrix A of $K_{m,n}$ has a special structure. What is it? And what about A^k?

Exercise 4.17 (⋆⋆). Rewrite algorithm 8 for DFS to compute Euler tours by means of Hierholzer's algorithm.

Exercise 4.18 (⋆). In chess a knight moves one step along a diagonal and then one step horizontally or vertically. In the left board below, the knight on square d4 can move on one of the eight marked squares. In the right board, the knight moves in a sequence of moves back to its original position.

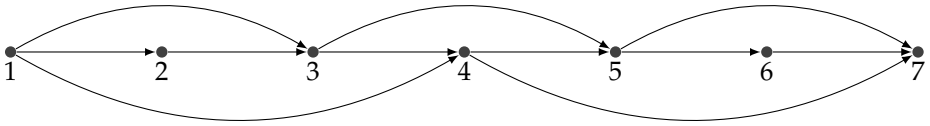


The so-called *knight's tour* is the problem of finding a sequence of moves for a knight that would visit each square of a chess board exactly once. How would we phrase the problem as a graph problem? What is the graph (V, E)?

Bonus: Implement an algorithm similar to algorithm 8 to find a knight's tour, starting at square a1.

Exercise 4.19 (⋆). In network-security analysis, an attack graph is a directed graph, where nodes model assets (e.g., hosts) and an edge from a node u to a node v indicates a possible attack that allows an attacker to move from asset u to asset v. A path in an attack graph is called an attack path.

How many attack paths are there from asset 1 to asset 7 in the attack graph below?



Exercise 4.20 (⋆). Find a planar drawing of $K_{2,4}$.

Exercise 4.21 (⋆). Prove that K_5 and $K_{3,3}$ are non-planar. (Hint: A planar drawing of K_5 contains a planar drawing of K_4, and one of $K_{3,3}$ contains one of $K_{2,3}$. Analyze the faces of those.)

Exercise 4.22 (⋆). We got a convex polytope B in $\mathbb{R}^3$ and consider its surface bd B, which consists of convex polygons. We would like to cut edges of bd B in a way that allows us to unroll bd B flat on the plane in one piece.

Somebody has a gut feeling on how to do this: From bd B we construct a graph $G = (V, E)$ as follows: V is the set of polygons and we place an edge $\{u, v\}$ when the polygons u and e share an edge. Then we consider a spanning tree of G.

How does this help to solve the problem?

Exercise 4.23 (⋆). Mr. Evil shot an unknown number of satellites into an orbit of earth, all at the same altitude. He attempts to build a triangular mesh with the satellites to track every human being on earth.[21] In each triangle he can track up to 10^5 people and we assume a world population of $8 \cdot 10^9$.

If Mr. Evil's claim is true, how many satellites would he need at least? How many links are then established between the satellites in total? And how many links would a spanning tree of the mesh have?

Exercise 4.24 (⋆). Mr. Evil is back. He now switched from a triangular mesh to a quadrilateral mesh, i.e., all faces are four-sided. How many satellites are now needed? How many links are formed? How many links would a spanning tree of the mesh have?

Exercise 4.25 (⋆). The five Platonic solids are given in fig. 3.3b.

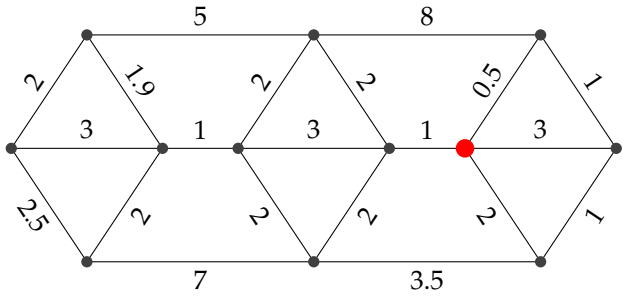
- Check Euler's formula for each of them.
- Describe the dual graph for each of them. What do you observe?
- Someone claims to have found a new Platonic solid: It has square faces and at each vertex four faces are meeting. Prove it wrong using Euler's formula.

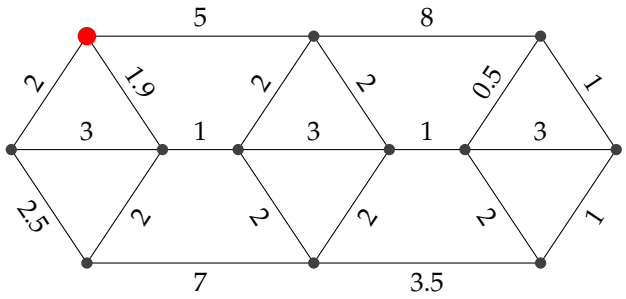
[21] We assume the mesh forms a convex polytope.

Exercise 4.26 (⋆)**.** Illustrate the traversal tree for BFS and DFS on K_5 and $K_{3,3}$. Neighbors are iterated in order of the vertex indices. (For the $K_{3,3}$ the vertex set $V = \{1, \ldots, 6\}$ is partitioned into $\{1, 2, 3\}$ and $\{4, 5, 6\}$.)

Exercise 4.27 (⋆)**.** Give simple examples of graphs where the shortest-path tree produced by Dijkstra's algorithm is not the MST.

Exercise 4.28 (⋆)**.** We are given two graphs below with one vertex marked red. Give for each of them the tree of shortest paths originating from the marked vertex according to Dijkstra's algorithm.





Exercise 4.29. Illustrate BFS on the (unweighted) graphs from exercise 4.28, starting at the red vertex.

Exercise 4.30. Illustrate DFS on the (unweighted) graphs from exercise 4.28, starting at the red vertex.

Exercise 4.31. Illustrate Kruskal's algorithm on the graphs from exercise 4.28.

Exercise 4.32. Illustrate Prims's algorithm on the graphs from exercise 4.28.

Part II

Numerical mathematics

Chapter 5

Computing with numbers

5.1 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 as "42" is by far superior to the Roman numeral "XLII" for the execution of arithmetic operations, such as addition and multiplication.

The decimal system's power allows for the ease at which we operate on numbers and to a certain extent is a key enabler for our modern society, let it be mundane everyday use, science and engineering, finance and business, or politics and administration. The following quote is by Whitehead[1] [51, 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.

5.1.1 Mathematical basics

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. We call this a *positional number system*, where the position of a digit determines the value it adds to the number. In this concrete case it is the base-10 positional number system or *decimal system* for short.

Every non-negative real number z can be represented in the decimal system as a sequence of digits. In a mathematically more precise notation, for each real number $z \geq 0$ there is a doubly

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

infinite sequence $(\dots, a_{-2}, a_{-1}; a_0, a_1, \dots)$ of digits $a_i \in \{0, \dots, 9\}$, such that

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

(Note how we use the semicolon to mark the position of a_0 in the doubly infinite sequence.) 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 $z \geq 0$ there is a doubly infinite sequence (a_i), with $a_i \in \{0, \dots, b-1\}$ called a *digit*, such that

$$z = \sum_{i=-\infty}^{\infty} a_i \cdot b^i. \tag{5.1}$$

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) is more favorable. It is a common notation to write the basis as subindex when there might be confusion. By this notation we have $11 = 11_{10} = 13_8 = 1011_2 = 14_7$. When the basis is larger than 10 then we use alphabetical digits $\mathrm{a}, \mathrm{b}, \mathrm{c}$ and so on, e.g., $11 = \mathrm{b}_{16}$ or $24 = 1\mathrm{a}_{14}$.

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. (5.1).[2] 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 "one" 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 $(\dots, 9, 9, 9; 0, 0, \dots)$.

5.1.2 Finite representations

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*

$$z = \underbrace{a_{m-1} b^{m-1} + \cdots + a_1 b^1 + a_0 b^0}_{\text{integral part}} + \underbrace{a_{-1} b^{-1} + \cdots + a_{-n} b^{-n}}_{\text{fractional part}} = \sum_{i=-n}^{m-1} a_i b^i. \tag{5.2}$$

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, while for $n > 0$ we can also represent non-integer numbers. If we choose $m = 0$ then we obtain numbers in the interval $[0, 1)$.

[2] If there is no such index then the series in eq. (5.1) 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., $z = 0$.

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 to the base b. We can rephrase eq. (5.2) such that

$$z = \left(\sum_{i=0}^{m+n-1} a_{i-n} b^i \right) \cdot b^{-n} \tag{5.3}$$

and therefore $z = k \cdot b^{-n}$ for some integer k. From eq. (5.3) 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 $-n, \ldots, m$ 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*[3] to $p = m - k + 1$ digits by setting

$$z' = \underbrace{a_m b^m + a_{m-1} b^{m-1} + \cdots + a_k b^k}_{p \text{ summands}} \approx z. \tag{5.4}$$

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

$$z' = \underbrace{(a_m b^{-1} + \cdots + a_k b^{-p})}_{a} \cdot b^{m+1} = a \cdot b^{m+1}. \tag{5.5}$$

The number a is called *mantissa*[4], 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

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

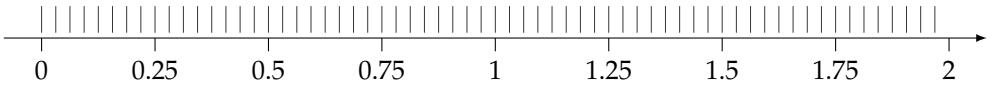
Its mantissa is 0.314 and its exponent is 1. Note that in eq. (5.5) the number $m+1$ is not considered fixed but part of the representation along with the mantissa, which allows the decimal point to "float". (Conversely, the n in eq. (5.3) is considered to be fixed.)

For technical-didactical reasons we assumed so far that $z > 0$, otherwise the index m in eq. (5.4) may not necessarily exist. However, once we fix m in eq. (5.5), we can of course set all digits $a_i = 0$ to represent $z' = 0$. Nevertheless, there is no normalized floating-point representation of zero, i.e., we cannot "normalize zero".

[3] Dt. Abschneiden
[4] Dt. Mantisse

Distribution of numbers. In fig. 5.1 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 $-2, \ldots, 1$. That is, each interval $[0, 2^e)$ for $e \in \{-2, \ldots, 1\}$ 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.[5]



(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 5.1: 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*[6] in the mantissa is zero.

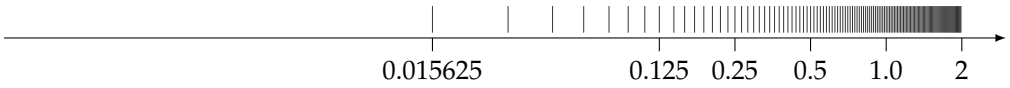
If we add one further digit to fixed-point representation as fractional digit then we place between each two neighboring numbers in fig. 5.1a an additional number. The same is true for fig. 5.1b if we add a digit to the mantissa. If, however, we add a digit to the exponent then we can extend the range of the exponent to $-6, \ldots, 1$ and we effectively add 16 numbers for each of the four intervals $[0, 2^k)$ with $k \in \{-6, -5, -4, -3\}$, some of which already being covered. The smallest positive number we can then represent is $0.0001_2 \cdot 2^{-6} = 2^{-10} = 0.0009765625$, while in $\mathbb{Q}_2^{1,6}$ 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. 5.2, 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

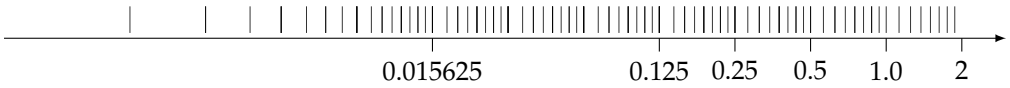
[5] The sum may exceed the range covered and cause an overflow, but this issue is of an essentially different nature.

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

than 616 decimal orders of magnitude.



(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 5.2: 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.

5.2 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 further use of it in section 5.3. In real-world applications, however, the basis b is essentially always two[7], but we also need to deal with negative numbers, which we ignored so far for the sake of simplicity. So let us assume $b = 2$ as the basis for the remainder of this section and discuss real-world number formats. A digit is therefore called a *bit* (binary digit).

5.2.1 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 $a_{n-1}, \ldots, a_0$ is an element of $Q_2^{n,0}$ and encodes the number

$$z = \sum_{i=0}^{n-1} a_i 2^i. \tag{5.6}$$

By eq. (5.6), 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.[8] In this format we would actually distinguish between the encoding of +0 and −0. We can also ask whether (+0) + (−0) shall result in +0 or −0 and whether (−0) + (+0) gives the same result, i.e., whether the operation + fulfills the commutative property[9]. Arithmetics with one's complement is more complicated.

[7]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.

[8]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.

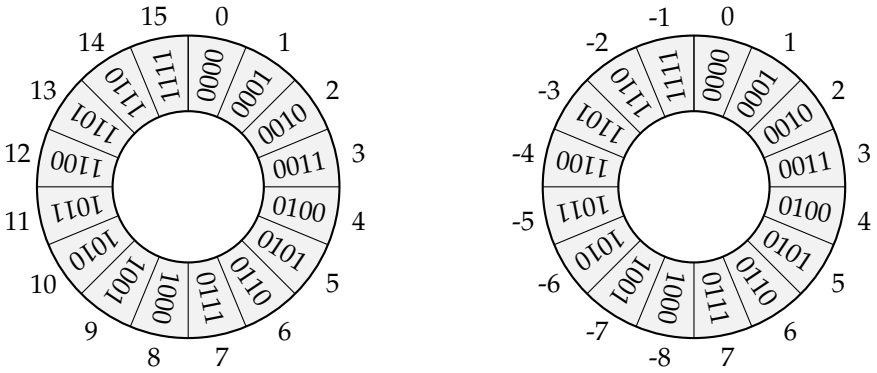
[9]The commutative property is an essential property and one should think twice sacrificing it. In particular, we do not end up with a so-called *commutative group* anymore from an algebraic point of view. On the other hand, there are other symmetry properties we should consider when defining the operator +. For instance, how would we argue a tendency towards +0 in the results and break symmetry with −0? In a certain sense, we would violate the philosophical

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 $a_{n-1}, \ldots, a_0$ encodes the number

$$z = -a_{n-1}2^{n-1} + \sum_{i=0}^{n-2} a_i 2^i. \tag{5.7}$$

By eq. (5.7), the number z can attain any integer number in the range $[-2^{n-1}, 2^{n-1} - 1]$.

Figure 5.3 illustrates eq. (5.6) and eq. (5.7). 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. 5.3 we have 4-bit unsigned and signed integers. In case of signed integers, adding 5 to −2 (encoded as 1110) gives us 3 (encoded as 0011). Likewise, for unsigned integers adding 5 to 14 (encoded as 1110) gives 3 (encoded as 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]



(a) 4-bit unsigned integers

(b) 4-bit signed integers

Figure 5.3: Unsigned and signed 4-bit integers. Adding numbers works the same way on a bit level.

5.2.2 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 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. 5.4. The mantissa a is obtained by

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

where in "$1.M_2$" we put the mantissa bits M as fractional part behind "1.", while in "$1 + M \cdot 2^{-p}$" we see M as an encoding of an unsigned integer.

principle of Occam's razor, and we should think twice again. By the way, as a personal remark, Occam's razor turned out to be an important design guideline for technical systems .

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

[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.

This definition differs slightly from b-adic floating-point numbers: Note that for $b = 2$ the only non-zero digit is 1. That is, 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. 5.4, is then given by

$$z = (-1)^S \cdot \underbrace{(1 + M \cdot 2^{-p})}_{a} \cdot 2^{E-B}, \tag{5.8}$$

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]

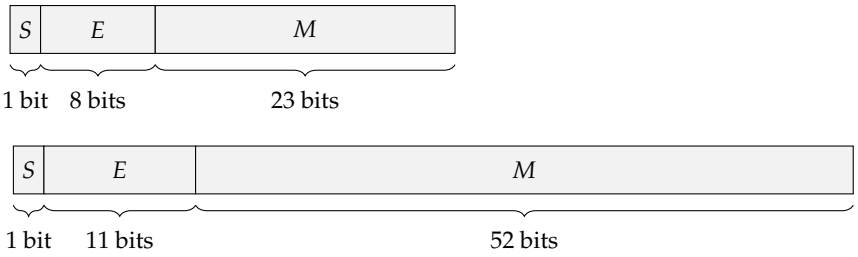


Figure 5.4: The binary layouts of IEEE 754 foating-point numbers with single and double precision. The sign bit is given by S, the exponent bits by E and the mantissa bits by M.

In addition to eq. (5.8) 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.[14] The largest possible exponent, when $E = 11\cdots1_2$, is used to represent $+\infty$ and $-\infty$, but also NaN (not a number).[15] This allows for the following arithmetic laws in IEEE 754:

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

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.[16] Goldberg [21] provides many more details regarding programming with IEEE 754 floating-point numbers.

[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!

[14] See also the footnote on distinguishing between +0 and −0 for the one's complement on page 94. The IEEE 754 standard defines all these cases.

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

[16] Processor bugs in the *floating point unit (FPU)* are not entirely unusual. The best known example is probably the infamous FDIV bug of early Intel Pentium processors [38]. Also, in the embedded domain some processors only provide a limited feature set of the IEEE 754.

5.2.3 Fixed-point formats

Fixed-point numbers typically span a significantly smaller range than floating-point numbers, as we learned in section 5.1.2. 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*[17] 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.

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.
- Q*n* 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* numbers 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. 5.5, then we simply read the bit pattern as a signed or unsigned integer and think of it being scaled by 2^{-n}, compare also with eq. (5.3).

Figure 5.5: 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 C99 or C11 – does not know fixed-point

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

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 [18]. The current and second version from 2008 is called *ISO/IEC TR 18037:2008* [32]. This standard adds fixed-point arithmetic data types and more.[18] The GCC C-compiler supports this standard since 2007 [20]. The Clang compiler of LLVM has some preliminary implementation since 2018 [9]. However, neither fully supports this standard yet.[19]

Embedded C defines two new type specifiers `_Fract` and `_Accum`, which can be used in combination with the type specifiers `short`, `long`, `signed`[20] and `unsigned` and gives rise to the 12 types in table 5.1. 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.

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:

```
1 sat fract x = -0.7r;                          // Suffix r for fract literals
2 sat fract y = -0.7r;
3 printf("x + y = %f\n", (double) (x+y));    // Prints -1.0 due to sat
4 sat accum a = 0.7k;                           // Suffix k for accum literals
5 sat accum b = 0.7k;
6 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 5.1. 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`.

5.3 Floating-point arithmetic

5.3.1 Rounding

In order to obtain from a real number z a floating-point number $\bar{z}$ 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. (5.4) 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.

[18] 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.

[19] 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.

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

Table 5.1: 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.

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

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

$$\mathrm{rd}_{2,10}(0.134) = 0.13 \qquad \mathrm{rd}_{2,10}(0.135) = 0.14 \qquad \mathrm{rd}_{3,10}(0.1996) = 0.200,$$

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.[21]

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*,[22] but there are also *round toward* 0, *round toward* $+\infty$, and *round toward* $-\infty$. The rounding mode then again influences how overflow is handled.[23]

5.3.2 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 5.3.1 the bounds

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

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

[22] 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 $+\infty$.

[23] 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.

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.[24]

5.3.3 Machine operations

Machine operations result in machine numbers. That is, the result of machine operations on machine numbers are rounded in order to again obtain machine numbers. Instead of the usual basic arithmetic operations $+, -, \ldots$ a processor therefore performs the following machine 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, except the above rule, rounding has typically far-reaching consequences such that many usual rules fail to hold! For instance, the order of operations matters because the usual *associative law* does not hold anymore. For example, let $b = 10$ and $s = 2$ for $\mathrm{rd}_{s,b}$ 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[25] 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.

[24] 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.

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

As a consequence the comparison of machine numbers can (virtually) never be done exactly but needs to be performed by means of thresholds.[26] Those so-called *threshold-based* or *epsilon-based comparisons* introduce a fixed application-dependent $\epsilon > 0$ in order to define the following comparison operators:

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

But note that also for these comparison operators many usual laws do not hold anymore. For instance, the *transitive law* fails for $\doteq$ and $\dot{\leq}$, e.g., it can be that $x \doteq y$ and $y \doteq z$, but still $x \dot{\neq} z$. Yet, *symmetry* of $\doteq$ and (a weaker form of) *antisymmetry* of $\dot{\leq}$ do hold, i.e., if $x \dot{\leq} y$ and $y \dot{\leq} x$ then $x \doteq y$.

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 later SSE floating-point units of Intel processors do not possess the extended double precision registers anymore and perform operations in single or double precision, which may lead to different results, depending on what FPU unit is utilized by the compiler.

5.4 Numerical analysis

5.4.1 Numerical algorithms

An *algorithm* transforms input into output in order to solve a specific problem. The problem *sorting* considers a list of numbers[27] 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 $(x_1, \ldots, x_n)$ as input and produces a real m-tuple $(y_1, \ldots, y_m)$ 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 $x = (x_1, \ldots, x_n)$ 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.[28] A concrete algorithm for this problem could follow the well known formula

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

[26] See also the warning option `-Wfloat-equal` for gcc.

[27] 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.

[28] 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.

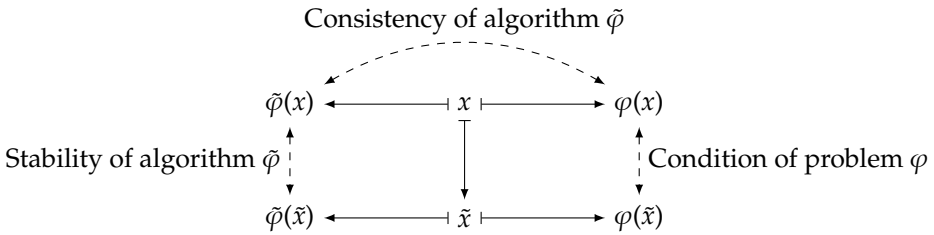


Figure 5.6: Quality of numerical computations from three perspectives: *Condition* of a problem, *stability* and *consistency* of an algorithm.

in order to compute the roots. Each of the operations $+, \cdot, \sqrt{}, \ldots$ we interpret as a computational step in the algorithm, which are composed together to form a map $\tilde{\varphi}$ that constitutes the algorithm. That is, we could "implement" the above formula by one of many possible sequences of steps, like the following:

$$\begin{pmatrix} a \\ b \\ c \end{pmatrix} \overset{\varphi_1}{\longmapsto} \begin{pmatrix} -b \\ b^2 \\ ac \\ 2a \end{pmatrix} \overset{\varphi_2}{\longmapsto} \begin{pmatrix} -b \\ b^2 \\ 4ac \\ 2a \end{pmatrix} \overset{\varphi_3}{\longmapsto} \begin{pmatrix} -b \\ b^2 - 4ac \\ 2a \end{pmatrix} \overset{\varphi_4}{\longmapsto} \begin{pmatrix} -b \\ \sqrt{b^2 - 4ac} \\ 2a \end{pmatrix} \overset{\varphi_5}{\longmapsto} \begin{pmatrix} -b + \sqrt{b^2 - 4ac} \\ 2a \end{pmatrix} \overset{\varphi_6}{\longmapsto} \left(\tfrac{-b + \sqrt{b^2 - 4ac}}{2a} \right)$$

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 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

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

This gives three quality measures in numerical computation, see fig. 5.6.[29]

[29] You may ask why fig. 5.6 is not symmetric, that is, why there is no arrow between $\tilde{\varphi}(\tilde{x})$ and $\varphi(\tilde{x})$. Note, this again is the consistency of $\tilde{\varphi}$. We just kept fig. 5.6 free of redundancies.

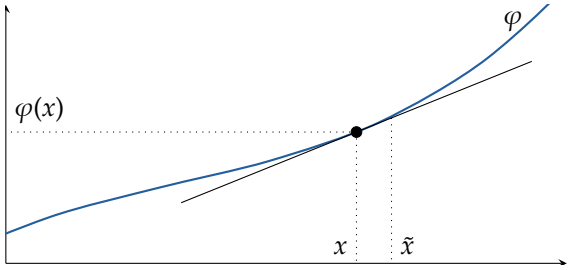


Figure 5.7: The tangent on the function graph of φ at position x has the slope $\varphi'(x)$. At a nearby position $\tilde{x}$ the function evaluates to $\varphi(\tilde{x})$, which can be approximated by the tangent.

5.4.2 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.[30]

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. 5.7. 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.[31] Similarly, we denote by $\dot{\leq}$ that $\leq$ holds only approximately. Then we have

$$\|\varphi(\tilde{x}) - \varphi(x)\| \mathrel{\dot{\leq}} \|\varphi'(x)\| \cdot \|\tilde{x} - x\|. \tag{5.9}$$

The above lines were motivated for the case $n = m = 1$. However, the beauty of mathematics (or rather the notation) unfolds here in a way such that ineq. (5.9) also holds for $n, m \geq 1$ if $\|\varphi'\|$ is suitably interpreted for higher dimensions: Let $\varphi(x) = (\varphi_1(x), \ldots, \varphi_m(x))$ then we denote by $\varphi'(x)$ the Jacobian matrix

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

The Jacobian of $\varphi\colon \mathbb{R}^n \to \mathbb{R}^m$ contains the partial derivatives of all m elements of $(\varphi_1(x), \ldots, \varphi_m(x))$ by all n dimensions of $x = (x_1, \ldots, x_n)$. Hence, it generalizes the tangent slope in fig. 5.7 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 Euclidean matrix norm of A. (Sometimes it turns out handy to switch to a different p-norm, but for now we leave

[30] 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.

[31] 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 $\|\tilde{x} - x\|^2$.

it there.[32]) Hence, the norm of the Jacobian is given by

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

Using this interpretation for $\|\varphi'\|$ generalizes ineq. (5.9) to higher dimensions.[33] Note that ineq. (5.9) 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*[34] if κ_{abs} or κ_{rel} are significantly greater than 1 and otherwise *well-conditioned*[35]. However, there is no general rule to what "significantly" actually means, so it depends on the specific application at hand.

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 $\varphi(x_1, x_2) =$

[32] 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 $x, y \in V$. 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 $\lim_{p\to\infty} \|x\|_p = \max_i |x_i|$. Likewise, we define the p-norm of a matrix $A = (a_{ij})$ as $\|A\|_p = \sqrt[p]{\sum_{i,j} |a_{ij}|^p}$.

[33] Actually, for $n = m = 1$ equality holds in ineq. (5.9). However, for higher dimensions this is not true.

[34] Dt. schlecht konditioniert

[35] Dt. gut konditioniert

$x_1 + x_2$. 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[36] 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 %!

5.4.3 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[37] of an algorithm results from the conditions of the individual steps ψ_i: If one step is ill-conditioned then the entire algorithm – the "chain of steps" — is *instable*.

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.[38] 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:

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

[37] In [37] the stability is also called *condition of an algorithm.*

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

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

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. But there is hope that a different algorithm might perform better.

We may know, or at least easily verify, that the following equality holds:

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

We interpret this equality as a mathematically reformulation of the problem. That is, as a numerical problem as such it is just the same. However, the point is here that the right hand side gives us a hint to a different algorithm – as a sequence of elementary steps – to the same problem:

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

This algorithm is now 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 problem φ 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:

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

because in general $\| \begin{pmatrix} x & y \end{pmatrix} \| \geq |y|$ for all $x, y \in \mathbb{R}$. That is, this mathematical framework tells us that there is no hope in searching for another algorithm that would be numerically stable in this case, because the issue lies within the numerical problem itself, not a particular numerical algorithm.

5.5 Summary

In this chapter we introduced the basics of numerical programming and numerical algorithms. We started by introducing the representation of numbers by means of the b-adic expansion,

which led us to fixed-point and floating-point numbers. We then switched to number formats as used in actual computational hardware, most prominently IEEE 754 floating-point numbers, but also fixed-point number formats and the Q-format. We discussed how floating-point arithmetic inherently comes with numerical inaccuracies and how ordinary algebraic laws of arithmetics do not hold anymore. Finally, we lifted the discussion to the level of numerical problems and algorithms and how to analyze them through the concepts of condition and stability.

5.6 Exercises

Exercise 5.1 (⋆)**.** Change the following number representation to a different basis:

- 101.01_2 into the decimal system and into basis 4.
- 13_8 into the decimal system and into the hexadecimal system.
- 12.3_4 into the decimal system and the binary system.
- $1a.c_{16}$ into the decimal system and the octal system.
- $0.\overline{4}_5$ into the decimal system.

Exercise 5.2 (⋆)**.** It is well known that $\frac{1}{3}$ has the decimal representation $0.\overline{3} = 0.3333\ldots$, which is periodic. This is easy to verify, because $3 \cdot 0.\overline{3} = 0.\overline{9} = 1.0$ and hence we can rearrange to

$$\frac{1.0}{3} = 0.\overline{3}.$$

Following the same reasoning, but in the binary system, we see that $11_2 \cdot 0.\overline{01}_2 = 0.111\ldots = 1.0$. In other words, $3 \cdot 0.\overline{01}_2 = 1$, which means that

$$\frac{1}{3} = 0.\overline{01}_2.$$

- Use the above technique to find the 2-adic expansion of the decimal number 0.1_{10} and verify that your solution is correct, i.e., check that a multiplication with 10 gives 1.0 again.
- Can 0.1_{10} be exactly represented using IEEE 754 numbers?

Exercise 5.3 (⋆⋆)**.** Write a program that converts numbers between different bases. That is, it takes as input a string in which a number to a basis b is given and outputs the number to a different basis b'.

(You can use this program to test your results of the above exercises. It makes sense to limit the basis at some reasonable value, in order not to run out of symbols for the digits.)

Exercise 5.4 (⋆⋆)**.** Write a program or notebook to reproduce fig. 5.2 for different choices of integral and fractional digits for base-2 fixed point numbers and different choices of mantissa length, exponent length and largest exponent for floating-point numbers. (Take care for a reasonable plot range!)

Test your code for these cases and verify against fig. 5.2:

- Fixed-point numbers with 1 integral bit and 6 fractional bits.

- Floating-point numbers with 5 mantissa, 2 exponent bits and largest exponent of 1.
- Floating-point numbers with 4 mantissa, 3 exponent bits and largest exponent of 1.

Further plot these examples:

- Fixed-point numbers with 1 integral bit and 7 fractional bits.
- Floating-point numbers with 5 mantissa, 3 exponent bits and largest exponent of 1.
- Floating-point numbers with 4 mantissa, 4 exponent bits and largest exponent of 1.

Exercise 5.5 (⋆)**.** In September, 2022, a preprint paper appeared from NVIDIA, Arm and Intel that proposes an FP8 floating-point format following IEEE 754. In here, the E4M3 encoding has 4 exponent bits and 3 mantissa bits. The bias is 7.

Like IEEE 754, subnormals are encoded by having the exponent bits set to zero. Unlike other IEEE 754 encodings, it does not know infinities. Still, NaN is encoded by having all exponent and mantissa bits set to one. Table 1 in the paper[39] summarizes the situation.

Similar to exercise 5.4, print all possible numbers on a number line.

Bonus: Since there are only 256 different encodings, we could print all numbers out on a A4 paper sheet. Let the sign bit be zero, then there are only 128 values, which could be printed on a 16×8 table with all 16 possible exponents on one axis and all 8 possible mantissas on the other axis. The table also contains NaN.

Exercise 5.6 (⋆)**.** Let us consider the set $Q_b^{m,n}$. Answer the following questions:

- How many numbers does this set posses?
- What is the smallest number in this set?
- What is the smallest positive[40] number in this set?
- What is the largest number in this set?
- What do we get if we add up the smallest positive and the largest number?

Exercise 5.7 (⋆)**.** Let us consider the set of floating-point numbers to base b, with given exponent $m + 1$ and mantissa length p, including denormalized numbers.

- What is the smallest number in this set?
- What is the smallest positive number in this set?
- What is the smallest normalized number in this set?
- What is the largest number in this set?
- What do we get if we add up the smallest positive and the largest number?

Exercise 5.8 (⋆)**.** Write a program in C that prints mantissa and exponent of a single- and double-precision floating-point number in binary. Test your program on $1.0, -1.0, 0.5, \infty, -\infty, 0.0, -0.0, 0.1$. For the last example, compare with exercise 5.2.

[39] See `https://arxiv.org/pdf/2209.05433.pdf`

[40] Recall: A number z is positive if $z > 0$. A number z is non-negative if $z \geq 0$.

Exercise 5.9 (⋆)**.** Write a program in C that outputs double-precision floating-point numbers given their binary encoding. That is, write a C program where you reinterpret a `uint64_t` as `double` and output both.

Use this program for the following examples:

- The largest finite double-precision number.
- The smallest positive normalized double-precision number x.
- $2x$.
- The smallest positive de-normalized double-precision number y.
- $2y$.
- The largest de-normalized double-precision number.
- An example for not-a-number.
- Infinity.

Exercise 5.10 (⋆)**.** Write a C99 program that tests the following floating point operations:

- All four combinations of adding $+0$ and -0. What can we say about commutativity?
- All four combinations of subtracting $+0$ and -0. How do we relate the results to the previous examples?
- Same for multiplication. What can we say about commutativity?
- Same for the operator `==` to test equality.
- Since C99, in `math.h` we have a macro called `islessequal()`. Test it for all four combinations like above. Is this relation forming a partial order (i.e., is it reflexive, antisymmetric and transitive)?
- Look up what the compiler flag `-ffast-math` does for gcc or clang.

Take care to use the correct number literals in C and the correct format specifiers for `printf()`. (Hints: Note that `%+f` as format specifier always prints the sign. Maybe implement the tests as macros, to keep the code cleaner. Also have a look for `math.h` at `cppreference.com` or a man page. Note that there is also a `float.h`.)

Exercise 5.11 (⋆)**.** Write a function `... mean(..., size_t n)` in C that takes an array of n `fract` numbers and returns the arithmetic mean.

Take care to use the correct return type for `mean()`.

Exercise 5.12 (⋆⋆)**.** An averaging filter is a simple FIR filter that outputs the arithmetic mean of the last n samples. More formally, the time-discrete output signal (y_k) is computed from the input signal by (x_k) by

$$y_k = \frac{1}{n} \sum_{i=0}^{n-1} x_{k-i}.$$

There is a simple and common way to speed up computation by avoiding computing the sum all over again, but remembering the sum. Let us denote the sum by

$$S_k = \sum_{i=0}^{n-1} x_{k-i}$$

then we can compute the filter output and maintain the sum by

$$y_k = \frac{1}{n} S_k$$
$$S_{k+1} = S_k + x_{k+1} - x_{k-(n-1)}.$$

The last n samples of the input signal are typically held in a ring buffer, which is initialized to zero. This way we can output and update the filter in constant time per sample, which makes it real-time capable for online computations, e.g., for position profile computation in motion control.

Implement in C the filter using S_k. Define a preprocessor macro `REAL` which can be set to `double` or `long _Accum`, which allows to switch between floating-point and fixed-point computation. Output $y_1, \ldots, y_{210}$ for the filter size $n = 100$ and the input signal

$$x_k = \begin{cases} \pi & k \leq 100 \\ 0 & \text{otherwise.} \end{cases}$$

You should observe that the floating-point version accumulates an error and therefore y_{210} did not return to zero again, although the input signal has been zero for more than 100 samples.

You can use the attached code and fill in the missing parts to save some time.

Exercise 5.13 (⋆). Find the results of the following rounding operations:

$\mathrm{rd}_{3,10}(0.4567)$	$\mathrm{rd}_{3,10}(0.4564)$	$\mathrm{rd}_{3,10}(0.4595)$
$\mathrm{rd}_{2,2}(0.12)$	$\mathrm{rd}_{2,2}(0.13)$	$\mathrm{rd}_{2,2}(0.51)$

Exercise 5.14 (⋆). We consider $z = \pi$ and an approximation $\tilde{z} = \frac{22}{7}$. What are the absolute and relative errors?

Exercise 5.15 (⋆). What is the relative machine accuracy of the FP8 floating-point number format E4M3 from exercise 5.5? How many decimal digits does that correspond to?

Exercise 5.16 (⋆). We consider `double` variables in C. Find three numbers a, b, c of type `double` with the property

$$(a + b) + c \neq a + (b + c).$$

This example proves that the operator + on floating-point numbers is not associative. You may check your solution with C code. But choose your example in a way such that it is easy to explain, i.e., do not just plug in three random numbers and demonstrate the result.

Then find three such numbers a, b, c where the associative does hold. Again, choose examples where it is easy to explain the reasoning.

Exercise 5.17 (⋆). We set $\epsilon = 10^{-10}$ and consider the epsilon-based comparison operator $\doteq$. Find three numbers a, b, c of type `double` with the properties

$$a \dot{\leq} b \qquad b \dot{\leq} c \qquad \neg a \dot{\leq} c$$

This example proves that the epsilon-based operator $\dot{\leq}$ on floating-point numbers is not transitive. You may check your solution with C code.

Exercise 5.18 (★). Invoke the following python code, which generates a `numbers.txt` file with random floating point numbers. (It is reproducible, i.e., it will always create the same numbers.)

