Numerical Methods
(NWI-WB105)

Laura Scarabosio
Radboud Universiteit Nijmegen

August 31, 2025

Preface

Most real-world mathematical problems cannot be solved exactly, but their solution can usually be approximated arbitrarily closely (in numerical form) by using suitable algorithms implemented on electronic computers. These notes are an introduction to the branch of mathematics involved with the construction and analysis of such algorithms: *numerical mathematics*.

Suppose that we are lucky, and manage to solve a mathematical problem exactly. This generally means that we found an exact mathematical expression for the solution, for example a definite integral or something as simple as $\sqrt{2\pi}\arcsin(1/5)$. For such a mathematical expression to become usable, it needs to be evaluated in numerical form (with enough decimal places), which requires a suitable algorithm as well. For the latter expression ($\sqrt{2\pi}\arcsin(1/5)$), such an algorithm is of course readily available in most modern pocket calculators. Also for the former expression (the definite integral), its numerical approximation is standard functionality in many modern programming languages. We conclude that even for problems that can be solved exactly, numerical mathematics still has an important role to play.

Numerical mathematics goes back a long time. This is illustrated by the fact that some well-known numerical methods are named after famous mathematicians such as Newton, Euler, Lagrange and Gauss. Today, numerical mathematics is still a very active branch of mathematics, ranging from very theoretical and mathematical to very practical and software focused. Numerical simulations are more and more used by scientists to study problems from physics, chemistry, biology, medicine and finance, and by engineers in many applications such as the design of planes and chips. For speeding up large-scale computations (e.g. weather forecasting or modeling 3D turbulent fluid flow) it is common practice to make use of *parallel computing*, a specialized topic with strong links to computer science. The combination of numerical mathematics and mathematical modelling with sophisticated computing platforms is often called *computational science* or *scientific computing*.

In these lecture notes a number of numerical methods is discussed and analyzed. Increasingly such methods are readily available in numerical software packages. In this course we will use the programming language Python, which offers a number of powerful libraries for numerical and scientific calculations (NumPy and SciPy) as well as plotting (Matplotlib). For many 'standard problems' this software will provide a fast and reliable solution if the problem size is not too big. Yet it is important that the limitations of the various methods are understood. A good understanding of these numerical methods can also be very useful when new problems are encountered since their numerical solution can often be based on modifications or extensions of existing methods.

Acknowledgements

These lecture notes are, up to minor modifications, the ones written by dr. J.F.B.M. Kraaijevanger for former editions of this course during the period 2018–2022. These were in turn an improved and extended version of those written by prof. dr. W.H. Hundsdorfer, who taught this course during the period 2009–2017.

Material marked with an asterisk

In these lecture notes, all material marked with an asterisk (*) is optional, so may in principle be skipped when preparing for the exam.

Contents

Chapter 1

Principles of Numerical Mathematics

Numerical mathematics is the branch of mathematics involved with the construction and analysis of algorithms for the approximate solution of mathematical problems in numerical form.
In this chapter we discuss a number of basic concepts in numerical mathematics. We start with explaining the finite precision of computers in Sections 1.1–1.3. Next, in Section 1.4, we focus on a general question that is relevant for any mathematical problem, namely:

> *How sensitive is the (exact) solution of the problem with respect to small variations in the input data?*

This sensitivity is an intrinsic property of the problem and will be referred to as the *condition* of the problem.
The solution of the problem can in principle be approximated with many algorithms, each consisting of a well-defined finite sequence of elementary operations to be implemented on a computer. Since each elementary operation produces a rounding error, there is also a general question that is relevant for any such algorithm, namely:

> *How sensitive is the computed (approximate) solution with respect to all the rounding errors?*

This sensitivity, which is studied in Section 1.5, is an intrinsic property of the algorithm and determines the *stability* of the algorithm.

1.1 Floating-Point Numbers in Computers

In digital computers, numbers are internally represented with a fixed amount of computer memory (usually 32 or 64 bits). In this section we will discuss in some detail the so-called *floating-point numbers*, which form the most important numeric data type (apart from the integers) in a computer. These numbers, which we will also loosely refer to as *machine numbers*, are of the form

$$x = \pm 0.d_1 d_2 \dots d_n \cdot B^m, \tag{1.1}$$

where

- B is the *base* of the representation,
- the digits d_j satisfy $d_j \in \{0, 1, \ldots, B-1\}$,
- n is the number of digits d_j used in the representation,
- $0.d_1 d_2 \ldots d_n$ is the *mantissa*, which has the value $\sum_{j=1}^{n} d_j B^{-j}$,
- m is the *exponent*,
- for $x \neq 0$ the exponent m is generally chosen such that $d_1 \neq 0$ (normalization),
- $\pm$ is the *sign*.

Modern computers internally use a binary representation ($B = 2$) and in that case the digits $d_j \in \{0, 1\}$ are called *bits*, which is short for *binary digits*. The exact implementation of the corresponding binary floating-point numbers is described in the so-called IEEE 754 standard, where IEEE stands for Institute of Electrical and Electronics Engineers. According to this standard, a 'double precision' number (in PYTHON: 'float') is represented with 64 bits:

- 1 bit for the sign,
- 52 bits for the mantissa, and
- 11 bits for the exponent.

The 52 bits reserved for the mantissa actually lead to an effective mantissa length (also called *precision*, or *number of significant digits*) of $n = 53$, since, after normalization, the first digit d_1 of a nonzero number is always 1 so does not need to be stored. This corresponds to a precision of about $n = 16$ in the decimal representation ($B = 10$).
The 11 bits for the exponent give rise to $2^{11} = 2048$ possibilities of which 2046 are reserved for covering the exponent range $-1021 \leq m \leq 1024$ of normalized numbers. This corresponds to an exponent range of about $-307 \leq m \leq 308$ in the decimal representation ($B = 10$). As described in the IEEE 754 standard, the remaining two cases are reserved for a proper handling of special numbers like $+\infty, -\infty, +0, -0$, NaN and—closely related to that—a proper handling of overflow, underflow and subnormal numbers (i.e., numbers where the normalization condition $d_1 \neq 0$ is dropped allowing them to get closer to zero). Here NaN stands for *Not a Number*, and it is the result of $0/0$ and other special operations.

1.2 Rounding Errors

In the previous section we saw that floating-point (machine) numbers have a finite precision. As a result, numerical calculations on a computer are generally not exact but suffer from rounding errors. In this section we focus on rounding errors. We denote the set of machine numbers by $\mathbb{F}$, and briefly describe how a given real number $x \in \mathbb{R}$ is rounded to a machine number $\widetilde{x} = \mathrm{fl}(x) \in \mathbb{F}$.

There exist a number of rounding modes, but the default mode in IEEE 754 is to round to the *nearest* number in $\mathbb{F}$. In case of a 'tie' (which means that there are two nearest numbers in $\mathbb{F}$), the rule is to round to the one that has an *even* last digit d_n in its representation (1.1). This rule only makes sense if B is even[1]. For $B = 2$ this implies that, in case of a tie, the last digit d_n of the rounded number is always zero.
The rounding of the exact value $x \in \mathbb{R}$ to the approximate value $\widetilde{x} = \mathrm{fl}(x)$ introduces, in general, an error. The *absolute* and *relative* error are defined as

$$\text{absolute error} = \widetilde{x} - x, \quad \text{relative error} = \frac{\widetilde{x} - x}{x},$$

where the latter is defined whenever $x \neq 0$. It is clear from the description of the rounding process that the rounded value $\widetilde{x}$ cannot differ more from the exact value x than half the size of a unit change of the n-th (last) significant digit of x,

$$|\widetilde{x} - x| \leq \frac{1}{2} B^{-n} B^m = \frac{1}{2} B^{m-n}.$$

For normalized $x \neq 0$ we have $|x| \geq d_1 B^{-1} B^m \geq B^{m-1}$, and it follows that

$$\left| \frac{\widetilde{x} - x}{x} \right| \leq \frac{1}{2} B^{1-n}. \tag{1.2}$$

The right-hand side of this inequality is a machine-dependent constant often denoted by '`eps`' and referred to as the *machine precision* or *machine epsilon*. It provides a uniform upper bound for the absolute value of the relative rounding error. For $B = 2$ and $n = 53$ the machine precision is equal to

$$\texttt{eps} = \frac{1}{2} B^{1-n} = 2^{-53} \approx 1.11 \cdot 10^{-16}. \tag{1.3}$$

Inequality (1.2) is often reformulated into the following convenient form,

$$\widetilde{x} = (1 + \varepsilon)x \quad \text{with} \quad |\varepsilon| \leq \texttt{eps}. \tag{1.4}$$

This form has the additional advantage that it is also valid for $x = 0$. In that case the rounded value is clearly $\widetilde{x} = 0$, so that (1.4) holds with $\varepsilon = 0$. In all cases, we will refer to the value ε in (1.4) as the 'relative rounding error', even though that name is strictly speaking only correct for $x \neq 0$.
We further note that (1.4) does *not* hold in case of *underflow* or *overflow*, which occurs when the exponent m is outside the allowed range (too small or too large, respectively). In practice, in the case of overflow, the program is interrupted, while, for underflow, rounding is still defined, and in some cases it is handled via allowing de-normalization (so $d_1 = 0$). In the remainder of this chapter we assume that we are dealing with the idealized situation of no underflow/overflow, so that (1.4) always holds.
We finally note that the definition of the machine epsilon is not part of the IEEE 754 standard. Although the definition given above is generally accepted in the global numerical mathematics

[1] For odd B there exist ties where both nearest numbers in $\mathbb{F}$ have an even last digit d_n.

community, there exists also a widely adopted alternative definition, where the machine epsilon is defined as the distance between 1 and the smallest $x \in \mathbb{F}$ with $x > 1$. One easily verifies that with this alternative definition, the value of the machine epsilon becomes B^{1-n}, which is twice the value defined above. This alternative definition is also adopted in PYTHON, where the value of the machine epsilon can be inspected by executing the command

```
print(numpy.finfo(float).eps)
```

(upon importing `numpy` first).

1.3 Floating-Point Arithmetic

The arithmetic operations $+, -, \times, /$ in a computer are not exact, but the IEEE 754 standard requires that their implementations $\oplus, \ominus, \otimes, \oslash$ produce the rounded value of the exact answer for all inputs $x, y \in \mathbb{F}$ (unless underflow/overflow occurs). For the floating-point version $\oplus$ of the addition $+$, for example, this means that for (almost) all $x, y \in \mathbb{F}$ we have

$$x \oplus y = \mathrm{fl}(x + y). \tag{1.5}$$

In view of (1.4), this implies

$$x \oplus y = (1 + \varepsilon)(x + y) \quad \text{with} \quad |\varepsilon| \leq \texttt{eps}. \tag{1.6}$$

Starting from $x, y \in \mathbb{R}$ (rather than $\mathbb{F}$), their sum is approximated as

$$x \oplus y = \mathrm{fl}(\mathrm{fl}(x) + \mathrm{fl}(y)). \tag{1.7}$$

Similar results hold for the arithmetic operations $-, \times, /$ and a number of other basic functions such as taking the square root.

1.4 The Condition of a Problem

A problem typically has input data and a solution that depends on the input data. Many problems can therefore be seen as a function $f : X \to Y$ from a space of input data X to a solution space Y. Solving the problem for a given $x \in X$ then simply means determining the function value $y = f(x)$.
In the following we are interested in the sensitivity of the solution $y = f(x)$ as a function of $x \in X$. It is clear that perturbations Δx in x will cause perturbations Δy in y. We will investigate how the size of Δy varies with the size of Δx, which is important to know how much we can trust computations with approximate inputs.

1.4.1 The scalar case

First we focus on the simple case where $X \subset \mathbb{R}$, $Y = \mathbb{R}$. We assume that $x \in X$ is given and consider the effect of a small perturbation Δx on the solution y. For the corresponding perturbation Δy, assuming f to be differentiable we have

$$\Delta y = f(x + \Delta x) - f(x) \approx f'(x)\Delta x. \tag{1.8}$$

Generally it is more interesting to look at relative perturbations. Assuming $x \neq 0$ and $y \neq 0$ we obtain

$$\left|\frac{\Delta y}{y}\right| \approx \gamma \left|\frac{\Delta x}{x}\right| \quad \text{with} \quad \gamma = \left|\frac{x f'(x)}{f(x)}\right|. \tag{1.9}$$

The number γ is the amplification factor of the relative error in x when it is propagated to the relative error in y. It is called the *condition number* of f at x.
If γ is large, small (relative) errors in the input x cause large (relative) errors in the solution $y = f(x)$ of the problem. In that case we call the problem *ill-conditioned* at x. If $|\gamma|$ is moderately sized, small (relative) errors in x cause small (relative) errors in y. In that case we call the problem *well-conditioned* at x.

Example 1.1. Consider the problem to compute $y = f(x) = \sqrt{x}$ for a given real number $x > 0$. One easily verifies that relation (1.9) holds with $\gamma = 1/2$, implying that the problem is well-conditioned. A relative error of 1% in x will lead to a relative error of approximately 0.5% in y. ◊

Example 1.2. Consider the problem to compute $y = f(x) = \sin x$ for a given real number $x \in (0, \pi)$. One easily verifies that relation (1.9) holds with $\gamma = x \cos x / \sin x$, implying that the problem is well-conditioned unless x is close to π. ◊

1.4.2 The general case

Next we consider the multi-dimensional case where $X \subset \mathbb{R}^n$, $Y = \mathbb{R}^m$. In that case we have

$$x = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}, \quad y = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{pmatrix} = f(x) = \begin{pmatrix} f_1(x_1, x_2, \ldots, x_n) \\ f_2(x_1, x_2, \ldots, x_n) \\ \vdots \\ f_m(x_1, x_2, \ldots, x_n) \end{pmatrix}.$$

In the multi-dimensional case, there are two notions of condition number: the componentwise condition number and the normwise condition number.
The *componentwise condition number* measures the sensitivity of a single output component y_i with respect to a single input component x_j. This is essentially the scalar case discussed above, so we can apply relation (1.9) (with $x = x_j$, $y = y_i$, $f = f_i$) to conclude that the corresponding condition number (of y_i with respect to x_j) is given by

$$\gamma_{ij} = \left|\frac{x_j \frac{\partial f_i}{\partial x_j}}{f_i(x)}\right|. \tag{1.10}$$

By varying i and j we obtain mn condition numbers γ_{ij}, which jointly represent the condition of the multi-dimensional problem $y = f(x)$. We call the problem *well-conditioned* at x if all these mn condition numbers are of moderate size. This generalizes the definition of the scalar case ($m = n = 1$).

When considering the *normwise condition number* instead, we adopt a global view. We note that (1.8) is also valid for the multi-dimensional case[2] with $\Delta x \in \mathbb{R}^n, \Delta y \in \mathbb{R}^m$ and where $f'(x)$ becomes the Jacobian matrix $Jf(x)$ of the function f at x,

$$Jf(x) = (\partial f_i/\partial x_j) = \begin{pmatrix} \partial f_1/\partial x_1 & \partial f_1/\partial x_2 & \cdots & \partial f_1/\partial x_n \\ \partial f_2/\partial x_1 & \partial f_2/\partial x_2 & \cdots & \partial f_2/\partial x_n \\ \vdots & \vdots & & \vdots \\ \partial f_m/\partial x_1 & \partial f_m/\partial x_2 & \cdots & \partial f_m/\partial x_n \end{pmatrix}.$$

The multi-dimensional version of (1.8) yields

$$\Delta y_i \approx \sum_j \frac{\partial f_i}{\partial x_j} \Delta x_j, \quad \left|\frac{\Delta y_i}{y_i}\right| \lesssim \sum_j \gamma_{ij} \left|\frac{\Delta x_j}{x_j}\right|, \tag{1.11}$$

with the same γ_{ij} as those defined in (1.10). Hence the global error estimate of the second approach is equal to the sum of the component-wise error estimates predicted by the first approach. Clearly, both approaches are related via a *superposition principle*.

Example 1.3. Consider the problem to compute the product $y = x_1 x_2$ of two nonzero reals x_1, x_2. It follows from (1.11) that $\Delta y \approx x_2 \Delta x_1 + x_1 \Delta x_2$ and

$$\left|\frac{\Delta y}{y}\right| \lesssim \left|\frac{\Delta x_1}{x_1}\right| + \left|\frac{\Delta x_2}{x_2}\right|,$$

implying that the problem is well-conditioned. ◊

Example 1.4. Consider the problem to compute the sum $y = x_1 + x_2$ of two nonzero reals x_1, x_2. In line with (1.11) we find $\Delta y = \Delta x_1 + \Delta x_2$ and

$$\left|\frac{\Delta y}{y}\right| \le \gamma_1 \left|\frac{\Delta x_1}{x_1}\right| + \gamma_2 \left|\frac{\Delta x_2}{x_2}\right| \quad \text{with} \quad \gamma_1 = \left|\frac{x_1}{x_1 + x_2}\right|, \ \gamma_2 = \left|\frac{x_2}{x_1 + x_2}\right|.$$

If x_1 and x_2 have the same sign we have $|\gamma_1| + |\gamma_2| = 1$, so in that case the problem is well-conditioned. If x_1 and x_2 have opposite sign, the problem becomes ill-conditioned for $x_1/x_2 \approx -1$. ◊

Example 1.5. Consider the problem to compute the difference $y = x_1 - x_2$ of two nonzero reals x_1, x_2. The analysis of this problem is almost identical to the analysis of the sum in the previous example. We find $\Delta y = \Delta x_1 - \Delta x_2$ and

$$\left|\frac{\Delta y}{y}\right| \le \gamma_1 \left|\frac{\Delta x_1}{x_1}\right| + \gamma_2 \left|\frac{\Delta x_2}{x_2}\right| \quad \text{with} \quad \gamma_1 = \left|\frac{x_1}{x_1 - x_2}\right|, \ \gamma_2 = \left|\frac{x_2}{x_1 - x_2}\right|.$$

[2] see Appendix section B.4

From the condition numbers γ_1, γ_2 it is seen that the problem is ill-conditioned if x_1 and x_2 are almost equal, or more precisely, if $x_1/x_2 \approx 1$. The fact that computing the difference of two almost equal numbers is ill-conditioned is well-known to numerical mathematicians, and is important enough to illustrate with a numerical experiment.
Let x_1 and x_2 be defined as

$$\begin{aligned} x_1 &= \pi \quad = 3.141\,592\,653\,589\,793..., \\ x_2 &= \frac{355}{113} = 3.141\,592\,920\,353\,982.... \end{aligned}$$

We approximate both numbers to 10 significant decimal digits,

$$\begin{aligned} \widetilde{x}_1 &= 3.141\,592\,654, \\ \widetilde{x}_2 &= 3.141\,592\,920. \end{aligned}$$

This gives (using exact arithmetic)

$$\begin{aligned} y &= x_1 - x_2 = -0.000\,000\,266\,764\,189..., \\ \widetilde{y} &= \widetilde{x}_1 - \widetilde{x}_2 = -0.000\,000\,266, \end{aligned}$$

clearly showing that the approximate difference $\widetilde{y}$ agrees with the exact difference to only 3 significant decimal digits, which is 7 less than the 10 significant digits of the input values. The mechanism behind this dramatic loss of significant digits is called *cancellation*, referring to the disappearance of the common leading digits when two almost equal numbers are subtracted. The loss of significant digits can of course be rephrased as an increase of the size of the relative error, and that is exactly what happens when dealing with large condition numbers, which in this example are equal to $\gamma_1 \approx 1.18 \cdot 10^7$ and $\gamma_2 \approx 1.18 \cdot 10^7$.
We emphasize that we used exact arithmetic here, so the subtraction operation itself did not *introduce* any error. Nevertheless the subtraction operation is fully responsible for the observed loss of accuracy as it *propagated* the already existing (relative) errors in the inputs x_1 and x_2 in a dramatic way. In the design of numerical algorithms, one should therefore always be aware of the possibility of cancellation. ◊

Example 1.6. Consider the problem to compute $y = f(x_1, x_2)$ for two positive reals x_1, x_2, where f is defined as $f(x_1, x_2) = (x_1 + x_2)^2 - x_2^2 = (x_1 + 2x_2)x_1$. The two condition numbers are given by

$$\gamma_1 = \left| \frac{x_1}{y} \frac{\partial f}{\partial x_1} \right| = \frac{2(x_1 + x_2)}{x_1 + 2x_2}, \quad \gamma_2 = \left| \frac{x_2}{y} \frac{\partial f}{\partial x_2} \right| = \frac{2x_2}{x_1 + 2x_2}.$$

Since $\gamma_1 + \gamma_2 = 2$, the problem is well-conditioned. ◊

Another useful consequence of the normwise condition number is that it allows for further generalization, where we look at X and Y as normed spaces. Considering a norm $\|\cdot\|_X$ on the space of input data ($\mathbb{R}^n$) and a norm $\|\cdot\|_Y$ on on the solution space ($\mathbb{R}^m$), it follows from $\Delta y \approx f'(x)\Delta x$ (the multi-dimensional version of (1.8)) that

$$\|\Delta y\|_Y \approx \|Jf(x)\Delta x\|_Y \leq \|Jf(x)\|\|\Delta x\|_X,$$

where $\|Jf(x)\|$ denotes the so-called *induced matrix norm*[3] of the Jacobian matrix. Assuming $x \neq 0$ and $y \neq 0$, we find

$$\frac{\|\Delta y\|}{\|y\|} \lesssim c \frac{\|\Delta x\|}{\|x\|} \quad \text{with} \quad c = \frac{\|x\| \|Jf(x)\|}{\|f(x)\|}, \tag{1.12}$$

where we have omitted the subscripts from the norms to improve readability. Inequality (1.12) provides a simple estimate (upper bound) for the condition of the problem to compute y from x. The condition of f at x is estimated using a single constant c, which is also called a condition number. The single condition number c is less informative than the mn condition numbers γ_{ij} defined in (1.10). We will use approach (1.12) when studying systems of linear equations in Chapter 2.

1.5 Error Propagation in Algorithms

In the previous section we studied the condition of the problem to compute $y = f(x)$ for a given $x \in X$. This gives an indication on the stability of the output with respect to perturbations of the input. We emphasize that the function $f : X \to Y$ in this analysis is exact, which means that its evaluation (for any given input $x \in X$) is not affected by rounding errors. In reality, however, the function f must be implemented in the computer as an algorithm A, which is a well-defined sequence of elementary operations $A_1, A_2, \ldots, A_N$ such as the arithmetic operations $+, -, \times, \div$ and evaluations of basic functions such as $\sqrt{}, \sin, \cos$. Each elementary operation A_k produces an intermediate result a_k, and the final elementary operation A_N produces $a_N = y$. Here we have assumed that y (and all intermediate values a_k) are scalar. Note that there is no loss of generality since each component y_i of $y \in \mathbb{R}^m$ is scalar.
Starting with the initial data $x_1, x_2, \ldots, x_n$, the data flow of any algorithm $A = (A_1, A_2, \ldots, A_N)$ can be visualized with a directed graph, see for example Figure 1.1.
The floating-point versions $\widetilde{A}_k$ of the elementary operations A_k are not exact, so, instead of applying f on a given input, we apply a perturbed version $\tilde{f}$ of it. In this section, we study the stability of the computer output with respect to perturbations of f due to rounding errors. As explained in Section 1.3, we assume that the floating-point version of each elementary operation A_k produces the rounded value $\widetilde{a}_k$ of the exact answer a_k,

$$\widetilde{a}_k = (1 + \eta_k) a_k \quad \text{with} \quad |\eta_k| \leq \texttt{eps}. \tag{1.13}$$

In other words, each application of an elementary floating-point operation $\widetilde{A}_k$ introduces a new relative rounding error η_k, $k = 1, 2, \ldots, N$. The corresponding absolute rounding errors are given by

$$\Delta a_k = \widetilde{a}_k - a_k = a_k \eta_k \quad \text{with} \quad |\eta_k| \leq \texttt{eps}. \tag{1.14}$$

We will study the combined effect of these N rounding errors on the output y by estimating the effect caused by a single rounding error $\Delta a_k = a_k \eta_k$ on y and then applying the principle of superposition.

[3] see Appendix section A.4

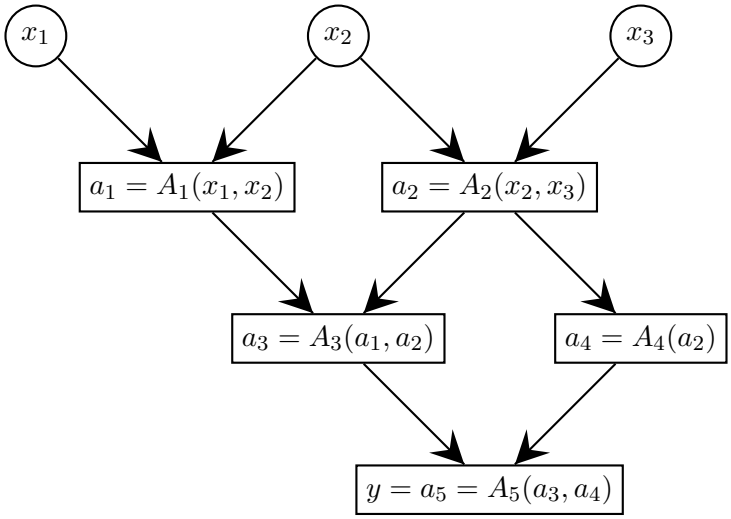


Figure 1.1: Directed graph representing an algorithm $A = (A_1, A_2, A_3, A_4, A_5)$ that implements a function $y = f(x_1, x_2, x_3)$.

So, let us consider the situation that we deal with a single rounding error $\Delta a_k = a_k \eta_k$ and have exact arithmetic otherwise. This rounding error showed up in (1.14) when the exact intermediate result a_k was rounded to the machine number $\widetilde{a}_k$. Since the value of y will depend on the value of a_k, we can write

$$y = R_k(a_k), \tag{1.15}$$

where R_k is the so-called *remainder function*, which maps the intermediate value a_k to the final value y. The corresponding effect on y is given by

$$\Delta y \approx R_k'(a_k)\Delta a_k = R_k'(a_k) a_k \eta_k. \tag{1.16}$$

Using linearization, we obtain the following expression for the combined effect of all 'relative rounding errors' η_k corresponding to the replacement of A_k by $\widetilde{A}_k$,

$$\begin{aligned} \Delta y &\approx \sum_{k=1}^{N} R_k'(a_k) a_k \eta_k, \\ \frac{\Delta y}{y} &\approx \sum_{k=1}^{N} \sigma_k \eta_k \quad \text{with} \quad \sigma_k = \frac{a_k R_k'(a_k)}{R_k(a_k)}. \end{aligned} \tag{1.17}$$

As noted earlier, the last elementary operation A_N produces $y = a_N$. This implies that the last remainder function R_N is the identity and $\sigma_N = 1$.

1.6 Computer solution of a problem

In this section, we combine the results from Section 1.4 and Section 1.5 to bound the total error occuring when solving a problem on a computer. Related to this, we define the concept of

stability of an algortihm. Finally, we define the complexity of an algorithm.

1.6.1 Total error

In Section 1.4, we studied stability of the output with respect to perturbations of the input, which depends on the problem but not on the algorithm used to solve it. In Section 1.5, we studied instead its stability with respect to rounding errors when performing algorithmic steps, so, stability with respect to perturbations of the input-to-output function f due to floating-point arithmetic. In this subsection, we consider both effects together to determine the total error occurring when solving a problem on a computer. For simplicity, we consider the scalar case for the output y.

We denote by $y = f(x)$ the exact solution to the problem, using exact arithmetic and no pertubations in the input. The quantity $\tilde{y} = \tilde{f}(\tilde{x})$ denotes instead the computer solution, subject to rounding errors for the algorithmic steps and perturbations to the input. Finally, $\tilde{y}_{\mathrm{round}x} = f(\tilde{x})$ is the solution obtained using exact arithmetic on the perturbed input. Using the triangle inequality and the results of the previous two sections, the absolute error in $\Delta y := y - \tilde{y}$ can be bounded as

$$|\Delta y| \lesssim |y - \tilde{y}_{\mathrm{round}x}| + |\tilde{y}_{\mathrm{round}x} - \tilde{y}| \tag{1.18}$$

$$|\Delta y| \lesssim \sum_{j=1}^{n} \left| \frac{\partial f}{\partial x_j} \Delta x_j \right| + \sum_{k=1}^{N} |R'(a_k) a_k \eta_k|, \tag{1.19}$$

where for the second summand we used the notation from the previous section. Similarly, for the relative error we have

$$\begin{aligned} \frac{|\Delta y|}{|y|} &\lesssim \sum_{j=1}^{n} \gamma_j \frac{|\Delta x_j|}{|x_j|} + |\boldsymbol{\sigma} \cdot \boldsymbol{\eta}| \\ &\lesssim \sum_{j=1}^{n} \gamma_j \frac{|\Delta x_j|}{|x_j|} + \|\boldsymbol{\sigma}\|_1 \|\boldsymbol{\eta}\|_\infty. \end{aligned} \tag{1.20}$$

In the inequalities above, we denoted $\boldsymbol{\sigma} := (\sigma_1, \ldots, \sigma_N)^\top$, where the entried are as in (1.17), and $\boldsymbol{\eta} := (\eta_1, \ldots, \eta_N)^\top$.

1.6.2 Stability of an algorithm

Broadly speaking, the stability of an algorithm is defined as *insensitivity with respect to rounding errors.* In the following, we try to make this somewhat informal definition of stability more precise with the help of error estimate (1.20).

First of all we note that all algorithms implementing a given function f react in an identical fashion to errors in the input data, as the condition number is determined by the function f only, and it does not depend on the decomposition of f into its elementary operations A_k. When comparing algorithms implementing the same function f, we can therefore safely ignore errors in the input data, and assume that the input x is exactly known. One cannot expect, however,

that the input x is a machine number, so it still has to be rounded to $\tilde{x} = \mathrm{fl}(x)$. Hence, in this case, the relative error $|\Delta x|/|x|$ is merely rounding error, which means (see Section 1.2) that it can be bounded by

$$\left|\frac{\Delta x}{x}\right| \leq \texttt{eps}.$$

Since the effect of this rounding error on $\Delta y/y$ is still the same for all algorithms, we could still ignore them when comparing two algorithms. Note, however, that we have defined stability as 'insensitivity with respect to rounding errors', which is an absolute concept, so we need to take the rounding errors of input data into account as well. It follows from (1.20) that the total effect of all rounding errors on y can be bounded by

$$\left|\frac{\Delta y}{y}\right| \lesssim \left(\underbrace{\sum_{j=1}^{n} \gamma_j}_{\gamma} + \underbrace{\sum_{k=1}^{N} |\sigma_k|}_{\sigma}\right) \texttt{eps} = (\gamma + \sigma)\texttt{eps}.$$

In the inequality above, note that both summands on the right-hand side depend on the input x. The number

$$S = \sup_{x \in X} (\gamma + \sigma)$$

is called the *stability number* of the algorithm. It is a measure for the sensitivity of the algorithm with respect to all the rounding errors involved. The first term quantifies the condition of the problem and represents the effect on y of the rounding errors in the input x. As discussed above, this term is the same for all algorithms that implement f. The second term depends on the details of the chosen algorithm and represents the effect on y of the intermediate rounding errors η_k, $k = 1, 2, \ldots, N$.
Note that, in (1.20), the propagation factor σ_N of the final rounding error η_N (corresponding to placing the final result $y = a_N$ into the computer) is always equal to $\sigma_N = 1$, so we can write

$$\sigma = \sigma' + 1 \quad \text{with } \sigma' = \sum_{k=1}^{N-1} |\sigma_k|.$$

The stability number S can therefore also be written as

$$S = \sup_{x \in X} (\gamma + \sigma' + 1). \tag{1.21}$$

We see that the stability number S has an 'unavoidable' part $\gamma + 1$ that only depends on (the condition of) f and not on the specific algorithm that implements f. It represents the effect of rounding errors in the input data and the effect of placing the final result y into the computer. The remaining part σ' depends on the details of the chosen algorithm. More precisely, it represents the effect on y of the intermediate rounding errors η_k, $k = 1, 2, \ldots, N-1$.
We call an algorithm *(numerically) stable* if σ' is of the same (or smaller) order of magnitude as the 'unavoidable' part $\gamma + 1$, which can be rephrased as

$$\sup_{x \in X} \frac{\sigma'}{\gamma + 1} \text{ is of moderate size,} \tag{1.22}$$

where the exact meaning of 'moderate size' depends on the specific situation, but may be thought of as '$\lesssim 1$' or '$\lesssim 10$'. Note that, in this definition of stability, no assumptions are made about the size of γ. It is therefore possible to have a large size of γ, which means that we are dealing with an ill-conditioned problem! In this case, a stable algorithm can still be valuable since it produces an error that is not much larger than the 'unavoidable' part $(\gamma+1)\texttt{eps}$. Of course, it is always better to deal with a well-conditioned problem, which means that γ is moderately-sized. In that case, a stable algorithm also has a moderately-sized stability number.

1.6.3 Computation of the Stability Number

Let an algorithm $A = (A_1, A_2, \ldots, A_N)$ be given that implements a real-valued function f. For the computation of its stability number (1.21) we need the condition number, entailed in γ, and the error propagation factors σ_k defined in (1.17). When f is differentiable, the computation of γ can be done using the partial derivatives $\partial f/\partial x_j$, entering the Jacobian matrix. The computation of σ_k, which is the condition number of y with respect to the intermediate result a_k, requires the derivative of the remainder function R_k that maps a_k to y. We illustrate this with a simple example.

Example 1.7. Consider the following algorithm $A = (A_1, A_2)$ for computing the sum $y = f(x_1, x_2, x_3) = x_1 + x_2 + x_3$ of three real numbers x_1, x_2, x_3:

$$\begin{aligned} A_1: &\quad a_1 = x_1 + x_2 \\ A_2: &\quad y = a_1 + x_3 \end{aligned}$$

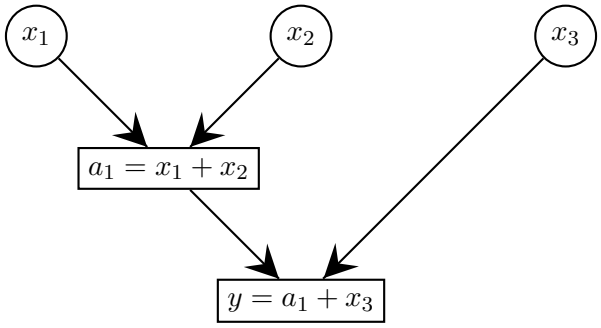


Figure 1.2: Directed graph representing the algorithm $y = (x_1 + x_2) + x_3$.

Of course, the floating point version $\widetilde{A} = (\widetilde{A}_1, \widetilde{A}_2)$ of the algorithm starts with the rounded values $\widetilde{x}_1 = \texttt{fl}(x_1)$, $\widetilde{x}_2 = \texttt{fl}(x_2)$, $\widetilde{x}_3 = \texttt{fl}(x_3)$ and proceeds with

$$\begin{aligned} \widetilde{A}_1: &\quad \widetilde{a}_1 = \widetilde{x}_1 \oplus \widetilde{x}_2 \\ \widetilde{A}_2: &\quad \widetilde{y} = \widetilde{a}_1 \oplus \widetilde{x}_3 \end{aligned}$$

One easily verifies that the three condition numbers γ_j are given by

$$\gamma_j = \frac{x_j}{y}\frac{\partial f}{\partial x_j} = \frac{x_j}{x_1 + x_2 + x_3}, \quad j = 1, 2, 3,$$

showing that the number $\gamma = |\gamma_1| + |\gamma_2| + |\gamma_3|$ is equal to

$$\gamma = \frac{|x_1| + |x_2| + |x_3|}{|x_1 + x_2 + x_3|}. \tag{1.23}$$

For the intermediate result $a_1 = x_1 + x_2$, the remainder function R_1 is defined as

$$R_1(a_1) = a_1 + x_3,$$

(with fixed x_3), implying that

$$\sigma_1 = \frac{a_1}{y}R_1'(a_1) = \frac{a_1}{y} = \frac{x_1 + x_2}{x_1 + x_2 + x_3},$$

so that

$$\sigma' = |\sigma_1| = \frac{|x_1 + x_2|}{|x_1 + x_2 + x_3|}. \tag{1.24}$$

Since

$$\sigma' = \frac{|x_1 + x_2|}{|x_1 + x_2 + x_3|} \le \frac{|x_1| + |x_2|}{|x_1 + x_2 + x_3|} \le \frac{|x_1| + |x_2| + |x_3|}{|x_1 + x_2 + x_3|} = \gamma,$$

we see that (1.22) holds, implying that algorithm A is stable.
If all inputs x_1, x_2, x_3 have the same sign, the problem is well-conditioned because in that case (1.23) shows that $\gamma = 1$. The problem may become ill-conditioned if not all signs are equal. If, for example, the sign of x_1 differs from those of x_2 and x_3, then

$$\gamma = \frac{|x_1| + |x_2 + x_3|}{|x_1 + x_2 + x_3|} = \frac{|\frac{x_1}{x_2+x_3}| + 1}{|\frac{x_1}{x_2+x_3} + 1|},$$

showing that γ becomes large if $x_1/(x_2 + x_3) \approx -1$. ◊

The above example is relatively simple with $N = 2$. For larger N the computation of all γ_j and σ_k can become quite tedious, though still straightforward. In the following we present two slightly more complex examples. Each example deals with a different algorithm for the function f defined in Example 1.6. The first algorithm turns out to be stable, but the second one may well become unstable.

Example 1.8. In Example 1.6 we showed that the problem to compute the value $y = f(x_1, x_2) = (x_1+x_2)^2-x_2^2 = x_1(x_1+2x_2)$ from two positive reals x_1, x_2 is well-conditioned since $\gamma = \gamma_1+\gamma_2 = 2$.
Here we consider the following algorithm implementing f:

$$\begin{aligned} a_1 &= x_1 + x_2 \\ a_2 &= a_1 + x_2 \\ y &= x_1 a_2 \end{aligned}$$

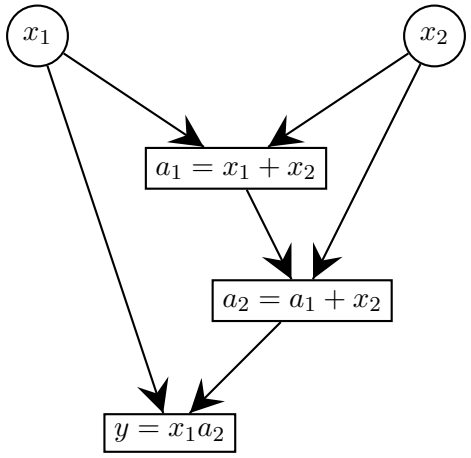


Figure 1.3: Directed graph representing the algorithm $y = x_1((x_1 + x_2) + x_2)$.

The directed graph associated with this algorithm is shown in Figure 1.3.
For the intermediate results $a_1 = x_1 + x_2$, $a_2 = a_1 + x_2$ the remainder functions are given by

$$R_1(a_1) = x_1(a_1 + x_2), \quad R_2(a_2) = x_1 a_2,$$

implying that

$$\sigma_1 = \frac{a_1}{y} R_1'(a_1) = \frac{a_1}{y} x_1 = \frac{a_1}{a_2} = \frac{x_1 + x_2}{x_1 + 2x_2},$$
$$\sigma_2 = \frac{a_2}{y} R_2'(a_2) = \frac{a_2}{y} x_1 = \frac{a_2}{a_2} = 1.$$

Hence we find

$$\gamma = \gamma_1 + \gamma_2 = 2,$$
$$\sigma' = |\sigma_1| + |\sigma_2| = \frac{x_1 + x_2}{x_1 + 2x_2} + 1 \leq 2,$$

showing that the algorithm is stable and has stability number $S = \sup_{x \in \mathbb{R}^2} (\gamma + \sigma' + 1) \leq 5$. $\Diamond$

Example 1.9. We consider the following alternative algorithm for implementing the same function f as in Example 1.8:

$$a_1 = x_1 + x_2$$
$$a_2 = a_1^2$$
$$a_3 = x_2^2$$
$$y = a_2 - a_3$$

As before, we assume that the inputs x_1 and x_2 are positive reals. The directed graph associated with this algorithm is displayed in Figure 1.4.

