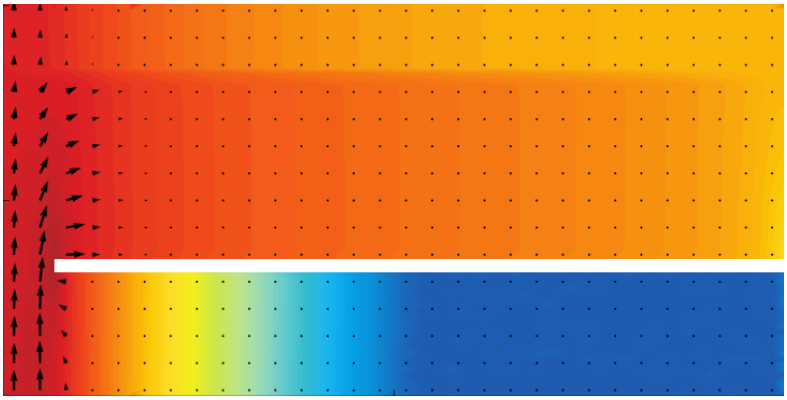
Numerical Mathematics

Lecture Notes

Patricio Farrell



electron current density in cross section of single photon system

Version: 16. April 2023
Please report typos to patricio.farrell@wias-berlin.de

Contents

1 Introduction

It is nearly impossible to argue why numerical mathematics is *not* important. It is such a vital part of our everyday life. Every technological device approximates at some point complex calculations with simpler ones or was even built on the basis of previous simulation results (like for example the single photon system on the title page). What *is* true though is the fact that despite its importance numerical or applied mathematics in general is surprisingly invisible! This is the unfortunate theorem of applied mathematics:

Theorem 1 (Unfortunate theorem of applied mathematics)**.** *Despite its importance and ubiquity, applied mathematics is surprisingly invisible in our everyday life.*

For holidays many pick up a book or two (or download them on their high-tech smartphones or tablets). Yet, it is very unlikely that these books contain popular mathematics. Numerical/computational mathematics in particular, however, always becomes palpable when something is *not* working. Take for example, the crash of the Ariane 5 rocket in 1996. You can watch a video of it here which unfortunately confirms several stereotypes (it seems at ESA work only gray-haired white men with white coats and horn-rimmed glasses):

`https://www.youtube.com/watch?v=gp_D8r-2hwk`.

Here something major went wrong. Several billion euros were burned (quite literally) in just under 40 seconds after lift-off. It turns out that (computational) mathematics was to blame: The rocket crashed due to a faulty data conversion from 64-bit floating point to 16-bit signed integer format. Unfortunately, the 64-bit floating point number was larger than what could be stored in 16-bit signed integer format.

This perception in terms of "negatives" certainly does not help to improve the image or may even be partially the cause of it. The other times when technological devices run smoothly no one thinks of mathematics as the very reason for it. The present lecture notes try to explain the beauty behind numerical mathematics while not shying away from presenting some more theoretical ideas behind it as well.

Every chapter is accompanied by MATLAB files to illustrate the material. If these are put into a folder called `Code` on the same level as these lecture notes, then one can open them by simply clicking on them.

2 Well-posedness, conditioning and stability

We discuss three important concepts in numerical analysis which are related but important to keep apart. While *well-posedness* and *conditioning* refer to the mathematical problem one wants to tackle, *stability* is a property of the algorithm one uses to solve the mathematical problem.

For example, let us formalize polynomial root-finding. So the problem is: Given a polynomial $p(x) = \sum_{i=0}^{n} a_i x^i$ (which can be entirely described by its coefficients), find its roots. The map F given by

$$F\colon \text{coefficient vector } (a_n, \dots, a_0)^T \mapsto \text{vector of all real roots}$$

achieves this. For example,

$$F\colon (1, 2)^T \mapsto -2.$$

We can also see that the map does not always yield a solution, e.g.

$$F\colon (1, 0, 1)^T \mapsto \{\},$$

since we only look for real roots. In the above example, the function F maps from $\mathbb{R}^{n+1}$ to $\mathbb{R}^k$ with $k \leq n$, depending on the number of (real) roots.

In general, a mathematical problem is not restricted to either Euclidean spaces or even finite-dimensional ones. For the input and output spaces we are free to choose any arbitrary normed (function) space such as the space of polynomials of degree not bigger than n as well as the space of continuous or integrable functions. So in general, we study a mathematical problem $F : V \to W$ for some normed real vector spaces V and W. We call V the *input data space* and W the *output data space*. The corresponding norms are denotes with $\|\cdot\|_V$ and $\|\cdot\|_W$. Thus, $F(x)$ denotes the output from the mathematical problem F with input x.

