Introduction to Numerical Mathematics

Lars Grüne
Chair of Applied Mathematics
Mathematical Institute
University of Bayreuth
95440 Bayreuth
Germany
lars.gruene@uni-bayreuth.de
num.math.uni-bayreuth.de

Lecture Notes

English translation of the 7th edition
Winter Semester 2023/2024

Preface

These notes are prepared for a lecture with the same name that will be taught during the winter semester 2023/2024 at the University of Bayreuth. It is the English translation of the seventh edition of a document originally developed during the winter semester 2002/2003 under the title "Numerical Mathematics I". This edition has been partially updated and corrected in comparison to the sixth edition from the winter semester 2018/2019. These lecture notes were translated from the German with the help of ChatGPT, followed by a careful proofreading.

This text serves as lecture notes and primarily aims to provide a concise written summary of the material covered in the lecture for the participants. It is not intended to be a replacement for a textbook on Numerical Mathematics. While writing various sections, I found the textbooks [1, 7, 8, 9] and the lecture notes [3, 4, 5] to be very helpful.

Bayreuth, February 2024 Lars Grüne

Contents

Chapter 1

Introduction

Numerical Mathematics — often also called Numerical Analysis or short Numerics — deals with the development and analysis of algorithms for solving mathematical problems on a computer. Unlike symbolic or analytic calculations, the goal here is not to obtain closed-form formulas or algebraic expressions as results, but quantitative numerical values[1], hence the name "Numerical Mathematics".

In this introductory course on Numerics, traditionally, various mathematical problems are addressed. In this lecture, we will focus on the following topics:

- Systems of linear equations
- Interpolation
- Integration
- Nonlinear equations and systems of nonlinear equations

An important area missing in this list are differential equations, which will be covered in detail in subsequent lectures on ordinary and partial differential equations.

The multitude of different problems from analysis and linear algebra means that various mathematical techniques from these areas are used for numerical solutions. For this reason, the first Numerics lecture is often perceived as a "grab bag" in which different topics are discussed seemingly without a clear connection. However, despite the diverse mathematics involved, there are a number of fundamental principles that are important in Numerics. Before we delve into the "hard" mathematics in the next chapter, we will briefly and informally explain these principles in this introduction.

1.1 Correctness

One of the essential tasks of numerical mathematics is to verify the correctness of algorithms, i.e., to ensure when and under what conditions the correct result is computed for

[1] which are often graphically presented

the given problem data. This verification should be done using mathematical methods, resulting in a formal mathematical proof that guarantees the correct functioning of an algorithm. In many cases, an algorithm will not provide an exact result in a finite number of steps but rather an approximate solution or a sequence of approximate solutions. In this case, it is also necessary to examine how large the error of the approximate solution is depending on the available parameters and how quickly the sequence of approximate solutions converges to the exact value.

1.2 Efficiency

Once the correctness of an algorithm has been established, the next step is to consider the efficiency of the algorithm. If the algorithm provides an exact result in a finite number of steps, the essential task is to count the number of operations. If a sequence of approximate solutions is computed, it is necessary to investigate the number of operations needed for computing the approximate solution and the convergence rate towards the exact solution.

There are often many different algorithms for solving a problem, and their efficiency can vary depending on the specific properties of the problem.

1.3 Robustness and Condition

Even if an algorithm theoretically delivers an exact result in a finite number of steps, this will rarely be the case in numerical practice. The reason for this lies in the so-called *roundoff errors*: Internally, a computer can only represent a finite set of numbers, so it is impossible to represent every real (and not even every rational) number exactly. We will examine this point more formally. For a given base $B \in \mathbb{N}$, every real number $x \in \mathbb{R}$ can be represented as

$$x = m \cdot B^e,$$

where $m \in \mathbb{R}$ is the mantissa, and $e \in \mathbb{Z}$ is the exponent. By choosing e appropriately, it is sufficient to use numbers of the form $m = \pm 0.m_1 m_2 m_3 \ldots$ with digits $m_1, m_2, \ldots$ such that $m_i \in \{0, 1, \ldots, B-1\}$. Computers typically use the base $B = 2$ because numbers are represented as binary numbers. In a computer, only a finite number of digits are available for m and e, e.g., l digits for m and n digits for e. We write $m = \pm 0.m_1 m_2 m_3 \ldots m_l$ and $e = \pm e_1 e_2 \ldots e_n$. Subject to the additional normalization condition $m_1 \neq 0$, there is a unique representation of the so-called *machine-representable numbers*

$$\mathcal{M} = \{x \in \mathbb{R} \mid \pm 0.m_1 m_2 m_3 \ldots m_l \cdot B^{\pm e_1 e_2 \ldots e_n}\} \cup \{0\}.$$

Numbers that do not belong to this set $\mathcal{M}$ must be rounded to obtain a machine-representable number.

These resulting inaccuracies obviously affect the results of numerical algorithms. The robustness of an algorithm (also referred to as numerical stability) is determined by how much these roundoff errors affect the result. In fact, robustness is mathematically considered for general errors, so it does not matter whether they are caused by roundoff or other error sources (input or transmission errors, inaccuracies in previous calculations, etc.).

An important tool for examining this problem is the concept of the *condition* of a mathematical problem. If we abstractly view the problem as a mapping $\mathcal{A} : D \to S$ from problem data $d \in D$ (e.g., matrices, measurements, etc.) to the solution $s = A(d) \in S$ of the problem (e.g., a vector), the condition indicates how small changes Δd in the data d affect the solution s, i.e., how large Δs in

$$s + \Delta s = A(d + \Delta d)$$

is compared to Δd. This is quantitatively determined by analyzing the derivative of $\mathcal{A}$. When small changes Δd cause significant changes Δs, the problem it called *ill conditioned*. Note that the condition is a property of the posed problem and is independent of the algorithm used. However, the robustness of an algorithm is particularly important for poorly conditioned problems since small errors in the algorithm can result in large errors in the result. This theory is maturely developed in the context of linear systems of equations, which we will also examine in detail within this framework.

1.4 Mathematical Techniques

As mentioned earlier, different problem classes require quite different mathematical techniques. However, there are some recurrent methods, i.e., techniques that appear in one way or another repeatedly. For example, consider the principle of orthogonality or orthogonal projection: We encounter this in the efficient solution of linear systems of equations (in the Householder algorithm), polynomial interpolation (with Chebyshev polynomials), and numerical integration (with Gaussian quadrature). All these methods have in common that, with respect to suitable criteria, they belong to the best methods in their class. However, their operation is not immediately apparent, as their derivation relies on sophisticated mathematical ideas. Other examples include Banach's fixed-point theorem, by which we can analyze many different iterative methods, or the Taylor expansion, which forms the basis of many algorithms in numerical analysis.

Chapter 2

Systems of linear equations

Algorithms for solving systems of linear equations form the basis for many applications in numerical mathematics and are therefore traditionally covered at the beginning of many numerical mathematics courses. When written out in detail, the problem is to determine numbers $x_1, \ldots, x_n \in \mathbb{R}$ for which the equation system

$$\begin{array}{ccccccccc} a_{1\,1}x_1 & + & a_{1\,2}x_2 & + & \ldots & + & a_{1\,n}x_n & = & b_1 \\ & & & & & & \vdots & & \\ a_{n\,1}x_1 & + & a_{n\,2}x_2 & + & \ldots & + & a_{n\,n}x_n & = & b_n \end{array} \tag{2.1}$$

is satisfied. The detailed notation in (2.1) is somewhat cumbersome, which is why we will write linear equation systems in the usual matrix form, namely as

$$Ax = b, \tag{2.2}$$

with

$$A = \begin{pmatrix} a_{1\,1} & a_{1\,2} & \ldots & a_{1\,n} \\ \vdots & \vdots & & \vdots \\ a_{n\,1} & a_{n\,2} & \ldots & a_{n\,n} \end{pmatrix}, \quad x = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} \quad \text{and} \quad b = \begin{pmatrix} b_1 \\ \vdots \\ b_n \end{pmatrix}. \tag{2.3}$$

This notation not only allows us to write an equation system much more concisely, but it will also be shown that certain properties of the matrix A determine what method can sensibly be used to solve (2.2).

A few short remarks on notation: We will typically denote matrices with capital letters (e.g., A) and vectors with lowercase letters (e.g., b). Their entries will be denoted with indexed lowercase letters, as in (2.3). A superscript "T" denotes transposed matrices and vectors, so for A and x from (2.3), for example,

$$x = (x_1, \ldots, x_n)^T, \quad A^T = \begin{pmatrix} a_{1\,1} & a_{2\,1} & \ldots & a_{n\,1} \\ \vdots & \vdots & & \vdots \\ a_{1\,n} & a_{2\,n} & \ldots & a_{n\,n} \end{pmatrix}.$$

Since the number n of equations in (2.1) is equal to the number of unknowns $x_1, \ldots, x_n$, A in (2.2) is a square matrix of dimension $n \times n$. For square matrices, it is known from

linear algebra that a unique solution to (2.2) exists if and only if the matrix is invertible. In this chapter, we will always assume that this is the case.

In the next section we will consider an application problem whose solution leads to the problem of solving a system of linear equations.

2.1 An Application: Least Squares Estimation

Our first application example is particularly important for many practical purposes, which is why we want to examine it in more detail.

Suppose we have conducted an experiment in which we obtained measurement values $z_1, z_2, \ldots, z_m$ for various input values $t_1, t_2, \ldots, t_m$. Based on theoretical considerations (e.g., based on an underlying physical law), we know of a function $f(t)$ for which $f(t_i) = z_i$ should hold. This function, in turn, depends on unknown parameters $x_1, \ldots, x_n$, and we write $f(t; x)$ to emphasize this. For example, $f(t; x)$ could be given by

$$f(t; x) = x_1 + x_2 t \quad \text{or} \quad f(t; x) = x_1 + x_2 t + x_3 t^2.$$

In the first case, the sought-after function describes a straight line, while in the second case, it describes a (generalized) parabola. If we assume that the function f exactly describes the experiment and there are no measurement errors, we could determine the parameters x_i by solving the (in general nonlinear) equation system

$$\begin{aligned} f(t_1; x) &= z_1 \\ &\vdots \\ f(t_k; x) &= z_k \end{aligned} \tag{2.4}$$

for x. In many practical cases, this equation system is linear, such as in the two examples above, where (2.4) reduces to $\widetilde{A}x = z$ with

$$\widetilde{A} = \begin{pmatrix} 1 & t_1 \\ \vdots & \vdots \\ 1 & t_m \end{pmatrix} \text{ or } \widetilde{A} = \begin{pmatrix} 1 & t_1 & t_1^2 \\ \vdots & \vdots & \vdots \\ 1 & t_m & t_m^2 \end{pmatrix} \text{ and } z = \begin{pmatrix} z_1 \\ \vdots \\ z_m \end{pmatrix}.$$

These systems of linear equations have m equations (one for each data pair (t_i, z_i)) and n unknowns (namely, the unknown parameters x_i), with m typically much larger than n. This equation system is said to be *over-determined*. Since measurement values of an experiment are practically always subject to errors, it is overly optimistic to assume that the equation system $\widetilde{A}x = z$ is solvable (over-determined equation systems often have no solution!). Instead of attempting the (presumably futile) task of finding an exact solution x to this system, we want to try to find the best possible approximation solution. That is, if $\widetilde{A}x = z$ is not solvable, we want to find an x such that $\widetilde{A}x$ is as close as possible to z. To do this, we need to choose a criterion for "as close as possible" that makes sense and allows for a straightforward solution. The so-called *Least Squares Problem* (also known as the *Method of Least Squares*) is a suitable choice:

Find $x = (x_1, \ldots, x_n)^T$ such that $\varphi(x) := \|\widetilde{A}x - z\|^2$ is minimized.

Here, $\|y\|$ denotes the Euclidean norm of a vector $y \in \mathbb{R}^n$, i.e.,

$$\|y\| = \sqrt{\sum_{i=1}^{n} y_i^2}.$$

To minimize the function φ, we set the gradient $\nabla\varphi(x)$ to zero. Note that the gradient of the function $g(x) := \|f(x)\|^2$ for any $f : \mathbb{R}^n \to \mathbb{R}^n$ is given by $\nabla g(x) = 2Df(x)^T f(x)$. Thus, we obtain

$$\nabla\varphi(x) = 2\widetilde{A}^T\widetilde{A}x - 2\widetilde{A}^T z.$$

If $\widetilde{A}$ has full column rank, the second derivative $D^2\varphi(x) = 2\widetilde{A}^T\widetilde{A}$ is positive definite, which means that every zero of the gradient $\nabla\varphi$ is a minimum of φ. Therefore, a vector x minimizes the function φ if and only if the so-called *Normal Equations* are satisfied:

$$\widetilde{A}^T\widetilde{A}x = \widetilde{A}^T z.$$

So, the least squares problem is solved as follows:

$$\text{solve } Ax = b \text{ with } A = \widetilde{A}^T\widetilde{A} \text{ and } b = \widetilde{A}^T z.$$

The initially seemingly complicated minimization problem for φ is thus reduced to solving a linear system of equations.

In addition to the method of least squares, there are many other applications in numerical analysis that lead to solving a linear system of equations. Some of these will be encountered later in this lecture, such as solving *nonlinear* systems of equations using the *Newton method*, which requires solving a sequence of linear systems, or *interpolation* of points using *splines*.

2.2 The Gauss Elimination Method

Now we introduce the first method for solving systems of linear equations. The *Gauss elimination method* is an intuitive approach that is relatively easy to implement. It is based on the simple fact that a linear system $Ax = b$ is easily solvable if the matrix A is in *upper triangular form*, i.e.,

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,n} \\ 0 & a_{2,2} & \dots & a_{2,n} \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \dots & 0 & a_{n,n} \end{pmatrix}.$$

In this case, we can solve $Ax = b$ easily using the recursive formula:

$$x_n = \frac{b_n}{a_{n,n}}, \quad x_{n-1} = \frac{b_{n-1} - a_{n-1,n}x_n}{a_{n-1,n-1}}, \dots, x_1 = \frac{b_1 - a_{1,2}x_2 - \dots - a_{1,n}x_n}{a_{1,1}}$$

or, compactly,

$$x_i = \frac{b_i - \sum_{j=i+1}^{n} a_{i,j} x_j}{a_{i,i}}, \quad i = n, n-1, \ldots, 1 \tag{2.5}$$

(with the convention $\sum_{j=n+1}^{n} a_{i,j} x_j = 0$). This method is known as *backward substitution.*

The idea of the Gauss elimination method is to transform the equation system $Ax = b$ into a system $\widetilde{A}x = \tilde{b}$ such that the matrix $\widetilde{A}$ is in upper triangular form. Let us illustrate this method with an example.

Example 2.1 Consider the linear equation system with matrix A and vector b as follows:

$$A = \begin{pmatrix} 1 & 5 & 6 \\ 7 & 9 & 6 \\ 2 & 3 & 4 \end{pmatrix}, \quad b = \begin{pmatrix} 29 \\ 43 \\ 20 \end{pmatrix}.$$

To transform matrix A into upper triangular form, we need to eliminate the entries 7, 2, and 3 below the diagonal by performing row operations. We start with the entry 2. To eliminate it, we subtract 2 times the first row from the third row, resulting in:

$$A_1 = \begin{pmatrix} 1 & 5 & 6 \\ 7 & 9 & 6 \\ 0 & -7 & -8 \end{pmatrix}$$

We perform the same operation on vector b:

$$b_1 = \begin{pmatrix} 29 \\ 43 \\ -38 \end{pmatrix}$$

Next, we eliminate the 7 in the third row by subtracting 7 times the first row from the second row:

$$A_2 = \begin{pmatrix} 1 & 5 & 6 \\ 0 & -26 & -36 \\ 0 & -7 & -8 \end{pmatrix}, \quad b_2 = \begin{pmatrix} 29 \\ -160 \\ -38 \end{pmatrix}$$

Finally, we eliminate the -7 in the third row by subtracting $\frac{7}{26}$ times the second row from the third row:

$$A_3 = \begin{pmatrix} 1 & 5 & 6 \\ 0 & -26 & -36 \\ 0 & 0 & \frac{22}{13} \end{pmatrix}, \quad b_3 = \begin{pmatrix} 29 \\ -160 \\ \frac{66}{13} \end{pmatrix}$$

Now, we have $\widetilde{A} = A_3$ and $\tilde{b} = b_3$. We can solve for x using backward substitution, resulting in:

$$x_3 = \frac{\frac{66}{13}}{\frac{22}{13}} = 3, \quad x_2 = \frac{-160 - 3 \cdot (-36)}{-26} = 2, \quad x_1 = \frac{29 - 2 \cdot 5 - 3 \cdot 6}{1} = 1.$$

□

We now formulate the algorithm for general square matrices A.

Algorithm 2.2 (Gaussian Elimination, Basic Version)
Let A be a matrix in $\mathbb{R}^{n \times n}$ and b be a vector in $\mathbb{R}^n$.

(1) For j from 1 to $n-1$ (j = column index of the entry to be eliminated)

(2) For i from n to $j+1$ (counting backward)
(i = row index of the entry to be eliminated)
Subtract $\frac{a_{ij}}{a_{jj}}$ times the j-th row from the i-th row:
Let $\alpha := \frac{a_{ij}}{a_{jj}}$ and compute

$$a_{ik} := a_{ik} - \alpha\, a_{jk} \text{ for } k = j, \ldots, n, \quad b_i := b_i - \alpha\, b_j$$

End of the i-loop

(3) End of the j-loop

□

It is possible that this algorithm does not lead to a result, even if the system of equations is solvable. The reason for this is in step (1), where division by the diagonal element a_{jj} occurs. It is assumed here, without explicit verification, that this element is nonzero, which is not necessarily the case. Fortunately, there is a way to address this.

Let us assume that we want to execute step (1) for given indices i and j, and $a_{jj} = 0$. Now, there are two possibilities: In the first case, $a_{ij} = 0$. In this case, there is no need to do anything because the element a_{ij} that should be brought to 0 is already equal to 0. In the second case, $a_{ij} \neq 0$. In this case, we can swap the i-th and j-th rows of matrix A as well as the corresponding entries in vector b, which achieves the desired property $a_{ij} = 0$, but not through elimination, but by swapping. This procedure is called *pivoting*, and the following algorithm indeed transforms any linear system into triangular form.

Algorithm 2.3 (Gaussian Elimination with Pivoting)
Let A be a matrix in $\mathbb{R}^{n \times n}$ and b be a vector in $\mathbb{R}^n$.

(1) For j from 1 to $n-1$ (j = column index of the entry to be eliminated)

(1a) If $a_{jj} = 0$, choose a $p \in \{j+1, \ldots, n\}$ with $a_{pj} \neq 0$ and
swap a_{jk} and a_{pk} for $k = j, \ldots, n$ as well as b_p and b_j

(2) For i from n to $j+1$ (counting backward)
(i = row index of the entry to be eliminated)
Subtract $\frac{a_{ij}}{a_{jj}}$ times the j-th row from the i-th row:
Let $\alpha := \frac{a_{ij}}{a_{jj}}$ and compute

$$a_{ik} := a_{ik} - \alpha\, a_{jk} \text{ for } k = j, \ldots, n, \quad b_i := b_i - \alpha\, b_j$$

End of the i-loop

(3) End of the j-loop

□

The element a_{pj} that is swapped with a_{jj} in step (1a) is called the *pivot element*. In Section 2.5 we will learn to know a strategy where — even if $a_{jj} \neq 0$ — an element $a_{pj} \neq 0$ is deliberately selected as the pivot element, since this can improve the robustness of the algorithm.

2.3 LR Factorization

In the previous version of the Gaussian method, we transformed the matrix A into upper triangular form and applied all the necessary operations to the vector b at the same time. There are alternative methods for solving linear systems where the vector b remains unchanged. We will now introduce a procedure in which the matrix A is decomposed into a product of two matrices L and R, i.e., $A = LR$, where R is an upper triangular matrix and L is a *lower triangular matrix*

$$L = \begin{pmatrix} l_{11} & 0 & \ldots & 0 \\ \vdots & \ddots & \ddots & \vdots \\ l_{n-1\,1} & \ldots & l_{n-1\,n-1} & 0 \\ l_{n1} & l_{n2} & \ldots & l_{nn} \end{pmatrix}.$$

The decomposition $A = LR$ is referred to as the *LR factorization*.

To solve $Ax = b$, one can solve $LRx = b$ in two steps: First, solve the equation system $Ly = b$. This can be done using *forward substitution*, similar to backward substitution (2.5):

$$y_i = \frac{b_i - \sum_{j=1}^{i-1} l_{ij} y_j}{l_{ii}}, \qquad i = 1, 2, \ldots, n \tag{2.6}$$

In the second step, solve the equation system $Rx = y$ by backward substitution. Then, we have

$$Ax = LRx = Ly = b,$$

which solves the desired system $Ax = b$.

The Gaussian elimination method can be extended in such a way that, except for row swaps in A, it yields an LR factorization of an invertible matrix A.

The subtraction "'i-th row $-$ α times j-th row"' in the m-th step of the algorithm can be achieved by left multiplication with the matrix

$$F_m = \begin{pmatrix} 1 & & & & & & \\ & \ddots & & & & & \\ & & \ddots & & & & \\ & -\alpha & & \ddots & & & \\ & & & & \ddots & & \\ & & & & & 1 \end{pmatrix} \quad \text{with} \quad F_m^{-1} = \begin{pmatrix} 1 & & & & & & \\ & \ddots & & & & & \\ & & \ddots & & & & \\ & \alpha & & \ddots & & & \\ & & & & \ddots & & \\ & & & & & 1 \end{pmatrix}$$

where α is in the i-th row and j-th column.

If Gaussian elimination can be performed without row swapping, then only operations with such matrices are needed. If we multiply the matrices F_1, $F_2, \ldots, F_q$ required for this process into a single matrix $F = F_q \cdot F_{q-1} \cdots F_1$, we obtain $A = LR$ with $L = F^{-1} = F_1^{-1} \cdot F_2^{-1} \cdots F_q^{-1}$, which is the LR factorization (note that the inverse of F^{-1}, a product of lower triangular matrices, is still a lower triangular matrix). The construction of L can also be done recursively as follows

$$L^{(0)} := \mathrm{Id}, \quad L^{(m+1)} = L^{(m)} F_{m+1}^{-1} \tag{2.7}$$

for $m = 0, \ldots, q-1$, and $L = L^{(q)}$.

If row swaps are necessary due to pivoting, then an LR factorization cannot be obtained in general.[1] The swapping of the p-th and j-th rows in A in the m-th step of the algorithm can be achieved by left multiplication with matrices of the form

$$P_m = \begin{pmatrix} 1 & & & & & & & & & \\ & \ddots & & & & & & & & \\ & & 1 & & & & & & & \\ & & & 0 & & & & 1 & & \\ & & & & 1 & & & & & \\ & & & & & \ddots & & & & \\ & & & & & & 1 & & & \\ & & & 1 & & & & 0 & & \\ & & & & & & & & 1 & \\ & & & & & & & & & \ddots \\ & & & & & & & & & & 1 \end{pmatrix}$$

where the 0 entries are precisely at the p-th and j-th positions. If such row swaps are required, you can also multiply the necessary matrices $P_1, \ldots, P_q$ into a single matrix $P = P_q \cdots P_1$ (P is called a *permutation matrix*). For simplicity, we assign a permutation

[1] In fact, for general invertible matrices, an LR factorization does not always exist.

matrix P_m to each elimination matrix F_m and set $P_m = \mathrm{Id}$ if no swap is performed before the m-th elimination. Therefore, we have the same number of permutation matrices P_m as elimination matrices F_m. Note that $P_m^{-1} = P_m$ always holds.

This permutation must be taken into account when constructing the matrix L using (2.7). To explain this, we use the following notation:

Let A_m be the matrix generated after the m-th step of the algorithm (i.e., $A_0 = A$ and $A_q = R$ if q is the total number of steps in the algorithm). Let $P^{(m)} = P_m \cdots P_1$ be the accumulated matrix of permutations up to step m. Finally, let $L^{(m)}$ be the lower triangular matrix with

$$P^{(m)} A = L^{(m)} A_m. \tag{2.8}$$

For this, we have $L^{(0)} = \mathrm{Id}$, and if no permutation is performed up to step m, we have $L^{(m)} = F_1^{-1} \cdot F_2^{-1} \cdots F_m^{-1}$ as described above.

Let P_{m+1} and F_{m+1} be the pivoting and elimination matrices in the $(m+1)$-th step. We want to calculate $L^{(m+1)}$ such that

$$P^{(m+1)} A = L^{(m+1)} A_{m+1} \tag{2.9}$$

holds. Since the Gaussian algorithm first performs pivoting and then elimination, we have

$$A_{m+1} = F_{m+1} P_{m+1} A_m, \quad \text{so} \quad A_m = P_{m+1} F_{m+1}^{-1} A_{m+1}.$$

It follows that

$$P^{(m+1)} A = P_{m+1} P^{(m)} A = P_{m+1} L^{(m)} A_m = P_{m+1} L^{(m)} P_{m+1} F_{m+1}^{-1} A_{m+1}.$$

Therefore, if we set

$$L^{(m+1)} := P_{m+1} L^{(m)} P_{m+1} F_{m+1}^{-1},$$

we get (2.9). Note that the multiplication of $L^{(m)}$ with P_{m+1} from the right just swaps the p-th and j-th columns; only row and column swaps are added compared to (2.7). Because this swap only affects the diagonal and entries below the diagonal up to column $j-1$, and since $p > j$ during swapping, $L^{(m)}$ remains a lower triangular matrix after this operation. We can calculate the matrices $L^{(m)}$ recursively, starting with $L^{(0)} = \mathrm{Id}$, and obtain $PA = LR$, an LR factorization of the row-swapped matrix A.

Further details on implementing this algorithm can be found in books such as Deuflhard/Hohmann [1], Schwarz/Köckler [8], or Stoer [9].

2.4 The Cholesky Method

Another way to obtain an LR factorization is through the following algorithm, known as the *Cholesky method*. This method works not for general matrices but only for *symmetric, positive definite* matrices, providing a particularly elegant form of the LR factorization.

Definition 2.4 (i) A matrix $A \in \mathbb{R}^{n \times n}$ is called *symmetric* if $A^T = A$.

(ii) A matrix $A \in \mathbb{R}^{n\times n}$ is called *positive definite* if $\langle x,\, Ax\rangle > 0$ for all vectors $x \in \mathbb{R}^n$ with $x \neq (0 \ldots 0)^T$.

Here, $\langle x,\, y\rangle$ denotes the *Euclidean* or *standard scalar product* in $\mathbb{R}^n$, i.e., for vectors $x,\, y \in \mathbb{R}^n$,

$$\langle x,\, y\rangle = x^T y = \sum_{i=1}^{n} x_i y_i.$$

□

This property is clearly restrictive, but it holds in many applications. For example, the least squares problem generally leads to a system of equations with a symmetric and positive definite matrix A.

For such matrices, it can be shown that there always exists an LR factorization that additionally has the elegant form $R = L^T$, meaning that it suffices to compute the lower triangular matrix L. The proof follows constructively from the following algorithm.

Algorithm 2.5 (Cholesky Method) Let $A \in \mathbb{R}^{n\times n}$ be a given symmetric and positive definite matrix.

(0) Set $i = 1$ and $j = 1$ (row and column indices of the current entry l_{ij} of L).

(1) (a) If $i > j$, set

$$l_{ij} = \frac{a_{ij} - \sum_{k=1}^{j-1} l_{ik} l_{jk}}{l_{jj}}$$

(b) If $i = j$, set

$$l_{ii} = \sqrt{a_{ii} - \sum_{k=1}^{i-1} l_{ik}^2}$$

(c) If $i < j$, set $l_{ij} = 0$

(2) If $i \leq n-1$, set $i := i+1$ and continue with (1); otherwise: If $j \leq n-1$, set $j := j+1$ and $i := 1$ and continue with (1); otherwise: End of the algorithm

□

Clearly, this algorithm is not as intuitive as Gaussian elimination. To prove that this algorithm provides the correct result, we explicitly write the equation $A = LL^T$ and solve for L. However, doing this directly for arbitrary dimensions can be very cumbersome, which is why we proceed by induction on the dimension of matrix A. In order to **start the induction**, for 2×2 matrices, we obtain:

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} = \begin{pmatrix} l_{11} & 0 \\ l_{21} & l_{22} \end{pmatrix} \begin{pmatrix} l_{11} & l_{21} \\ 0 & l_{22} \end{pmatrix} = \begin{pmatrix} l_{11}^2 & l_{11} l_{21} \\ l_{21} l_{11} & l_{21}^2 + l_{22}^2 \end{pmatrix}. \tag{2.10}$$

From this, we obtain the following:

$$\begin{aligned} l_{1\,1} &= \sqrt{a_{1\,1}} \\ l_{2\,1} &= a_{2\,1}/l_{1\,1} \\ l_{2\,2} &= \sqrt{a_{2\,2} - l_{2\,1}^2} \end{aligned}$$

which exactly corresponds to the calculations in Algorithm 2.5 for $i = 1, 2$ and $j = 1, 2$. The first and last equations are real solutions because A is positive definite, which implies $a_{1\,1} > 0$ and $\det(A)/a_{1\,1} > 0$.

For larger matrices, we proceed by induction. We express A as:

$$A = \begin{pmatrix} A_{n-1} & \bar{a}_n \\ \bar{a}_n^T & a_{n\,n} \end{pmatrix}$$

If A is symmetric and positive definite, then A_{n-1} also possesses these properties. As the **induction hypothesis**, we assume that the Cholesky algorithm for the $(n-1) \times (n-1)$ matrix A_{n-1} correctly computes the LR factorization in the form $A_{n-1} = L_{n-1}L_{n-1}^T$.

We express the sought lower triangular matrix L as:

$$L = \begin{pmatrix} L_{n-1} & 0 \\ \bar{l}_n^T & l_{n\,n} \end{pmatrix}$$

with $\bar{l}_n = (l_{n\,1} \ \ldots \ l_{n\,n-1})^T$.

For the **induction step**, we must verify the equations in the Cholesky algorithm for $i = n$ and $j = 1, \ldots, n$ (the equations for $i \leq n-1$ already yield the correct matrix L_{n-1} due to the induction hypothesis). To do this, we consider the equation $A = LL^T$, which, with the above notation, becomes:

$$\begin{pmatrix} A_{n-1} & \bar{a}_n \\ \bar{a}_n^T & a_{n\,n} \end{pmatrix} = \begin{pmatrix} L_{n-1} & 0 \\ \bar{l}_n^T & l_{n\,n} \end{pmatrix} \begin{pmatrix} L_{n-1}^T & \bar{l}_n \\ 0 & l_{n\,n} \end{pmatrix} = \begin{pmatrix} L_{n-1}L_{n-1}^T & L_{n-1}\bar{l}_n \\ (L_{n-1}\bar{l}_n)^T & \bar{l}_n^T\bar{l}_n + l_{n\,n}^2 \end{pmatrix}. \tag{2.11}$$

If we solve for the entries in the vector $\bar{l}_n$ through forward substitution from the equation system $L_{n-1}\bar{l}_n = \bar{a}_n$, we precisely obtain the equations for $l_{n,j}$, $j = 1, \ldots, n-1$ from Algorithm 2.5. Furthermore, by solving the equation $\bar{l}_n^T\bar{l}_n + l_{n\,n}^2 = a_{n\,n}$, we derive the equation for $j = n$ in Algorithm 2.5. It can be seen as follows that this equation has real solution: Certainly, there exists a solution $l_{n\,n}$, which may be complex. As a result, the factorization $A = LL^T$ exists, where L may have complex entries. Then, $\det(A) = \det(L)^2$, and from the induction hypothesis, we have $\det(A_{n-1}) = \det(L_{n-1})^2$ (due to $A_{n-1} = L_{n-1}L_{n-1}^T$). Moreover, $\det(L)^2 = \det(L_{n-1})^2 l_{n\,n}^2$ (due to the form of L), which implies:

$$l_{n\,n}^2 = \det(L)^2/\det(L_{n-1})^2 = \det(A)/\det(A_{n-1})$$

Since A is positive definite (and thus A_{n-1} is as well), $l_{n\,n}^2$ is real and positive, making $l_{n\,n}$ real.

2.5 Error Estimation and Condition Numbers

As explained in the introductory chapter, computers cannot represent all real numbers exactly. Therefore, all numbers are internally rounded to fit into the finite set of *machine-representable numbers*. This leads to *rounding errors*. Even when both the input values and the result of an algorithm are machine-representable numbers, such errors can occur because intermediate results of an algorithm (which may not be representable) are also rounded. Due to these errors, as well as input or measurement errors in the given data or errors from previous numerical computations, an algorithm typically computes not the exact solution x of the linear equation system

$$Ax = b$$

but an approximate solution $\tilde{x}$. To formalize this, we introduce a "perturbed" or "disturbed" equation system. First, we consider perturbations in the vector b; we will investigate perturbations in the matrix A in Theorem 2.13.

Therefore, we consider the equation system

$$A\tilde{x} = b + \Delta b,$$

for which $\tilde{x} = x + \Delta x$ is the exact solution. Here, the vector Δb is called the *residual* or the *defect* of the approximate solution $\tilde{x}$. We call the vector $\Delta x = \tilde{x} - x$ the *error* of the approximate solution $\tilde{x}$. Since rounding and other sources of errors typically only result in small errors, it is justified to assume that $\|\Delta b\|$ is "small". The goal of this section is to relate the size of the residual $\|\Delta b\|$ to the size of the error $\|\Delta x\|$. In particular, we want to examine how *sensitive* the size $\|\Delta x\|$ is to $\|\Delta b\|$, i.e., whether small residuals $\|\Delta b\|$ can cause large errors $\|\Delta x\|$. This analysis is independent of the solution method used, as we are only considering the equation system itself and not a specific method.

To conduct this analysis, we need the concept of a *matrix norm*. Matrix norms can be defined quite generally, but for our purposes, the concept of an *induced matrix norm* is sufficient. We will first recall various vector norms for vectors $x \in \mathbb{R}^n$. In this lecture, we typically use the *Euclidean norm* or 2*-norm*

$$\|x\|_2 = \sqrt{\sum_{i=1}^{n} x_i^2},$$

which we usually denote simply as $\|x\|$. Other norms include the 1*-norm*

$$\|x\|_1 = \sum_{i=1}^{n} |x_i|$$

and the *maximum* or ∞*-norm*

$$\|x\|_\infty = \max_{i=1,\dots,n} |x_i|.$$

We will see shortly that the last two norms have certain advantages in the context of systems of linear equations.

For all norms in $\mathbb{R}^n$, one can define an associated *induced matrix norm*.

Definition 2.6 Let $\mathbb{R}^{n\times n}$ be the set of $n \times n$ matrices, and let $\|\cdot\|_p$ be a vector norm in $\mathbb{R}^n$. Then, we define the *induced matrix norm* $\|A\|_p$ associated with $\|\cdot\|_p$ for $A \in \mathbb{R}^{n\times n}$ as follows:

$$\|A\|_p := \max_{\substack{x\in\mathbb{R}^n \\ \|x\|_p=1}} \|Ax\|_p = \sup_{x\in\mathbb{R}^n\setminus\{0\}} \frac{\|Ax\|_p}{\|x\|_p}.$$

□

The equality of the two expressions on the right side follows from the relationship $\|\alpha x\|_p = |\alpha|\,\|x\|_p$ for $\alpha \in \mathbb{R}$. It will be proven in the exercises that these are indeed norms on the vector space $\mathbb{R}^{n\times n}$.

Since there is usually no risk of confusion, we use the same symbol for vector norms and the induced matrix norms associated with them.

Theorem 2.7 For the induced matrix norms associated with the vector norms mentioned above and $A \in \mathbb{R}^{n\times n}$, the following equations hold:

$$\begin{aligned}
\|A\|_1 &= \max_{j=1,\ldots,n} \textstyle\sum_{i=1}^n |a_{ij}| && \text{(column sum norm)}\\
\|A\|_\infty &= \max_{i=1,\ldots,n} \textstyle\sum_{j=1}^n |a_{ij}| && \text{(row sum norm)}\\
\|A\|_2 &= \sqrt{\rho(A^TA)} && \text{(spectral norm),}
\end{aligned}$$

where $\rho(A^TA)$ denotes the maximum eigenvalue of the symmetric and positive definite matrix A^TA. □

Proof: We prove the equations by proving the corresponding inequalities.

"$\|A\|_1$, $\leq$": For any vector $x \in \mathbb{R}^n$, we have

$$\|Ax\|_1 = \sum_{i=1}^n \left|\sum_{j=1}^n a_{ij}x_j\right| \leq \sum_{i=1}^n\sum_{j=1}^n |a_{ij}|\,|x_j| = \sum_{j=1}^n |x_j| \sum_{i=1}^n |a_{ij}|.$$

Let j^* be the index for which the inner sum is maximized, i.e.,

$$\sum_{i=1}^n |a_{ij^*}| = \max_{j=1,\ldots,n} \sum_{i=1}^n |a_{ij}|.$$

Then, for vectors with $\|x\|_1 = 1$, we have

$$\sum_{j=1}^n |x_j| \sum_{i=1}^n |a_{ij}| \leq \underbrace{\sum_{j=1}^n |x_j|}_{=1} \sum_{i=1}^n |a_{ij^*}| = \sum_{i=1}^n |a_{ij^*}|,$$

which implies, since x was arbitrary, the inequality

$$\|A\|_1 \leq \sum_{i=1}^n |a_{ij^*}| = \max_{j=1,\ldots,n} \sum_{i=1}^n |a_{ij}|.$$

"$\geq$": For the j^*-th unit vector e_{j^*} we obtain

$$\|A\|_1 \geq \|Ae_{j^*}\|_1 = \sum_{i=1}^{n} \underbrace{\left| \sum_{j=1}^{n} a_{i\,j}[e_{j^*}]_j \right|}_{=|a_{i\,j^*}|} = \sum_{i=1}^{n} |a_{i\,j^*}| = \max_{j=1,\ldots,n} \sum_{i=1}^{n} |a_{i\,j}|$$

and thus, the claimed inequality.

"$\|A\|_\infty$": Similar to the 1-norm, with $x^* = (\pm 1, \ldots, \pm 1)^T$ instead of e_{j^*}.

"$\|A\|_2$, $\leq$": Since A^TA is symmetric, we can choose an orthonormal basis of eigenvectors $v_1, \ldots, v_n$, i.e., $\|v_i\|_2 = 1$ and $\langle v_i, v_j \rangle = 0$ for $i \neq j$. Let $x \in \mathbb{R}^n$ be an arbitrary vector with length 1, then x can be expressed as a linear combination $x = \sum_{i=1}^n \mu_i v_i$ with $\sum_{i=1}^n \mu_i^2 = 1$. Let λ_i be the eigenvalues of A^TA corresponding to the eigenvectors v_i, and let λ_{i^*} be the maximal eigenvalue, i.e., $\lambda_{i^*} = \rho(A^TA)$. Then, we have

$$\begin{aligned}
\|Ax\|_2^2 &= \langle Ax, Ax \rangle = \langle A^TAx, x \rangle \\
&= \sum_{i,j=1}^{n} \mu_i \mu_j \langle \underbrace{A^TAv_i}_{=\lambda_i v_i}, v_j \rangle \\
&= \sum_{\substack{i,j=1 \\ i \neq j}}^{n} \mu_i \mu_j \lambda_i \underbrace{\langle v_i, v_j \rangle}_{=0} + \sum_{i=1}^{n} \mu_i^2 \lambda_i \underbrace{\langle v_i, v_i \rangle}_{=1} = \sum_{i=1}^{n} \mu_i^2 \lambda_i.
\end{aligned}$$

Therefore, we have

$$\|Ax\|_2^2 = \sum_{i=1}^{n} \mu_i^2 \lambda_i \leq \underbrace{\sum_{i=1}^{n} \mu_i^2}_{=1} \lambda_{i^*} = \lambda_{i^*},$$

which implies that, since x was arbitrary,

$$\|A\|_2^2 \leq \lambda_{i^*} = \rho(A^TA).$$

"$\geq$": For proving the converse inequality, we use the inequality

$$\|A\|_2^2 \geq \|Av_{i^*}\|_2^2 = \langle A^TAv_{i^*}, v_{i^*} \rangle = \lambda_{i^*} \underbrace{\langle v_{i^*}, v_{i^*} \rangle}_{=1} = \lambda_{i^*} = \rho(A^TA).$$

□

For any matrix norm, we can define the corresponding *condition number* of an invertible matrix.

Definition 2.8 For a given matrix norm $\|\cdot\|_p$, the *condition number* of an invertible matrix A (with respect to $\|\cdot\|_p$) is defined as

$$\mathrm{cond}_p(A) := \|A\|_p \|A^{-1}\|_p.$$

□

When we consider the ratio between the error Δx and the residual Δb, we can either consider the *absolute values* of these values, i.e., $\|\Delta x\|_p$ and $\|\Delta b\|_p$, or, which is often more meaningful, the *relative sizes* $\|\Delta x\|_p/\|x\|_p$ and $\|\Delta b\|_p/\|b\|_p$. The following theorem shows how to estimate the error from the residual.

Theorem 2.9 Let $\|\cdot\|_p$ be a vector norm with the associated (and equally denoted) induced matrix norm. Let $A \in \mathbb{R}^{n\times n}$ be a given invertible matrix, and b, $\Delta b \in \mathbb{R}^n$ be given vectors. Let x, $\tilde{x} \in \mathbb{R}^n$ be the solutions to the linear systems

$$Ax = b \text{ and } A\tilde{x} = b + \Delta b.$$

Then, for the error $\Delta x = \tilde{x} - x$, the following estimates hold:

$$\|\Delta x\|_p \leq \|A^{-1}\|_p \|\Delta b\|_p \tag{2.12}$$

and

$$\frac{\|\Delta x\|_p}{\|x\|_p} \leq \text{cond}_p(A) \frac{\|\Delta b\|_p}{\|b\|_p}. \tag{2.13}$$

□

Proof: Let $C \in \mathbb{R}^{n\times n}$ and $y \in \mathbb{R}^n$ be any matrix and vector, respectively. Then, as proved in the exercises

$$\|Cy\|_p \leq \|C\|_p \|y\|_p. \tag{2.14}$$

Let $\tilde{x} = x + \Delta x$, and subtract the equation

$$Ax = b$$

from the equation

$$A(x + \Delta x) = b + \Delta b$$

to obtain

$$A\Delta x = \Delta b.$$

Since A is invertible, we can multiply both sides of the equation on the left by A^{-1} to obtain

$$\Delta x = A^{-1}\Delta b.$$

Thus, we have

$$\|\Delta x\|_p = \|A^{-1}\Delta b\|_p \leq \|A^{-1}\|_p \|\Delta b\|_p,$$

where we have used (2.14) with $C = A^{-1}$ and $y = \Delta b$. This proves (2.12). From (2.14) with $C = A$ and $y = x$, we have

$$\|b\|_p = \|Ax\|_p \leq \|A\|_p \|x\|_p,$$

and therefore

$$\frac{1}{\|x\|_p} \leq \frac{\|A\|_p}{\|b\|_p}.$$

If we apply this inequality first and then (2.12), we obtain

$$\frac{\|\Delta x\|_p}{\|x\|_p} \leq \frac{\|A\|_p \|\Delta x\|_p}{\|b\|_p} \leq \frac{\|A\|_p \|A^{-1}\|_p \|\Delta b\|_p}{\|b\|_p} = \mathrm{cond}_p(A) \frac{\|\Delta b\|_p}{\|b\|_p},$$

which proves (2.13). □

For matrices whose condition number $\mathrm{cond}_p(A)$ is large, small errors in the vector b (or rounding errors in the procedure) can lead to large errors in the result x. In this case, one speaks of *ill-conditioned* matrices. The following example illustrates this phenomenon.

Example 2.10 Consider the system of equations $Ax = b$ with

$$A = \begin{pmatrix} 1 & 0 \\ 1000 & 1 \end{pmatrix} \quad \text{and} \quad b = \begin{pmatrix} 0.001 \\ 1 \end{pmatrix}.$$

It is easy to check that the solution is given by $x = (0.001, 0)^T$. Similarly, one can easily verify that

$$A^{-1} = \begin{pmatrix} 1 & 0 \\ -1000 & 1 \end{pmatrix}$$

holds. In the row sum norm, we have $\|A\|_\infty = 1001$ and $\|A^{-1}\|_\infty = 1001$, and therefore,

$$\mathrm{cond}_\infty(A) = 1001 \cdot 1001 \approx 1,000,000.$$

For the perturbed right-hand side $\tilde{b} = (0.002, 1)^T$, i.e., $\Delta b = (0.001, 0)^T$, the solution obtained is $\tilde{x} = (0.002, -1)^T$. This means $\Delta x = (0.001, -1)^T$, and thus,

$$\frac{\|\Delta x\|_\infty}{\|x\|_\infty} = \frac{1}{0.001} = 1000 \quad \text{and} \quad \frac{\|\Delta b\|_\infty}{\|b\|_\infty} = \frac{0.001}{1} = 0.001.$$

The relative error of 0.001 in $\tilde{b}$ is amplified by a factor of $1,000,000$ compared to the relative error of 1000 in $\tilde{x}$, which is nearly equal to the condition number $\mathrm{cond}_\infty(A)$. □

In addition to perturbations in b, disturbances in A can also occur. In that case, the equation system $Ax = b$ is replaced by

$$(A + \Delta A)\tilde{x} = b \tag{2.15}$$

is solved. In this case, the error Δx in $\tilde{x} = x + \Delta x$ no longer depends linearly on the perturbation ΔA, and we need to define the condition concept differently. We do so as follows.