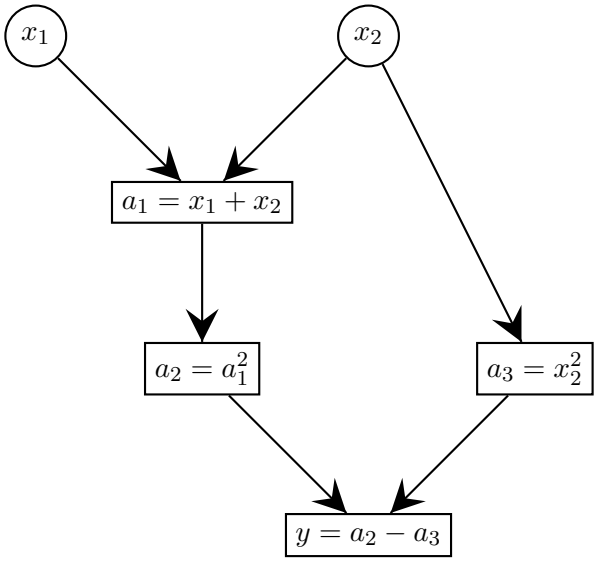


Figure 1.4: Directed graph representing the algorithm $y = (x_1 + x_2)^2 - x_2^2$.

For the intermediate results $a_1 = x_1 + x_2$, $a_2 = a_1^2$, $a_3 = x_2^2$ the remainder functions and their derivatives are given by

$$\begin{aligned} R_1(a_1) &= a_1^2 - a_3, & R_1'(a_1) &= 2a_1 \\ R_2(a_2) &= a_2 - a_3, & R_2'(a_2) &= 1 \\ R_3(a_3) &= a_2 - a_3, & R_3'(a_3) &= -1 \end{aligned}$$

This gives

$$\begin{aligned} \sigma_1 &= \frac{a_1}{y} R_1'(a_1) = \frac{a_1}{y} 2a_1 = \frac{2a_2}{a_2 - a_3}, \\ \sigma_2 &= \frac{a_2}{y} R_2'(a_2) = \frac{a_2}{y} = \frac{a_2}{a_2 - a_3}, \\ \sigma_3 &= \frac{a_3}{y} R_3'(a_3) = -\frac{a_3}{y} = \frac{-a_3}{a_2 - a_3}. \end{aligned}$$

Note that σ_1 and σ_3 can be expressed in terms of σ_2,

$$\sigma_1 = 2\sigma_2, \quad \sigma_3 = 1 - \sigma_2,$$

and that

$$\sigma_2 = \frac{a_2}{a_2 - a_3} = \frac{a_2}{y} = \frac{(x_1 + x_2)^2}{x_1(x_1 + 2x_2)} = \frac{(r+1)^2}{r(r+2)} = 1 + \frac{1}{r(r+2)},$$

where $r = x_1/x_2 > 0$. We see that $\sigma_2 > 1$ so that

$$\sigma = |\sigma_1| + |\sigma_2| + |\sigma_3| + |\sigma_4| = (2\sigma_2) + (\sigma_2) + (\sigma_2 - 1) + 1 = 4\sigma_2.$$

Since $\gamma = 2$ (see Example 1.6), we conclude that the algorithm is unstable if and only if σ_2 is large, which is the case if $r = x_1/x_2$ tends to zero.
Suppose we have input values $x_1 = 0.1234$, $x_2 = 123.4$, and we apply the above algorithm with floating point arithmetic with $B = 10$ and $n = 4$ significant digits. Then we find $\widetilde{y} = 20.00$, whereas the exact answer is equal to $y = 30.4703...$, implying a relative error of -34%. This bad result is a result of the instability of the algorithm since for the given input values we have $r = x_1/x_2 = 10^{-3}$ so that $\sigma \approx 2000$. Note that the two input values are machine numbers, so the only errors involved are the rounding errors η_k of the intermediate values a_k, $k = 1, 2, 3, 4$. Hence (1.20) implies

$$\left|\frac{\Delta y}{y}\right| \lesssim \sigma \cdot \texttt{eps} \approx 2000\,\texttt{eps}.$$

Since the machine precision in this case is equal to

$$\texttt{eps} = \frac{1}{2}B^{1-n} = \frac{1}{2} \cdot 10^{1-4} = 5 \cdot 10^{-4},$$

we obtain the upper bound $|\Delta y/y| \lesssim 2000 \cdot 5 \cdot 10^{-4} = 1$, showing that the theoretically predicted maximum |relative error| is 100%, which is consistent with (and of the same order of magnitude as) the observed relative error (-34%).
In contrast, if we apply the stable algorithm of Example 1.8 to the same input values and use the same floating point arithmetic ($B = 10, n = 4$), then we find $\widetilde{y} = 30.47$, which is the rounded value of the exact result.
The cause of the instability of the above algorithm is the elementary operation A_4, the final step of the algorithm. In that final step, two approximately equal numbers (a_2 and a_3) are subtracted from each other, causing a dramatic magnification of the relative errors in a_2 and a_3 due to cancellation (see Example 1.5). ◊

1.6.4 The complexity of an algorithm

So far we define concepts to study the accuracy and reliability of a computer implementation of a numerical method. Besides this, it is also important in applications to have *efficient* algorithms. Efficiency means that the computational complexity that is needed to control the error is as small as possible. The *complexity of an algorithm* is a measure of its running time. A way of estimating it is to count the number of floating point operations (*flops*) required by the algorithm. Although this does not tell the whole true, as memory management plays also an important role in the speed of an implementation, it usually gives a good indication of the computational cost. Since several algorithms, with different complexities, can be employed to solve the same problem, it is useful to introduce also the concept of *complexity of a problem*. This denotes the complexity of the algorithm with minimum complexity among those solving the given problem.
To denote the complexity, it is common to use the big-$\mathcal{O}$ notation. Namely, if we say that an algorithm has complexity $\mathcal{O}(n)$ for some $n \in \mathbb{N}$, we mean that the running time is approximately linear in n, for n large.

Example 1.10. Let $A \in \mathbb{R}^{m\times m}$ and $v \in \mathbb{R}^m$, for $m \in \mathbb{N}$. The matrix-vector multiplication Av requires m multiplications and $m-1$ sums per row. Since we have m rows, computing Av requires about $2m^2$ flops and therefore it has complexity $\mathcal{O}(m^2)$.

1.7 Further reading *

Did you know that roundoff errors might affect the training of neural networks? See the paper posted under Outlook in Brightspace in case you are interested!

Chapter 2

Systems of Linear Equations

The solution of linear systems plays a crucial role in many applications. A notable example is the numerical solution of partial differential equations, which needs the solution of very large linear systems.
In this chapter, we will discuss some basic numerical methods to solve systems of linear equations

$$Av = b\,, \tag{2.1}$$

with given matrix $A = (a_{ij}) \in \mathbb{R}^{m\times m}$ and vector $b = (b_i) \in \mathbb{R}^m$. The solution vector $v \in \mathbb{R}^m$ exists and is unique if and only if the matrix A is nonsingular.
We will investigate the condition of the problem to solve v from $Av = b$, where A and b are seen as input data for that problem. It will turn out that the condition can be nicely quantified in terms of the so-called condition number of the matrix A. The numerical stability of the numerical methods will also be addressed.
Finally we will have a look at overdetermined systems, which have more equations than unknowns. Then a least-squares solution can be found by solving the so-called normal equations or via a QR-decomposition.

2.1 The Condition Number of a Matrix

In this section we investigate the condition of the problem to solve v from $Av = b$, where A and b are seen as the input data for that problem, and $v = f(A, b) = A^{-1}b$ the solution. It will turn out that the condition can be nicely quantified in terms of the so-called *condition number* of the matrix A.
We assume that a vector norm on $\mathbb{R}^m$ and its *induced matrix norm* on $\mathbb{R}^{m\times m}$ are given. Both norms are denoted by $\|\cdot\|_p$, for $p \in \mathbb{N}$. As can be read in Appendix A.4, the induced matrix norm of a matrix $A \in \mathbb{R}^{m\times m}$ is defined as

$$\|A\|_p = \max_{v\neq 0} \frac{\|Av\|_p}{\|v\|_p} = \max_{\|v\|=1} \|Av\|_p.$$

We define the *condition number* of A as

$$\kappa_p(A) = \|A\|_p \cdot \|A^{-1}\|_p. \tag{2.2}$$

We now see how to express the error in the solution to a perturbed linear system in terms of the condition number of the system's matrix. Let us consider, along with the original linear system $Av = b$, the perturbed system $\widetilde{A}\widetilde{v} = \widetilde{b}$, where $\widetilde{A} = A + \Delta A$ and $\widetilde{b} = b + \Delta b$. Suppose that the perturbations ΔA and Δb satisfy

$$\|\Delta A\|_p \le \varepsilon_A \|A\|_p , \qquad \|\Delta b\|_p \le \varepsilon_b \|b\|_p . \tag{2.3}$$

For example, $\widetilde{A}$, $\widetilde{b}$ may stand for the computer representations of the exact A and b, with relative errors ε_A and ε_b. We want to know how much such errors will influence the solution of the system. In order to prove our main result we need a small lemma. There, we omit the subscript p for simplicity, and $\|\cdot\|$ denotes any induced matrix norm.

Lemma 2.1. *Let $F \in \mathbb{R}^{m\times m}$ be given with induced matrix norm $\|F\| < 1$. Then $I + F$ is nonsingular and $\|(I + F)^{-1}\| \le (1 - \|F\|)^{-1}$.*

Proof. Suppose $I + F$ is singular. Then there exists a nonzero vector $v \in \mathbb{R}^m$ with $v = -Fv$ and therefore also $\|Fv\| = \|v\|$. This implies that $\|F\| \ge 1$, which is a contradiction.
Let $v \in \mathbb{R}^m$ be arbitrary and $w = (I+F)^{-1}v$. Then $w+Fw = v$ and therefore $\|w\| = \|v-Fw\| \le \|v\| + \|Fw\| \le \|v\| + \|F\|\|w\|$. Hence $\|w\| \le (1 - \|F\|)^{-1}\|v\|$. Alternatively, the result can be proved using the Neumann series. □

Theorem 2.2. *Suppose A is nonsingular, $b \neq 0$, $\kappa_p(A)$ is the condition number of A using the p-norm and $\varepsilon_A \cdot \kappa_p < 1$. Then $\widetilde{A}$ is also nonsingular and*

$$\frac{\|\widetilde{v} - v\|_p}{\|v\|_p} \;\le\; \frac{\kappa_p}{1 - \varepsilon_A\, \kappa_p} \cdot (\varepsilon_A + \varepsilon_b) .$$

Proof. In this proof, we omit the subscript p to lighten the notation. Writing $\widetilde{A} = A + \Delta A = A(I + F)$ with $F = A^{-1}\Delta A$ we find $\|F\| \le \|A^{-1}\|\|\Delta A\| \le \varepsilon_A\kappa < 1$, where we used the submultiplicativity[1] of the induced matrix norm $\|\cdot\|$. It follows from Lemma 2.1 that $I + F$ is nonsingular and $\|(I + F)^{-1}\| \le (1 - \|F\|)^{-1} \le (1 - \varepsilon_A\kappa)^{-1}$. Note that $\widetilde{A}$ must then be nonsingular too, with inverse $\big(\widetilde{A}\big)^{-1} = (I + F)^{-1}A^{-1}$, so that

$$\|\big(\widetilde{A}\big)^{-1}\| \le \|(I + F)^{-1}\|\|A^{-1}\| \le (1 - \varepsilon_A\kappa)^{-1}\|A^{-1}\|.$$

Since

$$\Delta b = \widetilde{b} - b = \widetilde{A}\widetilde{v} - Av = \widetilde{A}(\widetilde{v} - v) + (\widetilde{A} - A)v = \widetilde{A}(\widetilde{v} - v) + (\Delta A)v,$$

we have

$$\widetilde{v} - v \;=\; \big(\widetilde{A}\big)^{-1}\Big(- (\Delta A)v + \Delta b\Big) .$$

Taking norms, and using

$$\|(\Delta A)v\| \le \|\Delta A\|\|v\| \le \varepsilon_A\|A\|\|v\|,$$
$$\|\Delta b\| \le \varepsilon_b\|b\| = \varepsilon_b\|Av\| \le \varepsilon_b\|A\|\|v\|,$$

[1] see Exercise A.1(b).

it follows that

$$\|\widetilde{v} - v\| \;\leq\; (1 - \varepsilon_A \kappa)^{-1} \|A^{-1}\| \left(\varepsilon_A \|A\| \|v\| \;+\; \varepsilon_b \|A\| \|v\| \right),$$

which furnishes the proof. □

The above result shows that the problem of solving a linear system $Av = b$ is *ill-conditioned* if the matrix A has a large condition number. This is a property of the linear system, and not of the numerical procedure that is used to solve the system. The matrix in Example 2.9 has a moderate condition number in any of the norms (A.1), and in that example we just had to change the computation a bit (by a permutation) to avoid large round-off errors.

Example 2.3. A notorious example of an ill-conditioned system involves the Hilbert matrix $H = (h_{ij})$. This is a symmetric matrix with elements given by

$$h_{ij} = \int_0^1 x^{i-1} x^{j-1} dx = \frac{1}{i + j - 1} \quad \text{for } 1 \leq i, j \leq m.$$

The first row of the table below shows that the condition number $\text{cond}_\infty(H)$ (as computed in PYTHON with the function '`numpy.linalg.cond`') quickly becomes large as m increases.

m	4	6	8	10	12	14
$\text{cond}_\infty(H)$	$2.84 \cdot 10^4$	$2.91 \cdot 10^7$	$3.39 \cdot 10^{10}$	$3.54 \cdot 10^{13}$	$3.92 \cdot 10^{16}$	$1.69 \cdot 10^{18}$
$\|v - e\|_\infty$	$1.95 \cdot 10^{-13}$	$4.15 \cdot 10^{-10}$	$2.35 \cdot 10^{-7}$	$8.15 \cdot 10^{-4}$	$3.85 \cdot 10^{-1}$	$7.56 \cdot 10^{0}$

If we use in PYTHON the function '`numpy.linalg.solve`' to solve the problem $Hv = b$ where $b = He$ with $e = (1, \ldots, 1)^T$ in $\mathbb{R}^m$, the computed solution vector v has a large error $\|v - e\|_\infty$, which is shown in the second row of the table. For $m = 10$ some components of v are only correct to 3 significant digits while for $m = 12$ and $m = 14$ the obtained vector v is pretty useless. This bad result is due the large condition number of H, which causes an unfavourable propagation of rounding errors. ◊

The following two remarks help getting more intuition on the meaning of the condition number of a matrix.

Remark 2.4. *(Spectral condition number) A norm that is often used for the condition number is the $\|\cdot\|_2$-norm, that is the Euclidean norm for vectors. In this case, the induced matrix norm can be written as $\|A\|_2 = \sigma_{\max}(A)$, where $\sigma_{\max}(A)$ denotes the largest singular value of A. It follows that $\|A^{-1}\|_2 = \sigma_{\max}(A^{-1}) = \frac{1}{\sigma_{\min}(A)}$, where $\sigma_{\min}(A)$ is the smallest singular value of A (which is positive because A is invertible). Then the condition number is $\kappa_2(A) = \frac{\sigma_{\max}(A)}{\sigma_{\min}(A)}$ and it is called spectral condition number. This has a nice geometrical interpretation, as it measures how much the application of the matrix A stretches a bounded region in $\mathbb{R}^m$.*

Remark 2.5. *For a non-singular matrix $A \in \mathbb{R}^{m \times m}$, it is possible to define its relative distance from the set of singular matrices with respect to the p-norm by*

$$\text{dist}_p(A) := \min \left\{ \frac{\|\delta A\|_p}{\|\delta A\|_p} : A + \delta A \text{ is singular} \right\}.$$

It can be shown [9, 3] that

$$\mathrm{dist}_p(A) = \frac{1}{\kappa_p(A)}.$$

This suggests that one can interpret a matrix with a large condition number as a matrix that behaves almost as a singular matrix of the form $A + \delta A$.

2.2 Gaussian Elimination and LU-Decomposition

A straightforward—but still popular—method to solve linear equations is Gaussian elimination. The system (2.1), written out, reads

$$\begin{aligned} a_{11}v_1 + a_{12}v_2 + \cdots + a_{1m}v_m &= b_1\,, \\ a_{21}v_1 + a_{22}v_2 + \cdots + a_{2m}v_m &= b_2\,, \\ \vdots \qquad\quad & \vdots \qquad\qquad \vdots \qquad\quad \vdots \\ a_{m1}v_1 + a_{m2}v_2 + \cdots + a_{mm}v_m &= b_m\,. \end{aligned} \tag{2.4}$$

Assume $a_{11} \neq 0$. Then we can eliminate v_1 from the last $m-1$ equations by subtracting from the i-th equation the first equation multiplied by $\lambda_{i1} = a_{i1}/a_{11}$. After this, the system becomes

$$\begin{aligned} a_{11}v_1 + a_{12}v_2 + \cdots + a_{1m}v_m &= b_1\,, \\ a^{(2)}_{22}v_2 + \cdots + a^{(2)}_{2m}v_m &= b^{(2)}_2, \\ \vdots \qquad\quad & \vdots \qquad\quad \vdots \\ a^{(2)}_{m2}v_2 + \cdots + a^{(2)}_{mm}v_m &= b^{(2)}_m, \end{aligned} \tag{2.5}$$

where $a^{(2)}_{ij} = a_{ij} - \lambda_{i1}a_{1j}$ and $b^{(2)}_i = b_i - \lambda_{i1}b_1$. This provides a system of $m-1$ equations for $v_2, \cdots, v_m$, with an extra equation that determines v_1. If $a^{(2)}_{22} \neq 0$ we can proceed in the same way to obtain a system of $m-2$ equations in the unknowns $v_3, \cdots, v_m$ with coefficients $a^{(3)}_{ij} = a^{(2)}_{ij} - \lambda_{i2}a^{(2)}_{2j}$ and $b^{(3)}_i = b^{(2)}_i - \lambda_{i2}b^{(2)}_2$, $\lambda_{i2} = a^{(2)}_{i2}/a^{(2)}_{22}$. Repeating this $m-1$ times we are finally left with a single equation $a^{(m)}_{mm}v_m = b^{(m)}_m$. Then, collecting the first equation from each step gives

$$\begin{aligned} a^{(1)}_{11}v_1 + a^{(1)}_{12}v_2 + \cdots + a^{(1)}_{1m}v_m &= b^{(1)}_1, \\ a^{(2)}_{22}v_2 + \cdots + a^{(2)}_{2m}v_m &= b^{(2)}_2, \\ \ddots \qquad & \vdots \qquad\quad \vdots \\ a^{(m)}_{mm}v_m &= b^{(m)}_m, \end{aligned} \tag{2.6}$$

where we denoted $a^{(1)}_{ij} = a_{ij}$ and $b^{(1)}_i = b_i$. This system is now easily solved:

$$v_m = \frac{1}{a^{(m)}_{mm}}b^{(m)}_m \quad \text{and} \quad v_i = \frac{1}{a^{(i)}_{ii}}\Big(b^{(i)}_i - \sum_{j=i+1}^{m} a^{(i)}_{ij}v_j\Big) \quad (i = m-1, \ldots, 2, 1)\,. \tag{2.7}$$

Example 2.6. We use Gaussian elimination to solve the linear system $Av = b$ with

$$A = \begin{pmatrix} 1 & \frac{1}{2} & \frac{1}{3} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} \end{pmatrix} \quad \text{and} \quad b = \begin{pmatrix} \frac{11}{6} \\ \frac{13}{12} \\ \frac{47}{60} \end{pmatrix}.$$

We note in passing that A is the Hilbert matrix for $m = 3$, cf. Example 2.3. The linear system has the unique solution $v = (1, 1, 1)^\top$. In the first step of Gaussian elimination, we set $A^{(1)} = A$ and $b^{(1)} = b$. We compute the multipliers $\lambda_{21} = \frac{1}{2}$ and $\lambda_{31} = \frac{1}{3}$. Subtracting from the second and third equations of the system the first row multiplied by λ_{21} and λ_{31} respectively, we obtain the equivalent system $A^{(2)}v = b^{(2)}$ where

$$A^{(2)} = \begin{pmatrix} 1 & \frac{1}{2} & \frac{1}{3} \\ 0 & \frac{1}{12} & \frac{1}{12} \\ 0 & \frac{1}{12} & \frac{4}{45} \end{pmatrix} \quad \text{and} \quad b^{(2)} = \begin{pmatrix} \frac{11}{6} \\ \frac{1}{6} \\ \frac{31}{180} \end{pmatrix}.$$

If we now subtract the second row multiplied by $\lambda_{32} = 1$ from the third one, we end up with the upper triangular system $A^{(3)}v = b^{(3)}$ where

$$A^{(3)} = \begin{pmatrix} 1 & \frac{1}{2} & \frac{1}{3} \\ 0 & \frac{1}{12} & \frac{1}{12} \\ 0 & 0 & \frac{1}{180} \end{pmatrix} \quad \text{and} \quad b^{(3)} = \begin{pmatrix} \frac{11}{6} \\ \frac{1}{6} \\ \frac{1}{180} \end{pmatrix}.$$

From the last system we immediately compute $v_3 = 1$ and then, by backward substitution, the remaining unknowns $v_1 = v_2 = 1$. ◊

The elements $a_{kk}^{(k)}$ are called *pivot elements*. If in the k-th step we find $a_{kk}^{(k)} = 0$, then there is some other $a_{\ell k}^{(k)} \neq 0$, because otherwise the first k columns of the original matrix A are linearly dependent, which contradicts that A is nonsingular. Hence we can perform a simple permutation, interchanging the rows k and ℓ, and then continue the procedure. In practice, it will be necessary to perform such a row interchange not only if $a_{kk}^{(k)}$ is precisely zero, but also if it is nearly zero, because division by a very small number will introduce very large numbers in the calculation, which may well lead to instability (large rounding errors). We therefore select the index $\ell \geq k$ such that $a_{\ell k}^{(k)}$ has the largest absolute value. This is called Gaussian elimination with *partial pivoting*. (One can also use *complete pivoting*, where also columns are interchanged, but in practice partial pivoting is usually sufficient.)

The permutations are done during the process. Afterwards all these row permutations together are described by a permutation matrix P such that the application of the Gaussian elimination process to the permuted system $PAv = Pb$ does not require any row permutations. Setting $A^{(1)} = PA$ and $b^{(1)} = Pb$ the total process thus can be described formally as

$$[PA, Pb] = [A^{(1)}, b^{(1)}] \;\to\; [A^{(2)}, b^{(2)}] \;\to\; \cdots \;\to\; [A^{(m)}, b^{(m)}] = [U, b^{(m)}], \tag{2.8}$$

where the first $k-1$ columns of $A^{(k)}$ have zero entries below the diagonal, and U is the resulting upper triangular matrix.

In the following, let I stand for the $m \times m$ identity matrix, and let E_{ij} be the $m \times m$ matrix that has entry 1 at the i,j-th position and entries 0 elsewhere. Then $E_{ij}A$ is the matrix with all rows zero, except for the i-th row which is just the j-th row of A. The steps in the Gaussian elimination can be described as

$$A^{(k+1)} = L_k A^{(k)}\,, \qquad b^{(k+1)} = L_k b^{(k)} \qquad \text{with} \quad L_k = I - \sum_{j:j>k} \lambda_{jk} E_{jk}\,. \tag{2.9}$$

These matrices L_k have the form

$$L_1 = \begin{pmatrix} 1 & & & & \\ -\lambda_{21} & 1 & & & \\ -\lambda_{31} & & 1 & & \\ \vdots & & & \ddots & \\ -\lambda_{m1} & & & & 1 \end{pmatrix}, \qquad L_2 = \begin{pmatrix} 1 & & & & \\ & 1 & & & \\ & -\lambda_{32} & 1 & & \\ & \vdots & & \ddots & \\ & -\lambda_{m2} & & & 1 \end{pmatrix},$$

etc., with numbers zero on the empty positions.

Theorem 2.7. *Suppose A is nonsingular. Then Gaussian elimination gives*

$$PA = L\,U\,, \qquad L = I + \sum_{i>j} \lambda_{ij}\, E_{ij}\,,$$

with permutation matrix P and upper triangular matrix U.

Proof. It is clear that $U = A^{(m)}$ equals $U = L_{m-1}L_{m-2}\cdots L_1 PA$, that is,

$$PA = (L_{m-1}L_{m-2}\cdots L_1)^{-1}U\,.$$

It remains to be shown that this inverse equals L. To see this we apply to L the same elimination steps that we applied to PA. Multiplication of L by L_1 from the left eliminates the elements of the first column below the diagonal, then multiplication by L_2 from the left clears the elements of the second column below the diagonal, and so on. Therefore $(L_{m-1}L_{m-2}\cdots L_1)L = I$. □

Such decomposition of PA into a lower triangular L and upper triangular U is called an *LU-decomposition* or *LU-factorization*. Solving $Av = b$ amounts to

(*i*) compute the LU-decomposition, and

(*ii*) solve the triangular systems $Lw = Pb$, $Uv = w$.

Note in particular that the theorem above gives us an explicit formula for L, as

$$L = \begin{pmatrix} 1 & & & & \\ \lambda_{21} & 1 & & & \\ \vdots & -\lambda_{32} & 1 & & \\ \vdots & \vdots & & \ddots & \\ \lambda_{m1} & -\lambda_{m2} & & & 1 \end{pmatrix},$$

so, when implementing the LU-decomposition, we obtain L by simply saving the multiplicators in the lower triangular part of its columns.

Saving the factors L and U is useful if we need to solve, one after the other, a number of linear systems $Av = b$ with the same matrix A but different vectors b. An example of this situation will be given in Section 4.2 for one of the versions of the modified Newton method.

Further we note that, for a given permutation, the factors L and U are unique. For, if we have a second decomposition $PA = \widetilde{L}\widetilde{U}$, then $\widetilde{L}^{-1}L = \widetilde{U}U^{-1}$. Since $\widetilde{L}^{-1}L$ is lower triangular and $\widetilde{U}U^{-1}$ is upper triangular, there is a diagonal D such that $L = \widetilde{L}D$ and $\widetilde{U} = DU$. The requirement that the diagonal elements of L and $\widetilde{L}$ are 1 specifies $D = I$.

Example 2.8. *In Example 2.6, we have $P = I$ because we did not perform pivoting. Moreover, we have*

$$L = \begin{pmatrix} 1 & 0 & 0 \\ \frac{1}{2} & 1 & 0 \\ \frac{1}{3} & 1 & 1 \end{pmatrix} \quad \textit{and} \quad U = \begin{pmatrix} 1 & \frac{1}{2} & \frac{1}{3} \\ 0 & \frac{1}{12} & \frac{1}{12} \\ 0 & 0 & \frac{1}{180} \end{pmatrix}.$$

◊

Stability of Gaussian elimination. It was mentioned above that (partial) pivoting is necessary because pivots close to zero will introduce large numbers in the calculation, which may well lead to instability (large rounding errors). We will illustrate this in the following example.

Example 2.9. Consider

$$A = \begin{pmatrix} 0.004 & 3 \\ 1 & 1 \end{pmatrix}, \qquad b = \begin{pmatrix} 3 \\ 2 \end{pmatrix}.$$

Clearly, the element $a_{11} = 0.004$ is close to zero, so should be avoided as the pivot. If we nevertheless choose a_{11} as the pivot, then performing Gaussian elimination with exact arithmetic gives

$$\begin{aligned} \lambda_{21} &= a_{21}/a_{11} = 1/0.004 = 250, \\ a_{22}^{(2)} &= a_{22} - \lambda_{21}a_{12} = 1 - 250 \times 3 = -749, \\ b_2^{(2)} &= b_2 - \lambda_{21}b_1 = 2 - 250 \times 3 = -748, \\ v_2 &= b_2^{(2)}/a_{22}^{(2)} = 748/749, \\ v_1 &= (b_1 - a_{12}v_2)/a_{11} = (3 - 3 \times v_2)/0.004 = 750/749. \end{aligned}$$

Of course, with exact arithmetic, the answer is exact too. In decimal representation, the solution is given by $v_1 = 1.0013...$ and $v_2 = 0.9986...$.

Now let us simulate floating-point arithmetic with base $B = 10$ and $n = 3$ significant digits.

Then we get

$$\widetilde{\lambda}_{21} = 1 \oslash 0.004 = 250, \tag{2.10a}$$

$$\widetilde{a}_{22}^{(2)} = 1 \ominus 250 \otimes 3 = -749, \tag{2.10b}$$

$$\widetilde{b}_{2}^{(2)} = 2 \ominus 250 \otimes 3 = -748, \tag{2.10c}$$

$$\widetilde{v}_2 = 748 \oslash 749 = 0.999, \tag{2.10d}$$

$$\widetilde{v}_1 = (3 \ominus 3 \otimes 0.999) \oslash 0.004 = (3 \ominus 3) \oslash 0.004 = 0. \tag{2.10e}$$

Hence, Gaussian elimination (with pivot $a_{11} = 0.004$) leads with floating point arithmetic to the extremely inaccurate result $\widetilde{v}_1 = 0$. Although the calculated $\widetilde{v}_2$ was equal to the rounded value 0.999 of the exact solution value $v_2 = 748/749$ (which is the best possible result), the corresponding (moderate) rounding error in $\widetilde{v}_2$ was magnified tremendously in the calculation of $\widetilde{v}_1$ due to severe cancellation. Indeed, the remainder function R that maps the intermediate result v_2 to the final result v_1 is given by $v_1 = R(v_2) = (b_1 - a_{12}v_2)/a_{11} = 750(1 - v_2)$, and the corresponding condition number (of v_1 with respect to v_2) at $v_2 = 748/749$ is equal to

$$\sigma = \frac{v_2}{v_1} R'(v_2) = -750 \frac{v_2}{v_1} = -748. \tag{2.11}$$

We note that the root cause of the cancellation is the small size of the pivot $a_{11} = 0.004$, which caused a large size of the multiplier $\lambda_{21} = 1/0.004 = 250$, and therefore a value of $v_2 = b_2^{(2)}/a_{22}^{(2)} = (b_2 - \lambda_{21}b_1)/(a_{22} - \lambda_{21}a_{12})$ close to b_1/a_{12}. In the calculation of $v_1 = (b_1 - a_{12}v_2)/a_{11}$ we must therefore compute the difference of the almost equal numbers b_1 and $a_{12}v_2$, leading to severe cancellation.

If we apply Gaussian elimination *with partial pivoting*, the first and second row must be interchanged and the element $a_{21} = 1$ becomes the pivot. In that case we find, using floating-point arithmetic, the approximate values $\widetilde{v}_1 = 1.00$ and $\widetilde{v}_2 = 0.997$, both within 0.2 percent of the exact value. The remainder function R that maps v_2 to v_1 is now given by $v_1 = R(v_2) = (b_2 - a_{22}v_2)/a_{21} = 2 - v_2$ with corresponding moderate condition number

$$\sigma = \frac{v_2}{v_1} R'(v_2) = -\frac{v_2}{v_1} = -748/750 \approx -1. \tag{2.12}$$

Clearly, there is no severe cancellation in this case. ◊

In the above example we saw that partial pivoting turned out to be successful: the largest pivot $a_{21} = 1$ performed much better than the smaller pivot $a_{11} = 0.004$. It might be tempting to expect that this is always the case, but that cannot be unconditionally true. Namely, if we would multiply the first equation of the above example by 1000, we would obtain

$$A = \begin{pmatrix} 4 & 3000 \\ 1 & 1 \end{pmatrix}, \qquad b = \begin{pmatrix} 3000 \\ 2 \end{pmatrix}.$$

The pivot with largest absolute value is now $a_{11} = 4$, but with this pivot choice we get the same bad results as with $a_{11} = 0.004$ in the unscaled case. This shows that scaling can affect the pivot

choice in partial pivoting, and (depending on the way of scaling) may lead to the wrong pivot. To better understand the role of scaling, we focus on the root cause of the bad results, which was the severe cancellation in the computation of v_1 from v_2. The severity of the cancellation was quantified by the condition number σ of the remainder function R that mapped v_2 to v_1. This condition number is equal to

$$\sigma = \frac{v_2}{v_1} R'(v_2) = \begin{cases} -\dfrac{a_{12} v_2}{a_{11} v_1} & \text{if } a_{11} \text{ is chosen as pivot (no row interchange),} \\ -\dfrac{a_{22} v_2}{a_{21} v_1} & \text{if } a_{21} \text{ is chosen as pivot (row interchange).} \end{cases}$$

The best pivot is the one that gives the smallest value of $|\sigma|$, so we should choose a_{11} if and only if $|a_{11}/a_{12}| \geq |a_{21}/a_{22}|$. Note that this criterion is independent of row scaling as it compares the *relative* size of the two pivot candidates, i.e., the size relative to the other element in the same row. We note that the criterion (for choosing a_{11} as the pivot) can also be rewritten as

$$\frac{|a_{11}|}{|a_{11}| + |a_{12}|} \geq \frac{|a_{21}|}{|a_{21}| + |a_{22}|}. \tag{2.13}$$

This shows that 'ordinary' partial pivoting can be improved by 'scaled' partial pivoting for 2×2-matrices. Criterion (2.13) is easily generalized to arbitrary $m \times m$-matrices, but in practice these scaling-based improvement techniques are seldom used. The reason is that the 'ordinary' partial pivoting strategy almost always behaves satisfactorily, with only few exceptions. There is a vast amount of literature on this topic which provides valuable insights into the stability behaviour of this method, but that is beyond the scope of this course. The brief summary (see [5]) is that 'ordinary' partial pivoting can be used with confidence and generally leads to a stable algorithm. If in doubt, 'complete' pivoting can be used for better stability, but this is more expensive and hardly used in practice.

Computational costs. One way to estimate the computational costs of an algorithm is to count the number of floating point operations $(+, -, \times, /)$, commonly referred to as the number of *flops*. In the following we will do a flop count for Gaussian elimination.
First note that the computation of $A^{(2)}$ from $A^{(1)} = A$ requires $(m-1)$ divisions, $(m-1)^2$ multiplications and $(m-1)^2$ subtractions. So for large m this takes approximately $2(m-1)^2$ flops. Then to find $A^{(3)}$ we need approximately another $2(m-2)^2$ operations. In total, finding the LU decomposition requires approximately $\sum_{j=1}^{m-1} 2j^2 \approx \int_0^m 2x^2 \, \mathrm{d}x = \frac{2}{3}m^3$ flops.
The computation of $b^{(2)}$ costs $2(m-1)$ flops, the computation of $b^{(3)}$ another $2(m-2)$ flops, etc. In total, the computation of $b^{(m)}$ takes $\sum_{j=1}^{m-1} 2j \approx \int_0^m 2x \, \mathrm{d}x = m^2$ flops. Likewise, solving the triangular system (2.6) via (2.7) costs approximately m^2 flops. For large m this is negligible compared to the $\frac{2}{3}m^3$ flops for finding L and U.
Gaussian elimination to solve (2.1) thus takes approximately $\frac{2}{3}m^3$ flops. We emphasize that estimating the computational costs this way, by counting the number of flops, only gives a rough indication for the actual runtime on a computer. Memory handling is not taken into account and computers increasingly possess vector and parallel capabilities. Still, it gives an indication.

Note in particular that inverting a matrix A would require $\frac{5}{3}m^3$ flops ($\frac{2}{3}m^3$ for the LU-decomposition, plus the solution of m linear systems with right-hand sides e_k, $k = 1, \ldots, m$). So, unless m is very small, it is *strongly discouraged to invert a matrix to solve a linear system.*

LU-decomposition for band matrices. We say that the $m \times m$ matrix $A = (a_{ij})$ has *lower bandwidth* p if $a_{ij} = 0$ whenever $i > j + p$ and *upper bandwidth* q if $a_{ij} = 0$ whenever $j > i + q$. Familiar examples of band matrices are *diagonal* matrices ($p = q = 0$), *tridiagonal* matrices ($p = q = 1$), *lower triangular* matrices ($p = m - 1$, $q = 0$) and *upper triangular* matrices ($p = 0$, $q = m - 1$).
If Gaussian elimination can be performed without permutations, then the computational costs are greatly reduced for band matrices A when using implementations that exploit the band structure, since L inherits the lower bandwidth p and U the upper bandwidth q of A. If $p \ll m$ and $q \ll m$ then the computation of L and U only requires approximately $p(2q+1)m$ flops, and the computation of the solution v from the systems $Lw = b$, $Uv = w$ only approximately $(2p + 2q+1)m$ flops. In total this costs $(2pq+3p+2q+1)m$ flops, which is linear in m and significantly less than the $\frac{2}{3}m^3$ flops required for a full matrix. Moreover, and often more importantly, *storage* of L and U is greatly reduced from m^2 positions to approximately $(p + q + 1)m$ positions when using the sparse format (that is, when saving only the non-zero entries).

Example 2.10. For tridiagonal systems we have $p = q = 1$. Let

$$A = \begin{pmatrix} \alpha_1 & \beta_2 & & \\ \gamma_2 & \alpha_2 & \ddots & \\ & \ddots & \ddots & \beta_m \\ & & \gamma_m & \alpha_m \end{pmatrix}, \quad L = \begin{pmatrix} 1 & & & \\ \lambda_2 & 1 & & \\ & \ddots & \ddots & \\ & & \lambda_m & 1 \end{pmatrix}, \quad U = \begin{pmatrix} \rho_1 & \sigma_2 & & \\ & \rho_2 & \ddots & \\ & & \ddots & \sigma_m \\ & & & \rho_m \end{pmatrix}.$$

We have $A = LU$ iff $\alpha_1 = \rho_1$ and $\gamma_j = \lambda_j \rho_{j-1}$, $\alpha_j = \lambda_j \sigma_j + \rho_j$, $\beta_j = \sigma_j$ $(2 \le j \le m)$. Assuming that Gaussian elimination can be applied to A without permutations, we can thus compute the $\lambda_j, \rho_j, \sigma_j$ recursively as $\rho_1 = \alpha_1$,

$$\lambda_j = \gamma_j / \rho_{j-1}\,, \quad \rho_j = \alpha_j - \lambda_j \beta_j\,, \quad \sigma_j = \beta_j \quad (2 \le j \le m)\,.$$

This requires $m-1$ divisions, multiplications and subtraction, so $3(m-1)$ flops. Finally, solving $Av = b$ reduces to $Lw = b$, $Uv = w$, and these two bi-diagonal systems together require $5m-4$ flops. In total we therefore just need $8m-7$ flops. Storage of L and U takes $3m-2$ positions. $\Diamond$

2.3 The Cholesky Decomposition

As mentioned in Appendix A.5, a matrix $A \in \mathbb{R}^{m \times m}$ is called *symmetric* if $A^T = A$, and a symmetric matrix is called *positive definite* if $v^T A v > 0$ for all nonzero $v \in \mathbb{R}^m$. Positive definite matrices arise in many applications, for example in numerical methods for partial differential equations. The next result shows that then the resulting systems can be solved with Gaussian elimination without permutations, and in the LU-decomposition we have $U = DL^T$ with a diagonal matrix D that has positive diagonal entries. Writing $C = \sqrt{D}L^T$ gives $A = C^T C$.