2.1 Well-posedness

Hadamard [16] postulated three properties which a mathematical problem should satisfy in order to accurately describe our physical reality. He called a problem *well-posed* if

- there exists a solution to it,
- the solution is unique
- and depends continuously on the input data.

In this sense, looking for the roots of $x + 2$ is a well-posed problem whereas looking for the (real) roots of $x^2 + 1$ is not. If the mathematical problem cannot be solved, it makes no sense to devise a root-finding algorithm. A priori, the best algorithm in the world is doomed to fail!

The final property means that if we have two input data sets which are *close* then also the mathematical problems should produce *similar* solutions. We will give an example of this in Chapter 4 in the context of polynomial interpolation. Notice, however, that the word *close* in this context is a bit vague. Despite its continuous dependence, it can still be true that varying the input a little bit, results in a relative large change in the output. The *condition number* helps to quantify exactly how large that change is. There are two common types of condition numbers: The normwise and componentwise condition numbers.

2.2 Condition number

2.2.1 Normwise condition number

For some norm given norms $\|\cdot\|_V$ and $\|\cdot\|_W$ on the linear spaces V and W which have to be specified for each application the *absolute normwise condition number* $\kappa_a \geq 0$ is defined as the smallest number such that

$$\|F(\tilde{x}) - F(x)\|_W \leq \kappa_a \|\tilde{x} - x\|_V \tag{2.1}$$

in the limit of $\tilde{x}$ approaching x. Note, that if the mathematical problem does not depend continuously on the data (i.e. it is *ill-posed*) formally the condition number becomes infinite, $\kappa_a = \infty$. Similarly, one can define the *relative normwise condition number* $\kappa_r \geq 0$ to be the smallest number such that

$$\frac{\|F(\tilde{x}) - F(x)\|_W}{\|F(x)\|_W} \leq \kappa_r \frac{\|\tilde{x} - x\|_V}{\|x\|_V} \tag{2.2}$$

in the limit of $\tilde{x}$ approaching x. Both κ_a and κ_r measure how small perturbations to the input data (e.g. the polynomial coefficients) impact the solution to the mathematical problem (the roots). That is,

$$\frac{\|\text{change in output}\|_W}{\|\text{output}\|_W} \;\leq\; \kappa_r \, \frac{\|\text{change in input}\|_V}{\|\text{input}\|_V}.$$

In other words the condition numbers describe how sensitive the output is with respect to the input. If the function F is differentiable, its derivative help to measure this sensitivity. In fact, we will see below that we can think of the condition numbers as some generalized derivative. For "large" condition numbers the mathematical problem is called *ill-conditioned*. On the other hand, for condition numbers close to zero, the problem is called *well-conditioned*.

We can obtain equivalent definitions for (2.1) and (2.2). By setting $\Delta x = \tilde{x} - x$ we have

$$\kappa_a = \lim_{\varepsilon \to 0^+} \sup_{\|\Delta x\|_V \leq \varepsilon} \left[\frac{\|F(x + \Delta x) - F(x)\|_W}{\|\Delta x\|_V} \right] \tag{2.3}$$

and

$$\kappa_r = \lim_{\varepsilon \to 0^+} \sup_{\|\Delta x\|_V \leq \varepsilon} \left[\frac{\|F(x + \Delta x) - F(x)\|_W}{\|F(x)\|_W} \Big/ \frac{\|\Delta x\|_V}{\|x\|_V} \right]. \tag{2.4}$$

From these definitions it become clear that for differentiable F, we obtain

$$\kappa_a = \|DF(x)\|_{V,W} \qquad \text{and} \qquad \kappa_r = \frac{\|DF(x)\|_{V,W}}{\|F(x)\|_W / \|x\|_V},$$

where $DF(x) = (\partial F_i(x)/\partial x_j)$ denotes the Jacobian of F at x and

$$\|DF(x)\|_{V,W} = \sup_{\Delta x \neq 0} \left\{ \frac{\|DF(x)\Delta x\|_W}{\|\Delta x\|_V} \right\}$$

the induced operator norm.