Definition 2.11 Let $GL(n) \subset \mathbb{R}^{n \times n}$ be the set of invertible matrices, and let $f : GL(n) \to \mathbb{R}^n$ be given by $f(A) := A^{-1}b$. Then we define the *absolute condition number* of the problem (2.15) as $\kappa_{abs} := \|Df(A)\|_p$ and the *relative condition number* as $\kappa_{rel} := \frac{\|A\|_p}{\|f(A)\|_p} \|Df(A)\|_p$.
□

This definition is based on the fact that the Taylor expansion $f(A + \Delta A) \approx f(A) + Df(A)\Delta A$ implies that for the solution $\tilde{x}$ of (2.15), we have

$$\tilde{x} = (A + \Delta A)^{-1}b = f(A + \Delta A) \approx f(A) + Df(A)\Delta A = x + Df(A)\Delta A.$$

This results in an approximate relationship

$$\|\Delta x\|_p \approx \|Df(A)\Delta A\|_p \leq \|Df(A)\|_p \, \|\Delta A\|_p$$

and with $x = A^{-1}b = f(A)$

$$\frac{\|\Delta x\|_p}{\|x\|_p} \approx \frac{\|Df(A)\Delta A\|_p}{\|x\|_p} \leq \frac{\|Df(A)\|_p \, \|\Delta A\|_p}{\|x\|_p} = \frac{\|A\|_p}{\|f(A)\|_p}\|Df(A)\|_p \frac{\|\Delta A\|_p}{\|A\|_p},$$

which motivates the definitions. To calculate $Df(A)$, the following lemma is helpful.

Lemma 2.12 The map $g : GL(n) \to GL(n)$, $g(A) = A^{-1}$ is differentiable, and for all $C \in \mathbb{R}^{n\times n}$, it holds that

$$Dg(A)C = -A^{-1}CA^{-1}.$$

Proof: Since A is invertible, $A+tC$ is also invertible for t sufficiently close to 0. Therefore, the expressions in the equation

$$\mathrm{Id} = (A + tC)(A + tC)^{-1}$$

are well-defined for $t \in (-\varepsilon, \varepsilon)$ and sufficiently small $\varepsilon > 0$. We can differentiate both sides with respect to t using the product and chain rules, yielding

$$0 = C(A + tC)^{-1} + (A + tC)\frac{d}{dt}(A + tC)^{-1},$$

which leads to

$$\frac{d}{dt}(A + tC)^{-1} = -(A + tC)^{-1}C(A + tC)^{-1}$$

as the result. Since $Dg(A)C$ is precisely the directional derivative of g at point A in the direction of C, we have $Dg(A)C = \frac{d}{dt}\big|_{t=0}(A + tC)^{-1}$, which confirms the claim. □

This leads to the following theorem.

Theorem 2.13 For the condition numbers in Definition 2.11, it holds with $x = A^{-1}b$

$$\kappa_{abs} \leq \|A^{-1}\|_p\|x\|_p \quad \text{and} \quad \kappa_{rel} \leq \mathrm{cond}_p(A).$$

Proof: We have

$$Df(A)C = \frac{d}{dt}\bigg|_{t=0} f(A + tC) = \frac{d}{dt}\bigg|_{t=0} g(A + tC)b = Dg(A)Cb$$

and, consequently, using Lemma 2.12, $Df(A)C = Dg(A)Cb = -A^{-1}CA^{-1}b = -A^{-1}Cx$. This results in

$$\kappa_{abs} = \|Df(A)\|_p = \max_{\|C\|_p=1} \|Df(A)C\|_p = \max_{\|C\|_p=1} \|A^{-1}Cx\|_p \leq \|A^{-1}\|_p \|x\|_p$$

and, as $f(A) = x$,

$$\kappa_{rel} = \frac{\|A\|_p}{\|x\|_p}\|Df(A)\|_p \leq \frac{\|A\|_p}{\|x\|_p}\|A^{-1}\|_p\|x\|_p = \|A\|_p\|A^{-1}\|_p = \mathrm{cond}_p(A).$$

□

The condition with respect to disturbances in A is determined by the same factors as the condition with respect to disturbances in b.

For poorly conditioned matrices, rounding errors in the algorithm can have a similar impact to errors in the matrix A or the right-hand side b. An important criterion in designing a solution method is that the method should still work reliably for poorly conditioned matrices. In the Gauss elimination method, one can achieve this by choosing the pivot elements in a way that minimizes rounding errors.

To do this, we need to consider which operations in Gauss elimination are particularly error-prone. While this can be done in great detail and formality, we will focus on a more heuristic criterion here: The computational operations in Gauss elimination involve subtractions

$$a_{ik} - \frac{a_{ij}}{a_{jj}}a_{jk}, \quad b_i - \frac{a_{ij}}{a_{jj}}b_j$$

In principle, in step (1a) of the algorithm, we can swap any other row with the j-th row as long as it has the same number of leading zero entries, which is true for rows $j, \ldots, n$. This gives us the freedom to replace the *row* index "'j"' with any index $p \in \{j, \ldots, n\}$, which is accomplished in the algorithm by swapping the j-th row with the p-th row. Note that the "'j"' in "'a_{ij}"' is the *column* index of the element to be eliminated, and it doesn't change with the row swapping. Therefore, row swapping can only affect the elements a_{jj}, a_{jk}, and b_j, or in other words, the fractions a_{jk}/a_{jj} and b_j/a_{jj}.

The main source of rounding errors in a subtraction "$c - d$" on a computer occurs when the difference to be calculated is small in magnitude, and the individual terms c and d are significantly larger in magnitude in comparison. To illustrate this, let us assume we are performing calculations in the decimal system with 5-digit precision. If we subtract the numbers 1.234 from 1.235, we get the correct result of 0.001, but if we subtract 1000.234 from 1000.235, due to internal rounding to 5 digits, the calculation becomes $1000.2 - 1000.2 = 0$, which introduces a significant error (this specific error is also known as *subtractive cancellation*). The strategy to minimize such errors in Gauss elimination is to choose the row index p during pivoting in a way that makes the expressions to be subtracted small in magnitude, a technique called *pivot search*. Since we can only influence the fractions a_{jk}/a_{jj} ($k = j, \ldots, n$) and b_j/a_{jj} through row swapping, we should choose p so that these fractions become as small as possible in magnitude. A simple approach often found in literature is to choose the *pivot element* a_{pj} (the denominator of the fractions) to maximize $|a_{pj}|$.

In the following Algorithm 2.14, we use a slightly more sophisticated strategy that takes into account the numerators of the appearing fractions as well.

Algorithm 2.14 (Gauss Elimination with Pivot Search)
Given a matrix $A \in \mathbb{R}^{n\times n}$ and a vector $b \in \mathbb{R}^n$.

(1) For j from 1 to $n-1$ (j = column index of the entry to be eliminated)

(1a) Choose from the row indices $p \in \{j, \ldots, n \,|\, a_{p\,j} \neq 0\}$ the one for which the expression

$$K(p) = \max\left\{ \max_{k=j,\ldots,n} \frac{|a_{p\,k}|}{|a_{p\,j}|}, \frac{|b_p|}{|a_{p\,j}|} \right\}$$

is minimized. If $p \neq j$, swap $a_{j\,k}$ and $a_{p\,k}$ for $k = j, \ldots, n$ as well as b_p and b_j.

(2) For i from n to $j+1$ (counting backwards)
(i = row index of the entry to be eliminated)
Subtract $a_{i\,j}/a_{j\,j}$ times the j-th row from the i-th row:

Set $\alpha := a_{i\,j}/a_{j\,j}$ and compute

$$a_{i\,k} := a_{i\,k} - \alpha\, a_{j\,k} \text{ for } k = j, \ldots, n, \quad b_i := b_i - \alpha\, b_j$$

End of the i loop

(3) End of the j loop

□

The pivot search method used here is called *column pivot search*, as it searches for the best pivot element $a_{p\,j}$ within the j-th column. An extended form is the *complete* or *total pivot search*, where rows are also searched, and column swaps may be performed as well. Good implementations of Gauss elimination always use such pivot search methods. These methods enhance robustness but do not provide complete protection against large errors in the case of poor conditioning — due to fundamental mathematical reasons, as we will explain in the next section.

A more general strategy for dealing with poorly conditioned systems of equations is called *preconditioning*, where a matrix $P \in \mathbb{R}^{n\times n}$ is sought such that the condition of PA is smaller than that of A, so that the better conditioned problem $PAx = Pb$ can be solved. We will revisit this when discussing iterative methods.

Another strategy for addressing poorly conditioned systems of equations, which we will examine in more detail, is the QR factorization (or QR decomposition) of a matrix.

2.6 QR Factorization

The LR factorization, which has been explicitly or implicitly the basis for the solution methods considered so far, has a significant drawback from a conditioning perspective. It can happen that the individual matrices L and R of the decomposition have much larger condition numbers than the decomposed matrix A.

Example 2.15 Consider the matrix

$$A = \begin{pmatrix} 0.001 & 0.001 \\ 1 & 2 \end{pmatrix} \quad \text{with} \quad A^{-1} = \begin{pmatrix} 2000 & -1 \\ -1000 & 1 \end{pmatrix}$$

and the LR factorization

$$L = \begin{pmatrix} 1 & 0 \\ 1000 & 1 \end{pmatrix}, \quad R = \begin{pmatrix} 0.001 & 0.001 \\ 0 & 1 \end{pmatrix}.$$

Due to

$$L^{-1} = \begin{pmatrix} 1 & 0 \\ -1000 & 1 \end{pmatrix} \quad \text{and} \quad R^{-1} = \begin{pmatrix} 1000 & -1 \\ 0 & 1 \end{pmatrix}$$

we have

$$\text{cond}_\infty(A) \approx 6000, \quad \text{cond}_\infty(L) \approx 1000000, \quad \text{and} \quad \text{cond}_\infty(R) \approx 1000.$$

So, the ∞-condition of L is approximately 166 times larger than that of A. □

Even if we neglect possible errors in the computation of R and L or reduce them through clever pivot selection, the poor conditioning of R and L can lead to large errors Δx due to the amplification of rounding errors during forward and backward substitution, especially when the matrix A itself is poorly conditioned. In the LR factorization, it can happen that the condition of the subproblems that arise in the algorithm is significantly worse than that of the original problem. Note that the condition of the original problem depends only on the problem statement, while that of the subproblems depends on the algorithm used, which is why they are also referred to as *numerical condition*.

To reduce the numerical condition, we now want to consider a different form of factorization in which the condition of the individual subproblems (or the associated matrices) is not greater than that of the original problem (i.e., the original matrix A).

To do this, we use the following matrices.

Definition 2.16 A matrix $Q \in \mathbb{R}^{n \times n}$ is called *orthogonal* if $QQ^T = \text{Id}$ holds[2]. □

Our goal now is an algorithm that decomposes a matrix A into a product QR, where Q is an orthogonal matrix and R is an upper triangular matrix.

It is clear that a system of equations in the form of $Qy = b$ is easy to solve by performing the matrix multiplication $y = Q^T b$. Therefore, we can solve the equation system $Ax = b$ as in the LR factorization by solving the subproblems $Qy = b$ and $Rx = y$.

Before deriving the corresponding algorithm, we want to prove that with this form of factorization, the condition indeed remains the same — at least for the Euclidean norm.

Theorem 2.17 Let $A \in \mathbb{R}^{n \times n}$ be an invertible matrix with a QR factorization. Then,

$$\text{cond}_2(Q) = 1 \text{ and } \text{cond}_2(R) = \text{cond}_2(A).$$

□

[2]The complex counterpart of this is the unitary matrix, which allows us to perform everything we do here in real numbers in the complex field.

Proof: Since Q is orthogonal, $Q^{-1} = Q^T$. For any vector $x \in \mathbb{R}^n$,

$$\|Qx\|_2^2 = \langle Qx, Qx\rangle_2 = \langle Q^T Qx, x\rangle_2 = \langle x, x\rangle_2 = \|x\|_2^2,$$

which implies $\|Qx\|_2 = \|x\|_2$. Therefore, for invertible matrices $B \in \mathbb{R}^{n\times n}$,

$$\|QB\|_2 = \max_{\|x\|_2=1} \|QBx\|_2 = \max_{\|x\|_2=1} \|Q(Bx)\|_2 = \max_{\|x\|_2=1} \|Bx\|_2 = \|B\|_2$$

and with $Qx = y$,

$$\|BQ\|_2 = \max_{\|x\|_2=1} \|BQx\|_2 = \max_{\|Q^T y\|_2=1} \|By\|_2 = \max_{\|y\|_2=1} \|By\|_2 = \|B\|_2,$$

since Q and $Q^T = Q^{-1}$ are both orthogonal. Hence,

$$\mathrm{cond}_2(Q) = \|Q\|_2\|Q^{-1}\|_2 = \|Q\mathrm{Id}\|_2\|Q^{-1}\mathrm{Id}\|_2 = \|\mathrm{Id}\|_2\|\mathrm{Id}\|_2 = 1$$

and

$$\begin{aligned}\mathrm{cond}_2(R) &= \mathrm{cond}_2(Q^T A) = \|Q^T A\|_2\|(Q^T A)^{-1}\|_2 = \|Q^T A\|_2\|A^{-1}Q\|_2\\ &= \|A\|_2\|A^{-1}\|_2 = \mathrm{cond}_2(A).\end{aligned}$$

□

While this theorem does not hold for other matrix norms, the estimate $\|x\|_p \le C_{p,q}\|x\|_q$ that is valid for any two vector norms ensures that at least there is no extreme degradation of the numerical condition concerning other induced matrix norms.

Example 2.18 As an example we reconsider the equation system from Example 2.15. A QR factorization of A is given by

$$Q = \frac{1}{\eta}\begin{pmatrix} -0.001 & -1 \\ -1 & 0.001 \end{pmatrix} \quad \text{and} \quad R = \begin{pmatrix} -\eta & -\eta - \frac{1}{\eta} \\ 0 & \frac{0.001}{\eta} \end{pmatrix} \approx \begin{pmatrix} -1 & -2 \\ 0 & 0.001 \end{pmatrix}$$

with

$$Q^{-1} = \frac{1}{\eta}\begin{pmatrix} -0.001 & -1 \\ -1 & 0.001 \end{pmatrix} \quad \text{and} \quad R^{-1} = \begin{pmatrix} -\frac{1}{\eta} & -\frac{4000}{\eta+\frac{1}{\eta}} \\ 0 & 1000\eta \end{pmatrix} \approx \begin{pmatrix} -1 & -2000 \\ 0 & 1000 \end{pmatrix}.$$

So, we have $\mathrm{cond}_\infty(Q) \approx 1$ and $\mathrm{cond}_\infty(R) \approx 6000$, which means there is no significant deterioration of the condition number in the ∞-norm. □

The idea of the algorithms for QR factorization is to consider the columns of the matrix A as vectors and transform them into the desired form through orthogonal transformations. Orthogonal transformations are precisely the linear transformations that can be represented by orthogonal matrices. Geometrically, these transformations are the ones that preserve the (Euclidean) length of the transformed vector as well as the angle between two vectors — nothing else was exploited in the proof of Theorem 2.17.

To implement such an algorithm, two possible transformations are available: rotations and reflections. Here, we will derive the *Householder algorithm*, named after its inventor, which is based on reflections[3]. We will first illustrate the idea geometrically.

Let $a_{\cdot j} \in \mathbb{R}^n$ be the j-th column of the matrix A. We want to find a reflection $H^{(j)}$ that maps $a_{\cdot j}$ to a vector of the form $a_{\cdot j}^{(j)} = (\underbrace{*, *, \ldots, *}_{j \text{ positions}}, 0)^T$. In other words, the vector should be reflected onto the plane $E_j = \text{span}(e_1, \ldots, e_j)$.

To construct this reflection, we consider general reflection matrices of the form

$$H = H(v) = \text{Id} - \frac{2vv^T}{v^T v}$$

where $v \in \mathbb{R}^n$ is an arbitrary vector (note that $vv^T \in \mathbb{R}^{n \times n}$ and $v^T v \in \mathbb{R}$). These matrices are called *Householder matrices*. Clearly, H is symmetric, and it satisfies

$$HH^T = H^2 = \text{Id} - \frac{4vv^T}{v^T v} + \frac{2vv^T}{v^T v}\frac{2vv^T}{v^T v} = \text{Id},$$

which means it is orthogonal. Geometrically, multiplying by H corresponds to reflection about the plane with the normal vector $n = v/\|v\|$.

To realize the desired reflection into the plane E_j, we must choose v appropriately. The following lemma helps with this.

Lemma 2.19 Consider a vector $w = (w_1, \ldots, w_n)^T \in \mathbb{R}^n$. For a given index $j \in \{1, \ldots, n\}$, consider

$$\begin{aligned} c &= \text{sgn}(w_j)\sqrt{w_j^2 + w_{j+1}^2 + \cdots + w_n^2} \in \mathbb{R} \\ v &= (0, \ldots, 0, c + w_j, w_{j+1}, \ldots, w_n)^T \\ H &= \text{Id} - \frac{2vv^T}{v^T v} \end{aligned}$$

with the conventions $\text{sgn}(a) = 1$ if $a \geq 0$, $\text{sgn}(a) = -1$ if $a < 0$, and $H = \text{Id}$ if $v = 0$. Then,

$$Hw = (w_1, w_2, \ldots, w_{j-1}, -c, 0, \ldots, 0)^T.$$

Furthermore, for any vector $z \in \mathbb{R}^n$ of the form $z = (z_1, \ldots, z_{j-1}, 0, \ldots, 0)^T$, we have $Hz = z$.

Proof: If $v \neq 0$, then

$$\begin{aligned} \frac{2v^T w}{v^T v} &= \frac{2((c + w_j)w_j + w_{j+1}^2 + \cdots + w_n^2)}{c^2 + 2cw_j + w_j^2 + w_{j+1}^2 + \cdots + w_n^2} \\ &= \frac{2(cw_j + w_j^2 + w_{j+1}^2 + \cdots + w_n^2)}{2cw_j + 2w_j^2 + 2w_{j+1}^2 + \cdots + 2w_n^2} = 1. \end{aligned}$$

[3] An algorithm based on rotations is the so-called Givens algorithm

From this, it follows that

$$Hw = w - v\frac{2v^T w}{v^T v} = w - v = \begin{pmatrix} w_1 \\ \vdots \\ w_{j-1} \\ w_j \\ w_{j+1} \\ \vdots \\ w_n \end{pmatrix} - \begin{pmatrix} 0 \\ \vdots \\ 0 \\ c + w_j \\ w_{j+1} \\ \vdots \\ w_n \end{pmatrix} = \begin{pmatrix} w_1 \\ \vdots \\ w_{j-1} \\ -c \\ 0 \\ \vdots \\ 0 \end{pmatrix}.$$

If $v = 0$, it is immediately seen that $w_{j+1} = \ldots = w_n = 0$ must hold. Thus, $c = w_j$, so $w_j + c = 2w_j$, which implies $w_j = 0$. All in all, this yields $w_j = w_{j+1} = \ldots = w_n = 0$ and thus the claim holds with $H = \mathrm{Id}$.

For the second claim, we use the fact that for vectors z of the given form, we have $v^T z = 0$, from which we immediately get $Hz = z$, and the claim follows. □

The idea of the algorithm now becomes apparent:

We set $A^{(1)} = A$ and construct $H^{(1)}$ in the first step according to Lemma 2.19 with $j = 1$ and $w = a^{(1)}_{\cdot 1}$, the first column of the matrix $A^{(1)}$. Thus, $A^{(2)} = H^{(1)}A^{(1)}$ is of the form

$$A^{(2)} = H^{(1)}A^{(1)} = \begin{pmatrix} a^{(2)}_{1\,1} & a^{(2)}_{1\,2} & \cdots & a^{(2)}_{1\,n} \\ 0 & a^{(2)}_{2\,2} & \cdots & a^{(2)}_{2\,n} \\ \vdots & \vdots & & \vdots \\ 0 & a^{(2)}_{n\,2} & \cdots & a^{(2)}_{n\,n} \end{pmatrix}.$$

In the second step, we construct $H^{(2)}$ according to Lemma 2.19 with $j = 2$ and $w = a^{(2)}_{\cdot 2}$, the second column of matrix $A^{(2)}$. Since the first column of matrix $A^{(2)}$ satisfies the conditions for the vector z in Lemma 2.19, the form follows:

$$A^{(3)} = H^{(2)}A^{(2)} = \begin{pmatrix} a^{(3)}_{1\,1} & a^{(3)}_{1\,2} & a^{(3)}_{1\,3} & \cdots & a^{(3)}_{1\,n} \\ 0 & a^{(3)}_{2\,2} & a^{(3)}_{2\,3} & \cdots & a^{(3)}_{2\,n} \\ 0 & 0 & a^{(3)}_{3\,3} & \cdots & a^{(3)}_{3\,n} \\ \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & a^{(3)}_{n\,3} & \cdots & a^{(3)}_{n\,n} \end{pmatrix}.$$

Continuing this process successively for $n-1$ steps yields the desired QR factorization with

$$Q^T = H^{(n-1)} \cdots H^{(1)} \text{ and } R = A^{(n)},$$

as it holds

$$\begin{aligned} QR &= {H^{(1)}}^T \cdots {H^{(n-1)}}^T A^{(n)} \\ &= {H^{(1)}}^T \cdots {H^{(n-2)}}^T A^{(n-1)} \\ &\vdots \\ &= {H^{(1)}}^T A^{(2)} \\ &= A^{(1)} = A \end{aligned}$$

Note that the QR factorization algorithm always works, even if A is not invertible, as the above considerations provide a constructive proof of its existence. The resulting matrix Q is always invertible, and the matrix R is invertible if and only if A is invertible.

In the following implementation of this algorithm, we do not explicitly calculate the matrix Q^T but only store the vectors $v^{(j)}$. This is sufficient to reconstruct the application of Q^T. In addition to the matrix R, we also calculate the vector $y = Q^T b$ in the algorithm. The matrix R is stored in the upper triangular part of matrix A, so the 0 elements are not explicitly stored. Instead, we store the entries $j+1, \dots, n$ of vector $v^{(j)}$ in A. The remaining j-th entry of $v^{(j)}$ is stored in a vector v. The multiplication $H^{(j)}w$ is performed in the form

$$d = \frac{2}{v^T v}, \quad e = dv^T w, \quad H^{(j)}w = w - ev$$

Algorithm 2.20 (QR Factorization using Householder Algorithm)
Input: Matrix $A = (a_{ij})$, vector $b = (b_i)$

(0) for i from 1 to n
set $v_i := 0$

End of i loop

(1) for j from 1 to $n-1$
set $c := \operatorname{sgn}(a_{jj}) \sqrt{\sum_{i=j}^{n} a_{i,j}^2}$
if $c = 0$, continue with $j+1$; otherwise
set $a_{jj} := c + a_{jj}$ *(the entries v_i, $i \geq j$ are now in a_{ij}, $i \geq j$)*
set $v_j := a_{j,j}$

set $d := 1/(ca_{jj})$

Computation of $H^{(j)}b^{(j)}$:
set $e := d\left(\sum_{i=j}^{n} a_{ij} b_i\right)$
for i from j to n set $b_i := b_i - e a_{ij}$
End of i loop

Computation of $H^{(j)}A^{(j)}$ for columns $j+1, \dots, n$:
(the j-th column is still needed to store v_j)
for k from $j+1$ to n
set $e := d\left(\sum_{i=j}^{n} a_{ij} a_{ik}\right)$
for i from j to n set $a_{ik} := a_{ik} - e a_{ij}$
End of i and k loops

Computation of the j-th column of $H^{(j)}A^{(j)}$:
(only the diagonal element changes, see Lemma 2.19)
set $a_{jj} = -c$
End of j loop

Output: $R = (a_{ij})_{i \leq j}$, $v^{(j)} = \{v_j, (a_{ij})_{i>j}\}$, $Q^T b = b^{(n)} = (b_i)$. □

When solving a linear system of equations, the invertibility of R should be tested before back-substitution (an upper triangular matrix is invertible if and only if all diagonal elements are nonzero). This can be done within the algorithm by checking if the c values (which form the diagonal elements of R) are nonzero.

The QR factorization can do more than just solve linear systems of equations. In Section 2.1, we learned about the linear least squares problem, where we seek a vector $x \in \mathbb{R}^n$ to minimize the 2-norm of the vector:

$$r = \widetilde{A}x - z$$

We have seen that this problem can be solved by solving the normal equations $\widetilde{A}^T\widetilde{A}x = \widetilde{A}^T z$. However, the matrix $\widetilde{A}^T\widetilde{A}$ can have a high condition number, making it desirable to use a robust method and avoid explicitly solving the normal equations. With the QR algorithm, both of these goals can be achieved, allowing us to directly solve the least squares problem.

The QR factorization (and the algorithm provided) can also be applied to non-square matrices $\widetilde{A}$ with n columns and $m > n$ rows. In this case, you modify the algorithm by allowing j to range from 1 to n and all other indices, except for k, to range from 1 to m in the computation of $H^{(j)}A^{(j)}$ up to m. The result is a factorization $\widetilde{A} = QR$ with

$$R = \begin{pmatrix} R_1 \\ 0 \end{pmatrix}$$

where $R_1 \in \mathbb{R}^{n\times n}$ is an upper triangular matrix. Note that the normal equations are solvable if and only if $\widetilde{A}$ has full column rank, which we assume. In this case, R_1 is also invertible.

If we choose x such that the vector

$$s = Q^T r = Q^T\widetilde{A}x - Q^T z$$

has the minimal 2-norm, then r will also have the minimal 2-norm because the orthogonality of Q^T implies $\|r\|_2 = \|s\|_2$. Because of the form of $R = Q^T\widetilde{A}$ the vector s^2 is independent of x and since

$$\|s\|_2^2 = \sum_{i=1}^{m} s_i^2 = \sum_{i=1}^{n} s_i^2 + \sum_{i=n+1}^{m} s_i^2 = \|s^1\|_2^2 + \|s^2\|_2^2,$$

this norm becomes mininal if and only of the norm $\|s^1\|_2^2$ becomes minimal. We thus look for an $x \in \mathbb{R}^n$ that mininizes

$$\|s^1\|_2 = \|R_1 x - y^1\|_2,$$

where y^1 denotes the first n components of the vector $y = Q^T z$. Since R_1 is an invertible upper triangular matrix, by backward substitution we can find a solution x of the system of equations $R_1 x = y^1$, for which

$$\|s^1\|_2 = \|R_1 x - y^1\|_2 = 0$$

holds and which thus obviously realized the minimum. In summary, we can directly solve the least squares problem with the QR factorization as follows:

Algorithm 2.21 (Solving the Least Squares Problem with QR Factorization)
Input: Matrix $\widetilde{A} \in \mathbb{R}^{m \times n}$ with $m > n$ and maximal column rank n, vector $z \in \mathbb{R}^m$

(1) Compute the factorization $\widetilde{A} = QR$ with $R = \begin{pmatrix} R_1 \\ 0 \end{pmatrix}$ and upper triangular matrix $R_1 \in \mathbb{R}^{n \times n}$

(2) Solve the equation system $R_1 x = y^{(1)}$ through backward substitution, where $y^{(1)}$ denotes the first n components of the vector $y = Q^T z \in \mathbb{R}^m$

Output: Vector $x \in \mathbb{R}^n$ that minimizes $\|\widetilde{A}x - z\|_2$. □

Geometrically, this algorithm projects the image of $\widetilde{A}$ onto the subspace spanned by $(e_1, \ldots, e_n)$ through the orthogonal transformation Q^T. Then, you can solve the resulting equation system in that subspace, as shown in Figure 2.1.

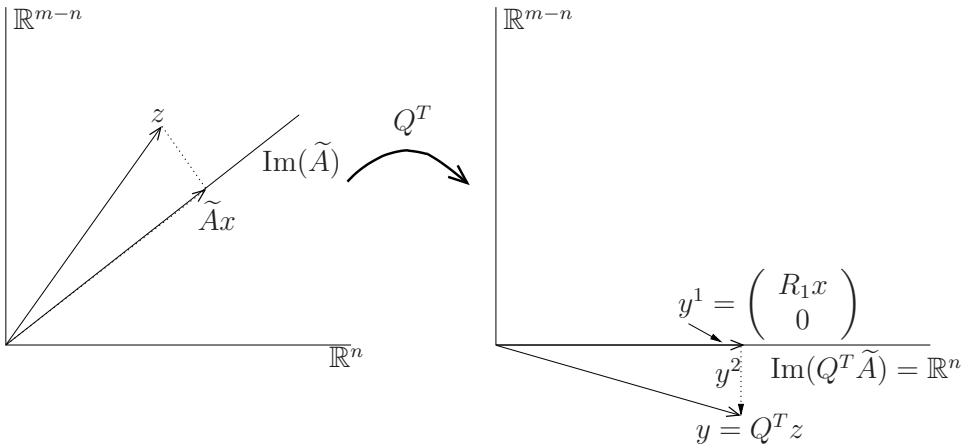


Figure 2.1: Illustration of Algorithm 2.21

2.7 Computational Effort

An important aspect in the analysis of numerical methods is to examine how long these methods typically take to achieve the desired result. Since this crucially depends on the performance of the computer used, one does not directly estimate the time but rather the number of arithmetic operations that an algorithm requires. In this context, the *floating-point operations*, such as addition, multiplication etc. of real numbers are typically the focus of the analysis, because they are by far the most time-consuming operations[4].

The methods we have considered so far provide a result after a finite number of steps (referred to as *direct methods*), where the number of operations depends on the dimension n of the matrix. To estimate the *computational effort*, it is sufficient to count the number

[4]In reality, multiplication, division, and square root calculations are somewhat more expensive than addition and subtraction, but we will neglect this here.

of floating-point operations (in terms of n). The most efficient way to do this depends on the algorithm's structure. Additionally, some basic calculus rules need to be applied to simplify the resulting expressions. Specifically, we need the equations

$$\sum_{i=1}^{n} i = \frac{(n+1)n}{2} \quad \text{and} \quad \sum_{i=1}^{n} i^2 = \frac{1}{3}n^3 + \frac{1}{2}n^2 + \frac{1}{6}n.$$

Let us start with the **backward substitution** and consider the multiplications and divisions: For $i = n$, one division is required; for $i = n-1$, one multiplication and one division are needed, and so on. This results in the number of these operations as

$$1 + 2 + 3 + \cdots + n = \sum_{i=1}^{n} i = \frac{(n+1)n}{2} = \frac{n^2}{2} + \frac{n}{2}.$$

For the number of additions and subtractions, we count

$$0 + 1 + 2 + \cdots + n - 1 = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2}.$$

So, the total number of operations is

$$\frac{n^2}{2} + \frac{n}{2} + \frac{n^2}{2} - \frac{n}{2} = n^2$$

floating-point operations. Since **forward substitution** works completely analogously, it requires the same number of operations.

For the **Gaussian elimination**, we consider the version without pivot search as shown in Algorithm 2.3, assuming the worst case scenario where Step (2) is performed every time. We proceed column by column, examining the elements that need elimination for each j. For each element in the j-th column that requires elimination, we need $2(n+2-j)+1$ operations (including the operation for b) to perform the required additions, multiplications, and divisions. In the j-th column, there are $n-j$ entries that need elimination for $i = n, \ldots, j+1$. Thus, for the j-th column, we have

$$(n-j)(2(n+2-j)+1) = 2n^2+4n-2nj-2jn-4j+2j^2+n-j = 2j^2-(4n+5)j+5n+2n^2$$

operations. This needs to be done for columns $j = 1, \ldots, n-1$, resulting in

$$\begin{aligned}
&\sum_{j=1}^{n-1} \left(2j^2 - (4n+5)j + 5n + 2n^2\right) \\
&= 2\sum_{j=1}^{n-1} j^2 - (4n+5)\sum_{j=1}^{n-1} j + \sum_{j=1}^{n-1} \left(5n + 2n^2\right) \\
&= \frac{2}{3}(n-1)^3 + (n-1)^2 + \frac{1}{3}(n-1) - (4n+5)\frac{(n-1)n}{2} + (n-1)(5n+2n^2) \\
&= \frac{2}{3}n^3 + \frac{3}{2}n^2 - \frac{13}{6}n
\end{aligned}$$

operations.

For the **Choleski method** we can count as follows: For each i and each $j < i$ we need to perform $j - 1$ multiplications and additions, and one Division, which alltogether makes

$$\sum_{j=1}^{i-1}(2(j-1)+1) = 2\sum_{j=1}^{i-1} j + \sum_{j=1}^{i-1}(-1) = i(i-1) - (i-1) = i^2 - 2i + 1$$

operations (note that this formula is also valid for $i = 1$). For $i = j$ we obtain $i-1$ additios and multiplications (for squaring the $l_{j\,j}$) and one square root operation, which yields a total of $2(i-1)+1$ operations. All in all for each i we thus have

$$i^2 - 2i + 1 + 2(i-1) + 1 = i^2 - 2i + 1 + 2i - 2 + 1 = i^2$$

operations, which results in the total number of operations

$$\sum_{i=1}^{n} i^2 = \frac{1}{3}n^3 + \frac{1}{2}n^2 + \frac{1}{6}n.$$

For the QR **factorization using Householder reflections**, we consider only the computation of R and $y = Q^T b$. For each $j = 1, \ldots, n-1$, we need to compute c (requiring $2(n-j+1)$ operations, with the calculation of 'sgn' being negligibly fast), $a_{j\,j}$, and d (additional 3 operations). For the computation of y, we first calculate e ($2(n-j+1)$ operations) and then y ($2(n-j+1)$ operations again), resulting in a total of $4(n-j+1)$ operations. For computing R, these calculations need to be done $(n-j)$ times, resulting in $(n-j)4(n-j+1) = 4n^2 + 4j^2 - 8nj + 4n - 4j$ operations for each j. In total, we have

$$\begin{aligned}&2(n-j+1) + 3 + 4(n-j+1) + 4n^2 + 4j^2 - 8nj + 4n - 4j\\ &= 4j^2 - (8n+10)j + 4n^2 + 10n + 9\end{aligned}$$

operations for each j, and thus the sum over all j gives

$$\begin{aligned}&\sum_{j=1}^{n-1}(4j^2 - (8n+10)j + 4n^2 + 10n + 9)\\ &= \frac{4}{3}(n-1)^3 + 2(n-1)^2 + \frac{2}{3}(n-1)\\ &\qquad - (8n+10)\frac{n(n-1)}{2} + 4n^2(n-1) + 10n(n-1) + 9(n-1)\\ &= \frac{4}{3}n^3 + \frac{14}{3}n - 9\end{aligned}$$

operations.

For the complete **solution of a system of linear equations**, we simply add up the operations of the sub-algorithms.

For the Gaussian algorithm, we have:

$$\frac{2}{3}n^3 + \frac{5}{2}n^2 - \frac{13}{6}n + n^2 = \frac{2}{3}n^3 + \frac{5}{2}n^2 - \frac{13}{6}n$$

operations. For the Cholesky factorization, we get:

$$\frac{1}{3}n^3 + \frac{5}{2}n^2 + \frac{1}{6}n + 2n^2 = \frac{1}{3}n^3 + \frac{5}{2}n^2 + \frac{1}{6}n$$

operations, and for the QR factorization:

$$\frac{4}{3}n^3 + 4n^2 + \frac{14}{3}n - 9 + n^2 = \frac{4}{3}n^3 + 4n^2 + \frac{14}{3}n - 9$$

operations. Considering that for large n, the leading n^3 terms dominate, we can conclude that the Cholesky factorization is approximately twice as fast as Gaussian elimination, and Gaussian elimination is roughly twice as fast as QR factorization.

To get an impression of the actual computation times, let's assume that we are using an Apple PC with an M2 processor, which achieves a performance of approximately 890 GFLOPS (FLOPS = floating-point operations per second; GFLOPS = GigaFLOPS = 10^9 Flops). Furthermore, let's assume (very optimistically) that we have implementations of the above algorithms that utilize this performance optimally. Then, for $n \times n$ systems of equations, the following computation times arise:

n	Cholesky	Gauss	QR
1000	0.38 ms	0.78 ms	1.50 ms
10000	0.37 s	0.78 s	1.50 s
100000	6.25 min	12.48 min	24.96 min
500000	0.54 d	1.08 d	2.16 d

It is evident that in the last case $n = 500,000$, the times are hardly acceptable for practical purposes.

When calculating the computational effort, it must be taken into account that the effort decreases when the matrix A has many zero entries that are "favorably" distributed. What is "favorable" depends strongly on the algorithm. For example, Gaussian elimination is very efficient for banded matrices where only the diagonal and some off-diagonal entries are nonzero. The Cholesky algorithm is efficient for so-called skyline matrices, which are banded matrices where the non-zero entries are compactly stored.

To conclude this section, we want to introduce a broader concept of effort estimation, which is often sufficient for practical purposes. Often, one is not interested in the exact number of operations for a given n but only in an estimation for large dimensions. Specifically, one wants to know how quickly the effort grows depending on n, i.e., how it behaves *asymptotically* as $n \to \infty$. This is referred to as the *order* of an algorithm.

Definition 2.22 An algorithm has the *order* $O(n^q)$ if $q > 0$ is the minimum number for which there exists a constant $C > 0$ such that the algorithm requires fewer than Cn^q operations for all $n \in \mathbb{N}$. □

This number q can be easily read from the above effort calculations: it is precisely the highest power of n that appears. Thus, forward and backward substitution have order $O(n^2)$, while Gaussian, Cholesky, and QR methods have order $O(n^3)$.

2.8 Iterative Methods

In the previous section, we saw that the methods considered so far — called *direct methods* — have order $O(n^3)$: if n is increased by a factor of ten, the number of operations and thus the computation time increases by a factor of a thousand, as seen above. For large systems of equations with several hundred thousand unknowns, which do occur in practice, this leads to unacceptable computation times.

A class of methods that has lower order is the class of *iterative methods*. However, there is a trade-off: with these methods, we cannot expect to obtain an exact solution (up to rounding errors), but we must accept a certain level of inaccuracy in the result from the outset.

The basic principle of iterative methods works as follows:

Starting from an initial vector $x^{(0)}$, a sequence of vectors

$$x^{(i+1)} = \Phi(x^{(i)}), \quad i = 0, 1, 2, \ldots,$$

is calculated iteratively using a computational rule $\Phi : \mathbb{R}^n \to \mathbb{R}^n$. This sequence converges to the solution x^* of the system of equations $Ax = b$ as $i \to \infty$, i.e. $\lim_{i\to\infty} \|x^{(i)} - x^*\|_p = 0$. When the desired accuracy is reached, the iteration is terminated, and the last value $x^{(i)}$ is used as an approximation of the result.

Before we look at specific examples of such methods, let us first review a theorem from analysis that is helpful for the analysis of iterative methods.

Theorem 2.23 (Banach's Fixed-Point Theorem) Let A be a closed subset of a complete normed space with norm $\|\cdot\|$ and let $\Phi : A \to A$ be a contraction, meaning that there exists a constant $k \in (0, 1)$ such that the inequality

$$\|\Phi(x) - \Phi(y)\| \leq k\|x - y\|$$

holds for all $x, y \in A$. Then there exists a unique fixed point $x^* \in A$ to which all sequences of the form $x^{(i+1)} = \Phi(x^{(i)})$ with any $x^{(0)} \in A$ converge. Furthermore, the *a priori* and *a posteriori* estimates

$$\|x^{(i)} - x^*\| \leq \frac{k^i}{1-k}\|x^{(1)} - x^{(0)}\|$$

and

$$\|x^{(i)} - x^*\| \leq \frac{k}{1-k}\|x^{(i)} - x^{(i-1)}\|$$

hold. □

Proof: First, we show that any sequence $(x^{(i)})_{i\in\mathbb{N}_0}$ of the form $x^{(i+1)} = \Phi(x^{(i)})$ with arbitrary $x^{(0)} \in A$ is a Cauchy sequence. Using induction, we can derive the following estimates for any $i, j \in \mathbb{N}_0$ with $j \geq i$:

$$\|x^{(j+1)} - x^{(j)}\| \leq k^{j-i}\|x^{(i+1)} - x^{(i)}\| \quad \text{and} \quad \|x^{(i+1)} - x^{(i)}\| \leq k^i\|x^{(1)} - x^{(0)}\| \tag{2.16}$$

From these inequalities, we obtain

$$\begin{aligned}
\|x^{(i+n)} - x^{(i)}\| &= \left\| \sum_{j=i}^{i+n-1} \left(x^{(j+1)} - x^{(j)} \right) \right\| \\
&\leq \sum_{j=i}^{i+n-1} \|x^{(j+1)} - x^{(j)}\| \leq \sum_{j=i}^{i+n-1} k^{j-i} \|x^{(i+1)} - x^{(i)}\| \\
&= \frac{1-k^n}{1-k} \|x^{(i+1)} - x^{(i)}\| \leq \frac{1}{1-k} \|x^{(i+1)} - x^{(i)}\| \\
&\leq \frac{k}{1-k} \|x^{(i)} - x^{(i-1)}\| \leq \frac{k^i}{1-k} \|x^{(1)} - x^{(0)}\|, \qquad (2.17)
\end{aligned}$$

Since $k^i \to 0$, this sequence is a Cauchy sequence.

Next, we show that $x^{(i)}$ converges to a fixed point of Φ. Since A is a subset of a complete space, the limit x^* of this Cauchy sequence exists and belongs to A, i.e., $\lim_{i\to\infty} x^{(i)} = x^* \in A$. Since Φ is a contraction, it is also continuous, so we have

$$\Phi(x^*) = \Phi(\lim_{i\to\infty} x^{(i)}) = \lim_{i\to\infty} \Phi(x^{(i)}) = \lim_{i\to\infty} x^{(i+1)} = x^*$$

This shows that x^* is a fixed point of Φ. Therefore, every sequence of the given form converges to a fixed point of Φ. Finally, we need to prove the uniqueness of the fixed point. Suppose there are two fixed points x^* and x^{**} of Φ with $x^* \neq x^{**}$, i.e., $\|x^* - x^{**}\| > 0$. From the contraction property, we have

$$\|x^* - x^{**}\| = \|\Phi(x^*) - \Phi(x^{**})\| \leq k\|x^* - x^{**}\| < \|x^* - x^{**}\|$$

This is a contradiction to $\|x^* - x^{**}\| > 0$, which proves the uniqueness.

Finally, we show the two estimates. Both follow from the inequality derived earlier:

$$\begin{aligned}
\|x^{(i)} - x^*\| = \lim_{n\to\infty} \|x^{(i+n)} - x^{(i)}\| &\leq \lim_{n\to\infty} \frac{1}{1-k} \|x^{(i+1)} - x^{(i)}\| \\
&= \frac{k}{1-k} \|x^{(i)} - x^{(i-1)}\| \leq \frac{k^i}{1-k} \|x^{(1)} - x^{(0)}\|.
\end{aligned}$$

□

2.9 Gauss-Seidel and Jacobi Methods

We now want to introduce two classic iterative methods that both work based on the same principle: The matrix A is decomposed into the difference of two matrices

$$A = M - N,$$

assuming that M is easy (i.e., with very little effort) to invert. Then, you choose an initial vector $x^{(0)}$ (e.g., the zero vector) and calculate iteratively

$$x^{(i+1)} = M^{-1}Nx^{(i)} + M^{-1}b, \quad i = 0, 1, 2, \ldots. \qquad (2.18)$$

If the decomposition (under suitable assumptions about A) is chosen appropriately, you can expect the vectors x_i to converge to the desired solution. This is precisely described in the following lemma.

Lemma 2.24 Consider the linear system of equations $Ax = b$ with an invertible matrix A and a decomposition $A = M - N$ with an invertible matrix M. Let $\|\cdot\|$ be a vector norm with the associated induced matrix norm such that the inequality $k = \|M^{-1}N\| < 1$ holds. Then, the iteration (2.18) converges to the solution x^* of the equation system for any initial value $x^{(0)}$, and the iteration mapping is a contraction for the norm $\|\cdot\|$ with contraction constant k. Moreover, the following estimates hold:

$$\|x^{(i)} - x^*\| \leq \frac{k}{1-k}\|x^{(i)} - x^{(i-1)}\| \leq \frac{k^i}{1-k}\|x^{(1)} - x^{(0)}\|.$$

Proof: First, we show that the mapping $\Phi : \mathbb{R}^n \to \mathbb{R}^n$ defined by $\Phi(x) = M^{-1}Nx + M^{-1}b$ is a contraction with respect to the induced matrix norm $\|\cdot\|$: It holds

$$\begin{aligned} \|\Phi(x) - \Phi(y)\| &= \|M^{-1}Nx + M^{-1}b - M^{-1}Ny - M^{-1}b\| \\ &= \|M^{-1}N(x-y)\| \leq \|M^{-1}N\|\,\|x-y\| = k\|x-y\|. \end{aligned}$$

Therefore, according to Banach's Fixed Point Theorem 2.23, the iteration (2.18) converges to a unique fixed point x^*, and the provided estimates hold.