Theorem 2.11. *Let A be symmetric and positive definite. Then there exists a unique upper triangular matrix C with positive diagonal entries such that $A = C^T C$. The entries of C^T are given as*

$$
\begin{aligned}
c_{11} &= \sqrt{a_{11}}, \\
c_{ij} &= \frac{1}{c_{jj}} \left(a_{ij} - \sum_{k=1}^{j-1} c_{ik} c_{jk} \right), \quad \text{for } i = 2, \ldots, m \text{ and } j = 1, \ldots, i-1, \\
c_{ii} &= \left(a_{ii} - \sum_{k=1}^{i-1} c_{ik}^2 \right)^{\frac{1}{2}}, \qquad \text{for } i = 2, \ldots, m.
\end{aligned}
$$

Proof. To prove this result we use induction on the dimension m. The statement of the theorem is obviously true if $m = 1$. Now let $m > 1$ and write

$$
A = \begin{pmatrix} \overline{A} & \overline{a} \\ \overline{a}^T & \alpha \end{pmatrix}, \qquad C = \begin{pmatrix} \overline{C} & \overline{c} \\ \overline{o}^T & \gamma \end{pmatrix},
$$

with scalars α, γ, with $\overline{A}, \overline{C} \in \mathbb{R}^{\overline{m} \times \overline{m}}$, $\overline{a}, \overline{c} \in \mathbb{R}^{\overline{m}}$, $\overline{m} = m-1$, and zero vector $\overline{o} \in \mathbb{R}^{\overline{m}}$. Since A is positive definite, the same holds for $\overline{A}$.
Suppose that $\overline{A} = \overline{C}^T \overline{C}$ and $\overline{C}$ is upper triangular with positive diagonal entries (induction assumption). Then $A = C^T C$ iff $\overline{a} = \overline{C}^T \overline{c}$ and $\alpha = \overline{c}^T \overline{c} + \gamma^2$. So, we can compute $\overline{c}$ by solving the lower triangular system $\overline{C}^T \overline{c} = \overline{a}$ and then set $\gamma = \sqrt{\alpha - \overline{c}^T \overline{c}}$. It seems this γ might be purely imaginary. However we have $\det(A) = \det(C^T) \det(C) = (\gamma \det(\overline{C}))^2$, and since the determinant of the positive definite matrix A is positive (it is the product of the eigenvalues and these are all positive) we must have $\gamma^2 > 0$.
We now turn to the formulae for the entries of C. The fact that $c_{11} = \sqrt{a_{11}}$ follows immediately from the induction argument for $i = 1$. For $i \geq 2$, the expressions for the entries are the forward substitution formulae for the solution of the linear system $\overline{C}^T \overline{c} = \overline{a}$ when $j \neq i$, and they correspond to the expression for γ when $j = i$. □

The decomposition $A = C^T C$ for positive definite A is called a *Cholesky decomposition.*

Example 2.12. *Consider the matrix*

$$
A = \begin{pmatrix} 1 & -1 & 2 \\ -1 & 5 & -4 \\ 2 & -4 & 6 \end{pmatrix}.
$$

The matrix is symmetric, and it is easily checked that it is also positive definite. It admits therefore a Cholesky decomposition $A = C^\top C$, and

$$
C = \begin{pmatrix} 1 & -1 & 2 \\ 0 & 2 & -1 \\ 0 & 0 & 1 \end{pmatrix}.
$$

◊

Computational cost. It can be seen from the construction used in the above proof—with the repeated solution of lower triangular systems—that the number of flops in a Cholesky decomposition is roughly $\frac{1}{3}m^3$, which is about half the number required for a general LU-decomposition.

2.4 Overdetermined Linear Systems and QR-decomposition

If A is an $m \times n$ matrix with $m > n$ and $b \in \mathbb{R}^m$, then the system $Av = b$ reads

$$\begin{aligned} a_{11}v_1 + a_{12}v_2 + \cdots + a_{1n}v_n &= b_1\,, \\ a_{21}v_1 + a_{22}v_2 + \cdots + a_{2n}v_n &= b_2\,, \\ \vdots \qquad\quad \vdots \qquad\qquad \vdots \qquad\quad \vdots \\ a_{m1}v_1 + a_{m2}v_2 + \cdots + a_{mn}v_n &= b_m\,. \end{aligned} \tag{2.14}$$

Since this system has more equations than unknowns, we call it an *overdetermined* system. It generally has no 'ordinary' solution $v \in \mathbb{R}^n$ (corresponding to a zero residual vector $Av - b$) but instead we can find a v that minimizes the norm of the residual vector,

$$\|Av - b\| = \min_{w \in \mathbb{R}^n} \|Aw - b\|, \tag{2.15}$$

where $\|u\| = \|u\|_2 = \sqrt{u^T u}$ is the Euclidean norm on $\mathbb{R}^m$. Such a v is called a *least-squares* solution of the overdetermined system (2.14).
Let us define the function $\varphi : \mathbb{R}^n \to \mathbb{R}$ defined by

$$\varphi(v) = \|Av - b\|^2 = (Av - b)^T(Av - b) = v^T A^T Av - 2b^T Av + b^T b.$$

One easily verifies that

$$\tfrac{1}{h}\Big(\varphi(v + hw) - \varphi(v)\Big) \;=\; 2(v^T A^T A - b^T A)w \;+\; h \cdot w^T A^T Aw\,,$$

and taking $h \to 0$ we see that

$$\varphi'(v) = 2(v^T A^T A - b^T A).$$

Consequently, the least-squares solution satisfies

$$A^T A\, v \;=\; A^T b\,. \tag{2.16}$$

These linear equations, with $n \times n$ matrix $A^T A$, are called the *normal equations*.
The matrix $A^T A$ is obviously symmetric. Moreover, it is easily seen that if the columns of A are linearly independent, then $A^T A$ is positive definite, so then the least-squares solution is unique and can be found by a Cholesky decomposition.
Least-squares problems often arise when one wants to fit some simple function to given (measured) data.

Example 2.13. Let points $(x_i, y_i) \in \mathbb{R}^2$ $(1 \le i \le m)$ be given. We want to find the coefficients of the polynomial $p(x) = \alpha x^2 + \beta x + \gamma$ such that the sum

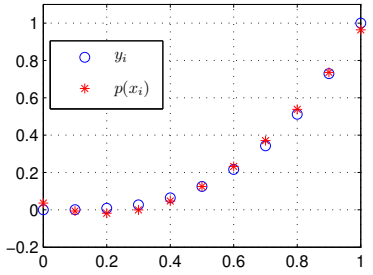
$$S = \sum_{i=1}^{m} |p(x_i) - y_i|^2 \quad \text{is minimized.}$$

One easily verifies that this is a least-squares problem for the overdetermined linear system $Av = b$ with $a_{ij} = x_i^{j-1}$, $b_i = y_i$ and $v = (\alpha, \beta, \gamma)^T \in \mathbb{R}^3$.

Let us take as a simple illustration

$$x_i = \tfrac{i-1}{m-1}\,, \quad y_i = x_i^3$$

for $i = 1, \ldots, m$ with $m = 11$. The values $p(x_i)$ of the quadratic polynomial with best least-squares fit are shown in the plot on the right.



◊

Solving a least-squares problem via the normal equations (and Choleski decomposition) works often satisfactorily, but not always. If the columns of the matrix A are almost linearly dependent, the singular nature of A is enforced in the matrix A^TA and may lead to problems. This is illustrated in the following example.

Example 2.14. Consider the overdetermined system $Av = b$ with

$$A = \begin{pmatrix} 1 & 1 \\ \varepsilon & 0 \\ 0 & \varepsilon \end{pmatrix}, \quad b = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \tag{2.17}$$

where $0 < \varepsilon \ll 1$. The normal equations $A^TAv = A^Tb$ take the form

$$\begin{pmatrix} 1+\varepsilon^2 & 1 \\ 1 & 1+\varepsilon^2 \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \end{pmatrix} \tag{2.18}$$

with exact solution $v_1 = v_2 = (2+\varepsilon^2)^{-1}$. If we have $\varepsilon = 10^{-10}$, then doing the calculations with floating point arithmetic with base $B = 10$ and $n = 16$ significant digits leads to a singular matrix A^TA (with all entries equal to one) so that the normal equations have no unique solution. ◊

In cases where the columns of the matrix A are almost linearly dependent, such as in the above example, one should use a more stable algorithm to solve the least-squares problem. In the following we discuss a stable algorithm, which is based on the so-called QR-decomposition of A.

Proposition 2.15 (QR-decomposition)**.** *Any matrix $A \in \mathbb{R}^{m\times n}$ with $m \ge n$ can be written as*

$$A = QR = Q \begin{pmatrix} R_1 \\ 0 \end{pmatrix}, \tag{2.19}$$

where $Q \in \mathbb{R}^{m\times m}$ is orthogonal and $R_1 \in \mathbb{R}^{n\times n}$ is upper triangular.

We will postpone the proof of this proposition until later, and assume that this decomposition is available. Left-multiplication of the residual vector $Av - b$ by Q^T yields

$$Q^T(Av - b) = \begin{pmatrix} R_1 \\ 0 \end{pmatrix} v - Q^T b = \begin{pmatrix} R_1 v \\ 0 \end{pmatrix} - \begin{pmatrix} c_1 \\ c_2 \end{pmatrix},$$

where $(c_1, c_2)^T$ is the partitioning of $c = Q^T b \in \mathbb{R}^m$ into $c_1 \in \mathbb{R}^n$ and $c_2 \in \mathbb{R}^{m-n}$. Since multiplication with the orthogonal matrix Q^T leaves the Euclidean norm of any vector in $\mathbb{R}^m$ invariant, we have

$$\|Av - b\|^2 = \|Q^T(Av - b)\|^2 = \|R_1 v - c_1\|^2 + \|c_2\|^2.$$

It follows (please check why!) that the least-squares solution v is the solution of the upper-triangular system

$$R_1 v = c_1 \tag{2.20}$$

with corresponding (minimum) residual norm $\|Av - b\| = \|c_2\|$. This elegant solution of the least-squares problem critically depends on the QR-decomposition of A mentioned in Proposition 2.15.

Householder transformations. Before we give the proof of Proposition 2.15 we will have a small intermezzo where we introduce so-called *Householder* transformations. These transformations are defined as orthogonal reflections in a hyperplane through the origin in $\mathbb{R}^m$. If we denote the unit normal vector of the hyperplane by $u \in \mathbb{R}^m$, then one easily verifies that the corresponding Householder matrix H is given by

$$H = I - 2uu^T \quad \text{with } u^T u = 1. \tag{2.21}$$

One also easily verifies that H is symmetric and orthogonal.
In actual computations it is convenient to use the slightly more general form

$$H = I - \beta w w^T \quad \text{with } \beta = \frac{2}{w^T w}, \tag{2.22}$$

where w is also a normal vector of the hyperplane, but need not have unit length.
The following lemma shows that we can always find a suitable hyperplane through the origin such that the corresponding Householder transformation maps a given nonzero vector $x \in \mathbb{R}^m$ to a scalar multiple of the first standard basis vector e_1.

Lemma 2.16. *Let a nonzero vector $x = (x_i) \in \mathbb{R}^m$ be given. Then there exists a Householder matrix (2.22) such that $Hx = \alpha e_1$, where $\alpha \neq 0$ is real and $e_1 = (1, 0, \ldots, 0)^T \in \mathbb{R}^m$. The constant α and the matrix $H = I - \beta w w^T$ can be chosen as*

$$\alpha = -\operatorname{sgn}(x_1)\|x\| = \begin{cases} -\|x\| & \text{if } x_1 \geq 0, \\ \|x\| & \text{if } x_1 < 0, \end{cases} \tag{2.23a}$$

$$w = x - \alpha e_1 = \begin{pmatrix} x_1 - \alpha \\ x_2 \\ \vdots \\ x_m \end{pmatrix}, \tag{2.23b}$$

$$\beta = \frac{2}{w^T w} = \frac{1}{\|x\|(\|x\| + |x_1|)}. \tag{2.23c}$$

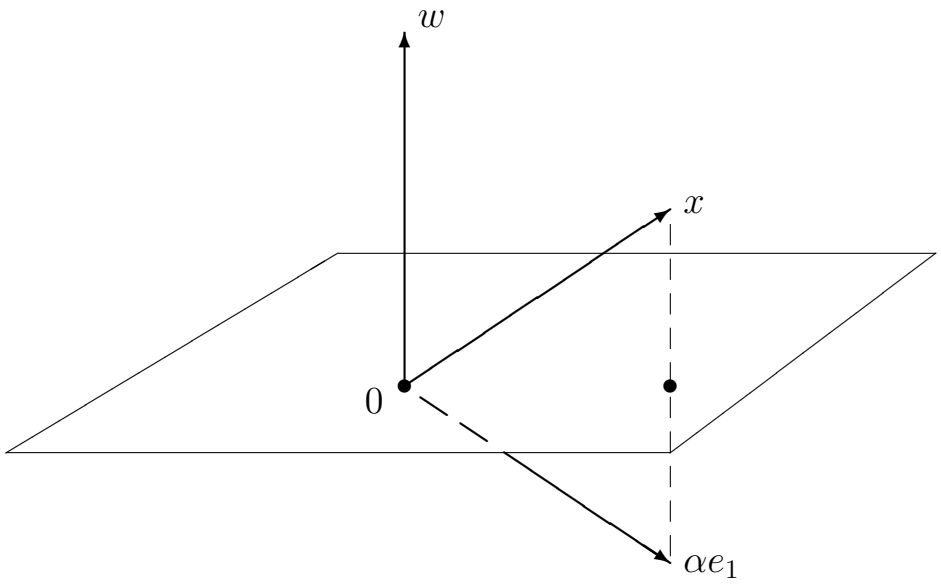


Figure 2.1: The Householder reflection of Lemma 2.16 maps x to αe_1.

Proof. First note that a Householder matrix is orthogonal, so the condition $Hx = \alpha e_1$ implies $\|x\| = \|Hx\| = \|\alpha e_1\| = |\alpha|$, and therefore $\alpha = \pm\|x\|$. Hence we have two choices for α. Further, since $Hx = \alpha e_1$ is equal to the (orthogonal) reflection of x in a hyperplane through the origin, the vector $w = x - \alpha e_1$ must be a normal vector of that hyperplane. The corresponding matrix H is given by (2.22). In order to avoid cancellation in the computation of the first component of $w = x - \alpha e_1$ (and to exclude the possibility that $w = 0$) we choose the sign of α opposite to that of x_1, which leads to (2.23a). Expression (2.23c) for β follows from

$$w^T w = (x - \alpha e_1)^T (x - \alpha e_1) = x^T x - 2\alpha x_1 + \alpha^2 = 2\|x\|^2 + 2|x_1|\|x\|.$$

One easily verifies that the constructed Householder matrix H defined by (2.22)–(2.23) indeed satisfies $Hx = \alpha e_1$. □

Remark 2.17. In actual computations (often with large m) the Householder matrix H is always stored in computer memory in the form (2.22). In other words, instead of storing all entries h_{ij} of H (which would require m^2 memory positions) only the scalar β and vector w are stored (requiring only $m + 1$ memory positions).
Also the calculation of Hv for any given vector $v \in \mathbb{R}^m$ can be done more efficiently by making use of the form (2.22),

$$Hv = (I - \beta w w^T)v = v - \beta(w^T v)w,$$

which only costs $4m$ flops. This is much less than the $m(2m-1)$ flops that would be needed for multiplying the full matrix $H = (h_{ij})$ with the given vector v. ◊

The proof of Proposition 2.15. The proof of Proposition 2.15 consists of the construction

of n Householder transformations $H_1, H_2, \ldots, H_n$ such that

$$H_n H_{n-1} \ldots H_1 A = R = \begin{pmatrix} R_1 \\ 0 \end{pmatrix}, \tag{2.24}$$

which shows that the QR-decomposition of A holds with $Q^T = H_n H_{n-1} \ldots H_1$.
The first step in establishing (2.24) is to choose the $m \times m$ Householder matrix $H_1 = I - \beta_1 w_1 w_1^T$ such that it maps the first column of $A \in \mathbb{R}^{m \times n}$ to a multiple of the first standard basis vector $e_1 = (1, 0, \ldots, 0)^T \in \mathbb{R}^m$. (Note that this is possible in view of Lemma 2.16.) Hence we have

$$H_1 A = \begin{pmatrix} \alpha_1 & \times & \cdots & \times \\ 0 & & & \\ \vdots & & A_2 & \\ 0 & & & \end{pmatrix}.$$

In the second step we apply the same procedure to the submatrix $A_2 \in \mathbb{R}^{(m-1)\times(n-1)}$, which means that we choose the $(m-1) \times (m-1)$ Householder matrix $\widehat{H}_2$ such that it maps the first column of A_2 to a multiple of $\widehat{e}_1 = (1, 0, \ldots, 0)^T \in \mathbb{R}^{m-1}$. Defining the matrix

$$H_2 = \begin{pmatrix} 1 & 0 & \cdots & 0 \\ 0 & & & \\ \vdots & & \widehat{H}_2 & \\ 0 & & & \end{pmatrix}$$

we obtain

$$H_2 H_1 A = \begin{pmatrix} \alpha_1 & \times & \times & \cdots & \times \\ 0 & \alpha_2 & \times & \cdots & \times \\ 0 & 0 & & & \\ \vdots & \vdots & & A_3 & \\ 0 & 0 & & & \end{pmatrix}.$$

Note that H_2 is also a Householder matrix because if we write $\widehat{H}_2 = \widehat{I} - \beta_2 \widehat{w}_2 \widehat{w}_2^T$ with $\widehat{I}$ the $(m-1) \times (m-1)$ identity matrix and $\widehat{w}_2 \in \mathbb{R}^{m-1}$, then $H_2 = I - \beta_2 w_2 w_2^T$ with $w_2 = (0, \widehat{w}_2^T)^T \in \mathbb{R}^m$. Proceeding in this way, we obtain the triangular form (2.24) after n steps. Note that in case $m = n$ we only need $n - 1$ steps, so we can omit H_n in that case. This completes the proof of Proposition 2.15.

Remark 2.18.* We note that the construction of the QR-decomposition of A via Householder transformations is known to be stable. It is also possible to apply the well-known *Gram-Schmidt orthogonalization process* to the columns of A, which will produce the upper triangular matrix R_1 and the first n columns of the orthogonal matrix Q of the QR-decomposition (2.19) of A. In the usual classical version of this method, however, there typically is a severe loss of orthogonality among the computed columns of Q. This shortcoming can be repaired by using a modified version of the Gram-Schmidt process, which is cheaper than but not quite as stable as the construction via Householder transformations. See [5] for more details. ♢

Example 2.19. Let us revisit Example 2.14, where the columns of the matrix A of the overdetermined system were almost linearly dependent causing the matrix A^TA of the normal equations to become singular in floating point arithmetic (with $B = 10$, $n = 16$) if $\varepsilon = 10^{-10}$. The matrix A and right-hand side b of this overdetermined system $Av = b$ were given by

$$A = \begin{pmatrix} 1 & 1 \\ \varepsilon & 0 \\ 0 & \varepsilon \end{pmatrix}, \quad b = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}.$$

We apply the method of Householder transformations (2.24) to reduce the overdetermined system to upper-triangular form by applying the orthogonal transformation $Q^T = H_2H_1$. For the 3×3 matrix $H_1 = I - \beta_1 w_1 w_1^T$ we find from Lemma 2.16 that

$$\begin{aligned} \alpha_1 &= -\|(1,\varepsilon,0)^T\| = -\sqrt{1+\varepsilon^2} = -1 + \mathcal{O}(\varepsilon^2) \approx -1, \\ w_1 &= \begin{pmatrix} 1 \\ \varepsilon \\ 0 \end{pmatrix} - \begin{pmatrix} \alpha_1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 1+\sqrt{1+\varepsilon^2} \\ \varepsilon \\ 0 \end{pmatrix} = \begin{pmatrix} 2+\mathcal{O}(\varepsilon^2) \\ \varepsilon \\ 0 \end{pmatrix} \approx \begin{pmatrix} 2 \\ \varepsilon \\ 0 \end{pmatrix}, \\ \beta_1 &= \left(1+\varepsilon^2+\sqrt{1+\varepsilon^2}\right)^{-1} = \tfrac{1}{2} + \mathcal{O}(\varepsilon^2) \approx \tfrac{1}{2}, \end{aligned}$$

where we have dropped the $\mathcal{O}(\varepsilon^2)$ terms since they will have no influence on the floating point calculations (with $B = 10$ and $n = 16$) if $\varepsilon = 10^{-10}$. Left-multiplying the overdetermined system $Av = b$ by H_1 gives the equivalent overdetermined system $H_1Av = H_1b$ with

$$H_1A = \begin{pmatrix} -1 & -1 \\ 0 & -\varepsilon \\ 0 & \varepsilon \end{pmatrix}, \quad H_1b = \begin{pmatrix} -1 \\ -\varepsilon \\ 0 \end{pmatrix},$$

where we have again suppressed all $\mathcal{O}(\varepsilon^2)$ terms. Continuing with H_2 gives the equivalent overdetermined system $Rv = c$ with

$$R = H_2H_1A = \begin{pmatrix} -1 & -1 \\ 0 & \varepsilon\sqrt{2} \\ 0 & 0 \end{pmatrix}, \quad c = H_2H_1b = \begin{pmatrix} -1 \\ \varepsilon/\sqrt{2} \\ -\varepsilon/\sqrt{2} \end{pmatrix}.$$

The least-squares solution can readily be solved from the triangular system (2.20), giving an excellent approximation of the exact least-squares solution $v_1 = v_2 = (2+\varepsilon^2)^{-1}$. ◊

Remark 2.20. The method of Householder transformations is not only useful for overdetermined systems (where $m > n$), but can also be used for the case $m = n$. Let us compare for $m = n$ the Householder method with Gaussian elimination. Just like we did in (2.8) for Gaussian elimination, we can describe the Householder method by a series of transformations

$$[A,b] = [A^{(1)},b^{(1)}] \;\to\; [A^{(2)},b^{(2)}] \;\to\; \cdots \;\to\; [A^{(m)},b^{(m)}] = [R,c],$$

but the transformations are now given by

$$A^{(k+1)} = H_k A^{(k)}, \quad b^{(k+1)} = H_k b^{(k)},$$

which differs from the Gaussian transformations (2.9) where we left-multiplied with the lower triangular matrices L_k. Although Gaussian elimination (with partial pivoting) is generally stable in practice (see the discussion at the end of Section 2.2), one can construct exceptional cases where it is not. In contrast, the method of Householder transformations (for the case $m = n$) can be shown to be always stable. This is best understood if we realize that the condition numbers (with respect to the Euclidean norm) of the intermediate matrices $A^{(k)}$ are all equal to that of A,

$$\mathrm{cond}_2(A^{(k)}) = \mathrm{cond}_2(A).$$

This fact, which follows from the orthogonality of the matrices H_k and can easily be proved by using Exercise A.5, shows that any rounding errors occurring in the intermediate system $[A^{(k)}, b^{(k)}]$ will have a similar effect on the final solution as rounding errors already present in the initial system $[A, b]$, see Theorem 2.2. This nice property, which is not shared by Gaussian elimination, is the theoretical justification for using orthogonal transformations. The price to pay is that the method of Householder transformations costs approximately $\frac{4}{3}m^3$ flops, which is about twice the amount needed for Gaussian elimination. ◊

2.5 Further reading *

What we have seen in this chapter are so-called *direct methods* to solve linear systems. There, we have looked for the "exact" solution to the system, up to matrix precision. Another set of methods are *iterative methods*, where one starts from an initial guess for the solution and computes iterates that, hopefully, approach the exact solution in the limit. These methods can be more efficient for large systems. Moreover, they do not need the explicit construction of the system matrix, but only a routine that computes matrix-vector multiplication for the system's matrix (so to say, they only need the *action* of the matrix on a vector), which can also be convenient for large scale applications. One of this methods is *gradient descent*, from which the stochastic gradient descent to train neural networks was derived. For square systems with symmetric positive definite matrices, *conjugate gradient*, which is a modified version of gradient descent, is usually the method of choice. If you are interested in iterative methods, you can consult, for example, Chapter 4 in [11].

We have also seen that system matrices with large condition numbers are undesirable, because they lead to more severe error propagation. In iterative methods, the condition number of the matrix influences how many iterations are needed to achieve a given tolerance in the solution. A tool that is commonly used to reduce the condition number of a matrix are *preconditioners*. Very roughly speaking, a preconditioner is a non-singular matrix $\mathcal{P}$ (not to be confused with the permutation matrix) which is relatively cheap to build but that approximates the inverse of the system matrix in some way. Then, instead of solving $Av = b$, we can solve equivalently $\mathcal{P}Av = \mathcal{P}b$. If $\mathcal{P}$ is a good approximation to A, the condition number of $\mathcal{P}A$ will be quite close to 1 (the condition number of the identity). Preconditioners are also mentioned in Chapter 4 in [11].

Chapter 3

Scalar Nonlinear Equations

A basic numerical problem is: find an x satisfying the nonlinear equation $f(x) = 0$, where f is a given function. Such problems arise very frequently. Sometimes the function f is given in analytic form, but there are also many applications where that is not the case. In fact, evaluation of function values $f(x)$ may require itself some numerical procedure.

Example 3.1. Let

$$f(x) = \int_0^x \varphi(s)\, ds - \mu\,,$$

with given $\mu > 0$ and φ a continuous real function. If this function φ is complicated, we will not have an explicit expression for the integral, and a numerical procedure may be needed to compute the value of $f(x)$ for given x. On the other hand, certain properties of f may be known. For instance, if $0 < \alpha \leq \varphi(s) \leq \beta$ (for all $s \geq 0$), then $f(x)$ is monotonically increasing and $\alpha x - \mu \leq f(x) \leq \beta x - \mu$ for $x \geq 0$. Therefore we know that the equation $f(x) = 0$ has a unique solution x_* in $[\mu/\beta, \mu/\alpha]$. The question is now: how can this solution x_* be computed with some prescribed accuracy? $\Diamond$

In this chapter we will discuss several numerical methods for finding approximations to the solution of $f(x) = 0$, or the related fixed point equation $g(x) = x$. We limit ourselves in this chapter to methods for scalar equations with one real variable. Extensions to systems of equations (with multiple unknowns) are considered in the next chapter.

3.1 The Condition of the Root Finding Problem

In the previous sections we have studied numerical methods for finding roots of nonlinear equations. These equations generally depend on one or more input parameters, which contain small errors such as data measurement errors or rounding errors. In this section we study the sensitivity of the roots with respect to small variations in the input parameters.
We will confine ourselves to investigating the sensitivity of a particular root with respect to a *single* input parameter $\lambda \in \mathbb{R}$, so that the problem consists of finding the root x of the equation

$$f(x, \lambda) = 0.$$

If there are more input parameters, the effect of each parameter can be studied separately. We assume that the root is simple, which is equivalent to $\partial f/\partial x \neq 0$. According to the implicit function theorem, the root x is a differentiable function of λ, so we can write $x = x(\lambda)$, which gives

$$f(x(\lambda), \lambda) = 0.$$

Differentiation with respect to λ yields

$$\frac{\partial f}{\partial x} x'(\lambda) + \frac{\partial f}{\partial \lambda} = 0,$$

so that

$$x'(\lambda) = -\frac{\partial f/\partial \lambda}{\partial f/\partial x}.$$

We conclude that the perturbation Δx in x caused by a perturbation $\Delta\lambda$ in λ is approximately

$$\Delta x \approx x'(\lambda)\Delta\lambda = -\frac{\partial f/\partial \lambda}{\partial f/\partial x}\Delta\lambda.$$

If $x \neq 0$ and $\lambda \neq 0$, recalling the definition of the relative condition number (1.9) (where now λ is the input and $x = x(\lambda)$ the output), we have

$$\gamma \approx \left|\frac{\partial f/\partial \lambda}{\partial f/\partial x}\right| \frac{|\lambda|}{|x|}. \tag{3.1}$$

This expression for the condition number shows that the root finding problem, assuming a simple root, is ill-conditioned when the function is almost flat around the root.
Note that, if for example $f(x) = \varphi(x) - d$ for a smooth function φ and $d \in \mathbb{R}$, that is, if we want to find x such that $\varphi(x) = d$, and if we consider d as parameter, then the expression (3.1) simplifies to $\gamma(d) \approx \left|\frac{1}{\partial f/\partial x}\right| \frac{|d|}{|x|}$.

3.2 Bisection

Suppose $f : \mathbb{R} \to \mathbb{R}$ is continuous on the interval $[a, b]$. If $f(a)$ and $f(b)$ have opposite signs we know from the intermediate value theorem (Theorem B.1) that there is an $x_* \in (a, b)$ such that $f(x_*) = 0$. Such x_* is called a *zero* of f or a *root* of the equation. Let us assume for the moment that x_* is the only root in $[a, b]$.
One of the simplest methods from numerical mathematics is *bisection*, where the interval is repeatedly halved, such that each new interval still contains a root. First we initialize $a_0 = a$, $b_0 = b$, and then for $k = 0, 1, 2, \ldots$ we take

$$\left\{\begin{array}{l} x_k = \frac{1}{2}(a_k + b_k)\,; \\ \text{if } f(x_k) = 0 \text{ we are done; otherwise} \\ \quad \text{if } \ \operatorname{sign}(f(x_k)) = \operatorname{sign}(f(a_k))\,, \text{ set } \ a_{k+1} = x_k\,, \ \ b_{k+1} = b_k\,, \\ \quad \text{if } \ \operatorname{sign}(f(x_k)) = \operatorname{sign}(f(b_k))\,, \text{ set } \ a_{k+1} = a_k\,, \ \ b_{k+1} = x_k\,. \end{array}\right. \tag{3.2}$$

It is clear that each interval $[a_k, b_k]$ contains a zero of f, and $b_k - a_k = 2^{-k}(b-a)$, because the interval is halved each time. Consequently

$$|x_k - x_*| \leq 2^{-(k+1)}(b-a) \qquad (k \geq 0)\,. \tag{3.3}$$

If we want an approximation x_k to x_* such that its absolute error has an absolute value less than or equal to a given tolerance Tol, which is a desired accuracy, then we can terminate the algorithm as soon as $2^{-(k+1)} \leq Tol/(b-a)$.
Bisection guarantees convergence of x_k towards x_*. However, this convergence is rather slow. For example, if $b-a = 1$ and we want to approximate x_* up to 12 decimal places, then we generally need to iterate until $k = 40$ to achieve this. Another drawback of the bisection method is that it cannot be extended to systems of equations. For the other methods treated in this chapter such extensions do exist.

3.3 Fixed Point Iteration

Let us consider the fixed point problem

$$g(x) = x\,, \tag{3.4}$$

where $g : \mathbb{R} \to \mathbb{R}$ is given. Starting with an initial guess x_0, we can compute successive approximations $x_1, x_2, \ldots$ by the iteration

$$x_{k+1} = g(x_k) \qquad (k = 0, 1, \ldots)\,. \tag{3.5}$$

This is called *fixed point iteration*, or *functional iteration*.

We will assume that the iteration function g is continuously differentiable on an interval $[a, b]$. As the following lemma shows, this guarantees that g satisfies a *Lipschitz condition* on that interval,

$$|g(\widetilde{x}) - g(x)| \leq L|\widetilde{x} - x| \quad \text{for all } x, \widetilde{x} \in [a, b]. \tag{3.6}$$

Lemma 3.2. *Let $g : [a, b] \to \mathbb{R}$ be continuously differentiable on the interval $[a, b]$. Then it satisfies a Lipschitz condition on $[a, b]$ with Lipschitz constant*

$$L = \max_{a \leq x \leq b} |g'(x)|.$$

Proof. According to the mean value theorem[1], for any pair $x, \widetilde{x} \in [a, b]$ there is a point ξ between x and $\widetilde{x}$ such that $g(\widetilde{x}) - g(x) = g'(\xi)(\widetilde{x} - x)$. □

We are interested in cases where the Lipschitz constant satisfies $L < 1$, and in that case we say that g is a contraction mapping (or is *contractive*) on $[a, b]$. The following theorem is a simple (scalar) version of the well-known *Banach fixed point theorem* (or *contraction mapping theorem*).

[1] see Theorem B.2

Theorem 3.3. *Let $g : [a, b] \to \mathbb{R}$ be continuously differentiable, and suppose*

(i) g maps the interval $[a, b]$ into itself, that is, $g([a, b]) \subset [a, b]$;

(ii) there is a number $\theta < 1$ such that $|g'(x)| \leq \theta$ for all $x \in [a, b]$.

Then there is a unique fixed point x_* of g in $[a, b]$. Moreover, for any $x_0 \in [a, b]$ the fixed point iteration (3.5) converges to x_*, and

$$|x_k - x_*| \leq \theta\, |x_{k-1} - x_*| \qquad (k \geq 1)\,. \tag{3.7}$$

Proof. Let $f(x) = g(x) - x$. This function is continuous with $f(a) \geq 0$, $f(b) \leq 0$, because (i) implies $g(a) \geq a$ and $g(b) \leq b$. According to the intermediate value theorem[2] there is a zero of f in $[a, b]$, and this is a fixed point of g.
Using Lemma 3.2 it follows from (ii) that g satisfies a Lipschitz condition (3.6) with constant $L = \theta < 1$. Hence g is contractive on $[a, b]$, which implies that the fixed point is unique (why?). Using Lipschitz condition (3.6) once more we also find

$$|x_k - x_*| = |g(x_{k-1}) - g(x_*)| \leq \theta |x_{k-1} - x_*|,$$

showing the validity of estimate (3.7) and convergence of x_k to x_*. □

The following theorem gives, for a given fixed point, a sufficient condition for local convergence.

Theorem 3.4. *Let $g : (a, b) \to \mathbb{R}$ be continuously differentiable, and suppose that $x_* \in (a, b)$ is a fixed point of g with $|g'(x_*)| < 1$. Then, for $\delta > 0$ small enough, g maps the interval $I_\delta = [x_* - \delta, x_* + \delta]$ into itself, and fixed point iteration converges to x_* for every starting point $x_0 \in I_\delta$.*

Proof. Since $|g'(x_*)| < 1$, we can find a value θ with $|g'(x_*)| < \theta < 1$. From the continuity of g' it follows that for $\delta > 0$ small enough we have

$$|g'(x)| \leq \theta \quad \text{for all } x \in I_\delta.$$

According to Lemma 3.2 this implies that g is contractive on I_δ with Lipschitz constant $L = \theta < 1$. For $x \in I_\delta$ we therefore have

$$|g(x) - x_*| = |g(x) - g(x_*)| \leq \theta |x - x_*| \leq \theta\delta < \delta,$$

showing that g maps the interval I_δ into itself. The result now follows by applying Theorem 3.3 (with the interval $[a, b]$ replaced by I_δ). □

Note that the above local convergence theorem (Theorem 3.4) presupposes the existence of a fixed point. This is not done in in the contraction mapping theorem (Theorem 3.3), where existence and uniqueness of a fixed point are proved.

[2] see Theorem B.1

The error bound (3.7) guarantees that in each step the error $|x_k - x_*|$ is (at least) reduced by the factor $\theta < 1$. This is what is called *linear convergence* and directly leads to the error bound

$$|x_k - x_*| \;\le\; \theta^k \, |x_0 - x_*| \qquad (k \ge 1)\,. \tag{3.8}$$

This error bound can be computed prior to the start of the iterative process, and is therefore called an *a-priori* bound. During the course of the iteration, a more realistic error bound can be obtained by noting that (3.7) also implies

$$|x_k - x_*| \;\le\; \theta |x_{k-1} - x_*| \;\le\; \theta \left(|x_{k-1} - x_k| + |x_k - x_*| \right),$$

which yields

$$|x_k - x_*| \;\le\; \frac{\theta}{1-\theta} \, |x_k - x_{k-1}| \qquad (k \ge 1)\,. \tag{3.9}$$

Since this error bound can only be evaluated after computation of x_{k-1} and x_k, it is called an *a-posteriori* bound. The practical value of this bound is that it can serve as the basis for a stopping criterion for the iterative process.

3.4 Newton's Method

Suppose $f : \mathbb{R} \to \mathbb{R}$ is differentiable, and we want to find a root x_* of the equation

$$f(x) = 0\,. \tag{3.10}$$

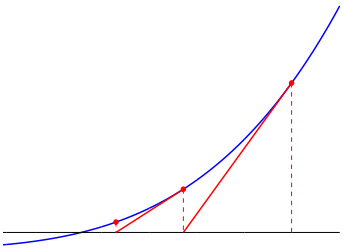
Given an approximation x_k, we can locally linearize the function f by using its 'tangent line approximation',

$$f(x) \approx f(x_k) + f'(x_k)(x - x_k), \tag{3.11}$$

and then find a new approximation x_{k+1} which is a zero of the linearization. This procedure leads to the iteration

$$x_{k+1} = x_k - \frac{1}{f'(x_k)} f(x_k) \qquad (k = 0, 1, 2, \ldots)\,. \tag{3.12}$$

This is known as *Newton's method* or *Newton-Raphson iteration*. In general, the convergence of this iteration is much faster than for the methods of the previous sections. However, convergence is only guaranteed if the initial value x_0 is sufficiently close to x_*.



The following result states the convergence of Newton's method (note that the symbol γ used there has nothing to do with the condition number).

Theorem 3.5. *Let $f : (a, b) \to \mathbb{R}$ be twice continuously differentiable, and assume that $x_* \in (a, b)$ satisfies $f(x_*) = 0$ and $f'(x_*) \neq 0$. Then there exist $\delta > 0$, $\gamma \ge 0$ such that for all x_0 with $|x_0 - x_*| \le \delta$ the Newton iteration converges to x_* and*

$$|x_k - x_*| \;\le\; \gamma \, |x_{k-1} - x_*|^2 \qquad (k \ge 1)\,. \tag{3.13}$$

Proof. For x close to x_* we have $f'(x) \neq 0$. Let $g(x) = x - \big(f'(x)\big)^{-1} f(x)$. Then x_* is a fixed point of g, and we have $g'(x) = \big(f'(x)\big)^{-2} f''(x) f(x)$. So, in particular $g'(x_*) = 0$. It follows from Theorem 3.4 that for $\delta > 0$ sufficiently small, the function g maps the interval $I_\delta = [x_* - \delta, x_* + \delta]$ into itself, and fixed point iteration for g, which is the same as Newton iteration for f, converges to x_* if our initial value x_0 lies in I_δ. Further we have

$$0 = f(x_k) + f'(x_k)(x_{k+1} - x_k)\,,$$

$$0 = f(x_*) = f(x_k) + f'(x_k)(x_* - x_k) + \tfrac{1}{2} f''(\xi_k)(x_* - x_k)^2\,,$$

with ξ_k between x_* and x_k (Taylor's theorem, see Appendix B.3). By subtraction it is seen that

$$x_{k+1} - x_* = \frac{1}{2}\frac{f''(\xi_k)}{f'(x_k)}(x_* - x_k)^2\,,$$

from which the proof follows with

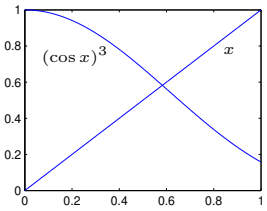
$$\gamma = \frac{1}{2}\max_{\xi,\eta \in I_\delta} \frac{|f''(\xi)|}{|f'(\eta)|} = \frac{1}{2}\frac{\max_{\xi\in I_\delta}|f''(\xi)|}{\min_{\eta\in I_\delta}|f'(\eta)|}. \qquad \square$$

Remark 3.6. The error bound (3.13) shows that Newton's method is (at least) *quadratically convergent*. This is significantly faster than linear convergence (3.7) where in each step the error is only reduced by a factor $\theta < 1$. ♢

Illustration. Consider the equation $f(x) = 0$ with

$$f(x) = x - (\cos x)^3\,.$$

By drawing the graphs of x and $(\cos x)^3$ it is clear that the equation will have a unique solution in the interval $[0, 1]$.



First we try to solve the equation with Newton's method. Using the starting value $x_0 = 0$ the iterates $x_1, x_2, \ldots, x_6$ are found to be:

x_0	x_1	x_2	x_3	x_4	x_5	x_6
0.000000	1.000000	0.515084	0.583029	0.582440	0.582440	0.582440

In this table, only the first six decimal digits are presented, and for the digits that are not yet correct a small font is used. After a somewhat hesitant start, with initial overshoot, the Newton iterates converge very rapidly to the correct value x_* which is computed with fourteen digits accuracy as $x_* = 0.58244007115820$.