```
#!/usr/bin/env python3

import random
random.seed(0)

if __name__ == '__main__':
    with open('numbers.txt', 'w') as f:

        nmbs = [random.uniform(0.0, 1.0) for _ in range(100)] + \
               [random.uniform(0.0, 1e-6) for _ in range(10000)]
        random.shuffle(nmbs)

        for x in nmbs:
            f.write('%.17f\n' % x)
```

Read the numbers from the generated file and compute sums in C or C++ in three different ways:

- Read numbers in `double` precision, compute and output the sum.
- Read numbers in `float` precision, compute and output the sum.
- Read numbers in `float` precision, reorder the numbers in a clever way such that the precision of the sum becomes higher. Defend your cleverness, argue why your solution makes sense.

Compare and interpret the results. (Of course, you need to output about 17 decimal digits in order to make sense.)

Exercise 5.19 (★). Condition of simple operations.

1. Let us consider function $f\colon \mathbb{R} \to \mathbb{R}$ with $f(x_2) = x_1 \cdot x_2$, which is a multiplication of x_2 with a constant factor x_1. Determine the absolute and relative condition of f.
2. We consider the division $x_1 \oslash x_2$ and reinterpret it as a multiplication $x_1 \odot \frac{1}{x_2}$. However, instead of investigating the division per se, we instead investigate the reciprocal function $h\colon \mathbb{R} \setminus \{0\} \to \mathbb{R}\colon x \mapsto 1/x$. Determine the absolute and relative condition of h.
3. Bonus exercise: Determine the absolute and relative condition of the "full" multiplication function $g(x_1, x_2) = x_1 \cdot x_2$. (Further bonus: Show that $\kappa_{\text{rel}} \geq 2$.)

Exercise 5.20 (★★). Stability of algorithms: Implement a C program to solve the quadratic equation $x^2 + 2px + q = 0$ for x in three steps:

1. Implement both methods `QuadEuation` and `QuadEquationAlt` in C presented in section 5.4.3 using `float` numbers.
2. Compare the stability of both methods:
   - Check the outcome of both methods for $p = 40$ and $q = 10$. The second method should be closer to the exact solution $-0.12519592524622823962\ldots$.
   - Check the outcome of both methods for $p = -40$ and $q = 10$. The first method should be closer to the exact solution $79.874804074753771760\ldots$.

Put your results back into the left-hand side of the quadratic equation and check by how much the result deviates from zero.

3. Implement a third method `QuadEquationHybrid` that switches between the better of the two previous methods depending on p and q.

Exercise 5.21 (⋆)**.** Demonstrate how $1 - \cos(x)$ suffers from cancellation by a C program with single-precision numbers. Use an alternative trigonometric formula that does better. Use another formula via the Taylor series with a single term. (You may compare your results with double-precision numbers.)

Exercise 5.22 (⋆)**.** In section 5.4.3 we solved quadratic equations. The underlying numerical issue was cancellation due to subtraction of almost equal numbers. In general, when we have a formula $a - b$, and we want to avoid subtraction, we can transform this formula as follows:

$$a - b = \frac{(a-b)(a+b)}{a+b} = \frac{a^2 - b^2}{a+b}.$$

If we are lucky enough then $a^2 - b^2$ simplifies or transforms to something numerically more favorable than $a - b$. Apply this technique to the following two formulas:

$$-p + \sqrt{p^2 - q} \qquad \text{and} \qquad \sqrt{x+1} - \sqrt{x}$$

Exercise 5.23 (⋆)**.** Take your solution of exercise 5.22 for the formula $\sqrt{x+1} - \sqrt{x}$ and demonstrate the improvement in a C or C++ programm:

- Implement the original and the improved formula, for float and double numbers. (In C++, you may implement template functions and note that `std::sqrt` is overloaded for different number types.)
- Test each for the four combination for $x = 10^5$ and $x = 10^8$.

Exercise 5.24 (⋆)**.** What is the condition of the problem $\varphi(x) = \sqrt{x+1} - \sqrt{x}$ and how is this related to the findings of exercise exercise 5.23?

Chapter 6

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. GPUs are so important for machine learning, because they form massively parallel linear-algebra machines and the machine learning framework TensorFlow has its name from tensors, which are multi-linear generalizations of matrices.

The theory and practice of linear algebra is very well understood, powerful and developed. Hence, we often approximate non-linear settings by linear ones. Prime examples can be found in control theory or cryptanalysis of ciphers.

6.1 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 $b = (b_1, \ldots, b_n)$. Then we know from linear algebra that there is a unique $x \in \mathbb{R}^n$ such that $A \cdot x = b$. Moreover, if we would know the inverse Matrix A^{-1} of A already then we could simply calculate $x = A^{-1} \cdot b$.

In the following, we will discuss how to solve linear systems, how to compute the inverse A^{-1} and what we can say in terms of numerical stability. Then we generalize the problem setting to non-regular, rectangular matrices A, which leads us to approximation tasks and regression, the principle of least squares and certain matrix factorizations of A.

[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 $\det A \neq 0$.

6.2 Gaussian elimination

6.2.1 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:

Note that the solution space of a linear system is not altered if we add multiples of one row in A and b to another row. This enables us to apply such row additions in a way that turns A into a simpler form, a so-called right-triangular form, which is then much simpler to solve. The strategy is to achieve the following transformations for A (and correspondingly for b), where all elements of A below the diagonal become zero and $\bullet$ stands for some real numbers:

$$\begin{pmatrix} \bullet & \bullet & \bullet & \dots & \bullet \\ \bullet & \bullet & \bullet & \dots & \bullet \\ \bullet & \bullet & \bullet & \dots & \bullet \\ \vdots & & & & \vdots \\ \bullet & \bullet & \bullet & \dots & \bullet \end{pmatrix} \rightsquigarrow \begin{pmatrix} \bullet & \bullet & \bullet & \dots & \bullet \\ 0 & \bullet & \bullet & \dots & \bullet \\ 0 & \bullet & \bullet & \dots & \bullet \\ \vdots & \vdots & & & \vdots \\ 0 & \bullet & \bullet & \dots & \bullet \end{pmatrix} \rightsquigarrow \begin{pmatrix} \bullet & \bullet & \bullet & \dots & \bullet \\ 0 & \bullet & \bullet & \dots & \bullet \\ 0 & 0 & \bullet & \dots & \bullet \\ \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & \bullet & \dots & \bullet \end{pmatrix} \rightsquigarrow \dots \rightsquigarrow \begin{pmatrix} \bullet & \bullet & \bullet & \dots & \bullet \\ 0 & \bullet & \bullet & \dots & \bullet \\ 0 & 0 & \bullet & \dots & \bullet \\ \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & 0 & \dots & \bullet \end{pmatrix}$$

This is achieved by the following row reduction operations: In a first step we subtract from the i-th equation the a_{i1}/a_{11}-multiple of the first equation for all $2 \le i \le n$. We receive a new i-th row in the matrix A:

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

Note that we do not alter the solution of the original system, yet we introduced leading zeros to all equations below the first one. So after the first step we have this new equivalent system:

$$\left(\begin{array}{c|ccc} a_{11} & a_{12} & \dots & a_{1n} \\ \hline 0 & a_{22}^{(1)} & \dots & a_{2n}^{(1)} \\ 0 & a_{32}^{(1)} & \dots & a_{3n}^{(1)} \\ & & \vdots & \\ 0 & a_{n2}^{(1)} & \dots & a_{nn}^{(1)} \end{array}\right) \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{pmatrix} = \left(\begin{array}{c} b_1 \\ \hline b_2^{(1)} \\ b_3^{(1)} \\ \vdots \\ b_n^{(1)} \end{array}\right), \tag{6.1}$$

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 $1 \le i, j \le n$. 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 $3 \le i \le n$. This introduces leading zeros in the second column of row 3 to n:

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} \\ 0 & a_{22}^{(1)} & a_{23}^{(1)} & \dots & a_{2n}^{(1)} \\ 0 & 0 & a_{32}^{(2)} & \dots & a_{3n}^{(2)} \\ & & & \vdots & \\ 0 & 0 & a_{n2}^{(2)} & \dots & a_{nn}^{(2)} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2^{(1)} \\ b_3^{(2)} \\ \vdots \\ b_n^{(2)} \end{pmatrix}. \tag{6.2}$$

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

$$\begin{aligned} r_{11} \cdot x_1 + r_{12}x_2 + \cdots + r_{1n}x_n &= b'_1 \\ r_{22}x_2 + \cdots + r_{2n}x_n &= b'_2 \\ &\vdots \\ r_{nn}x_n &= b'_n \end{aligned}, \tag{6.3}$$

which has a solution space identical to 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 $(x_1, \ldots, x_n)$. We can summarize the method of Gaussian elimination as follows:

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

Here we described the algorithm for matrices of real numbers. We can do the same also with complex numbers. In fact, we can do it for any algebraic field[2]. This is relevant for cryptographic protocols, many of which are built on finite fields. Another example is persistent homology in topological data analysis, where the so-called boundary matrix algorithm is essentially Gaussian elimination for matrices over $\mathbb{Z}_2$ (for practical applications).[3]

6.2.2 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[4] 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 the 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

[2] In abstract algebra, a field is a set F with two binary operations $+$ and $\cdot$, such that the usual laws of associativity, distributivity, commutativity and the existence of neutral and inverse elements holds, like for the usual triple $(\mathbb{R}, +, \cdot)$. More precisely, $(F, +)$ has to form a commutative group with a neutral element $0 \in F$, $(F \setminus \{0\}, \cdot)$ has to form a commutative group with neutral element $1 \in F$, and $\cdot$ has to distribute over $+$, i.e. $x \cdot (y + z) = x \cdot y + x \cdot z$. A set X with a binary operator $\circ\colon X \times X \to X$ forms a group $(X, \circ)$ if associativity and the existence of inverse and the neutral elements hold, like in $(\mathbb{Z}, +)$, but not in $(\mathbb{Z}, \cdot)$

[3] By $\mathbb{Z}_2$ we denote the quotient space $\mathbb{Z}/2\mathbb{Z}$, which is $\mathbb{Z}$ modulo 2, which has two elements and naturally corresponds to Boolean algebra. The addition $+$ is addition modulo 2, and hence like the logical XOR operator, and $\cdot$ is like the logical AND operator.

[4] 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 $\det A = 0$.

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. The solution space is then either empty (if the system cannot be solved) or it is an affine-linear subspace of dimension at least zero. When the dimension is zero then the space consists of a single solution, a single point.

6.2.3 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[5] operations. Altogether we have

$$\sum_{s=1}^{n-1}(n-s)^2+(n-s)=\sum_{s=0}^{n-1}s^2+s=\underbrace{\frac{(n-1)n(2n-1)}{6}}_{\text{in } A}+\underbrace{\frac{(n-1)n}{2}}_{\text{in } b}\sim\frac{n^3}{3} \tag{6.4}$$

modifications, where $\sim$ means *asymptotically equivalence*, cf. section 1.2.1. 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. (6.3) 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

$$\sum_{s=2}^{n}(s-1)=\sum_{s=1}^{n-1}s=\frac{(n-1)n}{2}\sim\frac{n^2}{2} \tag{6.5}$$

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.

6.2.4 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 $b_1,\ldots,b_n$ and solve all the systems $A\cdot x=b_i$.

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

$$A\cdot X=B.$$

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. (6.4) and eq. (6.5) we therefore

[5] 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.

have asymptotically

$$\frac{n^3}{3} + r\frac{n^2}{2} + r\frac{n^2}{2} \quad = \quad \frac{n^3}{3} + rn^2 \tag{6.6}$$

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, which is in $O(n^3)$, by eq. (6.6). 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 [42] for details.

6.3 Linear regression

6.3.1 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 $x \in \mathbb{R}^n$ solves the system as close as possible, i.e., which 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

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

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.[6] Least squares are a standard tool in regression analysis, they are a fundamental tool in data fitting[7] and often are used as an introduction into machine learning of linear models.

[6] 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.

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

6.3.2 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 $\|A \cdot x - b\|$. To this end, we consider the transformation $A\colon \mathbb{R}^n \to \mathbb{R}^m$ with a tiny abuse of notation. It transforms a point $x \in \mathbb{R}^n$ into a point $A \cdot x \in \mathbb{R}^m$. The set of all such transformed points form the set $\operatorname{im} A$, which is called the *image* of A:

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

Note that $\operatorname{im} A$ lives in $\mathbb{R}^m$, and from linear algebra we know that it forms in fact a linear subspace[8] 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. 6.1.

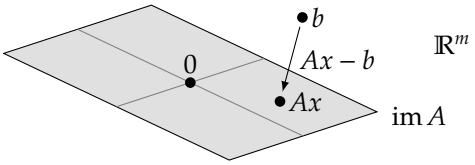


Figure 6.1: When we orthogonally project b onto the linear subspace $\operatorname{im} A$ then we hit a certain Ax which minimizes $\|Ax' - b\|$ among all $x' \in \mathbb{R}^n$.

The question is how to find this projection, without knowing $\operatorname{im} A$. 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 $A \cdot x' = x' \cdot A^\dagger$, with a slight abuse of notation: x' denotes a column vector right of A and a row vector left of $A^\dagger$. We therefore 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*

$$(A^\dagger A) \cdot x = (A^\dagger b). \tag{6.7}$$

Hence, any $x \in \mathbb{R}^n$ that solves eq. (6.7) 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 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 [37] for details.)

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

$$x = (A^\dagger A)^{-1} A^\dagger \cdot b \tag{6.8}$$

[8] Dt. Untervektorraum

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 may actually not be invertible, 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*[9] of A. The pseudoinverse is closely linked to the *singular value decomposition*, see [42] 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.

6.3.3 Fitting functions

Linear regression

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. 6.2 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 compatible with the framework of overdetermined linear equation systems, namely $f(x_i) = y_i$ for all $1 \leq i \leq m$, or in matrix notion:

$$\underbrace{\begin{pmatrix} 1 & x_1 \\ \vdots & \vdots \\ 1 & x_m \end{pmatrix}}_{A} \cdot \begin{pmatrix} d \\ k \end{pmatrix} = \begin{pmatrix} y_1 \\ \vdots \\ y_m \end{pmatrix} \tag{6.9}$$

If not all x_i are equal then A has rank 2, which is maximal, and by virtue of eq. (6.8) 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. (6.8) for the function $f(x) = kx + d$ plotted in fig. 6.2. 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[10]. Then eq. (6.7) has infinitely many solutions and there are infinitely many best fitting functions.[11]

General regression

We can generalize this idea of fitting functions as follows. If we recap eq. (6.9) then we can interpret this equation as follows: We have a linear combination of two base functions, the

[9] 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.

[10] Not regular, not invertible.

[11] 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 $y = \frac{1}{m} \sum_i y_i$. So the point (x_i, y) is the center of gravity of the data points.

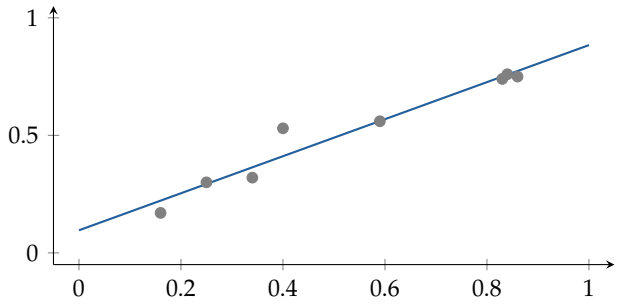


Figure 6.2: Eight data points give rise to an overdetermined linear equation system for a fitting linear function.

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 $g_1, \ldots, g_n$ and ask for the linear combination $f = \sum_{i=1}^{n} \alpha_i g_i$ for the data points $(x_1, y_1), \ldots, (x_m, y_m)$ 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

$$\underbrace{\begin{pmatrix} g_1(x_1) & \cdots & g_n(x_1) \\ & \vdots & \\ g_1(x_m) & \cdots & g_n(x_m) \end{pmatrix}}_{A} \cdot \begin{pmatrix} \alpha_1 \\ \vdots \\ \alpha_n \end{pmatrix} = \begin{pmatrix} y_1 \\ \vdots \\ y_m \end{pmatrix}. \tag{6.10}$$

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*[12] of this vector space, i.e., they shall be linearly independent. Otherwise the columns of A in eq. (6.10) 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 $1, \sin(x), \cos(x), \ldots, \sin(nx), \cos(nx)$ 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 $1, x, x^2, \ldots, x^n$ to form polynomials of the form $\sum_{i=0}^{n} \alpha_i x^i$. We call this *polynomial regression*. These base functions span the vector space of polynomial functions up to degree n. In this case eq. (6.10) yields

$$\begin{pmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^n \\ & & & \vdots & \\ 1 & x_m & x_m^2 & \cdots & x_m^n \end{pmatrix} \cdot \begin{pmatrix} \alpha_0 \\ \vdots \\ \alpha_n \end{pmatrix} = \begin{pmatrix} y_1 \\ \vdots \\ y_m \end{pmatrix}. \tag{6.11}$$

[12]The vectors $v_1, \ldots, v_d$ 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 $\lambda \in \mathbb{R}$.

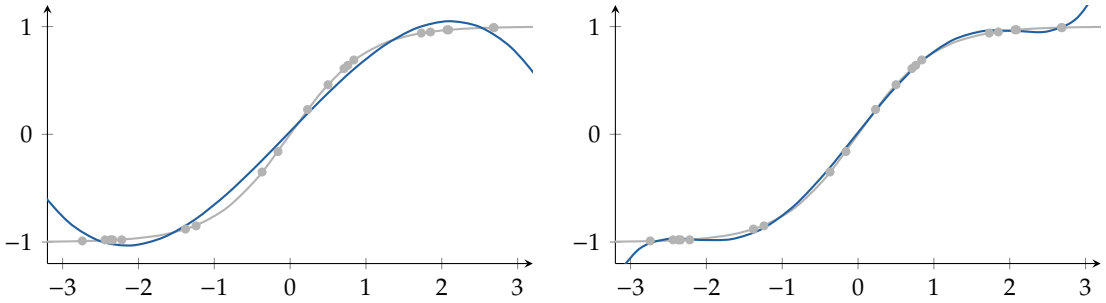


Figure 6.3: Polynomial fit of data points that sample tanh at 20 uniformly random points on $[-3, 3]$. Left: The polynomial of degree 3. Right: The polynomial of degree 5.

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. (6.9) versus eq. (6.11). In fig. 6.3 two examples of approximating polynomials are given. They have both been computed by eq. (6.8) applied to eq. (6.11).

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
```

Supervised learning

In an even more generalized setting we and end up in *supervised machine learning*. We are given a *training set* of inputs $x_i \in \mathbb{R}^n$ and desired outputs $y_i \in \mathbb{R}^n$, which are sometimes called *labels*.

We want to model the relationship between the input and the output by certain functions $f\colon \mathbb{R}^n \to \mathbb{R}^m$, such as feed-forward neural nets. We describe this function f through a list of parameters θ, like the weights in the neural net. That is, we consider a whole family of models f_θ and call this family the *hypothesis space*. In this hypothesis space, we look for the parameters θ that yield the model (hypothesis) that describes the data best. Furthermore, in machine learning, the error is called *loss* and often denoted by ℓ. A common choice is the 2-norm[13] of the error, i.e.,

$$\ell = \sum_i (y_i - f_\theta(x_i))^2.$$

We now look for the parameters θ that minimize the loss ℓ. However, interesting families of models (i.e., hypothesis spaces) are too complex such that we can directly compute the optimum via the Moore-Prenrose inverse. Instead, numerical optimization is applied, such as *stochastic gradient descent (SGD)* or adaptive versions of it, like Adam or AMSGrad. This is called *training* in machine learning.

[13] If we assume normal distributed errors on the labels then the error minimizer corresponds to the maximum likelihood estimator. This is the statistical justification for the model to "explains" the data best.

6.3.4 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

$$QA = \begin{pmatrix} R \\ 0 \end{pmatrix}, \tag{6.12}$$

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.