Since

$$\begin{aligned} \Phi(x^*) = x^* &\Leftrightarrow M^{-1}Nx^* + M^{-1}b = x^* \\ &\Leftrightarrow Nx^* + b = Mx^* \\ &\Leftrightarrow b = (M-N)x^* = Ax^* \end{aligned}$$

this fixed point is indeed the solution of the equation system. □

In iterative algorithms, we need a termination criterion to decide when to stop the iteration. There are several possibilities; a simple but efficient criterion is to specify $\varepsilon > 0$ and stop the iteration when the condition

$$\|x^{(i+1)} - x^{(i)}\| < \varepsilon \tag{2.19}$$

is met for a given vector norm $\|\cdot\|$. When using the vector norm for which Lemma 2.24 holds, this criterion ensures the accuracy

$$\|x^{(i+1)} - x^*\| \leq \frac{k}{1-k}\varepsilon$$

is guaranteed. Here, you can also use the relative error

$$\frac{\|x^{(i+1)} - x^{(i)}\|}{\|x^{(i+1)}\|} < \varepsilon$$

if you want to iterate until achieving the maximum possible computational accuracy. In that case, set ε as the machine precision (typically 10^{-8} for single precision and 10^{-16} for double precision).

Example 2.25 We illustrate such a procedure with the three-dimensional linear system

$$A = \begin{pmatrix} 15 & 3 & 4 \\ 2 & 17 & 3 \\ 2 & 3 & 21 \end{pmatrix} \quad \text{and} \quad b = \begin{pmatrix} 33 \\ 45 \\ 71 \end{pmatrix}.$$

For the decomposition $A = M - N$, we choose

$$M = \begin{pmatrix} 15 & 0 & 0 \\ 0 & 17 & 0 \\ 0 & 0 & 21 \end{pmatrix} \quad \text{and } N = -\begin{pmatrix} 0 & 3 & 4 \\ 2 & 0 & 3 \\ 2 & 3 & 0 \end{pmatrix},$$

i.e., we decompose A into its diagonal part M and the non-diagonal part $-N$. Diagonal matrices are easy to invert: You simply replace each element with its reciprocal, so

$$M^{-1} = \begin{pmatrix} 1/15 & 0 & 0 \\ 0 & 1/17 & 0 \\ 0 & 0 & 1/21 \end{pmatrix}.$$

This gives us

$$M^{-1}N = \begin{pmatrix} 0 & -1/5 & -4/15 \\ -2/17 & 0 & -3/17 \\ -2/21 & -1/7 & 0 \end{pmatrix} \quad \text{and} \quad M^{-1}b = \begin{pmatrix} 11/5 \\ 45/17 \\ 71/21 \end{pmatrix}.$$

Now, we calculate the vectors $x^{(1)}, \ldots, x^{(10)}$ according to the rule (2.18), with $x^{(0)} = (0\,0\,0)^T$ as the initial value. The results are (rounded to four decimal places for each entry):

$$\begin{pmatrix} 2.2000 \\ 2.6471 \\ 3.3810 \end{pmatrix}, \begin{pmatrix} 0.7690 \\ 1.7916 \\ 2.7933 \end{pmatrix}, \begin{pmatrix} 1.0968 \\ 2.0637 \\ 3.0518 \end{pmatrix}, \begin{pmatrix} 0.9735 \\ 1.9795 \\ 2.9817 \end{pmatrix}, \begin{pmatrix} 1.0090 \\ 2.0064 \\ 3.0055 \end{pmatrix},$$

$$\begin{pmatrix} 0.9973 \\ 1.9980 \\ 2.9982 \end{pmatrix}, \begin{pmatrix} 1.0009 \\ 2.0006 \\ 3.0005 \end{pmatrix}, \begin{pmatrix} 0.9997 \\ 1.9998 \\ 2.9998 \end{pmatrix}, \begin{pmatrix} 1.0001 \\ 2.0001 \\ 3.0001 \end{pmatrix}, \begin{pmatrix} 1.0000 \\ 2.0000 \\ 3.0000 \end{pmatrix}.$$

□

Depending on the choice of M and N, you obtain different methods. Here, we want to describe two methods in more detail and write out the iteration (2.18) explicitly for the entries $x_j^{(i+1)}$ of the vectors $x^{(i+1)}$, making the methods directly implementable. The first method is the one we used in Example 2.25.

Algorithm 2.26 (Jacobi method or **Total step method)**
We choose $M = M_J$ as a diagonal matrix

$$M_J = \begin{pmatrix} a_{1\,1} & 0 & \ldots & 0 \\ 0 & a_{2\,2} & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \ldots & 0 & a_{n\,n} \end{pmatrix}$$

and $N = N_J$ as $N_J = M_J - A$. Then, iteration (2.18) becomes

$$x_j^{(i+1)} = \frac{1}{a_{j\,j}} \left(b_j - \sum_{\substack{k=1 \\ k \neq j}}^{n} a_{j\,k} x_k^{(i)} \right), \quad \text{for } j = 1, \ldots, n.$$

□

A slightly different decomposition leads to the following method.

Algorithm 2.27 (Gauss-Seidel method or **Single step method)**
We choose $M = M_{GS}$ as a lower triangular matrix

$$M_{GS} = \begin{pmatrix} a_{1\,1} & 0 & \ldots & 0 \\ a_{2\,1} & a_{2\,2} & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ a_{n\,1} & \ldots & a_{n\,n-1} & a_{n\,n} \end{pmatrix}$$

and $N = N_{GS}$ as $N_{GS} = M_{GS} - A$. When we multiply iteration (2.18) from the left by M_{GS}, we get

$$M_{GS} x^{(i+1)} = N_{GS} x^{(i)} + b.$$

Now, we can determine the components $x_j^{(i+1)}$ by forward substitution, yielding

$$x_j^{(i+1)} = \frac{1}{a_{j\,j}} \left(b_j - \sum_{k=1}^{j-1} a_{j\,k} x_k^{(i+1)} - \sum_{k=j+1}^{n} a_{j\,k} x_k^{(i)} \right), \quad \text{for } j = 1, \ldots, n.$$

□

The following theorem provides a criterion under which these methods converge.

Theorem 2.28 Let A be a *(strictly) diagonally dominant* matrix, i.e., the inequality

$$|a_{i\,i}| > \sum_{\substack{j=1 \\ j \neq i}}^{n} |a_{i\,j}|$$

holds for all $i = 1, \ldots, n$.

Then, the assumption of Lemma 2.24 is satisfied for the Jacobi and Gauss-Seidel methods with the row sum norm $\| \cdot \|_\infty$, and the constants $k = k_J$ and $k = k_{GS}$ for both methods can be estimated by

$$k_{GS} \leq k_J = \max_{i=1,\ldots,n} \left(\sum_{\substack{j=1 \\ j \neq i}}^{n} \frac{|a_{i\,j}|}{|a_{i\,i}|} \right) < 1$$

In particular, both methods converge for all initial values to the solution of the equation system, and the estimates from Lemma 2.24 hold. □

Proof: It suffices to show the estimates $k = \|M^{-1}N\|_\infty < 1$ for both methods.

We start with the Jacobi method. Here, we have $M = M_J = \operatorname{diag}(a_{1\,1}, \ldots, a_{n\,n})$ and $N = N_J = M_J - A$. Since $M_J^{-1} = \operatorname{diag}(a_{1\,1}^{-1}, \ldots, a_{n\,n}^{-1})$, we have

$$M_J^{-1}N_J = \begin{pmatrix} 0 & -\frac{a_{1\,2}}{a_{1\,1}} & \cdots & -\frac{a_{1\,n}}{a_{1\,1}} \\ -\frac{a_{2\,1}}{a_{2\,2}} & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & -\frac{a_{n-1\,n}}{a_{n-1\,n-1}} \\ -\frac{a_{n\,1}}{a_{n\,n}} & \cdots & -\frac{a_{n\,n-1}}{a_{n\,n}} & 0 \end{pmatrix},$$

thus

$$\|M_J^{-1}N_J\|_\infty = \max_{i=1,\ldots,n} \sum_{\substack{j=1 \\ j\neq i}}^{n} \frac{|a_{i\,j}|}{|a_{i\,i}|}.$$

Due to strict diagonal dominance, this sum is strictly less than 1 for all i, so we have $k_J := \|M_J^{-1}N_J\|_\infty < 1$.

For the Gauss-Seidel method, let $M = M_{GS}$ and $N = N_{GS}$ be given from Algorithm 2.27. We set $k_{GS} := \|M_{GS}^{-1}N_{GS}\|_\infty$ and prove the desired estimate $k_{GS} < 1$ by showing $k_{GS} \leq k_J$.

To prove this inequality, let $x \in \mathbb{R}^n$ be any vector with $\|x\|_\infty = 1$ and $y = M_{GS}^{-1}N_{GS}x \in \mathbb{R}^n$, so $M_{GS}y = N_{GS}x$. We need to show that $\|y\|_\infty \leq k_J$. We show the estimate individually for the entries $|y_i|$ of y by induction on $i = 1, \ldots, n$:

For $i = 1$, we have from $N_{GS}x = M_{GS}y = a_{1\,1}y_1$:

$$\begin{aligned} |y_1| &= \left|\frac{1}{a_{1\,1}}[N_{GS}x]_1\right| \leq \frac{1}{|a_{1\,1}|}\sum_{j=2}^{n}|a_{1\,j}||x_j| \leq \sum_{j=2}^{n}\frac{|a_{1\,j}|}{|a_{1\,1}|}\underbrace{\|x\|_\infty}_{=1} \\ &= \sum_{j=2}^{n}\frac{|a_{1\,j}|}{|a_{1\,1}|} \leq \max_{q=1,\ldots,n}\sum_{\substack{j=1\\j\neq q}}^{n}\frac{|a_{q\,j}|}{|a_{q\,q}|} = \|M_J^{-1}N_J\|_\infty = k_J. \end{aligned}$$

For the induction step $i-1 \to i$, assuming $|y_j| \leq k_J$ for $j = 1, \ldots, i-1$, because of

$$[N_{GS}x]_i = [M_{GS}y]_i = \sum_{j=1}^{i} a_{i\,j}y_j = \sum_{j=1}^{i-1} a_{i\,j}y_j + a_{i\,i}y_i$$

we obtain

$$\begin{aligned} |y_i| &\leq \frac{1}{|a_{i\,i}|}\left(\sum_{j=1}^{i-1}|a_{i\,j}||y_j| + |[N_{GS}x]_i|\right) \leq \frac{1}{|a_{i\,i}|}\left(\sum_{j=1}^{i-1}|a_{i\,j}||y_j| + \sum_{j=i+1}^{n}|a_{i\,j}||x_j|\right) \\ &= \sum_{j=1}^{i-1}\frac{|a_{i\,j}|}{|a_{i\,i}|}|y_j| + \sum_{j=i+1}^{n}\frac{|a_{i\,j}|}{|a_{i\,i}|}|x_j| \leq \sum_{j=1}^{i-1}\frac{|a_{i\,j}|}{|a_{i\,i}|}\underbrace{k_J}_{<1} + \sum_{j=i+1}^{n}\frac{|a_{i\,j}|}{|a_{i\,i}|}\underbrace{\|x\|_\infty}_{=1} \\ &\leq \left(\sum_{j=1}^{i-1}\frac{|a_{i\,j}|}{|a_{i\,i}|} + \sum_{j=i+1}^{n}\frac{|a_{i\,j}|}{|a_{i\,i}|}\right) \leq \max_{q=1,\ldots,n}\sum_{\substack{j=1\\j\neq q}}^{n}\frac{|a_{q\,j}|}{|a_{q\,q}|} \\ &= \|M_J^{-1}N_J\|_\infty = k_J. \end{aligned}$$

□

The proof shows, in particular, that the contraction constant k_{GS} of the Gauss-Seidel method is less than or equal to that of the Jacobi method k_J, and in fact, the Gauss-Seidel method is often significantly faster in practice.

Strict diagonal dominance is a rather specific criterion but is often satisfied in practice when discretizing differential equations.

For the Gauss-Seidel method, we can provide another condition under which this method converges. For this, we first need another preparatory lemma that shows another condition on $M^{-1}N$ for the convergence of the methods.

Lemma 2.29 Consider the linear equation system $Ax = b$ with an invertible matrix A and a decomposition $A = M - N$ with an invertible matrix M. Let $\rho(E) := \max_i |\lambda_i(E)|$ denote the spectral radius, i.e., the maximum absolute value of the eigenvalues $\lambda_1(E), \dots, \lambda_d(E)$ of a matrix $E \in \mathbb{R}^{n\times n}$. Then, the iteration (2.18) converges to the solution x^* of the equation system for any initial values $x^{(0)}$, provided that $\rho(M^{-1}N) < 1$.

Proof: First, we prove the following property for arbitrary matrices $E \in \mathbb{R}^{n\times n}$: For any $\epsilon \in (0, 1)$, there exists a vector norm $\|\cdot\|_{E,\epsilon}$ such that the induced matrix norm satisfies the inequality

$$\|E\|_{E,\epsilon} \le \rho(E) + \epsilon \tag{2.20}$$

To prove this, we need to use the well-known Jordan Normal Form from Linear Algebra.

For any matrix $E \in \mathbb{R}^{n\times n}$, there exists an invertible matrix $S \in \mathbb{C}^{n\times n}$ such that $R = S^{-1}ES$ is in Jordan Normal Form. The matrix R has the eigenvalues of E as its diagonal elements, denoted as $r_{i\,i}$, and the elements above the diagonal satisfy $r_{i,\,i+1} \in \{0, 1\}$, while all other entries are zero.

To construct the norm $\|\cdot\|_{E,\epsilon}$, note that for any invertible matrix $C \in \mathbb{C}^{n\times n}$ the norm $\|x\|_C := \|C^{-1}x\|_\infty$ is a vector norm with induced matrix norm

$$\|A\|_C = \sup_{x\in\mathbb{C}^n\setminus\{0\}} \frac{\|C^{-1}Ax\|_\infty}{\|C^{-1}x\|_\infty} = \sup_{y=C^{-1}x\in\mathbb{C}^n\setminus\{0\}} \frac{\|C^{-1}ACy\|_\infty}{\|y\|_\infty} = \|C^{-1}AC\|_\infty.$$

For the given $\epsilon \in (0, 1)$, we set $C_\epsilon := SD_\epsilon$ with S from above and $D_\epsilon = \mathrm{diag}(1, \epsilon, \epsilon^2, \dots, \epsilon^{n-1})$. We write $R_\epsilon = C_\epsilon^{-1}EC_\epsilon$. It can be easily shown that for the elements $r_{\epsilon,i\,j}$ of R_ϵ, we have $r_{\epsilon,i\,j} = \epsilon^{j-i}r_{i\,j}$. In particular, $r_{\epsilon,i\,i} = r_{i\,i}$ and $r_{\epsilon,i,i+1} = \epsilon r_{i,i+1}$, while all other elements of R_ϵ are zero. Thus, we have

$$\begin{aligned}\|E\|_{C_\varepsilon} &= \|C_\varepsilon^{-1}EC_\varepsilon\|_\infty = \|R_\varepsilon\|_\infty = \max_{i=1,\dots,n}\sum_{j=1}^{n}|r_{\varepsilon,i\,j}| \\ &= \max_{i=1,\dots,n}\{|r_{i\,i}| + |\varepsilon r_{i,i+1}|\} \le \max_{k=1,\dots,d}\{|\lambda_k| + \varepsilon\} = \rho(E) + \varepsilon,\end{aligned}$$

which gives the desired estimate.

Now, for the proof of the lemma, let $\rho(M^{-1}N) < 1$ and choose any $\epsilon \in (0, 1 - \rho(M^{-1}N))$. Then, according to (2.20), we find a norm $\|\cdot\|_{M^{-1}N,\epsilon}$ such that $\|M^{-1}N\|_{M^{-1}N,\epsilon} \le \rho(M^{-1}N) + \epsilon < 1$. Therefore, the claim follows from Lemma 2.24. □

Remark 2.30 Note that the spectral radius, denoted by $k = \rho(M^{-1}N) + \epsilon$, provides a contraction constant and, therefore, through the estimates in Lemma 2.24, it gives a measure of the convergence speed. However, this is in the generally unknown norm $\|\cdot\|_{M^{-1}N,\epsilon}$. □

In fact, the condition in Lemma 2.29 is not only *sufficient* but also *necessary* for convergence, making it the sharpest possible criterion (we will not discuss the proof here).

Lemma 2.29 applies to arbitrary iterative methods of the form (2.18), and it can be used to prove the convergence of these methods for specific matrices or classes of matrices. For the Gauss-Seidel method, the following theorem provides a corresponding result.

Theorem 2.31 Let $A \in \mathbb{R}^{n\times n}$ be a symmetric, positive definite matrix. Then, the Gauss-Seidel method from Algorithm 2.27 converges to the solution x^* of the equation $Ax = b$ for all initial values $x^{(0)} \in \mathbb{R}^n$. □

Proof: We show that for symmetric and positive definite matrices A the Gauss-Seidel method satisfies the conditions of Lemma 2.29. Therefore, we need to prove that $\rho(M^{-1}N) < 1$. Let $\lambda \in \mathbb{C}$ be an eigenvalue of $M^{-1}N$ with eigenvector $z \in \mathbb{C}^n$, i.e.,

$$M^{-1}Nz = \lambda z,$$

or, equivalently,

$$2Nz = 2\lambda Mz. \tag{2.21}$$

We must show that $|\lambda| < 1$. Now, let $D := \operatorname{diag}(a_{1\,1}, \ldots, a_{n\,n})$. For all $i = 1, \ldots, n$, due to the positive definiteness of A, we have the inequality $a_{i\,i} = e_i^T A e_i > 0$, where e_i denotes the i-th standard basis vector. Therefore, D is also a symmetric and positive definite matrix. Because of the symmetry of A, we have the equations $A = D - N^T - N$ and $M = D - N^T$, and thus (using $M = A + N$),

$$2N = D - A + N - N^T \text{ and } 2M = D + A + N - N^T.$$

Substituting these expressions into (2.21) and multiplying the equation from the left by $\bar{z}^T$, we obtain

$$\bar{z}^T Dz - \bar{z}^T Az + \bar{z}^T(N - N^T)z = \lambda(\bar{z}^T Dz + \bar{z}^T Az + \bar{z}^T(N - N^T)z).$$

Since A and D are symmetric and positive definite, the values $a = \bar{z}^T Az$ and $d = \bar{z}^T Dz$ are real and positive. Since $N - N^T$ is skew-symmetric, $\bar{z}^T(N - N^T)z$ attains a purely imaginary value ib. From the above equation, we have

$$d - a + ib = \lambda(d + a + ib), \text{ so } \lambda = \frac{d - a + ib}{d + a + ib}.$$

This is the quotient of two complex numbers with the same imaginary part, and the numerator has a magnitude smaller than 1. Therefore, $|\lambda| < 1$, which was to be shown. □

Now, let us estimate the computation effort of these iterative methods. To keep the discussion concise, we restrict ourselves to a simple case. Suppose we consider a family of

matrices where the contraction constant k for a given norm $\|\cdot\|$ is independent of the dimension n of the problem. Then, with an appropriate choice of initial values $x^{(0)}$, the number of iterations N_ϵ required to achieve a given accuracy ϵ is independent of n.

For one iteration step and one component $x_j^{(i+1)}$, we need $n-1$ multiplications and additions for the sum and one division, totaling $2n-1$ operations. For the n components of $x^{(i+1)}$, we have $n(2n-1) = 2n^2 - n$ operations, resulting in a total of

$$N_\epsilon(2n^2 - n)$$

operations. In particular, under the assumptions made about the problems, these algorithms have a computational effort of order of $O(n^2)$, meaning that the computational effort grows significantly slower in n compared to Gaussian elimination or the Cholesky factorization.

Additional assumptions about A can significantly reduce the computational effort of these methods. Many large systems of equations have the property that in each row of the (very large) matrix A, only relatively few entries are nonzero, known as a *sparse matrix*. If we assume that, independent of n, each row of A has at most m nonzero entries, then the number of operations for computing $x_j^{(i+1)}$ is at most $2m-1$, and the total number of operations is $N_\epsilon 2(m-1)n$. This results in a complexity of $O(n)$, meaning that the number of operations grows linearly with n. However, under such conditions, the number of operations in Gaussian elimination or the Cholesky factorization may also decrease, typically not going below $O(n^2)$. An exception is "band matrices" with a very simple band structure, for which algorithms for LR factorization with complexity $O(n)$ can be formulated, as discussed in the exercises.

Poor conditioning of A can also cause difficulties in iterative methods, often manifesting as the iteration making no progress due to rounding errors before reaching the desired accuracy. Typically, poor conditioning is reflected in contraction constants k that are close to 1. Preconditioning, as mentioned earlier, can offer a solution, where a matrix P is chosen to make PA better conditioned than A, and then we solve $PAx = Pb$. One possible strategy for this was discussed in Exercise 5 on Sheet 2.

2.10 Relaxation

In this section, we want to briefly explain a variant of the two methods considered without proofs. This variant is based on the so-called *relaxation*, which can be performed based on either the Jacobi or the Gauss-Seidel method. The goal of this relaxation is to accelerate the convergence of these methods.

The idea is as follows: In the Jacobi method, you choose a real parameter $\omega > 0$ and modify the iteration as follows:

$$x_j^{(i+1)} = \frac{1}{a_{j\,j}} \left(b_j - \sum_{\substack{k=1 \\ k \neq j}}^{n} a_{j\,k} x_k^{(i)} \right) =: \Phi_j(x^{(i)})$$

from Algorithm 2.26 to

$$x_j^{(i+1)} = (1-\omega)x_j^{(i)} + \omega\Phi_j(x^{(i)}),$$

meaning that the new approximation value is chosen as a weighted sum between the old value and the new value provided by the iteration rule Φ.

The same approach is applied to the Gauss-Seidel method. According to Algorithm 2.27, for each component j, we have the rule:

$$x_j^{(i+1)} = \frac{1}{a_{j\,j}}\left(b_j - \sum_{k=1}^{j-1} a_{j\,k}x_k^{(i+1)} - \sum_{k=j+1}^{n} a_{j\,k}x_k^{(i)}\right) =: \Phi_j(x^{(i)}, x^{(i+1)}),$$

which is changed to

$$x_j^{(i+1)} = (1-\omega)x_j^{(i)} + \omega\Phi_j(x^{(i)}, x^{(i+1)})$$

In both methods, for $\omega < 1$ the modification is called *under-relaxation*, while for $\omega > 1$ we call it *over-relaxation*. In practice, over-relaxation is more commonly used, and it's also known as the SOR method (SOR = successive over-relaxation), often providing better results. Typical values for ω in practice range between 1.1 and 1.3.

The convergence proofs for these relaxed methods are similar to the corresponding proofs for the underlying methods. For details, see, e.g., Deuflhard/Hohmann [1, Chapter 8].

Let's take a closer look at the variant based on the Gauss-Seidel method: We first decompose the matrix A as follows: $A = D - L - R$

$$A = \underbrace{\begin{pmatrix} a_{1\,1} & 0 & \dots & 0 \\ 0 & a_{2\,2} & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \dots & 0 & a_{n\,n} \end{pmatrix}}_{=:D} - \underbrace{\begin{pmatrix} 0 & 0 & \dots & 0 \\ -a_{2\,1} & 0 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ -a_{n\,1} & \dots & -a_{n\,n-1} & 0 \end{pmatrix}}_{=:L} - \underbrace{\begin{pmatrix} 0 & -a_{1\,2} & \dots & -a_{1\,n} \\ 0 & \ddots & \ddots & \vdots \\ \vdots & \ddots & 0 & -a_{n-1\,n} \\ 0 & \dots & 0 & 0 \end{pmatrix}}_{=:R}$$

The relaxed computation rule given above can be written as

$$Dx^{(i+1)} = (1-\omega)Dx^{(i)} + \omega Lx^{(i+1)} + \omega Rx^{(i)} + \omega b. \tag{2.22}$$

To express this in the familiar form (2.18), we scale the given equation system to $\omega Ax = \omega b$ (which doesn't change the solution) and define an iteration method of the form (2.18) with $M = D - \omega L$ and $N = (1-\omega)D + \omega R$, which implies $M - N = \omega A$. In explicit form:

$$x^{(i+1)} = (D-\omega L)^{-1}((1-\omega)D + \omega R)x^{(i)} + (D-\omega L)^{-1}\omega b, \quad i = 0, 1, 2, \dots \tag{2.23}$$

Note that this rule is equivalent to equation (2.22). For $\omega = 1$, we recover the original Gauss-Seidel method. Since, in many cases, $\omega > 1$ provides faster convergence, the method (2.23) is referred to as the *SOR method* (SOR = successive overrelaxation), even though $\omega < 1$ is theoretically allowed.

For different ranges of ω and different assumptions about the structure of A, convergence of this method can be demonstrated. An example of such a result is provided by the following theorem.

Theorem 2.32 Let $A \in \mathbb{R}^{n\times n}$ be a symmetric, positive definite matrix. Then, the SOR method (2.23) with $\omega \in (0,2)$ converges for all initial values $x^{(0)} \in \mathbb{R}^n$ to the solution x^* of the equation system $Ax = b$. □

Sketch of proof: The proof follows a similar approach to the proof of Theorem 2.31, with equation (2.21) now being

$$2((1-\omega)D + \omega R)z = 2\lambda(D - \omega L)z$$

With similar transformations as in this proof, we eventually arrive at the equation

$$\lambda = \frac{(2-\omega)d - \omega a + i\omega b}{(2-\omega)d + \omega a + i\omega b}$$

from which we conclude $|\lambda| < 1$ for $\omega \in (0,2)$. □

The particular "art" in this method is to choose $\omega > 0$ in such a way that the method converges as fast as possible. Considering Remark 2.30, it is advisable to choose ω so that the spectral radius $\rho((D+\omega L)^{-1}((1-\omega)D + \omega R))$ becomes as small as possible. For specific structures of A, explicit formulas can be derived, but often one relies on "try-and-error" methods, where suitable values for ω are chosen based on numerical experience. An example where the number of iterations can be significantly reduced with the optimal choice of ω can be found in Schwarz/Köckler [8], Example 11.7 (Example 11.5 in the 4th edition).

2.11 The Conjugate Gradient Method

All the methods discussed so far are based on an additive decomposition of the matrix A. Another class of iterative methods follows a completely different idea; they are based on optimization methods. To conclude this section, we will briefly consider a simple representative of this class, the *Conjugate Gradient* or *CG Method*.

The CG method is a technique for solving linear systems of the form $Ax = b$, where A is a symmetric and positive definite matrix[5]. Instead of solving the linear system directly, it rather solves the minimization problem

$$\text{minimize} \quad f(x) = \frac{1}{2}x^T Ax - b^T x.$$

For a solution of this minimization problem, we have

$$0 = \nabla f(x) = Ax - b,$$

which, in turn, provides a solution to the original linear equation system. The CG method is technically a direct method because, at least in theory (i.e., without roundoff errors), it provides an exact result after a finite number of steps. However, it is categorized as

[5]Similar methods exist for general matrices, e.g., the CGS or BiCGstab methods, but they are more complex

an iterative method because the intermediate results of the process already approximate solutions, so the method is often terminated before reaching the exact solution. The approximation $x^{(i+1)}$ is determined from the previous approximation by finding a search direction $d^{(i)} \in \mathbb{R}^n$ and a step size $\alpha^{(i)} \in \mathbb{R}$, and then setting:

$$x^{(i+1)} = x^{(i)} + \alpha^{(i)} d^{(i)},$$

where $d^{(i)}$ and $\alpha^{(i)}$ are chosen to minimize $f(x^{(i+1)})$ and satisfy $f(x^{(i+1)}) < f(x^{(i)})$.

Choosing the Step Size: For a given search direction $d^{(i)}$, the step size $\alpha^{(i)}$ should be chosen to minimize $h(\alpha) = f(x^{(i)} + \alpha d^{(i)})$. This is a one-dimensional optimization problem since h maps from $\mathbb{R}$ to $\mathbb{R}$. Due to the structure of h or f, it can be calculated that the minimum is achieved at

$$\alpha = \frac{(b - Ax^{(i)})^T d^{(i)}}{{d^{(i)}}^T A d^{(i)}}.$$

Choosing the Search Direction: The choice of the search direction differs from step to step. In the first step, the direction of steepest descent is used. Since the gradient ∇f points in the direction of the steepest ascent, we select

$$d^{(0)} = -\nabla f(x^{(0)}) = -(Ax^{(0)} - b) = b - Ax^{(0)}.$$

For $i \geq 1$, the subsequent search directions are chosen so that they are orthogonal to the previous direction with respect to the scalar product $\langle x, y \rangle_A = x^T A y$. This ensures that the search is conducted uniformly in all directions of $\mathbb{R}^n$. Formally, $d^{(i)}$ is chosen such that:

$$\langle d^{(i)}, d^{(i-1)} \rangle_A = 0.$$

In addition to this condition (which is satisfied by many vectors $d^{(i)}$), the following approach is taken:

$$d^{(i)} = r^{(i)} + \beta^{(i)} d^{(i-1)} \text{ with } r^{(i)} = -\nabla f(x^{(i)}) = b - Ax^{(i)}.$$

In essence, we move in the direction of the negative gradient (which is also the residual of the linear equation system) but modify this direction by adding $\beta_i d^{(i-1)}$. This specific correction allows for an easy calculation of $\beta^{(i)}$ as:

$$\beta^{(i)} = -\frac{{r^{(i)}}^T A d^{(i-1)}}{{d^{(i-1)}}^T A d^{(i-1)}}.$$

The vector $d^{(i)}$ constructed in this way is indeed orthogonal to all previous search directions $d^{(0)}, \ldots, d^{(i-1)}$, although this is not immediately apparent and requires some effort to prove. Note that in all these calculations, the denominator of the fractions is not zero because A is positive definite.

It can be proven that the CG method (in theory, without roundoff errors) finds an exact solution to the problem after at most n steps. For large n, you may want to terminate the iteration earlier, i.e., when a specified level of accuracy is reached. Typically, the termination criterion used here is $\|r^{(i)}\| \leq \varepsilon$. Since $r^{(i)} = b - Ax^{(i)}$, this estimate controls the size of the residual, allowing us to obtain an estimate of the actual error using Theorem 2.9.

Chapter 3

Interpolation

Interpolation of functions or data is a common problem in both mathematics and many applications.

The general problem, known as *data interpolation*, arises when we have a set of data points (x_i, f_i) for $i = 0, \ldots, n$ (e.g., experimental measurements). The problem statement is as follows: we are looking for a function F such that the equation

$$F(x_i) = f_i \quad \text{for } i = 0, 1, \ldots, n \tag{3.1}$$

holds.

An important special case of this problem is *function interpolation*. Let's assume we have a real-valued function $f : \mathbb{R} \to \mathbb{R}$, which is, however, complicated to evaluate (e.g., because no explicit formula is known). An example of such a function is the Gaussian distribution function often needed in statistics

$$f(x) = \frac{1}{2\pi} \int_{-\infty}^{x} e^{-y^2/2} dy,$$

for which no closed formula exists.

The goal of interpolation is to determine a function $F(x)$ that is easy to evaluate and satisfies the equation

$$F(x_i) = f(x_i) \quad \text{for } i = 0, 1, \ldots, n \tag{3.2}$$

for given support points x_0, $x_1, \ldots, x_n$.

Using the notation

$$f_i = f(x_i),$$

we again obtain the condition (3.1), which is why (3.2) is indeed a special case of (3.1).

In this chapter, we will develop methods for solving (3.1), which can also be applied to the special case (3.2). The importance of this special case lies in the fact that when interpolating a function f, we can naturally define an *interpolation error* based on the distance between f and F, providing a measure of the method's quality. In data interpolation, this doesn't make much sense since there is no function f to measure the error against.

We will also examine methods tailored specifically for function approximation (3.2). In these methods, the choice of support points x_i arises from the procedure itself and cannot be arbitrarily specified.

3.1 Polynomial Interpolation

A simple but often very effective method for interpolation is to choose F as a polynomial, i.e., a function of the form

$$P(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_m x^m. \tag{3.3}$$

Here, the values a_i, $i = 0, \ldots, m$, are called the *coefficients* of the polynomial. The highest occurring power (in this case m, if $a_m \neq 0$) is called the *degree* of the polynomial. To emphasize that we are using polynomials in this section, we write "P" instead of "F" for the interpolation function. The space of polynomials of degree $\leq m$ is denoted by $\mathcal{P}_m$. This function space is an $m+1$-dimensional vector space over $\mathbb{R}$ or $\mathbb{C}$ with the basis $\mathcal{B} = \{1, x, \ldots, x^m\}$ since adding polynomials and multiplying them by scalars results in a polynomial of the same degree. Other bases of this vector space will be discussed in the exercises.

The problem of polynomial interpolation is to determine a polynomial P that satisfies (3.1). First, we need to decide the degree of the polynomial we are looking for. The following theorem helps us with this decision.

Theorem 3.1 Let $n \in \mathbb{N}$, and let data points (x_i, f_i) be given for $i = 0, \ldots, n$ such that the support points are pairwise distinct, i.e., $x_i \neq x_j$ for all $i \neq j$. Then there exists a unique polynomial $P \in \mathcal{P}_n$, i.e., of degree $\leq n$, that satisfies the condition

$$P(x_i) = f_i \quad \text{for } i = 0, 1, \ldots, n.$$

□

Proof: The coefficients a_i of the interpolating polynomial satisfy the linear system of equations

$$\begin{pmatrix} 1 & x_0 & \ldots & x_0^n \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & \ldots & x_n^n \end{pmatrix} \begin{pmatrix} a_0 \\ \vdots \\ a_n \end{pmatrix} = \begin{pmatrix} f_0 \\ \vdots \\ f_n \end{pmatrix}.$$

The determinant of this matrix is

$$\prod_{i=0}^{n} \left(\prod_{j=i+1}^{n} (x_j - x_i) \right)$$

and is non-zero if the x_i are pairwise distinct. Therefore, the matrix is invertible, and the system of equations has a unique solution. □

For $n+1$ given data points (x_i, f_i), a polynomial of degree n precisely matches the data. Now, for various reasons, it is not very efficient to actually solve this linear system of equations to determine the a_i (remember that direct solution of the linear system has a computational complexity of $O(n^3)$). Therefore, we will consider a different technique for computing the polynomial P. Note that this technique provides the same polynomial, but it is represented differently.

3.1.1 Lagrange Polynomials and Barycentric Coordinates

The idea behind Lagrange polynomials is based on a clever representation for polynomials. For the given support points $x_0, x_1, \ldots, x_n$, we define, for $i = 0, \ldots, n$, the *Lagrange Polynomials* L_i as

$$L_i(x) := \prod_{\substack{j=0 \\ j \neq i}}^{n} \frac{x - x_j}{x_i - x_j}.$$

It can be easily verified that these polynomials are all of degree n and, moreover, satisfy the equation

$$L_i(x_k) = \begin{cases} 1 & \text{for } i = k \\ 0 & \text{for } i \neq k \end{cases}$$

With the help of L_i, we can easily compute the interpolation polynomial explicitly.

Theorem 3.2 Let data points (x_i, f_i) for $i = 0, \ldots, n$ with pairwise distinct support points x_i be given. Then the unique interpolation polynomial $P(x)$ with $P(x_i) = f_i$ is given by

$$P(x) = \sum_{i=0}^{n} f_i L_i(x).$$

□

Proof: It is obvious that the given function is a polynomial of degree $\leq n$. Furthermore, we have

$$P(x_k) = \sum_{i=0}^{n} \underbrace{f_i L_i(x_k)}_{\substack{=0 \text{ if } i \neq k \\ = f_k \text{ if } i = k}} = f_k,$$

thus satisfying the desired condition (3.1). □

The Lagrange polynomials are orthogonal (even orthonormal) with respect to the scalar product

$$\langle P, Q \rangle := \sum_{i=0}^{n} P(x_i) Q(x_i)$$

in the space of polynomials $\mathcal{P}_n$, making them an orthonormal basis of $\mathcal{P}_n$ with respect to this scalar product. This is because every polynomial of degree $\leq n$ can be expressed as a sum of the L_i using

$$P = \sum_{i=0}^{n} P(x_i) L_i = \sum_{i=0}^{n} \langle P, L_i \rangle L_i$$

We will see later that orthogonality (although with respect to different scalar products) is a useful property in function interpolation.

Example 3.3 Consider the data points $(3, 68)$, $(2, 16)$, $(5, 352)$. The corresponding Lagrange polynomials are given by

$$L_0(x) = \frac{x-2}{3-2} \frac{x-5}{3-5} = -\frac{1}{2}(x-2)(x-5),$$

$$L_1(x) = \frac{x-3}{2-3}\frac{x-5}{2-5} = \frac{1}{3}(x-3)(x-5),$$
$$L_2(x) = \frac{x-2}{5-2}\frac{x-3}{5-3} = \frac{1}{6}(x-2)(x-3).$$

Thus, we obtain

$$P(x) = -68\,\frac{1}{2}(x-2)(x-5) + 16\,\frac{1}{3}(x-3)(x-5) + 352\frac{1}{6}(x-2)(x-3).$$

For $x = 3$, we get $P(3) = -68\frac{1}{2}(3-2)(3-5) = 68$, for $x = 2$, we calculate $P(2) = 16\frac{1}{3}(2-3)(2-5) = 16$, and for $x = 5$, we obtain $P(5) = 352\frac{1}{6}(5-2)(5-3) = 352$. □

By counting the necessary operations, it can be seen that the direct evaluation of the polynomial P in this form has a computational complexity of $O(n^2)$, which is significantly more efficient than solving a linear system of equations. For an efficient direct evaluation, the denominators of the Lagrange polynomials should be calculated and stored in advance, so they don't need to be recomputed for each evaluation of P.

However, we can make the evaluation even more efficient if we cleverly reformulate the evaluation of the Lagrange polynomials. To do this, we express the numerator of

$$L_i(x) = \prod_{\substack{j=0\\ j\neq i}}^{n} \frac{x-x_j}{x_i-x_j}.$$

as

$$\frac{\ell(x)}{x-x_i} \quad \text{with} \quad \ell(x) := \prod_{j=0}^{n} x - x_j.$$

We write the denominator using the so-called *barycentric coordinates*

$$w_i := \prod_{\substack{j=0\\ j\neq i}}^{n} \frac{1}{x_i-x_j}.$$

Then we have

$$L_i(x) = \ell(x)\frac{w_i}{x-x_i}$$

and thus

$$P(x) = \sum_{i=0}^{n} L_i(x) f_i = \sum_{i=0}^{n} \ell(x)\frac{w_i}{x-x_i} f_i = \ell(x)\sum_{i=0}^{n} \frac{w_i}{x-x_i} f_i.$$

Example 3.4 Consider again the data points $(3, 68)$, $(2, 16)$, $(5, 352)$. The corresponding ℓ is given by

$$\ell(x) = (x-2)(x-3)(x-5)$$

and the w_i are calculated as

$$w_0 = \frac{1}{3-2}\frac{1}{3-5} = -\frac{1}{2},$$
$$w_1 = \frac{1}{2-3}\frac{1}{2-5} = \frac{1}{3},$$

$$w_3 = \frac{1}{5-2}\,\frac{1}{5-3} = \frac{1}{6}.$$

This gives us

$$\begin{aligned} P(x) &= \ell(x)\left(\frac{-\frac{1}{2}}{x-3}\,68 + \frac{\frac{1}{3}}{x-2}\,16 + \frac{\frac{1}{6}}{x-5}\,352\right) \\ &= -\frac{1}{2}(x-2)(x-5)\,68 + \frac{1}{3}(x-3)(x-5)\,16 + \frac{1}{6}(x-2)(x-3)\,352, \end{aligned}$$

which is the same polynomial as before. □

To implement this procedure efficiently, we divide the computation into two algorithms.

Algorithm 3.5 (Computation of Barycentric Coordinates)
Input: Support points $x_0,\ldots,x_n$

(1) for i from 0 to n:

(2) set $w_i := 1$

(3) for j from 0 to n:

(4) if $j \neq i$, set $w_i := w_i/(x_i - x_j)$

(5) End of loops

Output: Barycentric coordinates $w_0,\ldots,w_n$ □

By counting the operations, it is easy to see that the computation of w_i requires exactly $2(n+1)n = 2n^2 + 2n = O(n^2)$ operations. This corresponds to the order of complexity of directly evaluating P. However, the trick is to calculate the w_i values in advance and then use the stored values in the evaluation of P.

Algorithm 3.6 (Evaluation of the Interpolation Polynomial)
Input: Support points $x_0,\ldots,x_n$, values $f_0,\ldots,f_n$, barycentric coordinates $w_0,\ldots,w_n$, evaluation point x

(0) set $l := 1$, $s := 0$ (variables for ℓ and $\sum_{i=0}^{n} \frac{w_i}{x-x_i} f_i$)

(1) for i from 0 to n

(2) set $y := x - x_i$

(3) if $y = 0$, set $P := f_i$ and terminate the algorithm

(4) set $l := l * y$

(5) set $s := s + w_i * f_i/y$

(6) End of loop

(7) Set $P := l * s$

Output: Polynomial value $P = P(x)$ □

By counting the operations, we see that the evaluation requires exactly $5(n+1)+1 = 5n+6 = O(n)$ operations. So, once the w_i values are calculated, the evaluation for a given x is significantly less computationally intensive than the direct evaluation of P. This is an important advantage, especially when graphically representing the polynomial, as it needs to be evaluated for many different x values.

3.1.2 Condition

In this section, we want to examine the condition of polynomial interpolation, considering the polynomial interpolation problem with fixed given support points. In this case, the mapping

$$\phi : (f_0, \ldots, f_n) \mapsto \sum_{i=0}^{n} f_i L_i$$

of the data vector $(f_0, \ldots, f_n)$ to the interpolating polynomial $P \in \mathcal{P}_n$ is a linear mapping $\phi : \mathbb{R}^{n+1} \to \mathcal{P}_n$. Therefore, we can calculate the (absolute) condition κ_{abs} as the induced operator norm

$$\kappa_{abs} := \|\phi\|_\infty = \sup_{\substack{f \in \mathbb{R}^{n+1} \\ f \neq 0}} \frac{\|\phi(f)\|_\infty}{\|f\|_\infty}$$

of this linear mapping. This induced operator norm is an extension of the induced matrix norm to linear mappings that are not necessarily defined by a matrix. Unlike linear systems of equations, here, we use absolute condition because a relative definition does not have an intuitive interpretation. On the polynomial space $\mathcal{P}_n$, we use the maximum norm

$$\|P\|_\infty := \max_{x \in [a,b]} |P(x)|,$$

of the space of continuous real-valued functions $C([a,b], \mathbb{R})$, where we choose $a = \min_{i=0,\ldots,n} x_i$ and $b = \max_{i=0,\ldots,n} x_i$ (note that we have not assumed any order of support points).

Theorem 3.7 Let x_0, $x_1, \ldots, x_n$ be pairwise distinct support points, and L_i be the corresponding Lagrange polynomials. Then the absolute condition of the interpolation problem with these support points is given by

$$\kappa_{\text{abs}} = \Lambda_n := \left\| \sum_{i=0}^{n} |L_i| \right\|_\infty ,$$

where Λ_n is called the Lebesgue constant. □

Proof: It holds that

$$\begin{aligned} |\phi(f)(x)| &= \left| \sum_{i=0}^{n} f_i L_i(x) \right| \leq \sum_{i=0}^{n} |f_i| \, |L_i(x)| \\ &\leq \|f\|_\infty \max_{x \in [a,b]} \sum_{i=0}^{n} |L_i(x)| = \|f\|_\infty \left\| \sum_{i=0}^{n} |L_i| \right\|_\infty = \|f\|_\infty \Lambda_n, \end{aligned}$$

for all $x \in [a,b]$, implying $\|\phi(f)\|_\infty \leq \|f\|_\infty \Lambda_n$ for all $f \in \mathbb{R}^{n+1}$ and thus $\|\phi\| \leq \Lambda_n$. For the converse inequality we construct a $g \in \mathbb{R}^{n+1}$ such that

$$|\phi(g)(x^*)| = \|g\|_\infty \left\| \sum_{i=0}^{n} |L_i| \right\|_\infty$$

holds for some $x^* \in [a,b]$. To this end, let $x^* \in [a,b]$ be the point where the function $x \mapsto \sum_{i=0}^{n} |L_i(x)|$ attains its maximum, i.e.

$$\sum_{i=0}^{n} |L_i(x^*)| = \left\| \sum_{i=0}^{n} |L_i| \right\|_\infty = \Lambda_n.$$

We choose $g \in \mathbb{R}^{n+1}$ as $g_i = \operatorname{sgn}(L_i(x^*))$. Then $\|g\|_\infty = 1$ und $g_i L_i(x^*) = |L_i(x^*)|$, implying

$$\|\phi(g)\|_\infty \geq |\phi(g)(x^*)| = \left| \sum_{i=0}^{n} g_i L_i(x^*) \right| = \left| \sum_{i=0}^{n} |L_i(x^*)| \right| = \|g\|_\infty \left| \sum_{i=0}^{n} |L_i(x^*)| \right| = \|g\|_\infty \Lambda_n,$$

from whish $\|\phi\| \geq \Lambda_n$ follows. Together the two inequalities show the claim $\kappa_{\text{abs}} = \|\phi\| = \Lambda_n$. □

The Lebesgue constant Λ_n depends on the number and location of support points. In the following Table 3.1, the conditions for the interval $[-1,1]$ and different numbers of equidistant support points $x_i = -1 + 2i/n$ as well as for the so-called Chebyshev support points $x_i = \cos[(2i+1)\pi/(2n+2)]$ are presented.

n	κ_{abs} for equidistant support points	κ_{abs} for Chebyshev support points
5	3.11	2.10
10	29.89	2.49
15	512.05	2.73
20	10986.53	2.90
60	$2.97 \cdot 10^{15}$	3.58
100	$1.76 \cdot 10^{27}$	3.90

Table 3.1: Condition κ_{abs} for different support points

It can be seen that the problem is very poorly conditioned for equidistant support points and large values of n.