For this problem we can also try fixed point iteration. A natural choice for the iteration function is $g(x) = (\cos x)^3$. However, with this choice we have $g'(x) = -3 \sin x\,(\cos x)^2$, which happens to be larger than one in modulus near x_*. Indeed, the iteration does not converge:

x_0	x_1	x_2	x_3	x_4	x_5	x_6
0.000000	1.000000	0.157728	0.963220	0.186051	0.949115	0.197546

Convergence can be achieved, however, with an other fixed point iteration function. Consider $g(x) = x - c\,f(x)$ with $c \in \mathbb{R}$ still to be determined. It is clear that for any $c \neq 0$ the fixed point of g will be the zero of f. To choose a suitable value of c, note that $f'(x) = 1 + 3\sin x\,(\cos x)^2 \in [1, 4]$ if $x \in [0, 1]$. Therefore $g'(x) = 1 - c\,f'(x)$ will assume values between $1 - c$ and $1 - 4c$. For $c = \frac{2}{5}$ we thus have $|g'(x)| \leq \frac{3}{5}$. With this choice the iteration converges:

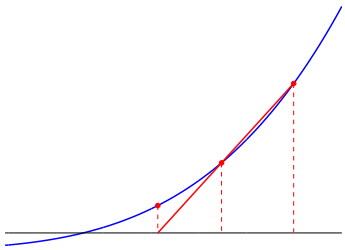
x_0	x_1	x_2	x_3	x_4	x_5	x_6
0.000000	0.400000	0.552554	0.578212	0.581848	0.582357	0.582428

In fact the rate of convergence is faster than might be expected. That is caused by the fact that $|g'(x_*)|$ turns out to be approximately 0.14. The value $\frac{3}{5}$ found above is a rather crude over-estimation.

Remark 3.7. * There are many variants of Newton's method. They can all be obtained by approximating the derivative $f'(x_k)$. For example, replacing $f'(x_k)$ by the difference quotient $(f(x_k) - f(x_{k-1}))/(x_k - x_{k-1})$ leads to the *secant method*

$$x_{k+1} = x_k - \frac{x_k - x_{k-1}}{f(x_k) - f(x_{k-1})}\, f(x_k)\,. \tag{3.14}$$

Here x_{k+1} depends on the two previous values x_k and x_{k-1}, so we now need two starting values x_0 and x_1. Computation of the derivative is avoided here. This can be an advantage, for example if f is not given in closed form, but only a recipe is provided to compute $f(x)$ for given x by some (numerical) procedure.



The convergence of this secant method is in general slower than for Newton's method. If x_{k-1} and x_k are close to x_* it can be shown under the conditions of Theorem 3.5 that

$$|x_{k+1} - x_*| \;\leq\; \widetilde{\gamma}\,|x_k - x_*||x_{k-1} - x_*|\,,$$

with a positive constant $\widetilde{\gamma}$. If $|x_k - x_*|$ is much smaller than $|x_{k-1} - x_*|$ this is obviously not as good as the quadratic bound in (3.13). In fact, under suitable technical assumptions one can prove an error bound $|x_{k+1} - x_*| \leq C|x_k - x_*|^p$ with order $p = \frac{1}{2}(1 + \sqrt{5}) \approx 1.62$. ◊

We now give a more general criterion to check the order of convergence of a method based on fixed point iteration.

Proposition 3.8. *Let g be the function defining a fixed point iteration. If $g \in C^{p+1}(\bar{I})$ for an open neighborhood I of the fixed point x_* and an integer $p \geq 1$, and if $g^{(i)}(x_*) = 0$ for $1 \leq i \leq p$ and $g^{(p+1)}(x_*) \neq 0$, then the fixed-point method with iteration function g has order $p + 1$. Moreover,*

$$\lim_{k\to\infty} \frac{x_{k+1} - x_*}{(x_k - x_*)^{(p+1)}} = \frac{g^{(p+1)}(\eta)}{(p+1)!}. \tag{3.15}$$

Proof. The smoothness assumption on g allows us to expand $x_{k+1} = g(x_k)$ in Taylor series centered at the fixed point $x_* = g(x_*)$:

$$x_{k+1} - x_* = \sum_{i=0}^{p} \frac{g^{(i)}(x_*)}{i!}(x_k - x_*)^i + \frac{g^{(p+1)}(\eta)}{(p+1)!}(x_k - x_*)^{(p+1)} - g(x_*),$$

for η a point between x_k and x_*. It follows that

$$\lim_{k\to\infty} \frac{x_{k+1} - x_*}{(x_k - x_*)^{(p+1)}} = \lim_{k\to\infty} \frac{g^{(p+1)}(\eta)}{(p+1)!}(x_k - x_*)^{(p+1)} = \frac{g^{(p+1)}(\eta)}{(p+1)!}.$$

□

Note that (3.15) gives the limit, as $k \to \infty$, of the constant in the convergence rate of the method. If x_* is a simple root, namely $f(x_*) \neq 0$, it is easy to check that, for Newton's method, $g'(x_*) = 0$, with $g(x) = x - \frac{f(x)}{f'(x)}$, but $g''(x_*) \neq 0$, from which second order accuracy follows.

Chapter 4

Systems of Nonlinear Equations

Fixed point iteration and Newton's method can also be used to solve *systems* of nonlinear equations, where we work with vectors $v \in \mathbb{R}^m$ instead of scalars $x \in \mathbb{R}$. For $F : \mathbb{R}^m \to \mathbb{R}^m$ we will use the notation

$$F(v) = \begin{pmatrix} F_1(v) \\ \vdots \\ F_m(v) \end{pmatrix} \qquad \text{for} \quad v = \begin{pmatrix} v_1 \\ \vdots \\ v_m \end{pmatrix} \in \mathbb{R}^m.$$

The $m \times m$ Jacobian matrix, containing all partial derivatives $\frac{\partial F_i(v)}{\partial v_j}$, is denoted as

$$JF(v) = \begin{pmatrix} \frac{\partial F_1(v)}{\partial v_1} & \cdots & \frac{\partial F_1(v)}{\partial v_m} \\ \vdots & & \vdots \\ \frac{\partial F_m(v)}{\partial v_1} & \cdots & \frac{\partial F_m(v)}{\partial v_m} \end{pmatrix}.$$

This will also be written more compactly as $v = \big(v_j\big)$, $F(v) = \big(F_j(v)\big) \in \mathbb{R}^m$ and $JF(v) = \big(\partial F_i(v)/\partial v_j\big) \in \mathbb{R}^{m\times m}$. Thus, subindices are used for the components. Of course, a vector itself may have a subindex already. For a sequence of vectors $u_0, u_1, u_2, \ldots$, the j-th component of $u_k \in \mathbb{R}^m$ can be denoted as $(u_k)_j$. It should always be clear from the context whether v_j is a vector itself or a component of a vector v.

4.1 Fixed Point Iteration

The fixed point iteration to find an approximate solution for the system $G(u) = u$ reads

$$u_{k+1} = G(u_k). \tag{4.1}$$

The result of Theorem 3.3 can be generalized to mappings $G : \mathcal{D} \to \mathcal{D}$, where $\mathcal{D}$ is a nonempty, closed subset of $\mathbb{R}^m$. We assume that there exists a norm $\|\cdot\|$ on $\mathbb{R}^m$ such that G is a *contraction mapping* on $\mathcal{D}$, that is, it satisfies a Lipschitz condition on $\mathcal{D}$ with Lipschitz constant $\theta < 1$,

$$\|G(\widetilde{v}) - G(v)\| \le \theta\|\widetilde{v} - v\| \quad \text{for all } v, \widetilde{v} \in \mathcal{D}. \tag{4.2}$$

The following result—and its generalization to Banach spaces[1]—is known as the *contraction mapping theorem* or *Banach fixed point theorem* (here with $\mathbb{R}^m$ as Banach space).

Theorem 4.1. *Let $G : \mathcal{D} \to \mathcal{D}$ be a contraction mapping on a nonempty, closed set $\mathcal{D} \subset \mathbb{R}^m$. Then G has a unique fixed point $u_* \in \mathcal{D}$. Moreover, for any $u_0 \in \mathcal{D}$ the fixed point iteration (4.1) converges to u_*, and*

$$\|u_k - u_*\| \le \theta \, \|u_{k-1} - u_*\| \qquad (k \ge 1)\,. \tag{4.3}$$

Proof. The proof consists of four steps.
1. First we show for any starting point $u_0 \in \mathcal{D}$ that fixed point iteration produces a sequence $\{u_k\}_{k=0}^{\infty}$ converging to a limit in $\mathcal{D}$. Note that it follows from $G(\mathcal{D}) \subset \mathcal{D}$ that iteration (4.1) is well-defined. Using

$$\|u_{k+1} - u_k\| = \|G(u_k) - G(u_{k-1})\| \le \theta \|u_k - u_{k-1}\| \quad (k \ge 1)$$

we find

$$\|u_{k+1} - u_k\| \le \theta \|u_k - u_{k-1}\| \le \theta^2 \|u_{k-1} - u_{k-2}\| \le \ldots \le \theta^k \|u_1 - u_0\|.$$

This implies that for all k, ℓ with $0 \le \ell < k$ we have

$$\begin{aligned} \|u_k - u_\ell\| &\le \|u_k - u_{k-1}\| + \|u_{k-1} - u_{k-2}\| + \ldots + \|u_{\ell+1} - u_\ell\| \\ &\le \theta^k \|u_1 - u_0\| + \theta^{k-1} \|u_1 - u_0\| + \ldots + \theta^{\ell+1} \|u_1 - u_0\| \\ &= \left(\sum_{j=l+1}^{k} \theta^j \right) \|u_1 - u_0\| \\ &\le \left(\sum_{j=l+1}^{\infty} \theta^j \right) \|u_1 - u_0\| = \frac{\theta^{l+1}}{1-\theta} \|u_1 - u_0\|. \end{aligned}$$

It follows that $\lim \|u_k - u_\ell\| = 0$ (as $k, \ell \to \infty$), showing that $\{u_k\}_{k=0}^{\infty}$ is a Cauchy sequence[2] in $\mathcal{D}$. Since $(\mathbb{R}^m, \|\cdot\|)$ is a Banach space[3], the sequence has a limit $u_* \in \mathbb{R}^m$. This limit must lie in $\mathcal{D}$ because $\mathcal{D}$ is closed.
2. Next we show that u_* is a fixed point of G by taking the limit $k \to \infty$ in the relation $u_{k+1} = G(u_k)$. The left-hand side converges to u_*, while the right-hand side converges to $G(u_*)$ since $\|G(u_k) - G(u_*)\| \le \theta \|u_k - u_*\| \to 0$. Hence $u_* = G(u_*)$.
3. Next we show that u_* is the unique fixed point of G in $\mathcal{D}$. Suppose that v_* is a fixed point in $\mathcal{D}$. Then we have $\|v_* - u_*\| = \|G(v_*) - G(u_*)\| \le \theta \|v_* - u_*\|$, implying that $(1-\theta)\|v_* - u_*\| \le 0$ and therefore $v_* = u_*$.
4. Finally note that

$$\|u_k - u_*\| = \|G(u_{k-1}) - G(u_*)\| \le \theta \|u_{k-1} - u_*\|,$$

[1] For the definition of a Banach space, see Appendix B.6
[2] For the definition of a Cauchy sequence, see Appendix B.6
[3] For the definition of a Banach space, see Appendix B.6

which proves (4.3). □

Note that, when applying the theorem above, one must not check that both the self-mapping property $G(\mathcal{D}) \subset \mathcal{D}$ and the contraction property (4.2) hold. The following lemma can be useful for checking the latter.

Lemma 4.2. *Let $G : \mathcal{D} \to \mathbb{R}^m$ be continuous on $\mathcal{D}$, where $\mathcal{D} = \overline{\Omega}$ is the closure of a nonempty open convex set $\Omega \subset \mathbb{R}^m$. Assume that G is continuously differentiable on Ω with bounded partial derivatives. Then G satisfies a Lipschitz condition on $\mathcal{D}$ with Lipschitz constant*

$$L = \sup_{v \in \Omega} \|JG(v)\|, \tag{4.4}$$

where $\|JG(v)\|$ denotes the induced matrix norm of the Jacobian matrix $JG(v)$.

Proof. For $v, \widetilde{v} \in \Omega$ the inequality $\|G(\widetilde{v}) - G(v)\| \leq L\|\widetilde{v} - v\|$ immediately follows from the mean value inequality (see Appendix B.5). That the inequality also holds for $v, \widetilde{v} \in \mathcal{D} = \overline{\Omega}$ follows from the continuity of G on $\mathcal{D}$ and the fact that we can approximate v and $\widetilde{v}$ arbitrarily closely with vectors in Ω. □

By combining this lemma with Theorem 4.1 above, we obtain the following corollary, which generalizes the local convergence result of Theorem 3.4.

Corollary 4.3. *Let $G : \Omega \to \mathbb{R}^m$ be continuously differentiable on an open set $\Omega \subset \mathbb{R}^m$, and suppose that $u_* \in \Omega$ is a fixed point of G with $\|JG(u_*)\| < 1$. Then, for $\delta > 0$ small enough, G maps the closed ball $\overline{B}_\delta = \{u \in \mathbb{R}^m : \|u - u_*\| \leq \delta\}$ into itself, and fixed point iteration converges to u_* for every starting point $u_0 \in B_\delta$.*

Proof. The proof is very similar to the proof of Theorem 3.4.
Since $\|JG(u_*)\| < 1$, we can find a value θ with $\|JG(u_*)\| < \theta < 1$. From the continuous differentiability of G on Ω it follows that for $\delta > 0$ small enough we have $\|JG(u)\| \leq \theta$ for all $u \in \overline{B}_\delta$. From Lemma 4.2 (with $\mathcal{D} = \overline{B}_\delta$) we conclude that G satisfies a Lipschitz condition on $\overline{B}_\delta$ with Lipschitz constant θ. This implies also that G maps $\overline{B}_\delta$ into itself since for any $u \in \overline{B}_\delta$ one has

$$\|G(u) - u_*\| = \|G(u) - G(u_*)\| \leq \theta\|u - u_*\| \leq \theta\delta < \delta.$$

The result now follows by applying Theorem 4.1 (with $\mathcal{D} = \overline{B}_\delta$). □

Remark 4.4. Note that the condition $\|JG(u_*)\| < 1$ in Corollary 4.3 depends on the chosen vector norm on $\mathbb{R}^m$. Without proof we mention that the existence of a vector norm on $\mathbb{R}^m$ such that the induced matrix norm of $JG(u_*)$ satisfies $\|JG(u_*)\| < 1$, is equivalent to the condition that $|\lambda| < 1$ for all eigenvalues λ of $JG(u_*)$. The local convergence properties of fixed point iteration are therefore essentially determined by the eigenvalues of $JG(u_*)$. Although a rigorous proof is beyond the scope of the course, this should not be surprising if one realizes that close to u_* one has $u_k - u_* \approx JG(u_*)(u_{k-1} - u_*)$ and hence also $u_k - u_* \approx JG(u_*)^k(u_0 - u_*)$. The matrix $JG(u_*)^k$ tends to zero (as $k \to \infty$) if and only if all eigenvalues of $JG(u_*)$ lie in the open unit disk in the complex plane. ◊

4.2 Newton's Method

Let $F : \mathbb{R}^m \to \mathbb{R}^m$. We want to find $u_* \in \mathbb{R}^m$ that solves the system of equations

$$F(u) = 0\,. \tag{4.5}$$

Just as we did in Section 3.4 for the scalar case, we can linearize[4] the function F around the latest approximation u_k,

$$F(u) \approx F(u_k) + JF(u_k)(u - u_k), \tag{4.6}$$

and define the next approximation $u = u_{k+1}$ as the zero of the linearization, that is,

$$JF(u_k)(u_{k+1} - u_k) = -F(u_k). \tag{4.7}$$

Newton's method for system (4.5) can therefore be compactly written as

$$u_{k+1} = u_k - \big(JF(u_k)\big)^{-1} F(u_k) \qquad (k = 0, 1, 2, \ldots)\,. \tag{4.8}$$

This produces vectors $u_k \in \mathbb{R}^m$ that converge—hopefully—towards u_*.
We emphasize that the inverse matrix occurring in the compact formulation (4.8) should *never* be explicitly calculated in actual implementations of the method. Computing the inverse of a matrix is unnecessary and may be very expensive in terms of computing time or storage. It suffices to simply solve the linear system (4.7), which leads to the following algorithm,

$$\text{Step 1:}\quad \text{Solve } s_k \in \mathbb{R}^m \text{ from the linear system } JF(u_k)s_k = -F(u_k), \tag{4.9a}$$
$$\text{Step 2:}\quad \text{Define } u_{k+1} = u_k + s_k. \tag{4.9b}$$

For systems with a large dimension m it can be advantageous not to work with the exact Jacobian matrix $JF(u_k)$, but with an approximation $A(u_k)$. The iteration (4.8) then becomes

$$u_{k+1} = u_k - \big(A(u_k)\big)^{-1} F(u_k) \qquad (k = 0, 1, 2, \ldots)\,. \tag{4.10}$$

The family of methods that use this strategy are called *quasi-Newton methods*. In general, the convergence will become slower, but each iteration step in (4.10) may be cheaper than in (4.8). There are a number of choices for such a modification. For example, (i) the partial derivatives $\partial F_i(v)/\partial v_j$ in the Jacobian matrix can be replaced by difference quotients $\frac{1}{h}\big(F_i(v+he_j) - F_i(v)\big)$, where e_j is the j-th standard basis vector in $\mathbb{R}^m$ (i.e., all components of e_j are 0 except for the j-th component which equals 1). An alternative option, which can be used if u_0 is known to be close to u_*, is to take (ii) $A(v) = JF(u_0)$, so that the partial derivatives only need to be computed once. This may also reduce the work needed to solve the linear systems in the iteration (because only one LU-decomposition will be needed; see Section 2.2).

Without proof we mention that the local convergence result of Theorem 3.5 for Newton's method can be generalized to systems of equations:

[4] See Appendix B.4

Theorem 4.5. *Let $F : \Omega \to \mathbb{R}^m$ be twice continuously differentiable on an open set $\Omega \subset \mathbb{R}^m$, and assume $u_* \in \Omega$ is given with $F(u_*) = 0$ and $JF(u_*)$ nonsingular. Then there exist $\delta > 0$, $\gamma \geq 0$ such that for all u_0 with $\|u_0 - u_*\| \leq \delta$ the Newton iteration converges to u_* and satisfies the quadratic convergence estimate*

$$\|u_k - u_*\| \;\leq\; \gamma \, \|u_{k-1} - u_*\|^2 \qquad (k \geq 1)\,. \tag{4.11}$$

When applying Newton's method to a function F, it would be useful to have insight into the so-called *domains of attraction* $\Omega(u_*)$ for all its zeros u_*. A domain of attraction $\Omega(u_*)$ consists of all starting vectors u_0 for which Newton's method converges towards u_*. However, as the next example shows, these sets are complicated in general.

Example 4.6. As illustration, consider the Newton iteration $z_{k+1} = z_k - f(z_k)/f'(z_k)$ in the complex plane $\mathbb{C}$ for the function $f(z) = z^3 - 1$, leading to

$$z_{k+1} = \tfrac{2}{3} z_k + \tfrac{1}{3} z_k^{-2}\,.$$

It can be checked that this iteration in $\mathbb{C}$ is the same as for the corresponding function $F : \mathbb{R}^2 \to \mathbb{R}^2$ with $F_1(x, y) = \mathrm{Re} f(x + iy)$ and $F_2(x, y) = \mathrm{Im} f(x + iy)$.

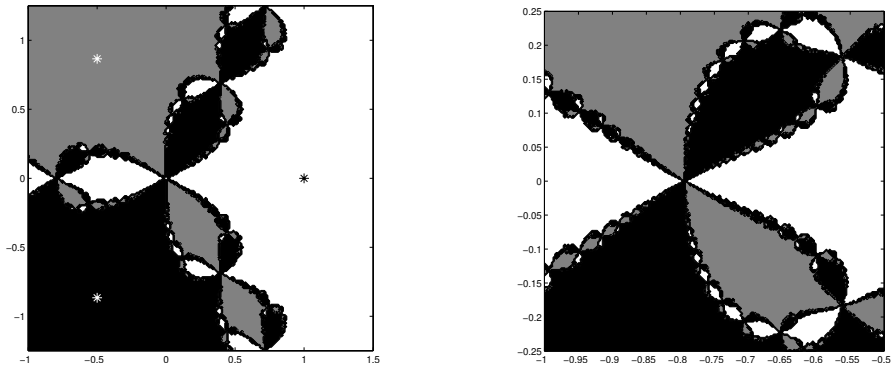


Figure 4.1: Domains of attraction of the Newton iteration. The right panel contains a zoom of the left panel around $z = -0.75$.

The domains of attraction for the three zeros $z_* = 1$ and $z_* = -\frac{1}{2} \pm \frac{1}{2} i\sqrt{3}$ are shown in Figure 4.1 as white, gray and black regions. The plot window in the left panel is given by $-1 \leq \mathrm{Re}\, z \leq 1.5$, $-1.25 \leq \mathrm{Im}\, z \leq 1.25$. The right panel, containing an enlargement around $z = -0.75$, reveals the fractal structure of the sets. ◊

4.3 Unconstrained Optimization *

We consider the problem of minimizing a function of several variables. More specifically, given a function $f : \mathbb{R}^m \to \mathbb{R}$, we are interested in the following:

$$\text{Find } u_* \in \mathbb{R}^m \text{ such that } f(u_*) = \min_{v \in \mathbb{R}^m} f(v).$$

This is called an *unconstrained minimization problem*, and f is usually called *objective function*. The problem is unconstrained because we are looking for the solution in the whole $\mathbb{R}^m$. If we looked for $\min_{v\in\Omega} f(v)$ with $\Omega \subset \mathbb{R}^m$, then we would have a *constrained minimization problem*. A constrained minimization problem can be turned in an unconstrained problem by introducing an auxiliary variable and the so-called Lagrangian, see Section 7.3 in [11].
To look for (local) minimizers in an unconstrained minimization problem, a first step is to look for stationary points. Then we can possibly check the behavior of the second derivatives at that point to see if it is a minimizer (and not a maximizer or saddle point). With this approach, we are interested in:

$$\text{Find } u_* \in \mathbb{R}^m \text{ such that } \nabla f(u_*) = 0.$$

This is exactly a root finding problem for a function from $\mathbb{R}^m$ to $\mathbb{R}^m$!
To find zeros of ∇f, one can therefore use the methods seen in the previous section, like Newton's method, or modifications of them. When using Newton's method, the function for which we are looking for the root is $F := \nabla f$, meaning that, for its Jacobian JF, we need second derivatives of f, more precisely its Hessian. This is addressed in Assignment 6 for a particular case. The interested reader is referred to Section 7.2 in [11] for more details.

Chapter 5

Polynomial Interpolation (and Approximation)

Suppose that distinct points $x_0, x_1, \ldots, x_m$ in an interval $[a, b]$ are given together with corresponding numbers $f_0, f_1, \ldots, f_m$. We want to find a function P that interpolates these data,

$$P(x_i) = f_i \qquad (i = 0, 1, \ldots, m)\,. \tag{5.1}$$

In this chapter we will consider *polynomial interpolation*, where P is a polynomial of degree m or less.

In general, the values f_i may originate from physical measurements or they can be the output produced by some other numerical algorithm. In Section 5.2 we focus on the special case that the points (x_i, f_i) are on the graph of a smooth function f, that is, $f_i = f(x_i)$. In that case we will assess the quality of the interpolation scheme by studying the interpolation error $f(x) - P(x)$.

In many cases we want to approximate a given function f on an interval $[a, b]$ by an interpolating polynomial P, and are we still free to choose the nodes x_j. Such polynomial approximations have many applications. For example, the integral $\int_a^b f(x)\,dx$ can then be approximated by $\int_a^b P(x)\,dx$, something we will do in Chapter 6. It will turn out in Section 5.3 that the so-called *Chebyshev nodes* are a particularly good choice.

In Section 5.6 we will look at interpolation with so-called *cubic splines*. These functions can smoothly interpolate while minimizing oscillations.

5.1 Interpolation Formulas

In this section, we essentially discuss how to represent an interpolating polynomial, that is, the choice of a basis for the space of polynomials of (at most) a given degree.

A first option that comes to mind is to use the monomial basis. Namely, if we have to interpolate a function at $m + 1$ distinct nodes $x_0, x_1, \ldots, x_m$, we write $P(x) = p_0 + p_1 x + \ldots + p_m x^m$, and, by imposing the interpolation conditions, we obtain the polynomial coefficients by solving the

system

$$\begin{pmatrix} 1 & x_0 & x_0^2 & \dots & x_0^m \\ 1 & x_1 & x_1^2 & \dots & x_1^m \\ \vdots & \vdots & & \vdots & \\ 1 & x_m & x_m^2 & \dots & x_m^m \end{pmatrix} \begin{pmatrix} p_0 \\ p_1 \\ \vdots \\ p_m \end{pmatrix} = \begin{pmatrix} f_0 \\ f_1 \\ \vdots \\ f_m \end{pmatrix}.$$

The numerical problem with this approach is that the matrix on the left-hand side, called *Vandermonde* matrix, is very ill-conditioned! We have to look, therefore, for alternative representations of the polynomial.
For given distinct nodes $x_0, x_1, \dots, x_m$, the *Lagrange polynomials* L_i are defined as

$$L_i(x) \;=\; \prod_{\substack{j=0 \\ j\neq i}}^{m} \frac{x - x_j}{x_i - x_j} \;=\; \frac{(x-x_0)\cdots(x-x_{i-1})(x-x_{i+1})\cdots(x-x_m)}{(x_i-x_0)\cdots(x_i-x_{i-1})(x_i-x_{i+1})\cdots(x_i-x_m)} \tag{5.2}$$

for $i = 0, 1, \dots, m$.

Theorem 5.1. *If the nodes $x_0, x_1, \dots, x_m$ are distinct, then there exists a unique interpolating polynomial of degree $\leq m$. This polynomial is given by*

$$P(x) \;=\; \sum_{i=0}^{m} L_i(x)\, f_i\,. \tag{5.3}$$

Proof. The Lagrange polynomial L_i has value 1 in x_i and value 0 in the other nodes x_j, $j \neq i$. It is therefore clear that P is an interpolating polynomial of degree $\leq m$.
Moreover, it is the only interpolating polynomial of degree $\leq m$. To see this, suppose Q is another one. Then $R = P - Q$ is a polynomial of degree $\leq m$ with $m+1$ zeros x_j, $j = 0, \dots, m$, which is only possible if R is identically zero. □

Formula (5.3) is called the *Lagrange interpolation formula*. In the form presented, this formula is mainly of theoretical interest and not directly suitable for evaluating the polynomial P numerically. It can be rewritten in various ways to make it more suitable for computations. Of course, in view of the uniqueness, the formulas are mathematically equivalent.
An elegant and computationally suitable rewrite of the Lagrange formula is the so-called *barycentric interpolation formula*. For its derivation we introduce the monic polynomial

$$\omega(x) \;=\; (x-x_0)(x-x_1)\cdots(x-x_m) \tag{5.4}$$

and the factors

$$w_i \;=\; \prod_{\substack{j=0 \\ j\neq i}}^{m} \frac{1}{x_i - x_j} \qquad (i = 0, 1, \dots, m)\,. \tag{5.5}$$

Note that we can write

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

and therefore

$$P(x) \;=\; \omega(x) \sum_{i=0}^{m} \frac{w_i}{x - x_i} f_i \,. \tag{5.6}$$

Since the sum of all Lagrange polynomials $L_i(x)$ is equal to one (exercise: why?), it follows that

$$1 = \sum_{i=0}^{m} L_i(x) = \omega(x) \sum_{i=0}^{m} \frac{w_i}{x - x_i}.$$

Dividing (5.6) by this expression and cancelling the common factor $\omega(x)$, we obtain the *barycentric interpolation formula* for P,

$$P(x) = \frac{\displaystyle\sum_{i=0}^{m} \frac{w_i}{x - x_i} f_i}{\displaystyle\sum_{i=0}^{m} \frac{w_i}{x - x_i}}. \tag{5.7}$$

This elegant formula expresses $P(x)$ as a weighted average of the data values f_i. This explains the name 'barycentric formula', although the weight factors $w_i/(x - x_i)$ are generally not all positive.

Usually one wants to find $P(x)$ for a large number of different values x. For a given data set $(x_i, f_i), i = 0, 1, \ldots, m$, the computation of the factors $w_0, w_1, \ldots, w_m$ costs $\mathcal{O}(m^2)$ flops, and after that we can use (5.7) to evaluate $P(x)$ for any given x with an additional $\mathcal{O}(m)$ flops. This is much cheaper than with the direct use of (5.3), where the evaluation of $P(x)$ costs $\mathcal{O}(m^2)$ flops for each value of x.

For certain choices of the nodes x_i, for instance equispaced and Chebyshev nodes, the corresponding factors w_i are known analytically, so they don't need to be computed.

Remark 5.2. Another popular way for multiple output values is the *Newton divided difference formula*, where the following representation of the interpolating polynomial is used,

$$\begin{aligned} P(x) \;=\;& a_0 \\ &+\; a_1(x - x_0) \\ &+\; a_2(x - x_0)(x - x_1) \\ &+\; \ldots \\ &+\; a_m(x - x_0)(x - x_1)\ldots(x - x_{m-1}). \end{aligned}$$

The coefficients $a_0, a_1, \ldots, a_m$ are easily computed recursively from the interpolation data (x_i, f_i), $i = 0, 1, \ldots, m$. Once the coefficients have been computed, the evaluation of P at any given x only costs $\mathcal{O}(m)$ flops if one uses the algorithm

$$P(x) = a_0 + (x - x_0)(a_1 + (x - x_1)(a_2 + \ldots (a_{m-1} + (x - x_{m-1})a_m)\ldots)).$$

This method is extensively discussed in the books [4], [13]. ◊

5.2 The Interpolation Error

To derive an expression for the error made by interpolation, we assume that the points (x_j, f_j) are on the graph of a smooth function f, that is, $f_j = f(x_j)$. The polynomial ω is defined as in (5.4).

Theorem 5.3. *Suppose $f : [a,b] \to \mathbb{R}$ is $(m+1)$ times continuously differentiable. Let P be the polynomial of degree $\le m$ that interpolates f at the distinct points $x_j \in [a,b]$ for $j = 0, 1, \ldots, m$. Then for any $x \in [a,b]$ there exists $\xi_x \in [a,b]$ such that*

$$f(x) - P(x) \;=\; \frac{1}{(m+1)!}\,\omega(x)\,f^{(m+1)}(\xi_x)\,.$$

Proof. Suppose $x \neq x_j$ for $j = 0, 1, \ldots, m$; otherwise there is nothing to prove. For this fixed x, let $\kappa \in \mathbb{R}$ be such that $f(x) - P(x) = \kappa\,\omega(x)$, and consider

$$Q(y) = f(y) - P(y) - \kappa\,\omega(y) \qquad (\text{for } y \in [a,b])\,.$$

By construction, we have $Q(y) = 0$ for $y = x$. Note that we also have $Q(y) = 0$ for $y = x_j$, $j = 0, 1, \ldots, m$. Therefore, Q has at least $m+2$ zeros in $[a,b]$. Between any two zeros of Q there is a zero of Q' (Rolle's theorem[1]), so we know that Q' has at least $m+1$ zeros in $[a,b]$. Subsequently, Q'' has at least m zeros in $[a,b]$, and so on. Finally, $Q^{(m+1)}$ has at least one zero $\xi_x \in [a,b]$. For this zero we have $0 = Q^{(m+1)}(\xi_x) = f^{(m+1)}(\xi_x) - (m+1)!\,\kappa$, that is, $\kappa = \frac{1}{(m+1)!} f^{(m+1)}(\xi_x)$, from which the result follows. □

Corollary 5.4. *Under the assumptions of the previous theorem, we have*

$$|f(x) - P(x)| \;\le\; \frac{1}{(m+1)!} \max_{\xi \in [a,b]} |f^{(m+1)}(\xi)| \cdot |\omega(x)|\,.$$

□

Illustration. Suppose $x_0 = \min_i x_i$ and $x_m = \max_i x_i$. We can use the interpolation formulas outside $[x_0, x_m]$, but this is not recommended. It will usually lead to large errors, due to the fact that the function values $|\omega(x)|$ quickly become very large outside the interval $[x_0, x_m]$.
An illustration with $m = 10$ is given in the left panel of Figure 5.1 for $f(x) = \cos(5x - 1)$ with equally spaced points $x_j = -1 + 2j/m \in [-1, 1]$. For this function we have $\frac{1}{(m+1)!}|f^{(m+1)}(\xi)| \le \frac{1}{(m+1)!} 5^{m+1}$, which is small for large m. Indeed with increasing m, the interpolation error becomes very small on $[-1, 1]$. However outside the interpolation interval we see that the error on the left becomes large right away.

A more serious problem arises when $\frac{1}{(m+1)!}|f^{(m+1)}|$ is not small. Then the error can exhibit large oscillations towards the endpoints of the interpolation interval. This is called the *Runge phenomenon*. An illustration is presented in the right panel of Figure 5.1 for $f(x) = (1+25x^2)^{-1}$ with the same grid spacing and $m = 10$ as before. In the middle of the interpolation interval the result is not too bad, and this would in fact become better with larger m but then the oscillations near the endpoints become much worse!

[1] See Theorem B.3

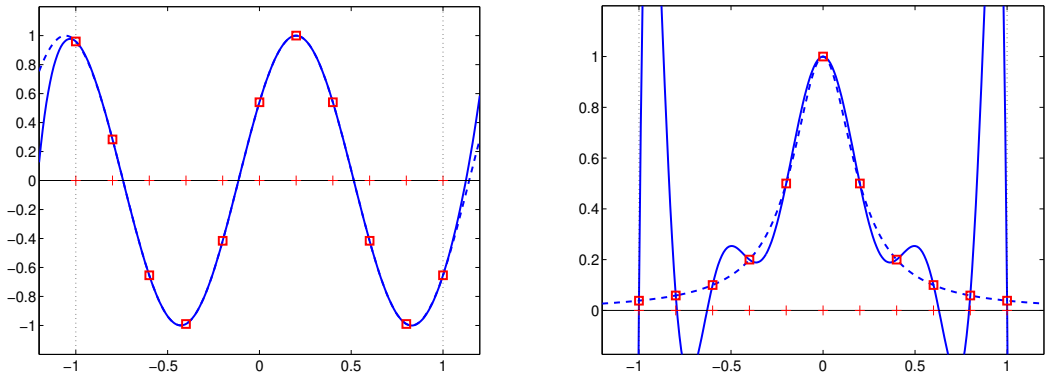


Figure 5.1: Interpolation with polynomials of degree $m = 10$, equidistant nodes $x_i = -1 + 0.2i$ in $[-1, 1]$ with $f_i = f(x_i)$, where $f(x) = \cos(5x - 1)$ (left panel) and $f(x) = 1/(1 + 25x^2)$ (right panel). The graph of f is given by the dashed line, the interpolating polynomial is the solid line.

5.3 Chebyshev Nodes

To some extent the large oscillations in the right panel of Figure 5.1 are due to the equal spacing of the nodes x_j used for that computation. A natural question is how to choose these nodes in $[a, b]$ such that $\max_{x\in[a,b]} |\omega(x)|$ is minimized. To answer this question, let us first take $[a, b] = [-1, 1]$; later it can be transformed back to arbitrary intervals.
Consider the *Chebyshev polynomials* T_n, defined for $n = 0, 1, 2, \ldots$ by the relation

$$\cos n\theta = T_n(\cos\theta), \tag{5.8}$$

or, equivalently, by setting $x = \cos\theta$,

$$T_n(x) \;=\; \cos(n \arccos(x)) \qquad (-1 \le x \le 1)\,. \tag{5.9}$$

For example, we have $\cos 2\theta = 2\cos^2\theta - 1$ and therefore $T_2(x) = 2x^2 - 1$. By using the identity $\cos(n + 1)\theta + \cos(n - 1)\theta = 2\cos\theta\cos n\theta$ we obtain the recursion

$$T_0(x) = 1\,, \quad T_1(x) = x\,, \quad T_n(x) \;=\; 2x\,T_{n-1}(x) - T_{n-2}(x) \quad (n \ge 2)\,. \tag{5.10}$$

This makes it clear that T_n is a polynomial of degree n with leading coefficient equal to 2^{n-1} (if $n \ge 1$). Note also that formula (5.10) can be used for all $x \in \mathbb{R}$. Moreover, if $n \ge 1$, it follows from (5.8) that $|T_n(x)| = 1$ for $n+1$ distinct points in $[-1, 1]$:

$$T_n(\eta_k) \;=\; (-1)^k \qquad \text{for} \quad \eta_k = \cos\left(\frac{k\pi}{n}\right), \quad k = 0, 1, \ldots, n\,, \tag{5.11a}$$

and that all roots of T_n are in the interval $(-1, 1)$:

$$T_n(\xi_k) \;=\; 0 \qquad \text{for} \quad \xi_k = \cos\left((k + \tfrac{1}{2})\frac{\pi}{n}\right), \quad k = 0, 1, \ldots, n - 1\,. \tag{5.11b}$$

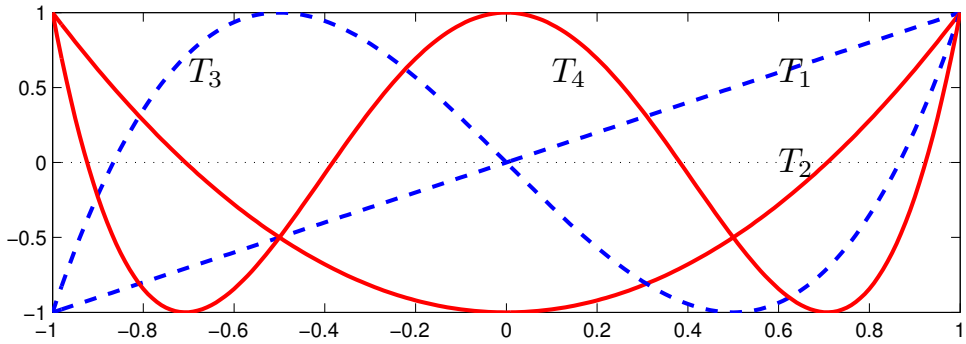


Figure 5.2: Chebyshev polynomials T_1, T_2, T_3, T_4 on $[-1, 1]$.

Lemma 5.5. *We have* $\max_{x\in[-1,1]} |T_n(x)| = 1$. *Moreover, if* $n \geq 1$ *and* S_n *is another polynomial of degree* n *with leading coefficient* 2^{n-1}, *then* $\max_{x\in[-1,1]} |S_n(x)| \geq 1$.

Proof. The property $\max_{x\in[-1,1]} |T_n(x)| = 1$ follows directly from (5.9) and (5.11a). Now suppose S_n and T_n have the same leading coefficient and $\max_{x\in[-1,1]} |S_n(x)| < 1$. Then $R(x) = T_n(x) - S_n(x)$ is a polynomial of degree $\leq n$–1, because the highest powers cancel. We know from (5.11a) that $T_n(\eta_k) = (-1)^k$ for $k = 0, 1, \ldots, n$. Since $|S(\eta_k)| < 1$ it follows that $\text{sign}(R(\eta_k)) = (-1)^k$. Hence R has a zero on each interval (η_k, η_{k-1}), $k = 1, 2, \ldots, n$. Therefore R has at least n zeros on $[-1, 1]$, which implies $R \equiv 0$. This contradicts the assumption $\max_{x\in[-1,1]} |S_n(x)| < 1$. □

Remark 5.6. * By counting the zeros a bit more carefully, it is also easy to show that $\max_{x\in[-1,1]} |S_n(x)| > 1$ if $S_n \neq T_n$. ◊

Now, returning to our problem, we see from Lemma 5.5 (with $n = m + 1$) that the minimum value of $\max_{x\in[-1,1]} |\omega(x)|$ is equal to 2^{-m}, and that this minimum is attained by the monic polynomial $\omega(x) = 2^{-m} T_{m+1}(x)$. This means that the optimal nodes x_j are the zeros of T_{m+1}, that is,

$$x_j = \cos\left((j + \tfrac{1}{2})\frac{\pi}{m+1}\right) \qquad (j = 0, 1, \ldots, m)\,. \tag{5.12}$$

For an arbitrary interval $[a, b]$ we use the transformation $x \mapsto \frac{1}{2}(a+b) + \frac{1}{2}(b-a)x$ to map $[-1, 1]$ onto $[a, b]$. It follows that the minimum value of $\max_{x\in[a,b]} |\omega(x)|$ is equal to $2^{-m} \cdot (\frac{1}{2}(b-a))^{m+1} = 2((b-a)/4)^{m+1}$, and that this minimum is attained for the nodes

$$x_j = \tfrac{1}{2}(a+b) + \tfrac{1}{2}(b-a)\cos\left((j + \tfrac{1}{2})\frac{\pi}{m+1}\right) \qquad (j = 0, 1, \ldots, m)\,. \tag{5.13}$$

Illustration. The results for these Chebyshev nodes on $[-1, 1]$ with $m = 10$ are shown in Figure 5.3, and compare favourably with those shown in Figure 5.1 for the equally spaced nodes. The very large oscillations at the endpoints for $f(x) = 1/(1 + 25x^2)$ have become less severe, but it is clear that the interpolation is still not accurate there. However, with these Chebyshev nodes the interpolation will improve quickly if we increase m, see Table 5.1, in contrast to the case with equidistant nodes.