The question is now how to solve $A \cdot x = b$ given the above QR decomposition, i.e., what x is minimizing $\|A \cdot x - b\|$? Note that Q does not change lengths of vectors, so we can actually minimize $\|QA \cdot x - Qb\|$ instead. Assume we know Q then we can also compute

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

with a vector $c \in \mathbb{R}^n$ and $d \in \mathbb{R}^{m-n}$. Now we observe that for any $x \in \mathbb{R}^n$

$$\|Ax - b\|^2 = \|Q(Ax - b)\|^2 = \|QAx - Qb\|^2 = \left\| \begin{pmatrix} Rx - c \\ -d \end{pmatrix} \right\|^2 = \|Rx - c\|^2 + \|d\|^2 \geq \|d\|^2. \tag{6.13}$$

Hence, the left-hand side is minimized exactly when $Rx = c$. We therefore look for the solution x of $Rx = c$ to minimize $\|Ax - b\|$. Now 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

$$A = Q^\dagger \begin{pmatrix} R \\ 0 \end{pmatrix}. \tag{6.14}$$

The essential step in the QR method is the computation of the matrix Q. A numerically stable algorithm to compute Q is based on so-called 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. (6.12). The product $Q_n \cdots Q_1$ is then Q in eq. (6.12). The other algorithm uses so-called 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. (6.14).)

```
>>> 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.

6.3.5 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 6.3.1 that x therefore minimizes the error expression

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

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 $(a_{k1}, \ldots, a_{kn})$ are all of equal length for $1 \leq k \leq m$. This way all equations obtain a kind of "unit weight".

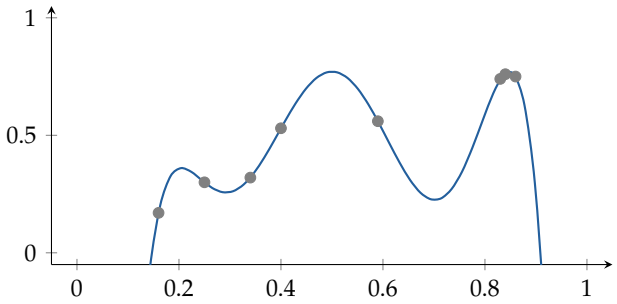


Figure 6.4: The polynomial of degree 6 fitting 8 data points shows strong oscillations.

Regularization. For certain systems the solution may not "behave very well". Consider for instance the polynomoial of degree 6 that is shown in fig. 6.4 and fits the point set from fig. 6.2. 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

$$\int_a^b f''(x)^2 \, \mathrm{d}x, \tag{6.15}$$

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

$$f''(x) = \sum_{i=2}^{n} \alpha_i x^{i-2} \cdot i(i-1) \tag{6.16}$$

We would like to minimize the error term given in eq. (6.15) for the α_k, so we set the partial derivatives to zero. We see from eq. (6.16) that α_0, α_1 actually do not matter so we only consider the $n-1$ equations for $\alpha_2, \ldots, \alpha_n$:

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

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

$$(c_{ki}) \cdot (\alpha_i) = 0 \qquad \text{where} \qquad c_{ki} = \begin{cases} 0 & \text{for } i \leq 2 \\ ik(i-1)(k-1)\frac{b^{i+k-3} - a^{i+k-3}}{i+k-3} & \text{for } i > 2 \end{cases} \tag{6.17}$$

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. (6.11) for the approximation of the data points, we end up with the following system:

$$\left(\begin{array}{ccccc} 1 & x_1 & x_1^2 & \dots & x_1^n \\ & & & \vdots & \\ 1 & x_m & x_m^2 & \dots & x_m^n \\ \hline 0 & 0 & \lambda c_{22} & \dots & \lambda c_{2n} \\ & & & \vdots & \\ 0 & 0 & \lambda c_{n2} & \dots & \lambda c_{nn} \end{array}\right) \cdot \begin{pmatrix} \alpha_0 \\ \vdots \\ \alpha_n \end{pmatrix} = \left(\begin{array}{c} y_1 \\ \vdots \\ y_m \\ \hline 0 \\ \vdots \\ 0 \end{array}\right). \tag{6.18}$$

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 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 $\lambda^2 \int_a^b f''(x)^2 \, \mathrm{d}x$ the *regularization term* and the method is called *regularization*. If we choose $\lambda = 0$ then we have the original non-regularized version, see fig. 6.5 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.

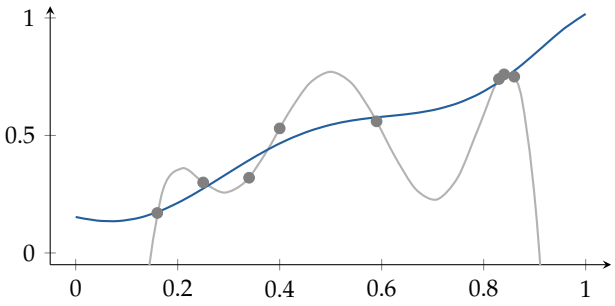


Figure 6.5: Regularized solution for a polynomial of degree 6 as in fig. 6.4. The black graph shows the result for $\lambda = 0.001$ and the gray graph for $\lambda = 0$, which equals therefore the non-regularized version. The strain energy is computed over the interval $[0, 1]$.

Regularizing training of neural nets. Similar ideas to equilibration and regularization is applied when training neural nets. In particular deep neural nets possess many model parameters, which are to be trained through gradient descent optimization, see section 6.3.3.

This gives those nets a high learning capacity – the hypothesis space is large – but they are also prone to overfitting leading to exactly the behavior as illustrated in fig. 6.4. This oscillating behavior hurts the generalization capability of the neural net, because similar input leads to strongly different output.

To mitigate overfitting (and to stabilize gradient descent training), a standard technique is to add L_1- or L_2-regularization: It causes a normalization of the weight vectors formed by the weights of neurons. This is similar to normalizing the coefficient vectors as described above for equilibration and causes a smoothed behavior as illustrated for regularization in fig. 6.5.

6.4 Summary

We discussed algorithms to solve linear systems and investigated beyond the usual time and space resources also the numerical quality. For didactical purposes we did so for the Gaussian elimination method, we also now as paper-and-pen method, although this is not the first choice for an actual implementation. We split the chapter into two parts: regular linear systems and overdetermined linear system. The latter allows for regression, i.e., for building mathematical models from data. We start from the first principle of error minimization, leading us to normal equations and QR matrix decomposition. The latter is one particular case of a whole class of matrix factorization methods.

Finally, we discussed how to influence the mathematical model to our favor, in terms of equilibration and regularization. In other words, this chapter builds up core foundations of machine learning.

6.5 Exercises

Exercise 6.1 (⋆⋆)**.** We are given the following linear equation system:

$$\begin{aligned} 2x + 3z &= 12 \\ -4x + 2y &= 8 \\ 2x + y + 4z &= 24 \end{aligned}$$

Perform Gaussian elimination and back substitution by hand, i.e., we are not just interested in the result but in the method. That is, follow the precise steps of the algorithm in the lecture note. Perform the pivoting as described in the lecture.

Exercise 6.2 (⋆)**.** We consider the following two linear systems:

$$\begin{aligned} x_2 &= y_1 \\ -x_1 &= y_2 \\ x_3 &= y_3 \end{aligned} \qquad \text{and} \qquad \begin{aligned} y_1 &= z_1 \\ y_3 &= z_2 \\ -y_2 &= z_3 \end{aligned} .$$

First, translate both systems into a matrix notation. Then turn the two systems into one system where (z_1, z_2, z_3) are expressed in terms of (x_1, x_2, x_3). Finally, solve the resulting system for (x_1, x_2, x_3) given $(z_1, z_2, z_3) = (1, 2, 3)$.

Bonus question: What is the geometric interpretation of these linear transformations and this exercise?

Bonus questions: What have we learned concerning a multi-layer perceptron (MLP) neural network with a layer having linear activation functions?

Exercise 6.3 (⋆)**.** We consider the following two linear systems:

$$\begin{aligned} 4x_1 + 5x_2 - 3x_3 &= y_1 \\ 3x_1 - 2x_2 + 2x_3 &= y_2 \end{aligned} \qquad \text{and} \qquad \begin{aligned} 4y_1 + 5y_2 &= z_1 \\ 3y_1 - 2y_2 &= z_2. \\ 2y_1 + 3y_2 &= z_3 \end{aligned}$$

As in the previous example, translate both systems into a matrix notation. Then turn the two systems into one system where (z_1, z_2, z_3) are expressed in terms of (x_1, x_2, x_3). What is the rank of the resulting coefficient matrix?

Bonus questions: What is a nice way to quickly see the answer by looking at the two original coefficient matrices? This is the underlying idea of Low Rank Adapation (LoRA) for parameter efficient fine tuning of large neural networks.

Exercise 6.4 (⋆)**.** Invert the coefficient matrix of the following linear system by Gaussian elimination with multiple right-hand sides:

$$\begin{aligned} 2x_2 &= y_1 \\ -2x_1 &= y_2 \end{aligned}.$$

Interpret the result geometrically.

Exercise 6.5 (⋆)**.** Show that $(AB)^\dagger = B^\dagger A^\dagger$ for matrices A and B.

Exercise 6.6 (⋆)**.** We are given the following overdetermined linear system in two variables.

$$\begin{aligned} 2x_1 + 3x_2 &= 4 \\ 3x_1 + 4x_2 &= 5 \\ 4x_1 + 5x_2 &= 5 \end{aligned}$$

Write down the normal equation system as in eq. (6.7). Check that his is a regular linear system by computing the rank of $A^\dagger A$, e.g., in Python or Matlab. Solve the system in Python or Matlab.

Exercise 6.7 (⋆)**.** Show that the Moore-Penrose inverse of a regular matrix A is equal to A^{-1}.

Exercise 6.8 (⋆⋆)**.** Solve this overdetermined linear system $A \cdot x = b$ by computing the Moore-Penrose inverse of A:

$$\begin{aligned} 2x_1 + 3x_2 &= 4 \\ 3x_1 + 4x_2 &= 5 \\ 4x_1 + 5x_2 &= 5 \end{aligned}$$

Furthermore, compute $A \cdot x - b$ for your solution $x = (x_1, x_2)$. Then choose some random vector $x' \in \mathbb{R}^2$ and compute $Ax' \cdot (Ax - b)$.

You may use Matlab or Python for your implementation. You may use a library implementation for matrix inversion, multiplication or tranposition, but you may not use a library function to directly compute the Moore-Penrose inverse.[14]

Exercise 6.9 (⋆⋆)**.** In Python or MATLAB we would like to fit a function $f(x) = k \cdot x + d$ through tree (x, y)-points $(0, 3), (1, 4), (2, 3)$ such that the least square error is minimized.

- What is the coefficient matrix A and the vector b of the corresponding overdetermined linear system?
- Compute the pseudo inverse of A using the formula in the lecture notes. (You do not need to compute it by hand, but you shall compute it by matrix operations and not just via a library function.)

[14] In Python, using numpy, you can use `A@B`, `A.T`, `np.linalg.inv(A)` for matrix multiplication, transpose and inversion of matrices A and B, where a matrix is of type `np.array`.

- Compute the resulting function f using the pseudo-inverse and plot the function graph.

Exercise 6.10 (⋆). Sample the sine function and fit a cubic polynomial. More precisely, evaluate $\sin(x)$ for $x \in \{-0.5\pi, -0.3\pi, -0.1\pi, 0.1\pi, 0.3\pi, 0.5\pi\}$ and fit a polynomial of degree 3 by means of the Moore-Penrose inverse and eq. (6.11). Plot the data, the polynomial function and the ground truth (i.e, the sine function) over the range $[-2, 2]$.

Bonus question: What do you observe when looking at the coefficients received?

Bonus question: What can we conclude about the basis functions for approximation if we know that the function to be approximated is an odd function? In machine learning, we call this an inductive bias for the hypothesis space.

Exercise 6.11 (⋆). In exercise 6.9, add a weight to the third point in the sense of equilibration. That is, we multiply the equation in the linear system, which belongs to the third point, by a factor λ. We denote by f_λ the resulting function $f_\lambda(x) = k \cdot x + d$ where λ is the weight of the third point.

Plot the resulting functions $f_0, f_{0.5}, f_1, f_2$ in Python or MATLAB. Interpret the results according to the different values of λ.

Exercise 6.12 (⋆). Sample the function $x \mapsto \sin(x + 0.1)$ uniformly at 11 points in the interval $[0, 2\pi]$. Fit a trigonometric polynomial with base functions $1, \sin(x), \cos(x), \sin(2x), \cos(2x)$ by means of Moore-Penrose. Plot the result on the interval $[-2, 8]$.

Bonus: Also fit a polynomial of degree 5, and compare the result.

Remark: This exercise is related to *physics-informed machine learning*. If we know something about the mathematical structure of our solutions from physical reasoning then we can exploit this knowledge by choosing our hypothesis space accordingly.

Exercise 6.13 (⋆). Let Q be an orthogonal matrix. Prove mathematically that $Q^{-1} = Q^\dagger$. (Hint: Show that $Q^\dagger \cdot Q = 1$ given that every element in $Q^\dagger \cdot Q$ is the inner product of two vectors.)

Exercise 6.14 (⋆). The documentation of numpy at `https://docs.scipy.org/doc/numpy/reference/generated/numpy.linalg.qr.html` defines the QR decomposition of a matrix A as $A = Q \cdot R$, which is slightly different than the one in the lecture notes.

Furthermore, the documentation says that the solution of the linear system $A \cdot x = b$ is given by

$$x = R^{-1} \cdot Q^\dagger \cdot b.$$

- Why is this so?
- What are the dimensions of the matrices Q and R?

Exercise 6.15 (⋆). We check the validity of eq. (6.13) of the lecture notes for a concrete example. That is, we check that $\|A \cdot x - b\| = \|d\|$ for the solution x of the following system:

$$\begin{pmatrix} 2 & 3 \\ 4 & 5 \\ 6 & 7 \end{pmatrix} \cdot x = \begin{pmatrix} 1 \\ 2 \\ 1 \end{pmatrix}.$$

Compute the solution x using the formula of the previous exercise. Note that numpy also knows a mode "complete" for `np.linalg.qr()`, which helps to obtain the matrices Q and R as in the lecture notes. (It is easier to call `qr()` twice, i.e., using two different modes.) You may also use `np.linalg.norm()`.

Chapter 7

Polynomial interpolation

7.1 Motivation

We often face the situation, where a function f is not given by a closed expression, but at finitely many positions $x_1, \ldots, x_n$ 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]

7.2 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]

$$f(x) = \sum_{k=0}^{\infty} a_k (x - x_0)^k, \tag{7.1}$$

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. (7.1):

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

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.

Processors – or rather FPUs – typically do not provide machine instructions for the exponential, logarithmic or trigonometric functions. Or maybe we would like to implement these functions in an FPGA using addition and multiplication only. Or maybe we want to trade accuracy versus speed. In all these 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 $[-0.25, 0.25]$.

7.3 Single interpolation polynomials

7.3.1 Existence

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

$$p(x_i) = y_i \tag{7.2}$$

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. (7.2), namely

$$\begin{pmatrix} 1 & x_1 & \ldots & x_1^{n-1} \\ & & \vdots & \\ 1 & x_n & \ldots & x_n^{n-1} \end{pmatrix} \cdot \begin{pmatrix} a_0 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix}, \tag{7.3}$$

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

7.3.2 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 $1 \le i \le n$. 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

$$f(x) - p(x) = \frac{f^{(n)}(\xi)}{n!} \cdot w(x) \tag{7.4}$$

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

$$\|f - p\| \le \frac{1}{n!} \, \|f^{(n)}\| \, \|w\|. \tag{7.5}$$

The left-hand side of eq. (7.5) is the maximum interpolation error on I, which 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. 7.1. More precisely, the extreme values of w get larger towards the boundary of I. This is known as *Runge's phenomenon*.

If we, however, redistribute the nodes to 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

$$x_i = \cos \frac{2i - 1}{2n} \pi \tag{7.6}$$

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. 7.1. The nodes given by eq. (7.6) are called *Chebyshev nodes* and they are the roots of the so-called *Chebyschev polynomial* of n-th degree. If we insert this into eq. (7.5), for Chebyshev nodes we therefore get an error bound of

$$\|f - p\| \le \frac{\|f^{(n)}\|}{n! \, 2^{n-1}}. \tag{7.7}$$

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. 6.4 in chapter 6. In fig. 7.2 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. (7.5) the term $\|f^{(n)}\|$ can grow towards ∞ faster than $\|w\|/n!$ goes to zero.

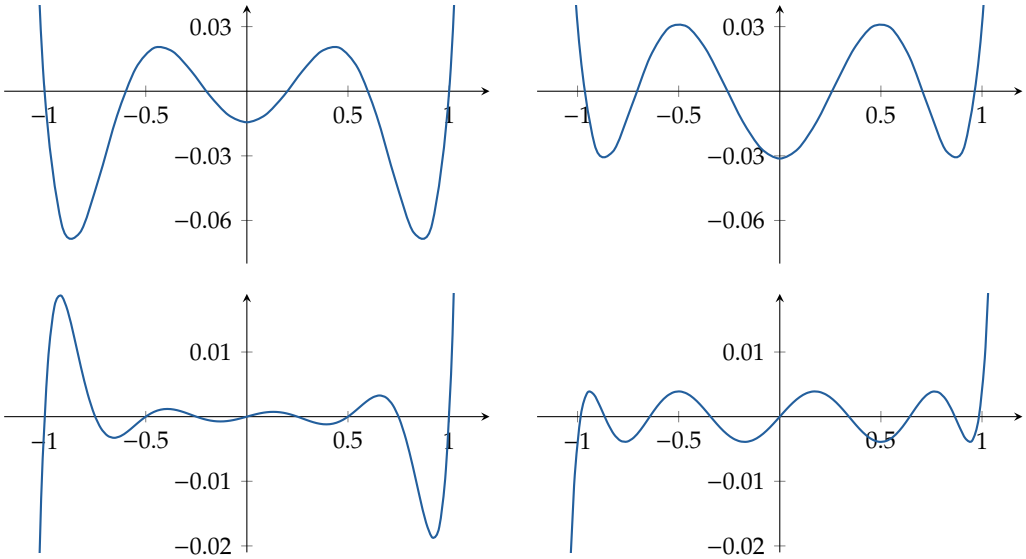


Figure 7.1: The node polynomial $w(x)$. Top: Polynomials of degree 6. Bottom: Polynomials of degree 9. Left: Uniform nodes on $[-1, 1]$. Right: Chebyshev nodes.

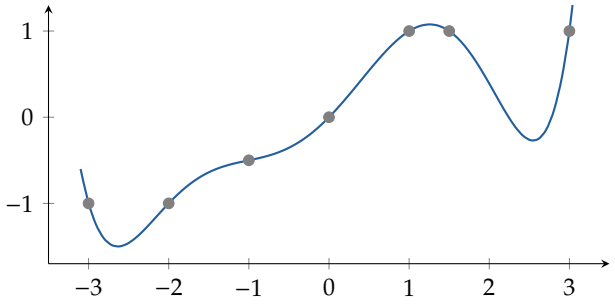


Figure 7.2: An interpolation polynomial of degree 6 that shows typical oscillation behavior.

7.3.3 Computing interpolation polynomials

In this section, we will introduce two different methods to compute interpolation polynomials: Lagrange's formulas and the Neville tableau. Of course, it would be possible to compute the interpolation polynomial by solving the linear system in eq. (7.3), but we can in fact compute them directly without paying the costs of 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 such k a polynomial L_k of degree $n-1$ that is 1 at x_k, but 0 at all other nodes, i.e., $L_k(x_i) = \delta_{ik}$. This directly leads to the sought interpolation polynomial

$$p(x) = \sum_{k=1}^{n} y_k L_k(x), \tag{7.8}$$

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:

$$L_k(x) = \prod_{\substack{i=1 \\ i \neq k}}^{n} \frac{x - x_i}{x_k - x_i}. \tag{7.9}$$

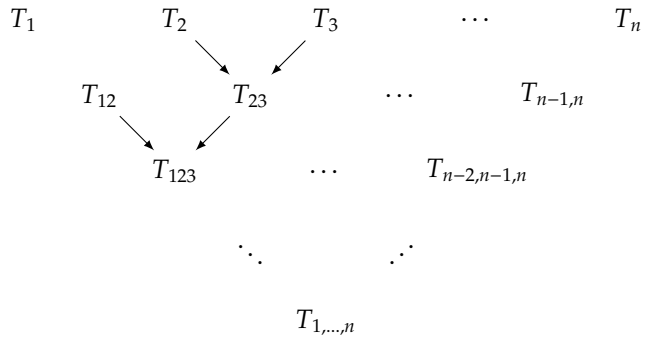
Plugging everything together gives *Lagrange's formula* for the interpolation polynomial:

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

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 [42] for details. It follows the classical approaches of *divide and conquer* and *dynamic programming* 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. In general, $T_{i,\dots,k}$ interpolates $x_i, \dots, x_k$ and eventually 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:

$$T_i(x) = y_i \tag{7.10}$$

$$T_{i,\dots,i+m}(x) = \frac{(x - x_{i+m}) T_{i,\dots,i+m-1}(x) + (x_i - x) T_{i+1,\dots,i+m}(x)}{x_i - x_{i+m}}. \tag{7.11}$$

Plainly implementing this recursion leads to many re-computations of the same sub-results, i.e., $T_{2,}$ is reached in multiple ways in the above tableau. A standard technique of dynamic

programming we discussed in section 2.3.1 is *memoization*: We remember the sub-results instead of re-computing them. This makes the total effort proportional to the size of the tableau.

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}:

$$T_{i,i+1} = \frac{(x - x_{i+1})y_i + (x_i - x)y_{i+1}}{x_i - x_{i+1}} \tag{7.12}$$

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.

Further methods

Lagrange's formula and the Neville tableau are two ways to compute the same interpolation polynomials. They lead to two different *forms* of the same polynomial. In fact, there is a rich history of various such forms. For instance there is also the *Newton form*, which is based on so-called *divided differences* and is related to Taylor series. There are further variants by Gauss, Stirling and Bessel.In [42], however, we find Lagrange and Neville as the two main methods.

7.4 Splines

7.4.1 Motivation

We learned in section 7.3 that polynomials of high degree tend to oscillate, as illustrated in fig. 7.2. For certain applications we could, of course, use a piecewise linear function that interpolates between the points. So let $x_1 < \dots < x_n$ denote n nodes with function values $y_1, \dots, y_n$. As illustrated in fig. 7.3a, 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}:

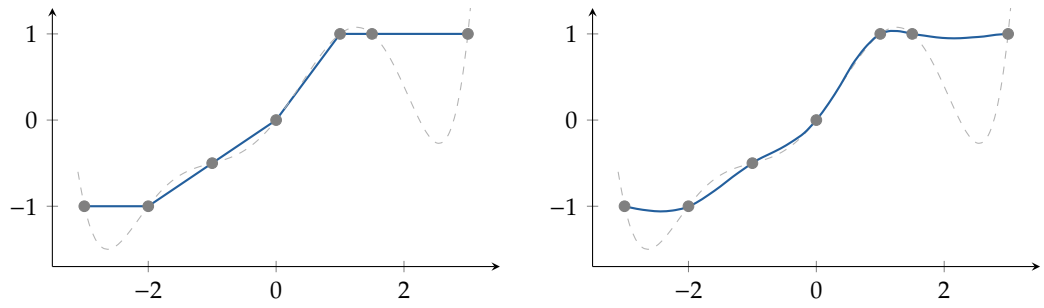
$$p_i(x) = \lambda\, y_i + \mu\, y_{i+1} \quad \text{with} \quad \lambda = \frac{x - x_{i+1}}{x_i - x_{i+1}} \text{ and } \mu = 1 - \lambda = \frac{x_i - x}{x_i - x_{i+1}}.$$

These are exactly the polynomials $T_{i,i+1}$ of the first row in the Neville tableau, see also eq. (7.12).

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.

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



(a) Piecewise linear function

(b) Natural cubic spline

Figure 7.3: Spline interpolation of the points in fig. 7.2. The dashed line is the interpolation polynomial of degree six. Left: A piecewise linear function (spline of degree 1). Right: The natural cubic spline.

7.4.2 Cubic splines

By a cubic spline f we mean a twice continuously differentiable spline of degree three that interpolates a tabulated function $(x_1, y_1), \ldots, (x_n, y_n)$. 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 $[x_i, x_{i+1}]$:

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

The first two equations establish the interpolation property. The next three equations establish C^0-, C^1-, and C^2-continuity, respectively. 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).

[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.

Figure 7.3b 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 section A.2.

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. 7.3b:

```
>>> 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)
```

7.5 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. (7.4), 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.

[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.

Two-point formula. In the simplest case we use $n = 2$ interpolation nodes and therefore receive an interpolation polynomial of degree 1. Following eq. (7.12) 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*

$$f'(x) \approx p'(x) = \frac{y_2 - y_1}{h} \tag{7.13}$$

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. 7.4a.

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. (7.8) 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

$$p'(x) = y_1 L_1'(x) + y_2 L_2'(x) + y_3 L_3'(x). \tag{7.14}$$

Let us compute:

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

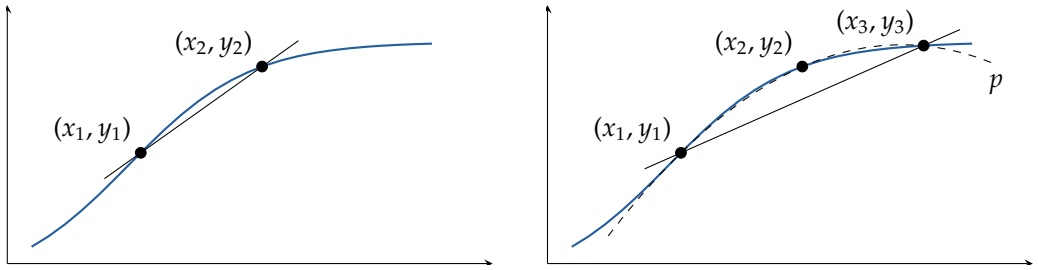
In order to compute f' at the middle node x_2 we obtain the so-called *central three-point formula*:

$$f'(x_2) \approx p'(x_2) = \frac{y_1 \cdot (-h) + y_3 \cdot h}{2h^2} = \frac{y_3 - y_1}{2h}. \tag{7.15}$$

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. (7.15) looks quite like eq. (7.13) 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. 7.4b. 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. 7.4b 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. (7.14), but we plug in the left resp. right node:

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



(a) Two-point formula

(b) Central three-point formula

Figure 7.4: Numerical derivative of $\tanh(x)$ using interpolation polynomials p with two (left) and three (right) nodes.

7.6 Numerical integration

We start with a similar situation as for numerical differentiation: Let $f\colon [a,b] \to \mathbb{R}$ be the function to be integrated. We replace f by a polynomial p and consider the integral of p over $[a,b]$ instead. However, while computing $f'(x)$ is a local problem at the position x, in order to integrate f, we have to consider f globally on $[a,b]$.

Remember that we learned two approaches for a polynomial interpolation of f in this chapter: A single polynomial or a spline. In the first case we obtain the basic integration formulas in section 7.6.1 and in the second case we obtain extended formulas section 7.6.2.

7.6.1 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

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

where $0 \le k \le n$ 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. (7.8) 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

$$A_k = \int_a^b L_k(x)\ \mathrm{d}x. \tag{7.16}$$

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. (7.9) 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. (7.16) 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

$$\alpha_k^{(n)} = \int_0^n \prod_{\substack{i=1 \\ i \neq k}}^{n} \frac{t - i}{k - i} \, \mathrm{d}t \tag{7.17}$$

we have

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

This now gives us the so-called *(closed) Newton-Cotes formula*[9] of degree n:

$$\int_a^b f(x) \, \mathrm{d}x \approx h \sum_{k=0}^{n} f(x_k)\alpha_k^{(n)}. \tag{7.18}$$

All that remains is to make a choice about n and to compute $\alpha_0^{(n)}, \ldots, \alpha_n^{(n)}$. Different choices of n then give us different *integration rules*, e.g., the Trapezoidal rule, Simpson's rule, Simpson's 3/8 rule, Boole's rule for $n = 0, 1, 2, 3$, respectively. In the following, we compute the first two examples.

Trapezoidal rule. We start with the case $n = 1$ and solve eq. (7.17):

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

Plugging the results in eq. (7.18) gives the *trapezoidal rule*:

$$\int_a^b f(x) \, \mathrm{d}x \approx h \, \frac{f(a) + f(b)}{2} \tag{7.19}$$

We can interpret this rule as replacing f by a single linear function interpolating at a and b, whose integral corresponds to the area of the resulting trapezoid.

[9] We can apply the very same procedure without including the boundary points a and b as interpolation nodes. This then yields the open Newton-Cotes formulas.

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* or *Simpson's 1/3 rule* or *Kepler's barrel rule*:

$$\int_a^b f(x)\ \mathrm{d}x \approx \frac{h}{3}\left(f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right). \tag{7.20}$$

7.6.2 Extended formulas

Applying the Newton-Cotes formulas for higher degree maneuvers ourselves into the problem of Runge's phenomenon. Hence, instead of determining the $\alpha_k^{(n)}$ for large n, we actually use the simpler Trapezoidal or Simpson rule, but apply it on sub-intervals of $[a, b]$. This then yields the *extended Newton-Cotes formulas* or *composite rules*.

We start with dividing $[a, b]$ into N parts of equal length

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

Each part is now a sub-interval $[z_{j-1}, z_j]$ with

$$z_j = a + jH$$

and we 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* and therefore we use $2N + 1$ nodes for N intervals and have $h = H/2$ in eq. (7.20). We denote the nodes as $x_j = a + j\frac{H}{2}$ for $0 \le j \le 2N$ and obtain

$$\begin{aligned}\int_a^b f(x)\ \mathrm{d}x &\approx \frac{h}{3}\left(f(x_0) + f(x_{2N}) + 2\sum_{k=1}^{N-1} f(x_{2k}) + 4\sum_{k=1}^{N-1} f(x_{2k-1})\right) \\ &= \frac{H}{6}\left(f(x_0) + 4f(x_1) + 2f(x_2) + 4f(x_3) + 2f(x_4) + \cdots + f(x_{2N})\right).\end{aligned}$$

7.7 Richardson extrapolation

Assume that we compute a numerical integral 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*.

7.7.1 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 $x_1, \ldots, x_n$ form the prefix of a sequence (x_i) that converges against x.

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 $(x_i) = (h, {}^{h}/_{2}, {}^{h}/_{4}, \ldots)$ converges to zero. The Neville recursion in eq. (7.11) for $f(0) = T_{1,\ldots,n}$ now becomes a bit simpler:

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

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 $p(x) = a_0 + a_2x^2 + a_4x^4 + \ldots$ for extrapolation, where the odd terms $x, x^3, \ldots$ are removed. Then we can actually substitute $t = x^2$, which gives $p(t) = a_0 + a_2t + a_4t^4 \ldots$ and nodes $t_i = x_i^2 = {}^{2h}/_{4^{i-1}}$. This leads finally to an improved Neville recursion

$$T_i = f(t_i) \tag{7.21}$$

$$T_{i,\ldots,i+m} = \frac{4^m T_{i+1,\ldots,i+m} - T_{i,\ldots,i+m-1}}{4^m - 1}. \tag{7.22}$$

7.7.2 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[10] and hence T is an even function. This now allows us to apply eq. (7.22) to extrapolate $T(0)$. This method is known as the *Romberg integration*.

7.8 Summary

Polynomial interpolation is a major powerhorse in numerical analysis, not only to turn a tabulated function into a continuous one, but also as a building block for various mathematical

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

operations like numerical differentiation and integration of functions. We briefly reviewed power series and in particular the Taylor series.

Then we phrased the problem of polynomial interpolation as a linear system, whose investigation led gave us the conditions under which we have a unique existence of interpolation polynomials. We continued by theoretically investigating the interpolation error, which eventually took us to Rhunge's phenomenon and Chebyshev polynomials as an optimal choice of interpolation polynomials under certain assumptions. We discussed two algorithmic approaches to compute interpolation polynomials without the need of solving the linear system: Lagrange's formula and Neville's algorithm.

The issue of oscillating interpolation polynomials of high degree motivated the introduction of splines as piecewise polynomial functions. Here the natural cubic spline is a particularly attractive choice in the technical domain, as it minimizes strain energy while still being twice differentiable although it is only of degree three.

On this basis we discussed numerical derivates and integrals based on polynomial interpolation, which lead to multiple formulas and rules. Finally, by revisiting Neiville's algorithm we discussed Richardson extrapolation as a general method to extrapolate sequences, and by leveraging this technique to the extended trapezoidal rule we obtained the powerful method of Romberg integration as a simple consequence.

7.9 Exercises

For the following exercises note that `numpy.poly1d()` provides an interpolation of polynomials that allows one to add and multiply polynomials with each other or with scalars.

Exercise 7.1 (⋆)**.** Compute the Taylor series of exp and sin around $x_0 = 0$. Plot the Taylor series with a finite sum of N summands, with $N \in \{3, 5, 7\}$.

Exercise 7.2 (⋆)**.** Compute the Taylor series of $x \mapsto \ln(1 + x)$ around $x_0 = 0$. (This is equivalent to computing the Taylor series of ln at $x_0 = 1$.)

- Based on this, give a formula for $\ln 2$.
- Plot the Taylor series with a finite sum of N summands, with $N \in \{3, 5, 7\}$.

Exercise 7.3 (⋆)**.** Leibniz gave a formula of π by computing the Taylor series of arctan. Repeat his idea. Plot the Taylor series with a finite sum of N summands, with $N \in \{3, 5, 7\}$.

Exercise 7.4 (⋆)**.** In Python, given the input points $(-1, 2), (0, 1), (1, 2), (2, 0)$, compute the interpolation polynomial p through those points by solving the linear system. (You can simply invert the coefficient matrix with `numpy.linag.inv()`.) Give the coefficient matrix and compute its determinant using the Vandermonde determinant formula and compare with the result from `numpy.linalg.det()`. Plot the interpolation polynomials and the input points.

Exercise 7.5 (⋆)**.** Let us define $f = \sin$ and consider the interpolation points x_1, x_2, x_3, x_4 with $x_k = \frac{k-1}{3}\frac{\pi}{2}$. Do the following:

- Compute the interpolation polynomial p.
- Compute the node polynomial w.
- Plot $f - p$ and $w/4!$ and compare with eq. (7.4) and eq. (7.5).

Exercise 7.6 (⋆)**.** Interpolate $f(x) = |x|$ on the interval $[-1, 1]$ using $n = 9$ interpolation nodes. Do this once with equidistant nodes and once with Chebyshev nodes.

- Compute and plot the interpolation polynomials.
- Plot the interpolation error, respectively.
- Plot the node polynomials.

Bonus: What happens if we increase n to 20 or 25? Compare with the documentation of `sci.interpolate.lagrange()` here. `https://docs.scipy.org/doc/scipy/reference/generated/scipy.interpolate.lagrange.html`.

Exercise 7.7 (⋆)**.** Implement the Lagrange polynomials for the input points of exercise 7.4.

- Compute and plot each Lagrange polynomial L_k.
- Verify that $L_k(x_i) = \delta_{ki}$.
- Verify that $\sum_{k=1}^{n} L_k(x) = 1$ for all x.
- Use them to compute the interpolation polynomial and plot it.

Exercise 7.8 (⋆⋆)**.** Implement Neville's algorithm for the input points of exercise 7.4.

- Compute and plot the intermediate polynomials $T_{i,\dots,i+m}$ for all i and m. Make use of memoization.
- Use them to compute the interpolation polynomial and plot it.

Exercise 7.9 (⋆)**.** Implement a variant of Neville's algorithm that evaluates the interpolation polynomial at a given position x without computing the entire polynomial as such. Test the implementation against the input points of exercise 7.4.

Exercise 7.10 (⋆ ⋆ ⋆)**.** Implementing a Neville tableau class, which allows to extend the tableau sequentially with interpolation points. That is, when the class is instantiated, it represents the zero function. When the first interpolation point is added, it represents a constant function. After the next, a linear function, and so on. Do so without recomputing the Neville tableau from scratch. Test the implementation against the input points of exercise 7.4.

Exercise 7.11 (⋆)**.** Let us compare cubic splines with interpolation polynomials. We are given seven input points $(-3, 0), (-1.5, 0), (-1, 1), (0, 0), (1, 0), (1.5, 1), (3, 0)$.

- Compute the interpolation polynomial, e.g., by inverting the linear system.
- Compute the natural spline using `scipy.interpolate.CubicSpline`. (Read the documentation to get natural spline.)
- Plot both results and compare them. (Hint: Restrict the plot to $[-3.1, 3.1]$ on the x-axis.)

Exercise 7.12 (⋆⋆)**.** We are going to compare in Python the two-point and the central three-point formula based on the derivative of the sine function.

- Choose 13 interpolation nodes $x_1, \dots, x_{13}$ equidistantly on the interval $[0, 2\pi]$. (Hint: Use `numpy.linspace()`.)

  We then consider the tabulated function f given x_i and $f(x_i) = y_i = \sin(x_i)$ for $1 \le i \le 13$. Plot f and the points (x_i, y_i).

- Compute the numerical derivatives at $x_1, \ldots, x_{12}$ using the two-point formula using the points x_i and x_{i+1} for $1 \leq i < 13$.
- Compute the numerical derivatives at $x_2, \ldots, x_{12}$ using the central three-point formula using the points x_{i-1} to x_{i+1} for $1 < i < 13$.
- Plot the above results together with the exact derivatives using $f'(x) = \sin'(x)$. Interpret the result.

Exercise 7.13 (⋆⋆). Let us compare the accuracy of extended trapezoidal rule and the extended Simpson's rule in Python as follows:

1. Implement a procedure `extended_trapezoidal(f, a, b, N)` that computes $\int_a^b f(x)\ \mathrm{d}x$ with the extended trapezoidal rule using $N + 1$ nodes. Test it for $f(x) = x$ and $f(x) = x^2$ over $[0, 1]$.
2. Implement a procedure `extended_simpson(f, a, b, N)` that computes $\int_a^b f(x)\ \mathrm{d}x$ with the extended Simpson's rule using $2N + 1$ nodes. Test it for $f(x) = x^2$ and $f(x) = x^4$ over $[0, 1]$.
3. Let $f(x) = e^x - (e - 1)$. Compare the two implementations with 13 nodes. (Take care to set N accordingly for each implementation!)

Exercise 7.14 (⋆). Extend exercise 7.13 by implementing `richardson_extrapolation(f, h, S)`, where $f(h), \ldots, f(h/2^{S-1})$ is used to compute $f(0)$. (Can be done in four lines of code.) You can use `scipy.interpolate.lagrange` for the interpolation polynomial. Test your code by calling `richardson_extrapolation(np.sin, 1.0, 4)`.

Then use Richardson extrapolation to implement `rhomberg_integration(f, a, b, S)`, such that `extended_trapezoidal(f, a, b, N)` is called, with N being $1, 2, \ldots, 2^{S-1}$. (This can also be done in four lines of code.) Also test it for $f(x) = e^x - (e - 1)$ with $S = 5$.

Part III

Computational Geometry

Chapter 8

Geometric computations

8.1 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.

The field of *computational geometry* is about algorithms and data structures in a geometric context. Devising algorithms and analyzing their computational efficiency concerning time and space is the dominant part in this research field. From a practical point of view, however, algorithm engineering in computational geometry also needs to address computing with geometric objects – like points and lines – just like we did for numbers in chapter 5.

A great deal of topics in computational geometry is actually of strong combinatorial nature, and this field is aptly called *combinatorial geometry*. Take for instance the street network of Austria on which we would like to compute shortest routes. Prima vista this is evidently a geometric problem. But at second glance, we can forget about the precise coordinates of cities and junctions and observe that the problem is actually driven by a semi-geometrical network structure: by the "topology" of the street network (combinatorial structure) which tells which street sections interconnect the network nodes and the length of these street sections (geometric structure). This is why computational geometry is closely related to combinatorics, topology and graph theory.[4]

8.2 Geometric constructions and predicates

A *predicate* is a boolean property of objects. For instance, the predicate $f\colon \mathbb{Z} \to \{\text{false}, \text{true}\}\colon f(x) \bmod 2 = 0$ on the set of whole numbers tells whether a number is even. A geometric example would be a predicate on the set of pair of lines telling whether the pair would intersect. Another predicate given a circle and a point would be whether the point lies within the circle.

Unlike a predicate, a geometric *construction* produces geometric objects, like points or lines. So while a predicate asks whether two lines intersect, a geometric construction would compute

[1] Geographic Information Systems

[2] Computer-Aided Design, Computer-Aided Manufacturing

[3] Printed-circuit board

[4] There is also a smaller subfield of "geometry of numbers" within discrete and combinatorial geometry, which is part of number theory. There is a tradition in Vienna towards this field, see for instance the subspace theorem by Schmidt.

the actual intersection point of two lines, or the circumcenter of a triangle, or the line between two points.

Many geometric algorithms require no constructions but require predicates only. For instance, given a finite point set $\{p_1, \ldots, p_n\} \in \mathbb{R}^2$ then we compute the convex hull (see section 3.1.3 and chapter 9) using predicates only. Indeed the geometric predicate telling whether a point lies left to a ray is all we need.

8.2.1 Construction of orthogonal vectors

A simple example for a geometric 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:

$$\begin{pmatrix} x \\ y \end{pmatrix} \cdot \begin{pmatrix} -y \\ x \end{pmatrix} = xy - yx = 0. \tag{8.1}$$

The vectors are orthogonal as the inner product is zero. For a rotation by 90° we have two ways to switch sign; the one in eq. (8.1) rotates in counterclockwise direction[5] the other in clockwise direction. This is also evident from the rotation matrix by a counterclockwise angle φ:

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

So a rotation by 90° can be done very quickly *without* loss in numerical precision and computationally fast as well. This is important because basic geometric predicates can be implemented using right-angle rotations of vectors.

8.2.2 Orientation of three points

We will later learn about the Graham scan algorithm for convex hulls, see algorithm 15. If we look carefully then all we need for this algorithm 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. 8.1. Given three points $p, q, r \in \mathbb{R}^2$, we may ask whether the triangle pqr is oriented counterclockwise, or whether r is left to the ray $\overrightarrow{pq}$, or whether the polygonal chain pqr forms a left turn.

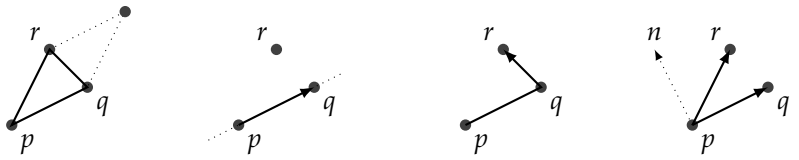


Figure 8.1: Four formulations of the same question: Is triangle pqr ccw oriented, is r left to $\overrightarrow{pq}$, is the polygonal chain pqr a left turn, and is the rotation of $\overrightarrow{pq}$ to $\overrightarrow{pr}$ less than a half turn? All are answered by $\Delta(p, q, r) > 0$.

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

[5] Well this statement actually only makes sense if we also tell what the coordinate system is, i.e., a typical orthogonal coordinate system with $(1, 0)$ pointing east and pointing north $(1, 0)$. Well, this again only makes sense for the typical northern hemi-sphere map reading of north (upwards) and east (rightwards).

instead consider the parallelogram formed by the vectors $\overrightarrow{pq}$ and $\overrightarrow{pr}$ in the left figure of fig. 8.1. There is a formula for the *signed* area of this parallelogram given by the following determinant:

$$\Delta(p,q,r) = \begin{vmatrix} p_x & q_x & r_x \\ p_y & q_y & r_y \\ 1 & 1 & 1 \end{vmatrix} = p_x q_y + q_x r_y + r_x p_y - p_x r_y - q_x p_y - r_x q_y. \tag{8.2}$$

By "signed area" we mean that $|\Delta(p,q,r)|$ is the area, but $\Delta(p,q,r)$ is positive if pqr is counterclockwise (ccw) oriented, negative if pqr is clockwise (cw) oriented and zero if p,q,r are collinear[6]. In other words, $\Delta(p,q,r)$ is twice the signed area of the triangle pqr. The matrix in eq. (8.2) is actually related to homogeneous coordinates, which are heavily used in computer graphics and robotics.[7]

The last figure in eq. (8.2) 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. (8.1) and hence

$$n \cdot (r-p) = \begin{pmatrix} -(q_y - p_y) \\ q_x - p_x \end{pmatrix} \cdot \begin{pmatrix} r_x - p_x \\ r_y - p_y \end{pmatrix} = (q_x - p_x) \cdot (r_y - p_y) - (q_y - p_y)(r_x - p_x) \tag{8.3}$$

On the other hand

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

and so we just learned that $\Delta(p,q,r) = n \cdot (r-p)$.[8] Note that eq. (8.3) only requires two multiplications, but at the expense of a numerical asymmetry as p_x and p_y are occurring twice.

Algorithm 12 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
```

We can summarize these insights in algorithm 12. 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.

8.2.3 Point location in circle

Given a circle and a point $p \in \mathbb{R}^2$, we can ask whether the point is within, on, or outside the circle. Depending on how the circle is given, the answers are more or less trivial.

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

[7] 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 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.

[8] And now we directly see that $\Delta(p,q,r)$ is the signed area of the parallelogram of the left figure in fig. 8.1.

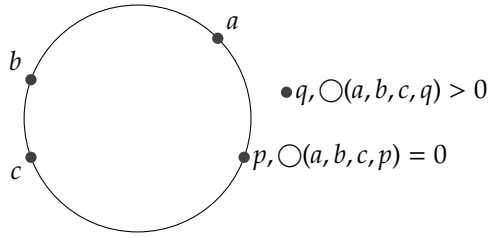


Figure 8.2: Illustration of the in-circle point location test for a circle spanned by three points.

In the simple case the circle is given by a center point $c \in \mathbb{R}^2$ and a radius $r \geq 0$. Then we simply check whether the distance of p to c is less, equal or greater to r, i.e., we compare $\|p - c\|$ against r. For numerical and efficiency reasons we of course avoid square roots when computing distances between points and instead compare squared distances as a general rule of thumb:

$$\left(p_x - c_x\right)^2 + \left(p_y - c_y\right)^2 \lessgtr r^2 \tag{8.4}$$

In the more involved case the circle is given by three points $a, b, c \in \mathbb{R}^2$. On other words, the circle is given as the circumcircle of the triangle abc. We assume that a, b, c are given in counterclockwise direction – we know already how to check – then we can test the sign of the following determinant

$$\bigcirc(a, b, c, p) = \begin{vmatrix} a_x & a_y & a_x^2 + a_y^2 & 1 \\ b_x & b_y & b_x^2 + b_y^2 & 1 \\ c_x & c_y & c_x^2 + c_y^2 & 1 \\ p_x & p_y & p_x^2 + p_y^2 & 1 \end{vmatrix}. \tag{8.5}$$

If $\bigcirc(a, b, c, p)$ equals zero then p is on the circle, if it is negative then p is within the circle and if it is positive then p is outside the circle. Note that no square roots are required here. A proof sketch behind this can be found in section A.3.

8.3 Predicates based on three-point-orientation

8.3.1 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. (Note that testing it vice versa is necessary!)

8.3.2 Point location in triangles and convex polygons

Assume we are given a convex polygon $P = (p_1, \ldots, p_n)$ with its vertices p_i given in ccw order, as illustrated in fig. 8.3. (We learn about convexity in chapter 3, but for now it suffices to know that a convex polygon has no dents.) We would like to test whether a given point r is in P or not. In particular, P could be a triangle.

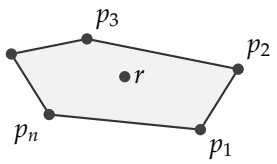


Figure 8.3: A point is in a convex polygon if it lies to the left of all edges in ccw direction.

The convexity of P makes this a very simple task as r is in P if and only if r lies to the left of all edges, i.e., to the left of all rays $\overrightarrow{p_1p_2}, \overrightarrow{p_2p_3}, \ldots, \overrightarrow{p_np_1}$. 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 13.

Algorithm 13 An algorithm that tests in linear time whether a point p is in a convex polygon P given in ccw direction.

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

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.

8.4 Summary

In this brief chapter we introduced first basics of computational geometry as a continuation of numerical mathematics for geometry. We distinguish between two major types geometric computations: Constructions and predicates. We gave three basic examples: The construction of orthogonal vectors, the predicate on the orientation of three points and the predicate on a point located in a circle. Based on the three-point-orientation predicate we further discussed the predicate of line intersection and point location in convex polygons.

8.5 Exercises

Exercise 8.1 ($\star$)**.** Let us consider the real 2×2 matrix $A(\varphi)$ that reflects a vector (x, y) in the plane by the line in direction $(\cos\varphi, \sin\varphi)$. Give the matrix $A(\varphi)$. For which φ is this geometric construction implemented without loss of numerical precision?

Exercise 8.2 ($\star$)**.** Let us compare the numerical quality of rotating vectors by rotation matrices.

1. In Python, write a procedure `rot_matrix(phi, dtype)` that takes an angle phi and a data type and returns the 2×2 rotation matrix for this angle.
2. For `dtype` being `np.float32` and `np.float64` run these experiments:
    - Output the matrix for 90°.

- Compute the matrix for 45°, raise the matrix to the power of 2, output and compare with the first result.
- Compute the matrix for 30°, raise the matrix to the power of 3, output and compare with the first result.
- Compute the matrix for 45°, raise the matrix to the powers of 10 and 18, output and compare with the first result.

Exercise 8.3 (⋆)**.** We are empirically investigating the numerics behind eq. (8.2) for the three-point-orientation predicate[9]:

- Implement a routine `orientation(p, q, r)` in Python with numpy that implements eq. (8.2).
- Set $p = (4, 4)$, $q = (6, 6)$ and $r^* = (1, 1)$. Then sample r from a little square centered at r^* and side length 100×10^{-16} through a regular 100×100 grid. Evaluate all 100^2 orientation tests, determine the sign – values are 1, 0 or -1 – and plot the result with `imshow()` in matplotlib. What do we expect and what do we get?
- Reorder the terms in the sum of eq. (8.2) and see what the effect is.

Exercise 8.4 (⋆)**.** We would like to test whether the line segments $\overline{AB}$ and $\overline{CD}$ intersect. To this end we check whether $\mathrm{ccw}(A, B, C)$ and $\mathrm{ccw}(A, B, D)$ give non-zero results of different signs. Argue why this procedure is correct or give a counter example.

Exercise 8.5 (⋆)**.** We are given four points $p_1, \ldots, p_4 \in \mathbb{R}^2$. Given an algorithm (in pseudo code) that tests whether (p_1, p_2, p_3, p_4) forms a convex polygon in ccw order.

Exercise 8.6 (⋆)**.** We are given four points $p_1, \ldots, p_4 \in \mathbb{R}^2$. Given an algorithm (in pseudo code) that tests whether p_1, p_2, p_3, p_4 lie in convex position, i.e., they all lie on the convex hull. (How is this exercise different to exercise 8.5?)

We assume that the four points lie in general position, i.e., no three points lie on a line, no two points have the same x- or y-coordinate.

[9] This example has been inspired, if not stolen, from some presentation of Stefan Schirra (Univ. Magdeburg). If you order a t-shirt with the pattern then cite Stefan Shirra.

Chapter 9

Convex hull

In a sense, what sorting is to algorithms is convex hull to computational geometry: They form the basis for many more complex algorithms and can be used to accelerate many computational geometry problems, like collision detection. Secondly, convex hulls are used for shape estimation from point clouds, like in robotics or computer vision. Actually, there is also a deeper connection between sorting and convex hulls, we will see later.

We introduced convexity and convex hulls from a mathematical perspective in section 3.1.3 in chapter 3. Here, we focus on algorithmic aspects of convex hulls in the plane and present two algorithms.

For a start, we recall fig. 3.5 where we illustrated the convex hull as the shape attained by a tight rubber band around nails forming the point set $p_1, \ldots, p_n$. Adding or removing a nail (point) in the interior of the convex hull does not change the rubber band (convex hull). This is a nice insight we can potentially use to speed-up any algorithm for convex hulls by reducing the number of input points: We take any three input points from fig. 3.5, form a triangle, and any input point in (the interior of) this triangle can be discarded, see fig. 9.1.[1]

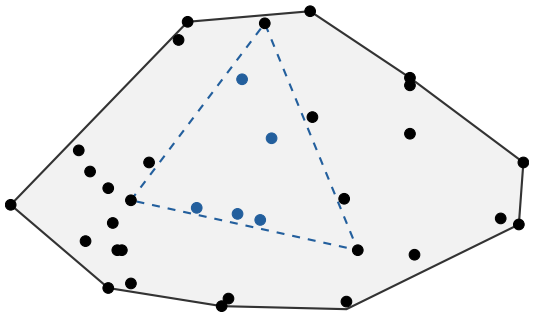


Figure 9.1: The blue triangle is formed by input points. Removing all input points (blue) within the triangle does not change the convex hull.

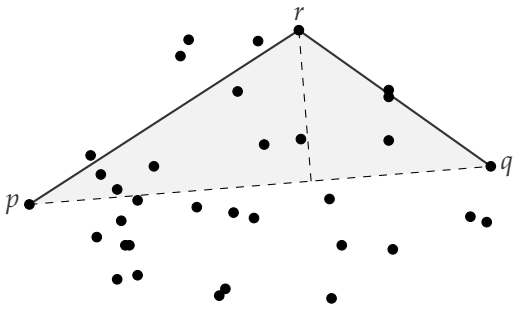
9.1 Quickhull

Discarding points in triangles leads us directly to the divide-and-conquer (see section 2.3.1) algorithm for convex hull, which is called *quickhull*. Its name is derived from the quicksort

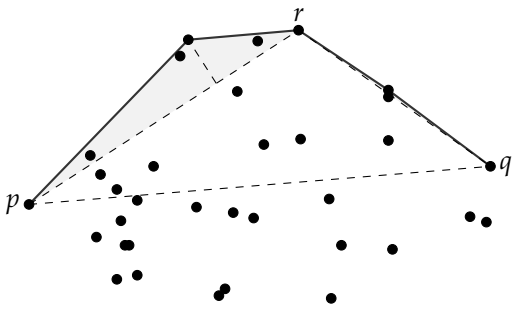
[1] A more practical variant is to remove all points of large axis-aligned rectangles instead of triangles, see [23].

algorithm [4]. The quickhull algorithm is implemented in the well known qhull [3] 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. 9.2a. 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, see fig. 9.2a.



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

Next it recursively repeats by constructing triangles and discarding points as in fig. 9.2b: 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 $\overline{rq}$: It constructs a triangle with the point that is left-most to $\overrightarrow{rq}$ and discards points, see fig. 9.2b. When no points are left it is done with the upper part. The entire algorithm is summarized in algorithm 14.

Algorithm 14 The quickhull algorithm to compute conv S.

```
procedure QUICKHULL(S)                                  ▷ Compute convex hull of point set S
    p ← arg min_{v∈S} v.x                               ▷ Left-most input point
    q ← arg max_{v∈S} v.x                               ▷ Right-most input point
    return PARTIALHULL(S, p, q) + PARTIALHULL(S, q, p)

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
    if |S| ≤ 1 then                                     ▷ Only at most one point left, we are done
        return [p] + S
    r ← arg max_{v∈S} d(v, pq)                          ▷ Maximum orthogonal distance
    return PARTIALHULL(S, p, r) + PARTIALHULL(S, r, q)
```

The essential idea can be generalized to convex hulls in $\mathbb{R}^d$. Instead of triangles we have simplices (see fig. 3.6) of higher dimension, see [4] for details.

Analysis. Let us consider algorithm 14. If the n input points are distributed in an unfavorable way then $O(n)$ many calls of PARTIALHULL cost $O(n)$ time each, which leads to a total of $O(n^2)$ time in the worst case. However, if the recursion is balanced well then the recursion depth is $O(\log n)$ and on each recursion level we spend in total $O(n)$ time, leading to total time complexity of $O(n \log n)$.

9.2 Graham scan

A simple algorithm that runs in $O(n \log n)$ time in the worst case is Graham's scan. It essentially orders the points by angles around an anchor point and then iteratively constructs the hull. Here, however, 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, which would be unfavorable in terms of numerics and speed.

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. 9.3 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.

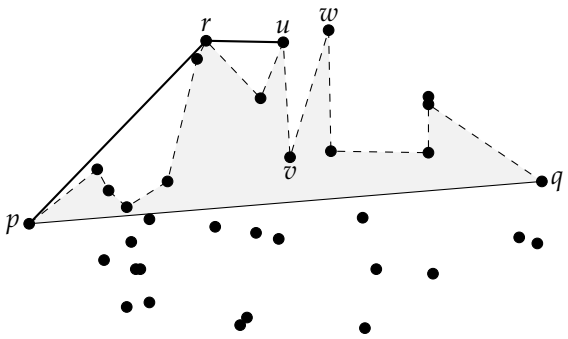


Figure 9.3: The Graham scan algorithm makes sure that while computing the upper part of the convex hull we only make right turns.

In PARTIALHULL in algorithm 15 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. 9.3 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 counting argument: Each vertex $v \in S$ is considered at most twice, because it is appended once and removed at most once. Hence, the POPBACK operation in the inner loop can only be executed $O(n)$ times.

Algorithm 15 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)

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
        H.APPEND(v)
    H.POPBACK( )                                 ▷ q is last element in H, remove it
    return H
```

9.3 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 would like to sort the numbers $x_1, \dots, x_n$. We start by creating 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. That is, we have an $O(n)$-reduction of sorting to convex hull. Since we know that sorting has a lower bound of $\Omega(n \log n)$, the same lower bound must hold for convex hull, see section 1.2.2 and section 1.2.3.

What we therefore also showed is that the Graham scan algorithm is worst-case *time optimal*, cf. section 1.2.2.

9.4 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[2], 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 do 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.

[2] One can vertically lift the input points in the plane on a paraboloid one dimension higher, compute the convex hull there, and project the edges back onto the plane, which gives the Delaunay triangulation. This works for points in $\mathbb{R}^d$ lifted to a paraboloid in $\mathbb{R}^{d+1}$. See also section 11.2.2.

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 [3] 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.]])
```

9.5 Summary

Convex hull algorithms form the basis for many other geometric algorithms and we mentioned a couple applications. Based on the mathematical foundation in chapter 3 we drew algorithmic conclusions which lead to quickhull as a first algorithm based on the divide-and-conquer paradigm and Graham scan as a second algorithm based on sorting and iterative construction. For the latter, we also reversed the perspective and showed that convex hull algorithms are at least as hard as sorting, leading to a lower bound on the time complexity of convex hull algorithms.