3.1.3 The Newton Scheme

We now consider another method for calculating interpolation polynomials, the so-called *Newton Scheme*, which has the advantage of being suitable for more general interpolation tasks that we will introduce in the following section. For the Newton Scheme, we first define the following values:

$$f_{[x_i]} = f_i, \quad i = 0, \ldots, n$$

and recursively calculate the so-called *divided differences* for support point sets of the form $\{x_l, x_{l+1}, \ldots, x_{l+k}\}$ with $0 \leq l < l+k \leq n$:

$$f_{[x_l, x_{l+1}, \ldots, x_{l+k}]} := \frac{f_{[x_l, x_{l+1}, \ldots, x_{l+k-1}]} - f_{[x_{l+1}, x_{l+2}, \ldots, x_{l+k}]}}{x_l - x_{l+k}}.$$

Figure 3.1 illustrates this recursive calculation.

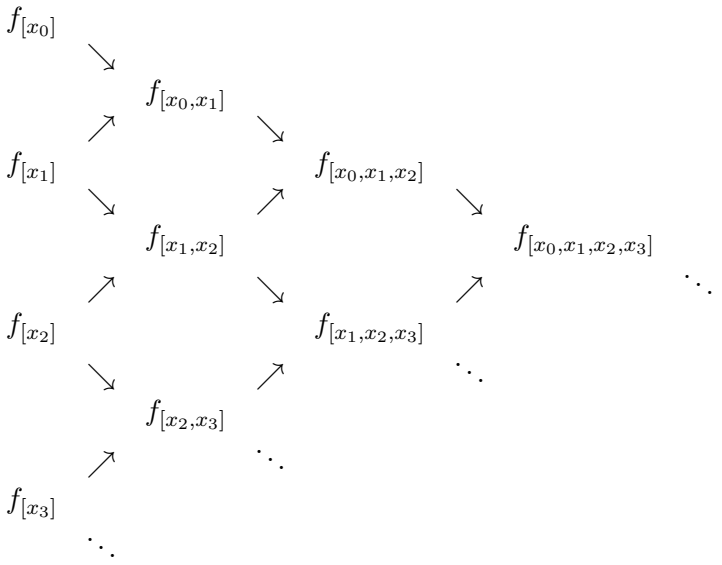


Figure 3.1: Illustration of the Newton Scheme

The following theorem shows how to calculate the interpolation polynomial from the divided differences.

Theorem 3.8 Let data (x_i, f_i) for $i = 0, \ldots, n$ be given with pairwise distinct support points x_i. Then the unique interpolation polynomial $P(x)$ with $P(x_i) = f_i$ for $i = 0, \ldots, n$ is given by

$$\begin{aligned} P(x) &= \sum_{k=0}^{n} \left(f_{[x_0,\ldots,x_k]} \prod_{j=0}^{k-1} (x - x_j) \right) \\ &= f_{[x_0]} + f_{[x_0,x_1]}(x - x_0) + f_{[x_0,x_1,x_2]}(x - x_0)(x - x_1) \\ &\quad + \ldots + f_{[x_0,\ldots,x_n]}(x - x_0)(x - x_1) \cdots (x - x_{n-1}). \end{aligned}$$

□

Proof: We first show that the interpolation polynomial $P(x) = a_n x^n + \ldots + a_1 x + a_0$ satisfies the equation

$$a_n = f_{[x_0,\ldots,x_n]}. \tag{3.4}$$

The assertion is clear for $n = 0$. To prove (3.4) for $n \geq 1$, we consider the interpolation polynomials P^0_{n-1} and P^n_{n-1} for the data $\{(x_k, f_k) : k = 1, \ldots, n\}$ and $\{(x_k, f_k) : k = 0, \ldots, n-1\}$, respectively. For these, the equation

$$P(x) = \frac{(x_0 - x)P^n_{n-1}(x) - (x_n - x)P^0_{n-1}(x)}{x_0 - x_n} \tag{3.5}$$

holds, as the right-hand side of this equation defines a polynomial of degree $\leq n$ that satisfies the interpolation condition. The proof of (3.4) now follows by induction on n: For $n-1 \to n$, we consider (3.5). From this equation, we have $a_n = (-a^n_{n-1} + a^0_{n-1})/(x_0 - x_n)$, where a^n_{n-1} and a^0_{n-1} are the leading coefficients of P^n_{n-1} and P^0_{n-1}, respectively. According to the induction assumption, $a^n_{n-1} = f_{[x_0,\ldots,x_{n-1}]}$ and $a^0_{n-1} = f_{[x_1,\ldots,x_n]}$, so

$$a_n = \frac{a^n_{n-1} - a^0_{n-1}}{x_0 - x_n} = \frac{f_{[x_0,\ldots,x_{n-1}]} - f_{[x_1,\ldots,x_n]}}{x_0 - x_n} = f_{[x_0,\ldots,x_n]}.$$

We now prove the theorem by induction on n. For $n = 0$, the statement is clear. For $n - 1 \to n$, according to the induction assumption,

$$P_{n-1}(x) = \sum_{k=0}^{n-1} \left(f_{[x_0,\ldots,x_k]} \prod_{j=0}^{k-1} (x - x_j) \right)$$

is the interpolation polynomial for the data $\{(x_i, f_i) : i = 0, \ldots, n-1\}$. Let P_n be the interpolation polynomial for $\{(x_i, f_i) : i = 0, \ldots, n\}$. Then, for P_n, according to (3.4), we have

$$\begin{aligned} P_n(x) &= f_{[x_0,\ldots,x_n]} x^n + a_{n-1} x^{n-1} + \ldots + a_0 \\ &= f_{[x_0,\ldots,x_n]} \left(\prod_{j=0}^{n-1} (x - x_j) \right) + Q_{n-1}(x) \end{aligned}$$

for some polynomial $Q_{n-1} \in \mathcal{P}_{n-1}$. This polynomial

$$Q_{n-1}(x) = P_n(x) - f_{[x_0,\ldots,x_n]} \left(\prod_{j=0}^{n-1} (x - x_j) \right)$$

satisfies the conditions $Q_{n-1}(x_i) = f_i$ for $i = 0, \ldots, n-1$, which implies that $Q_{n-1} = P_{n-1}$. Therefore, the claimed equation

$$\begin{aligned} P_n(x) &= f_{[x_0,\ldots,x_n]} \left(\prod_{j=0}^{n-1} (x - x_j) \right) + P_{n-1}(x) \\ &= f_{[x_0,\ldots,x_n]} \left(\prod_{j=0}^{n-1} (x - x_j) \right) + \sum_{k=0}^{n-1} \left(f_{[x_0,\ldots,x_k]} \prod_{j=0}^{k-1} (x - x_j) \right) \end{aligned}$$

follows. □

It's worth noting that for this polynomial, we only need the values from the first rows of the scheme. However, the other rows are necessary for calculating these values.

We revisit our example from the previous section.

Example 3.9 Consider the data points (3; 68), (2; 16), (5; 352). The divided differences are calculated as follows:

$$\begin{array}{ccccc} f_{[x_0]} = 68 & & & & \\ & \searrow & & & \\ & & f_{[x_0,x_1]} = \frac{68-16}{3-2} = 52 & & \\ & \nearrow & & \searrow & \\ f_{[x_1]} = 16 & & & & f_{[x_0,x_1,x_2]} = \frac{52-112}{3-5} = 30 \\ & \searrow & & \nearrow & \\ & & f_{[x_1,x_2]} = \frac{16-352}{2-5} = 112 & & \\ & \nearrow & & & \\ f_{[x_2]} = 352 & & & & \end{array}$$

This gives us the interpolation polynomial:

$$P(x) = P_2(x) = 68 + 52(x-3) + 30(x-3)(x-2).$$

For $x = 3$, we have $P(3) = 68$, for $x = 2$, we get $P(2) = 68 + 52(2-3) = 68 - 52 = 16$, and for $x = 5$, we calculate $P(5) = 68+52(5-3)+30(5-3)(5-2) = 68+104+180 = 352$. □

Even though the computation of the interpolation polynomial through the Newton scheme may appear complex, it offers several advantages. Firstly, additional data points (x_i, f_i) can be easily added, as the polynomials for more interpolation points can be obtained by adding extra terms to those with fewer interpolation points. (However, it's important to note that when a single value f_i changes, all the divided differences depending on f_i need to be recalculated.)

Secondly, both the calculation of divided differences and the evaluation of the polynomial can be efficiently implemented. The calculation of divided differences can be performed using the following algorithm, where the notation

$$F_0 = f_{[x_0]},\ F_1 = f_{[x_0,x_1]},\ \ldots,\ F_n = f_{[x_0,x_1,\ldots,x_n]}$$

is used.

Algorithm 3.10 (Calculation of Divided Differences)
Input: Data $(x_0, f_0), \ldots, (x_n, f_n)$.

(0) For i from 0 to n, set $F_i := f_i$; end of loop

(1) For k from 1 to n

(2) For i from n to k (counting downwards)

(3) Calculate $F_i := \frac{F_{i-1}-F_i}{x_{i-k}-x_i}$

(4) End of loops

Output: Divided differences $F_0, \ldots, F_n$ □

This algorithm has a computational complexity of $O(n^2)$. Once these F_i values are calculated, the interpolation polynomial can be evaluated using the following formula, known as the *Horner's scheme*.

Algorithm 3.11 (Calculation of $P(x)$ using Horner's Scheme)
Input: Interpolation points $x_0, \ldots, x_n$, divided differences $F_0, \ldots, F_n$ from Algorithm 3.10, $x \in \mathbb{R}$.

(0) Set $P := F_n$

(1) For i from $n-1$ to 0 (counting downwards)

(2) Calculate $P := F_i + (x - x_i)P$

(3) End of loop

Output: $P = P(x)$ □

Similar to the calculation of $P(x)$ using barycentric coordinates, this algorithm has a computational complexity of $O(n)$.

It is important to note that when using barycentric coordinates, it is easy to evaluate polynomials for different f_i values but with constant interpolation points x_i, as the barycentric coordinates w_i remain unchanged. In contrast, when using the Newton scheme, the F_i values must be completely recalculated for changed f_i values. However, adding a new interpolation point x_{n+1} is simpler with the Newton scheme (provided the complete scheme is saved), as the $F_0, \ldots, F_n$ remain the same, and only the bottom diagonal of the scheme needs to be recalculated. In the case of barycentric coordinates, all w_i values need to be recalculated in such a scenario.

3.2 Hermite Interpolation

As previously mentioned, we will now consider a generalized interpolation problem known as *Hermite Interpolation*[1], where, in addition to function values, derivatives can also be specified.

To formalize this, we relax the assumption made earlier that the interpolation nodes are pairwise distinct. We require that identical nodes appear next to each other in the ordering, which is, for instance, satisfied if the interpolation nodes are ordered in ascending order. For example,

$$x_0 < x_1 = x_2 = x_3 < x_4 < x_5 = x_6$$

is now a valid set of interpolation points. For each interpolation point x_i, we define its multiplicity, starting from 0. The previously mentioned set of interpolation nodes, along with their multiplicities, would be as follows:

x_i	x_0	$<$	x_1	$=$	x_2	$=$	x_3	$<$	x_4	$<$	x_5	$=$	x_6
d_i	0		0		1		2		0		0		1

[1] Named after the french mathematician Charles Hermite, 1822–1901.

The problem of Hermite interpolation now consists of finding a polynomial P given interpolation nodes $x_0, \ldots, x_n$ with multiplicities $d_0, \ldots, d_n$ and given values f_i^j such that the condition

$$P^{(d_i)}(x_i) = f_i^{(d_i)} \quad \text{for } i = 0, 1, \ldots, n \tag{3.6}$$

is satisfied, where $P^{(j)}$ represents the j-th derivative. In the special case of function interpolation, these values are given by

$$f_i^{(d_i)} = f^{(d_i)}(x_i) \tag{3.7}$$

where f is assumed to be sufficiently differentiable.

We denote the interpolation interval as $[a, b]$, and assume that all x_i lie in $[a, b]$. However, the boundary points a and b do not necessarily need to be interpolation points.

The following theorem shows that the problem is well-defined.

Theorem 3.12 For every set of support points $x_0, \ldots, x_n$ with multiplicities $d_i \leq r$ for $i = 0, \ldots, n$, there exists a unique polynomial $P \in \mathcal{P}_n$ that satisfies (3.6). □

Proof: Consider the linear mapping

$$\mu : \mathcal{P}_n \to \mathbb{R}^{n+1}$$

defined by

$$\mu(P) \mapsto \left(P^{(d_0)}(x_0), P^{(d_1)}(x_1), \ldots, P^{(d_n)}(x_n) \right).$$

This mapping is injective, meaning that $\mu(P) = 0$ implies $P \equiv 0$. This is because $\mu(P) = 0$ implies that P has at least $n+1$ zeros (counting multiplicities), which can only happen for a polynomial of degree n if it is the zero polynomial. Since $\dim(\mathcal{P}_n) = n+1 = \dim(\mathbb{R}^{n+1})$, the mapping μ is surjective, and therefore invertible, leading to the unique existence of P. □

Two special cases of this problem are worth mentioning:

(1) If the x_i are all distinct, we obtain the well-known interpolation problem.

(2) If all the x_i are the same, the Hermite function interpolation polynomial with (3.6) and (3.7) consists of the first $n+1$ terms of the Taylor expansion of f, given by

$$P(x) = \sum_{j=0}^{n} \frac{(x - x_0)^j}{j!} f^{(j)}(x_0). \tag{3.8}$$

It can be easily verified that this P satisfies the condition (3.6).

In the general case, Hermite interpolation polynomials can be computed using divided differences as discussed in Section 3.1.3. We only need to extend their definition for the case of coinciding support points. For that, we define

$$f_{[x_i]} := f^0_{i-d_i}, \quad i = 0, \ldots, n,$$

and

$$f_{[x_l,x_{l+1},\ldots,x_{l+k}]} := \frac{f_{[x_l,x_{l+1},\ldots,x_{l+k-1}]} - f_{[x_{l+1},x_{l+2},\ldots,x_{l+k}]}}{x_l - x_{l+k}}, \quad \text{if } x_l \neq x_{l+k}$$

$$f_{[x_l,x_{l+1},\ldots,x_{l+k}]} := \frac{f^{(k)}_{l-d_l+k}}{k!}, \quad \text{if } x_l = \ldots = x_{l+k}$$

This definition is meaningful only if identical nodes appear next to each other in the ordering $x_0, \ldots, x_N$, as otherwise not all possible cases are covered in the above definition.

With this definition, we can formulate the following theorem, which is completely analogous to Theorem 3.8.

Theorem 3.13 Let $x_0, x_1, \ldots, x_n$ be interpolation nodes with multiplicities d_i and corresponding values $f_i^{(d_i)}$, which are ordered such that identical nodes appear next to each other. Then, the unique interpolation polynomial $P(x)$ with $P^{(d_i)}(x_i) = f_i^{(d_i)}$ for $i = 0, \ldots, n$ is given by

$$\begin{aligned} P(x) &= \sum_{k=0}^{n} \left(f_{[x_0,\ldots,x_k]} \prod_{j=0}^{k-1} (x - x_j) \right) \\ &= f_{[x_0]} + f_{[x_0,x_1]}(x - x_0) + f_{[x_0,x_1,x_2]}(x - x_0)(x - x_1) \\ &\quad + \ldots + f_{[x_0,\ldots,x_n]}(x - x_0)(x - x_1)\cdots(x - x_{n-1}). \end{aligned}$$

□

Proof: Completely analogous to the proof of Theorem 3.8, with the (simple) case of $x_0 = x_n$ in the proof of (3.4) considered separately. □

3.2.1 Error Estimates

In this section, we consider the problem of function interpolation (3.2) for a given function $f : \mathbb{R} \to \mathbb{R}$, or its Hermite extension (3.6) with $f_i^j = f^{(j)}(x_i)$. We want to estimate how large the difference between the interpolating polynomial P and the function f is. Here, we denote by $[a, b]$ an interpolation interval with the property that all support points x_i satisfy $x_i \in [a, b]$.

For error analysis, we use divided differences and begin with some preparatory lemmas.

Lemma 3.14 Suppose the conditions of Theorem 3.13 are satisfied. Furthermore, assume that $f \in C^{r+1}[a, b]$. Then, we have

$$f(x) = P(x) + f_{[x_0,\ldots,x_n,x]} \prod_{j=0}^{n} (x - x_j)$$

for all $x \in [a, b]$.

Proof: One either has $x = x_i$ for some $i = 1, \ldots, n$. In this case, $P(x_i) = f(x_i)$ and the product on the right hand side equals zero. Hence, the assertion is true regardless of the value of $f_{[x_0,\ldots,x_n,x]}$. Otherwise, we have $x \neq x_i$ for all $i = 0, \ldots, n$, implying that the node sequence $x_0, \ldots, x_n, x$ extended by x satisfies the assumptions of Theorem 3.13. Hence,

$$\widetilde{P}(x) = P(x) + f_{[x_0,\ldots,x_i,x_{i+1},\ldots,x_n,x]} \prod_{j=0}^{n} (x - x_j)$$

which according to Theorem 3.13 is the Hermite interpolation polynomial through the support points $x_0, \ldots, x_n, x$. Hence, its value coincides with the value of f in the node x, i.e., $\widetilde{P}(x) = f(x)$. □

We will use this lemma to estimate the size of the interpolation error in terms of the term $f_{[x_0,\ldots,x_n,x]} \prod_{j=0}^{n}(x - x_j)$. For this purpose, we need the following formula.

Lemma 3.15 For the n-th divided difference of an n-times continuously differentiable function, the *Hermite-Genocchi formula* holds:

$$f_{[x_0,\ldots,x_n]} = \int_{\Sigma^n} f^{(n)} \left(x_0 + \sum_{i=1}^{n} s_i (x_i - x_0) \right) ds,$$

where, for $n \geq 1$,

$$\Sigma^n := \left\{ s = (s_1, \ldots, s_n) \in \mathbb{R}^n \;\middle|\; \sum_{i=1}^{n} s_i \leq 1 \text{ and } s_i \geq 0 \right\}$$

denotes the n-dimensional standard simplex. We define for $n = 0$,

$$\int_{\Sigma^0} f(x_0) ds = f(x_0)$$

Proof: We prove the formula by induction on n. For $n = 0$, the claim follows from $f_{[x_0]} = f(x_0)$ and the definition of the integral over Σ^0.

For the induction step from $n \to n + 1$, we first consider the special case where $x_0 = x_1 = \ldots = x_{n+1}$. In this case, we have

$$f_{[x_0,\ldots,x_{n+1}]} := \frac{f^{(n+1)}(x_0)}{(n+1)!} \quad \text{and} \quad x_0 + \sum_{i=1}^{n+1} s_i (x_i - x_0) = x_0,$$

which immediately proves the claim since, for all $n \geq 1$, we have the equation

$$\int_{\Sigma^n} 1 \, ds = \operatorname{Vol} \Sigma_n = \frac{1}{n!} \tag{3.9}$$

If $x_0 \neq x_{n+1}$, we denote for $s = (s_1, \ldots, s_{n+1}) \in \Sigma^{n+1}$ and $1 \leq i \leq j \leq n+1$, the vectors $s^{i,j} = (s_i, \ldots, s_j)$. With this notation, we have

$$\begin{aligned}
&\int\limits_{\sum_{i=1}^{n+1} s_i \leq 1} f^{(n+1)}\left(x_0 + \sum_{i=1}^{n+1} s_i(x_i - x_0)\right) ds^{1,n+1} \\
&= \int\limits_{\sum_{i=1}^{n} s_i \leq 1} \int\limits_{s_{n+1}=0}^{1-\sum_{i=1}^{n} s_i} f^{(n+1)}\left(x_0 + \sum_{i=1}^{n} s_i(x_i - x_0) + s_{n+1}(x_{n+1} - x_0)\right) ds_{n+1}\, ds^{1,n} \\
&= \frac{1}{x_{n+1} - x_0} \int\limits_{\sum_{i=1}^{n} s_i \leq 1} \left[f^{(n)}\left(x_{n+1} + \sum_{i=1}^{n} s_i(x_i - x_{n+1})\right)\right. \\
&\qquad\qquad \left. - f^{(n)}\left(x_0 + \sum_{i=1}^{n} s_i(x_i - x_0)\right)\right] ds^{1,n} \\
&= \frac{1}{x_{n+1} - x_0}\Big(f_{[x_1,\ldots,x_{n+1}]} - f_{[x_0,\ldots,x_n]}\Big) \;=\; f_{[x_0,\ldots,x_{n+1}]}
\end{aligned}$$

where we used the induction assumption in the penultimate step and the definition of divided differences in the last step. □

This formula can be used as an alternative definition of divided differences. Note that the integral expression remains unchanged upon reordering the support points, so no ascending sorting is required in this alternative definition.

We state two consequences of Lemma 3.15.

Corollary 3.16 For $f \in C^n[a,b]$, the following statements hold.

(i) The function $g : \mathbb{R}^{n+1} \to \mathbb{R}$ defined by

$$g(x_0, \ldots, x_n) = f_{[x_0,\ldots,x_n]}$$

is continuous.

(ii) There exists a $\xi \in [a,b]$ such that

$$f_{[x_0,\ldots,x_n]} = \frac{f^{(n)}(\xi)}{n!}.$$

Proof: (i) Continuity follows directly from the integral representation in Lemma 3.15.

(ii) From the integral representation in Lemma 3.15 and the mean value theorem of integral calculus, we have the equation

$$f_{[x_0,\ldots,x_n]} = \int_{\Sigma^n} f^{(n)}\left(x_0 + \sum_{i=1}^{n} s_i(x_i - x_0)\right) ds = f^{(n)}(\xi) \int_{\Sigma^n} 1\, ds$$

for some suitable $\xi \in [a,b]$. This implies the claim using (3.9). □

Now it is easy to estimate the interpolation error.

Theorem 3.17 Let f be $(n+1)$-times continuously differentiable, and let P be the Hermite interpolation polynomial for the support points $x_0, \ldots, x_n$. The following statements hold:

(i) For all $x \in [a,b]$, there exists $\xi \in [a,b]$ such that the equation

$$f(x) - P(x) = f_{[x_0,\ldots,x_n,x]} \prod_{j=0}^{n} (x - x_j) = \frac{f^{(n+1)}(\xi)}{(n+1)!} \prod_{j=0}^{n} (x - x_j)$$

holds.

(ii) For all $x \in [a,b]$, the inequality

$$|f(x) - P(x)| \leq \|f^{(n+1)}\|_\infty \left| \frac{\prod_{j=0}^{n}(x - x_j)}{(n+1)!} \right|$$

holds.

(iii) The following inequality holds:

$$\|f - P\|_\infty \leq \|f^{(n+1)}\|_\infty \frac{(b-a)^{n+1}}{(n+1)!}.$$

□

Proof: The equations in (i) follow directly from Lemma 3.14 and Corollary 3.16(ii). The inequalities (ii) and (iii) follow from the definition of the maximum norm. □

Remark 3.18 In the case where all x_i coincide, from Theorem 3.17(i) and (3.8), we obtain the equation

$$f(x) - \sum_{j=0}^{n} \frac{(x - x_0)^j}{j!} f^{(j)}(x_0) = \frac{f^{(n+1)}(\xi)}{(n+1)!} (x - x_0)^{n+1},$$

which is precisely the Lagrange remainder of the Taylor expansion. □

We illustrate Theorem 3.17 with two examples.

Example 3.19 Consider the function $f(x) = \sin(x)$ on the interval $[0, 2\pi]$. The derivatives of f are

$$f^{(1)}(x) = \cos(x), \quad f^{(2)}(x) = -\sin(x), \quad f^{(3)}(x) = -\cos(x), \quad f^{(4)}(x) = \sin(x), \ldots$$

For all these functions, $|f^{(k)}(x)| \leq 1$ for all $x \in \mathbb{R}$. With equidistant support points $x_i = 2\pi i/n$, we obtain the estimate

$$|f(x) - P(x)| \leq \max_{y \in [a,b]} \left| f^{(n+1)}(y) \right| \frac{(b-a)^{n+1}}{(n+1)!} \leq \frac{(2\pi)^{n+1}}{(n+1)!}.$$

This term converges very quickly to 0 for increasing n, so even for small n, we can expect a very good match between the functions. □

Example 3.20 Consider the so-called *Runge function* $f(x) = 1/(1+x^2)$ on the interval $[-5, 5]$. The exact derivatives lead to quite complicated terms, but you can verify that for even n, we have

$$\max_{y\in[a,b]} |f^{(n)}(y)| = |f^{(n)}(0)| = n!$$

and for odd n, at least approximately,

$$\max_{y\in[a,b]} |f^{(n)}(y)| \approx n!.$$

Thus, we obtain the estimate

$$|f(x) - P(x)| \leq \max_{y\in[a,b]} \left|f^{(n+1)}(y)\right| \frac{(b-a)^{n+1}}{(n+1)!} \approx (n+1)! \frac{(b-a)^{n+1}}{(n+1)!} = 10^{n+1}.$$

This term grows infinitely for large n, which means that the estimate does not provide a useful error bound. Indeed, for this function and equidistant support points, significant problems arise in numerical tests, with strong oscillations of the interpolating polynomial observed for large n, despite the function being well-behaved. □

Therefore, there are two reasons why polynomial interpolation with equidistant support points is problematic: Firstly, for unsuitable functions, the interpolation error can be large regardless of the number of support points. Secondly, numerically generated interpolation polynomials tend to exhibit strong oscillations for a large number of support points due to their extremely poor conditioning, even if the function to be interpolated is well-behaved.

In the following sections, we will explore other interpolation methods that circumvent these problems, either by using better-positioned support points or by avoiding high-degree polynomials.

3.3 Function Interpolation and Orthogonal Polynomials

In this section, we focus specifically on the issue of function interpolation 3.2 using polynomials. As mentioned earlier, this differs algorithmically from data interpolation 3.1 in that we can freely choose the support points x_i. This leads to the question of how one can optimally choose these support points for a given interpolation interval $[a, b]$. We want to solve this problem *without* assuming knowledge of the function f to be interpolated.

3.3.1 Orthogonal Polynomials

A crucial tool for this purpose is *orthogonal polynomials*, which we want to examine first. Orthogonality is always defined with respect to a scalar product, and here, we use the following scalar product.

Definition 3.21 Let $\omega : (a, b) \to \mathbb{R}^+$ be a positive and Lebesgue-integrable *weight function*[2]. Then, we define the scalar product on the space $\mathcal{P}$ of polynomials as follows:

$$\langle P_1, P_2\rangle_\omega := \int_a^b \omega(x) P_1(x) P_2(x) dx$$

[2]Note that ω may not be defined at the boundary points a and b when using the Lebesgue integral.

for P_1, $P_2 \in \mathcal{P}$. We denote the corresponding norm as

$$\|P\|_\omega := \sqrt{\langle P,\, P\rangle_\omega} = \sqrt{\int_a^b \omega(x)(P(x))^2 dx}$$

□

With this scalar product, we can now define orthogonality.

Definition 3.22 A sequence $(P_k)_{k\in\mathbb{N}_0}$ of polynomials with $P_k \in \mathcal{P}_k$ *exactly* of degree k (i.e., with a leading coefficient $\neq 0$) is called *orthogonal* with respect to ω if

$$\langle P_i,\, P_j\rangle_\omega = 0 \text{ for } i \neq j \text{ and } \langle P_i,\, P_i\rangle_\omega = \|P_i\|_\omega^2 =: \gamma_i > 0$$

holds. □

The following theorem shows that orthogonal polynomials always exist and can be calculated using a simple recursive formula.

Theorem 3.23 For any weight function $\omega : [a, b] \to \mathbb{R}^+$, there are uniquely determined orthogonal polynomials $(P_k)_{k\in\mathbb{N}_0}$ according to Definition 3.22 with a leading coefficient of 1. They satisfy the *recursive equation*

$$P_k(x) = (x + b_k)P_{k-1}(x) + c_k P_{k-2}(x), \quad k = 1, 2, \ldots$$

with initial values $P_{-1} \equiv 0$, $P_0 \equiv 1$, and coefficients

$$b_k = -\frac{\langle xP_{k-1},\, P_{k-1}\rangle_\omega}{\langle P_{k-1},\, P_{k-1}\rangle_\omega}, \quad k = 1, 2, \ldots, \quad c_1 = 0 \text{ and } c_k = -\frac{\langle P_{k-1},\, P_{k-1}\rangle_\omega}{\langle P_{k-2},\, P_{k-2}\rangle_\omega}, \quad k = 2, 3, \ldots.$$

Proof: We prove the theorem by induction on k, starting with $k = 1$. For this case, we have

$$b_1 = -\frac{\int_a^b x\omega(x)dx}{\int_a^b \omega(x)dx}$$

and, as a result of $P_1(x) = x - b_1$, the equation

$$\langle P_1, P_0\rangle_\omega = \langle x + b_1, P_0\rangle_\omega = \int_a^b x\omega(x)dx - \int_a^b \omega(x)dx \frac{\int_a^b x\omega(x)dx}{\int_a^b \omega(x)dx} = 0.$$

Therefore, P_1 satisfies the orthogonality property with respect to P_0 and has a leading coefficient of 1. Since no other value of b_1 satisfies the above equation, P_1 is uniquely determined.

For the induction step $k - 1 \to k$, assume that $P_0, P_1, \ldots, P_{k-1}$ are the first k orthogonal polynomials that satisfy the given recursive equation. Take any $P_k \in \mathcal{P}_k$ with a leading coefficient of 1. Since $P_{k-1} \in \mathcal{P}_{k-1}$ and the leading coefficients cancel out, we have $P_k -$

$xP_{k-1} \in \mathcal{P}_{k-1}$. Since $P_0, P_1, \ldots, P_{k-1}$ are linearly independent due to orthogonality, they form a basis for $\mathcal{P}_{k-1}$, even an orthogonal basis with respect to $\langle \cdot, \cdot \rangle_\omega$. Thus, the equation holds

$$P_k - xP_{k-1} = \sum_{j=0}^{k-1} d_j P_j \text{ with } d_j = \frac{\langle P_k - xP_{k-1},\, P_j \rangle_\omega}{\langle P_j,\, P_j \rangle_\omega}.$$

We now want to determine conditions on the coefficients d_j under the assumption that P_k is orthogonal to P_j for $j = 0, \ldots, k-1$. If P_k possesses this orthogonality property, then it must necessarily hold that

$$d_j = -\frac{\langle xP_{k-1},\, P_j \rangle_\omega}{\langle P_j,\, P_j \rangle_\omega} = -\frac{\langle P_{k-1},\, xP_j \rangle_\omega}{\langle P_j,\, P_j \rangle_\omega}.$$

For $j = 0, \ldots, k-3$, $xP_j \in \mathcal{P}_{k-2}$, so it can be expressed as a linear combination of $P_0, \ldots, P_{k-2}$, which means that $\langle P_{k-1},\, xP_j \rangle_\omega = 0$, and therefore, $d_0 = d_1 = \ldots = d_{k-3} = 0$ must hold. For the remaining coefficients d_{k-1} and d_{k-2}, it must be true that

$$d_{k-1} = -\frac{\langle P_{k-1},\, xP_{k-1} \rangle_\omega}{\langle P_{k-1},\, P_{k-1} \rangle_\omega}, \quad d_{k-2} = -\frac{\langle P_{k-1},\, xP_{k-2} \rangle_\omega}{\langle P_{k-2},\, P_{k-2} \rangle_\omega} = -\frac{\langle P_{k-1},\, P_{k-1} \rangle_\omega}{\langle P_{k-2},\, P_{k-2} \rangle_\omega},$$

where the last equation follows from the induction assumption

$$P_{k-1}(x) = (x + b_{k-1})P_{k-2}(x) + c_{k-1}P_{k-3}(x)$$

and the fact that

$$\langle P_{k-1},\, P_{k-1} \rangle_\omega = \langle P_{k-1},\, xP_{k-2} \rangle_\omega + \underbrace{\langle P_{k-1},\, b_{k-1}P_{k-2} + c_{k-1}P_{k-3} \rangle_\omega}_{= 0 \text{ due to orthogonality}}$$

Hence, we obtain

$$P_k = xP_{k-1} + d_{k-1}P_{k-1} + d_{k-2}P_{k-2} = (x + b_k)P_{k-1} + c_k P_{k-2}.$$

Since the coefficients b_k and c_k and the polynomial P_k are uniquely determined by this equation, the claim follows. □

Remark 3.24 For numerical purposes, the recursive formula provided here is generally not suitable, as its evaluation is numerically unstable, i.e., susceptible to rounding errors. Numerically stable algorithms for computing orthogonal polynomials are discussed, for example, in Chapter 6 of the book by Deuflhard/Hohmann [1]. □

For the construction of orthogonal polynomials, Theorem 3.23 allows us to start with a recursive formula for given coefficients and then identify the corresponding scalar product $\langle \cdot, \cdot \rangle_\omega$. It is worth noting that not every recursive formula automatically generates polynomials with a leading coefficient of 1. However, you can easily obtain these normalized polynomials through appropriate scaling.

Now, we will examine a specific recursive formula and the resulting *Chebyshev Polynomials*, which play an important role in interpolation. In the chapter on numerical integration, we will encounter other families of orthogonal polynomials.

The crucial recursive formula for interpolation is given by

$$T_k(x) = 2xT_{k-1}(x) - T_{k-2}(x). \tag{3.10}$$

with initial values $T_0(x) = 1$ and $T_1(x) = x$ (we scale slightly differently here than in Satz 3.23, which simplifies some calculations below). The following theorem summarizes the properties of the corresponding orthogonal polynomials, known as *Chebyshev Polynomials*.

Theorem 3.25 For the Chebyshev Polynomials T_k, $k = 0, 1, 2, \ldots$, given by the recursion (3.10), the following statements hold.

(i) For $x \in [-1, 1]$ the polynomials are given by $T_k(x) = \cos(k \arccos(x))$.

(ii) For $k \geq 1$, the T_k have the form

$$T_k(x) = 2^{k-1}x^k + \sum_{i=0}^{k-1} t_{k,i}x^i$$

(iii) Each T_k has exactly k zeros

$$x_{k,l} = \cos\left(\frac{2l-1}{2k}\pi\right) \text{ for } l = 1, \ldots, k.$$

These are called *Chebyshev Nodes*.

(iv) The T_k are orthogonal with respect to the weight function defined on $(-1, 1)$

$$\omega(x) = \frac{1}{\sqrt{1-x^2}}.$$

More precisely, it holds that

$$\langle T_i, T_j\rangle_\omega = \begin{cases} 0, & \text{if } i \neq j \\ \pi, & \text{if } i = j = 0 \\ \pi/2, & \text{if } i = j > 0 \end{cases}$$

□

Proof: (i)–(iii) can be verified by calculation, as seen in the current exercise sheet.

(iv): We consider the scalar product

$$\langle T_i, T_j\rangle_\omega = \int_{-1}^{1} \omega(x)T_i(x)T_j(x)dx$$

To solve this integral, we use the substitution $x = \cos(\alpha)$, so $dx = -\sin(\alpha)d\alpha$, and the representation of T_k from (i). Thus, we have

$$\begin{aligned} \int_{-1}^{1} \omega(x)T_i(x)T_j(x)dx &= \int_{\pi}^{0} \underbrace{\frac{1}{\sqrt{1-\cos^2(\alpha)}}}_{=\sin(\alpha)} \cos(i\alpha)\cos(j\alpha)(-\sin(\alpha))d\alpha \\ &= \int_0^{\pi} \cos(i\alpha)\cos(j\alpha)d\alpha = \frac{1}{2}\int_{-\pi}^{\pi} \cos(i\alpha)\cos(j\alpha)d\alpha \\ &= \frac{1}{4}\int_{-\pi}^{\pi} \cos((i+j)\alpha) + \cos((i-j)\alpha)d\alpha, \end{aligned}$$

where we used $\cos(a) = \cos(-a)$ in the penultimate step, and in the last step, we employed the equation $2\cos(a)\cos(b) = \cos(a+b) + \cos(a-b)$.

For $i \neq j$, it follows from $\sin(k\pi) = 0$ for $k \in \mathbb{Z}$ that

$$\langle T_i, T_j \rangle_\omega = \frac{1}{4}\left[\frac{1}{i+j}\sin((i+j)\alpha) + \frac{1}{i-j}\sin((i-j)\alpha)\right]_{-\pi}^{\pi} = 0,$$

for $i = j = 0$, we have

$$\langle T_0, T_0 \rangle_\omega = \frac{1}{4}\int_{-\pi}^{\pi} \cos(0) + \cos(0)d\alpha = \frac{1}{4}\Big[2\alpha\Big]_{-\pi}^{\pi} = \pi$$

and for $i = j > 0$, again using $\sin(k\pi) = 0$, we get

$$\langle T_i, T_i \rangle_\omega = \frac{1}{4}\int_{-\pi}^{\pi} \cos(2i\alpha) + \cos(0)d\alpha = \frac{1}{4}\left[\frac{1}{2i}\sin(2i\alpha) + \alpha\right]_{-\pi}^{\pi} = \frac{\pi}{2}$$

□

The significance of the Chebyshev Polynomials for function interpolation arises from the error estimation in Theorem 3.17(ii). There, we proved the inequality

$$|f(x) - P(x)| \leq \|f^{(n+1)}\|_\infty \left|\frac{(x-x_0)(x-x_1)\cdots(x-x_n)}{(n+1)!}\right|. \tag{3.11}$$

for $x \in [a,b]$. Now, we want to investigate how to choose the support points x_i so that this error bound becomes minimal. Since we do not want to specify a particular x (the estimate should be optimal for all x, i.e., for $\|f - P\|_\infty$), the task is to find support points $x_0, \ldots, x_n$ such that the expression

$$\max_{x\in[a,b]} |(x-x_0)(x-x_1)\cdots(x-x_n)| \tag{3.12}$$

becomes minimal.

Without loss of generality, we consider the interval $[a,b] = [-1,1]$ because if we find the optimal support points x_i on $[-1,1]$, the support points defined by $\tilde{x}_i = a + (x_i+1)(b-a)/2$ will also be optimal on $[a,b]$, and it holds that

$$\max_{x\in[a,b]} |(x-\tilde{x}_0)(x-\tilde{x}_1)\cdots(x-\tilde{x}_n)| = \left(\frac{b-a}{2}\right)^{n+1} \max_{x\in[-1,1]} |(x-x_0)(x-x_1)\cdots(x-x_n)|.$$

Now, for arbitrary support points $x_0, \ldots, x_n$, let's define the polynomial $R_{n+1}(x) = (x - x_0)(x-x_1)\cdots(x-x_n)$. This defines a polynomial with a leading coefficient of $a_{n+1} = 1$. The support points x_i are precisely the zeros of R_{n+1}, and the expression (3.12) is precisely the maximum norm $\|R_{n+1}\|_\infty$. Minimizing (3.12) is thus equivalent to the following problem: Among all polynomials R_{n+1} of degree $n+1$ with a leading coefficient of $a_{n+1} = 1$, find the one with the smallest maximum norm on $[-1,1]$.

The following theorem shows that the normalized Chebyshev polynomial $T_{n+1}/2^n$ is precisely the polynomial we are looking for.

Theorem 3.26 Let $\|\cdot\|_\infty$ be the maximum norm on $[-1,1]$. Then, for any polynomial R_{n+1} of the form $R_{n+1}(x) = (x-x_0)(x-x_1)\cdots(x-x_n)$ with pairwise distinct zeros x_i in the interval $[-1,1]$, the inequality

$$\|T_{n+1}/2^n\|_\infty \le \|R_{n+1}\|_\infty.$$

holds. In particular, the Chebyshev nodes

$$x_i = \cos\left(\frac{2i+1}{2n+2}\pi\right), \quad i = 0,\dots,n,$$

which are the zeros of T_{n+1}, minimize the expression (3.12) and thus the error estimate (3.11). □

Proof: From the representation $T_{n+1}(x) = \cos((n+1)\arccos(x))$, it immediately follows that $\|T_{n+1}\|_\infty \le 1$. Furthermore, we have

$$\begin{aligned} |T_{n+1}(x)| = 1 &\Leftrightarrow (n+1)\arccos(x) = m\pi \text{ for some } m \in \mathbb{N}_0 \\ &\Leftrightarrow x = \cos\left(\frac{m}{n+1}\pi\right) \text{ for some } m \in \mathbb{N}_0. \end{aligned}$$

We denote $\bar{x}_m := \cos\left(\frac{m}{n+1}\pi\right)$. Note that this defines exactly $n+2$ values $\bar{x}_0,\dots,\bar{x}_{n+1}$, and $T_{n+1}(\bar{x}_m) = 1$ if m is even or zero, and $T_{n+1}(\bar{x}_m) = -1$ if m is odd.

Now, we prove that for any polynomial Q_{n+1} of degree $n+1$ with leading coefficient $a_{n+1} = 2^n$, the inequality

$$\|Q_{n+1}\|_\infty \ge \|T_{n+1}\|_\infty \tag{3.13}$$

holds.

To prove this inequality, we assume the opposite, i.e., $\|Q_{n+1}\|_\infty < \|T_{n+1}\|_\infty$, and consider the difference $Q_{n+1} - T_{n+1}$. Since the leading coefficients cancel out, this is a polynomial of degree $\le n$. At the $n+2$ points $\bar{x}_m$, we have:

$$\begin{aligned} &\text{If } m \text{ is even or } 0: \quad T_{n+1}(\bar{x}_m) = 1, \quad Q_{n+1}(\bar{x}_m) < 1 \quad \Rightarrow \quad Q_{n+1}(\bar{x}_m) - T_{n+1}(\bar{x}_m) < 0 \\ &\text{If } m \text{ is odd}: \quad T_{n+1}(\bar{x}_m) = -1, \quad Q_{n+1}(\bar{x}_m) > -1 \quad \Rightarrow \quad Q_{n+1}(\bar{x}_m) - T_{n+1}(\bar{x}_m) > 0 \end{aligned}$$

Thus, $Q_{n+1} - T_{n+1}$ changes sign in each of the $n+1$ intervals $[\bar{x}_i, \bar{x}_{i+1}]$, $i = 0,\dots,n$, and therefore has (at least) $n+1$ roots. This is only possible for a polynomial of degree n if it is identically zero. So, $Q_{n+1} - T_{n+1} \equiv 0$, which implies $Q_{n+1} = T_{n+1}$, contradicting the assumption $\|Q_{n+1}\|_\infty < \|T_{n+1}\|_\infty$.

The result follows immediately by scaling T_{n+1} and Q_{n+1} by $1/2^n$, as every polynomial of the form stated in the theorem can be written as $R_{n+1} = Q_{n+1}/2^n$ for some Q_{n+1} satisfying (3.13). □

For the Chebyshev nodes x_i, we obtain in (3.12):

$$\max_{x\in[-1,1]} |(x-x_0)(x-x_1)\cdots(x-x_n)| = \frac{1}{2^n}.$$

For general intervals $[a, b]$, the transformed support points given by

$$\tilde{x}_i = a + (x_i + 1)\,(b - a)/2$$

lead to the following expression:

$$\max_{x\in[a,b]} |(x - \tilde{x}_0)(x - \tilde{x}_1)\cdots(x - \tilde{x}_n)| = \frac{(b-a)^{n+1}}{2^{2n+1}} = 2\left(\frac{b-a}{4}\right)^{n+1}.$$

For the Runge function from Example 3.20, we obtain from Theorem 3.17(ii) on the interpolation interval $[-5, 5]$:

$$\|f - P\|_\infty \le 2\left(\frac{5}{2}\right)^{n+1}.$$

Once again, we get an error bound that diverges as $n \to \infty$. Surprisingly, interpolation of the Runge function with Chebyshev nodes still works, as demonstrated in Exercise 34. It appears that the inequality underlying this result, $|f^{(n+1)}(\xi)| \le \|f^{(n+1)}\|_\infty$, is too pessimistic in this case.

Note that the endpoints -1 and 1 of the interpolation interval are not Chebyshev nodes and, therefore, not support points. Hence, this method uses interpolation outside the interval defined by the support points.