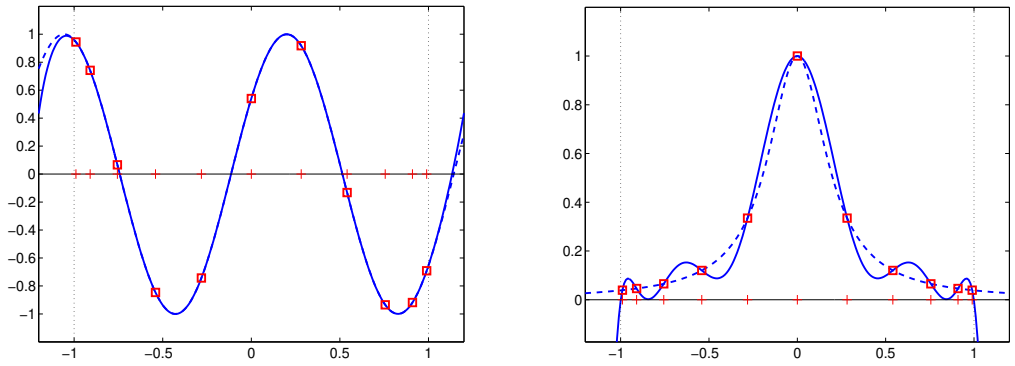


Figure 5.3: Interpolation with polynomials of degree $m = 10$, Chebyshev nodes in $[-1, 1]$ with $f_i = f(x_i)$, where $f(x) = \cos(5x-1)$ (left panel) and $f(x) = 1/(1+25x^2)$ (right panel). This is to be compared with Figure 5.1.

For the smooth function $f(x) = \cos(5x - 1)$ the errors are not visible inside the interpolation interval, but they are actually much smaller for the Chebyshev points (maximal error $7.09 \cdot 10^{-4}$ on $[-1, 1]$) than for the uniformly distributed points (maximal error $6.74 \cdot 10^{-3}$).

Table 5.1: Maximal error $\max_{x\in[-1,1]} |f(x) - P(x)|$, $f(x) = 1/(1 + 25x^2)$, for the Chebyshev nodes with increasing m.

m	10	20	30	40	50	60
Max. err.	$1.09 \cdot 10^{-1}$	$1.53 \cdot 10^{-2}$	$2.06 \cdot 10^{-3}$	$2.89 \cdot 10^{-4}$	$3.96 \cdot 10^{-5}$	$5.42 \cdot 10^{-6}$

Remark 5.7. It can be shown (but this is quite difficult) that the interpolating polynomials with Chebyshev nodes will converge for increasing m to $f(x)$, uniformly on $[a, b]$, provided that f is *continuously differentiable* on that interval.
A famous theorem of Weierstrass states that any *continuous* function f can be approximated by polynomials with arbitrarily high accuracy, uniformly on the interval $[a, b]$. A constructive proof of this result can be given using the Bernstein polynomials $B_n(x) = \sum_{j=0}^{n} \binom{n}{j} x^j (1-x)^{n-j} f(\frac{j}{n})$ with interval $[a, b] = [0, 1]$.
These polynomials $B_n(x)$ will converge uniformly to $f(x)$ for increasing n, but the convergence is very, very slow. From a practical numerical point of view these Bernstein polynomials are therefore not suited to approximate functions. ◊

5.4 The Lebesgue constant

The possibly highly oscillatory behavior of the interpolant given a choice of interpolation nodes is quantified by the so-called Lebesgue constant, which we now define. If P is the interpolating polynomial for the function f, then, for every x in a given interval $[a, b]$ containing all $m + 1$

interpolation points, (5.3) gives us

$$|P(x)| = \left| \sum_{i=0}^{m} f_i L_i(x) \right| \leq \max_i |f_i| \sum_{i=0}^{m} |L_i(x)|$$

and

$$\|P(x)\|_\infty \leq \|f\|_\infty \left\| \sum_{i=0}^{m} |L_i(x)| \right\|_\infty, \tag{5.14}$$

where $\|\cdot\|_\infty$ denotes here the maximum norm on $[a, b]$. The quantity on the right-hand side,

$$\Lambda_m := \left\| \sum_{i=0}^{m} |L_i(x)| \right\|_\infty, \tag{5.15}$$

is the *Lebesgue constant* associated to the interpolation nodes. Note that it depends on the choice of nodes only (and their number), not on the function to be interpolated. Equation (5.14) tells us that the Lebesgue constant measures how much maximum values (in absolute value) reached by f can get amplified be the interpolating polynomial, depending on the choice of nodes. It can be checked that the Lebesgue constant for equidistant nodes grows exponentially with m as $m \to \infty$, while the one for Chebyshev nodes grows at most logarithmically:

$$\Lambda_m^{equi} \sim \frac{2^{m+1}}{em \log m}, \quad \Lambda_m^{Cheb} \sim \frac{2}{\pi} \log(m+1), \tag{5.16}$$

e being the Nepier's constant. This explains the behavior of the interpolants seen in the previous sections.

Remark 5.8. * If we consider the map $X : C^0([a, b]) \to C^0([a, b])$ that associates to a given continuous function f its interpolating polynomial P for a given choice of nodes, then the Lebesgue constant is the operator norm of this mapping:

$$\Lambda_m = \sup_{f \in C^0([a,b])} \frac{\|P\|_\infty}{\|f\|_\infty}$$

(note that $\|\cdot\|_\infty$ is the standard norm on $C^0([a, b])$).

Remark 5.9. * The Lebesgue constant measures also how far a given interpolant is from the best polynomial approximation of a function on a polynomial space. Namely, if $E_m^*(f) := \inf_{p \in \mathbb{P}_m} \|f - p\|_\infty$ is the best approximation error for f in the space $\mathbb{P}_m$ of polynomials of degree at most m, then if P is the interpolating polynomial on given $m + 1$ nodes,

$$\|f - P\|_\infty \leq (1 + \Lambda_m) E_m^*(f),$$

see [10, p.13] for details.

5.5 Piecewise Linear Interpolation

Assume $a = x_0 < x_1 < \ldots < x_m = b$, and let $h_i = x_i - x_{i-1}$ for $i = 1, 2, \ldots, m$. To avoid the oscillatory behaviour that can be encountered with high-order interpolation, we can consider interpolation on each sub-interval $[x_{i-1}, x_i]$ with a low order polynomial P_i.
The simplest interpolation of this kind is piecewise linear interpolation,

$$P_i(x) = \frac{x - x_{i-1}}{h_i} f_i - \frac{x - x_i}{h_i} f_{i-1} \qquad \text{for } x_{i-1} \le x \le x_i \,. \tag{5.17}$$

From Corollary 5.4 it follows (why?) that, if $f_i = f(x_i)$ with a function f that is twice continuously differentiable, then

$$|f(x) - P_i(x)| \le \tfrac{1}{8} h_i^2 \max_{x_{i-1} \le \xi \le x_i} |f''(\xi)| \qquad \text{for all } x \in [x_{i-1}, x_i] \,. \tag{5.18}$$

Hence, any twice continuously differentiable function $f : [a, b] \to \mathbb{R}$ can be approximated by its piecewise linear interpolant with a maximum error satisfying

$$\max_{x \in [a,b]} |f(x) - P(x)| \le \tfrac{1}{8} h^2 \max_{\xi \in [a,b]} |f''(\xi)|, \tag{5.19}$$

where $h = \max_i h_i$.

5.6 Interpolation with Splines

In the previous section we saw that piecewise linear interpolation leads to an interpolation error of order $\mathcal{O}(h^2)$, see (5.19). Higher accuracies can be obtained if we take on each subinterval $[x_{i-1}, x_i]$ a polynomial P_i of higher degree, for example, of degree 3 and interpolating (x_j, f_j), $j = i–2, i–1, i, i+1$. However, at the nodes we will then have merely continuity of our interpolant, while in some cases higher global smoothness is needed. A very popular choice is to take a *spline* interpolant.

5.6.1 Definitions and general properties

In the following definition, $\mathbb{P}_k$ denotes the space of polynomials of degree at most k on a given interval.

Definition 5.10. *Let $a = x_0 < x_1 < \ldots < x_m = b$ be $m + 1$ distinct nodes in $[a, b]$. A function $S : [a, b] \to \mathbb{R}$ is a* spline *of degree $k \ge 1$ relative to the nodes $\{x_i\}_{i=0}^m$ if*

$$\begin{aligned} &S|_{[x_i, x_{i+1}]} \in \mathbb{P}_k, \quad \text{for } i = 0, \ldots, m-1, \\ &S \in C^{k-1}([a, b]). \end{aligned}$$

Note that piecewise linear interpolants as seen in the previous section are splines of degree 1. When $k = 3$, we have *cubic splines*, which are piecewise third order polynomials and have global

C^2-continuity. Splines are the basic ingredient for many numerical techniques, for instance in isogeometric analysis.
We denote by $\mathcal{S}_k$ the vector space of splines of degree k. From Definition 5.10, we see that we have $m(k+1)$ degrees of freedom ($k+1$ polynomial coefficients on m intervals) and $(m-1)k$ continuity constraints (from C^0 to C^{k-1}). Therefore, $\dim\mathcal{S}_k = m(k+1) - (m-1)k = m+k$. This means that the conditions in the definition of a spline do not determine the latter uniquely and we still have $m+k$ degrees of freedom. If we further require a spline to be interpolatory at the $m+1$ nodes, we are left with $k-1$ degrees of freedom. This means that, when $k > 1$, we need to impose further constraints. We have different possibilities for these and they usually impose some behavior at the boundaries, for instance:

$$S^{(j)}(a) = S^{(j)}(b) \text{ for } j = 0, \dots, k-1, \quad \text{the so-called } \textit{periodic} \text{ splines}, \tag{5.20}$$

if we have to fit a periodic function (note that the condition for $j = 0$ overlaps with the interpolation condition in this case), or

$$\begin{aligned} &\text{for } k = 2l-1 \text{ with } l \geq 2, : \\ &S^{(l+j)}(a) = S^{(l+j)}(b) = 0 \text{ for } j = 0, \dots, l-2, \quad \text{the so-called } \textit{natural} \text{ or } \textit{free} \text{ splines}. \end{aligned} \tag{5.21}$$

5.6.2 Interpolatory cubic splines

Here we focus on the case $k = 3$, which is quite popular in applications. We introduce the following notation:

$$f_i = S(x_i), \quad m_i := S'(x_i), \quad M_i := S''(x_i), \tag{5.22}$$

for $i = o, \dots, m$. Note that the first equality coincides with the interpolation conditions. Then, following Definition 5.10 and denoting $S_i := S|_{[x_i, x_{i+1}]}$, we have also the following conditions:

$$\begin{aligned} S_i(x_{i-1}) &= f_{i-1} && \text{for } i = 1, \dots, m \quad (C^0 - \text{continuity}), \\ S'_{i-1}(x_{i-1}) &= S'_i(x_{i-1}) && \text{for } i = 2, \dots, m \quad (C^1 - \text{continuity}), \\ S''_{i-1}(x_{i-1}) &= S'_i(x_{i-1}) && \text{for } i = 2, \dots, m \quad (C^2 - \text{continuity}), \end{aligned}$$

to which we can add, for instance, the conditions for natural splines:

$$S''(a) = S''(b) = 0. \tag{5.23}$$

Another possibility for cubic splines is to use *clamped* splines:

$$S'(a) = f'_0\,, \; S'(b) = f'_m, \tag{5.24}$$

namely imposing the slope at the boundaries.

Illustration. One of the motivations to look at splines was the oscillating behaviour of the high-order interpolants for the function $f(x) = 1/(1+25x^2)$ on $[-1, 1]$. The right panel of Figure 5.4 shows the natural cubic spline interpolating this function at the equally spaced points $x_i = -1 + 0.2i$, $i = 0, 1, \dots, m = 10$. The results are now very satisfactory, in particular when

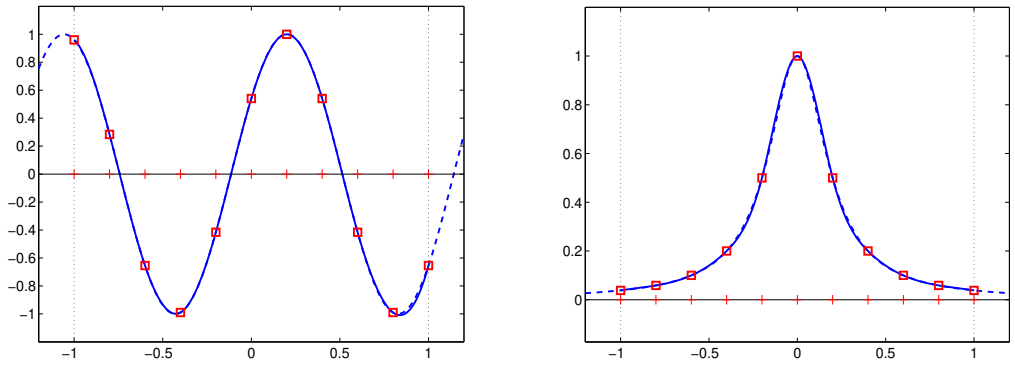


Figure 5.4: Natural spline interpolation with nodes $x_i = -1 + 0.2i$ in $[-1, 1]$ for $f_i = f(x_i)$, with $f(x) = \cos(5x - 1)$ (left panel) and $f(x) = 1/(1 + 25x^2)$ (right panel). As before, the graph of f is given by the dashed line, and the interpolant is the solid line.

compared the the previous results with the interpolants of degree 10. The spline produces indeed a 'good looking' interpolant. The theoretical justification is given by Theorem 5.11 below.
For the smooth function $f(x) = \cos(5x - 1)$ the maximal error on $[-1, 1]$ of the interpolating natural spline is $5.31 \cdot 10^{-2}$; for the clamped spline this error is smaller, $3.09 \cdot 10^{-3}$. Here the high-order polynomial interpolation with Chebyshev points is more accurate. The same happens with the function $f(x) = 1/(1+25x^2)$ if the number of points m gets larger. The maximal errors with increasing m are given in Table 5.2, which is to be compared with Table 5.1.

Table 5.2: Maximal error $\max_{x\in[-1,1]} |f(x) - P(x)|$, $f(x) = 1/(1 + 25x^2)$, for the natural splines with equidistant nodes and increasing m.

m	10	20	30	40	50	60
Max. err.	$2.20 \cdot 10^{-2}$	$3.18 \cdot 10^{-3}$	$8.24 \cdot 10^{-4}$	$2.78 \cdot 10^{-4}$	$1.12 \cdot 10^{-4}$	$5.27 \cdot 10^{-5}$

Construction of cubic splines. In order to fully determine a (cubic) spline, we need to be able to find an expression for it on each interval. Here we will see that, for cubic splines, this amounts to solving a tridiagonal system.
Using again the notation $S_i = S|_{[x_i,x_{i+1}]}$ and $h_i = x_i - x_{i-1}$, we know that, on each interval, the second derivative of a cubic spline is a first order polynomial. So, using the notation from 5.22, we can write

$$S''_{i-1}(x) = M_{i-1}\frac{x_i - x}{h_i} + M_i\frac{x - x_{i-1}}{h_i}, \quad x \in [x_i, x_{i+1}],$$

for $i = 1, \ldots, m$. Integrating twice, we obtain

$$S''_{i-1}(x) = M_{i-1}\frac{(x_i - x)^3}{6h_i} + M_i\frac{(x - x_{i-1})^3}{6h_i} + C_{i-1}(x - x_{i-1}) + \tilde{C}_{i-1}, \quad x \in [x_i, x_{i+1}],$$

for constants C_{i-1}, $\tilde{C}_{i-1}$ and $i = 1, \ldots, m$. Imposing the interpolation conditions we obtain

$$\tilde{C}_{i-1} = f_{i-1} - M_{i-1}\frac{h_i^2}{6}, \quad C_{i-1} = \frac{f_i - f_{i-1}}{h_i} - \frac{h_i}{6}(M_i - M_{i-1}),$$

which, considered for $i = 1, \ldots, m$, give rise to a tridiagonal system, the so-called *M-continuity* system:

$$\mu_i M_{i-1} + 2M_i + \lambda_i M_{i+1} = d_i \text{ for } i = 1, \ldots, m-1, \tag{5.25}$$

where

$$\mu_i = \frac{h_i}{h_i + h_{i+1}}, \quad \lambda_i = \frac{h_{i+1}}{h_i + h_{i+1}}, \quad d_i = \frac{6}{h_i + h_{i+1}} \left(\frac{f_{i+1} - f_i}{h_{i+1}} - \frac{f_i - f_{i-1}}{h_i} \right). \tag{5.26}$$

Since (5.25) has $m-1$ equations and $m+1$ unknowns, we add conditions such as for periodic and natural splines. These conditions are usually of the form

$$2M_0 + \lambda_0 M_1 = d_0, \quad \mu_m M_{m-1} + 2M_m = d_m, \tag{5.27}$$

with $0 \leq \lambda_0, \mu_m \leq 1$, d_0 and d_m given. For instance, if we set $\lambda_0 = \mu_m = d_0 = d_m = 0$, we obtain natural cubic splines.

Properties of cubic splines. For a given real function f, let S be an interpolating cubic spline with $S(x_i) = f(x_i)$ for $0 \leq i \leq m$. We will see in a moment that splines have an interesting minimization property with respect to the integral $\int_a^b f''(x)^2\,dx$, which can be viewed as a measure for the total bending of f over the interval $[a, b]$. This explains the absence of oscillations in Figure 5.4.

Theorem 5.11. *Suppose f is two times continuously differentiable on $[a, b]$, and the interpolating spline is either natural (with $S''(a) = S''(b) = 0$) or clamped (with $S'(a) = f'(a)$, $S'(b) = f'(b)$). Then*

$$\int_a^b S''(x)^2\,dx \;\leq\; \int_a^b f''(x)^2\,dx\,. \tag{5.28}$$

Proof. Inequality (5.28) follows immediately from the equality

$$\int_a^b f''(x)^2 dx \;=\; \int_a^b \big[f''(x) - S''(x)\big]^2\,dx \;+\; \int_a^b S''(x)^2 dx,$$

which we will show in the following. First observe that this equality is equivalent to

$$\int_a^b S''(x)\big[f''(x) - S''(x)\big]\,dx \;=\; 0. \tag{5.29}$$

To prove (5.29), note that its left-hand side can be rewritten by applying partial integration,

$$\int_a^b S''(x)\big[f''(x) - S''(x)\big]\,dx \;=\; S''(x)\big[f'(x) - S'(x)\big]\Big|_a^b - \int_a^b S'''(x)\big[f'(x) - S'(x)\big]\,dx\,.$$

For the natural and clamped splines it holds that

$$S''(a)[f'(a) - S'(a)] = S''(b)[f'(b) - S'(b)]\,,$$

giving

$$\int_a^b S''(x)\big[f''(x) - S''(x)\big]\,dx \;=\; \int_a^b S'''(x)\big[S'(x) - f'(x)\big]\,dx\,.$$

Further we know that S''' is piecewise constant, $S'''(x) = 6w_i$ for $x_{i-1} < x < x_i$, and therefore

$$\int_a^b S'''(x)\big[S'(x) - f'(x)\big]\,dx \;=\; \sum_{i=1}^{m} 6w_i \int_{x_{i-1}}^{x_i} \big(S'(x) - f'(x)\big)\,dx\,.$$

Since $\int_{x_{i-1}}^{x_i}(S'(x) - f'(x))\,dx = \big(S(x) - f(x)\big)\big|_{x_{i-1}}^{x_i} = 0$, this gives (5.29). □

Remark 5.12.* Without proof we mention the following approximation result: if f is four times continuously differentiable and S is the clamped spline with $S'(a) = f'(a)$ and $S'(b) = f'(b)$, then the interpolation error satisfies

$$\max_{x\in[a,b]} |f(x) - S(x)| \;\le\; \tfrac{5}{384} h^4 \max_{\xi\in[a,b]} |f^{(4)}(\xi)| \tag{5.30}$$

where $h = \max_i h_i$.

Remark 5.13.* Another important family of splines is the one of B-splines, see for instance [11, Sect. 8.7.2].

Chapter 6

Numerical Integration

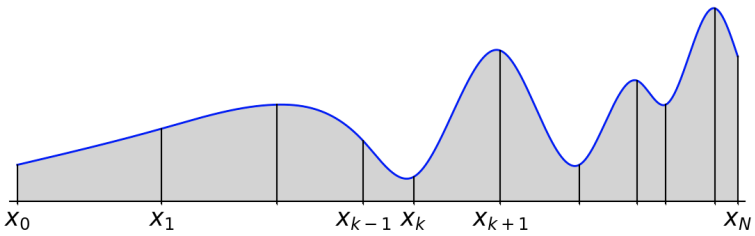
In this chapter we discuss *numerical quadrature*, that is, numerical approximation of integrals

$$I = \int_a^b f(x)\,dx \tag{6.1}$$

with given function $f : [a, b] \to \mathbb{R}$. We focus on *interpolatory* quadrature rules, namely, rules that can be seen as applying exact integration to an interpolant of the integrand.

6.1 Composite Integration Schemes

Since the function to be integrated may exhibit different behaviour on different parts of the integration interval, it can be natural to break down the integration into these parts.



Let us assume that we have some suitable partitioning

$$a = x_0 < x_1 < x_2 < \cdots < x_N = b\,,$$

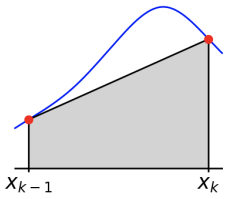
which divides the integration interval $[a, b]$ into N sub-intervals $[x_{k-1}, x_k]$. Then we can rewrite the integral I in (6.1) as

$$I = \sum_{k=1}^{N} I_k \quad \text{with} \quad I_k = \int_{x_{k-1}}^{x_k} f(x)\,dx\,. \tag{6.2}$$

It remains to find suitable approximations $\tilde{I}_k$ for the sub-integrals I_k. Let us first consider some simple examples, where we denote $h_k = x_k - x_{k-1}$.

On each sub-interval $[x_{k-1}, x_k]$ we can approximate $f(x)$ by a linear interpolating polynomial

$$f(x) \approx \frac{x_k - x}{h_k} f(x_{k-1}) + \frac{x - x_{k-1}}{h_k} f(x_k) .$$

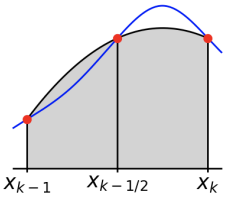


Integrating this linear interpolant leads to the *trapezoidal rule*

$$\tilde{I}_k = \tfrac{1}{2} h_k \Big(f(x_{k-1}) + f(x_k) \Big) . \tag{6.3}$$

Better accuracy can be obtained by using a quadratic interpolant on $[x_{k-1}, x_k]$ with nodes x_{k-1}, x_k and $x_{k-1/2} = \frac{1}{2}(x_{k-1} + x_k)$. A little calculation shows that this leads to the *Simpson rule*

$$\tilde{I}_k = \tfrac{1}{6} h_k \Big(f(x_{k-1}) + 4 f(x_{k-\frac{1}{2}}) + f(x_k) \Big) . \tag{6.4}$$



For the approximation of the integral $I = \sum_{k=1}^{N} I_k$ the application of such a quadrature rule leads to a so-called *composite* integration scheme $\tilde{I} = \sum_{k=1}^{N} \tilde{I}_k$. For example, if we have $h_k = h$ for all k, then the composite trapezoidal rule gives

$$\tilde{I} = \tfrac{1}{2} h f(x_0) + h f(x_1) + h f(x_2) + \ldots + h f(x_{N-1}) + \tfrac{1}{2} h f(x_N) , \tag{6.5}$$

and the composite Simpson rule yields

$$\begin{aligned} \tilde{I} = {} & \tfrac{1}{6} h f(x_0) + \tfrac{2}{3} h f(x_{\frac{1}{2}}) + \tfrac{1}{3} h f(x_1) + \tfrac{2}{3} h f(x_{\frac{3}{2}}) + \ldots \\ & \ldots + \tfrac{1}{3} h f(x_{N-1}) + \tfrac{2}{3} h f(x_{N-\frac{1}{2}}) + \tfrac{1}{6} h f(x_N) . \end{aligned} \tag{6.6}$$

Illustration. As a simple illustration of the performance of the composite formulas we compute approximations to

$$I = \int_0^{1/2} \sin(\pi x)\, dx = \frac{1}{\pi} ,$$

with a uniform partitioning $x_k = kh$, $h = 1/(2N)$. Along with the results $\tilde{I}_T$ for the composite trapezoidal rule (6.5) and $\tilde{I}_S$ for the composite Simpson rule (6.6), we also consider the result of the simple Riemann sum

$$\tilde{I}_R = h f(x_0) + h f(x_1) + \cdots + h f(x_{N-1}) . \tag{6.7}$$

The performance of these composite formulas is given in Table 6.1. It is obvious that the Riemann sum gives larger errors than the composite trapezoidal rule, while the composite Simpson rule gives the most accurate results by far. A closer inspection of the table reveals that doubling the number of sub-intervals N (that is, halving the interval length h) roughly corresponds to multiplying the error by a factor of 2^{-p}, where $p = 1$ for the Riemann sum, $p = 2$ for the composite trapezoidal rule, and $p = 4$ for the composite Simpson rule. Hence the errors seems to behave as $|I - \tilde{I}| = \mathcal{O}(h^p)$ where $p = 1, 2, 4$. We will explain this behaviour in Section 6.3.

Table 6.1: Errors $|I - \tilde{I}|$ for the Riemann sum $\tilde{I}_R$, the composite trapezoidal rule $\tilde{I}_T$ and the composite Simpson rule $\tilde{I}_S$, with $I = \int_0^{1/2} \sin(\pi x)\, dx$.

N	1	2	4	8	16
$\lvert I - \tilde{I}_R \rvert$	$3.2 \cdot 10^{-1}$	$1.4 \cdot 10^{-1}$	$6.6 \cdot 10^{-2}$	$3.2 \cdot 10^{-2}$	$1.6 \cdot 10^{-2}$
$\lvert I - \tilde{I}_T \rvert$	$6.8 \cdot 10^{-2}$	$1.6 \cdot 10^{-2}$	$4.1 \cdot 10^{-3}$	$1.0 \cdot 10^{-3}$	$2.6 \cdot 10^{-4}$
$\lvert I - \tilde{I}_S \rvert$	$7.2 \cdot 10^{-4}$	$4.2 \cdot 10^{-5}$	$2.6 \cdot 10^{-6}$	$1.6 \cdot 10^{-7}$	$1.0 \cdot 10^{-8}$

6.2 General Quadrature Formulas

In the previous section we considered two simple methods (the trapezoidal rule and Simpson's rule) to approximate the integral I_k on a sub-interval $[x_{k-1}, x_k]$. A change of variable yields

$$I_k = \int_{x_{k-1}}^{x_k} f(x)\, dx = h_k \int_0^1 f(x_{k-1} + t h_k)\, dt = h_k \int_0^1 g(t)\, dt, \tag{6.8}$$

where $h_k = x_k - x_{k-1}$ and $g(t) = f(x_{k-1} + t h_k)$. This shows that in order to find an approximation for I_k there is no loss of generality if we limit ourselves to integrals with $[0, 1]$ as integration interval,

$$J = \int_0^1 g(t)\, dt. \tag{6.9}$$

For the approximation of the integral J we consider the general *quadrature formula* (also called *quadrature method*)

$$\widetilde{J} = \sum_{i=1}^{s} b_i\, g(c_i), \tag{6.10}$$

with $b_i, c_i \in \mathbb{R}$ and with $s \geq 1$ an integer. The numbers b_i are called the *weights* and the numbers c_i the *nodes* of the quadrature formula. For example, with the trapezoidal rule we have $s = 2$, $b_1 = b_2 = \frac{1}{2}$, $c_1 = 0$, $c_2 = 1$, and Simpson's rule has $s = 3$, $b_1 = b_3 = \frac{1}{6}$, $b_2 = \frac{4}{6}$, $c_1 = 0$, $c_2 = \frac{1}{2}$, $c_3 = 1$.

We emphasize that for a given quadrature formula of the form (6.10) it follows from (6.8) that the corresponding approximation of I_k is given by

$$\tilde{I}_k = h_k \sum_{i=1}^{s} b_i g(c_i) = h_k \sum_{i=1}^{s} b_i f(x_{k-1} + c_i h_k). \tag{6.11}$$

We will now investigate the accuracy of quadrature formula (6.10). First note that every quadrature formula of this type is exact for the the zero polynomial. In the following definition we assume[1] that the zero polynomial has degree -1 or $-\infty$.

[1]The degree of the zero polynomial is usually left undefined, but in those cases in the literature where it is defined, it is usually set at -1 or $-\infty$. The latter value has the advantage that the rule $\deg(fg) = \deg(f) + \deg(g)$ is satisfied for all polynomials f and g.

Definition 6.1. *The order of quadrature formula (6.10) is defined as the largest integer p such that $\widetilde{J} = J$ whenever g is a polynomial of degree $\leq p-1$. In other words, it is the degree of the lowest degree polynomial for which the formula is not exact.*

One easily verifies that the order p of a quadrature method is well-defined and satisfies $p \geq 0$. The following theorem gives a simple algebraic characterization for higher orders than zero. We use the convention $c_i^0 = 1$, also if $c_i = 0$.

Theorem 6.2. *Let $q \geq 1$ be a given integer. The order p of the quadrature formula (6.10) satisfies $p \geq q$ if and only if*

$$\sum_{i=1}^{s} b_i \, c_i^{j-1} \;=\; \frac{1}{j} \qquad \textit{for } \; j = 1, 2, \ldots, q \,. \tag{6.12}$$

Proof. To emphasize the dependence of J and $\widetilde{J}$ on the function g we write

$$J(g) = \int_0^1 g(t)\, dt, \quad \widetilde{J}(g) = \sum_{i=1}^{s} b_i \, g(c_i).$$

One easily verifies that (6.12) is equivalent to

$$\widetilde{J}(t^{j-1}) = J(t^{j-1}), \quad j = 1, 2, \ldots, q, \tag{6.13}$$

so (6.12) is clearly necessary for order $\geq q$. To show sufficiency, first note that any polynomial g of degree $\leq q-1$ is a linear combination of the polynomials $1, t, t^2, \ldots, t^{q-1}$, that is,

$$g(t) = \sum_{j=1}^{q} \alpha_j t^{j-1}.$$

Using the linearity of both $J(g)$ and $\widetilde{J}(g)$ with respect to g it follows from (6.13) that

$$J(g) = J\Big(\sum_{j=1}^{q} \alpha_j t^{j-1}\Big) = \sum_{j=1}^{q} \alpha_j J(t^{j-1}) = \sum_{j=1}^{q} \alpha_j \widetilde{J}(t^{j-1}) = \widetilde{J}\Big(\sum_{j=1}^{q} \alpha_j t^{j-1}\Big) = \widetilde{J}(g).$$

□

For a method of order p we define its *error constant* as

$$C_{p+1} \;=\; \frac{1}{p!}\Big(\frac{1}{p+1} \;-\; \sum_{i=1}^{s} b_i \, c_i^p\Big). \tag{6.14}$$

Note that it follows from Definition 6.1 and Theorem 6.2 that this constant is nonzero. The relevance of this constant will become clear in the next section. It will turn out that the smaller (the absolute value of) this constant the better.
As a consequence of Theorem 6.2, we have the following result.

Corollary 6.3. *A quadrature rule with $s \geq 1$ distinct nodes has at least order s.*

Proof. Let c_i, $i = 1, 2, \ldots, s$ be the quadrature nodes. From Theorem 6.2, we know that the quadrature method has order $\geq s$ if and only if the weights b_i satisfy

$$\begin{pmatrix} 1 & 1 & \ldots & 1 \\ c_1 & c_2 & \ldots & c_s \\ \vdots & \vdots & & \vdots \\ c_1^{s-1} & c_2^{s-1} & \ldots & c_s^{s-1} \end{pmatrix} \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_s \end{pmatrix} = \begin{pmatrix} 1 \\ \frac{1}{2} \\ \vdots \\ \frac{1}{s} \end{pmatrix}. \tag{6.15}$$

The square matrix for this linear system is a so-called Vandermonde matrix, and it can be checked that it is nonsingular if and only if the nodes are distinct. As a consequence, for given distinct nodes c_i we can always find corresponding unique weights b_i such that we have order $p \geq s$. □

A natural question is whether the order can be larger than s.

Example 6.4. One easily verifies that the trapezoidal rule has order $p = s = 2$ and error constant $C_3 = -\frac{1}{12}$. Note that this method is based on linear interpolation, so it is no surprise that it is exact for all polynomials of degree ≤ 1.
A little more surprising is the fact that Simpson's rule has order $p = s + 1 = 4$ and $C_5 = -\frac{1}{2880}$. Note that this method is based on quadratic interpolation, so it must be exact for all polynomials of degree ≤ 2. It is therefore somewhat unexpected that this method has order 4, so is exact for all polynomials of degree ≤ 3. The underlying reason for having order $p = s + 1$ will be explained in Section 6.4. ◊

6.3 The Error for Composite Integration

In this section we derive an error bound for the composite integration scheme if the underlying quadrature formula (6.10) has order p.
We start with investigating the error $I_k - \tilde{I}_k$ on a sub-interval $[x_{k-1}, x_k]$. Recall from (6.8)-(6.11) that if we introduce $h_k = x_k - x_{k-1}$ and define the function g as

$$g(t) = f(x_{k-1} + t h_k), \quad 0 \leq t \leq 1,$$

then the exact integral $I_k = \int_{x_{k-1}}^{x_k} f(x)\, dx$ and its approximation $\tilde{I}_k$ can be written as

$$I_k = h_k J(g), \quad \tilde{I}_k = h_k \widetilde{J}(g), \tag{6.16}$$

where

$$J(g) = \int_0^1 g(t)\, dt, \quad \widetilde{J}(g) = \sum_{i=1}^{s} b_i g(c_i).$$

Note that the function g depends on k (so we should actually write g_k), but for better readability we suppress the index k.

We assume that f is $p+1$ times continuously differentiable on the integration interval $[a,b]$. This implies that g is $p+1$ times continuously differentiable on $[0,1]$, so we can use Taylor expansion[2] to obtain

$$g(t) = \underbrace{g(0) + tg'(0) + \cdots + \frac{t^{p-1}}{(p-1)!}g^{(p-1)}(0)}_{P(t)} + \underbrace{\frac{t^p}{p!}g^{(p)}(0)}_{Q(t)} + \underbrace{\frac{t^{p+1}}{(p+1)!}g^{(p+1)}(\xi)}_{R(t)},$$

where the value ξ occurring in the remainder term $R(t)$ depends on t and lies in the interval $[0,t] \subset [0,1]$. Note that the remainder term $R(t)$ can be uniformly bounded on the interval $[0,1]$ as follows,

$$|R(t)| \leq \frac{t^{p+1}}{(p+1)!} \max_{\xi\in[0,1]} |g^{(p+1)}(\xi)| \leq \frac{1}{(p+1)!} h_k^{p+1} \max_{x\in[a,b]} |f^{(p+1)}(x)|.$$

Hence there exists a constant $C > 0$ such that $\max_{t\in[0,1]} |R(t)| \leq Ch_k^{p+1}$, which can be conveniently expressed by using the 'big $\mathcal{O}$ notation' (see Appendix B.1) as

$$\max_{t\in[0,1]} |R(t)| = \mathcal{O}(h_k^{p+1}). \tag{6.17}$$

Using $g(t) = P(t) + Q(t) + R(t)$ we find from (6.16) that

$$\begin{aligned} I_k - \tilde{I}_k &= h_k(J(g) - \widetilde{J}(g)) \\ &= \underbrace{h_k(J(P) - \widetilde{J}(P))}_{0} + \underbrace{h_k(J(Q) - \widetilde{J}(Q))}_{h_k^{p+1} f^{(p)}(x_{k-1}) C_{p+1}} + \underbrace{h_k(J(R) - \widetilde{J}(R))}_{\mathcal{O}(h_k^{p+2})}. \end{aligned}$$

The first term is zero because a method of order p is exact for polynomials of degree $\leq p-1$. The expression for the second term follows from

$$\begin{aligned} J(Q) - \widetilde{J}(Q) &= \frac{g^{(p)}(0)}{p!} \left(\int_0^1 t^p dt - \sum_{i=1}^s b_i c_i^p \right) \\ &= \frac{h_k^p f^{(p)}(x_{k-1})}{p!} \left(\frac{1}{p+1} - \sum_{i=1}^s b_i c_i^p \right) = h_k^p f^{(p)}(x_{k-1}) C_{p+1}, \end{aligned}$$

where C_{p+1} denotes the error constant defined in (6.14). That the third term is $\mathcal{O}(h_k^{p+2})$ follows from (6.17) and

$$\begin{aligned} |J(R)| &= \left| \int_0^1 R(t)\, dt \right| \leq \int_0^1 |R(t)|\, dt \leq \max_{t\in[0,1]} |R(t)|, \\ |\widetilde{J}(R)| &= \left| \sum_{i=1}^s b_i R(c_i) \right| \leq \sum_{i=1}^s |b_i R(c_i)| \leq \left(\sum_{i=1}^s |b_i| \right) \max_{t\in[0,1]} |R(t)|. \end{aligned}$$

[2] see Appendix section B.3

We conclude that

$$I_k - \tilde{I}_k = h_k^{p+1} C_{p+1} f^{(p)}(x_{k-1}) + \mathcal{O}(h_k^{p+2}). \tag{6.18}$$

Now, let us consider the error $I - \tilde{I} = \sum_{k=1}^{N}(I_k - \tilde{I}_k)$ for the composite integration scheme. Then we directly obtain from (6.18) that

$$|I - \tilde{I}| \leq \sum_{k=1}^{N} \left(|C_{p+1}|\, h_k^{p+1} |f^{(p)}(x_{k-1})| + \mathcal{O}(h_k^{p+2}) \right).$$

Introducing the maximal mesh-width $h = \max_k h_k$, and noting that for $n = p+1$ and $n = p+2$ we have

$$\sum_{k=1}^{N} h_k^n \leq \sum_{k=1}^{N} h^{n-1} h_k = (b-a) h^{n-1},$$

it follows that

$$|I - \tilde{I}| \leq (b-a)\, |C_{p+1}|\, h^p \max_{x \in [a,b]} |f^{(p)}(x)| + \mathcal{O}(h^{p+1}). \tag{6.19}$$

This general error bound for composite integration schemes is the main result of this section. It shows *convergence of order* p upon refinement of the partitioning, $h \to 0$, under the assumption that f is $p+1$ times continuously differentiable on $[a, b]$. Also note that the dominant term of the error bound contains the factor $|C_{p+1}|$, so the smaller this value the better.

6.4 Super-convergence of symmetric quadrature rules

As we saw in Section 6.2, for given distinct nodes c_i there is a unique choice of the weights b_i that leads to an order $p \geq s$. Sometimes the order even satisfies $p > s$. Methods with $p > s$ are called *super-convergent*. In this section, we see that symmetries in the quadrature rule lead super-convergent methods. In the next section, we look at other rule exhibiting super-converges. We use the same notation as in (6.9) and (6.10), that is,

$$J = \int_0^1 g(t)\, dt, \qquad \tilde{J} = \sum_{i=1}^{s} b_i g(c_i).$$

Definition 6.5. *A quadrature formula is said to be symmetric if*

$$b_i = b_{s+1-i} \quad (\text{for } i = 1, 2, \ldots, s), \tag{6.20a}$$
$$c_i = 1 - c_{s+1-i} \quad (\text{for } i = 1, 2, \ldots, s). \tag{6.20b}$$

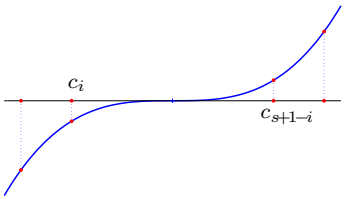
Having symmetry is quite natural; it means that $\int_0^1 g(t)\, dt$ is approximated the same way as the mirror integral $\int_0^1 g(1-t)\, dt$, and in general there is no reason to have a directional preference.