9.6 Exercises

Exercise 9.1 (⋆)**.** Consider n points $p_1, \ldots, p_n$ in the plane. Assume p_i is in the interior of the convex hull of these points, prove that removing p_i does not change the convex hull.

Exercise 9.2 (⋆)**.** Consider n points $p_1, \ldots, p_n$ in the plane. Assume no p_i is in the interior of their convex hull, prove that removing any p_i does change the convex hull.

Exercise 9.3 (⋆)**.** Construct a finite point set for quickhull that takes quadaratic runtime. That is, explain how to construct such a point set with n elements in general.

Exercise 9.4 (⋆)**.** We consider n points $p_1, \ldots, p_n$ in the plane and we consider to apply Graham scan to compute the convex hull.

Now we add another point p_{n+1} to our input. Recomputing the hull takes $O(n \log n)$ time. Can we do it in linear time if we are given the old convex hull of the first n points? Argue how or why not.

Chapter 10

Range searching

10.1 Introduction

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:

$C \leftarrow \emptyset$
for $q \in P$ **do**
  if $C = \emptyset \ \vee \ \min_{p \in C} d(p, q) \geq \varepsilon$ **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.

10.2 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.

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. 10.1 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.

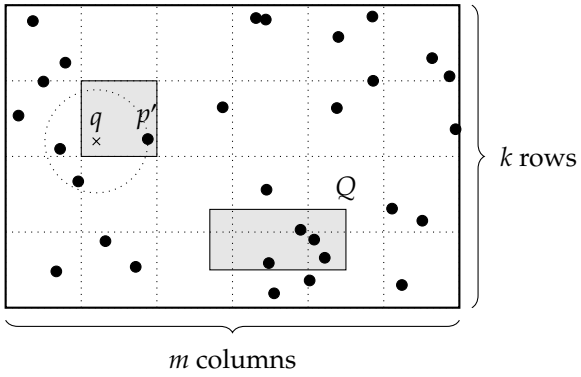


Figure 10.1: Geometric hashing for a point set in a rectangular region. It can be used for range queries for a query range Q or for nearest neighbor searches for a query point q. There is a candidate point p' in the cell of the query point q but the closest point is different to p'.

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. 10.1. 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. 10.1 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.

10.3 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.

10.3.1 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. 10.2. 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.

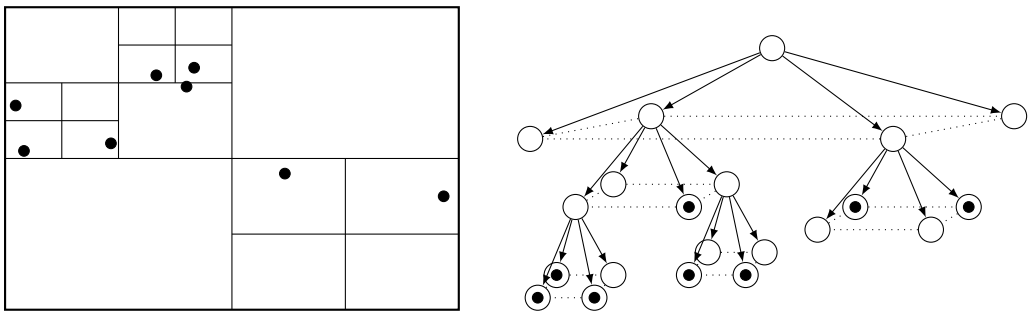


Figure 10.2: A quadtree with bucket capacity 1. Left: the cell decomposition of the quadtree. Right: The hierarchy of nodes, where each node covers a rectangular cell. The leaf nodes that contain a point are marked.

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. 10.2 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*.

10.3.2 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 level-wise between the coordinate axes. In fig. 10.3 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.

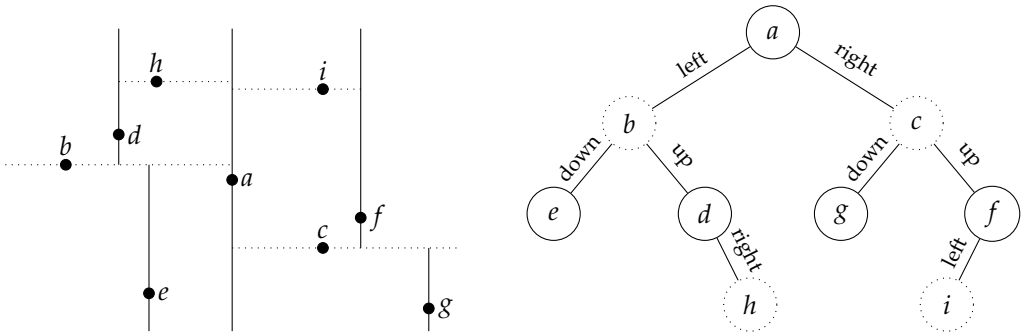


Figure 10.3: A k-d tree for a point set in $\mathbb{R}^2$. Left: The geometric tessellation scheme of the k-d tree. Right: The node hierarchy where every other level corresponds to a split along a different coordinate axis. Dashed lines on the left belong to dashed nodes on the right.

An orthogonal range search for a range Q is done by traversing the tree. In fig. 10.3 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.

10.4 Summary

In this chapter we have introduced range searching and nearest-neighbor searching as a fundamental tasks in computational geometry. We started with a naive solution for a point-cleanup task and then learned about two data structures that are able to speed up the task at hand. First, we presented geometric hashing as a simple and practical method and discussed its lmitations. Then we introduced hierarchical data structures, namely quadtrees and k-d trees, that are able to adapt to the distribution of the point set.

10.5 Exercises

Exercise 10.1 ($\star$). We construct a geometric hash over the square domain $R = [0, 10]^2 \subset \mathbb{R}^2$. Assume we have uniformly distributed 100 points on R and we erect a geometric hash with 100 cells. We distinguish two cases concerning the cell layout of the geometric hash:

- We use a 10×10 layout.
- We use a 1×100 layout.

Now we perform a range query with a random query range Q forming a square of side length 1 within R.

- How many cells need to be considered for one query on average?
- How many points are in those cells in total on average?
- How many points are in Q on average?

Exercise 10.2 ($\star$). We have a set of 10 000 straight-line segments and want to compute all crossing points. The line segments have been uniformly distributed on the unit square $[0, 1]$ and all of them have length 0.01.

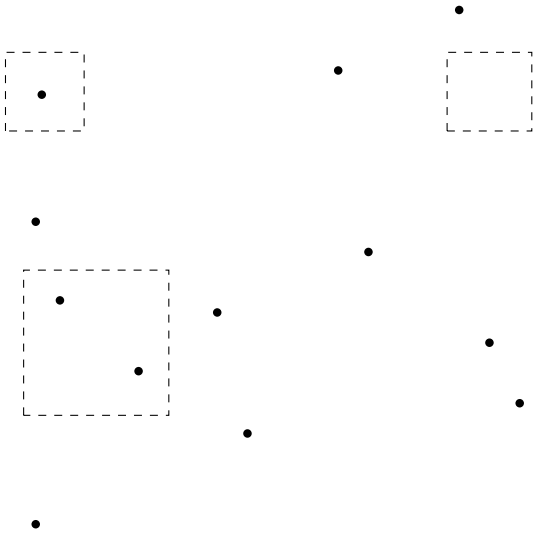
We apply two different methods:[1]

- Assume that we compute the crossing points brute force by pairwise checks. How many combinations of line segments do we test?
- We erect a 100×100 geometric hash. Similar to a point set, we register each line segment at each cell intersected by it. Describe how you can find all crossing points using the geometric hash.

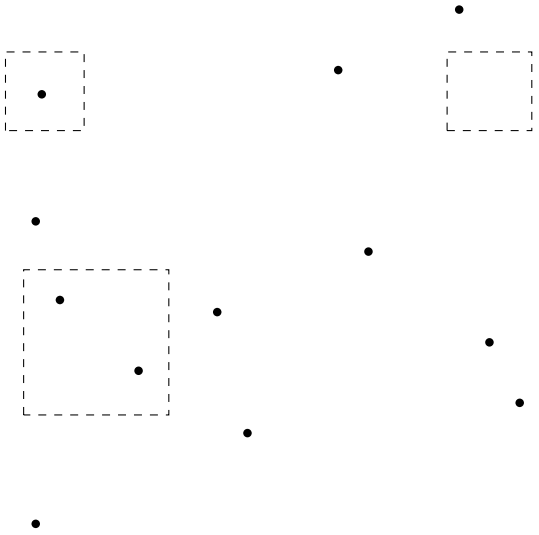
  How many pairwise checks between line segments are considered on average? (We are happy with a good upper bound, because the precise expected value of this number is more involved.)

[1] The algorithm by Bentley-Ottmann can do this in $O((n+k)\log n)$ time deterministically, where n is the number of line segments and k is the number of crossings. An algorithm by Chazelle and Edelsbrunner runs in optimal $O(n \log n + k)$ time.

Exercise 10.3 (⋆)**.** We are given the point set below. Sketch with a balanced k-d tree. (The axis-parallel lines suffice.)



Sketch a different solution:



Sketch how the trees are traversed for the three different query ranges (dashed boxes).

Exercise 10.4 ($\star$)**.** We have a quad tree over the unit square $[0,1]^2 \subset \mathbb{R}^2$ with 16 points and bucket size of 1. No two points are closer than $1.5 \cdot 10^{-3}$. What is the minimum and maximum depth of the tree?

Chapter 11

Voronoi diagram and Delaunay triangulation

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. 11.1, 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.

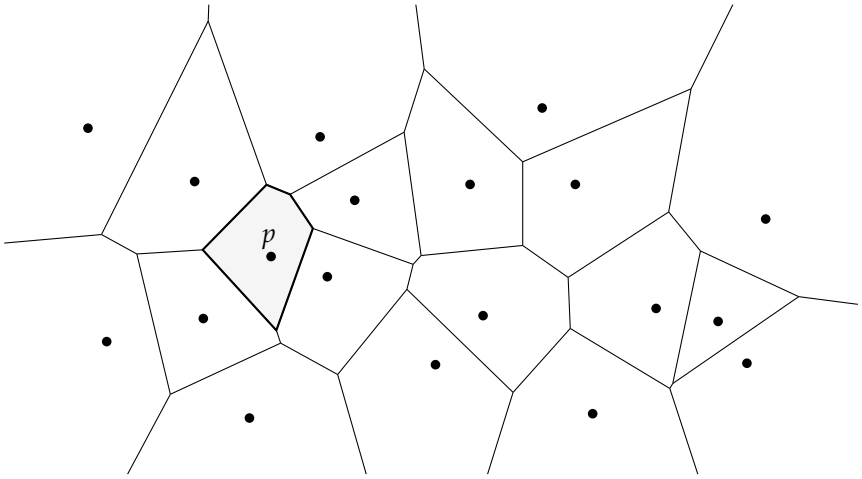


Figure 11.1: Voronoi diagram of points. The Voronoi region of p is shaded in gray and its boundary, the Voronoi polygon, is depicted thick.

11.1 Definition and properties

11.1.1 Voronoi diagram of points

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 $s_1, \ldots, s_n$ 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:

$$\mathrm{VR}(s) = \{p \in \mathbb{R}^2 \colon d(p,s) \leq d(p,s') \ \forall s' \in S\}. \tag{11.1}$$

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, which is slow and inaccurate. We define the *Voronoi polygon* $\mathrm{VP}(s)$ as the boundary of $\mathrm{VR}(s)$, which is denoted by $\mathrm{VP}(s) = \partial\,\mathrm{VR}(s)$.

Definition 18. The *Voronoi diagram* $\mathrm{V}(S)$ of a set $S = \{s_1, \ldots, s_n\}$ 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. (11.1) as

$$\mathrm{VR}(s) = \bigcap_{\substack{s' \in S \\ s' \neq s}} H(s,s') \tag{11.2}$$

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. 11.2. Each point x on the bisector is equidistant to s and s'. From eq. (11.2) 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.

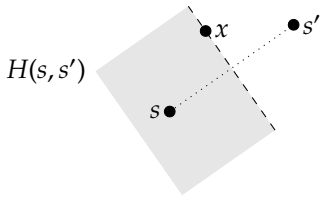
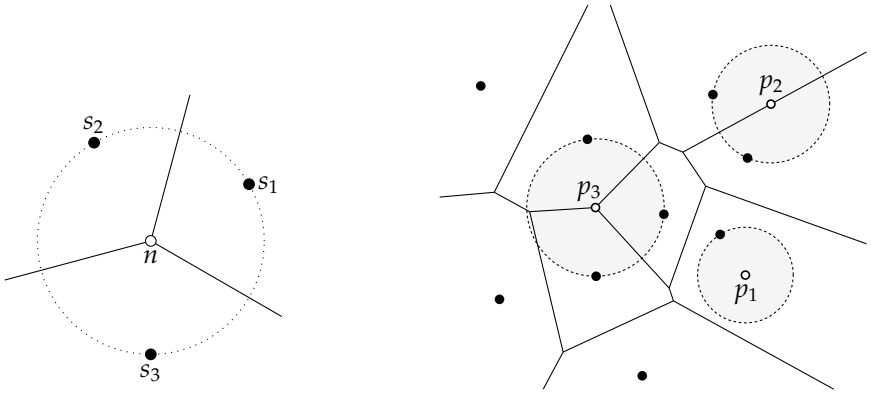


Figure 11.2: The half plane $H(s,s')$ with the bisector between s and s' as boundary.

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. 11.3a 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



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

Figure 11.3: 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.

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 VR(s), in second case p lies on a Voronoi edge and in the third case p lies on a Voronoi node, see fig. 11.3b.

11.1.2 Delaunay triangulation

We can also interpret V(S) as a planar graph, a PSLG in fact. Hence, we can consider the dual graph of V(S) as in fig. 11.4. What we obtain is the so-called *Delaunay triangulation* 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. Every Voronoi edge corresponds to a Delaunay edge. The infinitely long Voronoi edges are the Delaunay edges of the convex hull.

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. 11.5a.

In fact, if a Voronoi node is of degree higher than three – because more than tree sites $s_1, \ldots, s_k$ are cocircular – then we do not obtain a triangle but a (cocircular) k-gon and we have to triangulate this k-gon (in an arbitrary way). 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. 11.5a. 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.[2]

In fig. 11.5b 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

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

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

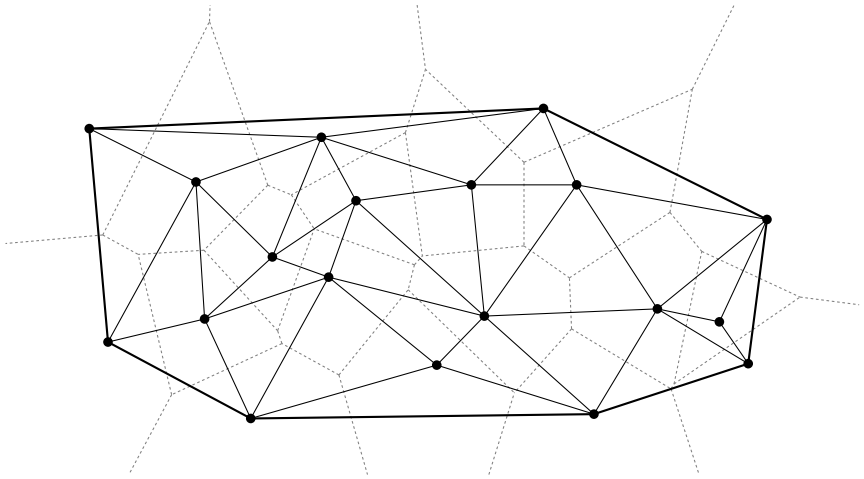
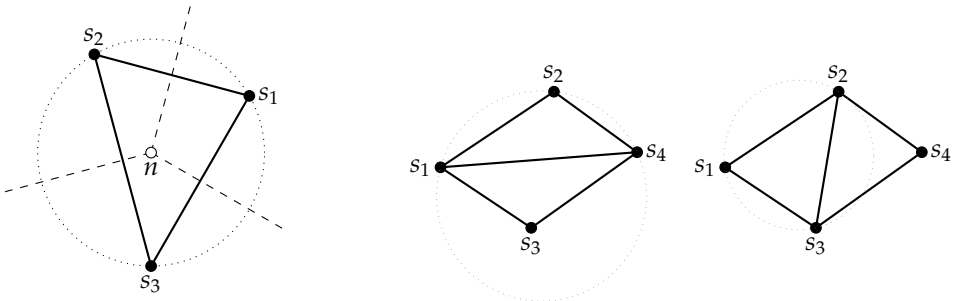


Figure 11.4: The Delaunay triangulation D(S) of a point set S is the dual graph of the Voronoi diagram V(S).



(a) A triangle for each node.

(b) Edge flip to obtain Delaunay triangles.

Figure 11.5: 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.

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. 11.5b 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 triangle particularly "aesthetic" and, for instance, interesting for finite element methods.

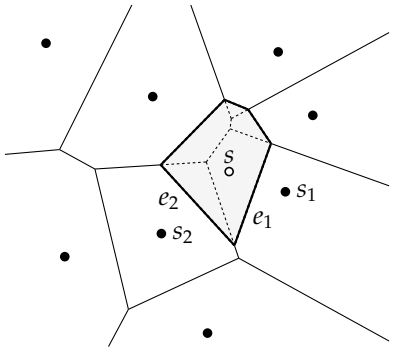


Figure 11.6: Inserting a site s creates a new Voronoi polygon VP(s) and removes the dashed line structure.

11.2 Computation

11.2.1 Incremental construction of Voronoi diagrams

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. 11.3a. 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 V(S) of a site set S, how to compute 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 V(S) in order to obtain V(S^+)? At the end 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. 11.6 we illustrate this situation.

Circular scan. Let $s_1 \in S$ be the[3] 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 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 the new polygon, we obtain the new Voronoi diagram 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,

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

which are based on floating-point arithmetic.[4] 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 [47] 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 V(S) is always a tree, i.e., connected and free of cycles. Roughly speaking, if the dashed structure in fig. 11.6 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 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 [47] 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.

11.2.2 Complexity and implementations

Interpreting the Delaunay triangulation and the Voronoi diagram as a planar graph gives us some combinatorial properties that important regarding the computational complexity. First, by corollary 2, we know that there are only a linear number of Delaunay triangles. By duality, there are only a linear number of Voronoi nodes. And by lemma 11 there are only linear number of Voronoi edges (or Delaunay edges).

This insight is important for the time complexity of algorithms and space complexity to store Voronoi diagrams and Delaunay triangulations. In particular, in can be shown that if we construct 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 V(S) we can compute D(S) in $O(n)$ time.

The qhull library – which is used by scipy and MATLAB – can actually compute V(S) and 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 vertically lifting them on a paraboloid then their convex hull forms a polyhedron, whose edges projected back onto the plane gives the D(S).

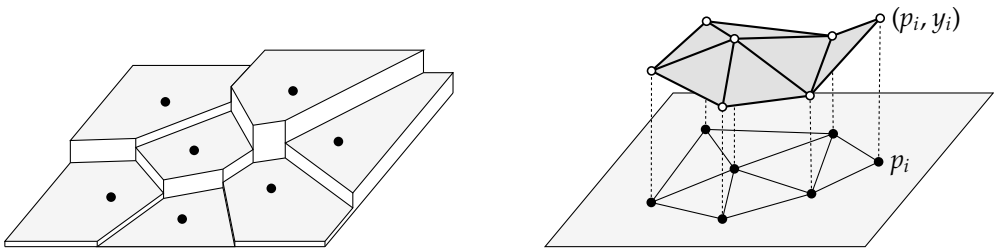
[4] There are alternatives to floating-point arithmetic, like *exact geometric computation (EGC)* based on libraries like Core or LEDA.

11.3 Applications

11.3.1 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 $p_1, \ldots, p_n \in D$ 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 7.4.1 we recall that a piecewise linear function is a simple solution to this problem. However, in contrast to fig. 7.3a 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.



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

Figure 11.7: Approximating a function $f\colon D \to \mathbb{R}$ over a domain $D \subset \mathbb{R}^2$ given at a point set $P = \{p_1, \ldots, p_n\}$ with function values $y_1, \ldots, y_n$. Left: A piecewise constant interpolation by computing V(P) and lifting each Voronoi region VR(p_i) by y_i. Right: A piecewise linear interpolation by computing D(P) and lifting each triangle vertex p_i by y_i.

Piecewise constant interpolation. For any $p \in D$ we assign some function value of the set $\{y_1, \ldots, y_n\}$. The most natural choice is probably to find the nearest neighbor p_i of p as the best representation among $\{p_1, \ldots, p_n\}$ 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 VR(p_i) and so we receive a function graph as in fig. 11.7a.

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 D(P) and lift each triangle vertex p_i by y_i. This gives as a continuous, triangulated surface as function graph, see fig. 11.7b.

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. 11.5b. Then such a point is better interpolated using the Delaunay triangulation at the right.

11.3.2 Euclidean MST and TSP

We could directly compute the EMST using, say, Kruskal's algorithm for the computation of the MST of arbitrary edge-weighted graphs applied to the complete Euclidean graph. Unfortunately, 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 $O(e \log e)$.

However, we can be shown that the EMST is part of the Delaunay triangulation, see fig. 11.8. 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 scheme (PTAS)* for ETSP.

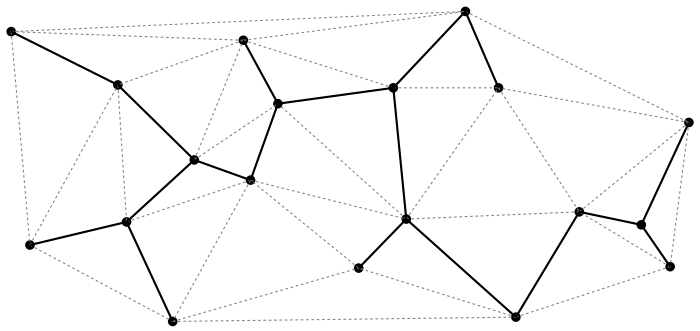


Figure 11.8: The Delaunay triangulation (dotted) contains the Euclidean minimum spanning tree (solid).

Chapter 12

Skeleton structures

12.1 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.

12.2 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. 12.1.

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.

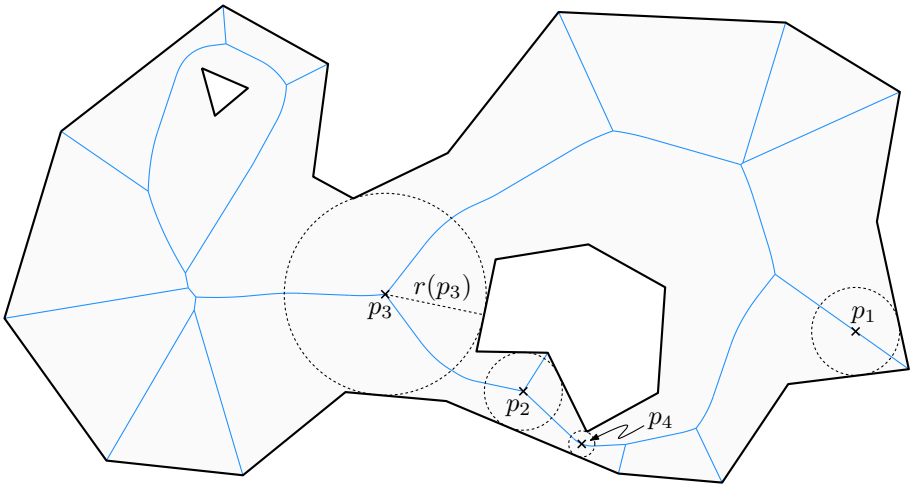


Figure 12.1: The medial axis $\mathrm{M}(P)$ in blue of a polygon with holes P shaded in gray. For some points p_i the disk $D(p_i, r(p_i))$ with clearance radius $r(p_i)$ is shown in dashed lines. The point p_4 constitutes a bottleneck.

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. 12.1. Then we can reconstruct P by

$$P = \bigcup_{p \in \mathrm{M}(P)} D(p, r(p)), \tag{12.1}$$

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. (12.1) is the inverse transformation of the medial axis transformation. We could exploit eq. (12.1) 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. 12.1.

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.

12.3 Generalized Voronoi diagrams

12.3.1 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 M(P) is actually part of a more general structure, namely the generalized Voronoi diagram V(P) of a polygon with holes P.

In section 11.1.1 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 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.

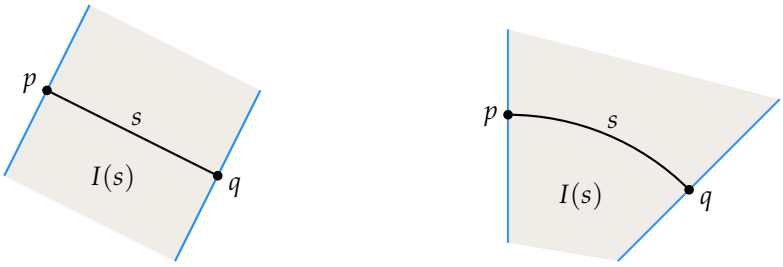
12.3.2 Straight-line segments and circular arcs

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. 12.2 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* I(s) of a site s. The idea is to restrict the Voronoi region VR(s) of a site s to a certain region, its cone of influence I(s). The cone of influence is illustrated in fig. 12.2 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 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 12.2: The Voronoi diagram of a straight-line segment s and its endpoints p, q and likewise for the circular arc. The Voronoi region VR(s) of s is restricted to its cone of influence I(s).

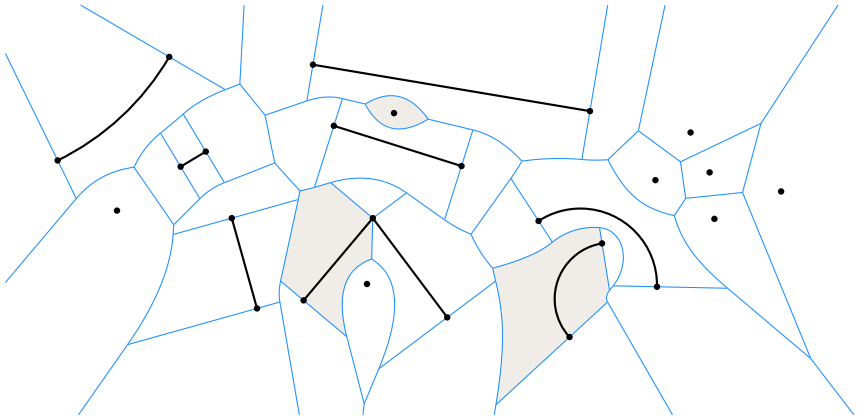


Figure 12.3: The generalized Voronoi diagram of points, straight-line segments and circular arcs. A few Voronoi regions are shaded in gray.

So now we can more or less copy over the definition of Voronoi diagram of points, see section 11.1.1, to the more general setting:

$$\mathrm{V}(S) = \bigcup_{s \in S} \mathrm{VP}(s) \tag{12.2}$$

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

$$\mathrm{VR}(s) = \{p \in \mathrm{I}(s) \colon d(p,s) \leq d(p,s')\ \forall s' \in S\}. \tag{12.3}$$

Figure 12.3 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. 12.4 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. (12.3) which we ignored so far: For circular arcs s the definition in eq. (12.3) 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. 12.5 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

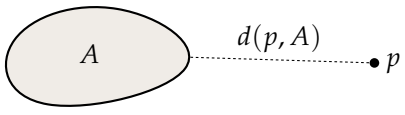


Figure 12.4: The infimum distance $d(p,A)$ between a point p and a point set A.

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

$$\mathrm{VR}(s) = \mathrm{cl}\{p \in \mathrm{int}\,\mathrm{I}(s) \colon d(p,s) \leq d(p,s')\ \forall s' \in S\}, \tag{12.4}$$

which means that we first take the topological interior and then form the topological closure.[2] In simple words, the boundary of 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 [29, 25].

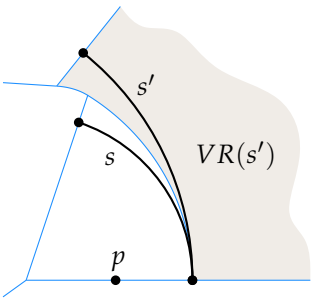


Figure 12.5: The point $p \in \mathrm{VR}(s)$. But since $d(p,s) = d(p,s')$ the point p would also be part of VR(s') if we would use eq. (12.3) instead of eq. (12.4).