3.4 Spline Interpolation

We have seen that polynomial interpolation can be problematic from a conditioning perspective when we have many support points that are not optimally chosen like Chebyshev nodes. This can particularly occur in data interpolation (3.1) when the support points are fixed and cannot be freely chosen. Therefore, in this section, we will explore an alternative interpolation technique that works smoothly even with a large number of support points.

The basic idea of *Spline Interpolation* is to choose the interpolating function (denoted as "S" for "Spline" here) not globally but only on each subinterval $[x_i, x_{i+1}]$ as a polynomial. These sub-polynomials should smoothly merge at the interval boundaries. Such a function, composed of smoothly merged piecewise polynomials, is called a *Spline*.

Formally, this concept is defined as follows:

Definition 3.27 Let $x_0 < x_1 < \ldots < x_n$ be support points and $k \in \mathbb{N}$. A continuous and $(k-1)$-times continuously differentiable function $S : [x_0, x_n] \to \mathbb{R}$ is called a *Spline of degree* k if, on each interval $I_i = [x_{i-1}, x_i]$ for $i = 1, \ldots, n$, it is represented by a polynomial P_i of degree $\le k$, i.e., for $x \in I_i$, we have:

$$S(x) = P_i(x) = a_{i\,0} + a_{i\,1}(x - x_{i-1}) + \ldots + a_{i\,k}(x - x_{i-1})^k = \sum_{j=0}^{k} a_{i\,j}(x - x_{i-1})^j.$$

The space of Splines of degree k with support point set $\Delta = \{x_0, x_1, \ldots, x_n\}$ is denoted as $S_{\Delta,k}$. □

Such a Spline, as defined in Definition 3.27, solves the interpolation problem if the additional condition (3.1) is satisfied, i.e., $S(x_i) = f_i$ for all $i = 0, \ldots, n$.

Before we delve into the computation of Splines, we establish a property of the function space $S_{\Delta,k}$.

Theorem 3.28 Let $\Delta = \{x_0, x_1, \ldots, x_n\}$ with $x_0 < x_1 < \ldots < x_n$, and let $k \in \mathbb{N}$ be given. Then, the space of Splines $S_{\Delta,k}$ is a $k+n$-dimensional vector space over $\mathbb{R}$. □

Proof: Clearly, $aS_1 + bS_2$ is also a Spline for $a, b \in \mathbb{R}$, making $S_{\Delta,k}$ a vector space. Since Splines depend linearly on the coefficients $a_{i\,j}$ and the functions $(x - x_{i-1})^j$ within each restricted interval $[x_{i-1}, x_i]$ are all linearly independent, it is sufficient to determine the number of free parameters by calculating the dimension. On the first interval I_1, P_1 can be freely chosen, giving $k+1$ free parameters. On each subsequent interval I_i for $i \geq 2$, the values of the j-th derivative $P_i^{(j)}(x_{i-1}) = j! a_{i\,j}$ for $j = 0, \ldots, k-1$ are already fixed, as the composite function S is continuous and $k-1$ times continuously differentiable at x_{i-1}. Hence, it must hold that $a_{i\,j} = P_{i-1}^{(j)}(x_{i-1})/j!$ for $j = 0, \ldots, k-1$. Therefore, only the coefficient $a_{i\,k}$ remains free on each of the remaining $n-1$ intervals, resulting in a total of $k+n$ free parameters. □

Remark 3.29 Instead of using the coefficients as in the proof, any other set of $n+k$ coefficients can be fixed, provided they satisfy the following condition: For any assignment of the fixed coefficients, the remaining coefficients can be chosen in such a way that the continuity and differentiability conditions are met. For each assignment of these $n+k$ coefficients, there exists exactly one Spline whose coefficients match the fixed ones. □

In many applications, *cubic splines* (Spline of degree $k = 3$) are of particular importance. We will discuss the reasons for this later. First, let us focus on the existence and uniqueness of the interpolating cubic spline $S \in S_{\Delta,3}$ for given data (x_i, f_i) with $x_0 < x_1 < \ldots < x_n$. Clearly, the spline satisfies the interpolation problem (3.1) if the conditions

$$\begin{aligned} &a_{i\,0} = f_{i-1} \text{ for } i = 1, \ldots, n \\ &\text{and} \\ &a_{n\,1} = \frac{f_n - a_{n\,0} - a_{n\,2}(x_n - x_{n-1})^2 - a_{n\,3}(x_n - x_{n-1})^3}{x_n - x_{n-1}} \end{aligned} \tag{3.14}$$

are met. This determines $n+1$ coefficients. As $\dim S_{\Delta,k} = n+3$, we have two additional coefficients that need to be fixed, typically in the form of boundary conditions, i.e. conditions on S or its derivatives at x_0 and x_n. The following lemma presents three possible boundary conditions.

Lemma 3.30 Given data (x_i, f_i) for $i = 0, \ldots, n$ with $x_0 < x_1 < \ldots < x_n$ and $\Delta = \{x_0, x_1, \ldots, x_n\}$, for each of the following *boundary conditions*

(a) $S''(x_0) = S''(x_n) = 0$ ("natural boundary conditions")

(b) $S'(x_0) = S'(x_n)$ and $S''(x_0) = S''(x_n)$ ("periodic boundary conditions")

(c) $S'(x_0) = f'(x_0)$ and $S'(x_n) = f'(x_n)$ ("Hermite boundary conditions," only meaningful for function interpolation)

there exists exactly one cubic spline $S \in S_{\Delta,3}$ that solves the interpolation problem and satisfies the corresponding boundary condition (a), (b), or (c).

Proof: Each of the provided boundary conditions can be expressed as conditions on two more coefficients of the spline. Therefore, together with (3.14), a total of $n+3$ coefficients are determined. As per Remark 3.29, exactly one spline satisfying these conditions exists, provided the remaining coefficients can be chosen to form a valid spline. □

Cubic splines are commonly used in applications such as computer graphics, and we want to explain the reason for this preference. One criterion for choosing the order of a spline, especially in graphical applications but also in "classical" interpolation problems, is that the curvature of the interpolating curve should be as small as possible. The curvature of a curve $y(x)$ at a point x is given by its second derivative $y''(x)$. The total curvature for all x in $[x_0, x_n]$ can be measured in various ways; here, we use the L_2 norm $\|\cdot\|_2$ for square-integrable functions, which is defined for $g : [x_0, x_n] \to \mathbb{R}$ as

$$\|g\|_2 := \left(\int_{x_0}^{x_n} g^2(x) dx \right)^{\frac{1}{2}}$$

The curvature of a twice continuously differentiable function $y : [x_0, x_n] \to \mathbb{R}$ over the entire interval can thus be measured using $\|y''\|_2$. The following theorem holds for this:

Theorem 3.31 Let $S : [x_0, x_n] \to \mathbb{R}$ be a cubic spline interpolating the data (x_i, f_i) for $i = 0, \ldots, n$, satisfying one of the boundary conditions (a)-(c) from Lemma 3.30. Let $y : [x_0, x_n] \to \mathbb{R}$ be a twice continuously differentiable function that also solves the interpolation problem and satisfies the same boundary conditions as S. Then, we have:

$$\|S''\|_2 \leq \|y''\|_2.$$

□

Proof: Substituting the obvious equation $y'' = S'' + (y'' - S'')$ into the squared norm $\|y''\|_2^2$, we get:

$$\begin{aligned}
\|y''\|_2^2 &= \int_{x_0}^{x_n} (y''(x))^2 dx \\
&= \underbrace{\int_{x_0}^{x_n} (S''(x))^2 dx}_{=\|S''\|_2^2} + \underbrace{2 \int_{x_0}^{x_n} S''(x)(y''(x) - S''(x)) dx}_{=:J} + \underbrace{\int_{x_0}^{x_n} (y''(x) - S''(x))^2 dx}_{\geq 0} \\
&\geq \|S''\|_2^2 + J.
\end{aligned}$$

Now we examine the term J. From each of the three boundary conditions, we obtain the equation:

$$\Big[S''(x)(y'(x)-S'(x))\Big]_{x=x_0}^{x_n} = S''(x_n)(y'(x_n)-S'(x_n))-S''(x_0)(y'(x_0)-S'(x_0)) = 0. \quad (3.15)$$

Using integration by parts, we have:

$$\int_{x_0}^{x_n} S''(x)(y''(x)-S''(x))dx = \big[S''(x)(y'(x)-S'(x))\big]_{x=x_0}^{x_n} - \int_{x_0}^{x_n} S'''(x)(y'(x)-S'(x))dx.$$

The first term is zero due to the earlier equation. On each interval $I_i = [x_{i-1}, x_i]$, $S(x) = P_i(x)$ is a cubic polynomial, so $S'''(x) \equiv d_i$ is constant for $x \in I_i$. Therefore, the second term satisfies

$$\begin{aligned}
\int_{x_0}^{x_n} S'''(x)(y'(x)-S'(x))dx &= \sum_{i=1}^{n} \int_{x_{i-1}}^{x_i} d_i(y'(x)-S'(x))dx \\
&= \sum_{i=1}^{n} d_i \int_{x_{i-1}}^{x_i} y'(x)-S'(x)dx \\
&= \sum_{i=1}^{n} d_i \underbrace{[(y(x_i)-y(x_{i-1})-S(x_i)+S(x_{i-1})]}_{=0,\text{ since } y(x_i)=S(x_i) \text{ and } y(x_{i-1})=S(x_{i-1})} = 0.
\end{aligned}$$

Thus, we have $J = 0$, and consequently, the claim is proven. □

This property also explains the name "Spline". Literally, a "spline" is a thin wooden strip. If you bend it to follow given points (i.e., "interpolate" them), the curvature, which approximately describes the necessary bending energy, is minimal — at least for small deflections of the strip.

We now move on to the practical computation of the spline coefficients for cubic splines. There are various approaches to this. For example, you can directly set up a linear system of equations for the $4n$ coefficients a_{ij} for $i = 1, \dots, n$, but this is not very efficient. Alternatively, you can use cleverly chosen basis functions for the vector space $S_{\Delta,3}$ (known as B-splines) and compute S in this basis. This approach is described in the book by Deuflhard/Hohmann. It leads to an n-dimensional linear system with a tridiagonal matrix A. However, in this method, the coefficients a_{ij} are not calculated; instead, the coefficients with respect to the B-spline basis are computed, and the evaluation of S must also be done using these basis functions.

Here, we present another approach where the coefficients a_{ij} are calculated directly so that S can be evaluated using the representation in Definition 3.27, similar to how Splines are implemented in MATLAB. In this case, we also obtain an n-dimensional linear system with a tridiagonal matrix A, resulting in a computational complexity of $O(n)$. We first consider the natural boundary conditions.

To do this, we first define the values:

$$f_i'' := S''(x_i) \quad \text{and} \quad h_i := x_i - x_{i-1}$$

for $i = 0, \ldots, n$ and $i = 1, \ldots, n$, respectively. From the interpolation condition and the requirement for the second derivative to be continuous, we obtain four equations for the sub-polynomials P_i:

$$P_i(x_{i-1}) = f_{i-1}, \quad P_i(x_i) = f_i, \quad P_i''(x_{i-1}) = f_{i-1}'', \quad P_i''(x_i) = f_i''. \tag{3.16}$$

Solving these equations, utilizing the differentiation rules for polynomials, for the coefficients $a_{i\,j}$, we obtain:

$$\begin{aligned} a_{i\,0} &= f_{i-1} \\ a_{i\,1} &= \frac{f_i - f_{i-1}}{h_i} - \frac{h_i}{6}(f_i'' + 2f_{i-1}'') \\ a_{i\,2} &= \frac{f_{i-1}''}{2} \\ a_{i\,3} &= \frac{f_i'' - f_{i-1}''}{6h_i}. \end{aligned}$$

Since the values h_i and f_i are directly available from the data, we only need to calculate the values of f_i''. Since the natural boundary conditions immediately yield $f_0'' = 0$ and $f_n'' = 0$, we only need to calculate the values of $f_1'', \ldots, f_{n-1}''$.

Note that in (3.16), we have already used the conditions on P_i and P_i'' at the support points. From the equations for the first derivatives that have not been used yet, we can now obtain equations for f_i''. From $P_i'(x_i) = P_{i+1}'(x_i)$, we get:

$$a_{i\,1} + 2a_{i\,2}(x_i - x_{i-1}) + 3a_{i\,3}(x_i - x_{i-1})^2 = a_{i+1\,1}$$

for $i = 1, \ldots, n-1$. By substituting the values of f_i'' and h_i according to the above equations and definitions, we obtain:

$$h_i f_{i-1}'' + 2(h_i + h_{i+1})f_i'' + h_{i+1}f_{i+1}'' = 6\frac{f_{i+1} - f_i}{h_{i+1}} - 6\frac{f_i - f_{i-1}}{h_i} =: \delta_i$$

for $i = 1, \ldots, n-1$. This provides exactly $n-1$ equations for the $n-1$ unknowns $f_1'', \ldots, f_{n-1}''$. In matrix form, this gives us the equation system:

$$\begin{pmatrix} 2(h_1+h_2) & h_2 & 0 & \cdots & \cdots & 0 \\ h_2 & 2(h_2+h_3) & h_3 & \ddots & & \vdots \\ 0 & h_3 & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & 0 \\ \vdots & & \ddots & \ddots & \ddots & h_{n-1} \\ 0 & \cdots & \cdots & 0 & h_{n-1} & 2(h_{n-1}+h_n) \end{pmatrix} \begin{pmatrix} f_1'' \\ f_2'' \\ \vdots \\ \vdots \\ f_{n-1}'' \end{pmatrix} = \begin{pmatrix} \delta_1 \\ \delta_2 \\ \vdots \\ \vdots \\ \delta_{n-1} \end{pmatrix}$$

To compute the interpolation spline, we first solve this equation system and then calculate the coefficients $a_{i\,j}$ from the f_k'' using the above formula.

For equidistant support points, i.e., $x_k - x_{k-1} = h_k = h$ for all $k = 1, \ldots, n$, both sides can be divided by h, resulting in the equation system:

$$\begin{pmatrix} 4 & 1 & 0 & \cdots & \cdots & 0 \\ 1 & 4 & 1 & \ddots & & \vdots \\ 0 & 1 & 4 & 1 & \ddots & \vdots \\ \vdots & \ddots & 1 & \ddots & \ddots & \vdots \\ \vdots & & \ddots & \ddots & \ddots & 1 \\ 0 & \cdots & \cdots & 0 & 1 & 4 \end{pmatrix} \begin{pmatrix} f_1'' \\ f_2'' \\ \vdots \\ \vdots \\ f_{n-1}'' \end{pmatrix} = \begin{pmatrix} \tilde{\delta}_1 \\ \tilde{\delta}_2 \\ \vdots \\ \vdots \\ \tilde{\delta}_{n-1} \end{pmatrix}$$

with $\tilde{\delta_k} = \delta_k / h$, which is an example of a linear equation system with a (obviously) diagonally dominant matrix.

For other boundary conditions, this equation system will change accordingly. If, for instance, we want to use the Hermite boundary conditions, the conditions $f_0'' = f_n'' = 0$ are replaced by $S'(x_0) = f_0'$ and $S'(x_n) = f_n'$, which can be expressed in the above coefficients as:

$$\begin{aligned} 2h_1 f_0'' + h_1 f_1'' &= \frac{6}{h_1}(f_1 - f_0) - 6f_0' \\ h_n f_{n-1}'' + 2h_n f_n'' &= -\frac{6}{h_n}(f_n - f_{n-1}) + 6f_n' \end{aligned}$$

These equations should be added as the first and last rows to the above equation system. For equidistant support points, the resulting system will be:

$$\begin{pmatrix} 2 & 1 & 0 & \cdots & \cdots & \cdots & \cdots & 0 \\ 1 & 4 & 1 & \ddots & & & & \vdots \\ 0 & 1 & 4 & 1 & \ddots & & & \vdots \\ \vdots & \ddots & 1 & \ddots & \ddots & \ddots & & \vdots \\ \vdots & & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & & \ddots & \ddots & \ddots & 1 & 0 \\ \vdots & & & & \ddots & 1 & 4 & 1 \\ 0 & \cdots & \cdots & \cdots & \cdots & 0 & 1 & 2 \end{pmatrix} \begin{pmatrix} f_0'' \\ f_1'' \\ \vdots \\ \vdots \\ f_{n-1}'' \\ f_n'' \end{pmatrix} = \begin{pmatrix} \tilde{\delta}_0 \\ \tilde{\delta}_1 \\ \vdots \\ \vdots \\ \tilde{\delta}_{n-1} \\ \tilde{\delta}_n \end{pmatrix}$$

with $\tilde{\delta_k} = \delta_k / h$ for $k = 1, \ldots, n-1$ and

$$\tilde{\delta}_0 = \frac{6}{h^2}(f_1 - f_0) - \frac{6}{h} f_0' \quad \text{and} \quad \tilde{\delta}_n = -\frac{6}{h^2}(f_n - f_{n-1}) + \frac{6}{h} f_n',$$

resulting in a once again diagonally dominant equation system.

Finally, let's briefly discuss the interpolation error in spline interpolation. Analyzing this error is quite lengthy, but the result, which we provide here without proof, is straightforward to state. The following theorem was proven by C.A. Hall and W.W. Meyer [2]:

Theorem 3.32 Let $S \in S_{\Delta,3}$ be the interpolating spline of a function f that is four times continuously differentiable, with Hermite boundary conditions and support points $\Delta = \{x_0, \ldots, x_n\}$. Then, for $h = \max_k(x_k - x_{k-1})$, the error estimate holds:

$$\|f - S\|_\infty \leq \frac{5}{384} h^4 \|f^{(4)}\|_\infty$$

□

3.5 Trigonometric Interpolation and Fourier Transformation

To conclude our chapter on interpolation, we will delve into another class of interpolation functions. The method of *trigonometric interpolation* that we will discuss here has applications that are quite different from pure interpolation; we will outline these applications in Section 3.5.3. But first, let us consider the method as an interpolation problem.

3.5.1 Interpolation with Trigonometric Polynomials

Trigonometric polynomials are defined as sums of sinusoidal and cosine functions, rather than powers of x.

Definition 3.33 A *trigonometric polynomial* of degree N is a complex-valued function $T : \mathbb{R} \to \mathbb{C}$ that for $n = 2N + 1$ is given by

$$T(x) = d_0 + \sum_{k=1}^{N} (d_k + d_{n-k}) \cos\left(k2\pi \frac{x - x_0}{p}\right) + i \sum_{k=1}^{N} (d_k - d_{n-k}) \sin\left(k2\pi \frac{x - x_0}{p}\right).$$

Here, $x_0 \in \mathbb{R}$ and $p > 0$ are given numbers, and $d_0, \ldots, d_{n-1} \in \mathbb{C}$ are the *coefficients* of the trigonometric polynomial. □

Remark 3.34 If we set $d_{-k} := d_{n-k}$ for $k = 1, \ldots, N$, then we can transform the expressions using the identities

$$\cos(-x) = \cos(x) \text{ and } \sin(-x) = -\sin(x),$$

which yields

$$\begin{aligned}
T(x) &= d_0 + \sum_{k=1}^{N} (d_k + d_{-k}) \cos\left(k2\pi \frac{x - x_0}{p}\right) + i \sum_{k=1}^{N} (d_k - d_{-k}) \sin\left(k2\pi \frac{x - x_0}{p}\right) \\
&= d_0 + \sum_{k=1}^{N} d_k \cos\left(k2\pi \frac{x - x_0}{p}\right) + d_{-k} \cos\left(-k2\pi \frac{x - x_0}{p}\right) \\
&\quad + i \sum_{k=1}^{N} d_k \sin\left(k2\pi \frac{x - x_0}{p}\right) + d_{-k} \sin\left(-k2\pi \frac{x - x_0}{p}\right) \\
&= \sum_{k=-N}^{N} d_k \left(\cos\left(k2\pi \frac{x - x_0}{p}\right) + i \sin\left(k2\pi \frac{x - x_0}{p}\right)\right) \\
&= \sum_{k=-N}^{N} d_k e^{ik2\pi \frac{x - x_0}{p}}.
\end{aligned}$$

□

Trigonometric polynomials are usually expressed with complex values. If T is supposed to be real-valued, then the coefficients $d_k = a_k + ib_k$ must satisfy the conditions

$$b_0 = 0, \quad a_k = a_{n-k}, \quad b_k = -b_{n-k} \text{ for all } k = 1, \ldots, N. \tag{3.17}$$

In this case, the trigonometric polynomial with coefficients $d_k = a_k + ib_k$ can be written as

$$T(x) = a_0 + \sum_{k=1}^{N} 2a_k \cos\left(k2\pi \frac{x - x_0}{p}\right) - \sum_{k=1}^{N} 2b_k \sin\left(k2\pi \frac{x - x_0}{p}\right).$$

Since a trigonometric polynomial can be complex-valued, we can also choose the f_j as complex numbers in the data to be interpolated, (x_j, f_j). If the f_i are real, then the computed coefficients automatically satisfy the conditions in (3.17). However, even in the real-valued case, it is recommended to initially use the trigonometric polynomial in complex form since the formulas for calculating the d_k are much simpler than those for directly computing a_k and b_k.

The trigonometric polynomial solves the following interpolation problem.

Let data points $(x_0, f_0), \ldots, (x_n, f_n)$ be given that satisfy the following conditions:

(i) The support points $x_j \in \mathbb{R}$ are equidistantly distributed on the interval $[x_0, x_0 + p]$, i.e.

$$x_j = x_0 + j\frac{p}{n}.$$

(ii) The values $f_j \in \mathbb{C}$ satisfy $f_0 = f_n$.

Then there exist complex coefficients $d_0, \ldots, d_{n-1} \in \mathbb{C}$ such that the corresponding trigonometric polynomial solves the interpolation problem:

$$T(x_j) = f_j \text{ for all } j = 0, \ldots, n.$$

These coefficients are given by the matrix multiplication:

$$\begin{pmatrix} d_0 \\ \vdots \\ d_{n-1} \end{pmatrix} = \frac{1}{n} \overline{V} \begin{pmatrix} f_0 \\ \vdots \\ f_{n-1} \end{pmatrix}, \tag{3.18}$$

where $\overline{V}$ is a so-called *Vandermonde matrix*:

$$\overline{V} = \begin{pmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \bar{\omega} & \bar{\omega}^2 & \dots & \bar{\omega}^{n-1} \\ 1 & \bar{\omega}^2 & \bar{\omega}^4 & \dots & \bar{\omega}^{2(n-1)} \\ 1 & \bar{\omega}^3 & \bar{\omega}^6 & \dots & \bar{\omega}^{3(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \bar{\omega}^{n-1} & \bar{\omega}^{(n-1)2} & \dots & \bar{\omega}^{(n-1)^2} \end{pmatrix}$$

with $\bar{\omega} = e^{-i2\pi/n}$.

The justification for (3.18) is as follows: The interpolation conditions imply the equations

$$f_j = T(x_j) = \sum_{k=-N}^{N} d_k e^{ik2\pi \frac{x_j - x_0}{p}}.$$

If we set $\omega = e^{i2\pi/n}$, we can write this equation as

$$f_j = \sum_{k=-N}^{N} d_k \omega^{jk}.$$

Since $\omega^n = 1$, we have $\omega^{j(-k)} = \omega^{j(n-k)}$ for $k > 0$. Due to $d_{-k} = d_{n-k}$, we can write $d_{-k}\omega^{j(-k)} = d_{n-k}\omega^{j(n-k)}$ for $k = 1, \dots, N$. This allows us to rewrite the equation as

$$f_j = \sum_{k=0}^{n-1} d_k \omega^{jk},$$

in matrix form

$$\begin{pmatrix} f_0 \\ \vdots \\ f_{n-1} \end{pmatrix} = V \begin{pmatrix} d_0 \\ \vdots \\ d_{n-1} \end{pmatrix}$$

with

$$V = \begin{pmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega & \omega^2 & \dots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \dots & \omega^{2(n-1)} \\ 1 & \omega^3 & \omega^6 & \dots & \omega^{3(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{n-1} & \omega^{(n-1)2} & \dots & \omega^{(n-1)^2} \end{pmatrix}.$$

From the identities $\omega\bar{\omega} = 1$ and $\sum_{i=0}^{n-1} \omega^{ij} = 0$ for $j = 1, \ldots, n-1$, which follow from the properties of the n-th roots of unity, we now obtain the equation $V\overline{V} = n\mathrm{Id}$, hence

$$V^{-1} = \frac{1}{n}\overline{V},$$

from which (3.18) follows.

Note that these calculations can be carried out independently of whether n is even or odd. For even n, the representation of the polynomial is slightly different from that in Definition 3.33 (and the calculations are somewhat more complicated than in Remark 3.34, which is why we do not perform them here). However, the interpretation of the d_k as coefficients of sine and cosine functions with different frequencies remains unchanged.

Due to time constraints, we do not delve into an analysis of the interpolation error here. If the f_j result from function values $f(x_j)$ of a function $f : \mathbb{R} \to \mathbb{C}$ using (3.2), it can be proven that the corresponding trigonometric polynomial T — under suitable conditions on f — provides an approximation of f.

We will now outline a method for the fast and numerically efficiently calculation the coefficients d_i.

3.5.2 Fast Fourier Transform

The coefficients $d_0, \ldots, d_{n-1}$ of the trigonometric interpolation polynomial are commonly referred to as the *discrete Fourier coefficients* of the data $f_0, \ldots, f_{n-1}$ (due to assumptions (i) and (ii) from above, neither the x_j nor the value f_n are needed in the calculation of the d_j). The mapping

$$\begin{pmatrix} f_0 \\ \vdots \\ f_{n-1} \end{pmatrix} \mapsto \begin{pmatrix} d_0 \\ \vdots \\ d_{n-1} \end{pmatrix} \tag{3.19}$$

from (3.18), which can be viewed as a mapping from $\mathbb{C}^n$ to $\mathbb{C}^n$, is called the *discrete Fourier transform* or simply the *DFT*.

From (3.18), it follows that this is a linear transformation; furthermore, we have already seen in the proof of (3.18) that the *inverse transform*

$$\begin{pmatrix} d_0 \\ \vdots \\ d_{n-1} \end{pmatrix} \mapsto \begin{pmatrix} f_0 \\ \vdots \\ f_{n-1} \end{pmatrix}, \tag{3.20}$$

is given by the linear mapping

$$\begin{pmatrix} f_0 \\ \vdots \\ f_{n-1} \end{pmatrix} = V \begin{pmatrix} d_0 \\ \vdots \\ d_{n-1} \end{pmatrix} \tag{3.21}$$

Both the discrete Fourier transformation (3.19) and the inverse transformation (3.20) are important mathematical operations. These could be performed using the mappings (3.18)

and (3.21), respectively, but the matrix-vector multiplication has a computational complexity of order $O(n^2)$. Therefore, it is sensible to use a different algorithm. The so-called *Fast Fourier Transform* (usually abbreviated as *FFT*) is a suitable choice. The idea behind this algorithm is based on the observation that the Fourier coefficients for a dataset $(f_0, \ldots, f_{n-1})$ with an even number of coefficients, i.e., $n = 2m$, can be easily computed if they are already known for the "halved" datasets $(f_0, f_2, \ldots, f_{2m-2})$ and $(f_1, f_3, \ldots, f_{2m-1})$ with even and odd coefficients, respectively.

Let $d_k^{[m,\mathrm{e}]}$ and $d_k^{[m,\mathrm{o}]}$ denote the Fourier coefficients of the halved datasets, then the following equations hold:

$$d_k^{[2m]} = \frac{1}{2}\left(d_k^{[m,\mathrm{e}]} + e^{-ik\pi/m} d_k^{[m,\mathrm{o}]}\right) \tag{3.22}$$

$$d_{m+k}^{[2m]} = \frac{1}{2}\left(d_k^{[m,\mathrm{e}]} - e^{-ik\pi/m} d_k^{[m,\mathrm{o}]}\right) \tag{3.23}$$

for $k = 0, \ldots, m-1$. These equations follow directly from matrix multiplication, as $\bar{\omega} = e^{-i2\pi/n} = e^{-i\pi/m}$:

$$\begin{aligned} d_k^{[2m]} &= \frac{1}{n}\sum_{j=0}^{n-1} f_j \bar{\omega}^{jk} \\ &= \frac{1}{2m}\left(\sum_{\substack{j=0 \\ j \text{ even}}}^{n-1} f_j \bar{\omega}^{jk} + \sum_{\substack{j=0 \\ j \text{ odd}}}^{n-1} f_j \bar{\omega}^{jk}\right) \\ &= \frac{1}{2}\left(\underbrace{\frac{1}{m}\sum_{j=0}^{m-1} f_{2j}\bar{\omega}^{2jk}}_{=d_k^{[m,\mathrm{e}]}} + \bar{\omega}^k \underbrace{\frac{1}{m}\sum_{j=0}^{m-1} f_{2j+1}\bar{\omega}^{2jk}}_{=d_k^{[m,\mathrm{o}]}}\right), \end{aligned}$$

where $\bar{\omega}^{2jk} = \tilde{\omega}^{jk}$ with $\tilde{\omega} = \bar{\omega}^2 = e^{-i2\pi/m}$.

Similarly, we obtain (3.23) from

$$\begin{aligned} d_{m+k}^{[2m]} &= \frac{1}{n}\sum_{j=0}^{n-1} f_j \bar{\omega}^{j(m+k)} \\ &= \frac{1}{2}\left(\underbrace{\frac{1}{m}\sum_{j=0}^{m-1} f_{2j}\bar{\omega}^{2j(m+k)}}_{=d_k^{[m,\mathrm{e}]}} + \underbrace{\bar{\omega}^{m+k}}_{=-\bar{\omega}^k} \underbrace{\frac{1}{m}\sum_{j=0}^{m-1} f_{2j+1}\bar{\omega}^{2j(m+k)}}_{=d_k^{[m,\mathrm{o}]}}\right), \end{aligned}$$

since

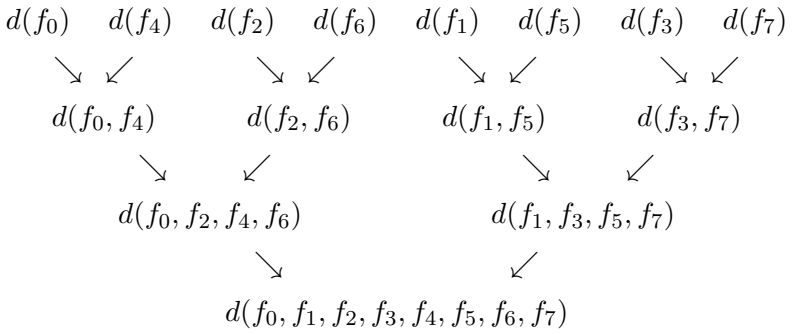
$$\bar{\omega}^{m+k} = \bar{\omega}^m \bar{\omega}^k = \underbrace{e^{-i\pi}}_{=-1}\bar{\omega}^k = -\bar{\omega}^k$$

and

$$\bar{\omega}^{2j(m+k)} = \bar{\omega}^{2jm}\bar{\omega}^{2jk} = (\underbrace{e^{-i2\pi}}_{=1})^j \bar{\omega}^{2jk} = \bar{\omega}^{2jk}$$

hold.

Since the Fourier transformation for datasets of length 1 is simply $d_0 = f_0$, equations (3.22) and (3.23) lead to the following scheme for computing the Fourier coefficients, where $d(\ldots)$ denotes the Fourier coefficients for the dataset indicated in parentheses:



Since the inverse Fourier transformation is based on a matrix multiplication with the same structure, it can be performed using a similar scheme. The actual implementation of this scheme is not straightforward, as the sorting of coefficients must be done efficiently. However, we won't delve into this aspect here. There is a version of this algorithm implemented in MATLAB, which also works for datasets where the number of data points n is not a power of two, although it is slightly slower for non-powers of two compared to powers of two.

Note that at each level of the scheme, $O(n)$ operations must be performed, as there are n values to compute. While the number of datasets decreases as we move down the scheme, the number of coefficients to compute increases accordingly. The number of levels is precisely $\log_2 n$. In this case, it is said that the algorithm has a complexity of $O(n \log n)$. The number of operations increases slightly faster than linearly ($O(n)$) but significantly slower than quadratically ($O(n^2)$).

3.5.3 Applications

As mentioned earlier, trigonometric interpolation is not the primary application of the Fourier Transformation. However, the form of trigonometric polynomials is helpful for understanding some essential applications, which we will briefly outline here. We will focus on the real-valued case, assuming that the trigonometric polynomial is in the form:

$$T(x) = a_0 + \sum_{k=1}^{N} 2a_k \cos\left(k2\pi\frac{x - x_0}{p}\right) - \sum_{k=1}^{N} 2b_k \sin\left(k2\pi\frac{x - x_0}{p}\right).$$

with $d_k = a_k + ib_k$. Furthermore, we assume that the interpolation data was generated from a function f by using $f_j = f(x_j)$, where f is a periodic function with a period $p > 0$, i.e., $f(x + p) = f(x)$ for all $x \in \mathbb{R}$.

Such functions f are mainly encountered in signal processing, where they describe signals transmitted over a data line (x would be a time variable in this interpretation).

It should be noted that the fact that $T(x)$ approximates the function $f(x)$ is not explicitly needed in any of the following applications, but it is essential for these applications to work.

Frequency Analysis

Unlike the coefficients in usual polynomials or splines, the Fourier coefficients of trigonometric polynomials have a clear physical meaning: Each of the Fourier coefficients a_k and b_k corresponds to a specific frequency in the sine or cosine. Let's write briefly:

$$c_{k,p}(x) = \cos\left(k2\pi\frac{x-x_0}{p}\right) \quad \text{and} \quad s_{k,p}(x) = \sin\left(k2\pi\frac{x-x_0}{p}\right).$$

So, $c_{k,p}(x) = c_{k,p}(x+q)$ and $s_{k,p}(x) = s_{k,p}(x+q)$ for $q = p/k$, which means that these functions are q-periodic. If we interpret x as time in seconds, these functions have a repetition frequency of k/p Hertz. The magnitude of the coefficients a_k and b_k indicates the strength of the component of an oscillation with a frequency of k/p Hz in the signal f.

When you graph the magnitude of the Fourier coefficients as a function of frequency, you can see which frequencies are present in the function being analyzed. The "magnitude" is given by the values $2\sqrt{a_k^2+b_k^2}$, which corresponds to the scaled (complex) norm $2|d_k|$ of the Fourier coefficient d_k. The scaling ensures that a simple sinusoidal oscillation with a frequency of k/p Hz corresponds to a value of $2|d_k| = 1$. The function displayed in the resulting graph is called "spectral density."

As an example, consider the function:

$$f(x) = \sin(2\pi\,50x) + \sin(2\pi\,120x).$$

Here, two oscillations with frequencies of 50 Hz and 120 Hz are superimposed, as approximately visible in the graphical representation of the function in Figure 3.2 on the left. However, in the representation of the spectral density in Figure 3.2 on the right, the two frequencies are clearly distinguishable.

The power of this frequency analysis becomes apparent when we add random noise $g(x)$ to the function (where $g(x)$ is a normally distributed random value with an expected value of 0 for each x). Such disturbances are practically unavoidable in signal transmission, although not always as drastic as in our model example here. In the graph of the function with noise in Figure 3.3 on the left, almost nothing is recognizable, but in the Fourier coefficients shown in Figure 3.3 on the right, the two frequencies of the original signal are still clearly visible.

Filtering

As we have seen above, the Fourier Transformation has the property that even with noisy data, you can still recognize the "main" frequencies. This property can be used for filtering noisy data.

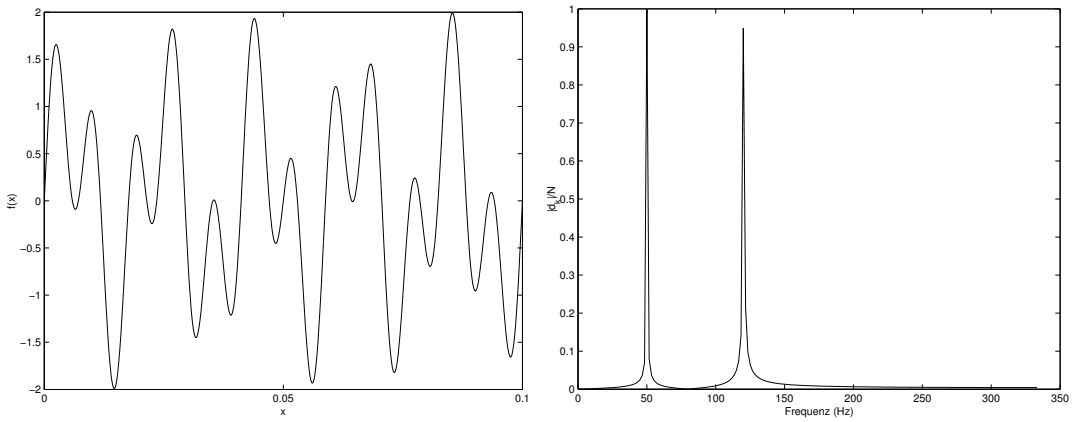


Figure 3.2: Graph and Fourier coefficients of $f(x) = \sin(2\pi 50x) + \sin(2\pi 120x)$

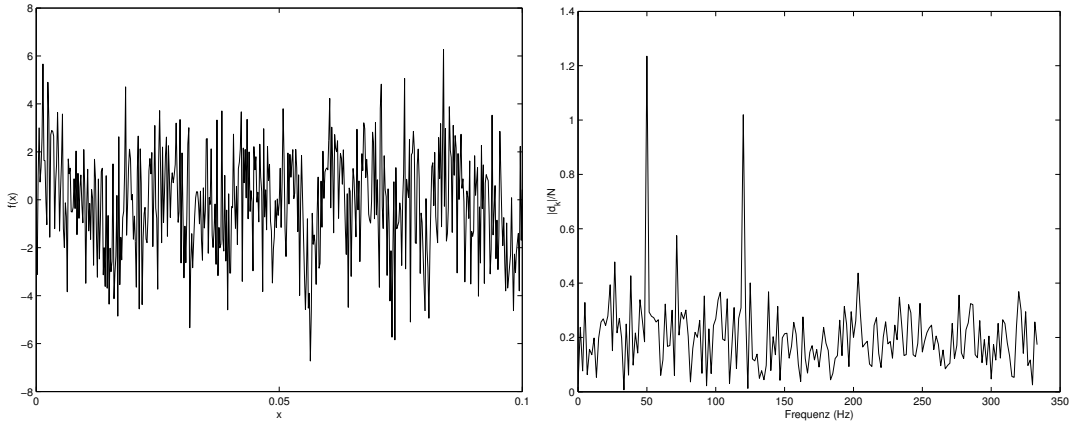


Figure 3.3: Graph and Fourier coefficients of $f(x) = \sin(2\pi 50x) + \sin(2\pi 120x) + g(x)$

Here, we describe a very simple method called a "low-pass filter":

Consider the function $f(x) = \sin(2\pi x)$ and its noisy version as mentioned above, $f(x) = \sin(2\pi x) + g(x)$, see Figure 3.4. Figure 3.5 on the left shows the Fourier coefficients of the noisy function. Now, we set a threshold value s and set all Fourier coefficients whose absolute value satisfies the inequality $2|d_k| < s$ to zero. Figure 3.5 on the right shows the filtered coefficients $(\tilde{d}_0, \ldots, \tilde{d}_{n-1})$ for $s = 0.5$. Finally, we perform a reverse transformation of the filtered coefficients and graphically represent the resulting dataset $(\tilde{f}_0, \ldots, \tilde{f}_{n-1})$ (Figure 3.6). The original signal is now clearly visible again.

Other Applications

Other applications include the decomposition of signals into different frequency ranges (Fourier Transformation → set all coefficients outside the desired frequency range to zero → inverse transformation) or data compression (Fourier Transformation → set all coefficients

with "low contribution" [according to a suitable criterion, e.g., considering a specified percentage of the largest magnitude coefficients] to zero → compact storage).

Since many of the described or sketched applications need to be performed in real-time, the necessity for fast algorithms and efficient implementation is evident.

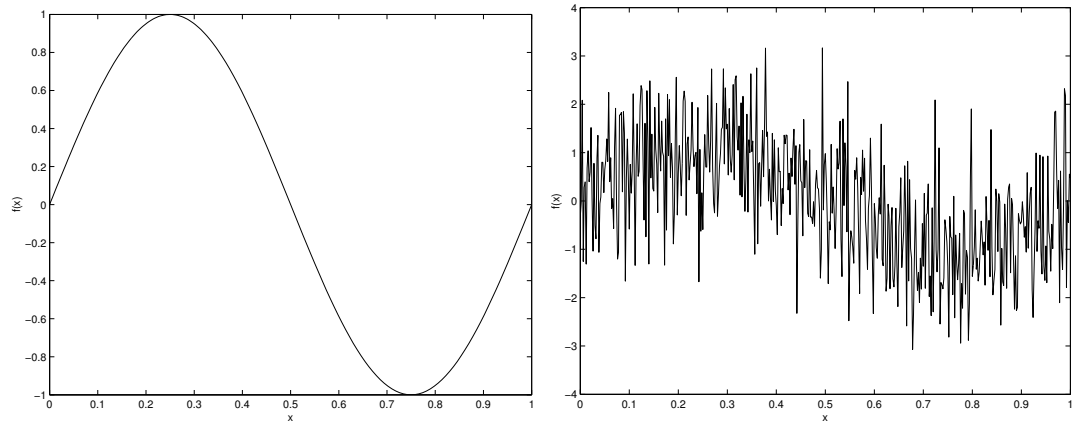


Figure 3.4: Graph of $f(x) = \sin(2\pi x)$ and $f(x) = \sin(2\pi x) + g(x)$

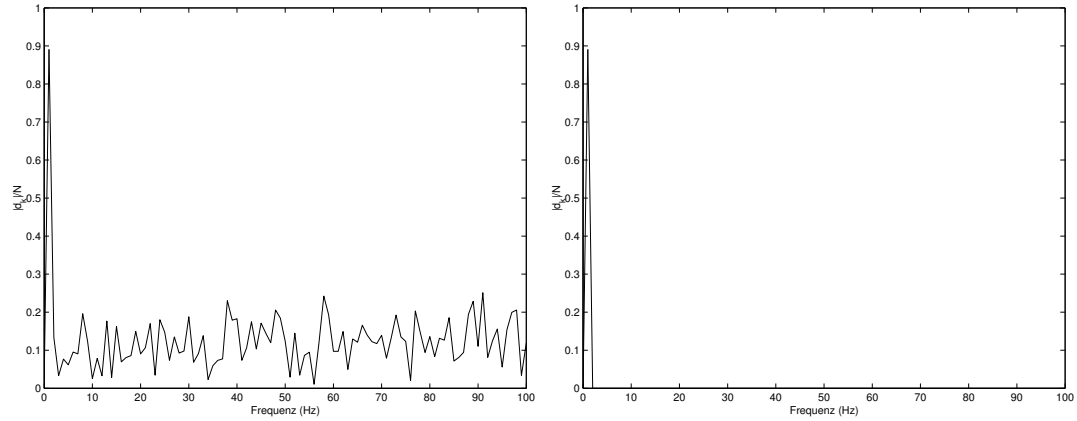


Figure 3.5: Original and filtered Fourier coefficients of $f(x) = \sin(2\pi\, x) + g(x)$

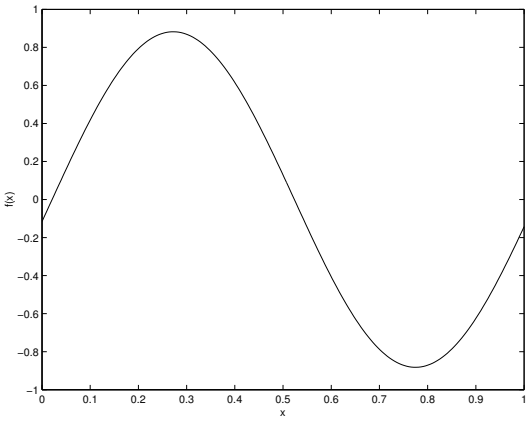


Figure 3.6: Reconstructed function from filtered Fourier coefficients

Chapter 4

Integration

The integration of functions is a fundamental mathematical operation required in many formulas. Unlike the derivative, which can be explicitly computed for virtually all differentiable functions, there are many functions for which integrals cannot be specified explicitly. Numerical integration methods, also referred to as "quadrature", play an essential role in this context, both as standalone algorithms and as the basis for other applications such as the numerical solution of differential equations.

The fundamental problem can be described as follows: For a function $f : \mathbb{R} \to \mathbb{R}$, the integral

$$\int_a^b f(x)dx \tag{4.1}$$

on an interval $[a, b]$ needs to be computed. Some methods can also be applied to infinite integration intervals.

We will explore various integration methods here: the classical "Newton-Cotes formulas" and "composite Newton-Cotes formulas" (also known as "iterated" or "summed" Newton-Cotes formulas), which are based on polynomial interpolation, as well as "Gaussian quadrature", which relies on orthogonal polynomials already known and "Romberg extrapolation", which exploits a detailed and clever analysis of the numerical error.