Theorem 6.6. *A symmetric quadrature formula always has an even order p.*

Proof. We will show for any given symmetric quadrature formula that $p \geq 2k-1$ implies $p \geq 2k$. The main intuition behind the proof is that, if a formula is symmetric, it integrates odd functions (and in particular polynomials of odd order) exactly to 0.
The assumption $p \geq 2k-1$ means that the method is exact for all polynomials of degree $\leq 2k-2$. To prove $p \geq 2k$ we only need to show that it is exact for polynomials of degree $2k-1$ too. Any polynomial g of degree $2k-1$ can be written as

$$g(t) = \alpha \left(t - \tfrac{1}{2}\right)^{2k-1} + R(t)$$

with R a polynomial of degree $\leq 2k-2$ and $\alpha \neq 0$ the leading coefficient of g.
Since R is integrated exactly by the quadrature formula, we only have to show that the formula is also exact for the integral $J = \int_0^1 (t - \frac{1}{2})^{2k-1}\, dt = 0$. The numerical approximation $\widetilde{J} = \sum_i b_i (c_i - \frac{1}{2})^{2k-1}$ can be rewritten by first switching to the new summation index $j = s+1-i$ and then using the symmetry relations to obtain



$$\widetilde{J} = \sum_j b_{s+1-j}(c_{s+1-j} - \tfrac{1}{2})^{2k-1} = \sum_j b_j (\tfrac{1}{2} - c_j)^{2k-1} = -\sum_j b_j (c_j - \tfrac{1}{2})^{2k-1} = -\widetilde{J},$$

which shows that $\widetilde{J} = 0$. □

This explains why the Simpson rule has order $p = 4$ (as observed in Example 6.4): it clearly has order $p \geq 3$ because it is based on quadratic interpolation, but the method is also symmetric.

6.5 Gauss Quadrature

As we will see shortly, orders larger than $s+1$ are possible with suitable choices of the nodes. The goal of this section is to answer the following questions:

Q1. Is there a maximum order possible for an interpolatory quadrature rule with s distinct nodes?

Q2. If so, how can we construct rule that achieve this maximum order?

Before answering the questions above, we remind that, in view of equation 6.15, it is sufficient to specify the nodes of a quadrature rule, as they determine the weights uniquely. When the quadrature rule is the exact integral of a *global* interpolant for the underlying integrand, as will be the case in this section, then this fact can be seen also from another computation. Given s distinct nodes, let p_{s-1} be the global interpolant of the integrand f in these nodes. Then, since the quadrature rule is interpolatory, we have

$$\int_a^b f(x)\,\mathrm{d}x \approx \int_a^b p_{s-1}(x)\,\mathrm{d}x = \sum_{i=1}^{s} w_i f(x_i), \tag{6.21}$$

where w_i and x_i are, respectively, the weights and nodes in $[a, b]$. Using (5.3) we write

$$p_{s-1}(x) = \sum_{i=1}^{s} f(x_i) L_i(x), \tag{6.22}$$

where L_i is the Lagrange polynomial associated to the node x_i. By equating (6.21) and (6.22) we obtain the following expression for the weights:

$$w_i = \int_a^b L_i(x)\,\mathrm{d}x, \quad i = 1, \dots, s. \tag{6.23}$$

We have seen in Section 6.2 that it is sufficient to specify a quadrature rule on a reference interval, and there we used $[0, 1]$ for the latter. For Gaussian quadrature rules, it is more convenient to use $[-1, 1]$ as reference interval. With a slight abuse of notation, we still denote by b_i and c_i, $i = 1, \dots, s$, the weigths and nodes on $[-1, 1]$.

6.5.1 Maximal order of interpolatory quadrature rules

We now see a result from which the answer to Q1 follows. In the following statement, note that the assumption on the quadrature formula having at least order s is automatically fulfilled by interpolatory quadrature formulas with distinct nodes because of Corollary 6.3.

Theorem 6.7. *Assume a quadrature formula on $[-1, 1]$ has order $p \geq s$ and let $q \geq 1$ be an integer. Then the quadrature formula has order $p \geq s + q$ if and only if*

$$\int_{-1}^{1} \omega(t) Q(t)\,\mathrm{d}t = 0, \quad \text{for all polynomials } Q \text{ of degree} \leq q - 1 \tag{6.24}$$

Here $\omega(t) = (t - c_1)(t - c_2) \dots (t - c_s)$ is the monic polynomial associated to the quadrature nodes, as in (5.4).

Proof. Any polynomial g of degree $\leq s + q - 1$ can be divided by the polynomial ω (of degree s) leading to the unique decomposition

$$g(t) = \omega(t) Q(t) + R(t), \tag{6.25}$$

where the quotient Q and remainder R are polynomials with $\deg(Q) \leq q - 1$ and $\deg(R) \leq s - 1$. Conversely, if Q and R are arbitrary polynomials with $\deg(Q) \leq q - 1$ and $\deg(R) \leq s - 1$, then the polynomial g defined in (6.25) has degree $\leq s + q - 1$.
From the decomposition (6.25), we see that the exact integral and numerical approximation satisfy

$$\begin{aligned} J &= \int_{-1}^{1} g(t)\,\mathrm{d}t = \int_0^1 \omega(t) Q(t)\,\mathrm{d}t + \int_{-1}^{1} R(t)\,\mathrm{d}t, \\ \widetilde{J} &= \sum_{i=1}^{s} b_i g(c_i) = \sum_{i=1}^{s} b_i \underbrace{\omega(c_i)}_{=0} Q(c_i) + \sum_{i=1}^{s} b_i R(c_i). \end{aligned}$$

Since the quadrature formula has order $p \geq s$, it is exact for R. Therefore the quadrature formula is exact for g if and only if $\int_{-1}^{1} \omega(t) Q(t)\,\mathrm{d}t = 0$. □
From this, the answer to Q1 follows.

Corollary 6.8. *An interpolatory quadrature formula with s distinct nodes has at most order $2s$.*

Proof. We proceed by contradiction. If the quadrature rule could have higher order, it would have at least order $2s+1$, namely, it would integrate all polynomials up to degree $2s$ exactly. But then when could choose $Q = \omega$ in (6.24), leading to $\int_{-1}^{1}(\omega(t))^2\,\mathrm{d}t = 0$. This would imply $\omega \equiv 0$, which contradicts the assumption of having s distinct nodes. □

The above theorem provides us with a criterion for a quadrature method to have order $p \geq s+q$. We will use this criterion in the derivation of the so-called *Gauss methods*, which are the quadrature rules with maximal order $p = 2s$. This will answer Q2. To construct Gauss quadrature rules, we need to rely on the concepts from the next subsection.

6.5.2 Basic notions on orthogonal polynomials

We consider the space $\mathbb{P}_n$ of polynomials of degree at most n, and as usual we confine ourselves on polynomials on $[-1,1]$. We note that the quantity

$$(p,q) := \int_{-1}^{1} p(t)q(t)\,\mathrm{d}t \tag{6.26}$$

defines a *scalar product* on the space of continuous functions on $[-1,1]$ (exercise: check!), so in particular on polynomials.
With a scalar product at hand, we can define the notion of orthogonality with respect to that scalar product. More precisely, we say that $p, q \in \mathbb{P}_n$ are orthogonal if and only if $\int_{-1}^{1} p(t)q(t)\,\mathrm{d}t = 0$.
Given any basis of $\mathbb{P}_n$, we can use the Gram-Schmidt orthogonalization process with the scalar product (6.26) to construct a basis of *orthogonal polynomials*. These correspond to the *Legendre polynomials*.
The Legendre polynomial $\mathcal{L}_n$ of degree n is given by the formula

$$\mathcal{L}_n(t) \;=\; \frac{1}{2^n\,n!}\,\frac{d^n}{dt^n}(t^2-1)^n \qquad (n = 0,1,2\ldots)\,, \tag{6.27}$$

where the multiplying factor is chosen such that $\mathcal{L}_n(1) = 1$. The fact that

$$\int_{-1}^{1} \mathcal{L}_n(t)\mathcal{L}_j(t)\,dt = 0 \qquad (j = 0,1,\ldots,n-1) \tag{6.28}$$

can be shown, by a somewhat tedious calculation, using $(j+1)$ times integration by parts (each time with differentiation for $\mathcal{L}_j$), and using the fact that the k-th derivative of $(t^2-1)^n$ is zero at $t = \pm 1$ when $k < n$.
Summing up, $\{\mathcal{L}_0, \mathcal{L}_1, \ldots, \mathcal{L}_n\}$ is a basis for $\mathbb{P}_n$ with $\int_{-1}^{1} \mathcal{L}_n(t)Q(t)\,dt = 0$ whenever Q is a polynomial of degree less than n.
The Legendre polynomials can easily be constructed by using $\mathcal{L}_0(t) = 1$, $\mathcal{L}_1(t) = t$ and the recursion formula

$$\mathcal{L}_n(t) \;=\; \tfrac{2n-1}{n}\,t\,\mathcal{L}_{n-1}(t) - \tfrac{n-1}{n}\,\mathcal{L}_{n-2}(t) \qquad (n \geq 2). \tag{6.29}$$

The first few polynomials are displayed in Figure 6.1.

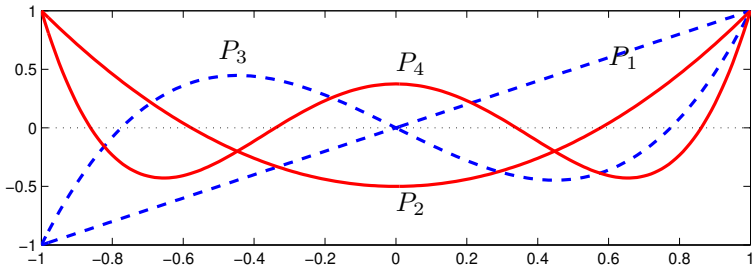


Figure 6.1: Legendre polynomials on $[-1, 1]$, here denoted by P_1, P_2, P_3, P_4 (instead of $\mathcal{L}_1, \mathcal{L}_2, \mathcal{L}_3, \mathcal{L}_4$).

6.5.3 Nodes of Gauss quadrature rules

We are now ready to construct Gauss quadrature rules and answer question Q2. We remind that we want to use s quadrature nodes to obtain a rule of order $2s$. Let $\{\mathcal{L}_i\}_{i=0}^{s}$ be a basis of $\mathbb{P}_s$ of orthogonal polynomials, namely the Legendre polynomials in our case. Theorem 6.7 with $q = s$ tells us that the monic polynomial ω associated to the quadrature nodes must be orthogonal to $\mathcal{L}_0, \ldots, \mathcal{L}_{s-1}$, which is a basis for $\mathbb{P}_{s-1}$. Since $\omega \in \mathbb{P}_s$, it follows that ω must be parallel to $\mathcal{L}_s$, that is, $\omega = \lambda \mathcal{L}_s$ for some $\lambda \in \mathbb{R} \setminus \{0\}$. This means that the zeroes of ω are the same as the zeroes of $\mathcal{L}_s$. The zeroes of ω are the quadrature nodes, so we obtain that the s nodes of the Gauss quadrature rule must be the zeroes of the s-th Legendre polynomial $\mathcal{L}_s$.
It is possible to prove the following properties of the nodes of a Gaussian quadrature rule, see e.g. [1, p.70]:

- the weights are all positive;
- the nodes are all internal nodes, that is they are in $(-1, 1)$.

The first property is desirable from a numerical point of view to have less numerical cancellation when applying the quadrature rule.
For $s = 1$, the node of the Gauss quadrature rule is the zero of $\mathcal{L}_1 = x$, so $c_1 = 0$. The associated weight is $b_1 = 2$ and the rule is called the *midpoint rule*. It is somewhat related to the trapezoidal rule, but the error constant of the midpoint rule is smaller in absolute value.
The nodes and weights of the first three Gauss methods are displayed in Table 6.2. For the higher order methods these coefficients c_i and corresponding b_i can be found in tables or computed numerically.

Remark 6.9. *For some applications it is natural to have either $c_1 = 0$ or $c_s = 1$. This leads to the Radau quadrature methods with order $2s - 1$. If both $c_1 = 0$ and $c_s = 1$ are prescribed, then an order $2s - 2$ can be reached, and this is called Lobatto quadrature. Gauss-Radau and Gauss-Lobatto quadrature rules are discussed in Assignment 11.*
Other popular methods are obtained by basing the nodes on the zeros or extrema of Chebyshev polynomials. The resulting methods are known as Clenshaw-Curtis formulas. They only have order s, but are still very accurate due to very small error constants. ♢

Table 6.2: Nodes and weights for the Gauss methods on $[-1,1]$ with $s = 1,2,3$.

s	p	$(b_1,\dots,b_s)$	$(c_1,\dots,c_s)$
1	2	2	0
2	4	$(1\,,\,1)$	$\left(-\frac{1}{\sqrt{3}}\,,\,\frac{1}{\sqrt{3}}\right)$
3	6	$\left(\frac{5}{9}\,,\,\frac{8}{9}\,,\,\frac{5}{9}\right)$	$\left(-\sqrt{\frac{3}{5}}\,,\,0\,,\,\sqrt{\frac{3}{5}}\right)$

6.5.4 Weighted integrals

In some applications, we need to compute *weighted* integrals, that is integrals of the form

$$\int_a^b f(x)w(x)\,\mathrm{d}x, \tag{6.30}$$

where w, called *weight* function (not to be confused with the quadrature weights!), is a non-negative, integrable function on (a,b). An instance where integrals of the form (6.30) is the computation of averages in probability and statistics: if X is a random variable on $\mathbb{R}$ and w is its density, then, for an integrable function f, $\mathbb{E}_X[f(X)] = \int_{-\infty}^{\infty} f(x)w(x)\,\mathrm{d}x$.
In case we are interested in a Gaussian quadrature rule for (6.30), we can proceed as in the previous subsections by replacing the scalar product (6.26) with the weighted one:

$$(p,q)_w := \int_a^b p(t)q(t)w(t)\,\mathrm{d}t. \tag{6.31}$$

Then, we will have different families of orthogonal polynomials, where orthogonality is defined in terms of the weight. Note that what we have done in the previous subsections is to consider the particular case $w \equiv 1$ on $[a,b]$.

Example 6.10. *For $w(t) = (1-t)^\alpha(1+t)^\beta$ and $\alpha,\beta > -1$ on $[-1,1]$ the family of orthogonal polynomials corresponds to Jacobi polynomials. Legendre polynomials are the Jacobi polynomials for $\alpha = \beta = 0$.*
For the case $\alpha = \beta = -\frac{1}{2}$, the weight is $w(t) = (1-t^2)^{-\frac{1}{2}}$, and the corresponding particular case of Jacobi polynomials are the Chebyshev polynomials already encountered in Chapter 5.

The Gauss quadrature formula for (6.30) is then obtained by selecting the nodes to be the zeroes of the s-th polynomial in the orthogonal system defined by the weight. The same properties for the quadrature nodes and weights (being inner nodes and positive quadrature weights) hold in the weighted case.

Remark 6.11.* So far we focused on integrals on bounded domains. When the interval is $[0,\infty)$ and the weight is $w(t) = e^{-t}$ (the density of an exponential random variable), the associated orthogonal polynomials are the Laguerre polynomials. When the integration interval is the whole $\mathbb{R}$ and $w(t) = e^{-t^2}$ (the density of a standard normal distribution), we have the Hermite polynomials.

6.6 Practical Error Estimation and Partitioning *

When using a composite quadrature rule, we can either choose the division into subintervals a priori or on-the-fly, in a way adapted to the integrand. The last strategy leads to *adaptive* quadrature methods. The idea is to start from a relatively coarse interval subdivision and use an indicator for the error in each subinterval to decide where to refine the subdivision. In this way, we have an efficient quadrature rule for the specific integral that we are considering. Those of you interested in reading more about this topic can consult, for instance, Subsection 9.7.2 in [11].

Chapter 7

Numerical Methods for Initial Value Problems

In this chapter we will discuss methods for the numerical solution of systems of ordinary differential equations (ODEs) of the general form

$$\begin{cases} u_1'(t) &= f_1(t, u_1(t), u_2(t), \dots, u_m(t)) \,, \\ u_2'(t) &= f_2(t, u_1(t), u_2(t), \dots, u_m(t)) \,, \\ & \vdots \\ u_m'(t) &= f_m(t, u_1(t), u_2(t), \dots, u_m(t)) \,, \end{cases}$$

with time $t \in [0, T]$. In vector notation, this can be written as $u'(t) = f(t, u(t))$ with unknown $u(t) = (u_i(t)) \in \mathbb{R}^m$ and given continuous function $f : [0, T] \times \mathbb{R}^m \to \mathbb{R}^m$. By specifying $u(t)$ at an initial time $t = t_0$, we arrive at an *initial value problem* for a system of ODEs,

$$u'(t) \;=\; f(t, u(t)) \,, \quad u(t_0) \;=\; u_0 \,, \tag{7.1}$$

with given initial value $u_0 \in \mathbb{R}^m$.
In the next section, we will first recall some basic facts on ODEs. Then, in Sections 7.2 and 7.3, we address so-called one-step numerical methods for their solution. In the last two sections, we look then at the approximation properties of these methods.

7.1 Basic facts on ODEs

Here we recall the main properties of ODEs which are needed for the development of numerical methods.

Definition 7.1. *In (7.1), if the right-hand side f depends on u only (and not on t directly), we say that the ODE system is* autonomous.

Note that an autonomous system is invariant with respect to time shifts: the solution to (7.1) at time t with initial condition at time t_0 is equal to the solution to (7.1) at time $t - t_0$ when replacing the initial condition by $u(0) = u_0$.

Before starting developing a numerical method to find an approximate solution to the ODE, it is important to make sure that, at the continuous level, the solution exists and is unique (at least locally). To have this, we need some very mild regularity requirements on the right-hand side.

Definition 7.2. *A function* $f : [0,T] \times \mathbb{R}^m \to \mathbb{R}^m$ *is* locally Lipschitz continuous *with respect its second argument at a point* $(\overline{t}, \overline{u})$ *if there exists an open neighborhood of* $\overline{t}$, $J : (\overline{t} - a, \overline{t} + a)$ *with* $a > 0$, *and an open neighborhood* $B_\rho(\overline{u})$ *of* $\overline{u}$ *(here* $B_\rho(\overline{u})$ *is the ball with radius* $\rho > 0$ *centered in* $\overline{u}$*) such that*

$$\|f(t,\widetilde{v}) - f(t,v)\| \le L\,\|\widetilde{v} - v\| \qquad \text{for all } t \in J \text{ and } v, \widetilde{v} \in B_\rho(\overline{u}), \tag{7.2}$$

for a (local) Lipschitz constant $L > 0$ *(where* $\|\cdot\|$ *denotes here a vector norm on* $\mathbb{R}^m$*).*

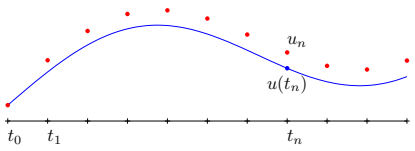
If in the definition above we can take $J = \mathbb{R}$ and $B_\rho(\overline{u}) = \mathbb{R}^m$, then the function is *globally Lipschitz* with respect to its second argument. Note also that, if a f is C^1 with respect to its second argument, then it is Lipschitz continuous.

Theorem 7.3 (Picard-Lindelöf – local existence and uniqueness)**.** *If* f *in* (7.1) *is locally Lipschitz continuous at a point* $(\overline{t}, \overline{u})$ *with respect to its second argument with Lipschitz constant* L *(and with neighborhood* $J = (\overline{t} - a, \overline{t} + a) \times B_\rho(\overline{u})$*), then* (7.1) *has a unique solution* $u \in C^1([\overline{t} - \alpha, \overline{t} + \alpha])$*, with* $\alpha = \min\left\{a, \frac{\rho}{M}, \frac{1}{L}\right\}$ *for* $M = \max_{(t,v)\in J\times B_\rho(\overline{u})} \|f(t,v)\|$.

For ease of treatment, in our following treatment we assume the right-hand side f to be globally Lipschitz continuous.

7.2 First definitions and examples

Without loss of generality, from now on we consider $t_0 = 0$ in (7.1).
The overall idea of most numerical methods is to choose points $t_n = n\tau$, $n = 0, 1, 2, \ldots$, in the time interval, with $\tau > 0$ being the *stepsize*, and to calculate numerical approximations $u_n \approx u(t_n)$ to the solution at these time points. For the moment, we assume the stepsize τ to be constant; variable stepsizes will be discussed later.



A numerical method is called a *one-step* method if, for very $n \ge 1$, u_n depends on u_{n-1} only and not on approximation at other time steps. Otherwise, the method is called a *multistep* method. For most of this chapter, we will focus on one-step methods, and we will come back to multistep methods at the end.
We will first consider some simple methods, and then see that they all belong to the class of so-called Runge-Kutta methods. Note that the solution of the initial value problem (7.1) satisfies

$$u(t_n) \;=\; u(t_{n-1}) \;+\; \int_{t_{n-1}}^{t_n} f(t, u(t))\, dt\,, \tag{7.3}$$

for $n = 1, 2, \ldots, N$, with number of steps N such that $N\tau = T$. This will provide a connection with the quadrature methods of Chapter 6.

Forward and implicit Euler. The simplest numerical method for solving initial value problem (7.1) is the *forward Euler method*, also called the *explicit Euler method*,

$$u_n \;=\; u_{n-1} + \tau f(t_{n-1}, u_{n-1})\,. \tag{7.4}$$

Another possibility is to use the *backward Euler method*, or *implicit Euler method*, which is

$$u_n \;=\; u_{n-1} + \tau f(t_n, u_n)\,. \tag{7.5}$$

It will be shown later (if f satisfies Lipschitz condition (7.2) and is continuously differentiable on $[0,T]\times\mathbb{R}^m$) that Euler's methods does converge to the exact solution, uniformly on the time interval $[0,T]$. However, the error will be proportional to τ only (convergence of order one).

Trapezoidal Rule. Applying the trapezoidal quadrature rule (6.3) to (7.3) gives the following ODE method, called the *implicit trapezoidal rule*, or Crank–Nicolson scheme,

$$u_n \;=\; u_{n-1} + \tfrac{1}{2}\tau f(t_{n-1}, u_{n-1}) + \tfrac{1}{2}\tau f(t_n, u_n)\,. \tag{7.6}$$

Note that relation (7.6) constitutes a system of (generally nonlinear) equations for the unknown vector u_n, so finding the new approximation u_n from u_{n-1} requires the solution of this system (e.g. with Newton's method). Since u_n is defined implicitly (via a system of equations), the method is called *implicit*.
Such an implicit system is avoided if we first compute a predictor u_n^* with Euler's method and substitute this in the right-hand side. The resulting method becomes

$$\begin{aligned} u_n^* \;&=\; u_{n-1} + \tau f(t_{n-1}, u_{n-1})\,,\\ u_n \;&=\; u_{n-1} + \tfrac{1}{2}\tau f(t_{n-1}, u_{n-1}) + \tfrac{1}{2}\tau f(t_n, u_n^*)\,. \end{aligned} \tag{7.7}$$

This method is known as the *explicit trapezoidal rule*, the *improved Euler method* and *Heun's method*.
As we will see later, both (7.6) and (7.7) are convergent with an error proportional to τ^2 (convergence of order two). Of course, computing the new approximation u_n with explicit method (7.7) is much easier than with its implicit counterpart (7.6). Nevertheless, we will see in Chapter 8 that there is an important class of initial value problems—the so called *stiff* problems—for which the implicit version (7.6) is to be preferred. The same holds for (7.4) versus the implicit counterpart (7.5). We will therefore not limit ourselves to the study of explicit methods, but also consider implicit methods in this chapter.

Illustration. On the time interval $[0,T]=[0,15]$ we consider the following initial value problem,

$$\begin{cases} v'(t) = (1-w(t))v(t), & v(0)=0.1,\\ w'(t) = (-1+1.2\,v(t))w(t), & w(0)=1. \end{cases} \tag{7.8}$$

Clearly this system is of the form (7.1) with

$$u(t) = \begin{pmatrix} u_1(t)\\ u_2(t)\end{pmatrix} = \begin{pmatrix} v(t)\\ w(t)\end{pmatrix}, \qquad u_0 = \begin{pmatrix} 0.1\\ 1\end{pmatrix},$$

and a function $f : [0,T] \times \mathbb{R}^2 \to \mathbb{R}^2$ defined as

$$f(t,x) = \begin{pmatrix} f_1(t,x_1,x_2) \\ f_2(t,x_1,x_2) \end{pmatrix} = \begin{pmatrix} (1-x_2)x_1 \\ (-1+1.2\,x_1)x_2 \end{pmatrix} \quad \text{for all } t \in [0,T], x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \in \mathbb{R}^2.$$

This system describes the population densities in a simple predator-prey (Lotka-Volterra) model, and its solution is known to be periodic in time. Although the solution and its period are not known exactly, we can obtain accurate approximations of these by using any of the above methods with a small enough stepsize τ.

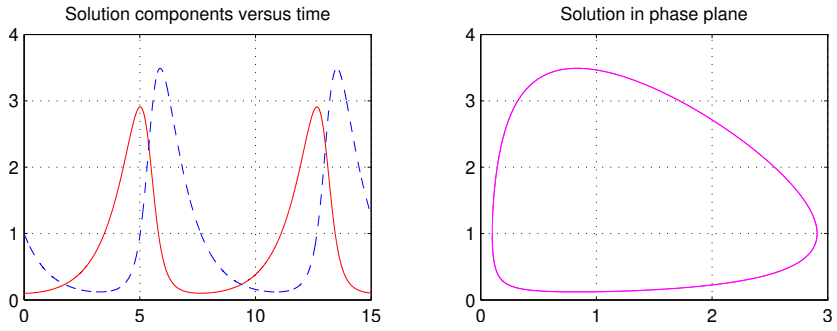


Figure 7.1: Solution for the Lotka-Volterra problem (7.8). Left panel: components $v(t)$ (solid line) and $w(t)$ (dashed line) as function of time t. Right panel: trajectory $(v(t), w(t))_{t\in[0,T]}$ in the phase plane.

An accurate approximation of the solution is displayed in Figure 7.1. The left panel of the figure shows $v(t)$ and $w(t)$ as function of $t \in [0, 15]$. The right panel gives a plot of the trajectory $(v(t), w(t))$ in the (v, w)-plane (the phase plane), making the periodic nature of the solution clearly visible.
In the following we try to compute the solution of problem (7.8) with Euler's method (7.4) and the explicit trapezoidal rule (7.7), first using $N = 100$ steps and then repeatedly doubling the number of steps. For both methods we compute the error $\|u(t_N) - u_N\|_2$ at the endpoint $t_N = T$ as function of the stepsize $\tau = T/N$. For the computation of these errors a very accurate reference value for $u(t_N)$ was used. The results are presented in Table 7.1.

Table 7.1: Error $\|u(t_N) - u_N\|_2$ for the Lotka-Volterra problem (7.8) at time $T = 15$, with number of steps $N = 100, 200, 400, \ldots$ and $\tau = T/N$.

N	100	200	400	800	1600	3200
Err. (7.4)	1.78	4.12	$9.87 \cdot 10^{-1}$	$3.64 \cdot 10^{-1}$	$1.59 \cdot 10^{-1}$	$7.49 \cdot 10^{-2}$
Err. (7.7)	$1.19 \cdot 10^{-2}$	$5.30 \cdot 10^{-3}$	$1.60 \cdot 10^{-3}$	$4.34 \cdot 10^{-4}$	$1.13 \cdot 10^{-4}$	$2.88 \cdot 10^{-5}$

The results with Euler's method are very inaccurate for $N \leq 400$. If more steps are taken, we see that the error halves (approximately) when the number of steps is doubled. It seems therefore that the error is inversely proportional to N for N sufficiently large. In other words, for $\tau > 0$ sufficiently small, the error appears to be directly proportional to τ.

The results with the explicit trapezoidal rule (7.7) are much better. The error now appears to be directly proportional to τ^2 if $\tau > 0$ is sufficiently small. Even though this method requires per step twice as much work as Euler's method, this extra effort per step clearly pays off. As we will see later, in Table 7.3, the results can still be considerably improved using higher-order methods.

7.3 Runge-Kutta Methods

All methods discussed so far belong to the class of so-called *Runge-Kutta methods*. For such a method, a single step $(t_{n-1}, u_{n-1}) \mapsto (t_n, u_n)$ requires the computation of a number of intermediate vectors $k_{n,1}, k_{n,2}, \ldots, k_{n,s}$. These intermediate vectors $k_{n,i}$ and the new approximation u_n are defined by

$$k_{n,i} \;=\; u_{n-1} \;+\; \tau \sum_{j=1}^{s} a_{ij} f(t_{n-1} + c_j\tau, k_{n,j}) \qquad (i = 1, 2, \ldots, s)\,, \tag{7.9a}$$

$$u_n \;=\; u_{n-1} \;+\; \tau \sum_{i=1}^{s} b_i f(t_{n-1} + c_i\tau, k_{n,i})\,. \tag{7.9b}$$

The method is specified by the coefficients a_{ij}, b_i, c_i, $(i, j = 1, 2, \ldots, s)$ where the b_i are called the *weights*, the c_i are the *nodes*, and s is the *number of stages*. The nodes are assumed to be in $[0, 1]$, and they are related to the coefficients a_{ij} by

$$c_i \;=\; \sum_{j=1}^{s} a_{ij}\,. \tag{7.10}$$

We will see in the next section the reason for this relationship. As already mentioned above, all methods discussed so far fit in this framework. Several other examples will be given later. Runge-Kutta methods can be conveniently represented by the tableau of coefficients on the right, known as *Butcher tableau*.

$$\begin{array}{c|ccc} c_1 & a_{11} & \cdots & a_{1s} \\ \vdots & \vdots & & \vdots \\ c_s & a_{s1} & \cdots & a_{ss} \\ \hline & b_1 & \cdots & b_s \end{array}$$

The Runge-Kutta method (7.9) is called *explicit* if $a_{ij} = 0$ for all i, j with $j \geq i$, and *implicit* otherwise. For an explicit method it is easily seen from (7.9a) that the first intermediate vector is $k_{n,1} = u_{n-1}$, and that the subsequent intermediate vectors can be computed recursively since we have an explicit expression for $k_{n,i}$ in terms of u_{n-1} and the already computed $k_{n,j}$, $j < i$. If the method is implicit it will be tacitly assumed that the implicit relations are uniquely solvable. We finally mention that the intermediate vectors $k_{n,i}$ are approximations to $u(t_{n-1} + c_i\tau)$, but in general less accurate than the approximation u_n to $u(t_n)$.

Remark 7.4. For autonomous differential equations $u'(t) = f(u(t))$, where f does not explicitly depend on t, the relations (7.9) that define the Runge-Kutta step $(t_{n-1}, u_{n-1}) \to (t_n, u_n)$ take

the simpler form

$$k_i \;=\; u_{n-1} \,+\, \tau \sum_{j=1}^{s} a_{ij} f(k_j) \qquad (i = 1, 2, \ldots, s)\,, \tag{7.11a}$$

$$u_n \;=\; u_{n-1} \,+\, \tau \sum_{i=1}^{s} b_i f(k_i)\,. \tag{7.11b}$$

In order to achieve the simplest formulation, we have also written k_i instead of $k_{n,i}$, but it should be realized that the intermediate vectors k_i do depend on n. ◊

Example 7.5. The general explicit two-stage Runge-Kutta method can be represented by the tableau

$$\begin{array}{c|cc} c_1 & 0 & 0 \\ c_2 & a_{21} & 0 \\ \hline & b_1 & b_2 \end{array}$$

with $c_1 = 0$ and $c_2 = a_{21}$. The choice of the coefficients a_{21}, b_1, b_2 determines the method. For autonomous systems $u'(t) = f(u(t))$ the method reads

$$\left\{ \begin{array}{lcl} k_1 &=& u_{n-1}, \\ k_2 &=& u_{n-1} + \tau a_{21} f(k_1), \\ u_n &=& u_{n-1} + \tau b_1 f(k_1) + \tau b_2 f(k_2), \end{array} \right.$$

with intermediate vectors k_1 and k_2 changing from step to step—which can be emphasized by giving k_1, k_2 an extra subindex n, as in (7.9). The method can also be written, more compactly, as

$$u_n = u_{n-1} + \tau b_1 f(u_{n-1}) + \tau b_2 f\left(u_{n-1} + \tau a_{21} f(u_{n-1})\right). \tag{7.12}$$

If $a_{21} = 1$, $b_1 = b_2 = \frac{1}{2}$, we retrieve the explicit trapezoidal rule (7.7) with tableau

$$\begin{array}{c|cc} 0 & 0 & 0 \\ 1 & 1 & 0 \\ \hline & 1/2 & 1/2 \end{array}$$

Another popular choice is $a_{21} = \frac{1}{2}$, $b_1 = 0$, $b_2 = 1$, which leads to the *explicit midpoint rule* or *modified Euler method*, and corresponds to the tableau

$$\begin{array}{c|cc} 0 & 0 & 0 \\ 1/2 & 1/2 & 0 \\ \hline & 0 & 1 \end{array}$$

◊

7.4 Order of Consistency

In this and the following section we will prove convergence. To do this, we use the same overall strategy as in interpolation or integration whenever we had a subdivision of the domain and the global approximation is obtained by "gluing" together the approximations on each subinterval:

we first estimate the error in each piece of the subdivision, the local error, and then we sum all those up to estimate the global error.
We assume again that we take N steps with stepsize $\tau = T/N$ so that we obtain approximations $u_n \approx u(t_n)$ with $t_n = n\tau$ ranging over the whole interval $[0, T]$. In each of these steps, bringing us from a point t_{n-1} to t_n, there will be some (small) error in approximating the exact solution $u(t)$. This corresponds to the local error. The global error is the error at the endpoint $u(t_N) - u_N$, and it is the cumulative effect of N of these local errors. Therefore, to get an upper bound for this accumulated global error, we not only need to have a bound for the local errors, but we also need to estimate how these local errors will propagate and affect the final result.
In this section we will only study bounds for the local errors. In the next section we will study how these local errors propagate and derive bounds for the global errors.
Consider a Runge-Kutta method, written in an abstract, compact way as

$$u_n \;=\; u_{n-1} + \tau\, \Phi_\tau(t_{n-1}, u_{n-1}, u_n)\,, \tag{7.13}$$

with an *increment function* Φ_τ. See, for instance, formula (7.12) for explicit two-stage methods. While (7.13) is exact for the numerical solution, it is not for the exact solution. The local error is the error of the time-stepping scheme when applied to the *exact* solution, that is, replacing u_{n-1} resp. u_n by $u(t_{n-1})$ resp. $u(t_n)$ in (7.13). Namely, inserting the exact solution values into (7.13), we get

$$u(t_n) \;=\; u(t_{n-1}) + \tau\, \Phi_\tau(t_{n-1}, u(t_{n-1})) + \tau\, d_n\,, \tag{7.14}$$

with a residual τd_n called the *local error*.

Definition 7.6. *The method is* consistent *if* $\lim_{\tau\to 0} d_n = 0$, *uniformly for* $t_n \in [0, T]$.
The method is said to be consistent of order p *if*

$$d_n = \mathcal{O}(\tau^p) \quad \textit{(as } \tau \to 0\textit{, uniformly for } t_n \in [0, T]\textit{)}.$$

The local error $\tau d_n = \mathcal{O}(\tau^{p+1})$ can be nicely interpreted as the error introduced in one step: it is the difference between the exact solution at time t_n and the numerical approximation that would have been obtained if we had started with $u_{n-1} = u(t_{n-1})$ at time t_{n-1}. This is illustrated in Figure 7.2 for the scalar case ($m = 1$).

Example 7.7. In this example we derive the order of consistency for Euler's method (7.4). The increment function for this method is simply given by $\Phi_\tau(t, v) = f(t, v)$. The local errors are therefore given by

$$\tau d_n \;=\; u(t_n) - u(t_{n-1}) - \tau f(t_{n-1}, u(t_{n-1}))\,.$$

We assume that f is continuously differentiable on $[0, T] \times \mathbb{R}^m$. This implies that u is twice continuously differentiable on $[0, T]$. Taylor expansion of $u(t_n) = u(t_{n-1} + \tau)$ yields

$$u(t_n) = u(t_{n-1}) + \tau u'(t_{n-1}) + R(t_{n-1}, \tau),$$

where the norm of the remainder term is bounded by[1]

$$\|R(t_{n-1}, \tau)\| \le \tfrac{1}{2}\tau^2 \max_{t\in[t_{n-1},t_n]} \|u''(t)\|.$$

[1] see Appendix section B.3

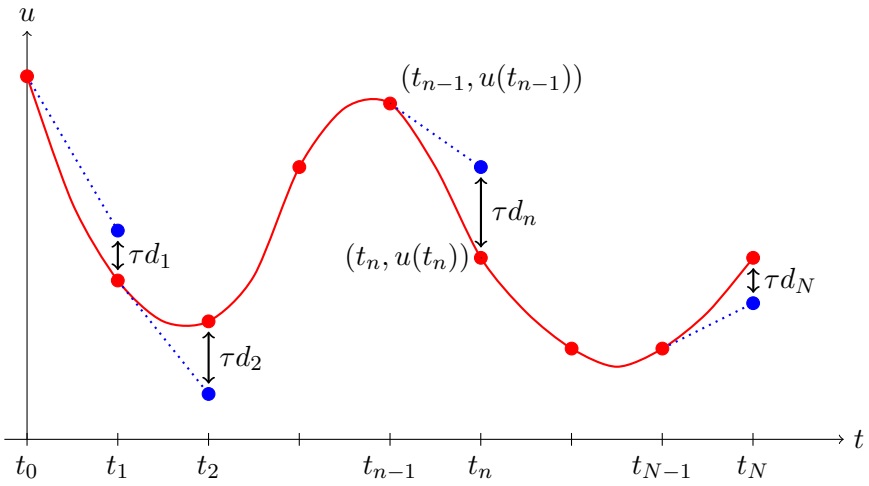


Figure 7.2: Illustration of the local errors τd_n for the scalar case ($m = 1$). The red curve is the graph of the exact solution. The blue dots are obtained by doing a single step with the numerical method starting from a point $(t_{n-1}, u(t_{n-1}))$ on the exact solution curve.

Since $f(t_{n-1}, u(t_{n-1})) = u'(t_{n-1})$, it follows that

$$\|d_n\| \leq C\tau \quad \text{with } C = \tfrac{1}{2} \max_{t\in[0,T]} \|u''(t)\|.$$

The method is therefore consistent of order $p = 1$. ◊

Order Conditions. To obtain a Runge-Kutta method with order of consistency p for arbitrary, smooth solutions u—simply called a *method of order p*—there are a number of algebraic conditions on the coefficients of the method that must be satisfied. When deriving these so-called *order conditions* it will always be assumed that f is p times continuously differentiable on $[0, T] \times \mathbb{R}^m$. The way to derive the order conditions is by making Taylor expansions in powers of τ of one step of the numerical method and the exact solution.