12.3.3 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 V(P) of P simply as V(S). Depending on the application and the context we may restrict the Voronoi diagram to P and ignore everything outside P.

In fig. 12.6 we show the Voronoi diagram V(P) of the same polygon with holes P as in fig. 12.1. 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 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 $O(c,r) = \{p \in \mathbb{R}^2 \colon \|p - c\| < r\}$ 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 M(P) and V(P) have meaning can be extended to a significantly more general class.

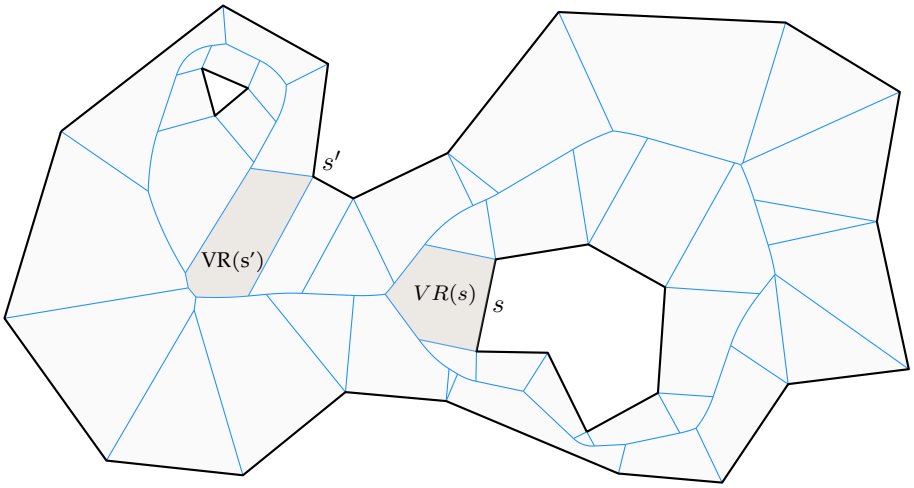


Figure 12.6: The Voronoi diagram V(P) of a polygon with holes P gives a nearest-neighbor cell decomposition of P. A few cells (Voronoi regions) are shaded in gray. The Voronoi diagram contains the medial axis M(P), cf. fig. 12.1.

12.3.4 Computing generalized Voronoi diagrams

Geometry. If the site set S consists of points only then the Voronoi polygons 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 [29] 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 [25].

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 [29].

12.4 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. 12.7: The points where the fires meet are exactly given by the Voronoi diagram V(S).

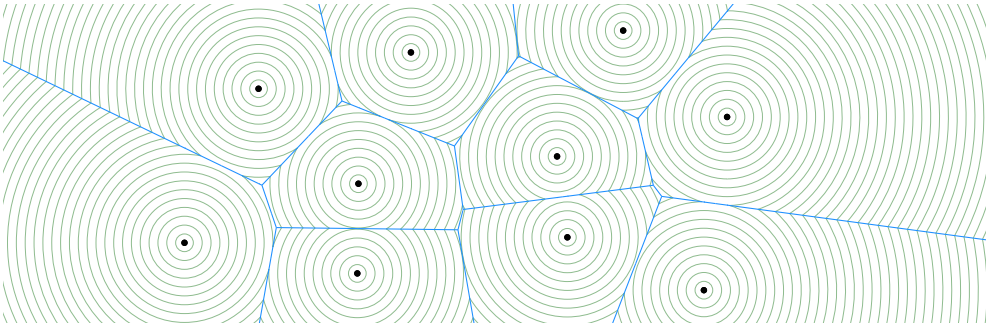


Figure 12.7: Grassfire model for a point set S. It sends out unit-speed offset waves from the sites in S and they interfere on V(S).

Hence, we could actually define 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. 12.8 we illustrate the grassfire wavefronts emanated by the points and straight-line segments of a polygon with holes P. The interference patterns give as 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 V(P) has been computed, an offset curve is computed in an easy, fast and numerically stable way by traversing the Voronoi diagram.

12.5 Straight skeletons

We call V(P) and 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. 12.8. 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 V(P). It is called the *straight skeleton* S(P). Unlike the generalized Voronoi diagram, the straight skeleton consists of straight-line segments only, hence its name, see fig. 12.9.

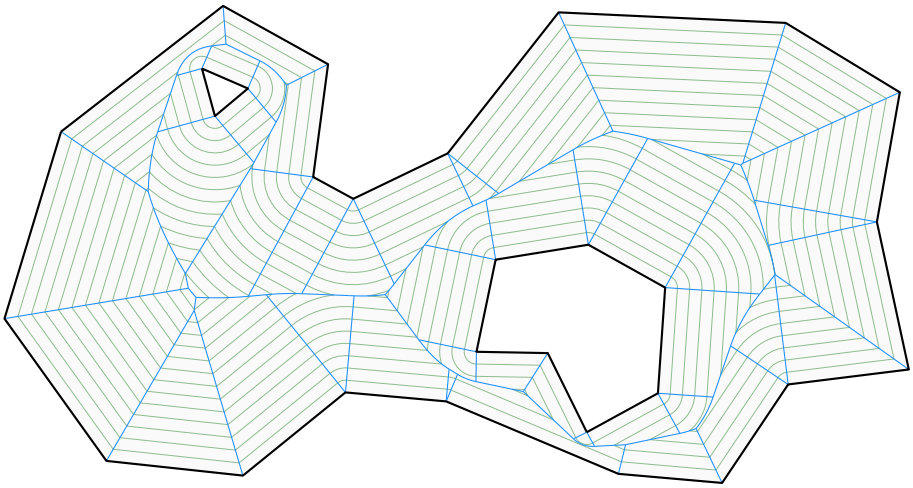


Figure 12.8: The offset curves given by grassfire wavefront curves. Its interference patterns create the Voronoi diagram.

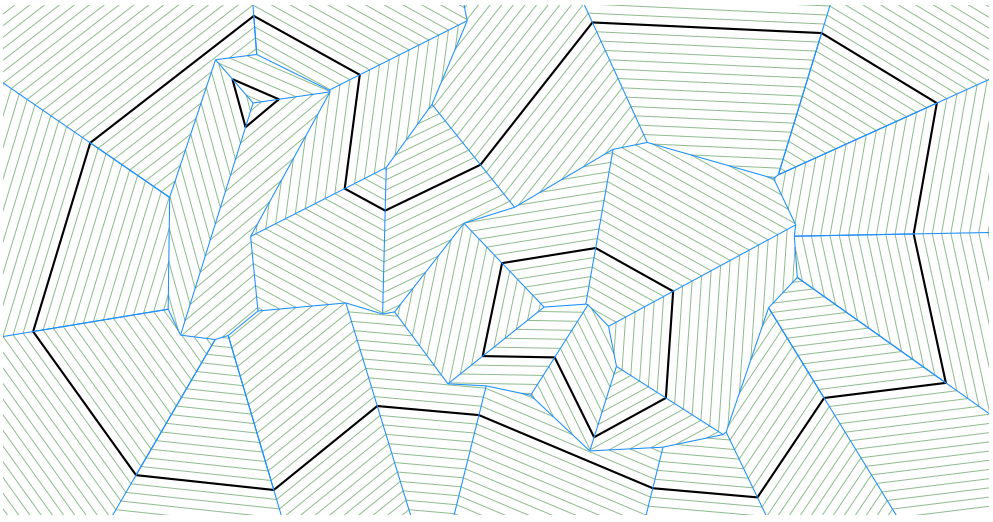


Figure 12.9: Mitered offset curves and the straight skeleton S(P) of a polygon with holes P.

Part IV

Appendices

Appendix A

Selected details

A.1 Q-learning

A.1.1 Temporal difference learning

The value iteration algorithm in algorithm 16 is a model-based approach as it is based on the transition probabilities of the underlying Markov decision process (MDP). One way to get rid of those is to also learn a transition model from experience. Assume we apply the policy π and make a state transition $s \to s'$ through the action a obtained from $\pi(s)$, receiving the reward r. Then we *learned* that

$$V_\pi(s) \leftarrow \underbrace{r + \gamma V_\pi(s')}_{\text{new belief}}. \tag{A.1}$$

If V would be the optimal value function V^* then eq. (A.1) would be an equation. But we can use eq. (A.1) as an update rule similar to value iteration.

If we would update $V_\pi(s)$ through this rule then we incorporated our *experience* from receiving r and we propagated our *belief* in the value $V(s')$ of the successor state s' back to the current state s. Doing this would be a bad idea:

- All we may have previously learned about $V_\pi(s)$ is lost, while we may have not learned anything yet for $V_\pi(s')$.
- Different actions a taken in state s may lead to different successor states s' and rewards r, and we would also *accumulate* this experience.

Instead, we keep a fraction of our previous belief in $V_\pi(s)$ and update it with a fraction of the new experience, i.e., we build a convex combination with factor $\alpha \in [0, 1]$ of old belief and new experience:

$$V_\pi(s) \leftarrow (1 - \alpha)V_\pi(s) + \alpha\left(r + \gamma V_\pi(s')\right) \tag{A.2}$$

By setting α we control how strong the new experience influences our update. If we set $\alpha = 1$ then we obtain eq. (A.1) and we forget what we have learned before. If we set $\alpha = 0$ then we do not learn the new experience.

We can rephrase the rule in eq. (A.2) as follows, which better motivates its name *temporal*

difference (TD) learning:

$$V_\pi(s) \leftarrow V_\pi(s) + \alpha \cdot \underbrace{\left(\underbrace{r + \gamma V_\pi(s')}_{\text{new belief}} - \underbrace{V_\pi(s)}_{\text{old belief}} \right)}_{\text{temporal difference}} \tag{A.3}$$

Note that eq. (A.3) is a *model-free* rule. If we explore the (unknown) underlying MDP sufficiently, such that we visit all states and actions infinitely often, then we would hope that eq. (A.3) converges to the optimal value function V^*.

A.1.2 Q-function and Q-learning

A popular example for a model-free value-iteration algorithm is the *Q-learning* algorithm. In order to solve eq. (H.9) it introduces the *action-value function* Q_π, which is defined similarly to V_π, as follows:

$$Q_\pi(s, a) = E_\pi(G_t \mid s_t = s, a_t = a). \tag{A.4}$$

So Q_π is the expected return when starting in state s, taking action a, and then following the policy π. In analogy to the optimal value function V^* we define:

$$Q^*(s, a) = \max_\pi Q_\pi(s, a). \tag{A.5}$$

This is the expected return when starting in state s, taking action a, and then following the best policy. So in other words

$$\begin{aligned} Q^*(s, a) &= Q_{\pi^*}(s, a) \\ &= \sum_{s' \in \mathcal{S}} p(s', r \mid s, a) \Big(r + \gamma V^*(s') \Big), \end{aligned} \tag{A.6}$$

where the second equation follows from eq. (H.7). This allows us to rephrase V^* and π^* in terms of Q^*:

$$V^*(s) = \max_{a \in \mathcal{A}} Q^*(s, a) \qquad \text{by eq. (H.9)} \tag{A.7}$$

$$\pi^*(s) = \arg\max_{a \in \mathcal{A}} Q^*(s, a). \tag{A.8}$$

In particular, we can plug eq. (A.7) into eq. (A.6) and thereby rephrase the Bellman equation from eq. (H.9) as follows:

$$Q^*(s, a) = \max_{a' \in \mathcal{A}} \sum_{s' \in \mathcal{S}} p(s', r \mid s, a) \Big(r + \gamma Q^*(s', a') \Big).$$

The Q-learning algorithm now applies TD learning from eq. (A.3) to this Bellman equation, which leads to this update rule:

$$Q(s, a) \leftarrow Q(s, a) + \alpha \cdot \Big(r + \gamma \max_{a'} Q(s', a') - Q(s, a) \Big). \tag{A.9}$$

If $\mathcal{S}$ and $\mathcal{A}$ are finite this can be implemented as a table for Q. If $\mathcal{S}$ and $\mathcal{A}$ are continuous then we need to approximate Q by a function, e.g., a neural network. The latter leads to *Deep Q-learning*.

A.2 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 7.4.2.

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 $1 \le i \le n-2$

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

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 $1 \le i \le n-3$

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

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 [42] for details. Solving the linear system gives 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.

A.3 Proof sketch for in-circle point location

In section 8.2.3 we learned about a determinate in eq. (8.5) that allows for location tests of points p for circles given by three points a, b, c. What follows is a proof sketch.

Let us the consider the case where $\bigcirc(a, b, c, p) = 0$. The determinant is zero when the fourth row of the determinant is a linear combination of the other three rows, i.e.,

$$\begin{pmatrix} p_x \\ p_y \\ p_x^2 + p_y^2 \\ 1 \end{pmatrix} = \alpha \begin{pmatrix} a_x \\ a_y \\ a_x^2 + a_y^2 \\ 1 \end{pmatrix} + \beta \begin{pmatrix} b_x \\ b_y \\ b_x^2 + b_y^2 \\ 1 \end{pmatrix} + \gamma \begin{pmatrix} c_x \\ c_y \\ c_x^2 + c_y^2 \\ 1 \end{pmatrix}.$$

for some coefficients α, β, γ. We can reinterpret this equation as three equations

$$\begin{aligned} 1 &= \alpha + \beta + \gamma \\ p &= \alpha a + \beta b + \gamma c \\ \|p\|^2 &= \alpha \|a\|^2 + \beta \|b\|^2 + \gamma \|c\|^2 \end{aligned}$$

The first two lines say that α, β, γ are coefficients of an affine combination, namely p is an affine combination of the three points a, b, c. Let us denote by $m \in \mathbb{R}^2$ the center and by r the radius of the circumcircle of a, b, c. Then we can rephrase the third equation as follows

$$\begin{aligned} \|p\|^2 &= \alpha \|a\|^2 + \beta \|b\|^2 + \gamma \|c\|^2 \\ \|p\|^2 + \|m\|^2 &= \alpha(\|a\|^2 + \|m\|^2) + \beta(\|b\|^2 + \|m\|^2) + \gamma(\|c\|^2 + \|m\|^2) \\ \|p\|^2 + \|m\|^2 - 2p \cdot m &= \alpha(\|a\|^2 + \|m\|^2 - 2a \cdot m) + \beta(\|b\|^2 + \|m\|^2 - 2b \cdot m) + \gamma(\|c\|^2 + \|m\|^2 - 2c \cdot m) \\ (\|p - m\|)^2 &= \alpha(\|a - m\|)^2 + \beta(\|b - m\|)^2 + \gamma(\|c - m\|)^2 \\ (\|p - m\|)^2 &= r^2. \end{aligned}$$

Note that we used $r = \|a-m\| = \|b-m\| = \|c-m\|$ and $p \cdot m = (\alpha a + \beta b + \gamma c) \cdot m$. This fourth equation is therefore equivalent to saying p is on the circle and $\bigcirc(a, b, c, p)$ is zero on the circumcircle of a, b, c. Within the circle and outside the circle it is either entirely positive or entirely negative as

$\bigcirc(a, b, c, p)$ is continuous in p. However, note that $\bigcirc(a, b, c, p)$ tends towards ∞ when p goes to infinity because the co-factor

$$\begin{vmatrix} a_x & a_y & 1 \\ b_x & b_y & 1 \\ c_x & c_y & 1 \end{vmatrix}$$

of $\bigcirc(a, b, c, p)$ is positive for counterclockwise a, b, c. Hence, $\bigcirc(a, b, c, p)$ is positive everywhere outside the circle (and negative inside).

Part V

Trash bin, scratchpads and unfinished

Appendix B

Trash bin

B.1 Newton iteration for roots

B.2 Vibration reduction for servo drives

Motivation. Let us consider the following problem from motion control in industrial machine automation: We want to move a servo drive from one position to another. During the positioning, we accelerate and decelerate masses which causes the excitation of certain resonance frequencies. For instance, we may transport liquid in an open bin or move a tool of a CNC mill. Vibrations may therefore spill the liquid or make the CNC tool to leave its desired path.

One method for vibration reduction is *spectral shaping* of the positioning signals: If we manage to shape a positioning signal such that its Fourier transform – the spectrum – is low at the known resonance frequencies then we can reduce these pathological vibrations.

Problem setting. We first need to define what we mean by a position signal. For this application we concentrate on the velocity signal $v(t$ for the servo drive. Since acceleration is finite we require that v is a continuous. Without loss of generality, we assume that we perform the positioning in the time interval $[-1, 1]$ and we move from position 0 to 1. So we set $v(t) = 0$ for $t \notin [-1, 1]$ and we require

$$\int_{-1}^{1} v(t) \ \mathrm{d}t = 1.$$

The trick is now that we come up with a suitable set of base functions $g_1, \ldots, g_n$ such that $v = \sum_k \alpha_k g_k$ and we optimize the α_k to obtain a well behaving spectrum. Natural candidates are the trigonometric functions. If we further assume that acceleration and deceleration is symmetric then v is an even function and therefore $v(t) = v(-t)$. Hence, we can restrict the base functions to the cosines:

$$g_k(t) = \begin{cases} \frac{1}{2} \cdot (1 - \cos(k\pi \cdot (t+1))) & \text{for } t \in [-1, 1] \\ 0 & \text{otherwise} \end{cases}.$$

Note that the g_k are continuous, $g_k(t) = g_k(-t)$ and $\int_{-1}^{1} g_k(t) \ \mathrm{d}t = 1$. Hence the same holds for any linear combination

$$v(t) = \sum_{k=1}^{n} \alpha_k g_k(t)$$

of these g_k if

$$\sum_{k=1}^{n} \alpha_k = 1. \tag{B.1}$$

System of equations and equilibration. Let us denote by $G_k = \mathcal{F}(g_k)$ the Fourier transform of g_k and let us denote by $\omega_1, \ldots, \omega_m$ the pathological frequencies. Note that the Fourier transform is linear, so

$$V(\omega) = \sum_{k=1}^{n} \alpha_k G_k(\omega)$$

is the Fourier transform $\mathcal{F}(v)$ of v. So what we aim for is $V(\omega_j) = 0$ for all w_j. Now I do some magic stuff[1] because $V(\omega) \in \mathbb{C}$, so I use $|G_k|$ and write done in matrix notation:

$$\begin{pmatrix} |G_1(\omega_1)| & \ldots & |G_n(\omega_1)| \\ & \vdots & \\ |G_n(\omega_m)| & \ldots & |G_n(\omega_m)| \end{pmatrix} \cdot \begin{pmatrix} \alpha_1 \\ \vdots \\ \alpha_n \end{pmatrix} = 0$$

From eq. (B.1) we obtain the equation

$$\begin{pmatrix} 1 & \ldots & 1 \end{pmatrix} \cdot \begin{pmatrix} \alpha_1 \\ \vdots \\ \alpha_n \end{pmatrix} = (1)$$

Furthermore, we favor low maximum velocity. The optimum velocity signal regarding this aspect would be $v(t) = 0.5$ for $t \in [-1, 1]$, which would be discontinuous but integrate to 1. We could choose uniform sample points $t_1, \ldots, t_s \in [-1, 1]$, without the boundaries, and ask for

$$\begin{pmatrix} g_1(t_1) & \ldots & g_n(t_1) \\ & \vdots & \\ g_1(t_s) & \ldots & g_n(t_s) \end{pmatrix} \cdot \begin{pmatrix} \alpha_1 \\ \vdots \\ \alpha_n \end{pmatrix} = \begin{pmatrix} 0.5 \\ \vdots \\ 0.5 \end{pmatrix}$$

We now have three groups of linear equations that together form a overdetermined system. If it would be a regular system of linear equations then we can multiply an equation with a factor unequal to zero without changing the solution. In an overdetermined system, however, we seek for a solution that minimizes the least square error and multiplying an equation with a factor changes its weight in the error term.

Hence, we may *equilibrate* the equations to make them weigh the same relative to each other. In our case we want that the three groups have all the same weight. In addition we introduce a weight λ for the significance of the frequency group and a weight μ for the velocity group. This

[1] Note that $|V| \leq \sum_k |\alpha_k||G_k|$, but now I drop the $|.|$ from α_k. This is fine if the $\alpha_k \geq 0$, but who can say.

gives in total the following system:

$$\begin{pmatrix} 1 & \dots & 1 \\ \frac{\lambda}{m}|G_1(\omega_1)| & \dots & \frac{\lambda}{m}|G_n(\omega_1)| \\ & \vdots & \\ \frac{\lambda}{m}|G_n(\omega_m)| & \dots & \frac{\lambda}{m}|G_n(\omega_m)| \\ \frac{\mu}{s}g_1(t_1) & \dots & \frac{\mu}{s}g_n(t_1) \\ & \vdots & \\ \frac{\mu}{s}g_1(t_s) & \dots & \frac{\mu}{s}g_n(t_s) \end{pmatrix} \cdot \begin{pmatrix} \alpha_1 \\ \vdots \\ \alpha_n \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \\ 0.5\frac{\mu}{s} \\ \vdots \\ 0.5\frac{\mu}{s} \end{pmatrix}$$

The rank of the coefficient matrix is maximal. Using the pseudoinverse we can determine the $\alpha_1, \dots, \alpha_n$ for given frequencies $\omega_1, \dots, \omega_m$, time samples $t_1, \dots, t_s$ and weights λ and μ.

Appendix C

TODO

- Add index entries everywhere.

Appendix D

Introduction (old)

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 and the like. On the other hand, physics or chemistry are natural sciences. They investigate phenomena that we observe in our reality.[1]

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[2] 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 in an exact manner – without numerical loss – by a (digital) computer, but dealing with real numbers is fundamentally impossible for a computer.[3] We are left with approximations of real numbers and each operation introduces loss of precision. The field of numerical analysis investigates the challenges that arise from this fact.[4]

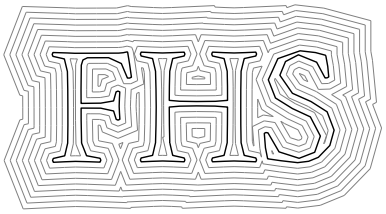
[1] The natural sciences use mathematics for modeling and as a language on one hand and as a tool to draw conclusions and extrapolations on the other hand. The classical engineering sciences sit on top of the natural sciences and leverage insights on our real world for technical mechanisms and constructions. Computer science is in some sense different to the classical engineering sciences as it builds upon mathematics directly. Brooks reflects on the nature of programming in his seminal work in the section called "The Joys of the Craft", when he refers to programming as "slightly removed from pure thought-stuff" and "building castles in the air, from air, creating by exertion of the imagination" [39, p. 7], which highlights the mathematical nature of programming.

[2] And we also find complex numbers, for instance. But from this point of view, complex numbers are essentially no different to real numbers, just pairs of which.

[3] 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. In particular, there are more real numbers than computer programs or Turing machines. So in this sense we really leave the natural habitat of computer science.

[4] Another yet very different example where digital computers face a challenge, but nature does not, is randomness. As far as we know true randomness – the absent of full determinism, at least the lack of full predictability – occurs in natural processes, like thermal motion of molecules or radioactive decay. But it does not occur in a digital computer in the sense of a deterministic Turing machine. To have true randomness in computers, we need additional hardware that uses natural randomness. This observation is essential for cyber security, when an element of unpredictability is required, e.g., in key generation.

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.[5] 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.



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?[6] How do we compute them in a computationally 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 three parts: Numerical programming, numerical mathematics and computational geometry. The first part deals with number representation and numerical analysis. It forms a foundation for numerical computations. The second part is about numerical algorithms within mathematics, in particular linear algebra and calculus.

This first two parts are often summarized by the field called *numerical mathematics*, but since we put some emphasis also on technical details and programming, we instead call the first part *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.

  [37] 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.)

  [42] 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 [21] that discusses many details of floating-point units.

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

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

[46] 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.

  [6] 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.

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

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

  [19] 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.

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

Acknowledgements. Simon Schindler, Martin Mayr

New course structures

Title of combined lecture notes:

> Algorithms in industry. Selected topics in combinatorics, optimization and numerics.

ILV Selected Algorithms and Optimization:

> Theoretical and experimental methods and criteria for evaluating the efficiency of algorithms and data structures. Analysis of selected algorithms and data structures with reference to different application domains. Optimisation possibilities of algorithmic implementations. Selected techniques and informatic implementations of mathematical optimisation methods with reference to different application domains.

- Basics
  - Algorithm analysis:
    O notation, time and space complexity, models of computation
  - Graphs I (combinatorial):
    directed and undirected graphs; paths, cycles and trees; weighted graphs
  - Performance optimization:
    Techniques to make code faster, measuring wall clock time; profiling techniques
- Mathematical optimization
  - Systems of linear equations
    Used to be in numerical mathematics. Optimization methods directly giving optimal solution (QR decomposition). Gaussian elimination with pivoting (numerics), linear regression (normal eqution, Moore-Penrose, QR, fitting functions)
    Maybe constrained linear optimization aka linear programming, maybe have this as chapter title. And factor out pivoting to numerical programming part.
  - Gradient descent and iterative methods
    SGD, adaptive methods, Newtons method
  - Meta heuristics and searching
    As gradient-free optimization; hill climbing, genetic programming

ILV Numerics and Industrial Algorithms:

> Numerical error analysis, selected numerical methods (regular and overdetermined linear systems of equations, polynomial interpolation, numerical integration and differentiation), selected geometrical algorithms and data structures (convex hull, range search, Voronoi diagrams, Delaunaytriangulation, skeletal structures) and referencing to industrial applications.

- Numerical programming:[7]
  - Representation of numbers
    b-adic expansion, hardware number formats
  - Computing with numbers
    Floting-point arithmetic, numerical analysis
  - Polynomial interpolation
    Maybe factor out differentiation and integration into dedicated chapter
- Computational geometry:
  - Geometric computations
    Maybe move to numerical programming
  - Convex hull
  - Range searching
  - Graphs II (planar and geometric)
  - Voronoi diagrams and Delaunay triangulation
  - Skeleton structures

[7] Re-merger of numerical programming and numerical mathematics

Appendix E

Introduction to continuous optimization

Similar to the many faces of artificial intelligence (AI), also the field of mathematical optimization can be viewed from a large number of viewpoints, touching various different applied disciplines, like *operations research*, *control theory*, *artificial intelligence*, *computer science*, *statistics*, *economics* and so on. We can roughly divide the field into two main branches: continuous and discrete optimization. This chapter gives an introduction to continues optimization.

Mathematical programming. Before we start, we shall briefly clarify a possible confusion with another term: *mathematical programming* is a synonym for optimization. Here the term "programming" is meant in the sense of "creating a plan", namely an optimal plan.[1] Also "computer programming" in this original meaning means to create a plan for the temporal sequence of instructions.[2] Special forms of optimization problems then go by names like *linear programming*, *quadratic programming*, *integer linear programming (ILP)*, or *dynamic programming*. The latter we already know from section 2.3.1. It is not really about computer programming but rather a special for form of mathematical programming.

E.1 Phrasing optimization problems

Mathematical optimization is about finding the best solution x among a non-empty set X of feasible solutions. What makes one solution better than the other is formalized by an *objective function* $f\colon X \to \mathbb{R}$, which is to be minimized or maximized. Although the field of mathematical optimization is vast and diverse, all of it deals with the following problem: Solve

$$\underset{x \in X}{\text{minimize}} \quad f(x) \tag{E.1}$$

Since minimizing f is the same as maximizing $-f$, we do not distinguish between the two cases in terms of methods; it is a matter of taste.

The f comes by many different names. In machine learning we minimize f and call it *loss function*, like when we optimize the weights of neural net to solve a supervised learning task. If f has a more economical flavor, we call it *cost function*, like when we search for shortest paths in graphs. In other cases we want to maximize f. In evolutionary algorithms, we maximize the *fitness function* of individuals in a population. In economics, we maximizing the *profit function* or the *utility function*.

[1] A video cassette recorder (VCR) or a theater is "programmed", too.

[2] Also the formal language *Plankalkül* of Konrad Zuse has in its name the word "plan".

Continuous versus discrete. When X is a continuum we speak of *continuous optimization*. Otherwise X is a discrete set and we speak of *discrete optimization*. When we say that X is a continuum then we mean X is a subset of or equal to $\mathbb{R}^d$ or $\mathbb{C}^d$. It further means that the objective function $f\colon X \to \mathbb{R}$ typically has same regularity properties, like being continuous, or differentiable, or *continuously differentiable*[3] or so on. A common notation is C^k to denote all k-times continously differentiable function $\mathbb{R}^d \to \mathbb{R}$ and an $f \in C^\infty$ is also called a *smooth function*. Polynomial functions are smooth.