4.1 Newton-Cotes Formulas

The basic idea behind any numerical integration formula is to approximate the integral (4.1) as a sum

$$\int_a^b f(x)dx \approx (b-a)\sum_{i=0}^{n} \alpha_i f(x_i) \tag{4.2}$$

Here, the x_i are called *support points* and the α_i are the *weights* of the integration formula.

We first consider the case in which the support points x_i can be arbitrarily chosen and derive a formula to compute meaningful weights α_i for the x_i.

The idea of the Newton-Cotes formulas is to approximate the function f first by an interpolation polynomial $P \in \mathcal{P}_n$ to the support points $x_0, \ldots, x_n$ and then calculate the (exact) integral over this polynomial. We now perform this construction:

To obtain an explicit expression in the $f(x_i)$, we use the representation of P by means of Lagrange polynomials

$$P(x) = \sum_{i=0}^{n} f(x_i) L_i(x) \quad \text{where} \quad L_i(x) = \prod_{\substack{j=0 \\ j \neq i}}^{n} \frac{x - x_j}{x_i - x_j},$$

as described in Section 3.1.1. The integral over P then becomes

$$\begin{aligned} \int_a^b P(x)dx &= \int_a^b \sum_{i=0}^{n} f(x_i) L_i(x)\, dx \\ &= \sum_{i=0}^{n} f(x_i) \int_a^b L_i(x)\, dx. \end{aligned}$$

To calculate the weights α_i in (4.2), we set

$$(b-a) \sum_{i=0}^{n} \alpha_i f(x_i) = \sum_{i=0}^{n} f(x_i) \int_a^b L_i(x) dx.$$

Setting the individual terms equal and solving for α_i yields

$$\alpha_i = \frac{1}{b-a} \int_a^b L_i(x) dx. \tag{4.3}$$

These α_i can then be explicitly calculated because the integrals over the Lagrange polynomials L_i have explicit solutions. These α_i depend on the choice of the support points x_i but not on the function values $f(x_i)$.

For equidistant support points

$$x_i = a + \frac{i(b-a)}{n}$$

the weights from (4.3) are given in Table 4.1 for $n = 0, \ldots, 7$.

Note that the weights always sum up to 1 and are symmetric in i, i.e., $\alpha_i = \alpha_{n-i}$. Furthermore, the weights are independent of the interval boundaries a and b. Some of these formulas have their own names. For example, the Newton-Cotes formula for $n = 0$ is called the *Rectangle Rule*, for $n = 1$ the *Trapezoidal Rule*, for $n = 2$ *Simpson's Rule* or *Kepler's Barrel Rule*, and the formula for $n = 3$ is known as *Newton's* 3/8 *Rule*.

From the estimation of the interpolation error, we can derive an estimate for the integration error

$$F_n[f] := \int_a^b f(x)dx - (b-a) \sum_{i=0}^{n} \alpha_i f(x_i) = \int_a^b f(x)dx - \int_a^b P(x)dx$$

where the support points x_i do not necessarily have to be equidistant.

n	α_0	α_1	α_2	α_3	α_4	α_5	α_6	α_7
0	1							
1	$\frac{1}{2}$	$\frac{1}{2}$						
2	$\frac{1}{6}$	$\frac{4}{6}$	$\frac{1}{6}$					
3	$\frac{1}{8}$	$\frac{3}{8}$	$\frac{3}{8}$	$\frac{1}{8}$				
4	$\frac{7}{90}$	$\frac{32}{90}$	$\frac{12}{90}$	$\frac{32}{90}$	$\frac{7}{90}$			
5	$\frac{19}{288}$	$\frac{75}{288}$	$\frac{50}{288}$	$\frac{50}{288}$	$\frac{75}{288}$	$\frac{19}{288}$		
6	$\frac{41}{840}$	$\frac{216}{840}$	$\frac{27}{840}$	$\frac{272}{840}$	$\frac{27}{840}$	$\frac{216}{840}$	$\frac{41}{840}$	
7	$\frac{751}{17280}$	$\frac{3577}{17280}$	$\frac{1323}{17280}$	$\frac{2989}{17280}$	$\frac{2989}{17280}$	$\frac{1323}{17280}$	$\frac{3577}{17280}$	$\frac{751}{17280}$

Table 4.1: Weights of the Newton-Cotes Formulas from (4.3) for equidistant support points x_i

Theorem 4.1 For $n \geq 1$ and α_i being the weights calculated according to (4.3) for the support points $a \leq x_0 < \ldots < x_n \leq b$, let $h := (b-a)/n$ and $z_i := n(x_i - a)/(b-a) \in [0, n]$ for $i = 0, \ldots, n$. Then the following holds.

(i) There exist constants c_n depending only on $z_0, \ldots, z_n$ and n such that for all $f \in C^{n+1}([a, b])$, the estimate

$$|F_n[f]| \leq c_n h^{n+2} \|f^{(n+1)}\|_\infty$$

holds.

(ii) For even n, $f \in C^{n+2}([a, b])$, and symmetrically distributed support points x_i, i.e., $x_i - a = b - x_{n-i}$ for $i = 0, \ldots, n$ (e.g., equidistant support points), there exist constants d_n depending only on $z_0, \ldots, z_n$ and n such that the estimate

$$|F_n[f]| \leq d_n h^{n+3} \|f^{(n+2)}\|_\infty$$

holds. □

Proof: (i) From Theorem 3.17(ii), we know that

$$|f(x) - P(x)| \leq \left\|f^{(n+1)}\right\|_\infty \left|\frac{(x-x_0)(x-x_1)\cdots(x-x_n)}{(n+1)!}\right| = \frac{1}{(n+1)!} \left\|f^{(n+1)}\right\|_\infty \prod_{i=0}^{n} |x - x_i|.$$

Thus, we have

$$|F_n[f]| \quad \leq \quad \frac{1}{(n+1)!} \left\|f^{(n+1)}\right\|_\infty \int_a^b \prod_{i=0}^{n} |x - x_i| \, dx,$$

From this, the claimed estimation follows with

$$c_n \quad = \quad \frac{1}{(n+1)!} \frac{1}{h^{n+2}} \int_a^b \prod_{i=0}^{n} |x - x_i| \, dx \quad = \quad \frac{1}{(n+1)!} \left(\frac{n}{b-a}\right)^{n+2} \int_a^b \prod_{i=0}^{n} |x - x_i| \, dx$$

$$\left(\text{Substitution: } z = n\tfrac{x-a}{b-a},\ z_i = n\tfrac{x_i - a}{b-a}\right) \quad = \quad \frac{1}{(n+1)!} \int_0^n \prod_{i=0}^{n} |z - z_i| \, dz$$

Note that the z_i are in the interval $[0, n]$, so the resulting expression is independent of a and b.

(ii) From the construction of the Newton-Cotes formulas, it immediately follows that polynomials $Q \in \mathcal{P}_n$ are exactly integrated because the interpolating polynomial $P \in \mathcal{P}_n$ over which integration is performed coincides with Q in this case. The proof of (ii) follows from the somewhat surprising property that for even n and symmetrically distributed support points x_i the Newton-Cotes formulas are exact also for polynomials $Q \in \mathcal{P}_{n+1}$. To prove this property, let $Q \in \mathcal{P}_{n+1}$, and let $P \in \mathcal{P}_n$ be the interpolating polynomial at the support points x_i. Then, according to Theorem 3.17(i), for each $x \in [a, b]$, there exists a point $\xi \in [a, b]$ such that

$$Q(x) = P(x) + \underbrace{\frac{Q^{(n+1)}(\xi)}{(n+1)!}}_{=:\gamma}(x - x_0)(x - x_1) \cdots (x - x_n)$$

holds. However, because $Q^{(n+1)}$ is a polynomial of degree 0 and thus constant, γ is independent of ξ and, therefore, also of x. Consequently, we have

$$F_n[Q] = \int_a^b Q(x)\,dx - \int_a^b P(x)\,dx = \gamma \int_a^b \prod_{i=0}^{n}(x - x_i)\,dx.$$

From the symmetry of x_i, we have $x - x_i = x - (-x_{n-i} + a + b)$ for $x \in [a, b]$, and therefore

$$\begin{aligned}
\int_a^{(a+b)/2} \prod_{i=0}^{n}(x - x_i)dx &= \int_a^{(a+b)/2} \prod_{i=0}^{n}(x - (-x_i + a + b))dx \\
\Big(\text{Substitution: } x = -(x - a - b)\Big) &= -\int_b^{(a+b)/2} \prod_{i=0}^{n}(-x + x_i)dx \\
&= -\int_{(a+b)/2}^{b} \prod_{i=0}^{n}(x - x_i)dx,
\end{aligned}$$

which implies

$$\int_a^b \prod_{i=0}^{n}(x - x_i)\,dx = 0,$$

and thus, $F_n[Q] = 0$, which means that Q is exactly integrated. To prove statement (ii), let f be given as claimed. Let $Q \in \mathcal{P}_{n+1}$ be an interpolation polynomial at the support points $x_0, x_1, \ldots, x_n, x_{n+1}$, where x_{n+1} is a support point distinct from $x_0, \ldots, x_n$ and is otherwise arbitrary within the interval $[x_0, x_n]$. Then, the interpolation polynomials $P \in \mathcal{P}_n$ for f and Q at support points $x_0, \ldots, x_n$ coincide, and according to Theorem 3.17(ii) (analogous to part (i) of this proof with $n + 1$ instead of n) and the previously proven fact $F_N[Q] = 0$,

we have

$$\begin{aligned}
|F_n[f]| &= \left| \int_a^b f(x)\,dx - \int_a^b P(x)\,dx \right| \\
&= \left| \int_a^b f(x)\,dx - \int_a^b Q(x)\,dx + \underbrace{\int_a^b Q(x)\,dx - \int_a^b P(x)\,dx}_{=F_n[Q]=0} \right| \\
&= \left| \int_a^b f(x)\,dx - \int_a^b Q(x)\,dx \right| \\
&\leq \frac{1}{(n+2)!} \|f^{(n+2)}\|_\infty \int_a^b |x - x_{n+1}| \prod_{i=0}^{n} |x - x_i|\,dx.
\end{aligned}$$

Since the right-hand side of this inequality is continuous in x_{n+1} and the left-hand side is independent of x_{n+1}, the estimation also holds for $x_{n+1} = x_{n/2}$. Therefore, this integral expression can now be estimated analogously to part (i) by

$$\frac{1}{(n+2)!} \|f^{(n+2)}\|_\infty \int_a^b |x - x_{n/2}| \prod_{i=0}^{n} |x - x_i|\,dx \leq d_n h^{n+3} \|f^{(n+2)}\|_\infty,$$

where

$$d_n = \frac{1}{(n+2)!} \int_0^n |z - z_{n/2}| \prod_{i=0}^{n} |z - z_i|\,dz.$$

□

Note that part (ii) of the theorem only works for even values of n because the symmetry of the $n+1$ support points required in the proof is only possible when $n+1$ is odd. Additionally, the formulation and proof of the theorem are only meaningful for $n \geq 1$ due to the division by n, but similar estimates can be obtained for $n = 0$.

The constants c_n and d_n depend on n and the arrangement of the "scaled" support points z_i and can be explicitly calculated for given values. We demonstrate this calculation for $n = 1$ and the support points $x_0 = a$ and $x_1 = b$. In this case, we have $z_0 = 0$ and $z_1 = 1$. Thus, we obtain

$$\begin{aligned}
c_1 &= \frac{1}{(1+1)!} \int_0^1 \prod_{i=0}^{1} |z - z_i|\,dz = \frac{1}{2} \int_0^1 (z-0)(1-z)dz = \frac{1}{2} \int_0^1 z - z^2 dz \\
&= \frac{1}{2} \left[\frac{1}{2} z^2 - \frac{1}{3} z^3 \right]_0^1 = \frac{1}{2} \left(\frac{1}{2} - \frac{1}{3} \right) = \frac{1}{12}.
\end{aligned}$$

In Table 4.2, the error estimates calculated using this technique for $n = 1, \ldots, 7$ and equidistant support points are approximately given, where $M_n := \|f^{(n)}\|_\infty$.

Note that the formulas with odd $n = 2m+1$ only provide a slight improvement over the formulas with even $n = 2m$, but they require an additional function evaluation. Formulas with even n are therefore preferable.

n	1	2	3	4	5	6	7
	$\frac{(b-a)^3 M_2}{12}$	$\frac{(b-a)^5 M_4}{2880}$	$\frac{(b-a)^5 M_4}{6480}$	$\frac{5.2(b-a)^7 M_6}{10^7}$	$\frac{2.9(b-a)^7 M_6}{10^7}$	$\frac{6.4(b-a)^9 M_8}{10^{10}}$	$\frac{3.9(b-a)^9 M_8}{10^{10}}$

Table 4.2: Error estimates of the Newton-Cotes formulas for equidistant support points

As expected, accuracy increases with increasing n, but only if the norm of the derivatives $\|f^{(n)}\|_\infty$ does not increase simultaneously with n. Therefore, the same fundamental problems as with interpolation polynomials arise, which is not surprising since they are the basis of this method. However, another problem arises here: for $n = 8$ and $n \geq 10$, some of the weights α_i can become negative. This can lead to numerical issues (e.g., subtractive cancellations) that should be avoided. For this reason, it is not advisable to keep increasing the degree of the underlying polynomial. Nevertheless, Newton-Cotes formulas are important as they form the basis for a range of more efficient integration formulas, which we will discuss in the following sections.

4.2 Composite Newton-Cotes Formulas

One possible solution to the problem of increasing polynomial degrees in integration using Newton-Cotes formulas works similarly to interpolation but is simpler. In interpolation, we transitioned from polynomials to splines, which are piecewise polynomials. To maintain a "nice" approximation in interpolation, we had to impose conditions at the seams that enforce some smoothness of the approximating function. This required deriving coefficients through a linear system of equations.

In integration, this procedure is unnecessary. Similar to splines, we use piecewise polynomials to derive composite Newton-Cotes formulas but do not impose complex conditions at the seams. This is because we are interested not in a nice approximation of the function but rather in a good approximation of the integral. In practice, we do not actually compute the underlying piecewise polynomials but apply the Newton-Cotes formulas to the subintervals as follows:

Let N be the number of subintervals on which we want to use the Newton-Cotes formula of degree n, with $n + 1$ support points each. We set

$$x_i = a + ih, \quad i = 0, 1, \ldots, nN, \quad h = \frac{b-a}{nN}$$

and split the integral (4.1) via

$$\int_a^b f(x)dx = \int_{x_0}^{x_n} f(x)dx + \int_{x_n}^{x_{2n}} f(x)dx + \cdots + \int_{x_{(N-1)n}}^{x_{Nn}} f(x)dx = \sum_{k=1}^{N} \int_{x_{(k-1)n}}^{x_{kn}} f(x)dx.$$

On each subinterval $[x_{(k-1)n}, x_{kn}]$, we then apply the Newton-Cotes formula, i.e., we approximate

$$\int_{x_{(k-1)n}}^{x_{kn}} f(x)dx \approx nh \sum_{i=0}^{n} \alpha_i f(x_{(k-1)n+i})$$

and sum up the partial approximations as follows:

$$\int_a^b f(x)dx \approx nh \sum_{k=1}^{N} \sum_{i=0}^{n} \alpha_i f(x_{(k-1)n+i}).$$

The resulting integration error

$$F_{N,n}[f] := \int_a^b f(x)dx - nh \sum_{k=1}^{N} \sum_{i=0}^{n} \alpha_i f(x_{(k-1)n+i})$$

can be obtained as the sum of the errors $F_n[f]$ on the subintervals. Therefore, from Theorem 4.1(i), we obtain the estimate

$$\begin{aligned} |F_{N,n}[f]| &\leq \sum_{k=1}^{N} c_n h^{n+2} \max_{y \in [x_{(k-1)n}, x_{kn}]} |f^{(n+1)}(y)| \\ &\leq N c_n h^{n+2} \|f^{(n+1)}\|_\infty = \frac{c_n}{n}(b-a)h^{n+1}\|f^{(n+1)}\|_\infty. \end{aligned}$$

For even n, from Theorem 4.1(ii) we get the estimate

$$|F_{N,n}[f]| \leq \frac{d_n}{n}(b-a)h^{n+2}\|f^{(n+2)}\|_\infty.$$

Next, we present the composite Newton-Cotes formulas for $n = 1, 2, 4$ along with their error estimations. In all the formulas, the support points x_i are chosen as $x_i = a + ih$.

$n = 1$, **Trapezoidal Rule:**

$$\int_a^b f(x)dx \approx \frac{h}{2}\left(f(a) + 2\sum_{i=1}^{N-1} f(x_i) + f(b)\right)$$

$$|F_{N,1}[f]| \leq \frac{b-a}{12}h^2\|f^{(2)}\|_\infty, \quad h = \frac{b-a}{N}$$

$n = 2$, **Simpson's Rule:**

$$\int_a^b f(x)dx \approx \frac{h}{3}\left(f(a) + 2\sum_{i=1}^{N-1} f(x_{2i}) + 4\sum_{i=0}^{N-1} f(x_{2i+1}) + f(b)\right)$$

$$|F_{N,2}[f]| \leq \frac{b-a}{180}h^4\|f^{(4)}\|_\infty, \quad h = \frac{b-a}{2N}$$

$n = 4$, **Milne's Rule:**

$$\begin{aligned} \int_a^b f(x)dx \approx \frac{2h}{45}\Bigg(&7\Big(f(a) + f(b)\Big) + 14\sum_{i=1}^{N-1} f(x_{4i}) \\ &+32\sum_{i=0}^{N-1}\Big(f(x_{4i+1}) + f(x_{4i+3})\Big) + 12\sum_{i=0}^{N-1} f(x_{4i+2})\Bigg) \end{aligned}$$

$$|F_{N,4}[f]| \leq \frac{2(b-a)}{945}h^6\|f^{(6)}\|_\infty, \quad h = \frac{b-a}{4N}$$

Let's illustrate the practical implications of these error estimates with an example.

Example 4.2 The integral

$$\int_0^1 e^{-x^2/2} dx$$

is to be numerically approximated with a guaranteed accuracy of $\epsilon = 10^{-10}$. The derivatives of the function $f(x) = e^{-x^2/2}$ can be computed relatively easily, and we have

$$f^{(2)}(x) = (x^2-1)f(x), \quad f^{(4)}(x) = (3-6x^2+x^4)f(x), \quad f^{(6)}(x) = (-15+45x^2-15x^4+x^6)f(x).$$

With some calculation (or from their graphical representation), it can be seen that all these functions attain their maximum absolute value on $[0, 1]$ at $y = 0$, from which the following values follow:

$$\|f^{(2)}\|_\infty = 1, \quad \|f^{(4)}\|_\infty = 3, \quad \text{and} \quad \|f^{(6)}\|_\infty = 15.$$

Solving the provided error estimates $F_{N,n}[f] \leq \epsilon$ for the Trapezoidal, Simpson's, and Milne's rules yields the following conditions on h:

$$\begin{aligned}
h &\leq \sqrt{\tfrac{12\epsilon}{(b-a)\|f^{(2)}\|_\infty}} &&\approx \tfrac{1}{28867.51} && \text{(Trapezoidal Rule)} \\
h &\leq \sqrt[4]{\tfrac{180\epsilon}{(b-a)\|f^{(4)}\|_\infty}} &&\approx \tfrac{1}{113.62} && \text{(Simpson's Rule)} \\
h &\leq \sqrt[6]{\tfrac{945\epsilon}{2(b-a)\|f^{(6)}\|_\infty}} &&\approx \tfrac{1}{26.12} && \text{(Milne's Rule)}
\end{aligned}$$

The right-hand side fractions give the maximum allowable value for h. To achieve this, $1/(nN) \leq h$ must hold for the number $nN + 1$ of support points. Since nN is an integer multiple of n, we need 28869 support points for the Trapezoidal Rule, 115 support points for Simpson's Rule, and 29 support points for Milne's Rule. □

4.3 Gaussian Quadrature

So far, we have obtained numerical integration formulas by searching for suitable weights for arbitrarily given support points. However, similar to interpolation, we can now attempt to determine the support points using a suitable method and thereby reduce the numerical error. In a Newton-Cotes formula of degree n, there are $n + 1$ support points and $n + 1$ weights, resulting in $2n + 2$ free parameters. The goal of Gaussian Quadrature is to choose these parameters optimally in the Newton-Cotes formulas, where "optimal" in this context means that we want to maximize the degree of polynomials that are integrated exactly. The Newton-Cotes formulas are constructed so that the formula of degree n exactly integrates polynomials of degree n for arbitrary support points. For symmetric support points and even n, they even exactly integrate polynomials of degree $n+1$. If we "naively" argue with the dimension of the spaces and the number of free parameters, one might assume that with a clever choice of support points and weights, polynomials of degree $2n + 1$ can be exactly integrated, as the dimension $2n + 2$ of the polynomial space $\mathcal{P}_{2n+1}$ then matches the number of free parameters.

Before we delve into the theory of Gaussian Quadrature, let us illustrate this with the example of $n = 1$. We want to determine support points x_0 and x_1 as well as weights α_0 and α_1 so that the equation

$$\int_a^b Q(x)dx = (b-a)(\alpha_0 Q(x_0) + \alpha_1 Q(x_1))$$

is satisfied for every $Q \in \mathcal{P}_3$. Since both sides of this equation are linear in the coefficients of Q, it suffices to determine the parameters such that the equation is satisfied for the elements of the monomial basis $\mathcal{B} = \{1, x, x^2, x^3\}$ of $\mathcal{P}_3$. So we must have

$$\begin{aligned} b-a &= (b-a)(\alpha_0 + \alpha_1) \\ \frac{b^2-a^2}{2} &= (b-a)(\alpha_0 x_0 + \alpha_1 x_1) \\ \frac{b^3-a^3}{3} &= (b-a)(\alpha_0 x_0^2 + \alpha_1 x_1^2) \\ \frac{b^4-a^4}{4} &= (b-a)(\alpha_0 x_0^3 + \alpha_1 x_1^3) \end{aligned}$$

This is a (non-linear) system of equations with 4 equations and 4 unknowns, which has the solution

$$\begin{aligned} \alpha_0 &= \frac{1}{2}, \qquad x_0 = -\frac{b-a}{2}\frac{1}{\sqrt{3}} + \frac{b+a}{2} \\ \alpha_1 &= \frac{1}{2}, \qquad x_1 = +\frac{b-a}{2}\frac{1}{\sqrt{3}} + \frac{b+a}{2} \end{aligned}$$

The corresponding integration formula is called the *Gauss-Legendre rule*.

In fact, it can be shown that the resulting systems of equations are always solvable. A direct proof is quite cumbersome; a much more elegant — and also more general — approach is to prove this with orthogonal polynomials, which we will do next.

To do this, we consider the more general numerical integration problem

$$\int_a^b \omega(x) f(x) dx \approx \sum_{i=0}^{n} \lambda_i f(x_i)$$

where $\omega : (a, b) \to \mathbb{R}$ is a non-negative *weight function*. Note that the factor $(b-a)$ from the Newton-Cotes formulas (4.2) is included in the weights λ_i here. The previously discussed integration problem without a weight function is included here as the special case $\omega(x) \equiv 1$, but other functions ω are also possible, e.g., $\omega(x) = 1/(1-x^2)$ on the interval $[-1, 1]$. If f is bounded by a polynomial, it also makes sense to consider integration problems on infinite integration intervals with exponentially decaying weight functions, e.g., with $\omega(x) = e^{-x}$ on $[0, \infty)$ or with $\omega(x) = e^{-x^2}$ on $(-\infty, \infty)$.

The following theorem shows how to choose the support points and weights optimally, i.e., such that polynomials of degree $2n + 1$ are integrated exactly.

Theorem 4.3 The Gaussian quadrature formula

$$\int_a^b \omega(x) f(x) dx \approx \sum_{i=0}^{n} \lambda_i f(x_i)$$

is exact for $f = P \in \mathcal{P}_{2n+1}$ if the following two conditions are met:

(1) The support points $x_0, \ldots, x_n$ are the zeros of the orthogonal polynomial P_{n+1} according to Definition 3.22 with respect to the weight function ω.

(2) The weights λ_i are (analogous to the Newton-Cotes formulas) given by

$$\lambda_i = \int_a^b \omega(x) L_i(x) dx$$

with the Lagrange polynomials

$$L_i(x) = \prod_{\substack{j=0 \\ j \neq i}}^{n} \frac{x - x_j}{x_i - x_j}.$$

□

Proof: We first show that the formula is exact for $P \in \mathcal{P}_n$. For $P \in \mathcal{P}_n$, we have

$$P(x) = \sum_{i=0}^{n} P(x_i) L_i(x)$$

and therefore

$$\begin{aligned} \int_a^b \omega(x) P(x) dx &= \int_a^b \omega(x) \sum_{i=0}^{n} P(x_i) L_i(x) dx \\ &= \sum_{i=0}^{n} P(x_i) \int_a^b \omega(x) L_i(x) dx = \sum_{i=0}^{n} P(x_i) \lambda_i. \end{aligned}$$

It remains to be shown that the formula is also exact for $P \in \mathcal{P}_{2n+1}$. Let $P \in \mathcal{P}_{2n+1} \setminus \mathcal{P}_n$. Then P can be written as

$$P(x) = Q(x) P_{n+1}(x) + R(x)$$

with $Q, R \in \mathcal{P}_n$. Thus, we have

$$\begin{aligned} &\int_a^b \omega(x) P(x) dx - \sum_{i=0}^{n} \lambda_i P(x_i) \\ &= \underbrace{\int_a^b \omega(x) Q(x) P_{n+1}(x) dx}_{= \langle Q, P_{n+1} \rangle_\omega} + \int_a^b \omega(x) R(x) dx - \sum_{i=0}^{n} \lambda_i Q(x_i) P_{n+1}(x_i) - \sum_{i=0}^{n} \lambda_i R(x_i). \end{aligned}$$

Since $Q \in \mathcal{P}_n$, Q can be written as a linear combination of the orthogonal polynomials $P_0, \ldots, P_n$. From $\langle P_k, P_{n+1} \rangle_\omega = 0$ for $k = 0, \ldots, n$, it follows that $\langle Q, P_{n+1} \rangle_\omega = 0$. Since the x_i are the zeros of P_{n+1}, we have

$$\sum_{i=1}^{n} \lambda_i Q(x_i) P_{n+1}(x_i) = 0.$$

Thus,

$$\int_a^b \omega(x)P(x)dx - \sum_{i=1}^n \lambda_i f(x_i) = \int_a^b \omega(x)R(x)dx - \sum_{i=1}^n \lambda_i R(x_i) = 0,$$

because $R \in \mathcal{P}_n$ and the formula is exact for these polynomials. □

Remark 4.4 Analogous to the proof of Theorem 4.1, here we can also estimate the integration error through the interpolation error, leading to the inequality

$$|F_n[f,\omega]| \leq \frac{\langle P_{n+1},\, P_{n+1}\rangle_\omega}{(2n+2)!}\|f^{(2n+2)}\|_\infty.$$

Here P_{n+1} is the orthogonal polynomial for weight function ω according to Definition 3.22 with leading coefficient 1. □

We conclude this section with some examples of Gaussian quadrature formulas.

Example 4.5 (i) For $\omega(x) = 1$ and the integration interval $[-1, 1]$, we obtain the *Legendre polynomials* as orthogonal polynomials; for $n = 1$, these yield the zero or support points $x_{0/1} = \pm 1/\sqrt{3}$ and weights $\lambda_{0/1} = 1$. The corresponding integration formula is called the *Gauss-Legendre rule*.

(ii) For $\omega(x) = 1/\sqrt{1-x^2}$ on $[-1, 1]$, we obtain the well-known *Chebyshev polynomials* T_n. Here, the integration formula can be explicitly given as

$$\frac{\pi}{n+1}\sum_{i=0}^n f\left(\cos\left(\frac{2i+1}{2n+2}\pi\right)\right).$$

Due to the singularities of the weight function at the boundary points, this formula cannot be used in composite form.

(iii) For $\omega(x) = e^{-x}$ on $[0, \infty)$ or $\omega(x) = e^{-x^2}$ on $(-\infty, \infty)$, the corresponding polynomials are called *Laguerre* and *Hermite polynomials*, respectively. For these, there are no closed-form expressions for the zeros and weights, so they must be determined numerically. This numerical calculation can be done using a numerically efficient evaluation of the recursive formulas from Section 3.3, see Deuflhard/Hohmann [1], Section 9.3.2. □

The Gaussian-Legendre integration from (i) is rarely used today, as methods like Romberg extrapolation are often more efficient. However, Gaussian quadrature is useful when integrating explicitly with given weight functions like in (ii) or on an infinite time horizon as in (iii).

4.4 Romberg Extrapolation

So far, we have examined integration formulas in which we explicitly specified the number of support points. In this and the following section, we will explore methods where the number of support points is variable.

These methods are based on the (composite) trapezoidal rule, which we denote here as $h = (b-a)/N$ with $T(h)$:

$$T(h) = \frac{h}{2}\left(f(a) + 2\sum_{j=1}^{N-1} f(a+jh) + f(b)\right).$$

For this formula, there exists a theorem proven independently by Euler and McLaurin, which goes beyond the error estimations considered so far.

Theorem 4.6 Let $f \in C^{2m+1}([a,b])$ and $h = (b-a)/N$ for some $N \in \mathbb{N}$. Then, for the trapezoidal rule, the equation holds:

$$T(h) = \int_a^b f(x)dx + \tau_2 h^2 + \tau_4 h^4 + \ldots + \tau_{2m} h^{2m} + R_{2m+2}(h)h^{2m+2}$$

with coefficients

$$\tau_{2k} = \frac{B_{2k}}{(2k)!}\left(f^{(2k-1)}(b) - f^{(2k-1)}(a)\right),$$

where B_{2k} are the so-called *Bernoulli numbers*. The remainder term is uniformly bounded by

$$|R_{2m+2}(h)| \leq C_{2m+2}(b-a)\,\|f^{(2m)}\|_\infty,$$

where C_{2m+2} is a constant independent of h. □

The formal proof of this theorem is quite complicated, which is why we do not present it here. A sketch of the proof can be found, for example, in the book by Stoer [9], Section 3.3.

The important aspect of this formula is that the coefficients τ_{2k} do not depend on h (and thus not on the number of support points), only the remainder term depends on h. The function $T(h)$ can be seen as a polynomial (disturbed by the remainder term $R_{2m+2}(h)h^{2m+2}$). The explicit form of the Bernoulli numbers will not be needed when applying this formula. For large k, it can be shown that

$$B_{2k} \approx (2k)!$$

holds, so the series (unlike the Taylor series) generally does not converge for $m \to \infty$ when higher derivatives of f grow and/or h is large. However, the above expression is meaningful for finite m and small h.

To present the extrapolation method in a more general context, we use the following definition.

Definition 4.7 Let $T(h)$ be a numerical method for approximating the value

$$\tau_0 = \lim_{h\to 0} T(h).$$

An *asymptotic expansion in h^p* of this method up to order pm is a representation of the form

$$T(h) = \tau_0 + \tau_p h^p + \tau_{2p} h^{2p} + \ldots + \tau_{mp} h^{mp} + O(h^{(m+1)p})$$

with constants τ_{ip}, $i = 0, \ldots, m$. Here, the *Landau symbol* $O(h^k)$ denotes any expression with the property that $O(h^k)/h^k \leq C$ for some $C > 0$ and all sufficiently small $h > 0$. □

According to Theorem 4.6, the trapezoidal rule possesses such an asymptotic expansion in h^2 up to order $2m$.

Such an asymptotic expansion forms the basis for *extrapolation*. Extrapolation refers to the method of evaluating an interpolation polynomial for a function $g(x)$ with support points on an interval $[a, b]$ at a point $x^* \notin [a, b]$. We will apply this method to the function $T(h)$ as follows:

(1) Calculate the approximation values $T(h_j)$ for m different step sizes $h_1, \ldots, h_m$

(2) Compute the interpolation polynomial $P(h^p)$ using the data $(h_i^p, T(h_i))$, $i = 1, \ldots, m$, and evaluate it at $h^p = 0$.

Thus, we aim to obtain an approximate value of T for step size $h = 0$ from values of T for large step sizes (i.e., "coarse" approximations of the integral), hoping that this will provide a more accurate approximation of the integral value.

We illustrate the method with a simple example with $m = 2$. Consider the function $f(x) = x^4$ on $[0, 1]$. Obviously, the desired integral is $\int_0^1 x^4 dx = 1/5 =: \tau_0$. Now, let's consider the trapezoidal rule for $N_1 = 1$ and $N_2 = 2$, which gives $h_1 = 1$ and $h_2 = 1/2$. This yields

$$T(1) = \frac{h_1}{2}(f(0) + f(1)) = \frac{1}{2}(0 + 1) = \frac{1}{2}$$

and

$$T\left(\frac{1}{2}\right) = \frac{h_2}{2}\left(f(0) + 2f\left(\frac{1}{2}\right) + f(1)\right) = \frac{1}{4}\left(0 + \frac{2}{16} + 1\right) = \frac{9}{32}.$$

The error is thus $|1/5 - 1/2| = 3/10 = 0.3$ or $|1/5 - 9/32| = 13/160 = 0.08125$.

If we interpolate T by a polynomial P in h^2 at the points h_1^2 and h_2^2, we obtain P as follows:

$$P(h^2) = T(h_1) + \frac{7}{24}(h^2 - h_1^2),$$

as can be easily verified by calculation. When evaluated at $h^2 = 0$, we get

$$P(0) = \frac{1}{2} + \frac{7}{24}(-1) = \frac{5}{24},$$

which, due to $|1/5 - 5/24| = 1/120 = 0.008\overline{3}$, provides a significantly better approximation of the integral value than $T(h_1)$ or $T(h_2)$.

Before we examine the improvement achievable in Theorem 4.8, let's focus on the implementation. To efficiently program this method, we need an algorithm for quickly calculating the value of an interpolation polynomial at a given point. For $k \geq 2$, from Formula (3.5), we have the identity

$$P_{i,k}(x) = \frac{(x_{i-k+1} - x)P_{i,k-1}(x) - (x_i - x)P_{i-1,k-1}(x)}{x_{i-k+1} - x_i},$$

which is also known as the *Lemma of Aitken*. Here, $P_{i,k}(x)$ is the interpolation polynomial through the data points $(x_{i-k+1}, f_{i-k+1}), \ldots, (x_i, f_i)$. By applying this formula with $x = 0$ and the values $(x_i, f_i) = (h_i^p, T(h_i))$, we can easily deduce that the values $T_{i,k} = P_{i,k}(0)$ can be obtained using the recursive formula:

$$\begin{aligned} T_{i,1} &:= T(h_i), \quad i = 1, 2, \ldots \\ T_{i,k} &:= T_{i,k-1} + \frac{T_{i,k-1} - T_{i-1,k-1}}{\left(\frac{h_{i-k+1}}{h_i}\right)^p - 1}, \quad k = 2, 3, \ldots; \quad i = k, k+1, \ldots \end{aligned}$$

This calculation, referred to as the *extrapolation scheme*, can be graphically represented similarly to the calculation of divided differences, as shown in Fig. 4.1 (also see Fig. 3.1).

$$\begin{array}{ccccccccc} T_{1,1} & & & & & & & & \\ & \searrow & & & & & & & \\ T_{2,1} & \to & T_{2,2} & & & & & & \\ & \searrow & & \searrow & & & & & \\ T_{3,1} & \to & T_{3,2} & \to & T_{3,3} & & & & \\ & \searrow & & \searrow & & \searrow & & & \\ T_{4,1} & \to & T_{4,2} & \to & T_{4,3} & \to & T_{4,4} & & \\ \vdots & & \vdots & & \vdots & & \vdots & \ddots & \end{array}$$

Figure 4.1: Illustration of the extrapolation scheme

Note that k can be increased arbitrarily by appending further rows, and the previously calculated values for larger k can still be used. The next theorem provides the accuracy of the approximation $T_{k,k}$.

Theorem 4.8 Let $T(h)$ be a method with an asymptotic expansion according to Definition 4.7 in h^p up to order pm. Then, for all sufficiently small step sizes $h_1, \ldots, h_m$, the approximation error

$$\varepsilon_{i,k} := |T_{i,k} - \tau_0|$$

satisfies the equation

$$\varepsilon_{i,k} = |\tau_{kp}| h_{i-k+1}^p \ldots h_i^p + \sum_{j=i-k+1}^{i} O(h_j^{(k+1)p}).$$

□

Proof: We first prove the following equation for the Lagrange polynomials L_i at support points $x_1, \ldots, x_n$:

$$\sum_{i=1}^{n} L_i(0) x_i^m = \begin{cases} 1, & \text{if } m = 0, \\ 0, & \text{if } 1 \le m \le n-1 \\ (-1)^{n-1} x_1 \cdots x_n, & \text{if } m = n \end{cases} \tag{4.4}$$

To prove (4.4), we consider the polynomial $P(x) = x^m$. Clearly, for $m \leq n-1$, this is the interpolation polynomial through the data points (x_i, x_i^m), $i = 1, \ldots, n$: the polynomial passes through these $n-1$ points and has degree $\leq n-1$. Therefore,

$$P(x) = x^m = \sum_{i=1}^{n} L_i(x) P(x_i) = \sum_{i=1}^{n} L_i(x) x_i^m.$$

The assertion (4.4) for $m = 1, \ldots, n-1$ follows from this by substituting $x = 0$.

For $m = n$, the polynomial $x^m = x^n$ also passes through the given $n-1$ points but is of degree n, and hence, it is not the unique interpolation polynomial due to its higher degree. Therefore, in this case, we consider the difference between x^n and the unique interpolation polynomial of degree $\leq n-1$, i.e.,

$$Q(x) = x^n - \sum_{i=1}^{n} L_i(x) x_i^n.$$

This polynomial has a leading coefficient of 1 and, because $\sum_{i=1}^{n} L_i(x_j) x_i^n = x_j^n$, it has the n roots $x_1, \ldots, x_n$. Consequently,

$$Q(x) = (x - x_1) \cdots (x - x_n),$$

and thus,

$$\sum_{i=1}^{n} L_i(0) x_i^n = -Q(0) = -(-x_1) \cdots (-x_n) = (-1)^{n-1} x_1 \cdots x_n,$$

which verifies (4.4).

To prove the theorem, we consider the asymptotic expansion of $T(h)$. This leads to the equation for $j = 1, \ldots, m$:

$$T_{j,1} = T(h_j) = \tau_0 + \tau_p h_j^p + \ldots + \tau_{kp} (h_j^p)^k + O(h_j^{(k+1)p}). \tag{4.5}$$

We demonstrate the claim for $i = k$, and estimates for $i > k$ follow by renumbering the step sizes h_i. Let $P(h^p)$ be the interpolation polynomial in h^p for the data $(h_1^p, T(h_1)), \ldots, (h_k^p, T(h_k))$. Furthermore, let $L_1(x), \ldots, L_k(x)$ be the Lagrange polynomials corresponding to the support points $h_1^p, \ldots, h_k^p$. Then, we have:

$$P(h^p) = \sum_{i=1}^{k} L_i(h^p) T_{i,1}.$$

Using (4.4) applied to $x_i = h_i^p$ for $i = 1, \ldots, k$ and (4.5), we obtain:

$$\begin{aligned} T_{k,k} &= P(0) \\ &= \sum_{i=1}^{k} L_i(0) T_{i,1} \\ &= \sum_{i=1}^{k} L_i(0) \Big(\tau_0 + \tau_p h_i^p + \ldots + \tau_{kp} (h_i^p)^k + O(h_i^{(k+1)p}) \Big) \\ &= \tau_0 + \underbrace{\tau_{kp} (-1)^{k-1} h_1^p \cdots h_k^p + \sum_{i=1}^{k} O(h_i^{(k+1)p})}_{|\cdot| = \epsilon_{k,k}}, \end{aligned}$$

and thus, we have proved the claim. □

The theorem states that when we add a row with step size h_m to the extrapolation scheme, we can expect the error to decrease by a factor of order h_m^p, which is second order (h_m^2) in the trapezoidal scheme. Note that this statement holds only when all used step sizes are sufficiently small. Intuitively, the step sizes h_i must lie within a range where the values $T(h_i)$ provide enough information about the value $\tau_0 = \lim_{h \to 0} T(h)$.

In a practical implementation, it is advisable to choose the sequence of step sizes in a decreasing manner, i.e., $h_{i+1} < h_i$. Additionally, h_i is typically chosen as part of a base step size $h = (b-a)/N$, i.e., $h_i = h/N_i$ for $N_i \in \mathbb{N}$. For the trapezoidal method with $p = 2$, this leads to the following algorithm:

Algorithm 4.9 (Romberg Extrapolation)

(0) Choose a maximum number of iterations $i_{\max}$, a base step size $h = (b-a)/N$, and a sequence of step sizes $h_1, h_2, \ldots$ with $h_i = h/N_i$, $N_{i+1} > N_i$. Set $i := 1$.

(1) Calculate $T_{i,1} := T(h_i) = \dfrac{h_i}{2} \left(f(a) + 2 \sum_{j=1}^{N \cdot N_i - 1} f(a + j h_i) + f(b) \right)$.

(2) Calculate

$$T_{i,k} := T_{i,k-1} + \frac{T_{i,k-1} - T_{i-1,k-1}}{\left(\dfrac{N_i}{N_{i-k+1}} \right)^2 - 1} \text{ for } k = 2, \ldots, i$$

(3) If $i \geq i_{\max}$ or $T_{i,i}$ is accurate enough, end the algorithm; otherwise, set $i = i + 1$ and return to (1).

□

The termination criterion "accurate enough" is, of course, not very precise. Typically, a desired accuracy ϵ is specified, and iterations continue until $|T_{i+1,i+1} - T_{i,i}| < \epsilon$ holds. A

relative termination criterion is, for example, given by $|T_{i+1,i+1} - T_{i,i}| < \epsilon \cdot |T_{i+1,i+1}|$. If there is a bound for the integral

$$\text{absint} := \int_a^b |f(x)|dx,$$

one can alternatively iterate until $|T_{i+1,i+1} - T_{i,i}| < \epsilon \cdot \text{absint}$ holds. The difference lies in using the integrated quantity of $|f|$ rather than the value of the integral to weight ϵ, which provides a more easily achievable criterion when the sign of f changes. These termination criteria work reasonably well in practice but only ensure the desired accuracy is reached when the base step size h is sufficiently small.

In implementation, the choice of the sequence N_i also plays a crucial role. With cleverly chosen N_i, you can reuse precomputed function values $f(x_j)$ when calculating $T(h_i)$. For every step size $h = (b-a)/N$, the equation holds:

$$\begin{aligned} T(h/2) &= \frac{h}{4}\left(f(a) + 2\sum_{j=1}^{2N-1} f(a+jh/2) + f(b)\right) \\ &= \underbrace{\frac{h}{4}\left(f(a) + 2\sum_{j=1}^{N-1} f(a+jh) + f(b)\right)}_{=T(h)/2} + \frac{h}{2}\sum_{j=1}^{N} f(a+(2j-1)h/2), \end{aligned}$$

so calculating $T(h/2)$ only requires N evaluations of f if the remaining $N-1$ values are covered using $T(h)$. This can be exploited by choosing $N_i = 2^{i-1}$; this sequence of step sizes is called the Romberg sequence and is commonly used in the implementation of the extrapolation method. With this choice of N_i, you can replace the equation in step (1) for $i \geq 2$ with:

$$T_{i,1} := \frac{1}{2}T_{i-1,1} + h_i \sum_{k=1}^{N \cdot N_{i-1}} f(a+(2k-1)h_i).$$

4.5 Adaptive Romberg Quadrature

In all the previous methods, we chose the support points independently of the integrand f. However, it is reasonable to expect that for certain f, a clever placement of support points dependent on f can yield significantly better results. Consider, for example, the "needle pulse function":

$$f(x) = \frac{1}{10^{-4} + x^2}$$

in Figure 4.2.

For this function, it makes sense to concentrate many support points near 0 and use relatively few support points outside of this region.

To be able to distribute support points variably, we divide the integration interval into subintervals and approximate the integral on each subinterval using a numerical formula.

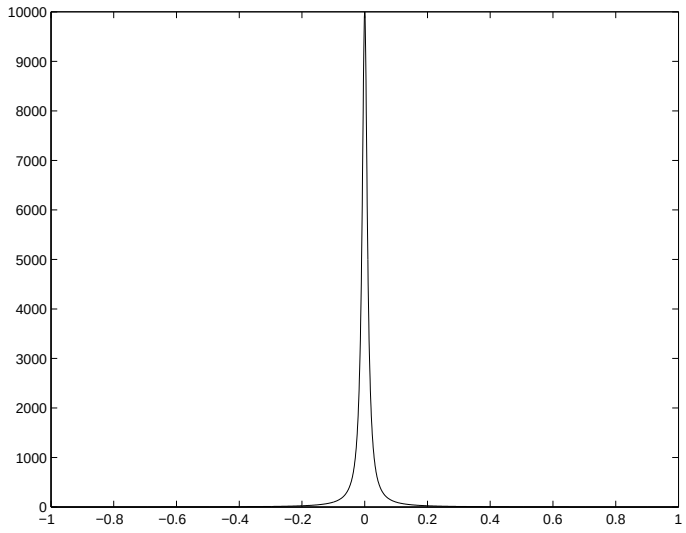


Figure 4.2: Plot of the needle pulse function

Regions where subintervals are small receive more support points than regions with larger subintervals.