Example 7.8 (Conditions for order one and two)**.** In the derivation of order conditions we will confine ourselves to autonomous differential equations $u'(t) = f(u(t))$. As we have seen in Exercise 39 of Assignment 12, this does not lead to a loss of generality.
We will only consider the local error in the first step, setting $n = 1$, that is, we study $\tau d_1 = u(t_1) - u_1$ starting with $u(0) = u_0$. Again this does not lead to a loss of generality since the analysis is essentially the same for all local errors τd_n.
We want to find the order p such that $u(t_1) - u_1 = \mathcal{O}(\tau^{p+1})$, and we consider here the conditions for having $p = 1$ and $p = 2$.
For this, it is convenient to rewrite one step of a Runge-Kutta method as

$$u_1(\tau) = u_0 + \tau \sum_{i=1}^{s} b_i F_i,$$

with

$$F_i = f(k_i), \quad k_i = u_0 + \tau \sum_{j=1}^{s} a_{ij} F_j.$$

Differentiating with respect to τ we have

$$u_1'(\tau) = \sum_{i=1}^{s} b_i F_i + \tau \sum_{i=1}^{s} b_i F_i', \tag{7.15}$$

and, evaluating the last expression at $\tau = 0$,

$$u_1'(0) = \sum_{i=1}^{s} b_i F_i = \sum_{i=1}^{s} b_i f(k_i(0)) = \left(\sum_{i=1}^{s} b_i \right) f(u_0).$$

Since $u(t_1) = u(\tau) = u_0 + \tau u'(0) + \mathcal{O}(\tau^2) = u_0 + \tau f(u_0) + \mathcal{O}(\tau^2)$ and $u_1(\tau) = u_0 + \tau u_1'(0) + \mathcal{O}(\tau^2)$, the method has order at least $p = 1$ if and only if

$$\sum_{i=1}^{s} b_i = 1. \tag{7.16}$$

For $p = 2$, we proceed similarly. Differentiating (7.15) once again gives

$$u_1''(\tau) = 2 \sum_{i=1}^{s} b_i F_i' + \tau \sum_{i=1}^{s} b_i F_i''.$$

We also have

$$F_i' = f'(k_i) k_i', \quad k_i' = \sum_{j=1}^{s} a_{ij} F_j + \tau \sum_{j=1}^{s} a_{ij} F_j'.$$

Let $c_i = \sum_{j=1}^{s} a_{ij}$ (which ensures invariance under autonomization, see Exercise 39 in Assignment 12). Evaluating k_i' at 0 (and remembering that $F_j(0) = f(u_0)$), we have

$$k_i'(0) = \sum_{j=1}^{s} a_{ij} f(u_0) = c_i f(u_0).$$

Consequently

$$F_i'(0) = c_i f'(u_0) f(u_0)$$

and

$$u_1''(0) = 2 \sum_{i=1}^{s} b_i c_i f'(u_0) f(u_0).$$

To have order (at least) $p = 2$, we need therefore that *both* (7.16) and

$$\sum_{i=1}^{s} b_i c_i = \frac{1}{2} \tag{7.17}$$

are fulfilled.

$\Diamond$

Table 7.2: Order conditions of Runge-Kutta methods for $p = 1, 2, 3, 4$, with the summations from 1 to s, and $c_i = \sum_j a_{ij}$.

Order p	Order conditions	
1	$\sum_i b_i = 1$	
2	$\sum_i b_i c_i = 1/2$	
3	$\sum_i b_i c_i^2 = 1/3$	$\sum_{i,j} b_i a_{ij} c_j = 1/6$
4	$\sum_i b_i c_i^3 = 1/4$	$\sum_{i,j} b_i c_i a_{ij} c_j = 1/8$
	$\sum_{i,j} b_i a_{ij} c_j^2 = 1/12$	$\sum_{i,j,k} b_i a_{ij} a_{jk} c_k = 1/24$

For increasing order p, the derivation of these order conditions does become rather technical, and we report them in Table 7.2.

The order conditions in this table are cumulative. So, for order $p = 4$ there are in total 8 conditions on the coefficients. The number of order conditions quickly increases for higher orders. For example, to have $p = 5$ we get 9 new conditions (giving in total 17 conditions); then, for order $p = 6$ there are 20 additional conditions (in total 37 conditions). Nevertheless, many high-order methods have been constructed which are used in popular codes.

For explicit methods it can be shown (see Corollary 8.8 in the next chapter) that $p \leq s$. Moreover, it is known that $p = s$ can be achieved with explicit methods only for s up to four.

Example 7.9. Two well-known and widely-used explicit Runge-Kutta methods are given by the following tableaus, where we have suppressed the display of all values $a_{ij} = 0$ with $j \geq i$:

a)
$$\begin{array}{c|ccc} 0 & & & \\ 1/3 & 1/3 & & \\ 2/3 & 0 & 2/3 & \\ \hline & 1/4 & 0 & 3/4 \end{array}$$

b)
$$\begin{array}{c|cccc} 0 & & & & \\ 1/2 & 1/2 & & & \\ 1/2 & 0 & 1/2 & & \\ 1 & 0 & 0 & 1 & \\ \hline & 1/6 & 1/3 & 1/3 & 1/6 \end{array} \qquad (7.18)$$

Method (7.18.a) with $p = s = 3$ is *Heun's third-order method*. Method (7.18.b) is known as the *classical Runge-Kutta method* and has $p = s = 4$. It can be seen as a generalization of the Simpson quadrature rule. ◊

Illustration. Consider again the Lotka-Volterra model (7.8), but now solved with the methods (7.18.a) and (7.18.b) with varying number of steps N and $\tau = T/N$. The errors $\|u(t_N) - u_N\|_2$ at the endpoint $t_N = T$ are given in Table 7.3. These errors are to be compared with those in Table 7.1 for Euler's method (7.4) and the explicit trapezoidal rule (7.7). We see, in particular for the fourth-order classical Runge-Kutta method (7.18.b), that there is a spectacular improvement in the results compared to the low-order methods (7.4) and (7.7).

Table 7.3: Error $\|u(t_N) - u_N\|_2$ for the Lotka-Volterra problem (7.8) at time $T = 15$, with number of steps $N = 100, 200, 400, \ldots$ and $\tau = T/N$.

N	100	200	400	800	1600	3200
Err. (7.18.a)	$6.8 \cdot 10^{-3}$	$8.2 \cdot 10^{-4}$	$1.0 \cdot 10^{-4}$	$1.3 \cdot 10^{-5}$	$1.6 \cdot 10^{-6}$	$2.0 \cdot 10^{-7}$
Err. (7.18.b)	$9.7 \cdot 10^{-5}$	$8.7 \cdot 10^{-6}$	$6.3 \cdot 10^{-7}$	$4.2 \cdot 10^{-8}$	$2.7 \cdot 10^{-9}$	$1.7 \cdot 10^{-10}$

7.5 Order of Convergence

As starting point for deriving convergence results we consider (7.13) and (7.14), which we repeat here for convenience:

$$u_n = u_{n-1} + \tau\, \Phi_\tau(t_{n-1}, u_{n-1})\,, \tag{7.19a}$$
$$u(t_n) = u(t_{n-1}) + \tau\, \Phi_\tau(t_{n-1}, u(t_{n-1})) + \tau\, d_n\,, \tag{7.19b}$$

for $n = 1, 2, \ldots, N$ and $\tau = T/N$. The local errors τd_n provide a measure for the accuracy of the method, but we are actually interested in the *global errors* $e_n = u(t_n) - u_n$.

Definition 7.10. *Method (7.19a) is called convergent of order p for the solution u if*

$$e_n = \mathcal{O}(\tau^p) \quad \textit{(as } \tau \to 0\textit{, uniformly for } t_n \in [0, T]\textit{)}.$$

Our strategy is now to use the analysis on the local error (consistency) as from the previous section to draw conclusions on the global error (convergence). We note however that, since u_n depends on u_{n-1}, u_{n-1} depends on u_{n-2}, and so on, we need to be able to control the propagation of local errors over the time steps. This is expressed by the following property. Without loss of generalization, we state it for explicit methods, because, for the implicit function theorem, we can rewrite an implicit method as an explicit one.

Definition 7.11. *Consider a one-step method and its perturbed version:*

$$u_n = u_{n-1} + \tau \Phi_\tau(t_{n-1}, u_{n-1})$$
$$v_n = v_{n-1} + \tau\, (\Phi_\tau(t_{n-1}, u_{n-1}) + \delta_n)\,,$$

with $v_0 = u_0 + \delta_0$. The method is null-stable *if there exists $\overline{\tau} > 0$ and $C < \infty$ such that, for all $\varepsilon > 0$, $\tau \in (0, \overline{\tau}]$ and $\|\delta_n\| \le \varepsilon$,*

$$\|v_n - u_n\| \le C\varepsilon, \quad \textit{for all } 0 \le n \le N.$$

In the previous definition, the constant C is independent of τ but it can depend on the final time T. When setting $\delta_n = d_n$ with d_n as in (7.14), we can reinterpret the consistency error as a perturbation, and it is not hard to see that consistency and null-stability imply convergence, as the next result shows.

Theorem 7.12. *A null-stable one-step method with consistency order p is convergent of order p. That is, for such a method there exists a constant $C < \infty$ such that, for all $u \in C^p([t_0, t_0 + T])$,*

$$\max_{0 \le n \le N} \|u(t_n) - u_n\| \le C\tau^p. \tag{7.20}$$

Proof. We set $\delta_n = d_n$ in (7.14). Because of the assumption on consistency, $\|d_n\| \le C_c\tau^p$. We set therefore $v_n := u(t_n)$ and $C_c\tau^p := \varepsilon$ in the definition of null-stability, which implies $\|u(t_n) - u_n\| \le C_{ns}\varepsilon \le C_{ns}C_c\tau^p$ and the claim follows with $C := C_{ns}C_c$.

□

Remark 7.13. The result above can be summarized as

$$\text{null stability} + \text{consistency} \Rightarrow \text{convergence}.$$

The reverse implication is also true, and the equivalence result is known as Lax–Richtmyer equivalence theorem.

The condition in Definition 7.11 is not very handy to be verified. The next result present a sufficient condition for null-stability which is easier to check in practice.

Proposition 7.14. *If the increment function $\Phi_\tau(t, v)$ is (globally) Lipschitz continuous with respect to v, namely there exists $L > 0$ such that*

$$\begin{cases} \|\Phi_\tau(t, \widetilde{v}) - \Phi_\tau(t, v)\| \le L\|\widetilde{v} - v\| \\ \text{for all } v, \widetilde{v} \in \mathbb{R}^m \text{ and all } t, \tau \text{ with } t, t + \tau \in [0, T] \text{ and } \tau \in (0, \overline{\tau}], \end{cases} \tag{7.21}$$

for some $\overline{\tau} > t_0$, then the one-step method is null-stable.

In the statement above, we have asked global rather than local Lipschitz continuity just for simplicity. To prove the proposition, we need the following auxiliary result.

Lemma 7.15 (Discrete Gronwall)**.** *Let $(\varphi_n)_{n\ge0}$, $(k_n)_{n\ge0}$ and $(p_n)_{n\ge0}$ be non-negative sequences and let $g_0 \ge 0$ such that*

$$\varphi_0 \le g_0, \quad \varphi_n \le g_0 + \sum_{s=0}^{n-1} p_s + \sum_{s=0}^{n-1} k_s\varphi_s, \text{ for } n \ge 1.$$

Then

$$\varphi_n \le \left(g_0 + \sum_{s=0}^{n-1} p_s\right) \exp\left(\sum_{s=0}^{n-1} k_s\right), \quad \text{for } n \ge 1.$$

We know go back to the proof of Proposition 7.14.

Proof. We use here the same notation as in Definition 7.11. We define $w_j := v_j - u_j,$, $j \ge 0$, which satisfies the following recursion:

$$w_j = w_{j-1} + \tau\big(\Phi_\tau(t_{j-1}, v_{j-1}) - \Phi_\tau(t_{j-1}, u_{j-1}) + \delta_j\big), \quad j \ge 1.$$

Using the Lipschitz assumption on the increment function, we have

$$\|w_j\| \le (1+\tau L)\|w_{j-1}\| + \tau\|\delta_j\| \le (1+\tau L)\|w_{j-1}\| + \tau\varepsilon,$$

which can be rewritten as

$$\|w_j\| - \|w_{j-1}\| \le \tau L\|w_{j-1}\| + \tau\varepsilon.$$

Summing the previous expression for $j = 1, \dots, n$, we have

$$\|w_n\| \le \|w_0\| + \tau L \sum_{j=1}^{n} \|w_{j-1}\| + \sum_{j=1}^{n} \tau\varepsilon \le \|w_0\| + \tau L \sum_{j=0}^{n-1} \|w_j\| + (T-t_0)\varepsilon,$$

where we remind that t_0 and T are the initial and final times, respectively. We can then apply the discrete Gronwall inequality, obtaining

$$\|w_n\| \le (\|w_0\| + (T-t_0)\varepsilon)\, e^{N\tau L} = (\|w_0\| + (T-t_0)\varepsilon)\, e^{(T-t_0)L},$$

for $0 \le n \le N$.

□

We can then use this sufficient criterion in Theorem 7.12.

Corollary 7.16. *A one-step method which is consistent of order p and has an increment function which is Lipschitz continuous as in (7.21) is convergent of order p.*

The following lemma shows that for any Runge-Kutta method, the Lipschitz condition for its increment function follows immediately from the Lipschitz condition for the function f.

Lemma 7.17. *Suppose the Lipschitz condition (7.2) with Lipschitz constant L holds for the function f. Consider a Runge-Kutta method (7.9), and let*

$$\alpha = \max_i \sum_j |a_{ij}|, \quad \beta = \sum_j |b_j|.$$

Then the increment function satisfies Lipschitz condition (7.21) for any pair $(\overline{L}, \overline{\tau})$ with

$$\overline{L} = \frac{\beta L}{1-\alpha L\overline{\tau}}, \qquad 0 < \overline{\tau} < \frac{1}{\alpha L}.$$

Proof. In order to prove Lipschitz condition (7.21), let arbitrary $v, \widetilde{v} \in \mathbb{R}^m$ be given and t, τ with $0 \le t < t+\tau \le T$. We need to find an upper bound for $\|\Phi_\tau(t,\widetilde{v}) - \Phi_\tau(t,v)\|$ in terms of $\|\widetilde{v} - v\|$.
As illustrated in Figure 7.3, we consider two 'parallel' steps of Runge-Kutta method (7.9) with stepsize τ, starting at (t,v) and $(t,\widetilde{v})$, respectively. For the first step $(t,v) \to (t+\tau, w)$ we denote the intermediate vectors by k_i, $i = 1, \dots, s$, so that

$$k_i = v + \tau \sum_j a_{ij} f(t + c_j\tau, k_j) \quad (i = 1, 2, \dots, s),$$
$$w = v + \tau \sum_i b_i f(t + c_i\tau, k_i).$$

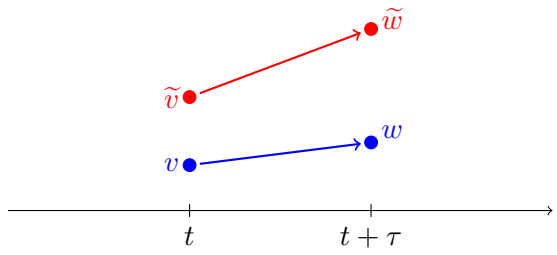


Figure 7.3: Two 'parallel' Runge-Kutta steps $(t, v) \to (t + \tau, w)$ and $(t, \widetilde{v}) \to (t + \tau, \widetilde{w})$.

For the second (parallel) step $(t, \widetilde{v}) \to (t + \tau, \widetilde{w})$ we have a similar set of relations involving $\widetilde{v}, \widetilde{w}$ and intermediate vectors $\widetilde{k}_i$.
The increments corresponding to both steps are given by

$$\Phi_\tau(t, v) = \frac{1}{\tau}(w - v), \quad \Phi_\tau(t, \widetilde{v}) = \frac{1}{\tau}(\widetilde{w} - \widetilde{v}).$$

We use the following abbreviations,

$$\Delta v = \widetilde{v} - v, \quad \Delta w = \widetilde{w} - w, \quad \Delta k_i = \widetilde{k}_i - k_i, \quad \Delta\Phi = \Phi_\tau(t, \widetilde{v}) - \Phi_\tau(t, v).$$

Then

$$\|\Delta k_i\| \le \|\Delta v\| + L\tau \sum_j |a_{ij}| \cdot \|\Delta k_j\| \le \|\Delta v\| + \alpha L\tau \max_j \|\Delta k_j\|, \tag{7.22a}$$

$$\|\Delta w - \Delta v\| \le L\tau \sum_i |b_i| \cdot \|\Delta k_i\| \le \beta L\tau \max_j \|\Delta k_j\|. \tag{7.22b}$$

Inequality (7.22a) gives

$$(1 - \alpha L\tau) \max_j \|\Delta k_j\| \le \|\Delta v\|.$$

Hence, if we choose $\overline{\tau} > 0$ so small that $1 - \alpha L\tau_* > 0$, then for $\tau \in (0, \overline{\tau}]$ we have

$$\max_j \|\Delta k_j\| \le \frac{1}{1 - \alpha L\tau} \|\Delta v\|.$$

Combined with inequality (7.22b) this gives

$$\|\Delta\Phi\| = \frac{1}{\tau}\|\Delta w - \Delta v\| \le \beta L \max_j \|\Delta k_j\| \le \frac{\beta L}{1 - \alpha L\tau} \|\Delta v\|,$$

from which the result follows. □

We thus see that for any Runge-Kutta method, consistency of order p implies convergence of order p, under the assumption of the Lipschitz condition (7.2) for the function f. In this case, the order conditions we have seen before allow us to conclude the order of convergence, too.

7.6 Stepsize Selection*

To solve an initial value problem numerically, we can take a method and apply it on $[0, T]$ with constant stepsizes $\tau = T/N$. Repeating the computation with one or more other stepsizes, for instance with $\tau/2, \tau/4, ...$, makes it possible to estimate the error (and improve accuracy via Richardson extrapolation).
However, to get an efficient scheme it is important to adapt the stepsizes to match the variations in the solution. This is similar to the case of quadrature formulas, but with ODEs we have in general no idea in advance how the solution will behave, or where it will exhibit rapid variations.
In the following, let *Tol* be a tolerance specified by the user. We will try to select the stepsizes such that the (estimated) local errors are bounded by this number *Tol*.
Consider an attempted step from t_{n-1} to $t_n = t_{n-1} + \tau_n$ with stepsize τ_n. Suppose the method has order p and we have an available estimate E_n for the norm of the local error,

$$E_n \;=\; \gamma_n \, \tau_n^{\hat{p}+1} \;+\; \mathcal{O}(\tau_n^{\hat{p}+2}) \,,$$

with $\hat{p} \approx p$ and some unknown constant $\gamma_n > 0$. Here $\hat{p} = p$ if E_n is an estimate of the genuine local error of the method, but often the estimate E_n is quite rough and $\hat{p}$ may be less than p. For example, the estimate may be obtained by comparing u_n with the result $\hat{u}_n$ obtained from a lower-order method, say of order $\hat{p} = p - 1$, embedded in the primary method, similar as for quadrature methods.
Having the estimate E_n available, two cases can occur: $E_n > Tol$ or $E_n \le Tol$. In the first case we decide to reject this step and to redo it with a smaller stepsize $\tau_n = \tau_{new}$, where we aim at $E_{new} = Tol$. In the second case the step is accepted, and we continue the integration with $\tau_{n+1} = \tau_{new}$ for the new step from t_n to t_{n+1}, again aiming at $E_{new} = Tol$.
The constant γ_n is not known, but we will have approximately $E_n = \gamma_n \, \tau_n^{\hat{p}+1}$ and $E_{new} = \gamma_n \, \tau_{new}^{\hat{p}+1}$. If we require $E_{new} = Tol$, we can eliminate γ_n to arrive at

$$\tau_{new} \;=\; r \, \tau_n \,, \qquad r \;=\; \big(Tol \,/\, E_n \big)^{1/(\hat{p}+1)} \,. \tag{7.23}$$

Because rough estimates are used, the expression for the new stepsize found in most codes has the form

$$\tau_{new} \;=\; \min \big(r_{max}, \max \left(r_{min}, \vartheta r \right) \big) \cdot \tau_n \,, \tag{7.24}$$

where r_{max} and r_{min} are a maximal and minimal growth ratio, and $\vartheta < 1$ serves to make the estimate conservative so as to avoid repeated rejections. Typical values are $\vartheta \in [0.7, 0.9]$, $r_{min} \in [0.1, 0.5]$ and $r_{max} \in [1.5, 10]$.

Example 7.18. ($p = 2$, $\hat{p} = 1$). As an example we will provide the explicit trapezoidal rule (7.7) with a simple stepsize control and illustrate the resulting solver. Since the explicit Euler result $u_n^* = u_{n-1} + \tau_n f(t_{n-1}, u_{n-1})$ is available already in a step with (7.7), $\tau = \tau_n$, we can use

$$E_n \;=\; \| \, u_n - u_n^* \, \| \tag{7.25}$$

as an estimator for the norm of the local error. Notice that for $u_{n-1} = u(t_{n-1})$ this estimator satisfies

$$E_n \;=\; \tfrac{1}{2} \tau_n^2 \, \| u''(t_{n-1}) \| \;+\; \mathcal{O}(\tau_n^3) \,.$$

This provides an accurate estimator for the local error of Euler's method, but we will use this nevertheless for the second-order method, where we expect it to give an upper bound. Then, in formula (7.23) we have $\hat{p} = 1$. By choosing an appropriate norm for computing E_n and by making a choice for the parameters in (7.24) the stepsize control can now be used.
As an illustration, we apply this method with error estimator to the problem

$$\begin{cases} v'(t) = 1 + v(t)^2 w(t) - 4v(t)\,, & v(0) = 1.01\,, \\ w'(t) = 3v(t) - v(t)^2 w(t)\,, & w(0) = 3\,, \end{cases} \tag{7.26}$$

with end time $T = 20$. It is a simplified chemical model with two chemical species. With these initial values the variation of the solution is small at first. Around time $t = 8$ and $t = 15$ there are large variations, and the stepsize should then become smaller.

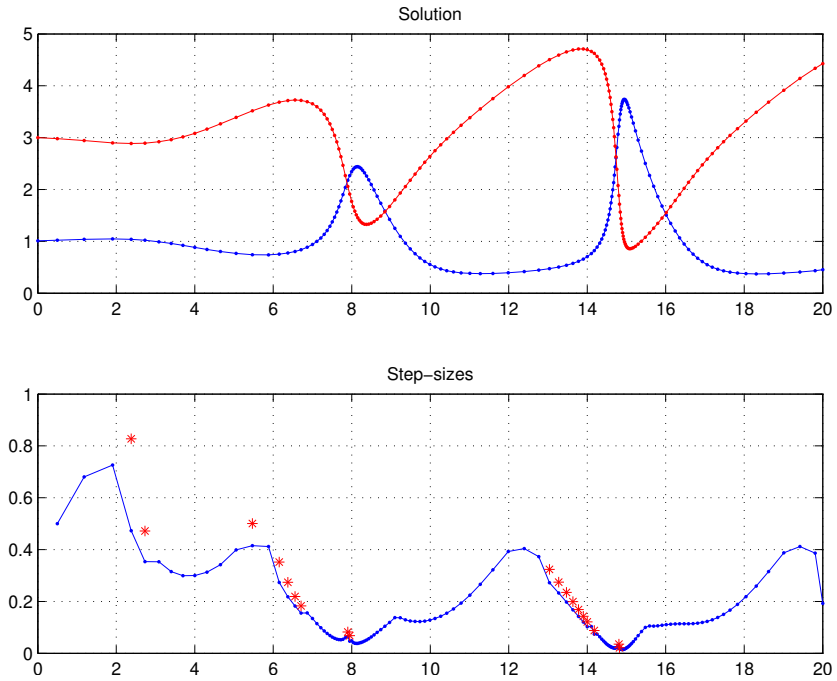


Figure 7.4: Variable stepsizes for (7.26). Top panel: the two solution components versus time. Bottom panel: accepted and rejected ($*$) stepsizes.

The results are shown in Figure 7.4, with plots of the solution components $v_n \approx v(t_n)$, $w_n \approx w(t_n)$ versus t_n, and also the stepsizes τ_n versus t_n. These results have been obtained with $Tol = 10^{-2}$, $\vartheta = 0.85$, $r_{min} = 0.5$, $r_{max} = 1.5$ and the maximum norm on $\mathbb{R}^2$. As we see in the figure, there are quite a number of rejections, so here it might be better to take ϑ slightly smaller, say 0.8. More importantly, for smaller tolerances a larger number of steps will be taken. Better efficiency will then be obtained using methods with a higher order. ◊

It should be stressed that *Tol* is just a target value for the local errors; it is *not* a guaranteed upper bound for the global errors. When an ODE code (no matter how sophisticated) is used

for a new class of problems, some representative problems should first be tried with different values of *Tol* to get a feeling of the actual global accuracy.

Remark 7.19. For the first step, a suitable starting stepsize is required, which must somehow capture the initial solution variation. Sophisticated ODE codes do this automatically. It can be based on the norms of $\|u_0\|$ and $\|f(t_0, u_0)\|$, by requiring that an Euler step $u_0 + \tau f(t_0, u_0)$ does not differ more than a certain percentage from u_0.
Further we note that there are many variants for the stepsize selection. For instance, we may look at relative errors. Also, instead of $E_n \lesssim Tol$ one can aim at having $E_n \lesssim (\tau_n/T)\, Tol$, which is called error control per unit step. That would be similar to the strategy described in Section 6.6 for quadrature methods with automatic partitioning. ◊

Chapter 8

Stiff Initial Value Problems and Stability

In practice, initial value problems (7.1) with high dimension m appear very often. Usually such problems are stiff. In this chapter the concept of stiffness will be discussed. As we will see, for stiff problems numerical methods should have suitable stability properties, and implicit Runge-Kutta methods (7.9) then become important.
It is important to understand that the concept of convergence seen in the previous chapter and the one of stability as we will see here are two separate things. While the notion of convergence is about the behavior of the numerical approximation as the time step size goes to 0, stability is about the behavior for a finite time step size. A numerical method can be of high order but unstable, and as such perform poorly for stiff problems unless the time step size is taken to be very small. In other words, when looking at the error of the method in dependence of the time step size, the pre-asymptotic behavior before we observe convergence can be very long.

8.1 Motivation

From the previous chapter, the usefulness of implicit methods is not clear. For instance, both explicit and implicit Euler are of order one, so implicit methods do not bring advantages in convergence properties, in general. There is an obvious disadvantage: being implicit, implicit methods take more work per step, due to the need of solving a possibly nonlinear system of equations. In this section, we see that implicit methods do bring advantages for an important class of initial value problems, namely stiff problems.

Example 8.1. As an illustration, consider the simple linear problem

$$\begin{aligned} v'(t) &= \lambda_1 v(t) - \lambda_2 w(t), & v(0) &= v_0, \\ w'(t) &= \lambda_2 w(t), & w(0) &= w_0, \end{aligned} \tag{8.1}$$

with $t \in [0,1]$, $v_0 = 1$, $w_0 = 0.1$ and $\lambda_2 \ll \lambda_1 = -1$. The differential equation is of the form $u'(t) = Au(t)$ with a 2×2 matrix A and λ_1, λ_2 are the eigenvalues of A. The exact solution is

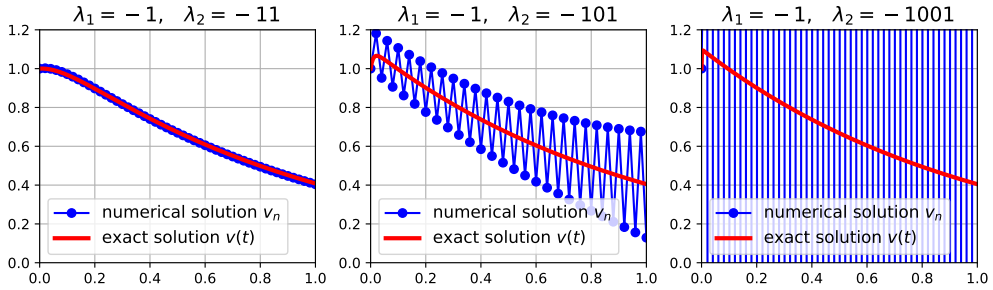


Figure 8.1: Results v_n for the explicit Euler method with $\tau = \frac{1}{50}$ and $\lambda_2 = -11$ (left), $\lambda_2 = -101$ (middle), $\lambda_2 = -1001$ (right). The red curve is the graph of the exact solution.

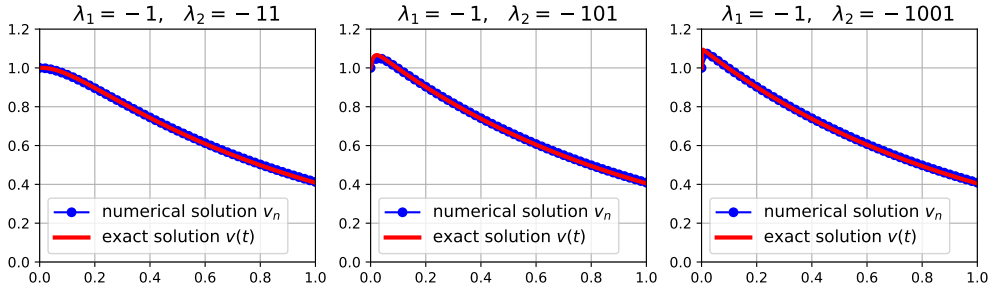


Figure 8.2: Results v_n for the implicit Euler method with $\tau = 1/50$ and $\lambda_2 = -11$ (left), $\lambda_2 = -101$ (middle), $\lambda_2 = -1001$ (right). The red curve is the graph of the exact solution.

given by

$$v(t) \;=\; \Big(v_0 - \tfrac{\lambda_2}{\lambda_1-\lambda_2} w_0\Big) e^{\lambda_1 t} + \Big(\tfrac{\lambda_2}{\lambda_1-\lambda_2} w_0\Big) e^{\lambda_2 t}\,, \qquad w(t) \;=\; w_0 e^{\lambda_2 t}\,. \tag{8.2}$$

Since $\lambda_2 \ll -1$, the terms with $e^{\lambda_2 t}$ become negligible after a short while, but we will see that having $\lambda_2 \ll -1$ will cause difficulties for explicit methods.
In Figure 8.1 the numerical approximations v_n for the first solution component v are plotted versus t_n for the explicit Euler method with stepsize $\tau = \frac{1}{50}$ and with λ_2 becoming more and more negative. It is clear from these pictures that the explicit Euler method cannot be used with this stepsize if $|\lambda_2|$ gets too large.
The behaviour of the implicit Euler method is very different. Figure 8.2 contains the same plots, again with $\lambda_1 = -1$ fixed and λ_2 becoming more and more negative, but here the numerical approximations stay close to the exact solution no matter how large $|\lambda_2|$ becomes. ◊

To understand the different behaviour of the explicit and implicit Euler method observed in this example, let us consider an arbitrary linear problem

$$u'(t) = A\,u(t), \quad u(0) = u_0\,, \tag{8.3}$$

with constant matrix $A \in \mathbb{R}^{m\times m}$. Assume the matrix A is diagonalizable,

$$A = V\Lambda V^{-1}\,, \qquad \Lambda = \mathrm{diag}(\lambda_i)\,. \tag{8.4}$$

Here the $\lambda_i \in \mathbb{C}$ are the eigenvalues of A and the i-th column of $V \in \mathbb{C}^{m\times m}$ contains the corresponding eigenvector. Setting $w(t) = V^{-1}u(t)$, we have $w'(t) = \Lambda w(t)$, that is $w_i'(t) = \lambda_i w_i(t)$, and therefore $w(t) = \exp(t\Lambda)w(0)$ where $\exp(t\Lambda) = \mathrm{diag}(\exp(t\lambda_i))$. In terms of u we therefore have

$$u(t) \;=\; V \exp(t\Lambda)\, V^{-1}u_0\,. \tag{8.5}$$

It is clear that $u(t)$ remains bounded as $t \to \infty$ for arbitrary initial values $u_0 \in \mathbb{R}^m$ if and only if

$$\mathrm{Re}\,\lambda_i \le 0 \qquad \text{for all } \; 1 \le i \le m\,. \tag{8.6}$$

Application of the explicit Euler method to (8.3) gives a different picture. We then have $u_n = (I + \tau A)u_{n-1}$, and since $I + \tau A = V(I + \tau\Lambda)V^{-1}$ this leads to

$$u_n \;=\; V(I + \tau\Lambda)^n\, V^{-1}u_0\,. \tag{8.7}$$

To have boundedness of u_n as $n \to \infty$ for arbitrary initial values $u_0 \in \mathbb{R}^m$ we now obtain the condition

$$|1 + \tau\lambda_i| \;\le\; 1 \qquad \text{for all } \; 1 \le i \le m\,. \tag{8.8}$$

On the other hand, for the implicit Euler method we get $u_n = (I - \tau A)^{-1}u_{n-1}$, which gives

$$u_n \;=\; V(I - \tau\Lambda)^{-n}\, V^{-1}u_0\,. \tag{8.9}$$

To have boundedness of u_n as $n \to \infty$ for arbitrary initial values $u_0 \in \mathbb{R}^m$ we now have the condition

$$|1 - \tau\lambda_i|^{-1} \;\le\; 1 \qquad \text{for all } \; 1 \le i \le m\,. \tag{8.10}$$

Returning to our example, the matrix A for (8.1) has eigenvalues $\lambda_1 = -1$ and $\lambda_2 \ll -1$. Therefore, the boundedness condition (8.8) for the explicit Euler method reads $\tau|\lambda_2| \le 2$, and if this is not satisfied we can get exponential growth of the solutions, $\|u_n\| \sim C\rho^n$, with a growth factor $\rho = \tau|\lambda_2| - 1 > 1$. In Figure 8.1 this is $\rho = 1.02$ for the middle panel, and $\rho = 19.02$ for the right panel. So, even though the large negative eigenvalue λ_2 hardly influences the solution, it causes numerical difficulties with the explicit Euler method. In contrast to this, for the implicit Euler method the boundedness condition (8.10) is clearly satisfied for any stepsize $\tau > 0$. Note that these observations do not mean that explicit Euler does not work at all for this kind of problems, but only that, in order to have meaninful results, we need to choose the time step size for explicit Euler to be very small.

Stiff problems. Stiffness is not a precisely defined mathematical concept, but rather an operational one. In general, stiff problems are considered to be those for which implicit methods perform (much) better than explicit ones. Trying to make things a bit more rigorous, here we give a *possible* definition of stiffness. Consider an ODE system written as

$$\begin{aligned} u'(t) &= Au(t) + \varphi(t), \\ u(0) &= u_0, \end{aligned}$$

where the matrix $A \in \mathbb{R}^{m\times m}$ is diagonalizable. Then we can say that the problem is stiff if:

(i) for all eigenvalues λ_i of A, $i = 1, \dots, m$, $\mathrm{Re}\lambda_i < 0$;

(ii) $\frac{\max_{i=1,\dots,m} \mathrm{Re}\lambda_i}{\min_{i=1,\dots,m} \mathrm{Re}\lambda_i} \ll 1$.

The condition (i) means that the dynamics described by the ODE is damping, and (ii) hints on the presence of slow/fast dynamics, or in other words, different time scales. The quantity $\frac{\max_{i=1,\dots,m} \mathrm{Re}\lambda_i}{\min_{i=1,\dots,m} \mathrm{Re}\lambda_i}$ is sometimes referred to as stiffness quotient. When the right-hand side of the ODE is a nonlinear function $f(t, u)$ of u, we can apply the concept above once we consider a linearization of the right-hand side. In that case, the matrix A will correspond to the Jacobian matrix of f around a point.
The idea behind the fact that explicit methods in general work worse than implicit ones for stiff problems is that, although the exact dynamics is dominated by the long time scales (eigenvalues with smaller real part *in absolute value*), the time step of the numerical method has to be adjusted to the fast time scales (eigenvalues with real part large in absolute value).

8.2 Absolute Stability and Stability Regions

To find methods suitable for stiff problems we will first look at the simple test equation

$$\begin{aligned} u'(t) &= \lambda u(t) \\ u(0) &= 1, \end{aligned} \tag{8.11}$$

with $\lambda \in \mathbb{C}$. The reason behind this is that, once we linearize the right-hand side of an ODE, we end up with (8.3), which has, up to a change of basis, solutions which decay exponentially, see (8.5). We use this model problem to study the stability of numerical methods. This way of proceeding is rather common in mathematics, namely simplifying an original problem to allow rigorous study but with a simplification that still retains the main features we are interested in. Problem (8.11) is usually referred to as *Dahlquist test equation*.
Application of a Runge-Kutta method to the test equation will give

$$u_n \;=\; R(\tau\lambda)\, u_{n-1}, \tag{8.12}$$

with a function R determined by the coefficients of the method. We will see that R is a rational function and is called the *stability function* of the method.
If we now compare the exact to the analytical solution at the points used for the time discretization, we see that

$$\begin{aligned} u(t_n) = e^{\lambda t_n} = e^{\lambda n\tau} = (e^{\tau\lambda})^n, \\ u_n = (R(\tau\lambda))^n \end{aligned}$$

(we remind that we set $u(0) = 1$ and we assumed a constant time step size such that $t_n = n\tau$). When $\mathrm{Re}\lambda < 0$ the exact solution decays exponentially in time. Therefore, if we want the numerical solution to reproduce the same physics in the sense of also giving a decaying solution, we need $|R(\tau\lambda)| < 1$. This leads to the following definition, where we set $z := \tau\lambda$.

Definition 8.2. *The set*

$$\mathcal{S} = \{z \in \mathbb{C} : |R(z)| < 1\} \tag{8.13}$$

is called the region of absolute stability or simply the stability region of a the method.

Definition 8.3. *A method is called* absolutely stable *if its region of absolute stability includes the left half-plane, that is, if*

$$\mathbb{C}^- := \{z \in \mathbb{C} : \operatorname{Re} z < 0\} \subseteq \mathcal{S}.$$

A-stability expresses the fact that, even for $|\lambda| \to \infty$ (with negative real part), the method is stable for any choice of the time step size. A method that A-stable is also called *unconditionally stable*, it is called *conditionally stable* otherwise.

Example 8.4. Applying the explicit Euler method to the Dahlquist test equation (8.11), we have

$$u_n = u_{n-1} + \tau\lambda u_{n-1} = (1 + \tau\lambda)u_{n-1}.$$

Its stability function is therefore $R(z) = 1 + z$. The method is stable for $|R(z)| = |1 + z| < 1$, so the stability region for explicit Euler is the part of the complex plane inside the circle centered at -1 and with radius 1. So, if $|\lambda|$ is large, we need a very small time step size τ to still fall in the stability region.

For implicit Euler, application of the method to the test equation gives

$$u_n = (1 - z)^{-1}u_{n-1},$$

so $R(z) = (1 - z)^{-1}$. The stability region is the part of the complex plane outside the circle centered in 1 and with radius 1. In particular, the implicit Euler method is A-stable. This explains why it gave us physically meaningful solutions in the example of Subsection 8.1, independently of how large the absolute value of the smallest eigenvalue was.

Explicit and implicit Euler are particular cases of the so-called θ-method: see Assignment 14 for details!
Figure 8.3 shows the stability regions of the Crank-Nicolson scheme (θ-scheme for $\theta = \frac{1}{2}$) and implicit Euler.

$\Diamond$

Relevance of the test equation. We started out in (7.1) with general initial value problems, possibly of high dimension, nonlinear and with difficult solutions. But the concepts of stability region and A-stability are related to the simple scalar test equation $u'(t) = \lambda u(t)$, for which we know already the solution $u(t) = e^{\lambda t}u(0)$.
As already hinted at the end of the previous subsection, it turns out that the linear scalar stability concepts are very relevant for general nonlinear systems. If we want to know how numerical solutions for a general ODE system $u'(t) = f(t, u(t))$ react to perturbations, a first step is to consider the local influence of small perturbations. But that can be studied for a linearized problem $u'(t) = Au(t)$, where A stands for a (frozen) Jacobian matrix of f. Then if A is diagonalizable we are led to the test problem $u'(t) = \lambda u(t)$, with λ eigenvalues of A.
In the literature there are many theoretical results in the opposite direction: starting from assumptions on the numerical method for the test equation (e.g. A-stability) one can show stability and convergence results for interesting classes of linear and nonlinear systems.

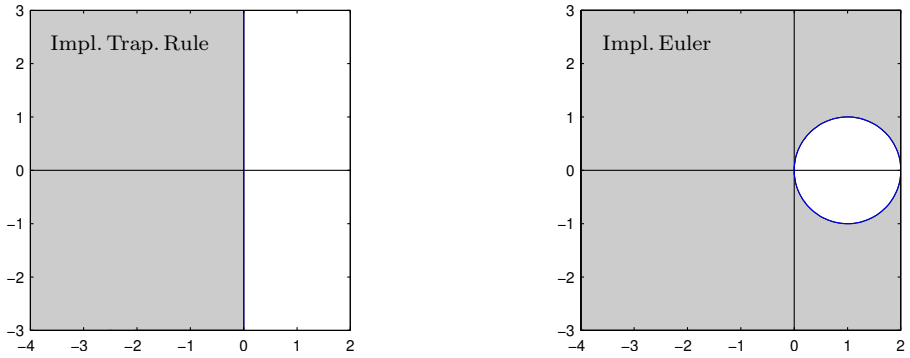


Figure 8.3: Stability regions $\mathcal{S}$ (shaded areas) for the implicit trapezoidal rule (left) and the implicit Euler method (right).