Set of solutions. The optimization problem in eq. (E.1) actually gives us the "performance" of the best solution in terms of the objective f. Often, however, we are interested in the actual solution itself, i.e., the x^* that attains the minimum. Then the problem is to solve

$$\begin{array}{ll} \arg\min & f(x) \\ x \in X & \end{array} \tag{E.2}$$

Note that eq. (E.2) could have more than one solution or even infinitely many, like for $\arg\min_{x\in\mathbb{R}} \sin x$. Formally, we define $\arg\min_{x\in X} f(x) = \{x \in X\colon f(x) = \min_{x\in X} f(x)\}$ as the set of all minimizers of f. For instance, $\arg\min_{x\in\mathbb{R}} \sin(x) = \{2k\pi\colon k \in \mathbb{Z}\}$. When we then write "let $x^* = \arg\min_{x\in X} f(x)$", we mean x^* to be *any* element of this set.

Existence of solutions. Note that eq. (E.1) may not be well defined, i.e., eq. (E.2) might be empty. If X is finite then a solution always exists: We could actually order all elements of X by f. If we consider $\min_{x\in\mathbb{R}}(x-1)^2$ then it is clear that the solution is 0. At least for some $x \in \mathbb{R}$ the objective $(x-1)^2$ will be minimal. But what about $\min_{x\in\mathbb{R}} \exp(x)$? We know that $\exp(x) > 0$ for all x, yet the solution is not 0 because for no $x \in \mathbb{R}$ we have $\exp(x) = 0$. But we can make $\exp(x)$ arbitrarily close to zero by choosing x arbitrarily close to $-\infty$. The catch here is that the set $\{\exp(-x)\colon x \in \mathbb{R}\}$ has no minimum. Also the set $\{1, 0.5, 0.25, 0.125, \dots\}$ has no minimum. But they have an *infimum*, which is defined as the largest lower bound of the set. It is zero in these examples. Or for short, $\inf_{x\in\mathbb{R}} \exp(x) = 0$. In the following we will assume that the arg min is non-empty and hence min exists for our problems.[4]

Constrained optimization. In many cases the objective function f is defined over the reals or $\mathbb{R}^d$ for some dimension $d \in \mathbb{N}$. Often f is actually of some simple form, like of linear or quadratic form. Optimizing such f is easy, if we would optimize over the entire $\mathbb{R}^d$. However, the difficulty of the optimization problem actually arises from the fact that we are only interested in a specific subset $X \subset \mathbb{R}^d$. And this subset X is typically specified by some constraints that form this subset X. These constraints are commonly given by n equations $g_1(x) = 0, \dots, g_n(x) = 0$ and m inequalities $h_1(x) \le 0, \dots, h_m(x) \le 0$.

That is, we look for $\min_{x\in X} f(x)$ with $X = \{x \in \mathbb{R}^d\colon g_1(x) = 0, \dots, g_n(x) = 0, h_1(x) \le 0, \dots, h_m(x) \le 0\}$ and call this a *constrained optimization problem* to emphasize this focus on forming X. And instead of this rather cumbersome way of writing down the optimization problem over X it is common to phrase constrained optimization problems in this form:

$$\begin{array}{lll} \underset{x}{\text{minimize}} & f(x) & \\ \text{subject to} & g_i(x) = 0 & i = 1, \dots, n, \\ & h_i(x) \le 0 & i = 1, \dots, m \end{array} \tag{E.3}$$

[3] That means that it is differentiable and its derivative is continuous.

[4] It can be shown that any subset $X \subset \mathbb{R}$ bounded from below has an infimum $\inf X$. This this essentially the completeness property of the real numbers.

Here we even omit that $x \in \mathbb{R}^d$, which is typically clear from the context. There is also a shorter notation commonly used:

$$\begin{aligned} \min_x \quad & f(x) \\ \text{s.t.} \quad & g_i(x) = 0 \qquad i = 1, \dots, n, \\ & h_i(x) \leq 0 \qquad i = 1, \dots, m \end{aligned} \tag{E.4}$$

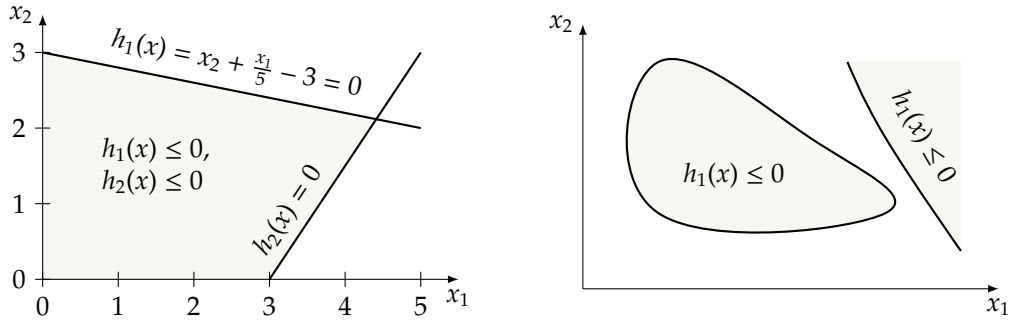
In many practical examples n is zero such that there are only inequality constraints.

Geometric interpretation. For many optimization problems it helps a lot to look at the problem from a geometric perspective, especially for constrained optimization. Let us consider a single inequality constraint $h_i(x) \leq 0$.

If h_i is linear then $h_i(x) = 0$ describes a hyperplane in $\mathbb{R}^d$, like a straight line in $\mathbb{R}^2$. Then $h_i(x) \leq 0$ is the half space with the boundary described by $h_1(x) = 0$. In fact, in section 3.1.2 we defined halfspaces $H^-(n, \lambda)$ as the set of points $x \in \mathbb{R}^d$ with $n \cdot x \leq \lambda$, which means $n \cdot x - \lambda \leq 0$. In other words, if the inequalities are linear then the set X of feasible solutions forms a polyhedron, as we learned in section 3.1.2. It might even form a polytope.

In fig. E.1a we have two linear constraints $h_1(x) \leq 0$ and $h_2(x) \leq 0$. For h_1 we have the equation explicitly given as $h_1(x) = x_2 + \frac{x_1}{5} - 3$, where $x = (x_1, x_2) \in \mathbb{R}^2$. The inequality $h_1(x) \leq 0$ can be rewritten as $x_2 \leq 3 - \frac{x_1}{5}$. The second equation could be phrased as $h_2(x) = 1.5(x_1 - 3) - x_2$. Then $h_2(x) \leq 0$ is equivalent to $x_2 \geq 1.5(x_1 - 3)$. The set $X = \{x \in \mathbb{R}^2 \colon h_1(x) \leq 0, h_2(x) \leq 0\}$ of feasible solutions is shaded in grey and forms a polyhedron. (It is not bounded, so it is not a polytope.)

When the functions h_i are non-linear then the constraints $h_i(x) \leq 0$ can lead to more complicated shapes. Typically we require for h_i some tameness, otherwise we could think of crazy examples, say, $h_i(x) = 1$ if x has only rational coordinates and 0 otherwise. In the real world we may face examples like $h_1(x) = x_1^2 + x_2^2 - 1$, and $h_1(x) \leq 0$ is then the unit disk with radius 1 and the origin as center. The linear case and the circle are examples of a much more general class, namely when h_1 is given by polynomials. Then $h_1(x) = 0$ gives an *algebraic curve* in the case of $\mathbb{R}^2$, or, more generally, algebraic varieties. Taking a geometric viewpoint brings us deep into the field of *algebraic geometry*, which goes beyond the scope of this course.



(a) Linear constraints (b) General constraints

Figure E.1: Geometric interpretations of constraints.

In fig. E.1b we illustrate an example where a single inequality h_1 constraint can lead to a disconnected set X of feasible solutions. Figuratively speaking, we can visualize the function graph of h_1 and think of the isolines of the function graph, which we would receive by a contour

plot. The isoline at height 0 is the solution of $h_1(x) = 0$, and everywhere where the function graph is at this level or lower forms the set $\{x\colon h_1(x) \leq 0\}$. In fig. E.2 we have an example where this set forms an infinite chessboard.

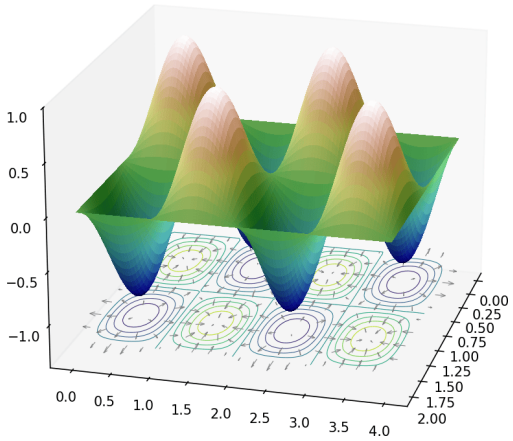


Figure E.2: The non-linear constraint $h_1(x) = \sin(\pi x_1)\sin(\pi x_2)$ leads to a chessboard-like set of feasible solutions $h_1(x) \leq 0$. The contour plot and the gradient field is shown at the bottom.

E.2 Unconstrained optimization

E.2.1 One dimensional optimization

The reader probably faced the first optimization problems already at school, in the context of "curve sketching"[5]. For instance, we are given a polynomial function $f(x) = x^3 - 3x^2 - 4x + 12$ over the reals and we are asked to find the minima, maxima, saddle points, turning points of f. We will focus on minmizing f and hence ask for

$$\min_x \quad x^3 - 4x^2 - 3x + 18 \tag{E.5}$$

We call a point x with $f'(x) = 0$ a *critical point* of f. *Fermat's theorem* says that every local minimum or maximum is a critical point. But the opposite is not true, e.g., also saddle points are critical points.

So one strategy to solve eq. (E.5) is to determine the critical points by solving $f'(x) = 0$ and then classify them by other means, for instance their curvature f''. If $f''(x) > 0$ then x is a local minimum. In order to find the global minimum, we can look at all local minima, which are often only finitely many or we know by other means – like when f is convex – that there is only one local minimum.

So the methods we apply are from calculus and in order for these methods to apply we require some regularity for the objective function, like $f \in C^2$. In the above example we have $f'(x) = 3x^2 - 8x - 3 = (x - 3)(3x + 1)$, so the two critical points are $x = -1/3$ and $x = 3$, see fig. E.3. Furthermore, $f''(x) = 6x - 8$ and hence $x = 3$ is a local minimum as $f''(3) > 0$.

But note that the global minimum of f is not at $x = 3$: the objective becomes arbitrarily small as $x \to -\infty$. In other words, eq. (E.5) is ill-posed. But the following constrained optimization

[5]Dt. Kurvendiskusssion

problem has a real solution:

$$\min_{x \in [0,\infty)} \quad x^3 - 4x^2 - 3x + 18 \tag{E.6}$$

Since the objective function can be factorized as $f(x) = (x+2)(x-3)^2$ we see that on $[0,\infty)$ indeed $x = 3$ is the global (and only) minimum. Whereas the problem

$$\min_{x \in [-3,\infty)} \quad x^3 - 4x^2 - 3x + 18 \tag{E.7}$$

has its minimum at the boundary of the feasible set, namely at $x = -3$.

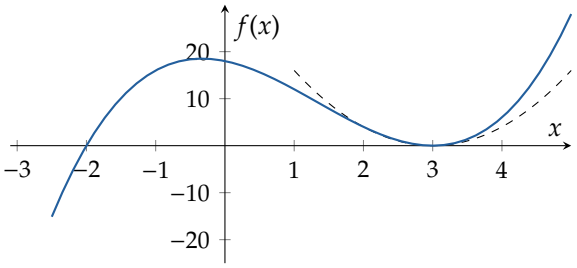


Figure E.3: Minimizig $f(x) = x^3 - 4x^2 - 3x + 18$ by means of calculus. A local minimum is at $x = 3$ with the approximating parabola shown by the dashed line.

Before we go into higher dimensions, let us sharpen our intuition behind this method. Assume that f can by approximated by a Taylor series. If we stop this approximation after the second term we get as an approximation at x^*:

$$f(x) \approx f(x^*) + f'(x^*) \cdot (x - x^*) + f''(x^*) \cdot \frac{(x - x^*)^2}{2!} \tag{E.8}$$

Here $f'(x^*)$ captures the slope of f at x^* and $f''(x^*)$ captures the curvature. For critical points x^* we have $f'(x^*) = 0$ and then this formula becomes

$$f(x) \approx f(x^*) + f''(x^*) \cdot \frac{(x - x^*)^2}{2!}. \tag{E.9}$$

We interpret this formula as follows: f is at all critical points locally approximated by an untilted parabola. The parabola points upwards when $f''(x^*) > 0$, and then x^* is a minimum. For our example in fig. E.3, the approximating parabola is $5(x-3)^2$.

E.2.2 Higher dimensional optimization

The approach of locally approximating a function f up to curvature can be generalized to higher dimensions and therefore yields again a strategy to solve a higher dimensional minimization problem of the type $\min_x f$ with $f\colon \mathbb{R}^d \to \mathbb{R}$. So first we need to generalize f' and f'' to higher dimensions.

The generalization of f' is called the *gradient* ∇f of f. Let us denote $x = (x_1, \ldots, x_d)$ then

$$\nabla f(x) = \left(\frac{\partial f}{\partial x_1}(x), \ldots, \frac{\partial f}{\partial x_d}(x)\right) \tag{E.10}$$

is the vector of partial derivatives along the d variables $x_1, \ldots, x_d$ of f at x.

A powerful intuition is given by this geometric interpretation: We think of ∇f being a vector field $\mathbb{R}^d \to \mathbb{R}^d$, assigning each point in $\mathbb{R}^d$ a direction vector in $\mathbb{R}^d$, namely the direction of highest slope, the strongest ascent in the landscape formed by the function graph of $f\colon \mathbb{R}^d \to \mathbb{R}$. We call it the *gradient field*. In fig. E.2 and fig. E.4 the gradient field is shown at the bottom of the plot. The gradient field points away from local minima and points towards local maxima. The gradient field is orthogonal to the isolines, as every hiker knows. If it is raining on the function graph of f then the water flows in the direction of $-\nabla f$. That is, the water flow is the (negative) gradient flow. And this gradient flow is the zero vector at critical points, like minima, maxima or saddle points. By the way, when we speak in machine learning of the *vanishing gradient problem* then we refer to exactly this gradient in the optimization problem that looks for the best weights for the neural net solving the supervised learning task. We will come back to this in chapter F when we learn about gradient descent methods.

Now we define *critical points* as those where the gradient vanishes, i.e., where $\nabla f = 0$. In fig. E.4 this is the bottom of the (elliptic) paraboloid. Still *Fermat's theorem* says that minima are at critical points, so we identify all critical points by solving $\nabla f = 0$ and then need a tool to sort the critical points out in minima, maxima, or saddle points.

Again the curvature can tell us whether a critical point is a minimum, so we need a higher-dimensional generalization of f'', which is the so-called *Hessian matrix* $H_f(x)$, which is defined as the matrix of all second partial derivatives of f at x:

$$H_f(x) = \begin{pmatrix} \frac{\partial^2 f}{\partial x_1 \partial x_1}(x) & \dots & \frac{\partial^2 f}{\partial x_1 \partial x_d}(x) \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_d \partial x_1}(x) & \dots & \frac{\partial^2 f}{\partial x_d \partial x_d}(x) \end{pmatrix}. \tag{E.11}$$

This allows us to generalize eq. (E.8) to

$$f(x) \approx f(x^*) + \nabla f(x^*) \cdot (x - x^*) + \frac{1}{2}(x - x^*)^\dagger H_f(x^*)(x - x^*). \tag{E.12}$$

Here $v^\dagger$ denotes the transpose of v. For critical points x^* we have $\nabla f(x^*) = 0$ and then this formula becomes

$$f(x) \approx f(x^*) + \frac{1}{2}(x - x^*)^\dagger H_f(x^*)(x - x^*). \tag{E.13}$$

Let us rewrite $(x - x^*)^\dagger H_f(x^*)(x - x^*)$ to the form of $v^\dagger A v$ for some matrix A and a vector v. We call such a matrix A *positive definite*[6] if $v^\dagger A v > 0$ for all vectors $v \neq 0$. Note that in eq. (E.13) this means that the function f is going upwards in all directions from x^*. In other words, this means that we have a local minimum at x^*. In a geometric interpretation $v^\dagger A v$ describes a (elliptic) paraboloid that points upwards.

So let us recapitulate: For a twice differentiable function $f\colon \mathbb{R}^d \to \mathbb{R}$ we find the critical points by solving $\nabla f = 0$ and then take all critical points x^* where the Hessian matrix $H_f(x^*)$ is positive definite to identify all local minima. We again have the geometric interpretation that f is locally approximated at x^* by an upright (elliptic) paraboloid.

Now it would be nice to have a simple method to check whether $H_f(x^*)$ is positive definite. If the second partial derivatives are continuous then H_f is symmetric. And symmetric matrices are positive definite if and only if its eigenvalues are real and positive. Indeed, we can then find

[6] A closely related term is a *positive definite kernel*, which plays an important role in kernel-based machine learning, like for *support vector machine (SVM)*, *principal component analysis (PCA)* and k-means clustering.

a coordinate system of $\mathbb{R}^d$ in which $H_f(x^*)$ is a diagonal matrix and its positive eigenvalues are on the diagonal. The eigenvalues correspond to the axis lengths of the ellipses that form the isolines of the elliptic paraboloid as in fig. E.4. If you want to know more about this then *quadric surfaces* is the keyword to search for.

E.3 Constrained optimization

For constrained optimization problems with equation constraints, we can use the *Lagrange multiplier method*. Let $f\colon \mathbb{R}^d \to \mathbb{R}$ be the objective function and let $g_1, \dots, g_m\colon \mathbb{R}^d \to \mathbb{R}$ be $m \leq d$ constraint functions, all of them being in C^1. Then the method of Lagrange multipliers allows us to solve

$$\begin{aligned} \min_x \quad & f(x) \\ \text{s.t.} \quad & g_i(x) = 0 \qquad i = 1, \dots, m \end{aligned} \tag{E.14}$$

In literature, typically the $g_1, \dots, g_m$ are summarized as a single $g\colon \mathbb{R}^d \to \mathbb{R}^m$ and then the problem can be briefly phrased as $\min f(x)$ s.t. $g(x) = 0$.

A simple example could be to minimize $f\colon \mathbb{R}^2 \to \mathbb{R}\colon f(x_1, x_2) = 2(x_1-1)^2 + (x_2-1)^2 + 1$ subject to $g_1(x_1, x_2) = x_1 + x_2 + 1 = 0$. Note that g_1 describes a straight line. We could rephrase $g_1(x) = 0$ as $x_2 = -x_1 - 1$. The objective function f describes an elliptic paraboloid with a minimum at $(1, 1)$ at height 1. Its isolines are ellipses, see fig. E.4. In a geometric interpretation we intersect the hyperplane vertically erected over the straight line given by g_1 with the paraboloid defined by f, which gives a curve in $\mathbb{R}^3$, and ask for its lowest point.

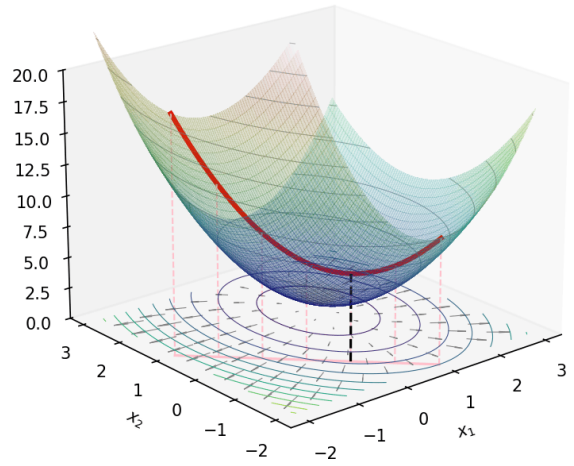


Figure E.4: Minimizing $f(x_1, x_2) = 2(x_1 - 1)^2 + (x_2 - 1)^2 + 1$ subject to $g(x_1, x_2) = x_1 + x_2 + 1 = 0$ using Lagrange multipliers. We search for the lowest point of the red line, marked by the dashed black line. The isolines and the gradient field ∇f are shown at the bottom.

The interesting insight of the Lagrange multiplier method is that we can turn the constrained optimization problem into an unconstrained one by incorporating the constrained $g(x) = 0$ into the objective function. This extended objective function is called the *Lagrangian function* $\mathcal{L}$. It is done as follows: Let $\lambda \in \mathbb{R}^m$, which we call the Lagrange multipliers, then the Lagrangian function is defined as

$$\mathcal{L}(x, \lambda) = f(x) + \lambda^\intercal g(x) = f(x) + \sum_{i=1}^{m} \lambda_i g_i(x). \tag{E.15}$$

Then the solution x^* of eq. (E.14) is a critical point of the Lagrangian function $\mathcal{L}$ in the variables $x_1, \ldots, x_d, \lambda_1, \ldots, \lambda_m$. So we solve the system

$$\frac{\partial \mathcal{L}}{\partial x_1} = 0, \qquad \ldots, \qquad \frac{\partial \mathcal{L}}{\partial x_d} = 0, \qquad \frac{\partial \mathcal{L}}{\partial \lambda_1} = 0, \qquad \ldots, \qquad \frac{\partial \mathcal{L}}{\partial \lambda_m} = 0 \tag{E.16}$$

of equations in order to solve the optimization problem eq. (E.14). However, what we require for Lagrange multipliers to work is the following condition: The gradients $\nabla g_i(x^*)$ of the constraints g_i at x^* must be linearly independent. We can combine the m gradients to a $m \times d$ matrix, called the *Jacobian matrix* $J_g(x)$, which is defined as

$$J_g(x) = \begin{pmatrix} \frac{\partial g_1}{\partial x_1}(x) & \ldots & \frac{\partial g_1}{\partial x_d}(x) \\ \vdots & \ddots & \vdots \\ \frac{\partial g_m}{\partial x_1}(x) & \ldots & \frac{\partial g_m}{\partial x_d}(x) \end{pmatrix} = \begin{pmatrix} \nabla g_1(x) \\ \vdots \\ \nabla g_m(x) \end{pmatrix}. \tag{E.17}$$

Then we can phrase the above condition as follows: The rank of $J_g(x^*)$ at the solution x^* must be full, that is, rank $J_g(x^*) = m$.

In our example shown in fig. E.4 we have for the Lagrangian function

$$\mathcal{L}(x_1, x_2, \lambda) = 2(x_1 - 1)^2 + (x_2 - 1)^2 + 1 + \lambda(x_1 + x_2 + 1). \tag{E.18}$$

Setting all its partial derivatives to zero gives us the following system of equations:

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial x_1} &= 4(x_1 - 1) + \lambda = 0, \\ \frac{\partial \mathcal{L}}{\partial x_2} &= 2(x_2 - 1) + \lambda = 0, \\ \frac{\partial \mathcal{L}}{\partial \lambda} &= x_1 + x_2 + 1 = 0. \end{aligned}$$

Luckily, this system is linear and yields $x_1 = 0$, $x_2 = -1$ and $\lambda = 4$. So our solution is $x^* = (0, -1)$. We only have to check that $J_g(x^*)$ is of full rank. In this case $J_g(x) = (\frac{\partial g_1}{\partial x_1}(x), \frac{\partial g_1}{\partial x_2}(x)) = (1, 1)$, which is of full rank, and therefore x^* is indeed a critical point. However, the method of the Lagrange multipliers does not tell us that a critical point (x_1, x_2) is a minimum. This conclusion must be provided by other means.

We will not go into further details here, but recall one central idea here: We solve a constrained optimization problem by turning it into an unconstrained one, by incorporating the constraint function into the objective function. Lagrange multipliers are a very important tool in economics, physics and classical engineering.

E.4 Beyond Fermat and Lagrange

The methods of Fermat and Lagrange are very powerful and form the basis of optimization theory in the continuous setting. If we look back to the initial problem definitions in section E.1 then we see that we basically treated all continuous problems to large generality. Only in the case of constraint continuous optimization we wish to also be able to handle inequality constraints in addition to the equality constraints treated by Lagrange multipliers. Here in 1951 the so-called *Karush-Kuhn-Tucker (KKT)* conditions filled the gap. In fact, this was the Master thesis of Karush [33] in the year 1939.

For many practical applications, however, we would have troubles to cast the given methods into algorithms. For a certain class of problems – like fitting a polynomial through a set

of points – we could come up with closed formulas based on Fermat and Lagrange. But algorithms generally implementing the methods based on Fermat or Lagrange would be difficult to implement and involve the symbolic computation of quite general systems of equations.

As a result, concrete algorithms for different continuous optimization problems solve the task numerically and iteratively the one or other way, more or less following the ideas of Fermat and Lagrange. We name two prominent examples:

- There are *gradient descent* algorithms, which are most prominently used in machine learning. This is the reason why machine learning frameworks like TensorFlow or PyTorch come with a battery of different gradient descent optimizers. We will learn about this in chapter F.
- When the objective function is not differentiable or even not continuous – and hence there is no gradient ∇f to descent along – then there algorithms like the *Nelder-Mead method*, which is also called the *downhill simplex method*.

Furthermore, many problems in practice have actually a simpler structure, like a linear or quadratic objective function, which we can exploit. Linear optimization problems are mostly interesting due to their constraints: The entire problem boils down to finding the optimum on the boundary of the feasible set X shaped by the constraints. The *simplex algorithm* for linear optimization is such an example. We will learn about this in chapter G.

E.5 Summary

E.6 Exercises

Exercise E.1. Some example of inequalities forming X

Exercise E.2. A half circle formed by constraints

Exercise E.3. some example of a simple linear optimization problem with linear constraints

Exercise E.4. Show that $\mathbb{Q}$ is countable by enumerating it somehow. In order to do so it is convenient to observe that $\mathbb{Q}$ is basically the same as $\mathbb{Z} \times \mathbb{N} = \{(p, y) \colon p \in \mathbb{Z}, y \in \mathbb{N}\}$.

Exercise E.5. Show that for the infinite horizon recall G_t with discount factor γ the following holds:

$$G_t = r_{t+1} + \gamma G_{t+1}. \tag{E.19}$$

Exercise E.6. Polynomial approximation by least squares optimization.

Appendix F

Gradient descent

Appendix G

Linear optimization

Linear optimization. In *linear optimization* the objective function $f\colon \mathbb{R}^d \to \mathbb{R}$ is linear, so we can write it as $f(x) = c^\dagger x$ for some $c \in \mathbb{R}^d$. The objective function grows in direction of c and is constant orthogonally to the vector c.

What makes linear optimization interesting is when we add m linear constraints $g_1(x) \leq 0, g_m(x) \leq 0$ to it. Each constraint g_i is of the form $g_i(x) = a_i^\dagger x - b_i$ and restricts the feasible set X to a half space orthogonal to $a_i \in \mathbb{R}^d$. The feasible set X is then a convex polyhedron, among which we search for the point x^* that minimizes f. We had a corresponding illustration in fig. E.1a.

The m constraints $g_i(x) \leq 0$ can be summarized to one inequality $A \cdot x \leq b$ with $A \in \mathbb{R}^{m \times d}$ and $b \in \mathbb{R}^m$. The linear optimization problem is then

$$\begin{aligned} \min_x \quad & c^\dagger x \\ \text{s.t.} \quad & A \cdot x \leq b \end{aligned} \tag{G.1}$$

We will discussion this setting further after we introduced convexity in chapter 3. The *linear optimization* problem is also called *linear programming*. In fact, mathematical optimization is also called *mathemtical programming*. Here the term "programming" refers to planning, like a director "programs" a theater when he schedules the plays or like you used to "program" a VCR in the 1990s.

Mathematical optimization is in general a difficult problem, so we strive to simplify matters. First of all, recall that $\arg\min_x f(x)$ may have multiple solutions. If $f\colon \mathbb{R}^d \to \mathbb{R}$ then this means that f possesses multiple global minima. This cannot happen when we require convexity for f. Convex optimization is much simpler than general optimization. So let us talk about convexity.

Appendix H

Introduction to discrete optimization

Discrete optimization problems are optimization problems $\min_{x \in X} f(x)$ over a *discrete set* X. This means that X is a finite set or at most a countable set.

The notion of *countable sets* is a fundamental term in set theory and means that we can assign every element $x \in X$ a unique natural number. In other words, a countable set X can be enumerate such that each element is called after some finite time. Two simple examples are finite sets or $\mathbb{N}$ itself. But also $\mathbb{Z}$ is countable as we can enumerate it as $0, 1, -1, 2, -2, \ldots$.

The domain of *discrete mathematics* is about studying discrete structures, that is, discrete sets that have some additional mathematical structure, such as in graph theory, combinatorics, number theory, cryptography, algorithm theory, databases, software engineering and so on. So if we have an optimization problem from one of these we face a discrete optimization problem. Computer science deals with discrete mathematics mostly, so optimization problems from computer science are mostly discrete.