The idea of *adaptive integration* is to choose the size of these subintervals efficiently based on f, making them adaptive to f. This should be done without prior complex analysis of f and without additional information (such as derivatives of f), and it should be fast to ensure that the computational effort in determining the interval size does not negate the gained advantage. Additionally, it should be reliable, meaning it guarantees that a given error threshold is met.

These last two criteria - low computational cost and high reliability - appear to be in conflict, as the more certain such a method needs to be, the more effort is required to determine the interval size. A good compromise between effort and reliability is achieved through the concept of *a posteriori error estimation*, which we will discuss below.

Formally, an error estimator is defined by the following definition:

Definition 4.10 A numerically computable quantity $\bar{\epsilon}$ is called an "error estimator" for the actual error ϵ of a numerical method if there exist independent constants κ_1 and κ_2 such that the inequality holds:

$$\kappa_1 \epsilon \leq \bar{\epsilon} \leq \kappa_2 \epsilon$$

□

Such error estimators are then computed iteratively for all subintervals of the integration interval; where the estimated error is large, smaller subintervals are chosen to increase the number of support points and reduce the error.

Now, we will derive such an error estimator $\bar{\epsilon}$ for the extrapolation method. We consider the Romberg method with an integration interval $[a, b] = [x, x + h]$ and a base step size of

$h = b - a$, i.e., $N = 1$. On this interval, according to Theorem 4.8, the approximation is as follows:

$$\epsilon_{i,k} = \left| T_{i,k} - \int_x^{x+h} f(y)\, dy \right| \approx |\tau_{2k}| h_{i-k+1}^2 \dots h_i^2. \tag{4.6}$$

According to Theorem 4.6, the coefficients satisfy:

$$\tau_{2k} = \frac{B_{2k}}{(2k)!} \left(f^{(2k-1)}(x+h) - f^{(2k-1)}(x) \right) \approx \frac{B_{2k}}{(2k)!} f^{(2k)}(x) h$$

where this approximation is valid for small h. The approximate constants $\bar{\tau}_{2k}$ depend on the integrand f. Substituting this approximation for τ_{2k} into (4.6), we obtain:

$$\epsilon_{i,k} \approx |\tau_{2k}| h_{i-k+1}^2 \dots h_i^2 \approx |\bar{\tau}_{2k}| \gamma_{i,k} h^{2k+1} \tag{4.7}$$

with $\gamma_{i,k} = (N_{i-k+1} \cdots N_i)^{-2}$. The exponent $2k+1$ of h here depends on the column index k in the extrapolation scheme. Within a column, it holds that

$$\frac{\epsilon_{i+1,k}}{\epsilon_{i,k}} \approx \frac{\gamma_{i+1,k}}{\gamma_{i,k}} = \left(\frac{N_{i-k+1}}{N_{i+1}} \right)^2 \ll 1$$

(here the notation $a \ll b$ means "a is much smaller than b"). In other words, for small h (since all the approximations used here are only valid for small h), integration errors within a column decrease rapidly, i.e., it holds that

$$\epsilon_{i+1,k} \ll \epsilon_{i,k}. \tag{4.8}$$

We now make the *assumption* that the same holds within the rows, meaning we assume that

$$\epsilon_{i,k+1} \ll \epsilon_{i,k}. \tag{4.9}$$

This assumption means that transitioning to a higher order $k+1$ in the scheme results in a smaller error. This inequality is also valid for sufficiently small h, but it may not necessarily hold for the practical values of h used, which can be a potential source of error in the method.

The construction of an error estimator for an error $\epsilon_{i,k}$ is based on comparing the values $T_{i,k-1}$ and $T_{i,k}$. Relative to the coarse approximation $T_{i,k-1}$, the more accurate approximation $T_{i,k}$ is nearly exact, so that

$$\epsilon_{i,k-1} = \left| T_{i,k-1} - \int_x^{x+h} f(y)\, dy \right| \approx |T_{i,k-1} - T_{i,k}| \tag{4.10}$$

should hold. This method has the significant advantage that the error is calculated from values that are already present in the scheme, anyway, thus requiring no additional computation effort.

Now, suppose we have calculated an extrapolation diagram with k rows and k columns. Under the assumptions (4.8) and (4.9), it is easy to see that in this diagram, the error $\epsilon_{k,k}$ is the smallest, meaning that $T_{k,k}$ represents the most accurate approximation. It would,

therefore, be sensible to obtain an error estimator for $\epsilon_{k,k}$. However, with the above idea, this is not possible because we would need the value $T_{k,k+1}$ for this purpose, which is not included in the diagram and would require considerable effort to compute. Thus, we settle for an estimator for the "second-best" error $\epsilon_{k,k-1}$. The following lemma shows that the informal approximation (4.10) can be formally framed within the framework of Definition 4.10.

Lemma 4.11 If assumption (4.9) for $\epsilon_{k,k-1}$ and $\epsilon_{k,k}$ holds or, more precisely, if there exists $\alpha < 1$ such that the inequality

$$\epsilon_{k,k} \leq \alpha \epsilon_{k,k-1}$$

holds, then the value

$$\bar{\epsilon}_{k,k-1} := |T_{k,k-1} - T_{k,k}|$$

is an error estimator for $\epsilon_{k,k-1}$ in the sense of Definition 4.10 with $\kappa_1 = 1-\alpha$ and $\kappa_2 = 1+\alpha$.

Proof: Let us briefly denote $I = \int_x^{x+h} f(y)\,dy$. Then, we have

$$\bar{\epsilon}_{k,k-1} = |(T_{k,k-1} - I) - (T_{k,k} - I)| \leq \epsilon_{k,k-1} + \epsilon_{k,k}.$$

On the other hand, it follows from $\epsilon_{k,k} < \epsilon_{k,k-1}$

$$\bar{\epsilon}_{k,k-1} = |(T_{k,k-1} - I) - (T_{k,k} - I)| \geq \epsilon_{k,k-1} - \epsilon_{k,k}.$$

Together, with $\alpha \geq \epsilon_{k,k}/\epsilon_{k,k-1}$, we obtain the desired estimates

$$(1-\alpha)\epsilon_{k,k-1} \leq \left(1 - \frac{\epsilon_{k,k}}{\epsilon_{k,k-1}}\right)\epsilon_{k,k-1} = \epsilon_{k,k-1} - \epsilon_{k,k} \leq \bar{\epsilon}_{k,k-1}$$

and

$$\bar{\epsilon}_{k,k-1} \leq \epsilon_{k,k-1} + \epsilon_{k,k} = \left(1 + \frac{\epsilon_{k,k}}{\epsilon_{k,k-1}}\right)\epsilon_{k,k-1} \leq (1+\alpha)\epsilon_{k,k-1}.$$

Even though this error estimator estimates the error for $T_{k,k-1}$, in practical calculations, one should use the value $T_{k,k}$ as the result of the algorithm since, according to assumption (4.9), it is more accurate than $T_{k,k-1}$.

Based on this error estimator, we now describe the adaptive algorithm. Here, we consider a simple version that works with a fixed predefined order k. A more refined version, where the order k is also adaptively chosen, can be found, for example, in the book by Deuflhard and Hohmann [1].

We want to calculate the integral

$$\int_a^b f(x)\,dx$$

using the extrapolation formula for a given k step by step, while maintaining a predefined accuracy *tol* on each subinterval of the computation.[1] To do this, we choose an initial step size $h_1 \leq b - a$, set $i := 1$, and $x_i := a$, and calculate the approximations

$$T^i_{k,k-1} \approx \int_{x_i}^{x_i+h_i} f(x)\,dx \quad \text{and} \quad T^i_{k,k} \approx \int_{x_i}^{x_i+h_i} f(x)\,dx \tag{4.11}$$

[1] Here, you can optionally use a relative termination criterion as well.

for the partial integral on $[x_i, x_i + h_i]$. We initially assume that we know the integration error $\epsilon^i_{k,k-1}$ for $T^i_{k,k-1}$. According to (4.7), we have

$$\epsilon^i_{k,k-1} \approx \delta_i h_i^{2k+1},$$

so $\delta_i \approx \epsilon^i_{k,k-1} h_i^{-(2k+1)}$. To guarantee $\epsilon^i_{k,k-1} \leq tol$, the calculation should ideally be performed with the *ideal step size*

$$h_i^* = \sqrt[2k+1]{\frac{tol}{\epsilon^i_{k,k-1}}}\, h_i.$$

To calculate this ideal step size without using the unknown value $\epsilon^i_{k,k-1}$, we replace it with the error estimator

$$\bar{\epsilon}^i_{k,k-1} = |T^i_{k,k-1} - T^i_{k,k}|.$$

Since this estimator is not exact, we introduce a "safety parameter" $\rho < 1$ and calculate the new step size as

$$\tilde{h}_i := \sqrt[2k+1]{\frac{\rho\, tol}{\bar{\epsilon}^i_{k,k-1}}}\, h_i.$$

If $\tilde{h}_i$ is smaller than h_i, we repeat the calculation with $h_i = \tilde{h}_i$. If (possibly after repeating the calculation) $\tilde{h}_i \geq h_i$, we move on to the next subinterval. We choose $\tilde{h}_i$ as the new step size, i.e. we set $x_{i+1} = x_i + h_i$ and $h_{i+1} = \tilde{h}_i$. This is reasonable because, due to the continuity of $f^{(2k)}$ for small h_i,

$$\delta_{i+1} \approx \delta_i$$

holds, so

$$\epsilon^{i+1}_{k,k-1} = \delta_{i+1} h_{i+1}^{2k+1} \approx \delta_i h_{i+1}^{2k+1} = \delta_i \tilde{h}_i^{2k+1} = tol$$

is the expected error with this step size. If $x_{i+1} + h_{i+1} > b$, we reduce the new step size to $h_{i+1} = b - x_{i+1}$ to avoid integrating beyond the upper limit. It is also recommended to introduce an upper bound $h_{\max}$ for the allowed maximum step size and limit h_{i+1} to $h_{\max}$.

If $x_{i+1} < b$, we set $i := i + 1$ and continue with (4.11).

If $x_{i+1} = b$, we have reached the upper integration limit, and we can calculate the desired integral approximation as $\sum_{j=1}^{i} T^j_{k,k}$ since we can rely on the already calculated more accurate values $T^i_{k,k}$.

Figure 4.3 shows the application of this algorithm to the needle impulse function $f(x) = \frac{1}{10^{-4}+x^2}$, with the parameters $[a, b] = [-1, 1]$, $k = 3$, $tol = 10^{-5}$, $\rho = 0.8$, and base step size $h = 1$.

In this case, the integration interval was divided into 27 individual intervals by the adaptive algorithm. The minimum step size was 0.005532, and the maximum was 0.350629. [2]

The overall expected accuracy in this method is $m \cdot tol$, where m is the number of steps performed. This accuracy can, of course, only be calculated after the algorithm has produced its result.

[2] MATLAB files for this algorithm will be provided on the lecture homepage at the end of the semester.

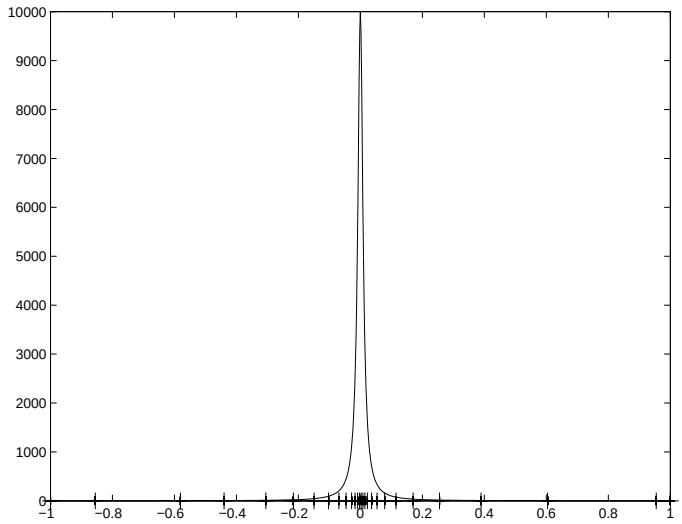


Figure 4.3: Adaptive subintervals in the integration of the needle impulse function

Despite the very efficient "self-adjustment" to the integrands, problems can arise in adaptive Romberg quadrature, as in all numerical methods. If the step sizes are too large or if the assumptions are not met, the error estimators can provide unreliable values. Actual errors can then be "overlooked", resulting in incorrect accuracy calculations and convergence of the method with incorrect values. In such cases, one speaks of *pseudo convergence*. Strategies for detecting this situation can also be found in the book by Deuflhard/Hohmann [1].

Whether an adaptive or a non-adaptive method should be preferred for a given problem depends on the task and the application area of the integration routine. Adaptive methods provide a reliable and efficient strategy for solving integration problems for many integrands f, especially when they are not uniform or no a priori analysis is performed. If more information (e.g., about derivatives) of the integrand are available or can be computed relatively inexpensively compared to the effort of the integration problem, for instance when a specific function needs to be integrated frequently within an algorithm, non-adaptive methods will generally be more efficient, provided that the necessary parameters (order, step size, etc.) are well tuned.

4.6 Higher-Dimensional Integration

In practice, we often need to integrate functions not only on $\mathbb{R}$ but also on $\mathbb{R}^d$. In this overview section, we will focus on problems of a simple form

$$\int_{a_1}^{b_1} \int_{a_2}^{b_2} \dots \int_{a_d}^{b_d} f(x_1, x_2, \dots, x_d) dx_d \dots dx_1$$

with $d \geq 2$.

For low dimensions, we can generalize the previous approach relatively directly. We define the set of d-dimensional grid points as

$$S_d := \left\{ \begin{pmatrix} \bar{x}_1 \\ \vdots \\ \bar{x}_d \end{pmatrix} \middle| \bar{x}_j \in \{a_j, a_j + h_j, a_j + 2h_j, \ldots, a_j + N_j h_j = b_j\}, j = 1, \ldots, d \right\}.$$

Graphically, these points form a regular grid in $\mathbb{R}^d$, known as a *tensor grid*. For fixed values $\bar{x}_1, \ldots, \bar{x}_{d-1}$, we can approximate

$$\int_{a_d}^{b_d} f(\bar{x}_1, \ldots, \bar{x}_{d-1}, x_d) dx_d \approx (b_d - a_b) \sum_{i=0}^{N_d - 1} \alpha_i f(\bar{x}_1, \ldots, \bar{x}_{d-1}, \bar{x}_{d,i}) =: I_d(\bar{x}_1, \ldots, \bar{x}_{d-1})$$

with $\bar{x}_{d,i} = a_d + ih_d$. For the next component $d-1$, we approximate

$$\begin{aligned} \int_{a_d}^{b_d} \int_{a_d}^{b_d} f(\bar{x}_1, \ldots, \bar{x}_{d-2}, x_{d-1}, x_d) dx_d dx_{d-1} &\approx (b_d - a_b) \sum_{i=0}^{N_d - 1} \alpha_i I_d(\bar{x}_1, \ldots, \bar{x}_{d-1,i}) \\ &=: I_d(\bar{x}_1, \ldots, \bar{x}_{d-2}) \end{aligned}$$

with $\bar{x}_{d-1,i} = a_{d-1} + ih_{d-1}$. Iterative continuation ultimately leads to a formula

$$\int_{a_1}^{b_1} \int_{a_2}^{b_2} \ldots \int_{a_d}^{b_d} f(x_1, x_2, \ldots, x_d) dx_d \ldots dx_1 \approx \prod_{i=1}^{d} (b_i - a_i) \sum_{j=0}^{N-1} \beta_j f(\bar{x}_{j,1}, \ldots, \bar{x}_{j,d}),$$

where the vectors $(\bar{x}_{j,1}, \ldots, \bar{x}_{j,d})$ for $j = 0, \ldots, N-1$, $N = (N_1 + 1) \cdot \ldots \cdot (N_d + 1)$, traverse all vectors in the set of grid points S_d, and the β_i are obtained through addition and multiplication from the α_i of the Newton-Cotes formulas. Error estimates for this method can be derived from the one-dimensional error estimates.

While this method works reasonably well for low dimensions $d = 2, 3$, it quickly becomes impractical for higher dimensions. The problem is that the number of grid points in S_d increases very rapidly with growing d. If all N_j are identically equal to $\bar{N} \in \mathbb{N}$, the number of elements in S_d is $(\bar{N} + 1)^d$, which grows exponentially with d. For example, with $N = 9$, i.e. 10 grid points per coordinate direction, S_d contains 10^d grid points. For $d = 10$ this already amounts to 10 billion grid points. This phenomenon is known as the "curse of dimensionality".

Possible remedies include the following approaches, which we only briefly outline here:

- Methods on sparse grids: Here, specific points from S_d are deliberately omitted, which, under certain conditions (such as sufficient differentiability of the integrand f), contribute little to the integral value. This can reduce the growth of the number of elements in S_d with d.

- Methods that exploit special structures: Consider an integrand of the form

$$f(x_1, x_2, \ldots, x_n) = f_1(x_1) + f_2(x_2) + \ldots + f_d(x_d).$$

This can be integrated by integrating the individual component functions. The effort is, therefore, d times the effort of one-dimensional integration, which is much less than the effort of general d-dimensional integration (with 10 grid points per coordinate direction and $d = 10$, we only need 100 instead of 10 billion grid points!). By exploiting such (as well as other) special structures, the effort of numerical integration can be significantly reduced.

- Monte Carlo methods: Here, the elements of the grid point set are not determined by a deterministic rule like in the definition of S_d but by random numbers. This initially has some disadvantages: the computed integral value is itself a random number, which must be taken into account when interpreting the result. Moreover, the accuracy increases relatively slowly with an increase in the number of grid points, so these methods are not suitable for highly accurate calculations. On the other hand, the number of grid points required for a certain level of accuracy also increases only relatively slowly with the dimension. This means that these methods can actually be much better in high dimensions than non-random methods. Additionally, Monte Carlo methods are very easy to implement.

Chapter 5

Systems of Nonlinear Equation

Nonlinear equations or systems thereof must be solved in many applications of mathematics. Typically, the solutions of nonlinear equations are defined as the roots of a function $f : \mathbb{R}^n \to \mathbb{R}^n$, for which we seek an $x^* \in \mathbb{R}^n$ such that

$$f(x^*) = 0.$$

Example (Square Root Computation): Compute $x^* \in \mathbb{R}$ such that $f(x) = 0$ for $f(x) = x^2 - 2$. The unique positive real solution is $\sqrt{2}$. A numerical method for finding roots can thus be used, in particular, for the computation of square roots.

5.1 Fixed-Point Iteration

Fixed-point iteration is a relatively simple method based on the idea of formulating the solution of a nonlinear equation system as a fixed-point equation. If we define

$$g(x) = f(x) + x,$$

then $f(x^*) = 0$ is equivalent to $g(x^*) = x^*$. A point $x \in \mathbb{R}^n$ with $g(x) = x$ is called a *fixed point* of g. Instead of searching for a root of f, we can alternatively search for a fixed point of g.

Recall Banach's Fixed-Point Theorem 2.23.

Theorem 2.23 (Banach's Fixed-Point Theorem) Let A be a closed subset of a complete normed space with norm $\|\cdot\|$, and let $\Phi : A \to A$ be a contraction, i.e., there exists a constant $k \in (0, 1)$ such that the inequality

$$\|\Phi(x) - \Phi(y)\| \leq k\|x - y\|$$

holds. Then, there exists a unique fixed point $x^* \in A$, and all sequences of the form $x^{(i+1)} = \Phi(x^{(i)})$ with any $x^{(0)} \in A$ converge to x^*. Moreover, the *a priori* and *a posteriori* estimates hold:

$$\|x^{(i)} - x^*\| \leq \frac{k^i}{1-k}\|x^{(1)} - x^{(0)}\| \text{ and } \|x^{(i)} - x^*\| \leq \frac{k}{1-k}\|x^{(i)} - x^{(i-1)}\|.$$

The simplest idea for finding a fixed point x^* of g is to iterate this mapping.

Algorithm 5.1 (Fixed-Point Iteration) Given a function $g : \mathbb{R}^n \to \mathbb{R}^n$ and an initial value $x^{(0)} \in \mathbb{R}^n$, as well as a termination tolerance $\epsilon > 0$.

(0) Set $i = 0$ (index).

(1) Set $x^{(i+1)} := g(x^{(i)})$.

(2) If $\|x^{(i+1)} - x^{(i)}\| \leq \epsilon$ or if a maximum number of iterations is exceeded, terminate the algorithm;
otherwise, set $i := i + 1$ and go to (1).

□

If g satisfies the conditions of Banach's Fixed-Point Theorem on a neighborhood A of x^*, then this procedure converges for all $x^{(0)} \in A$. The Banach Fixed-Point Theorem guarantees the accuracy

$$\|x^{(i+1)} - x^*\| \leq \frac{k}{1-k}\epsilon.$$

Of course, not every mapping g satisfies the requirements of this theorem. If g is continuously differentiable, the contraction property can be relatively easily verified.

Theorem 5.2 Let $D \subset \mathbb{R}^n$ and $g \in C^1(D, \mathbb{R}^n)$ with a fixed point $x^* \in D$. For an arbitrary vector norm $\|\cdot\|$, let $A = \{x \in \mathbb{R}^n \,|\, \|x - x^*\| \leq \delta\} \subset D$ for some $\delta > 0$. Then, g satisfies the conditions of the Banach Fixed-Point Theorem with respect to this norm if

$$\max_{x \in A} \|Dg(x)\| =: k < 1,$$

where $Dg(x) \in \mathbb{R}^{n \times n}$ denotes the Jacobian matrix, which is the matrix-valued derivative of the function g at point x, and $\|\cdot\|$ is the matrix norm induced by the given vector norm. In particular, the fixed-point iteration converges for all initial values $x^{(0)} \in A$. □

Proof: Using the Mean Value Theorem of Calculus, we have for all $x, y \in A$ the inequality

$$\|g(x) - g(y)\| \leq \sup_{z \in A} \|Dg(z)\| \|x - y\| \leq k\|x - y\|.$$

Thus, g is a contraction with contraction constant k.

It remains to show that g maps the set A to itself. For this, let $x \in A$, so $\|x - x^*\| \leq \delta$. We need to show that $g(x) \in A$, which means $\|g(x) - x^*\| \leq \delta$ holds. This follows because

$$\|g(x) - x^*\|_\infty = \|g(x) - g(x^*)\|_\infty \leq k\|x - x^*\|_\infty \leq k\delta \leq \delta.$$

□

For the following corollary, in which another sufficient criterion for the convergence of the fixed-point iteration is derived, we recall the spectral radius $\rho(A) = \max_i |\lambda_i(A)|$ of a matrix A introduced in Lemma 2.29.

Corollary 5.3 Let $D \subset \mathbb{R}^n$ and $g \in C^1(D, \mathbb{R}^n)$ with a fixed point $x^* \in \operatorname{int} D$. Suppose $\rho(Dg(x^*)) < 1$ for the spectral radius of the matrix $Dg(x^*)$. Then, there exists a neighborhood A of x^* such that the fixed-point iteration converges to x^* for all initial values $x^{(0)} \in A$.

Proof: As in the proof of Lemma 2.29, the existence of a vector norm $\|\cdot\|$ and the corresponding induced matrix norm is established with

$$k_\epsilon := \|Dg(x^*)\| \leq \rho(Dg(x^*)) + \epsilon < 1.$$

For any sufficiently small ball $A = B_\rho(x^*) = \{x \in \mathbb{R}^n \,|\, \|x - x^*\| < \rho\}$ around x^*, the inequality $\sup_{x \in A} \|Dg(x)\| \leq k_A < 1$ holds due to the continuity of Dg. Thus, the claim follows from Theorem 5.2. □

With Corollary 5.3, the following corollary can be proven for $n = 1$ as well. Here, $g^{-1}(x)$ denotes the inverse function of $g(x)$.

Corollary 5.4 Let $g \in C^1(\mathbb{R}, \mathbb{R})$ with $g(x^*) = x^*$. Suppose $|g'(x^*)| \neq 1$. Then, there exists a neighborhood A of x^* such that one of the two iterations

i) $x^{(i+1)} = g(x^{(i)})$

ii) $x^{(i+1)} = g^{-1}(x^{(i)})$

converges to x^* for all initial values $x^{(0)} \in A$.

Proof: For one-dimensional functions, we have $\rho(g'(x^*)) = |g'(x^*)|$. Since from the assumption, either

$$|g'(x^*)| < 1 \quad \text{or} \quad |(g^{-1})'(x^*)| = \frac{1}{|g'(x^*)|} < 1$$

follows, the claim is established using Corollary 5.3. □

We refer to methods that converge only for initial values in a neighborhood of the desired point as *locally convergent* methods. Examples in which the fixed-point iteration works well can be found in the exercise problems.

The following example shows that the transition to the inverse function as described in the previous theorem is not always practical and introduces another method to address the issue of non-convergence.

Example 5.5 Consider the square root computation from the introductory example with $f(x) = x^2 - 2$. The associated fixed-point mapping is given by $g(x) = x^2 + x - 2$, and the derivative at $x^* = \sqrt{2}$ is $g'(x^*) = 2x^* + 1 \approx 3.8284271$, making the Banach Fixed-Point Theorem inapplicable here. While it is theoretically straightforward to compute the inverse function in this case, i.e., $g^{-1}(x) = \sqrt{x + 9/4} - 1/2$, using this mapping as an iteration rule would require computing a square root in each step. In other words, we would have replaced the computation of *one* square root $\sqrt{2}$ with the computation of *multiple* square roots, which is certainly not efficient.

A simple solution in this case is to scale the original function by $-1/2$: The function $f(x) = -x^2/2+1$ clearly has the same roots as the original f. For the fixed-point mapping $g(x) = -x^2/2 + x + 1$, we have $|g'(x^*)| = |-x^* + 1| \approx 0.41421356$, making the method locally convergent. The following result sequence for the initial value $x^{(0)} = 1$ demonstrates convergence:

```
x( 0) = 1.00000000000000
x( 1) = 1.50000000000000
x( 2) = 1.37500000000000
x( 3) = 1.42968750000000
x( 4) = 1.40768432617188
x( 5) = 1.41689674509689
x( 6) = 1.41309855196381
x( 7) = 1.41467479318270
x( 8) = 1.41402240794944
x( 9) = 1.41429272285787
x(10) = 1.41418076989350
```

Such scaling works "almost" always in one dimension. □

It is worth noting that the fixed-point iteration is one of the few methods that can be used to directly solve general nonlinear equation systems in $\mathbb{R}^n$ without requiring additional information such as derivatives, etc. However, we will see later that the method converges relatively slowly.

5.2 The Bisection Method

In this chapter, we will consider a method that works only in $\mathbb{R}^1$ and cannot be generalized to higher dimensions. The bisection method, which we are about to discuss, is very simple and intuitive and converges globally under minimal conditions.

So, we consider a continuous real-valued function $f : \mathbb{R} \to \mathbb{R}$ and seek a root, that is, a value $x^* \in \mathbb{R}$ with $f(x^*) = 0$. The following algorithm computes such a root.

Algorithm 5.6 (Bisection Method) Given a function $f : \mathbb{R} \to \mathbb{R}$ and values $a < b$ such that $f(a)f(b) < 0$ (i.e., $f(a)$ and $f(b)$ have opposite signs), and a desired accuracy $\epsilon > 0$.

(0) Set $i = 0$ (iteration index) and $a_0 = a$, $b_0 = b$.

(1) Set $x^{(i)} = a_i + (b_i - a_i)/2$.

(2) If $f(x^{(i)}) = 0$ or $(b_i - a_i)/2 < \epsilon$, end the algorithm.

(3) If $f(x^{(i)})f(a_i) < 0$, set $a_{i+1} = a_i$, $b_{i+1} = x^{(i)}$
If $f(x^{(i)})f(a_i) > 0$, set $a_{i+1} = x^{(i)}$, $b_{i+1} = b_i$
Set $i = i + 1$ and go to step (1).

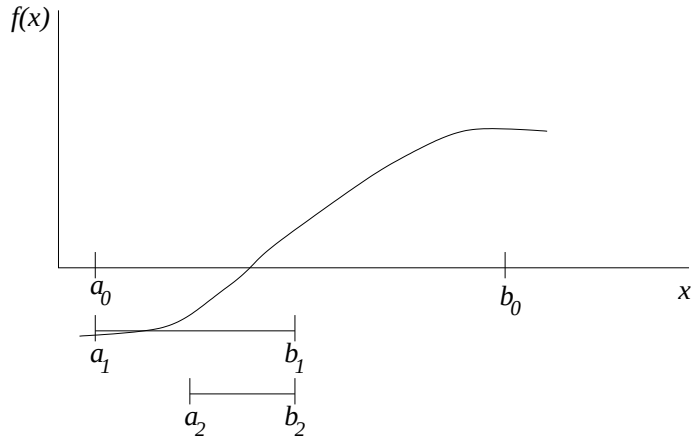


Figure 5.1: Bisection Method

□

Figure 5.1 illustrates this method. The points a_i, b_i are the interval boundaries of the intervals $[a_i, b_i]$, which enclose the root and are successively halved. This is where the name "bisection" comes from (= division into two parts).

The selection of the new values a_{i+1} and b_{i+1} ensures that $f(a_{i+1})$ and $f(b_{i+1})$ have opposite signs. Therefore, since f is continuous, a root must be located between these values. When the termination condition $(x^{(i)} - a_i) < \epsilon$ is reached, it is guaranteed that there is a root x^* with $|x^* - x^{(i)}| < \epsilon$, so $x^{(i)}$ is an approximate root.

The bisection method has some very advantageous properties:

- It works for general continuous functions.
- It always produces a result, provided suitable initial values a and b can be found (it is said to *converge globally*).
- The number of steps to reach the desired accuracy depends only on a and b, not on f.

When applied to the square root computation with $f(x) = x^2 - 2$ and an initial interval of $[1, 2]$, the following values are obtained.

```
i= 0: [1.000000, 2.000000], x( 0)=1.500000
i= 1: [1.000000, 1.500000], x( 1)=1.250000
i= 2: [1.250000, 1.500000], x( 2)=1.375000
i= 3: [1.375000, 1.500000], x( 3)=1.437500
i= 4: [1.375000, 1.437500], x( 4)=1.406250
i= 5: [1.406250, 1.437500], x( 5)=1.421875
i= 6: [1.406250, 1.421875], x( 6)=1.414062
i= 7: [1.414062, 1.421875], x( 7)=1.417969
i= 8: [1.414062, 1.417969], x( 8)=1.416016
```

```
i= 9: [1.414062, 1.416016], x( 9)=1.415039
i=10: [1.414062, 1.415039], x(10)=1.414551
i=11: [1.414062, 1.414551], x(11)=1.414307
i=12: [1.414062, 1.414307], x(12)=1.414185
i=13: [1.414185, 1.414307], x(13)=1.414246
i=14: [1.414185, 1.414246], x(14)=1.414215
i=15: [1.414185, 1.414215], x(15)=1.414200
i=16: [1.414200, 1.414215], x(16)=1.414207
i=17: [1.414207, 1.414215], x(17)=1.414211
i=18: [1.414211, 1.414215], x(18)=1.414213
i=19: [1.414213, 1.414215], x(19)=1.414214
i=20: [1.414213, 1.414214], x(20)=1.414214
i=21: [1.414213, 1.414214], x(21)=1.414213
i=22: [1.414213, 1.414214], x(22)=1.414214
i=23: [1.414214, 1.414214], x(23)=1.414214
```

The reason why other methods are often used in practice, even in one dimension, is that the bisection method, like the fixed-point iteration, converges relatively slowly to the desired value x^*. To understand this, we need to introduce appropriate concepts for measuring convergence speeds.

5.3 Order of Convergence

The order of convergence provides a concept to analyze iterative procedures in terms of their speed. Here, we will consider three different orders of convergence: linear convergence, superlinear convergence, and quadratic convergence.

Iterative procedures generate a sequence of approximate solutions $x^{(i)}$ that converge to the exact solution x^*. The order of convergence is defined based on the error

$$\|x^{(i)} - x^*\|,$$

which indicates how quickly this error converges to zero.

The following definition describes the three types of order of convergence we want to examine here.

Definition 5.7 Consider an iterative procedure that provides a sequence of approximate solutions $x^{(i)}$ for the exact solution x^*. Then, we define the following orders of convergence.

(i) The procedure is called *linearly convergent* if there exists a constant $c \in (0, 1)$ such that the inequality

$$\|x^{(i+1)} - x^*\| \leq c\|x^{(i)} - x^*\| \quad \text{for all } i = 0, 1, 2, \ldots$$

holds.

(ii) The procedure is called *superlinearly convergent* if there exist constants $c_i \in (0,1)$ for $i = 0, 1, 2, \ldots$ such that the conditions

$$c_{i+1} \leq c_i,\ i = 0, 1, 2, \ldots, \quad \lim_{i\to\infty} c_i = 0$$

and the inequality

$$\|x^{(i+1)} - x^*\| \leq c_i \|x^{(i)} - x^*\| \quad \text{for all } i = 0, 1, 2, \ldots$$

hold.

(iii) The procedure is called *quadratically convergent* if there exists a constant $q > 0$ such that the inequality

$$\|x^{(i+1)} - x^*\| \leq q \|x^{(i)} - x^*\|^2 \quad \text{for all } i = 0, 1, 2, \ldots$$

holds.

□

Remark 5.8 By iteratively applying these inequalities, we obtain error estimates as follows.

$$\begin{aligned}
\|x^{(i)} - x^*\| &\leq c^i \|x^{(0)} - x^*\| \\
\|x^{(i)} - x^*\| &\leq \prod_{k=0}^{i-1} c_k \|x^{(0)} - x^*\| \\
\|x^{(i)} - x^*\| &\leq \frac{1}{q} (q\|x^{(0)} - x^*\|)^{2^i}.
\end{aligned}$$

Note that the third inequality provides a meaningful error estimate only when $q\|x^{(0)} - x^*\| < 1$ or $\|x^{(0)} - x^*\| < 1/q$, indicating that the initial value $x^{(0)}$ is already close enough to the exact result x^*. □

Figure 5.2 illustrates the dependence of errors for $c = 0.5$, $c_i = \frac{4}{(i+1)^2+7}$, $q = 2$, and $\|x^{(0)} - x^*\| = 1/4$.

While it can be observed that quadratic convergence approaches zero faster than superlinear, and superlinear converges faster than linear, this graph does not directly reveal the order of convergence for a specific curve.

This can be more easily determined when considering the *logarithm of the error* instead of the error itself. For logarithms, the following rules apply.

$$\log(ab) = \log(a) + \log(b), \quad \log(1/q) = -\log(q), \quad \text{and} \quad \log(c^d) = d \log(c).$$

Thus, for the three types of convergence from Definition 5.7, we obtain the inequalities.

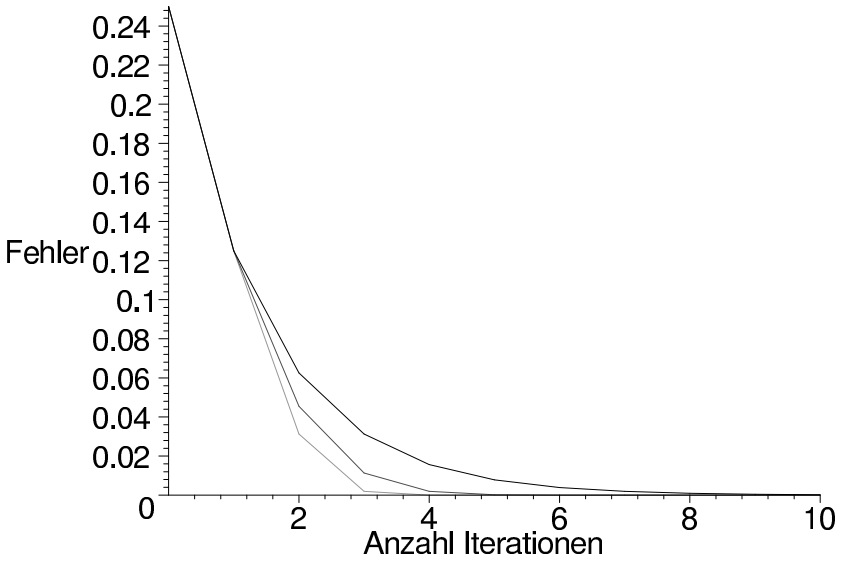


Figure 5.2: Orders of convergence: linear, superlinear, and quadratic (from top to bottom)

$$\begin{aligned}
\log(\|x^{(i+1)} - x^*\|) &\leq \log(\|x^{(i)} - x^*\|) + \log(c) \\
\log(\|x^{(i+1)} - x^*\|) &\leq \log(\|x^{(i)} - x^*\|) + \log(c_i) \\
\log(\|x^{(i+1)} - x^*\|) &\leq 2\log(\|x^{(i)} - x^*\|) + \log(q).
\end{aligned} \tag{5.1}$$

With the abbreviation $K = \log(\|x^{(0)} - x^*\|)$ from Remark 5.8, we have the following estimates.

$$\begin{aligned}
\log(\|x^{(i)} - x^*\|) &\leq i\log(c) + K \\
\log(\|x^{(i)} - x^*\|) &\leq \sum_{k=0}^{i-1} \log(c_k) + K \\
\log(\|x^{(i)} - x^*\|) &\leq 2^i(\log(q) + K) - \log(q).
\end{aligned}$$

Note that the logarithm approaches negative infinity as the error converges to zero.

When graphically representing these last three estimates, in the linear case, you get a straight line with a negative slope, in the quadratic case, you get a curve of the form $i \mapsto -C2^i + D$ for $C > 0$, $D \in \mathbb{R}$, and in the superlinear case, you get a curve in between, with increasing curvature. Figure 5.3 shows the typical behavior of these curves for the base 10 logarithm.

Indeed, these curves would look similar for any other logarithm base, but the logarithm to the base 10 has the special property that you can directly read the accuracy when measuring it in terms of the number of correct decimal places in the result, at least in the one-dimensional case, i.e., for $x \in \mathbb{R}$:

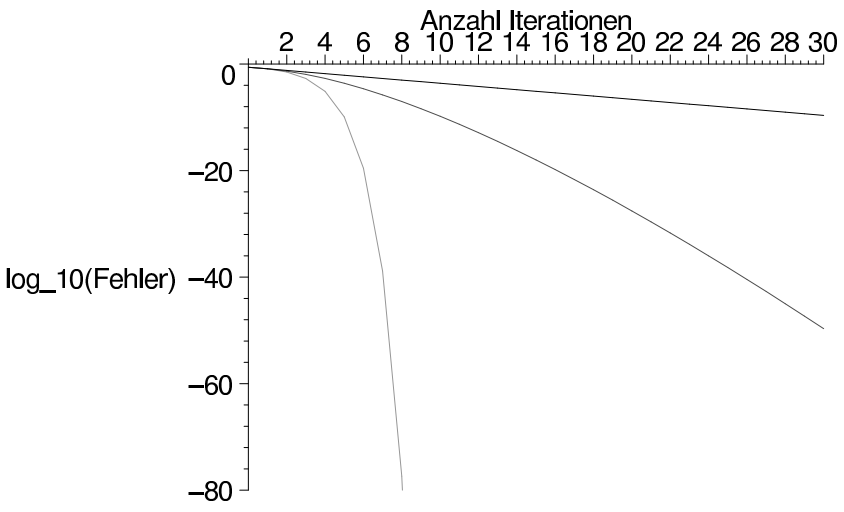


Figure 5.3: Orders of convergence: linear, superlinear, and quadratic (from top to bottom)

To do this, let $d = \log_{10}(|x^{(i)} - x^*|)$. We assume that d is negative, which is the case when $|x^{(i)} - x^*| < 1$ (the following considerations apply only when $x^{(i)}$ is already sufficiently close to x^*). Now, let $m > 0$ be the largest integer that is strictly smaller than $-d$. Then,

$$|x^{(i)} - x^*| = 10^d < 10^{-m} = 0.\underbrace{0\cdots0}_{(m-1)\text{-times}}1.$$

Any positive number smaller than 10^{-m} is of the form

$$0.\underbrace{0\cdots0}_{m\text{-times}} * * * \cdots,$$

where the asterisk "$*$" symbolizes arbitrary digits. So,

$$|x^{(i)} - x^*| \leq 0.\underbrace{0\cdots0}_{m\text{-times}} * * * \cdots,$$

which means that $x^{(i)}$ and x^* must agree in at least the first m decimal places. This way, the number of correct decimal places in the result can be directly determined from d.

From the inequalities (5.1), we can deduce that in linear convergence, the number of correct digits increases by about 1 every $1/(-\log(c))$ steps[1], and in quadratic convergence (neglecting the $\log(q)$ term), it roughly doubles with each step. Superlinear convergence falls between these values: as with linear convergence, the number of correct digits increases by a certain number of steps, but as the iteration progresses, the number of steps required for the increase decreases.

Table 5.1 summarizes the characteristics of the different considered orders of convergence.

Now, we want to determine the convergence order of the procedures we have considered so far.

[1] If $1/(-\log(c)) < 1$, this should be understood as the number of correct digits increasing by $-\log(c)$ on average per step.

Convergence Order	Linear	Superlinear	Quadratic
Definition	$\|x^{(i+1)} - x^*\| \leq c\|x^{(i)} - x^*\|$ for $c \in (0,1)$	$\|x^{(i+1)} - x^*\| \leq c_i\|x^{(i)} - x^*\|$ for $c_i \in (0,1)$, $c_i \searrow 0$	$\|x^{(i+1)} - x^*\| \leq q\|x^{(i)} - x^*\|^2$ for $q > 0$
Curve in log plot	Straight line	Curve with negative curvature	$\approx i \mapsto -C2^i + D$
Number of correct digits	Increases by 1 every $1/(-\log(c))$ steps	Similar to linear, but with increasing speed	Doubles approximately every step

Table 5.1: Characteristics of Different Convergence Orders

For the fixed-point iteration, from the assumptions of the Banach Fixed-Point Theorem, we obtain the error estimate

$$\|x^{(i+1)} - x^*\| = \|g(x^{(i)}) - g(x^*)\| \leq k\|x^{(i)} - x^*\|,$$

resulting in linear convergence with $c = k$. A special case occurs when $Dg(x^*) = 0$ and g is twice continuously differentiable. In this case, using the Taylor expansion around x^* for the components g_j of g, we have the equation

$$g_j(x) - x_j^* = g_j(x) - g_j(x^*) = \frac{1}{2}\sum_{k,l=1}^{n} \frac{\partial^2 g_j(\xi_j)}{\partial x_k \partial x_l}(x_k - x_k^*)(x_l - x_l^*),$$

where x_j^* denotes the j-th component of x^* and ξ_j is a point on the line segment connecting x to x^*. Since the second derivatives of each component g_j are assumed to be continuous, they are bounded for x in a neighborhood N of x^*. Thus, with $q = r/2$, we have

$$\begin{aligned}\|x^{(i+1)} - x^*\|_\infty &= \|g(x^{(i)}) - x^*\|_\infty \\ &= \max_{j=1,\ldots,n} |g_j(x^{(i)}) - x_j^*| \\ &\leq \max_{j=1,\ldots,n} q(x_j^{(i)} - x_j^*)^2 \leq q\|x^{(i)} - x^*\|_\infty^2,\end{aligned}$$

resulting in quadratic convergence.

For the bisection method from Algorithm 5.6, we need to define the error somewhat differently. In this method, it is possible that the value $x^{(i)}$ happens to be very close to the desired root and then moves away in further iteration steps before ultimately converging. In fact, in this case, the error should not be defined in terms of the distance $|x^{(i)} - x^*|$ but rather in terms of the interval size $(b_i - a_i)$, as from the construction, we immediately obtain the estimate $|x^{(i)} - x^*| \leq (b_i - a_i)/2$. This interval size halves in each step; therefore, we have linear convergence with $c = 1/2$.

In the following sections, we will explain procedures that converge quadratically or at least superlinearly.

Remark 5.9 In the chapter on linear systems of equations, we already learned about iterative methods, namely the Jacobi and Gauss-Seidel methods and the CG method — all of which also have a linear order of convergence. □

5.4 The Newton Method

In this section, we will explore another method for solving nonlinear systems of equations, the Newton method, also known as Newton-Raphson method. Compared to the methods we have examined so far (except for fixed-point iteration with a vanishing derivative), this method converges much faster; it is quadratically convergent.

Unlike the bisection method or fixed-point iteration, the Newton method requires not only the function f itself but also its derivative. We will describe the method first in $\mathbb{R}^1$ and then extend it to $\mathbb{R}^n$.

The idea behind the Newton method is as follows: Calculate the tangent $g(x)$ of f at point $x^{(i)}$, which is the line given by

$$g(x) = f(x^{(i)}) + f'(x^{(i)})(x - x^{(i)}).$$

Then, choose $x^{(i+1)}$ as the root of g, which means

$$f(x^{(i)}) + f'(x^{(i)})(x^{(i+1)} - x^{(i)}) = 0 \Rightarrow f'(x^{(i)})x^{(i+1)} = f'(x^{(i)})x^{(i)} - f(x^{(i)}).$$

Solving for $x^{(i+1)}$ then yields

$$x^{(i+1)} = x^{(i)} - \frac{f(x^{(i)})}{f'(x^{(i)})}$$

The idea is illustrated in Figure 5.4.