8.3 Stability of Runge–Kutta methods

In this section, we see a set of results about the stability of Runge–Kutta methods. We first tackle a quite general statement from which many useful properties will follow.

Theorem 8.5. *Consider a consistent Runge–Kutta method with s stages, with Butcher tableau*

$$\begin{array}{c|c} \boldsymbol{c} & \boldsymbol{A} \\ \hline & \boldsymbol{b}^\top \end{array},$$

where $\boldsymbol{c}, \boldsymbol{c} \in \mathbb{R}^s$ and $\boldsymbol{A} \in \mathbb{R}^{s\times s}$. Then:

(i) for every $z \in \mathbb{C}$ which is not an eigenvalue of $\boldsymbol{A}$, the stability function of the scheme is given by $R(z) = 1 + z\boldsymbol{b}^\top(\boldsymbol{I} - z\boldsymbol{A})^{-1}\mathbf{1}$, where $\boldsymbol{I} \in \mathbb{R}^{s\times s}$ is the identity matrix and $\mathbf{1} \in \mathbb{R}^s$ denotes the vector with value 1 in each entry;

(ii) $R(z) = e^z + \mathcal{O}(|z|^{p+1})$ for $z \to 0$, with p the order of consistency of the method.

Proof. We first address *(i)*. The application of the Runge–Kutta method to the Dahlquist test equation gives

$$k_i = u_{n-1} + \tau\lambda \sum_{j=1}^{s} a_{ij}k_j, \quad i = 1, \dots, s,$$
$$u_n = u_{n-1} + \tau\lambda \sum_{i=1}^{s} b_i k_i,$$

which can be written more compactly with vector notation as

$$(\boldsymbol{I} - z\boldsymbol{A})\boldsymbol{k} = u_{n-1}\mathbf{1}$$
$$u_n = u_{n-1} + z\boldsymbol{b}^\top\boldsymbol{k},$$

for $z = \tau\lambda$. If z is not an eigenvalue of $\boldsymbol{A}$, we can invert the matrix and obtain $\boldsymbol{k} = (\boldsymbol{I} - z\boldsymbol{A})^{-1}\mathbf{1}u_{n-1}$. Inserting this in the second equation gives

$$u_n = u_{n-1} + z\boldsymbol{b}^\top(\boldsymbol{I} - z\boldsymbol{A})^{-1}\mathbf{1}u_{n-1},$$

from which (i) follows.
The result in (ii) follows from the definition of order of consistency. Indeed, at the first time step the exact and numerical solutions to the test equation are, respecitely:

$$u(\tau) = e^{\lambda\tau} = e^z,$$
$$u_1 = R(z).$$

By definition of consistency order $|u(\tau) - u_1| = |e^z - R(z)| = \mathcal{O}(|z|^{p+1})$, therefore showing (ii). □

We now see some useful consequencies of the result above.

Corollary 8.6. *For an* explicit *Runge–Kutta method with s stages, $R(z) \in \mathbb{P}_s$.*

Proof. For an explicit method, the matrix $(\boldsymbol{I} - z\boldsymbol{A})$ is lower triangular with ones on the diagonal. Therefore, the claim follows by solving $(\boldsymbol{I} - z\boldsymbol{A})\boldsymbol{k} = u_{n-1}\mathbf{1}$ with forward substitution. □

Corollary 8.7. *There does* not *exist an explicit Runge–Kutta method which is A-stable.*

Proof. We have seen that, for an explicit method $R(z)$ is a polynomial. However, no polynomial on the complex numbers is bounded for $|z| \to \infty$, we required by A-stability. □

Corollary 8.8. *For an* explicit *Runge–Kutta method with s stages, the order is bounded by $p \leq s$.*

Proof. From claim (ii) in Theorem 8.5, $R(z) = e^z + \mathcal{O}(|z|^{p+1})$. On the other hand, we know that, being the method explicit, $R(z) \in \mathbb{P}^s$. Then, because of the approximation properties of polynomials, it must necessarily hold that $p + 1 \leq s + 1$, that is, $p \leq s$. □

We saw already in the previous chapter that having $p = s$ is possible for explicit methods with s up to four. The stability regions of stability functions for explicit methods are displayed in Figure 8.4.

Corollary 8.9. *For an* implicit *Runge–Kutta method with s stages, the stability function is a rational function $R(z) = \frac{P(z)}{Q(z)}$, with $P, Q \in \mathbb{P}_s$.*

Proof. See Assignment 14. □

Remark 8.10. *The claims above show that for explicit methods the stability function is a* polynomial *approximation to the exponential, while for implicit methods it is a* rational *approximation.*

Remark 8.11. *The results above also tell us that, while explicit methods* cannot *be A-stable, implicit method* can *(but do not need to!).*

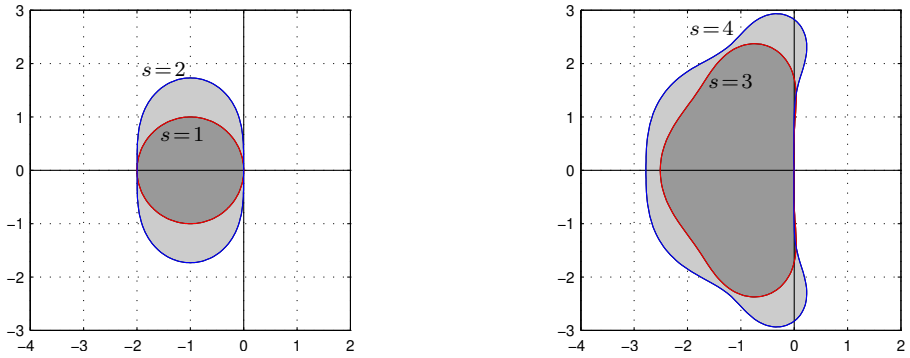


Figure 8.4: Stability regions $\mathcal{S}$ for the stability functions $R(z) = 1 + z + \frac{1}{2!}z^2 + \cdots + \frac{1}{s!}z^s$ of degree $s = 1, 2, 3, 4$.

8.4 A Semi-Discrete Initial-Boundary Value Problem*

Many stiff initial value problems for ODEs have their origin in partial differential equations (PDEs). We will illustrate this by considering the partial differential equation

$$\frac{\partial}{\partial t}w(x,t) \;=\; \gamma\frac{\partial^2}{\partial x^2}w(x,t) \;+\; \kappa\, w(x,t)\,(1 - w(x,t))\,, \tag{8.14}$$

where $\gamma = \frac{1}{10}$, $\kappa = 10$ and $w(x,t)$ stands for a concentration of a biological species that varies over space and time. The PDE combines the simple local ODE model $w' = \kappa w(1-w)$ for population growth with spatial diffusion $\partial w/\partial t = \gamma\partial^2 w/\partial x^2$. If $w(x,0) \in (0,1)$ in a given point x at time $t = 0$, then the ODE part will initiate a growth towards $w = 1$, but at the same time the diffusion will cause a flow from regions with high w towards regions with lower w. We consider equation (8.14) for $t \in [0,T]$ and $0 < x < 1$. Together with the initial condition $w(x,0) = w_0(x)$, we also impose the boundary conditions $w(0,t) = 0$, $w(1,t) = 0$,[1] giving an *initial-boundary value problem* for a PDE.

Now suppose we impose a spatial grid $x_j = jh$, $j = 1, \ldots, m$, with $h = \frac{1}{m+1}$ the mesh-width in space. Using different quotients we can approximate the second spatial derivatives of w as follows,

$$\frac{\partial^2}{\partial x^2}w(x,t) \;\approx\; \frac{1}{h^2}\Big(w(x-h,t) - 2w(x,t) + w(x+h,t)\Big).$$

Using this finite difference approximation at the grid points x_j, $j = 1, 2, \ldots, m$, we obtain the following ODE system,

$$u_j'(t) \;=\; \frac{\gamma}{h^2}\Big(u_{j-1}(t) - 2u_j(t) + u_{j+1}(t)\Big) \;+\; \kappa\, u_j(t)\,(1 - u_j(t))\,, \tag{8.15}$$

where the components $u_j(t)$ approximate the PDE solution at the grid points, $u_j(t) \approx w(x_j,t)$. The boundary conditions $w(0,t) = w(1,t) = 0$ for the PDE are taken into account by setting

[1] Actually, these boundary conditions are not very realistic for a biological model, but they are taken here for simplicity.

$u_0(t) = u_{m+1}(t) = 0$. The initial condition $w(x,0) = w_0(x)$ for the PDE is translated to an initial condition for the ODE system by imposing $u_j(0) = w_0(x_j)$, $j = 1, \ldots, m$.
It can be shown (beyond the scope of these notes) that the error in this approximation will be $\mathcal{O}(h^2)$ provided w is smooth. So we will take m large, implying that $h = 1/(m+1)$ is small. For $m = 100$, Figure 8.5 shows the solution of (8.15) at various times, starting with the initial profile

$$u_j(0) = w_0(x_j) = \frac{1}{10}\exp(-100(x_j - \tfrac{1}{4})^2) + \frac{1}{4}\exp(-100(x_j - \tfrac{3}{4})^2).$$

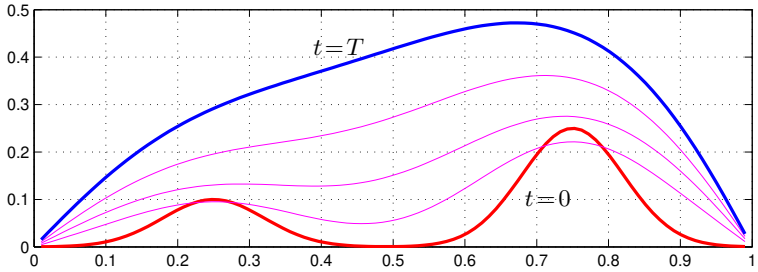


Figure 8.5: Time evolution for (8.15) with $m = 100$. The solution is plotted versus x at various times: at start time $t = 0$ (fat red line), at final time $t = T = \frac{1}{4}$ (fat blue line), and various intermediate times $t = \frac{1}{16}, \frac{2}{16}, \frac{3}{16}$ (thin magenta lines).

The ODE system (8.15) obtained from PDE (8.14) is often called a *semi-discrete system* because space (x) has been discretized but time (t) is still continuous. We can write the system in vector form as

$$u'(t) = Au(t) + g(u(t)),$$

with matrix $A \in \mathbb{R}^{m\times m}$ and nonlinear function g defined by

$$A = \frac{\gamma}{h^2}\begin{pmatrix} -2 & 1 & & \\ 1 & -2 & \ddots & \\ & \ddots & \ddots & 1 \\ & & 1 & -2 \end{pmatrix}, \qquad g(v) = \kappa\begin{pmatrix} v_1(1-v_1) \\ v_2(1-v_2) \\ \vdots \\ v_m(1-v_m) \end{pmatrix} \tag{8.16}$$

for $v = (v_j) \in \mathbb{R}^m$. The symmetric tridiagonal matrix A is negative definite since one can check that the matrix $-A$ is positive definite. Therefore, $A = V\Lambda V^{-1}$ where V is orthogonal and $\Lambda = \mathrm{diag}(\lambda_i)$ contains the eigenvalues $\lambda_i < 0$. The eigenvalues of A are known to be in the interval $[-4\gamma/h^2, 0]$, with the most negative ones close to $-4\gamma/h^2$.
For large m, the initial value problem for the ODE system can be efficiently solved with suitable implicit methods. Then the stepsize only needs to be adjusted to the smoothness of the exact solution to keep the local errors small. On the other hand, when using an explicit method like Euler's method (7.4) or the explicit trapezoidal rule (7.7), very small stepsizes are necessary if the mesh-width h becomes small. The requirement that $\tau\lambda \in \mathcal{S}$ for all eigenvalues λ of A leads to the condition that the whole segment $[-4\gamma\tau/h^2, 0]$ of the negative real axis must fit in the

stability region $\mathcal{S}$ (cf. the left panel of Figure 8.4). This leads for these explicit methods to the stepsize restriction

$$\frac{\gamma\,\tau}{h^2} \;\leq\; \tfrac{1}{2}\,. \tag{8.17}$$

The nonlinear term $g(u)$ only contains moderate contributions, and therefore the stability restriction (8.17) will give a fair estimate of the possible time stepsizes. To find accurate approximations to the PDE the mesh-width h must be small, and therefore the explicit methods are not suited for solving this problem.

8.5 Further reading*

In this chapter, we have seen absolute stability, which is the most common concept of stability used. However, in the same spirit of absolute stability there are other definitions of stability, for instance L-stability and B-stability, see Sections IV.3 and IV.12 in [8].

Apart from Runge–Kutta methods, another way of systematically build high-order integration schemes are *multistep methods*. While Runge–Kutta methods achieve high order by introducting intermediate stages within one time step, multistep methods use one function evaluation per time step, but use previous times steps to increase the accuracy. Namely, in multistep methods with q steps the value of u_n is determined using not u_{n-1} only but rather $u_{n-1}, \ldots, u_{n-q}$ and the corresponding evaluations of the ODE's right-hand side. Main classes of multistep methods are the Adams methods and the BDF methods, where BDF stands for backward differentiation formula. The interested reader can take a look at Chapter III in [7].

References

[1] Canuto, C., Hussaini, M. Y., Quarteroni, A., Zang, T. A. (2007): *Spectral methods: fundamentals in single domains*. Springer Science & Business Media.

[2] Garling, D.J.H. (2013): *A course in mathematical analysis*, volumes I & II, Cambridge University Press.

[3] Gastinel, N. (1983) *Linear Numerical Analysis* Kershaw Publishing, London.

[4] Gautschi, W. (2012): *Numerical Analysis,* 2nd edition, Birkhäuser.

[5] Golub, G.H., Van Loan, C.F. (2013): *Matrix Computations*, 4th edition, The Johns Hopkins University Press.

[6] Hairer, E., Wanner, G. (2005): *Introduction à l'Analyse Numérique*, University of Geneva, https://archive-ouverte.unige.ch/unige:12656

[7] Hairer, E., Wanner, G., Nørsett, S. P. (1993). *Solving Ordinary Differential Equations I: Nonstiff Problems*. New York: Springer Berlin Heidelberg.

[8] Wanner, G., Hairer, E. (1996): *Solving ordinary differential equations II*. New York: Springer Berlin Heidelberg.

[9] Kahan, W. (1966): *Numerical linear algebra*, Canadian Mathematical Bulletin, 9(5), 757-801.

[10] Rivlin, T. (1974): *The Chebyshev Polynomials*. John Wiley and Sons, New York.

[11] Quarteroni, A., Sacco, R., Saleri, F. (2010): *Numerical mathematics*, 2nd edition, Springer Science & Business Media.

[12] Spijker, M.N., van de Griend, J.A. (2008): *Inleiding tot de Numerieke Wiskunde*, Univ. of Leiden, https://pub.math.leidenuniv.nl/~spijkermn/numa.pdf

[13] Stoer, J., Bulirsch, R. (2002): *Introduction to Numerical Analysis*, 3rd edition, Springer.

The material for these notes has been primarily taken from the books [4], [13] and the lecture notes [6]. Also available on the internet, and in Dutch, are the lecture notes [12].

Appendix A

Concepts and Results from Linear Algebra

A.1 The Vector Space $\mathbb{R}^n$

The real vector space $\mathbb{R}^n$ consists of all column vectors

$$v = \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{pmatrix},$$

where $v_i \in \mathbb{R}, i = 1, 2, \ldots, n$. We will also write $v = (v_i)$ and $v = (v_1, v_2, \ldots, v_n)^T$, where the 'transpose' operator 'T' transforms the row vector $v = (v_1, v_2, \ldots, v_n)$ to the corresponding column vector in $\mathbb{R}^n$. Addition (of two vectors v and w) and scalar multiplication (of a real scalar λ and a vector v) are defined component-wise,

$$(v_i) + (w_i) = (v_i + w_i), \quad \lambda(v_i) = (\lambda v_i).$$

A.2 Norms on the Vector Space $\mathbb{R}^n$

The vector space $\mathbb{R}^n$ becomes a *normed* vector space $(\mathbb{R}^n, \|\cdot\|)$ if we equip it with a norm $\|\cdot\|$, which is a real-valued function on $\mathbb{R}^n$ with the following three properties:

(i) $\|v\| > 0$ for all $v \in \mathbb{R}^n$ with $v \neq 0$,

(ii) $\|\lambda v\| = |\lambda| \|v\|$ for all $\lambda \in \mathbb{R}$ and $v \in \mathbb{R}^n$,

(iii) $\|v + w\| \leq \|v\| + \|w\|$ for all $v, w \in \mathbb{R}^n$.

Some common norms on $\mathbb{R}^n$ are

$$\begin{cases} \|v\|_1 = |v_1| + |v_2| + \ldots + |v_n| & \text{(the } \textit{sum norm} \text{ or } l_1\text{-norm)}, \\ \|v\|_2 = \sqrt{v_1^2 + v_2^2 + \ldots + v_n^2} & \text{(the } \textit{Euclidean norm} \text{ or } l_2\text{-norm)}, \\ \|v\|_\infty = \max\{|v_1|, |v_2|, \ldots, |v_n|\} & \text{(the } \textit{maximum norm} \text{ or } l_\infty\text{-norm)}, \end{cases} \tag{A.1}$$

for $v = (v_i) \in \mathbb{R}^n$.
For two column vectors $v, w \in \mathbb{R}^n$ we define their *(standard) inner product* by

$$v^T w = v_1 w_1 + v_2 w_2 + \ldots + v_n w_n.$$

The Euclidean norm of v equals $\|v\|_2 = \sqrt{v^T v}$.

A.3 The Vector Space $\mathbb{R}^{m\times n}$

The real vector space $\mathbb{R}^{m\times n}$ consists of all $m \times n$-matrices

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix},$$

where $a_{ij} \in \mathbb{R}, 1 \le i \le m, 1 \le j \le n$. We will also write $A = (a_{ij})$. Addition (of two matrices A and B) and scalar multiplication (of a real scalar λ and a matrix A) are defined component-wise,

$$(a_{ij}) + (b_{ij}) = (a_{ij} + b_{ij}), \quad \lambda(a_{ij}) = (\lambda a_{ij}).$$

We will identify a matrix $A = (a_{ij})$ with its associated linear mapping $A : \mathbb{R}^n \to \mathbb{R}^m$, which is defined via matrix-vector multiplication,

$$(Av)_i = \sum_{j=1}^{n} a_{ij} v_j, i = 1, 2, \ldots, m \quad \text{for all } v = (v_j) \in \mathbb{R}^n.$$

When dealing with two matrices $A = (a_{ij}) \in \mathbb{R}^{m\times p}$, $B = (b_{ij}) \in \mathbb{R}^{p\times n}$, the matrix product $AB \in \mathbb{R}^{m\times n}$ is defined as the matrix whose associated linear mapping $AB : \mathbb{R}^n \to \mathbb{R}^m$ is the composite mapping $v \to A(Bv)$, which leads to

$$(AB)_{ij} = \sum_{k=1}^{p} a_{ik} b_{kj} \quad \text{for all } i = 1, 2, \ldots, m \text{ and } j = 1, 2, \ldots, n.$$

For $A = (a_{ij}) \in \mathbb{R}^{m\times n}$, the *transpose* matrix A^T is obtained by interchanging rows and columns, which corresponds to $A^T = (a_{ji}) \in \mathbb{R}^{n\times m}$. This generalizes the definition of the transpose v^T of a (column) vector $v \in \mathbb{R}^m = \mathbb{R}^{m\times 1}$, which is the corresponding row vector in $\mathbb{R}^{1\times m}$. For a matrix product AB one has $(AB)^T = B^T A^T$, and for a nonsingular square matrix one has $\left(A^{-1}\right)^T = \left(A^T\right)^{-1}$.

A.4 Norms on the Vector Space $\mathbb{R}^{m\times n}$

The vector space $\mathbb{R}^{m\times n}$ becomes a *normed* vector space $(\mathbb{R}^{m\times n}, \|\cdot\|)$ if we equip it with a norm $\|\cdot\|$. A natural choice for this norm can be made if we have already defined a norm $\|\cdot\|_V$ on $V = \mathbb{R}^n$ and a norm $\|\cdot\|_W$ on $W = R^m$. This norm is called the *induced matrix norm* and is defined as

$$\|A\| = \max_{\substack{v\in V\\ v\neq 0}} \frac{\|Av\|_W}{\|v\|_V} = \max_{\substack{v\in V\\ \|v\|_V=1}} \|Av\|_W \quad \text{for all } A \in \mathbb{R}^{m\times n}. \tag{A.2}$$

For a given matrix A it is the smallest number α such that $\|Av\|_W \leq \alpha\|v\|_V$ for all $v \in V$. In particular we have

$$\|Av\|_W \leq \|A\|\|v\|_V \quad \text{for all } v \in V. \tag{A.3}$$

One easily verifies that the induced matrix norm $\|\cdot\|$ satisfies all three properties required for a norm.
If we choose $p \in \{1, 2, \infty\}$ and equip both $V = \mathbb{R}^n$ and $W = \mathbb{R}^m$ with the ℓ_p-norm, one can show that the corresponding induced matrix norm $\|\cdot\|_p$ is given by

$$\begin{cases} \|A\|_1 &= \max_{1\leq j\leq n} \sum_{i=1}^m |a_{ij}|, \\ \|A\|_2 &= \sqrt{\text{maximum eigenvalue of } A^TA}, \\ \|A\|_\infty &= \max_{1\leq i\leq m} \sum_{j=1}^n |a_{ij}|. \end{cases} \tag{A.4}$$

A.5 Orthogonal and Symmetric Matrices

A matrix $U \in \mathbb{R}^{m\times m}$ is called *orthogonal* if $U^TU = I$. This means that U is nonsingular with inverse $U^{-1} = U^T$, and also that the columns u_j of U are orthonormal, that is

$$u_i^T u_j = \begin{cases} 0 & \text{if } i \neq j, \\ 1 & \text{if } i = j. \end{cases}$$

If U is orthogonal, then U^T is also orthogonal, so the rows of U are also orthonormal.
A matrix $A \in \mathbb{R}^{m\times m}$ is called *symmetric* if $A^T = A$. If A is symmetric, then it is known that it has an orthonormal basis of eigenvectors $u_i \in \mathbb{R}^m$ corresponding to real eigenvalues λ_i, so it can be written as $A = U\Lambda U^T$ with real diagonal $\Lambda = \text{diag}(\lambda_i)$ and orthogonal $U = [u_1, u_2, \ldots, u_m]$.
A symmetric matrix A is called *positive definite* if $v^TAv > 0$ for all nonzero vectors $v \in \mathbb{R}^m$, and *negative definite* if $v^TAv < 0$ for all nonzero vectors $v \in \mathbb{R}^m$. Using the decomposition $A = U\Lambda U^T$ one easily verifies that positive definiteness is equivalent to the positivity of all eigenvalues λ_i, and negative definiteness to the negativity of all eigenvalues. This is a special case of the following more general result involving the so-called *Rayleigh quotients* $(v^TAv)/(v^Tv)$ of A.

Proposition A.1. *Consider for a symmetric matrix $A \in \mathbb{R}^{m\times m}$ with eigenvalues $\lambda_1 \leq \lambda_2 \leq \ldots \leq \lambda_m$ the set R_A of all its Rayleigh quotients,*

$$R_A = \left\{ \frac{v^TAv}{v^Tv} : v \in \mathbb{R}^m, v \neq 0 \right\}.$$

Then we have $R_A = [\lambda_1, \lambda_m]$.

Proof. Using the decomposition $A = U\Lambda U^T$ we find for nonzero v that

$$\frac{v^T A v}{v^T v} = \frac{(U^T v)^T \Lambda (U^T v)}{(U^T v)^T (U^T v)} = \frac{w^T \Lambda w}{w^T w} = \frac{\lambda_1 w_1^2 + \lambda_2 w_2^2 + \ldots + \lambda_m w_m^2}{w_1^2 + w_2^2 + \ldots + w_m^2},$$

where $w = U^T v$. We see that the set R_A consists of all possible convex combinations of the eigenvalues λ_i, $i = 1, 2, \ldots, m$, which implies $R_A = [\lambda_1, \lambda_m]$. □

A.6 Exercises

Exercise A.1.. Let $\|\cdot\|$ denote the induced matrix norm on $\mathbb{R}^{m\times m}$ corresponding to some vector norm on $\mathbb{R}^m$.

(a) Show that we have $\|I\| = 1$ for the identity matrix $I \in \mathbb{R}^{m\times m}$.

(b) Show that the matrix norm is *submultiplicative*, that is,

$$\|AB\| \leq \|A\|\|B\| \quad (\text{for all } A, B \in \mathbb{R}^{m\times m}).$$

Exercise A.2.. Let $\|\cdot\|$ denote a vector norm on $\mathbb{R}^m$. The corresponding induced matrix norm on $\mathbb{R}^{m\times m}$ is also denoted by $\|\cdot\|$. Show that for any nonsingular matrix $A \in \mathbb{R}^{m\times m}$ we have

$$\|A^{-1}\| = \left(\min_{v\neq 0} \frac{\|Av\|}{\|v\|}\right)^{-1} = \left(\min_{\|v\|=1} \|Av\|\right)^{-1}.$$

Exercise A.3.. Prove for any matrix $A \in \mathbb{R}^{m\times n}$ the expressions for the induced matrix norms $\|A\|_1$ and $\|A\|_\infty$ in (A.4).

Exercise A.4.. (a) Prove for any $A \in \mathbb{R}^{m\times n}$ the expression for $\|A\|_2$ in (A.4).
(Hint: apply Proposition A.1 to the symmetric matrix $A^T A$.)

(b) Prove for nonsingular $A \in \mathbb{R}^{m\times m}$ that

$$\|A^{-1}\|_2 = 1/\sqrt{\text{minimum eigenvalue of } A^T A}.$$

(Hint: make use of Exercise A.2.)

Exercise A.5.. Let $U \in \mathbb{R}^{m\times m}$ be orthogonal.

(a) First show that $\|Uv\|_2^2 = \|v\|_2^2$ for all $v \in \mathbb{R}^m$, and then that $\|U\|_2 = 1$.

(b) Show that $\|UA\|_2 = \|AU\|_2 = \|A\|_2$ for any $A \in \mathbb{R}^{m\times m}$.

Exercise A.6.. Suppose $A \in \mathbb{R}^{m\times m}$ is symmetric with eigenvalues λ_i, $i = 1, 2, \ldots, m$.

(a) Show that $\|A\|_2 = \max_i |\lambda_i|$.
(Hint: use the previous exercise.)

(b) If A is also nonsingular, show that $\|A^{-1}\|_2 = (\min_i |\lambda_i|)^{-1}$.

Appendix B

Concepts and Results from Analysis

B.1 Some Notation: big $\mathcal{O}$, little o and $\sim$

For two real functions φ and ψ we will write

$$\varphi(t) \;=\; \mathcal{O}(\psi(t)) \quad (t \to 0) \tag{B.1}$$

if there are $C, \delta > 0$ such that

$$|\varphi(t)| \leq C|\psi(t)| \quad \text{for all } t \text{ with } 0 < |t| < \delta.$$

In other words, the order term $\mathcal{O}(\psi(t))$ may be interpreted as an unspecified function whose absolute value is bounded by a constant times $|\psi(t)|$ in a 'punctured' neighbourhood of $t = 0$. Here the word 'punctured' means that the point $t = 0$ has been removed from its neighbourhood $(-\delta, \delta)$.
Apart from the 'big $\mathcal{O}$ notation' above, we mention two other important notations. The first one is the 'little o notation', where we write

$$\varphi(t) \;=\; o(\psi(t)) \quad (t \to 0) \tag{B.2}$$

if $\varphi(t)/\psi(t) \to 0$ as $t \to 0$. The second one is the 'asymptotic equivalence notation', where we write

$$\varphi(t) \sim \psi(t) \quad (t \to 0) \tag{B.3}$$

if $\varphi(t)/\psi(t) \to 1$ as $t \to 0$.

These notations are widely used in mathematics, and are slightly more general than described above. For example, we could replace the specification $t \to 0$ by $t \to \infty$ or $t \downarrow 0$. This would correspond to taking different limits in (B.2) and (B.3). In the 'big $\mathcal{O}$ notation' (B.1) this would correspond to taking a neighbourhood (t_0, ∞) of $t = \infty$ or a punctured right neighbourhood $(0, \delta)$ of $t = 0$.

Exercise B.1.. Show that assertions (a)-(h) below are correct. Give in cases (a)–(d) a suitable neighbourhood ($(-\delta, \delta)$, $(0, \delta)$ or (t_0, ∞)) and a corresponding value for the constant C.

(a) $6t^2 + 7t^3 = \mathcal{O}(t^2) \quad (t \to 0)$

(b) $6t^2 + 7t^3 = \mathcal{O}(t^3) \quad (t \to \infty)$

(c) $10\sin(3\sqrt{t}) = \mathcal{O}(\sqrt{t}) \quad (t \downarrow 0)$

(d) $10\sin(3\sqrt{t}) = \mathcal{O}(1) \quad (t \to \infty)$

(e) $6t^2 + 7t^3 = o(t) \quad (t \to 0)$

(f) $6t^2 + 7t^3 = o(t^4) \quad (t \to \infty)$

(g) $6t^2 + 7t^3 \sim 6t^2 \quad (t \to 0)$

(h) $6t^2 + 7t^3 \sim 7t^3 \quad (t \to \infty)$

B.2 Intermediate Value, Mean Value and Rolle's Theorem

Theorem B.1 (Intermediate Value Theorem)**.** *Let $f : [a, b] \to \mathbb{R}$ be continuous. Then for any value y_0 between $f(a)$ and $f(b)$ there exists $x_0 \in [a, b]$ with $f(x_0) = y_0$.*

Theorem B.2 (Mean Value Theorem)**.** *Let $f : [a, b] \to \mathbb{R}$ be continuous on $[a, b]$ and differentiable on (a, b), where $a < b$. Then there exists $\xi \in (a, b)$ with*

$$f'(\xi) = \frac{f(b) - f(a)}{b - a}.$$

A special case of the mean value theorem is obtained if $f(a) = f(b)$. The resulting theorem is called Rolle's theorem.

Theorem B.3 (Rolle's Theorem)**.** *Let $f : [a, b] \to \mathbb{R}$ be continuous on $[a, b]$ and differentiable on (a, b), where $a < b$. If $f(a) = f(b)$ then there exists $\xi \in (a, b)$ with*

$$f'(\xi) = 0.$$

B.3 Taylor Approximations

We will often make use of Taylor approximations. We first discuss the case of real functions $f : [a, b] \to \mathbb{R}$ and then the case of vector-valued functions $f : [a, b] \to \mathbb{R}^n$.

Real functions. Let a function $f : [a, b] \to \mathbb{R}$ be given, which is $k + 1$ times continuously differentiable on $[a, b]$. We can approximate f on that interval by choosing $c \in [a, b]$ and define the Taylor polynomial of f about $x = c$ as the polynomial P_k of degree at most k that has the same derivatives of order $j = 0, 1, \ldots, k$ in $x = c$ as the function f. This polynomial is given by

$$\begin{aligned} P_k(x) &= f(c) + (x - c)f'(c) + \frac{(x - c)^2}{2} f''(c) + \ldots + \frac{(x - c)^k}{k!} f^{(k)}(c) \\ &= \sum_{j=0}^{k} \frac{(x - c)^j}{j!} f^{(j)}(c). \end{aligned} \tag{B.4}$$

According to Taylor's theorem (see for example [2, Theorem 7.6.2]) we can write

$$f(x) = P_k(x) + R_{k+1}(x), \tag{B.5}$$

where the remainder term $R_{k+1}(x)$ can be expressed as

$$R_{k+1}(x) = \frac{(x-c)^{k+1}}{(k+1)!} f^{(k+1)}(\xi), \tag{B.6}$$

where ξ lies between c and x. Note that by taking $k = 0$, $c = a$, $x = b$ we recover the mean value theorem (Theorem B.2).

Vector-valued functions. It is possible (see [2]) to extend Taylor's theorem to functions $f : \mathbb{R}^n \to \mathbb{R}^m$, but that is beyond the scope of this course. For this general class of functions we will only discuss the case $k = 1$ (linearization) in section B.4.

We do need, however, an extension of the above results to the case $n = 1, m \geq 1$, that is, to vector-valued functions $f : [a, b] \to \mathbb{R}^m$. In that case, the Taylor approximation P_k and remainder term R_{k+1} defined in (B.4), (B.5) are vector-valued functions too, and the derivatives $f^{(j)}(c)$ must be interpreted component-wise. Unfortunately, expression (B.6) is generally *not* valid for vector-valued functions. Although it is valid for the individual components of f, the intermediate value ξ will generally be different for each component of f. Fortunately, there exists an alternative expression for the remainder term, which is also valid for the vector-valued case. This is the so-called integral form (see for example [2, Theorem 8.7.3]),

$$R_{k+1}(x) = \frac{1}{k!} \int_c^x (x-t)^k f^{(k+1)}(t)\, dt. \tag{B.7}$$

From the integral form we easily find the following upper bound for the norm of the remainder term,

$$\|R_{k+1}(x)\| \leq \frac{|x-c|^{k+1}}{(k+1)!} \max_{t\in[a,b]} \|f^{(k+1)}(t)\| \quad (\text{for all } x \in [a, b]). \tag{B.8}$$

For the scalar case ($m = 1$) this upper bound (with $\|\cdot\| = |\cdot|$) can also be obtained from (B.6).

Remark B.4. In many applications it is convenient to write $x = c + h$, where $h \to 0$ or $h \downarrow 0$. In that case it follows from (B.4), (B.5), (B.8) that

$$f(c+h) = \sum_{j=0}^{k} \frac{h^j}{j!} f^{(j)}(c) + \mathcal{O}(h^{k+1}), \tag{B.9}$$

where the order term $\mathcal{O}(h^{k+1})$ should be interpreted as an unspecified vector-valued function whose norm is $\mathcal{O}(h^{k+1})$. ◊

B.4 Linearization

In this section we consider functions $g : \Omega \to \mathbb{R}^m$ where the domain $\Omega \subset \mathbb{R}^n$ is a non-empty open set. We use the following notation,

$$v = \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{pmatrix}, \quad g(v) = \begin{pmatrix} g_1(v_1, v_2, \ldots, v_n) \\ g_2(v_1, v_2, \ldots, v_n) \\ \vdots \\ g_m(v_1, v_2, \ldots, v_n) \end{pmatrix}.$$

We call the function g *continuously differentiable* on Ω if all partial derivatives $\frac{\partial}{\partial v_j} g_i(v)$ exist and are continuous on Ω. (Note that this implies that g is continuous on Ω as well.) In that case it is known (see [2]) that we can approximate g in the neighbourhood of any given $v \in \Omega$ by its *linearization* around v,

$$g(v + h) \approx g(v) + g'(v)h, \tag{B.10}$$

where $h \in \mathbb{R}^n$ has a small norm and $g'(v)$ is the *Jacobian matrix* of the function g at v,

$$g'(v) = (\partial g_i / \partial v_j) = \begin{pmatrix} \partial g_1/\partial v_1 & \partial g_1/\partial v_2 & \cdots & \partial g_1/\partial v_n \\ \partial g_2/\partial v_1 & \partial g_2/\partial v_2 & \cdots & \partial g_2/\partial v_n \\ \vdots & \vdots & & \vdots \\ \partial g_m/\partial v_1 & \partial g_m/\partial v_2 & \cdots & \partial g_m/\partial v_n \end{pmatrix}.$$

If $m = 1$ then $g'(v)$ is a row vector, for $n = 1$ it is a column vector, and for $m = n$ it is a square matrix.
The quality of linearization (B.10) depends on the size of the remainder term $r(h)$ defined by

$$g(v + h) = g(v) + g'(v)h + r(h).$$

Under the above assumptions (g is continuously differentiable on Ω) it can be shown (see [2]) that

$$\lim_{h \to 0} \frac{\|r(h)\|}{\|h\|} = 0,$$

which is usually expressed by the following equivalent, but more compact, notation

$$g(v + h) = g(v) + g'(v)h + o(\|h\|) \quad (h \to 0).$$

Under the stronger assumption that g is twice continuously differentiable on Ω (that is, all second order partial derivatives $\frac{\partial^2}{\partial v_j \partial v_k} g_i(v)$ exist and are continuous on Ω), we even have

$$g(v + h) = g(v) + g'(v)h + \mathcal{O}(\|h\|^2) \quad (h \to 0), \tag{B.11}$$

which is a compact notation for the existence of a constant $C \geq 0$ such that

$$\|r(h)\| \leq C\|h\|^2 \quad \text{for all } h \in \mathbb{R}^n \text{ sufficiently close to zero.}$$

B.5 The Mean Value Inequality

Assume that a norm $\|\cdot\|_V$ on $V = \mathbb{R}^n$ is given and a norm $\|\cdot\|_W$ on $W = R^m$. Let $g : \Omega \to \mathbb{R}^m$ be continuously differentiable on a non-empty open subset $\Omega \subset \mathbb{R}^n$. Let $v, \widetilde{v} \in \Omega$ be given such that the line segment connecting v and $\widetilde{v}$ is contained in Ω. We define the mappings $w : [0,1] \to \Omega$ and $\varphi : [0,1] \to \mathbb{R}^m$ by

$$w(t) = (1-t)v + t\widetilde{v}, \quad \varphi(t) = g(w(t)).$$

We have

$$g(\widetilde{v}) - g(v) = \varphi(1) - \phi(0) = \int_0^1 \varphi'(t)\,\mathrm{d}t,$$

where the integral of the vector-valued function $\varphi'(t)$ is defined component-wise. Hence

$$\|g(\widetilde{v}) - g(v)\|_W = \left\| \int_0^1 \varphi'(t)\,\mathrm{d}t \right\|_W \leq \int_0^1 \|\varphi'(t)\|_W\,\mathrm{d}t, \tag{B.12}$$

where the inequality follows by writing the integrals as a limit of Riemann sums, together with application of the triangle inequality for the norm $\|\cdot\|_W$.
An application of the chain rule yields

$$\varphi'(t) = g'(w(t))w'(t) = g'(w(t))(\widetilde{v} - v).$$

Using (A.3) it follows that

$$\|\varphi'(t)\|_W \leq \|g'(w(t))\|\,\|\widetilde{v} - v\|_V,$$

where $\|\cdot\|$ denotes the induced matrix norm defined in (A.2). In combination with (B.12) we find

$$\|g(\widetilde{v}) - g(v)\|_W \leq \|\widetilde{v} - v\|_V \max_{0\leq t\leq 1} \|g'(w(t))\|. \tag{B.13}$$

This is known as the mean value inequality (see [2, Theorem 17.2.2]).

B.6 Completeness of Normed Vector Spaces

In this section we list a number of definitions and results involving real normed vector spaces $(V, \|\cdot\|)$.

- A sequence $\{v_k\}$ in $(V, \|\cdot\|)$ is called *convergent* if there exists an element $v_* \in V$ such that $\lim_{k\to\infty} \|v_k - v_*\| = 0$.
- A sequence $\{v_k\}$ in $(V, \|\cdot\|)$ is called a *Cauchy sequence* if $\lim_{k,\ell\to\infty} \|v_k - v_\ell\| = 0$.
- The real normed vector space $(V, \|\cdot\|)$ is called *complete* (or a *real Banach space*) if every Cauchy sequence is convergent.
- Any finite-dimensional real normed vector space $(V, \|\cdot\|)$ is complete.

- Any two norms $\|\cdot\|$, $\|\cdot\|'$ on a real finite-dimensional vector space V are *equivalent*, which means that there exist two constants $c_2 \geq c_1 > 0$ such that

$$c_1\|v\| \leq \|v\|' \leq c_2\|v\| \quad \text{for all } v \in V.$$

  Note that this implies that concepts like convergence of sequences, openness of sets and continuity of functions are independent of the chosen norm on V.