This looks rather cumbersome. What happens if we simplify the problem and study some (differentiable) function $f\colon \mathbb{R} \to \mathbb{R}$, that is $V = W = \mathbb{R}$? This case corresponds to studying the conditioning of evaluating the function f at x. For real numbers the standard norm is the absolute value $|\cdot|$. In this case, the Jacobian becomes simply a derivative and the operator norm is its absolute value,

$$|f'(x)|_{\mathbb{R},\mathbb{R}} = \sup_{\Delta x \neq 0} \left\{ \frac{|f'(x)\Delta x|}{|\Delta x|} \right\} = |f'(x)|.$$

Then the absolute and relative condition numbers simplify to

$$\kappa_a = |f'(x)| \qquad \text{and} \qquad \kappa_r = \left| \frac{f'(x)x}{f(x)} \right|.$$

The above expressions one can also extend to functions defined on finite domains.

We finish with another example. Polynomial root finding can become highly ill-conditioned – even ill-posed. Consider the polynomial $p_\varepsilon(x) = x^2 - \varepsilon$ and its perturbation $p(x) = x^2$ with roots $\pm\sqrt{\varepsilon}$ and double root at $x = 0$, respectively. Then the absolute condition number κ_a according to (2.1) would need to satisfy in the discrete maximum norm ($V = \mathbb{R}^3$ and $W = \mathbb{R}^2$), for example,

$$\sqrt{\varepsilon} = \left\| \begin{pmatrix} \sqrt{\varepsilon} \\ -\sqrt{\varepsilon} \end{pmatrix} \right\|_\infty = \left\| F\begin{pmatrix} 1 \\ 0 \\ -\varepsilon \end{pmatrix} - F\begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} \right\|_\infty \leq \kappa_a \left\| \begin{pmatrix} 1 \\ 0 \\ -\varepsilon \end{pmatrix} - \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} \right\|_\infty = \kappa_a \varepsilon.$$

For small ε this is impossible since $\sqrt{\varepsilon}$ dominates ε. Hence, $\kappa_a = \infty$.

Demos

- FunctionEvaluationConditonNumber.m

2.2.2 The condition number of a linear operator

The previously discussed condition numbers have a special form if we make the additional assumption that the mapping (the operator) $F\colon V \to W$ is *linear*, that is for all $x, y \in V$ and $\alpha, \beta \in \mathbb{R}$ we have

$$F(\alpha x + \beta y) = \alpha F(x) + \beta F(y).$$

If we consider the vector space of all linear operators, we can supply it with the induced *operator norm*

$$\|F\|_{V,W} := \sup_{x\neq 0} \frac{\|F(x)\|_W}{\|x\|_V} = \sup_{\|x\|_V=1} \|F(x)\|_W.$$

We say the linear operator is *bounded* if $\|F\|_{V,W} < \infty$. For bounded linear operators the operator norm becomes actually a norm as one can easily verify. From the definition we deduce the important property

$$\|F(x)\|_W \leq \|F\|_{V,W}\|x\|_V$$

for all $x \in V$. From the definition it is clear that $\|F\|_{V,W}$ is the smallest constant for which the above inequality holds. Due to linearity we deduce

$$\|F(\tilde{x}) - F(x)\|_W = \|F(\tilde{x} - x)\|_W \leq \|F\|_{V,W}\|\tilde{x} - x\|_V \tag{2.5}$$

for all $x, \tilde{x} \in V$. In other words, for linear operators the operator norm is equivalent to the absolute condition number. In fact, the following theorem holds, where we allow the condition numbers to become infinite (indicating an ill-posed problem).

Theorem 2. *Let $F\colon V \to W$ be a linear operator. Then*

$$\kappa_a = \|F\|_{V,W} \in [0, \infty].$$

and

$$\kappa_r \leq \frac{\|F\|_{V,W}}{\inf_{\|x\|_V=1} \|F(x)\|_W} \in [0, \infty].$$

If F is bijective, we find

$$\kappa_r \leq \|F\|_{V,W}\|F^{-1}\|_{W,V}.$$

Proof. The first statement follows from (2.5) and the fact that $\|F\|_{V,W}$ is the smallest constant for which it holds. For the second statement

$$\frac{\|x\|_V}{\|F(x)\|_W} = \frac{1}{\|F(x)\|_W/\|x\|_V} = \frac{1}{\|F(x/\|x\|_V)\|_W} \leq \frac{1}{\inf_{\|x\|_V=1} \|F(x)\|_W}.$$

Hence,

$$\frac{\|F(\tilde{x}) - F(x)\|_W}{\|F(x)\|_W} \leq \frac{\|F\|_{V,W}}{\inf_{\|x\|_V=1} \|F(x)\|_W} \frac{\|\tilde{x} - x\|_V}{\|x\|_V}.$$

Comparing this inequality with the definition of the relative condition number (2.4) yields the second claim. Finally, if F is bijective, its inverse F^{-1} exists. Then for $y = F(x)$, we find

$$\begin{aligned}\|F^{-1}\|_{W,V} &= \sup_{y\neq 0} \frac{\|F^{-1}(y)\|_V}{\|y\|_W} = \sup_{x\neq 0} \frac{\|x\|_V}{\|F(x)\|_W} \\ &= \sup_{\|x\|_V=1} \frac{1}{\|F(x)\|_W} = \frac{1}{\inf_{\|x\|_V=1} \|F(x)\|_W}.\end{aligned}$$

□

We point out that if the linear operator F is bounded, its absolute condition number is finite. If it is additionally injective, also its relative condition number is finite.