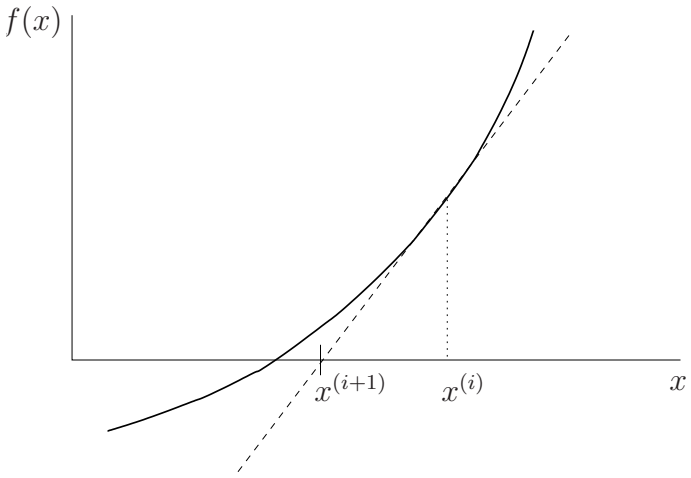


Figure 5.4: Newton Method

Formally, the algorithm in $\mathbb{R}^1$ is as follows.

Algorithm 5.10 (Newton's method) Given a function $f : \mathbb{R} \to \mathbb{R}$, its derivative $f' : \mathbb{R} \to \mathbb{R}$, an initial value $x^{(0)} \in \mathbb{R}$, and a desired accuracy $\varepsilon > 0$. Set $i = 0$.

(1) Calculate $x^{(i+1)} = x^{(i)} - \frac{f(x^{(i)})}{f'(x^{(i)})}$

(2) If $|x^{(i+1)} - x^{(i)}| < \varepsilon$, terminate the algorithm;
otherwise, set $i = i + 1$ and return to step (1).

□

The following example demonstrates the progress of the Newton method for the previously known example in Example 5.5.

Example 5.11 Consider the function $f(x) = x^2 - 2$ with $f'(x) = 2x$ and the root $x^* = \sqrt{2} \approx 1.4142135623731$. The iteration formula for Newton's method is:

$$x^{(i+1)} = \frac{1}{2}x^{(i)} + \frac{1}{x^{(i)}}$$

Let's start with the initial value $x^{(0)} = 2$. The iterations yield (correct decimals are underlined):

$$\begin{array}{lllllll} x^{(1)} & = & \frac{1}{2} \cdot 2 + \frac{1}{2} & = & \frac{3}{2} & = & 1.5 \\ x^{(2)} & = & \frac{1}{2} \cdot \frac{3}{2} + \frac{1}{\frac{3}{2}} & = & \frac{17}{12} & = & 1.\underline{41}\overline{6} \\ x^{(3)} & = & \frac{1}{2} \cdot \frac{17}{12} + \frac{1}{\frac{17}{12}} & = & \frac{577}{408} & \approx & 1.\underline{4142}156862745 \\ x^{(4)} & = & \frac{1}{2} \cdot \frac{577}{408} + \frac{1}{\frac{577}{408}} & = & \frac{665857}{470832} & \approx & 1.\underline{4142135623746} \end{array}$$

□

In contrast to the bisection method, the Newton method can be extended to systems of nonlinear equations in $\mathbb{R}^n$. To this end, we have to find a meaningful generalization of the one-dimensional iteration formula of the Newton method for $f : \mathbb{R} \to \mathbb{R}$, i.e. for

$$x^{(i+1)} = x^{(i)} - \frac{f(x^{(i)})}{f'(x^{(i)})}.$$

Let $f : \mathbb{R}^n \to \mathbb{R}^n$ be a vector-valued function. The derivative at a point $x \in \mathbb{R}^n$, denoted as usual by $Df(x)$, is now not a real number but a matrix.

Certainly, we cannot simply substitute $Df(x^{(i)})$ into the iteration formula for $x^{(i+1)}$, as we cannot divide by a matrix. So, instead of dividing by $Df(x^{(i)})$, we multiply by its inverse, which means we calculate $[Df(x^{(i)})]^{-1} f(x^{(i)})$. This leads to the following algorithm.

Algorithm 5.12 (Newton's method in $\mathbb{R}^n$, Version 1) Given a function $f : \mathbb{R}^n \to \mathbb{R}^n$, its derivative $Df : \mathbb{R}^n \to \mathbb{R}^{n \times n}$, an initial value $x^{(0)} \in \mathbb{R}^n$, and a desired accuracy $\varepsilon > 0$. Set $i = 0$.

(1) Calculate $x^{(i+1)} = x^{(i)} - [Df(x^{(i)})]^{-1} f(x^{(i)})$

(2) If $\|x^{(i+1)} - x^{(i)}\| < \varepsilon$, terminate the algorithm; otherwise, set $i = i + 1$ and return to step (1).

□

We illustrate the process of this algorithm with the following example.

Example 5.13 We are looking for a solution to the nonlinear system of equations:

$$\begin{aligned} x_1^2 + x_2^2 &= 1 \\ x_1 &= 0 \end{aligned}$$

This is equivalent to finding a root $x^* \in \mathbb{R}^2$ of the function $f : \mathbb{R}^2 \to \mathbb{R}^2$ given by:

$$f(x) = \begin{pmatrix} x_1^2 + x_2^2 - 1 \\ x_1 \end{pmatrix}$$

For the desired solution x^*, we must simultaneously have $f_1(x^*) = 0$ and $f_2(x^*) = 0$. For the given function f, the solution is easy to see: f_1 is equal to zero when $x_1^2 + x_2^2 = 1$, which means $\|x\| = 1$. And f_2 is equal to zero when $x_1 = 0$. Therefore, the set of possible solutions consists of all vectors $(x_1, x_2)^T$ with $\|x\| = 1$ and $x_1 = 0$, which means $x^* = (0, 1)^T$ or $x^* = (0, -1)^T$.

The partial derivatives of f_1 and f_2 are:

$$\frac{\partial f_1}{\partial x_1}(x) = 2x_1, \quad \frac{\partial f_1}{\partial x_2}(x) = 2x_2, \quad \frac{\partial f_2}{\partial x_1}(x) = 1, \quad \frac{\partial f_2}{\partial x_2}(x) = 0.$$

So, the Jacobian matrix $Df(x)$ is

$$Df(x) = \begin{pmatrix} 2x_1 & 2x_2 \\ 1 & 0 \end{pmatrix}$$

For example, for $x = (0, 1)^T$, the Jacobian is

$$Df(x) = \begin{pmatrix} 0 & 2 \\ 1 & 0 \end{pmatrix}$$

The inverse of the Jacobian is given by

$$Df(x)^{-1} = \begin{pmatrix} 0 & 1 \\ \frac{1}{2x_2} & -\frac{x_1}{x_2} \end{pmatrix}$$

With $x^{(i)} = (x_1^{(i)}, x_2^{(i)})^T$, the iteration formula is then given by

$$x^{(i+1)} = x^{(i)} - \begin{pmatrix} 0 & 1 \\ \frac{1}{2x_2^{(i)}} & -\frac{x_1^{(i)}}{x_2^{(i)}} \end{pmatrix} \begin{pmatrix} (x_1^{(i)})^2 + (x_2^{(i)})^2 - 1 \\ x_1^{(i)} \end{pmatrix}.$$

With an initial value of $x^{(0)} = (1,1)^T$, we obtain the following iterations (correct decimal places are underlined)

$$\begin{aligned}
x^{(1)} &= \begin{pmatrix}1\\1\end{pmatrix} - \begin{pmatrix}0 & 1\\ \frac{1}{2} & -1\end{pmatrix}\begin{pmatrix}1\\1\end{pmatrix} &&= \begin{pmatrix}0\\ \frac{3}{2}\end{pmatrix} &&= \begin{pmatrix}0\\ 1.\underline{5}\end{pmatrix}\\
x^{(2)} &= \begin{pmatrix}0\\ \frac{3}{2}\end{pmatrix} - \begin{pmatrix}0 & 1\\ \frac{1}{3} & 0\end{pmatrix}\begin{pmatrix}\frac{5}{4}\\0\end{pmatrix} &&= \begin{pmatrix}0\\ \frac{13}{12}\end{pmatrix} &&= \begin{pmatrix}0\\ 1.0\underline{8}\overline{3}\end{pmatrix}\\
x^{(3)} &= \begin{pmatrix}0\\ \frac{13}{12}\end{pmatrix} - \begin{pmatrix}0 & 1\\ \frac{6}{13} & 0\end{pmatrix}\begin{pmatrix}\frac{25}{144}\\0\end{pmatrix} &&= \begin{pmatrix}0\\ \frac{313}{312}\end{pmatrix} &&\approx \begin{pmatrix}0\\ 1.\underline{00}32051282\end{pmatrix}\\
x^{(4)} &= \begin{pmatrix}0\\ \frac{313}{312}\end{pmatrix} - \begin{pmatrix}0 & 1\\ \frac{156}{313} & 0\end{pmatrix}\begin{pmatrix}\frac{625}{97344}\\0\end{pmatrix} &&= \begin{pmatrix}0\\ \frac{195313}{195312}\end{pmatrix} &&\approx \begin{pmatrix}0\\ 1.\underline{00000}51200\end{pmatrix}
\end{aligned}$$

□

In practice, you would not calculate $[Df(x)]^{-1}$ by hand but use a numerical routine. Numerically, however, it is not very efficient to compute the inverse of the matrix $Df(x^{(i)})$. Instead, you can solve the linear equation system $Df(x^{(i)})\Delta x^{(i)} = f(x^{(i)})$, which provides a vector $\Delta x^{(i)}$ such that $\Delta x^{(i)} = [Df(x^{(i)})]^{-1} f(x^{(i)})$. This leads to the following more efficient version of the Newton method.

Algorithm 5.14 (Newton's method in $\mathbb{R}^n$, Version 2)
Given a function $f : \mathbb{R}^n \to \mathbb{R}^n$, its derivative $Df : \mathbb{R}^n \to \mathbb{R}^{n\times n}$, an initial value $x^{(0)} \in \mathbb{R}^n$, and a desired accuracy $\epsilon > 0$. Set $i = 0$.

(1) Solve the system of linear equations $Df(x^{(i)})\Delta x^{(i)} = f(x^{(i)})$ and calculate $x^{(i+1)} = x^{(i)} - \Delta x^{(i)}$.

(2) If $\|\Delta x^{(i)}\| < \epsilon$, terminate the algorithm. Otherwise, set $i = i + 1$ and go to step (1).

□

The following theorem shows the convergence properties of the Newton method.

Theorem 5.15 Let D be an open and convex set in $\mathbb{R}^n$, and let $f : D \to \mathbb{R}^n$ be continuously differentiable with invertible Jacobian $Df(x)$ for all $x \in D$. Assume there exists $\omega > 0$ such that the following *affine-invariant Lipschitz condition* holds:

$$\|Df(x)^{-1}(Df(x + sv) - Df(x))v\| \le s\omega\|v\|^2$$

for all $s \in [0,1]$, all $x \in D$, and all $v \in \mathbb{R}^n$ with $x + v \in D$. Let $x^* \in D$ be a root of f. Then, for all initial values $x^{(0)} \in \mathbb{R}^n$ with

$$\rho := \|x^* - x^{(0)}\| < \frac{2}{\omega} \quad \text{and} \quad B_\rho(x^*) = \{x \in \mathbb{R}^n \mid \|x - x^*\| < \rho\} \subseteq D,$$

the iterates $x^{(i)}$ generated by Newton's method remain in the ball $B_\rho(x^*)$ for all $i \in \mathbb{N}$ and converge to x^*, i.e.,

$$\|x^{(i)} - x^*\| < \rho \text{ for all } i > 0 \quad \text{and} \quad \lim_{i\to\infty} x^{(i)} = x^*.$$

The convergence rate can be estimated as follows:

$$\|x^{(i+1)} - x^*\| \leq \frac{\omega}{2}\|x^{(i)} - x^*\|^2,$$

meaning that the method converges locally quadratically. Furthermore, from the given conditions, it follows that x^* is the unique root in $B_{2/\omega}(x^*)$.

Proof: First, we prove the following auxiliary statement for all $x, y \in D$:

$$\|Df(x)^{-1}(f(y) - f(x) - Df(x)(y-x))\| \leq \frac{\omega}{2}\|y - x\|^2 \tag{5.2}$$

To prove (5.2), we use the mean value theorem of calculus in $\mathbb{R}^n$. According to this theorem, we have:

$$f(y) - f(x) - Df(x)(y-x) = \int_0^1 (Df(x + s(y-x)) - Df(x))(y-x)ds.$$

Exploiting the affine-invariant Lipschitz condition, we obtain

$$\begin{aligned}
&\|Df(x)^{-1}(f(y) - f(x) - Df(x)(y-x))\| \\
&= \left\| Df(x)^{-1}\left(\int_0^1 (Df(x + s(y-x)) - Df(x))(y-x)ds\right)\right\| \\
&\leq \int_0^1 s\omega\|y-x\|^2 ds \;=\; \frac{\omega}{2}\|y-x\|^2,
\end{aligned}$$

which proves (5.2).

Now, from the iteration formula and $f(x^*) = 0$, we obtain

$$\begin{aligned}
x^{(i+1)} - x^* &= x^{(i)} - Df(x^{(i)})^{-1}f(x^{(i)}) - x^* \\
&= x^{(i)} - x^* - Df(x^{(i)})^{-1}(f(x^{(i)}) - f(x^*)) \\
&= Df(x^{(i)})^{-1}(f(x^*) - f(x^{(i)}) - Df(x^{(i)})(x^* - x^{(i)})).
\end{aligned}$$

Using (5.2), this yields:

$$\|x^{(i+1)} - x^*\| \leq \frac{\omega}{2}\|x^{(i)} - x^*\|^2,$$

which proves the claimed estimate. If $\|x^{(i)} - x^*\| < \rho$ holds, then

$$\|x^{(i+1)} - x^*\| \leq \frac{\omega}{2}\|x^{(i)} - x^*\|\,\|x^{(i)} - x^*\| < \underbrace{\rho\omega/2}_{<1}\|x^{(i)} - x^*| < \rho,$$

which means that for $\|x^{(0)} - x^*\| < \rho$ the sequence remains in $B_\rho(x^*)$ and converges to x^*.

To prove the uniqueness of x^* in $B_{2/\omega}(x^*)$, let x^{**} be another root of f in this ball. Then $\|x^{**} - x^*\| < 2/\omega$ holds. Substituting this into (5.2), we get

$$\|x^{**} - x^*\| = \|Df(x^*)^{-1}(0 - 0 - Df(x^*)(x^{**} - x^*))\| \leq \underbrace{\frac{\omega}{2}\|x^{**} - x^*\|}_{<1} \|x^{**} - x^*\|,$$

which is only possible if $x^{**} = x^*$. □

Remark 5.16 If f is twice continuously differentiable with an invertible Jacobi matrix, then the affine-invariant Lipschitz condition is always satisfied in a (sufficiently small) neighborhood of x^*, since in this case, $\|Df(x + sv) - Df(x)\| \leq Cs\|v\|$ holds, from which the given condition follows with $\omega = \|Df(x)^{-1}\|C$. In this case, local quadratic convergence (i.e., quadratic convergence for $x^{(0)}$ sufficiently close to x^*) is assured. □

In Theorem 5.15, we used the assumption "$\|x^* - x^{(0)}\| \leq 2/\omega$". This is not a condition introduced solely for proof purposes: In fact, there can be situations where the Newton method does not converge or converges very slowly with an unsuitable initial value $x^{(0)}$. The following example illustrates this.

Example 5.17 Consider the function $f(x) = 5x/4 - x^3/4$ with derivative $f'(x) = 5/4 - 3x^2/4$. Clearly, this function has a zero at $x^* = 0$ with $f'(0) = 5/4 \neq 0$, and it is infinitely continuously differentiable. The iteration formula in this case is given by

$$x^{(i+1)} = x^{(i)} - \frac{5x^{(i)}/4 - (x^{(i)})^3/4}{5/4 - 3(x^{(i)})^2} = \frac{2(x^{(i)})^3}{-5 + 3(x^{(i)})^2}.$$

With the initial value $x^{(0)} = 1$, we obtain

$$\begin{aligned} x^{(1)} &= -1, \\ x^{(2)} &= 1, \\ x^{(3)} &= -1, \\ x^{(4)} &= 1, \\ &\vdots \end{aligned}$$

The method oscillates indefinitely between 1 and -1 and does not converge. □

Thus, the method is indeed only locally convergent, emphasizing the importance of having a reasonable initial value $x^{(0)}$ for the iteration. In the book by Deuflhard/Hohmann, a method is described to identify early on whether convergence is likely or not.

A significant drawback of the Newton method is the requirement for the derivative of the function f in the iteration formula. In practice, the derivative can be replaced by a suitable numerical approximation, avoiding the explicit computation of $Df(x^{(i)})$. For

instance, replacing $f'(x)$ with $(f(x + f(x)) - f(x))/f(x)$ yields the so-called *Steffensen method* in $\mathbb{R}^1$.

The Newton method can also be used to find extremal points of functions $g : \mathbb{R}^n \to \mathbb{R}$ by setting $f := \nabla g : \mathbb{R}^n \to \mathbb{R}^n$. In this case, the Jacobi matrix Df equals the Hessian matrix D^2g, and to avoid the exact computation of D^2g, the so-called BFGS method can be employed. Iteratively, approximations $B^{(i)} \approx D^2g(x^{(i)})$ are generated from an initial estimate $B^{(0)} \approx D^2g(x^{(0)})$.

Since the numerical approximation of derivatives is generally sensitive to rounding errors, the numerical stability of such methods must be carefully examined, preferably with additional information about f.

In the next section, we will study an alternative derivative-free method that works in $\mathbb{R}^1$ in more detail.

5.5 The Secant Method

The secant method is another method that works only in $\mathbb{R}^1$. It is similar to but slightly different from Newton's method, as the new approximation $x^{(i+1)}$ is calculated not only from $x^{(i)}$ but also from the two preceding values $x^{(i-1)}$ and $x^{(i)}$.

First, we provide an intuitive description, followed by the formal one.

The idea behind this method is as follows: initially, we consider the secant line $g(x)$ through $f(x^{(i-1)})$ and $f(x^{(i)})$, which means the line:

$$g(x) = f(x^{(i)}) + \frac{(x^{(i)} - x)}{x^{(i)} - x^{(i-1)}} \left(f(x^{(i-1)}) - f(x^{(i)}) \right)$$

We then choose $x^{(i+1)}$ as the root of this line, so:

$$\begin{aligned} 0 &= f(x^{(i)}) + \frac{(x^{(i)} - x^{(i+1)})}{x^{(i)} - x^{(i-1)}} \left(f(x^{(i-1)}) - f(x^{(i)}) \right) \\ \Leftrightarrow \quad x^{(i)} - x^{(i+1)} &= -f(x^{(i)}) \frac{x^{(i)} - x^{(i-1)}}{f(x^{(i-1)}) - f(x^{(i)})} \\ \Leftrightarrow \quad x^{(i+1)} &= x^{(i)} - f(x^{(i)}) \frac{x^{(i)} - x^{(i-1)}}{f(x^{(i)}) - f(x^{(i-1)})} \end{aligned}$$

The idea is graphically illustrated in Figure 5.5.

Formally, the method can be described as follows.

Algorithm 5.18 (Secant Method) Let $f : \mathbb{R} \to \mathbb{R}$ be a function, and let $x^{(0)} \in \mathbb{R}$, $x^{(1)} \in \mathbb{R}$, and a desired accuracy $\epsilon > 0$ be given. Set $i = 1$.

(1) Calculate $x^{(i+1)} = x^{(i)} - \dfrac{(x^{(i)} - x^{(i-1)}) f(x^{(i)})}{f(x^{(i)}) - f(x^{(i-1)})}$

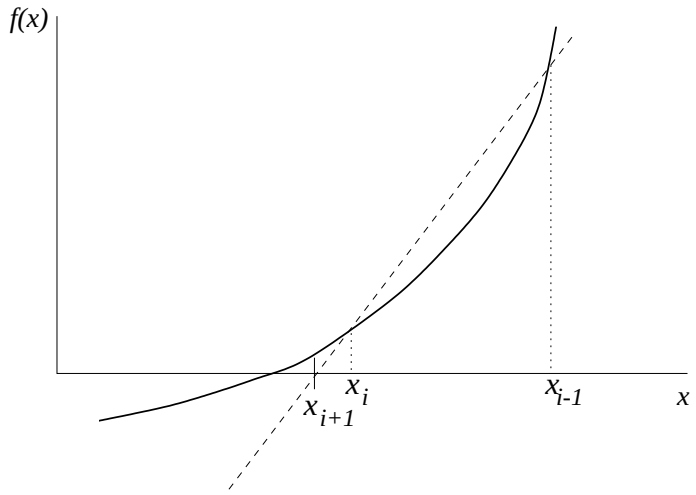


Figure 5.5: Secant Method

(2) If $|x^{(i+1)} - x^{(i)}| < \epsilon$, end the algorithm. Otherwise, set $i = i + 1$ and go to (1).

□

The following theorem shows the convergence order of this method.

Theorem 5.19 Let $f : \mathbb{R} \to \mathbb{R}$ be twice continuously differentiable with $f(x^*) = 0$, and let $x^{(0)}$ and $x^{(1)}$ be sufficiently close to x^*. Furthermore, assume $f'(x^*) \neq 0$. Then, the secant method converges superlinearly. □

Sketch of proof: When $x^{(i)}$ and $x^{(i-1)}$ are sufficiently close to x^*, the Taylor expansion of f around x^* yields the estimate:

$$f(x^{(i)}) - f(x^{(i-1)}) \approx f'(x^*)(x^{(i)} - x^{(i-1)}) \tag{5.3}$$

and for $j = i - 1$ and $j = i$, we have the approximations:

$$f(x^{(j)}) \approx f'(x^*)(x^{(j)} - x^*) + \frac{1}{2} f''(x^*)(x^{(j)} - x^*)^2. \tag{5.4}$$

From the iteration formula, we obtain the equation:

$$x^{(i+1)} - x^* = \frac{(x^{(i-1)} - x^*)f(x^{(i)}) - (x^{(i)} - x^*)f(x^{(i-1)})}{f(x^{(i)}) - f(x^{(i-1)})}.$$

Using (5.3), we get the approximation for the denominator:

$$f(x^{(i)}) - f(x^{(i-1)}) \approx f'(x^*)(x^{(i)} - x^{(i-1)})$$

and using (5.4), we obtain for the enumerator:

$$\begin{aligned}
&(x^{(i-1)} - x^*)f(x^{(i)}) - (x^{(i)} - x^*)f(x^{(i-1)}) \\
&\quad \approx (x^{(i-1)} - x^*)\left(f'(x^*)(x^{(i)} - x^*) + \frac{1}{2}f''(x^*)(x^{(i)} - x^*)^2\right) \\
&\qquad - (x^{(i)} - x^*)\left(f'(x^*)(x^{(i-1)} - x^*) + \frac{1}{2}f''(x^*)(x^{(i-1)} - x^*)^2\right) \\
&\quad = \frac{1}{2}f''(x^*)(x^{(i-1)} - x^*)(x^{(i)} - x^*)\left((x^{(i)} - x^*) - (x^{(i-1)} - x^*)\right) \\
&\quad = \frac{1}{2}f''(x^*)(x^{(i-1)} - x^*)(x^{(i)} - x^*)(x^{(i)} - x^{(i-1)}).
\end{aligned}$$

Thus, we have:

$$|x^{(i+1)} - x^*| \approx \underbrace{\frac{1}{2}\left|\frac{f''(x^*)}{f'(x^*)}\right| |x^{(i-1)} - x^*|}_{\approx c_i} |x^{(i)} - x^*|$$

and this proves the claimed superlinear convergence.

We repeat Example 5.5 and 5.11 for this method.

Example 5.20 Consider once again the function $f(x) = x^2 - 2$ with a root $x^* = \sqrt{2} \approx 1.4142135623731$. The iteration formula for the Secant method is given by

$$\begin{aligned}
x^{(i+1)} &= x^{(i)} - \frac{(x^{(i)} - x^{(i-1)})((x^{(i)})^2 - 2)}{(x^{(i)})^2 - (x^{(i-1)})^2} \\
&= x^{(i)} - \frac{(x^{(i)} - x^{(i-1)})((x^{(i)})^2 - 2)}{(x^{(i)} - x^{(i-1)})(x^{(i)} + x^{(i-1)})} \\
&= x^{(i)} - \frac{(x^{(i)})^2 - 2}{x^{(i)} + x^{(i-1)}}.
\end{aligned}$$

With $x^{(0)} = 2$ and $x^{(1)} = 1$, we obtain the following values (correct decimal places are underlined):

$$\begin{aligned}
x^{(2)} &= 1 - \tfrac{1^2-2}{1+2} &&= \tfrac{4}{3} &&= 1.\overline{3} \\
x^{(3)} &= \tfrac{4}{3} - \tfrac{\left(\frac{4}{3}\right)^2-2}{\frac{4}{3}+1} &&= \tfrac{10}{7} &&= 1.\underline{4}\overline{285714} \\
x^{(4)} &= \tfrac{10}{7} - \tfrac{\left(\frac{10}{7}\right)^2-2}{\frac{10}{7}+\frac{4}{3}} &&= \tfrac{41}{29} &&\approx 1.\underline{41}37931034483 \\
x^{(5)} &= \tfrac{41}{29} - \tfrac{\left(\frac{41}{29}\right)^2-2}{\frac{41}{29}+\frac{10}{7}} &&= \tfrac{816}{577} &&\approx 1.\underline{4142}114384749 \\
x^{(6)} &= \tfrac{816}{577} - \tfrac{\left(\frac{816}{577}\right)^2-2}{\frac{816}{577}+\frac{41}{29}} &&= \tfrac{66922}{47321} &&\approx 1.\underline{41421356}26889
\end{aligned}$$

The Secant method converges significantly faster than linear but slower than the Newton method. □

The Secant method is also locally convergent, meaning that the initial values must be sufficiently close to x^* to guarantee convergence of the method. In such cases, the Bisection method can be a useful strategy to find good initial values near x^*. A strategy of this kind is employed, for instance, in the root-finding routine `fzero` in MATLAB.

5.6 The Gauss-Newton Method for Nonlinear Least Squares Problems

To conclude this chapter, we want to revisit the least squares problem introduced in Section 2.1. We initially considered the linear least squares problem, which we will summarize here with slightly different notation:

Given a matrix $A \in \mathbb{R}^{m\times n}$ with $m > n$ and a vector $z \in \mathbb{R}^m$, find the vector $x \in \mathbb{R}^n$ that minimizes the squared norm:

$$\|Ax - z\|_2^2$$

In our theoretical analysis, we found that this vector x is the solution to the normal equations:

$$A^T Ax = A^T z$$

These equations represent a "standard" linear system in $\mathbb{R}^n$ and can be solved numerically, for example, using the Cholesky decomposition.

For numerical solutions, we also introduced the QR decomposition in Algorithm 2.21, which leads to a factorization $A = QR$ with:

$$R = \begin{pmatrix} \overline{R} \\ 0 \end{pmatrix}$$

where $\overline{R} \in \mathbb{R}^{n\times n}$ is an upper triangular matrix. We can then compute the vector x through backward substitution. This approach is often computationally more robust than solving the normal equations, especially when the matrix $A^T A$ is ill-conditioned.

However, the normal equations provide a way to theoretically analyze the solutions of the linear least squares problem. To this end, we introduce the following definition.

Definition 5.21 Let $m \geq n$, and $A \in \mathbb{R}^{m\times n}$ be a full-rank matrix. We define the *pseudo inverse* $A^+ \in \mathbb{R}^{n\times m}$ of A as

$$A^+ = (A^T A)^{-1} A^T.$$

□

It is easy to see that the solution to the linear least squares problem $\min \|Ax - z\|_2^2$ can be expressed as

$$x = A^+ z.$$

This equivalence holds because

$$x = A^+ z \iff x = (A^T A)^{-1} A^T z \iff A^T Ax = A^T z.$$

In other words, $x = A^+ z$ solves the normal equations.

Remark 5.22 One can verify that $A^+ = (A^TA)^{-1}A^T \in \mathbb{R}^{n\times m}$ satisfies the following properties, known as the *Penrose axioms*.

$$\begin{array}{llll} (i) & (A^+A)^T = A^+A & (ii) & (AA^+)^T = AA^+ \\ (iii) & A^+AA^+ = A^+ & (iv) & AA^+A = A \end{array}$$

In fact, A^+ is the *unique* $n \times m$ matrix that satisfies these axioms: For any matrix A^+ that satisfies these axioms, the linear mappings $P := A^+A$ and $\overline{P} := AA^+$ are orthogonal projections, i.e., they satisfy $P^T = P = P^2$ and $\overline{P}^T = \overline{P} = \overline{P}^2$. From (iv) it follows that $\overline{P}A = A$, which means the image of $\overline{P}$ contains the image of A. However, since im$A$ has dimension rk$A = n$, and $\overline{P}$ as the product of an $n \times m$ and an $m \times n$ matrix has an image with dimension $\leq n$, the images must coincide. Since for an orthogonal projection, the image and kernel are orthogonal to each other (because for $x \in \text{im}\overline{P}$, $y \in \ker\overline{P}$, we have $\langle x, y\rangle = \langle \overline{P}x, y\rangle = \langle x, \overline{P}^T y\rangle = \langle x, \overline{P}y\rangle = \langle x, 0\rangle = 0$), $\overline{P}$ is uniquely determined by its image and therefore independent of the choice of A^+.

Furthermore, A^+ is surjective because from (iv) and the dimension of A^+, we have

$$n = \dim \text{im}\, A = \dim \text{im}\, AA^+A \leq \dim \text{im}\, A^+ \leq n.$$

From (iii) it follows that $PA^+ = A^+$, so dim im$P = n$, and thus $P = \text{Id}$. Therefore, P is independent of A^+, too. If A_1^+ and A_2^+ are two matrices that satisfy the four axioms, then it must hold that $A_1^+A = P = A_2^+A$ and $AA_1^+ = \overline{P} = AA_2^+$. This implies

$$A_1^+ \overset{\text{(iii)}}{=} \underbrace{A_1^+A}_{=A_2^+A} A_1^+ = A_2^+ \underbrace{AA_1^+}_{=AA_2^+} = A_2^+AA_2^+ \overset{\text{(iii)}}{=} A_2^+.$$

□

Remark 5.23 The pseudoinverse can be represented using the *singular value decomposition* of the matrix A. This states that for every matrix $A \in \mathbb{R}^{m\times n}$, there exist two orthogonal matrices $U \in \mathbb{R}^{m\times m}$ and $V \in \mathbb{R}^{n\times n}$ (i.e., $U^TU = \text{Id}$, $V^TV = \text{Id}$) and a matrix $S \in \mathbb{R}^{m\times n}$ in which only the diagonal elements s_{ii} can take non-zero values, which satisfy

$$A = USV^T.$$

The diagonal entries of S are unique (up to permutation) and coincide with the square roots of the eigenvalues of the matrix A^TA, called the *singular values* of A. A proof for the existence of U, S, and V as described above can be found in many linear algebra textbooks, e.g., in [6].

Since U and V are invertible as orthogonal matrices, the matrix A has full rank if and only if matrix S has full rank, which in turn happens if and only if all diagonal elements satisfy $s_{ii} \neq 0$. Thus, S^TS is a square diagonal matrix with entries s_{ii}^2, so $(S^TS)^{-1}$ is a diagonal matrix with entries $1/s_{ii}^2$, and $S^+ = (S^TS)^{-1}S$ has non-zero entries $s_{ii}^+ = 1/s_{ii}$. Therefore, the pseudoinverse of A is given by

$$\begin{aligned} A^+ &= (A^TA)^{-1}A^T = ((USV^T)^TUSV^T)^{-1}(USV^T)^T \\ &= (VS^TU^TUSV^T)^{-1}VSU^T = V(S^TS)^{-1}V^TVSU^T = VS^+U^T. \end{aligned}$$

□

The computation of the pseudoinverse through the singular value decomposition is numerically more stable than using the formula from Definition 5.21. However, as we will see in the following, we can completely avoid the explicit use of A^+ when solving nonlinear least squares problems. The pseudoinverse A^+ primarily simplifies the theoretical representation of the solution to the least squares problem. It plays an important role in the generalization of the problem to the nonlinear least squares problem.

The nonlinear least squares problem, like in the linear case, is given as a minimization problem. For $D \subseteq \mathbb{R}^n$ and $m > n$, consider a twice continuously differentiable mapping

$$f : D \to \mathbb{R}^m$$

and aim to solve the problem

$$\text{minimize } g(x) := \|f(x)\|_2^2 \text{ over } x \in D \tag{5.5}$$

As in the linear least squares problem, the interpretation and application of this problem are as follows: Let z_i be measurements corresponding to parameters t_i for $i = 1, \ldots, m$. Based on theoretical considerations (e.g., an underlying physical law), it is known that there exists a function $h : \mathbb{R} \times \mathbb{R}^n \to \mathbb{R}$ such that the (ideal) measurements for an appropriate parameter vector $x^* \in \mathbb{R}^n$ satisfy the equation $h(t_i, x^*) = z_i$. With

$$f(x) = \begin{pmatrix} h(t_1, x) - z_1 \\ \vdots \\ h(t_m, x) - z_m \end{pmatrix}$$

the equation $f(x^*) = 0$ would hold. However, since we must consider measurement errors, this equation is typically only approximately satisfied. Therefore, we seek a solution to the least squares problem (5.5).

An example of such an application is provided by data for population growth. Here, under ideal conditions and with unbounded resources, an exponential growth of the form $z = h(t, x) = x_1 e^{x_2 t}$ is expected, which is a nonlinear function of $x = (x_1, x_2)^T$. In case resources are limited, the logistic growth given by

$$h(t, x) = \frac{x_3}{1 + \left(\frac{x_3}{x_1} - 1\right) \exp(-x_2 t)}$$

with $x = (x_1, x_2, x_3)^T$ is usually more appropriate. We will show in the lecture how well these functions match real data of the human population on earth.

We will first derive the algorithm for solving nonlinear least squares problems informally, then write it down formally, and finally formulate and prove the exact convergence properties.

In solving the problem (5.5), we consider only local minima within the interior of D, excluding minimum points on the boundary ∂D. Furthermore, we restrict ourselves to local minima $x^* \in D$ of $g : \mathbb{R}^n \to \mathbb{R}$ that satisfy the sufficient conditions

$$Dg(x^*) = 0 \text{ and } D^2 g(x^*) \text{ is positive definite} \tag{5.6}$$

The derivative of $g(x) = \|f(x)\|_2^2 = f(x)^T f(x)$ satisfies

$$Dg(x)^T = 2Df(x)^T f(x),$$

so to find candidate minimum points, we need to solve the nonlinear equation system

$$G(x) := Df(x)^T f(x) = 0 \tag{5.7}$$

If we use the Newton's method for this purpose, we obtain the iteration $x^{(i+1)} = x^{(i)} - \Delta x^{(i)}$ with iteratively solvable equation systems

$$DG(x^{(i)})\Delta x^{(i)} = G(x^{(i)}), \quad i = 0, 1, 2, \ldots, \tag{5.8}$$

where

$$DG(x) = Df(x)^T Df(x) + D^2 f(x)^T f(x).$$

If a local minimum x^* with (5.6) exists, then $DG(x) = \frac{1}{2} D^2 g(x)^T$ is positive definite at $x = x^*$, and therefore also for all x in a neighborhood of x^*. This implies that $DG(x)$ is invertible, and the Newton's method is applicable.

If the least squares problem is indeed a solvable system of equations (i.e., in the above interpretation, there are no measurement errors), then $f(x^*) = 0$, and the problem is called *compatible*. In this case, we have

$$DG(x^*) = Df(x^*)^T Df(x^*).$$

Even though compatibility is an ideal case and rarely occurs in practice, we will assume for simplicity that $f(x^*)$ is approximately equal to zero for x near x^*, i.e.,

$$DG(x) \approx Df(x)^T Df(x)$$

Based on this informal consideration, we replace the derivative $DG(x^{(i)})$ with the term $Df(x^{(i)})^T Df(x^{(i)})$ in the iteration scheme (5.8). This allows us to avoid using the second derivative, resulting in

$$Df(x^{(i)})^T Df(x^{(i)})\Delta x^{(i)} = G(x^{(i)}) = Df(x^{(i)})^T f(x^{(i)}), \quad i = 0, 1, 2, \ldots \tag{5.9}$$

These are precisely the normal equations for the linear least squares problem

$$\text{minimize } \|Df(x^{(i)})\Delta x^{(i)} - f(x^{(i)})\|_2^2$$

with the solution

$$\Delta x^{(i)} = Df(x^{(i)})^+ f(x^{(i)}),$$

which we use for the iteration. This leads to the following algorithm.

Algorithm 5.24 (Gauss-Newton Method)
Given a function $f : \mathbb{R}^n \to \mathbb{R}^m$, its derivative $Df : \mathbb{R}^n \to \mathbb{R}^{m \times n}$, an initial value $x^{(0)} \in \mathbb{R}^n$, and a desired accuracy $\varepsilon > 0$, set $i := 0$.

(1) Solve the linear least squares problem $\|Df(x^{(i)})\Delta x^{(i)} - f(x^{(i)})\|_2^2 = \min$ and compute $x^{(i+1)} = x^{(i)} - \Delta x^{(i)}$

(2) If $\|\Delta x^{(i)}\| < \varepsilon$, terminate the algorithm,
otherwise set $i := i + 1$ and go to (1)

□

Remark 5.25 Similar to how we need to solve a sequence of linear equation systems to solve the nonlinear equation system in the Newton's method, here we solve a sequence of linear least squares problems to solve the nonlinear least squares problem. □

The following theorem, which is a generalization of the convergence theorem 5.15 of the Newton's method, shows that despite the simplifications made, the algorithm converges. Quadratic convergence, however, is achieved only in compatible problems.

Theorem 5.26 Let $D \subset \mathbb{R}^n$ be an open and convex set, and let $f : D \to \mathbb{R}^m$ with $m > n$ be a continuously differentiable function whose Jacobian matrix $Df(x)$ has full rank for all $x \in D$. Let $x^* \in D$ be a solution of the least squares problem $\|f(x)\|_2^2 = \min$ in D that satisfies (5.6). For $\omega > 0$, assume the following *affine-invariant Lipschitz condition*

$$\|Df(x)^+(Df(x + sv) - Df(x))v\| \leq s\omega\|v\|^2$$

holds for all $s \in [0, 1]$, all $x \in D$, and all $v \in \mathbb{R}^n$ such that $x + v \in D$. Furthermore, for $0 \leq \kappa^* < 1$, assume the inequality

$$\|Df(x)^+ f(x^*)\| \leq \kappa^*\|x - x^*\| \tag{5.10}$$

holds for all $x \in D$.

Then, for all initial values $x^{(0)} \in \mathbb{R}^n$ with

$$\rho := \|x^* - x^{(0)}\| < \frac{2(1 - \kappa^*)}{\omega} \quad \text{and} \quad B_\rho(x^*) \subseteq D$$

the iteration sequence $x^{(i)}$ defined by the Gauss-Newton method for $i > 0$ remains in the ball $B_\rho(x^*)$ and converges to x^*, i.e.,

$$\|x^{(i)} - x^*\| < \rho \text{ for all } i > 0 \quad \text{and} \quad \lim_{i\to\infty} x^{(i)} = x^*.$$

The convergence rate can be estimated as

$$\|x^{(i+1)} - x^*\| \leq \left(\frac{\omega}{2}\|x^{(i)} - x^*\| + \kappa^*\right)\|x^{(i)} - x^*\| = \frac{\omega}{2}\|x^{(i)} - x^*\|^2 + \kappa^*\|x^{(i)} - x^*\|,$$

i.e., the method converges locally linearly. If the problem is compatible, then $\kappa^* = 0$ and the method converges locally quadratically. Furthermore, under the given assumptions x^* is the unique solution in $B_{2(1-\kappa^*)/\omega}(x^*)$. □

Proof: The proof proceeds similarly to the proof for the Newton's method (Theorem 5.15). As in that proof, from the affine-invariant Lipschitz condition we obtain the inequality

$$\|Df(x)^+(f(y) - f(x) - Df(x)(y - x))\| \leq \frac{\omega}{2}\|y - x\|^2$$

for all $x, y \in D$.

Since the Jacobian matrix Df has full rank on D by assumption, the pseudoinverse $Df(x)^+ = (Df(x)^T Df(x))^{-1} Df(x)^T$ is well-defined, and we have

$$Df(x)^+ Df(x) = (Df(x)^T Df(x))^{-1} Df(x)^T Df(x) = \mathrm{Id}_{\mathbb{R}^n}$$

for all $x \in D$. Thus, we obtain

$$\begin{aligned} x^{(i+1)} - x^* &= x^{(i)} - x^* - \Delta x^{(i)} = x^{(i)} - x^* - Df(x^{(i)})^+ f(x^{(i)}) \\ &= Df(x^{(i)})^+ \left(f(x^*) - f(x^{(i)}) - Df(x^{(i)})(x^* - x^{(i)}) \right) - Df(x^{(i)})^+ f(x^*). \end{aligned}$$

From the assumptions of the theorem, we have

$$\|x^{(i+1)} - x^*\| \leq \left(\frac{\omega}{2} \|x^{(i)} - x^*\| + \kappa^* \right) \|x^{(i)} - x^*\|.$$

For $\|x^{(i)} - x^*\| \leq \rho$, this implies

$$\|x^{(i+1)} - x^*\| \leq \left(\frac{\omega}{2} \rho + \kappa^* \right) \|x^{(i)} - x^*\|$$

with

$$k := \frac{\omega}{2} \rho + \kappa^* < \frac{\omega}{2} \frac{2(1 - \kappa^*)}{\omega} + \kappa^* = 1.$$

Using induction, we deduce

$$\|x^{(i)} - x^*\| \leq k^i \|x^{(0)} - x^*\| < \rho,$$

so the sequence remains in $B_\rho(x^*)$ for all $i \geq 1$, and it converges to x^* because $k^i \to 0$ as $i \to \infty$.

For compatible problems, $f(x^*) = 0$, and thus $\kappa^* = 0$, leading to quadratic convergence. The uniqueness follows analogously to the proof of Theorem 5.15. □

Remark 5.27 The parameter κ^* can be seen as a measure of the "non-compatibility" of the nonlinear least squares problem. Because for x^* from (5.6), we have the equation $0 = Dg(x^*) = 2Df(x^*)^T f(x^*)$, and therefore

$$Df(x^*)^+ f(x^*) = (Df(x^*)^T Df(x^*))^{-1} Df(x^*)^T f(x^*) = 0$$

which means that inequality (5.10) is satisfied for $x = x^*$ for all $\kappa^* \geq 0$, making it a real condition only for $x \neq x^*$. The parameter κ^* depends on $f(x^*)$ and the Lipschitz constant of $Df(x)^+$.

From the perspective of the proof, it is easy to see that $\kappa^* < 1$ is necessary for convergence since this κ^* accounts for the linear convergence factor. An in-depth analysis of the method using statistical methods shows that for $\kappa^* \geq 1$ the solution $\tilde{x} = x + \Delta x$ can in addition be drastically affected by arbitrarily small disturbances $\|\Delta z\|$ in the measurements $\tilde{z} = z + \Delta z$, indicating that the problem is extremely ill-conditioned for $\kappa^* \geq 1$. Even if the method converges in such a case, the obtained solution is likely practically unusable. □

Bibliography

[1] Deuflhard, P. ; Hohmann, A.: *Numerische Mathematik. I: Eine algorithmisch orientierte Einführung.* 3. Auflage. Berlin : de Gruyter, 2002

[2] Hall, C. A. ; Meyer, W. W.: Optimal Error Bounds for Cubic Spline Interpolation. In: *J. Approx. Theory* 16 (1976), S. 105–122

[3] Kloeden, P. E.: *Einführung in die Numerische Mathematik.* Vorlesungsskript, J.W. Goethe–Universität Frankfurt am Main, 2002

[4] Lempio, F.: *Numerische Mathematik I: Methoden der Linearen Algebra.* Bayreuther Mathematische Schriften, Band 51, 1997

[5] Lempio, F.: *Numerische Mathematik II: Methoden der Analysis.* Bayreuther Mathematische Schriften, Band 56, 1998

[6] Liesen, J. ; Mehrmann, V.: *Lineare Algebra.* Wiesbaden : Vieweg+Teubner Verlag, 2012

[7] Oevel, W.: *Einführung in die Numerische Mathematik.* Spektrum Verlag, Heidelberg, 1996

[8] Schwarz, H. R. ; Köckler, N.: *Numerische Mathematik.* 5. Auflage. Stuttgart : B. G. Teubner, 2004

[9] Stoer, J.: *Numerische Mathematik I.* 9. Auflage. Springer Verlag, Heidelberg, 2005

Index