H.0.1 Program optimization

Let us start with a simple, maybe a bit obscure example: Program optimization as we discussed it in chapter 2 fits into the framework of (discrete) optimization problems: Let P be an algorithmic problem and x be a program that solves P. Let us be more concrete: Let x be a Unicode string forming a Python program that solves P, and let X be the set of all x that do so. Next, we evaluate the performance of x by some performance measure $f\colon X \to \mathbb{R}$, like the wallclock time, or the time taken in the RAM model of computing, or the space taken, or maybe even a combination. Then chapter 2 was about heuristics and techniques to solve

$$\min_{x \in X} \quad f(x). \tag{H.1}$$

Well, we did not really solve eq. (H.1) in chapter 2: We learned about heuristics to turn a given x into another x^* with $f(x^*) < f(x)$ and hopefully $f(x^*) \ll f(x)$, but we would never find out whether x^* is optimal.

But for the sake of argument, we could indeed ask whether we could really find an optimal x^*, really solve eq. (H.1). First of all, X is countable. This is true because the set of all strings over a finite alphabet – like Unicode strings – are countable: We could enumerate the string of length 0, then length 1, then length 2 and so on. Some of these strings form valid Python programs, and yet again some of them solve P. Picking the f-minimizing $x \in X$, however, is still difficult because X is still infinite. We can make a constrained optimization problem by limiting the length $|x|$ of the Unicode string x by some constant n and receive an inequality constraint $h_1(x) = |x| - n \leq 0$. Now X is finite and we could pick the f-minimizing $x \in X$ s.t. $h_1(x) \leq 0$.

It is probably needless to say: This is not really a viable approach in practice. Yet we can learn a few very principal observations from this thought experiment, which we will discuss as follows.

Countable is not computable. The set X might be countable from a set-theoretic point of view, but this does not mean that our way of forming X by enumerating all Unicode strings x and testing whether x is a P-solving Python program is computable. Indeed, it is not. In order to decide whether x solves the problem P, we would in particular be able to decide whether x terminates at all. That is, we would need to solve the *halting problem*, which we cannot, as we recall from section 1.2.3.

Optimizing is searching. Even if we limit $|x|$ by a rather restrictive n, say $n = 1000$, and even if we limit the alphabet from Unicode to ASCII or even further to a set of q characters, say $q = 64$, then X is still prohibitively large.[1] So instead of enumerating all elements of X we have to more smartly search within X. And indeed, discrete optimization is a lot about searching algorithms. This is also the reason why AI is a lot about searching, because AI is about intelligent agents in the sense of Russel and Norvig [44] and optimizing their behavior. For instance, instead of enumerating all paths $u \rightsquigarrow v$ in a graph to find the shortest one, we use the much more efficient Dijkstra's algorithm. Or instead of enumerating all possible moves in a chess game, we use much more efficient techniques in searching the game tree of admissible moves, like the *minimax* algorithm or *alpha-beta pruning*.

For the example of program optimization there also smarter ways to search in X. For instance, there is a technique called *genetic programming*, where is a form of evolutionary algorithm for the optimization of programs. A population of individuals is formed, where each individual is representing a program, like a tree of basic program building blocks. Then evolutionary operators – mutation, selection, crossover – are applied to evolve the population from generation to generation. We will learn more about such strategies in chapter I.

No calculus for discrete optimization. We conclude that discrete optimization on very large X is very difficult and obtaining really the optimum is often out of reach, like in program optimization. On the other hand, in continuous optimization X is even larger: $\mathbb{R}$ or any non-empty interval is uncountable. Still we have learned about powerful methods to solve these problems. This seems odd.

The resolution to this seemingly paradox is that calculus is just very powerful and its power stems from the mathematical structure of a continuum $X \subset \mathbb{R}^d$, such that we can build limits in it, and the regular structure of smooth $f\colon X \to \mathbb{R}$, such that we can build derivatives of it. These strong mathematical structures lead to powerful mathematical optimization tools as we have seen in chapter E.

On the other hand, on a discrete set X we lack these structures. In particular, if we would like to speak of a continuous $f\colon X \to \mathbb{R}$ we would need to have a so-called topology on X that gives the notion of open neighborhoods, on which continuity can be defined. Now we could some trivial topology on X, even on the set of programs above, and in fact there is the so-called discrete topology, but a trivial topology will lead to a useless notion of continuity and we have won nothing than mathematical buerocracy.[2]

---

[1] For an alphabet of size q the set of all strings up to length n has a cardinality of $\sum_{k=0}^{n} q^k = \frac{q^{n+1}-1}{q-1}$. With $q = 64$ and $n = 1000$ we get 1.513×10^{1806} strings.

[2] For instance, every function $f\colon X \to \mathbb{R}$ is continuous if X is equipped with the discrete topology.

This is a general observation: continuous mathematics is typically simpler than discrete mathematics. For instance, calculating $\sum_{k=0}^{n} k^7$ is difficult while $\int_0^n x^7\,dx$ is easy. Or let us consider another optimization problem from motion control, namely finding a time-optimal position signal $p\colon [0,\infty) \to \mathbb{R}$ for a servo drive to move from position 0 at rest to a target position t at rest. We have physical constraints like limited velocity $|\dot{p}|$ and limited acceleration $|\ddot{p}|$. This problem is quite simple in the continuous setting, but it is much harder in a time-discrete setting.

H.0.2 Sequential decision making and reinforcement learning

An important class of optimization problems is on optimal acting, like making optimal decisions. This is the central paradigm in the field of *operations research*, in *control theory*, in *game theory* and in *artificial intelligence (AI)*, and in particular in *reinforcement learning (RL)*. The above example of finding an optimal trajectory in motion control can be seen as an example, but we would like to treat this matter in more generality.

Intelligent agents. For this we first need a model in order to tell what we mean by "decision" or "acting". Russel and Norvig gave in their classical book on AI, *Artificial Intelligence: A Modern Approach* [44], a model of an *intelligent agent* that is interacting with an environment: It places actions to an environment through actuators and receives percepts about the environment through sensors, see fig. H.1. This is done sequentially, leading to a *sequential decision making* problem of choosing actions.

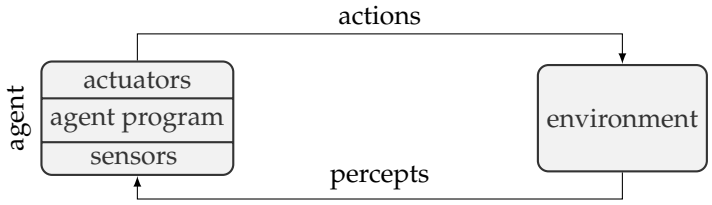


Figure H.1: The model of an intelligent agent as given by Russel and Norvig [44].

Inside the agent there is an *agent program* that maps the – at least potentially – the entire percept sequence received so far to the next action. In addition, the agent might be equipped with built-in knowledge about the environment. If the agent is a chess player then the environment is a chess game and built-in knowledge of the agent refers to the knowledge of chess rules or more. The agent program could be a mathematical mapping from the current chess board to the next move. Then we rather speak of an *agent function*. The agent function could be as simple as a lookup table from the last perception to the next action, which would suffice to play tic-tac-toe in an optimal way.

Russel and Norvig call an agent *rational* if it selects the "right" actions, and an action is "right" if it leads to "good" consequences in the environment and how good a consequence or a state of the environment is, is measured by a *performance measure*.[3] The goal is to optimize performance, so finding a good agent program is an optimization problem and the objective function is defined by means of the performance measure.

[3]There is a strong relation to the philosophical subfield of ethics, which revolves around the question "How shall one act?" There are different schools of thought in so-called normative ethics, like Kant's deontological ethics, utilitarianism, or virtue ethics. Utilitarianism promotes actions that maximize the sum of the utilities of all affected people. This idea is erected on the philosophical position of conquentialism, where ethical value is derived from the consequences of an action. This is also the maxim of virtually all AI approaches.

Reinforcement learning. There are many approaches to come up with suitable agent programs for a given task. Some of them might be quite static, like a fixed lookup table as we mentioned it above for the tic-tac-toe use case. Particularly interesting – especially in the context of AI – are agent programs that learn from the interaction with the environment. The field that studies how we can computationally learn models from data is called *machine learning*. When we use *machine learning* for the agent program then we are in the regime of *reinforcement learning (RL)*. The standard literature here is the book by Sutton and Barto, *Reinforcement Learning: An Introduction* [48].

The model in fig. H.1 is very general and it fits to many problem formulations, including control theory, game theory, and operations research. Sutton and Barto give a more detailed model of the agent, which allows us to describe the RL problem more formally as an optimization problem, see fig. H.2.[4] The environment is described by a state and the set of all possible states is the state space $\mathcal{S}$. The agent picks an action a from a set of possible actions, the action space $\mathcal{A}$. He does so based on the current state s, formalized by the *policy* $\pi\colon \mathcal{S} \to \mathcal{A}$. That is, the agent picks $a = \pi(s)$ as the next action. Upon this action a the environment changes its state to a new state $s \in \mathcal{S}$, which is communicated to the agent, along with an (immediate) reward $r \in \mathbb{R}$.

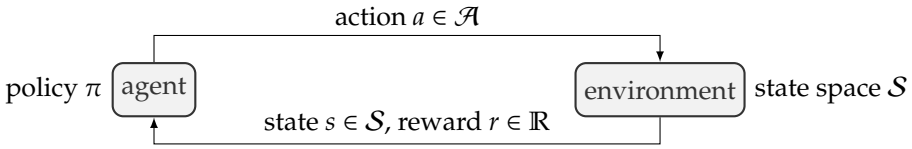


Figure H.2: The standard model of RL by Sutton and Barto [48].

Let us give two examples to illustrate how this modeling is applied:

- Assume our optimization problem is to play tic-tac-toe as in fig. H.3a. The environment is essentially the tic-tac-toe board. It may have different properties, like material properties, weight, and so on. But for our optimization problem all that matters is the configuration of Xs and Os on the 3×3 board. So we can formalize the space as an X-O-N string of length 9, where X stands for a cross, O for a circle, and N for an empty field. The action space $\mathcal{A}$ for the agent – we assume he plays Xs – is the set of all empty fields on the board. Sending an $a \in \mathcal{A}$ to the environment means placing an X at this position.

- Assume we have the control task of the *inverted pendulum*, also known as *pole balancing* task, see fig. H.3b. The environment consists of a car that moves along a horizontal line and has mounted a pole vertically upwards that can rotate around its lower end on the car. The goal is to balance the pole upright. The action space $\mathcal{A}$ could be $[-1, 1]$, namely the acceleration (at some unit) of the cart along the line. The state space needs to cover at least the information the agent requires to fulfill the control task. In this case we could start with the signed angle of the pole deviating from the upright position. So the state space is $[-\pi, \pi]$.

The optimization goal is now to find policies π that maximize the reward over time. To make this more precise we have to tell what we mean by "over time". We introduce the notion of *return*, which is an accumulation of rewards. Say the environment is in state s_0, the agent picks action a_0, which leads to a new state s_1 and a reward r_1. And this repeats in the fashion that $a_n = \pi(s_n)$ leads to s_{n+1} and r_{n+1}. At each time step t the agent receives future rewards $r_{t+1}, r_{t+2}, \ldots$

[4] To be precise, in [48] this model is presented for so-called *Markov decision processes* as the underlying optimization problem.

(a) Tic-tac-toe (b) Pole balancing

Figure H.3: Two typical RL tasks: Tic-tac-toe and pole balancing.

and we want to maximize the sum $G_t = \sum_{n=0}^{\infty} r_{t+1+n}$ of future rewards, which we call the return. However, in general this infinite sum will not converge. There are two common ways to deal with this:

1. We define a *reward* with *finite horizon* T as

$$G_t = r_{t+1} + \cdots + r_{t+T} = \sum_{n=0}^{T-1} r_{t+1+n}. \tag{H.2}$$

   By the horizon T we control how far we look into the future. If $T = 1$ then only the immediate reward r_{t+1} defines G_t and we call the agent *myopic*.

2. We define a *discounted reward* with infinite horizon and a *discount factor* $\gamma \in [0, 1)$ as

$$G_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots = \sum_{n=0}^{\infty} \gamma^n r_{t+1+n}. \tag{H.3}$$

   The discount factor γ controls how much influence rewards in the far future have on the return. If $\gamma = 0$ then only the immediate reward r_{t+1} defines G_t and the agent is again called *myopic*. As γ approaches 1 future rewards have move influence.

   If the r_{t+1+n} are absolutely bounded then G_t converges as $\gamma < 1$. In fact, G_t even converges if $|r_{t+1+n}| \in O(n^c)$ for some c, i.e., if r_{t+1+n} grows at most polynomially.[5]

The return is the objective function of RL and so we can state the optimization problem of RL as follows: RL:

$$\max_{\pi} \quad G_t \tag{H.4}$$

Markov decision processes. Now is the time for a couple of confessions and clean up of a few details we generously skipped. What we framed as RL in general is actually the special case when the environment can be modeled as a *Markov decision process (MDP)*. A MDP is sort of finite state machine where the transition $s \to s'$ from one state s to another state s' is stochastic.

The *Markov property* of this stochastic process says that the transition probability of the transition $s \to s'$ solely depends on s. That is, this process has now "memory" of the history. A

[5] You recall that continuous mathematics is easier than discrete mathematics? To show that G_t converges for $r_{t+1+n} = \lambda n^c$ we realize that $\sum_{n=0}^{\infty} \gamma^n n^c$ can be bounded by the integral $\int_0^{\infty} x^c \gamma^x$, thinking of "area beneath the curve", which is besides constant factors the integral $\int_0^{\infty} x^c e^{-x}$, which is finite. The latter can be checked by partial integration. Or we recall that it is, besides constant factors, the probability density function of the gamma distribution, whose integral must be 1.

MDP as a model is quite general, but we should nevertheless keep in mind that the assumption of the Markov property might be hard to defend for your particular problem.[6]

To sum up, we speak of a MDP if there are fixed conditional probabilities $p(s', r \mid s, a)$ for the transition $s \to s'$ to happen, which then gives the reward r, given the action a in state s. Then G_t is actually a random variable and what we want to maximize is its expected return $E_\pi(G_t)$ when following the policy π. The optimization problem eq. (H.4) then becomes

$$\max_\pi \quad E_\pi(G_t). \tag{H.5}$$

Stochastic policies. Furthermore, so far we modeled the policy π as a function $\mathcal{S} \to \mathcal{A}$. For many RL algorithms, however, we model π as a *stochastic policy*, that is $\pi(s)$ is a probability distribution over $\mathcal{A}$. Then instead of computing $a = \pi(s)$ we actually randomly sample $a \sim \pi(s)$. This again changes the character of the random variable G_t. When we implement this as code then we typically model $\pi(s)$ as a normal distribution $N(\mu, \sigma)$, where μ is the mean and σ the standard deviation. So for every state s we have a μ_s and σ_s giving rise to $\pi(s) = N(\mu_s, \sigma_s)$.

Exploration versus exploitation. The underlying mechanism of RL for all kind of RL algorithms is now that the agent "wanders" through the state space by placing actions and receives rewards and new states. This is the opportunity for an RL agent to learn what actions give rise to good rewards. However, the huge catch here is that an a_t might yield a bad or mediocre immediate reward r_t but larger return G_t through higher later rewards r_{t+T}.[7] If the agent is eager exploiting the current knowledge – based on little interaction so far – he might miss the experience gained by longer exploration. In fig. H.4 the mountain car problem is illustrated: The agent has to learn to accelerate the car up to the right mountain peak. However, the physics of the environment is setup such that it cannot reach the goal without first gaining momentum from the left slope. It has to learn to first accelerate to the left, against the immediate direction of the goal.

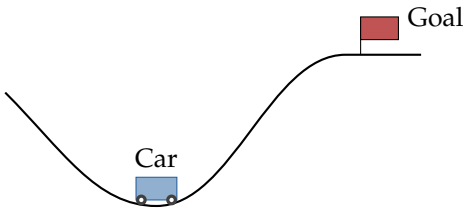


Figure H.4: The mountain car problem: The agent has to learn to move the car up to the right peak, but it has to first move left to gain momentum.

The balance between *exploration* and *exploitation* of the currently learned policy is a central theme in RL. If we employ stochastic policies then we can control the level of exploration by the standard deviations of the policy. Typically we start with a high standard deviation, which is then reduced over time, by whatever mechanism.

Algorithm strategies. For RL there are now a couple of strategies to solve the optimization problem eq. (H.5), leading to a couple of categories of RL algorithms. We will not go into details here, but we will mention a few important ones:

[6] In fact, if you look close enough, it will not hold up for many real-world problems, but it still might not hurt in practice. Just like we assume for many real-world data that it is normally distributed, even if it is not.

[7] In psychology, this is called *delayed gratification*. The famous *marshmallow experiment* is related to this concept.

- *Value function methods* are a class of algorithms that are based on the so-called *value function* $V_\pi \colon \mathcal{S} \to \mathbb{R} \colon s \mapsto E_\pi(G_t \mid s_t = s)$ for the policy π, i.e., the expected return when starting in state s and following the policy π.

  In some sense, similar to mathematical notions in game theory, the value function tells us how good it is to be in state s. We can now consider the optimal policy when starting at state s as follows:

$$\pi_s^* = \arg\max_\pi V_\pi(s). \tag{H.6}$$

  Recall that π_s^* may actually comprise multiple policies; we assume π_s^* is any element of the set. More importantly, however, the question is whether for different starting states s and s' the optimal policies π_s^* and $\pi_{s'}^*$, can behave very differently in terms of the expected return. It turns out that for infinite horizon with discount factor greater than zero for the return, the starting state does not matter: If, after some steps, both policies would reach a common state then they would agree on the actions to take to reach the optimal return from this state on.[8] This allows us to speak of an optimal policy π^* in the above respect, dropping the starting state in the subscript. Now we can define V^* as V_{π^*} and call it the optimal value function.

  If we would know V^* from an oracle then we could reconstruct the optimal policy π^* as follows: Assume we are in state s and we pick the action a then with probability $p(s', r \mid s, a)$ we reach state s' and receive reward r. The expected return is

$$E_{\pi^*}(G_t \mid s_t = s, a_t = a) = \sum_{s' \in S} p(s', r \mid s, a) \Big(r + \gamma V^*(s') \Big). \tag{H.7}$$

  This return shall be maximized by choosing the best action a. In other words, the optimal policy $\pi^* \colon \mathcal{S} \to \mathcal{A}$ does the following:

$$\pi^*(s) = \arg\max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} p(s', r \mid s, a) \Big(r + \gamma V^*(s') \Big). \tag{H.8}$$

  Note that if there is an ambiguity concerning $\arg\max_a$ then there are multiple optimal policies and vice versa.

  The question is how to find V^*? It turns out that V^* fulfills a recursive identity. This essentially stems from the recursion $G_t = r_{t+1} + \gamma G_{t+1}$. More precisely, let us assume we have an MDP where $p(s', r \mid s, a)$ denotes the probability of the state transition $s \to s'$, leading to a return r, given action a at state s. Then for the value function V^* the following holds:

$$\begin{aligned} V^*(s) &= E_{\pi^*}(G_t \mid s_t = s) \\ &= \max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} p(s', r \mid s, a) \Big(r + \gamma V^*(s') \Big). \end{aligned} \tag{H.9}$$

[8] In detail this is not so trivial. In fact, they might take different actions as long as the return is the same. However, it suffices that there are $\pi_1 \in \pi_{s_1}^*$ and $\pi_2 \in \pi_{s_2}^*$ that agree on actions once they reached a common state. Furthermore, if we reach each state from each state in finite time with positive probability then the policies agree on all states since the return looks on the infinite trajectory of states, given that the discount factor is greater than zero. We had a similar argument when we discussed Dijkstra's algorithm in section 4.2.3. The common underlying concept here is dynamic programming. It is seldom to meet a fundamental concept in the intersection of computer science, operations research and control theory.

This recursive equation is called the *Bellman equation*. So-called *value iteration* algorithms solve it to obtain the optimal value function V^* and hence the optimal policy π^*. The value iteration algorithm does what it says: We compute a sequence of value functions $V_0, V_1, V_2, \ldots$ that converge to V^*. We may start with $V_0(s) = 0$ for all $s \in \mathcal{S}$ and then update V_{n+1} from V_n by the right-hand side of eq. (H.9). See algorithm 16 for the algorithm.

The Bellman equation is actually a necessary condition for *dynamic programming* to work. We already know dynamic programming from the section 2.3.1, where we discussed algorithm techniques for problems that are comprised of subproblems of the same type, as in the recursion in eq. (H.9). Bellman also coined the term *curse of dimensionality* in this context, which is now primarily known for the issue that the need of training data grows exponentially with the dimension of the model parameters.

- *Policy gradient methods* are a class of algorithms that directly optimize the policy π by gradient descent. The policy is typically modeled as a neural network. That is, as in supervised learning, we have a parameterized model π_θ and we optimize the model parameters θ as follows:

$$\max_\theta \quad E_{\pi_\theta}(G_t). \tag{H.10}$$

  A popular example of this type is the *proximal policy optimization (PPO)* algorithm.

Model-based versus model-free. The value iteration algorithm in algorithm 16 needs to know the MDP, i.e., it needs to know $p(s', r \mid s, a)$. This is called a *model-based* approach, as it is based on a model of the environment. This is a requirement often not fulfilled in real-world tasks. A model-free approach does not require a model of the environment. A value-based, model-free RL algorithm would be *Q-learning*, which is based on the paradigm of *temporal difference (TD)* learning. See section A.1 in the appendix for more details.

Discrete versus continuous. The two RL examples in fig. H.3 illustrate an important classification of RL tasks: The state space and the action space can either be discrete or continuous. In the tic-tac-toe example both are discrete, while in the pole balancing task both are continuous. Different RL algorithms are suitable for different types of state and action spaces.

Besides the crude distinction between discrete and continuous, it makes sense to think of the mathematical structure of the state and action space. That is, if we have to states $s, s' \in \mathcal{S}$ then it may or may not make sense to compare states like $s \leq s'$ or consider differences $s - s'$ or ratios s/s' between. For instance, the quotient of two postal codes makes no sense, even if they are numbers. Therefore – similar to *statistical data types* – we shall be aware of whether we have *nominal*, *ordinal*, *interval*, and *ratio* state spaces, and what the permissible operations on states are.

Sometimes it is possible to discretize a continuous state or action space, but we have to keep the curse of dimensionality in mind. Vice versa, some discrete spaces we may simply interpret continuously, but this is less suitable if the statistical data type does not admit.

Algorithm 16 Value iteration algorithm.

for $n = 0, 1, 2, \ldots$ **do**
  for $s \in \mathcal{S}$ **do**
    $V_{n+1}(s) \leftarrow \max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} p(s', r \mid s, a) \left(r + \gamma V_n(s') \right)$
  if $\|V_{n+1} - V_n\| < \epsilon$ **then** ▷ *We reached an equilibrium*
    break

Similar to other machine learning (ML) fields, the advent of *Deep RL* caused a big leap in performance of RL algorithms. The idea is basically to use deep neural networks to approximate the policy π or (other functions). However, when we use general approximators, like neural nets, to model $\pi\colon \mathcal{S} \to \mathcal{A}$ then we make assumptions about the structure of $\mathcal{S}$ and $\mathcal{A}$, such as when $\|s - s'\|$ is small then the states $s, s' \in \mathcal{S}$ are similar, for a suitable interpretation of "similar" in your application.

H.0.3 Searching and backtracking

H.1 Summary

H.2 Exercises

Appendix I

Meta heuristics

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] Manindra Agrawal, Neeraj Kayal, and Nitin Saxena. "PRIMES is in P." In: *Annals of Mathematics* 160.2 (2004), pp. 781–793. DOI: 10.4007/annals.2004.160.781.

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

[4] 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.

[5] M.A. Bender, E.D. Demaine, and M. Farach-Colton. "Cache-oblivious B-trees." In: *Proceedings 41st Annual Symposium on Foundations of Computer Science*. 2000, pp. 399–409. DOI: 10.1109/SFCS.2000.892128.

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

[7] CGAL Editorial Board. *CGAL: Computational Geometry Algorithms Library*. URL: https://www.cgal.org/.

[8] Randal E. Bryant and David R. O'Hallaron. *Computer Systems: A Programmer's Perspective*. 3rd ed. Pearson, 2018, p. 1128. ISBN: 978-0134092669.

[9] *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).

[10] *clock_getres(2)*. Linux man-pages. URL: https://www.kernel.org/doc/man-pages/.

[11] Thomas H. Cormen et al. *Introduction to Algorithms*. 4th ed. The MIT Press, 2022. ISBN: 9780262046305.

[12] Scott A. Crosby and Dan S. Wallach. "Denial of Service via Algorithmic Complexity Attacks." In: *12th USENIX Security Symposium (USENIX Security 03)*. Washington, D.C.: USENIX Association, 2003-08.

[13] Erik D. Demaine and Joseph O'Rourke. *Geometric Folding Algorithms: Linkages, Origami, Polyhedra*. Cambridge University Press, 2007. ISBN: 978-0-521-85757-4.

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

[15] Reinhard Diestel. *Graph Theory*. 5th ed. Springer-Verlag, 2017. ISBN: 978-3-11-017875-5.

[16] Edsger W. Dikstra. "EWD1305: Answers to questions from students of Software Engineering." 2000-11. URL: http://www.cs.utexas.edu/users/EWD/ewd13xx/EWD1305.PDF.

[17] Ulrich Drepper. *What Every Programmer Should Know About Memory*. 2007-11. URL: http://people.redhat.com/drepper/cpumemory.pdf.

[18] *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).

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

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

[21] 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.

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

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

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

[25] 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.

[26] John L. Hennessy and David A. Patterson. *Computer Architecture*. 5th ed. Morgan Kaufmann, 2012. ISBN: 978-0-12-383872-8.

[27] John L. Hennesy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. 5th ed. Morgan Kaufmann, 2011. ISBN: 978-0-12-383872-8.

[28] *High Precision Event Timer (HPET) Specification*. Tech. rep. Rev 1.0a. 2004-10. URL: https://web.archive.org/web/20090204075023/http://www.intel.com/hardwaredesign/hpetspec_1.pdf.

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

[30] Stefan Huber. *Stalgo: A Library for the Computation of Straight Skeltons*. URL: https://www.sthu.org/software/stalgo/.

[31] Stefan Huber. *Vroni: A Library for the Computation of Voronoi Diagrams*. URL: https://www.cosy.sbg.ac.at/~held/projects/vroni/vroni.html.

[32] *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.

[33] William Karush. "Minima of Functions of Several Variables with Inequalities as Side Constraints." MA thesis. Department of Mathematics, University of Chicago, 1939. URL: http://pi.lib.uchicago.edu/1001/cat/bib/4111654.

[34] Donald E. Knuth. "Big Omicron and big Omega and big Theta." In: *SIGACT News* 8.2 (1976-04), pp. 18–24. ISSN: 0163-5700. DOI: 10.1145/1008328.1008329.

[35] Donald E. Knuth. "Computer programming as an art." In: *Communications of the ACM* 17.12 (1974-12), pp. 667–673. ISSN: 0001-0782. DOI: 10.1145/361604.361612.

[36] Donald E. Knuth. *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. 3rd ed. Addison-Wesley, 1997. ISBN: 0-201-89683-4.

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

[38] 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.

[39] Frederick P. Brooks. *The Mythical Man-Month: Essays on Software Engineering*. 1st ed. Addison-Wesley, 1975. ISBN: 978-0-201-00650-6.

[40] *perf: Linux profiling with performance counters*. 2024. URL: https://perf.wiki.kernel.org/index.php/Main_Page (visited on 08/12/2024).

[41] Alexander Pikus. *The Art of Writing Efficient Programs*. Packt Publishing, 2021, p. 464. ISBN: 978-1800208117.

[42] 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.

[43] Alfréd Rényi and Rolf Sulanke. "Über die konvexe Hülle von n zufällig gewählten Punkten." In: *Zeitschrift für Wahrscheinlichkeitstheorie und verwandte Gebiete* 2 (1963), pp. 75–84. DOI: 10.1007/BF00535300.

[44] Stuart Russel and Peter Norvig. *Artificial Intelligence: A Modern Approach*. 4th ed. Pearson, 2020. ISBN: 978-0134610993.

[45] Robert Sedgewick and Kevin Wayne. *Algorithms in C++*. 4th ed. Addison-Wesley, 2011. ISBN: 978-0-321-87758-3.

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

[47] 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.

[48] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. 2nd ed. MIT Press, 2018. ISBN: 978-0262039246.

[49] *TLSF: Two-Level Segregate Fit Memory Allocator*. 2024. URL: http://rtportal.upv.es/rtmalloc (visited on 08/14/2024).

[50] *Valgrind*. 2024. URL: http://valgrind.org/ (visited on 08/12/2024).

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

Index