A word on notation: Matrices are a class of very well known linear operators. Since it is somewhat unusual to write $A(x)$ for Ax, for linear operators F we will often drop the parentheses and write Fx instead of $F(x)$.

2.2.3 Componentwise condition number

The previously introduced of normwise condition numbers is sometimes too restrictive. Consider for example the linear system $Ax = b$ with the diagonal matrix

$$A = \begin{pmatrix} 1 & 0 \\ 0 & \varepsilon \end{pmatrix} \qquad \text{which implies} \qquad A^{-1} = \begin{pmatrix} 1 & 0 \\ 0 & \frac{1}{\varepsilon} \end{pmatrix}$$

for $\varepsilon > 0$. Intuitively, solving this linear system with a diagonal matrix should be well-conditioned since the linear system decouples into two equations which can be solved separately. However, using the normwise condition number we have in the discrete maximum norm

$$\kappa_r(A) = \|A\|_\infty \|A^{-1}\|_\infty = \frac{1}{\varepsilon}.$$

So for small ε this type of condition number grows!

If we assume for simplicity that $V = \mathbb{R}^n$ and $W = \mathbb{R}^m$, it is instructive to study the more sensitive *componentwise error*

$$F_i(\tilde{x}) - F_i(x) = F_i(x + \Delta x) - F_i(x)$$

for $1 \le i \le m$. If we assume that F is differentiable and apply the mean-value theorem to the scalar function $g(t) = F(x + t\Delta x)$, then we may estimate for some $\tau \in [0, 1]$

$$\begin{aligned}
\left| \frac{F_i(x + \Delta x) - F_i(x)}{F_i(x)} \right| &= \frac{1}{|F_i(x)|} \left| \sum_{j=1}^{n} \frac{\partial F_i(x + \tau \Delta x)}{\partial x_j} \Delta x_j \right| \\
&= \frac{1}{|F_i(x)|} \left| \sum_{j=1}^{n} \frac{\partial F_i(x + \tau \Delta x)}{\partial x_j} x_j \frac{\Delta x_j}{x_j} \right| \\
&\le \frac{1}{|F_i(x)|} \left(\sum_{j=1}^{n} \left| \frac{\partial F_i(x + \tau \Delta x)}{\partial x_j} \right| |x_j| \right) \max_{1 \le j \le n} \left| \frac{\Delta x_j}{x_j} \right| \\
&= \frac{|\nabla F_i(x + \tau \Delta x)|^T \, |x|}{|F_i(x)|} \max_{1 \le j \le n} \left| \frac{\Delta x_j}{x_j} \right|.
\end{aligned}$$

In the last expression the absolute value sign is applied componentwise to the gradient and the vector x. Similar to before, we rearrange this to

$$\max_{1 \le i \le m} \left| \frac{F_i(x + \Delta x) - F_i(x)}{F_i(x)} \right| \Bigg/ \max_{1 \le j \le n} \left| \frac{\Delta x_j}{x_j} \right| \le \max_{1 \le i \le m} \frac{|\nabla F_i(x + \tau \Delta x)|^T \, |x|}{|F_i(x)|}$$

where again the absolute value sign is applied componentwise to the Jacobian and the vector x. If we understand the division componentwise, the right-hand side becomes

$$\left\| \frac{|DF(x+\tau\Delta x)|\,|x|}{|F(x)|} \right\|_\infty .$$

For vanishing Δx this motivates our definition of the *componentwise condition number*

$$\kappa_r^c = \left\| \frac{|DF(x)|\,|x|}{|F(x)|} \right\|_\infty .$$

Let us compute the componentwise condition number for the linear system in the introductory example. In this case

$$\kappa_r^c = \left\| \frac{\left|\begin{pmatrix} 1 & 0 \\ 0 & \varepsilon \end{pmatrix}\right| \left|\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}\right|}{\left|\begin{pmatrix} 1 & 0 \\ 0 & \varepsilon \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}\right|} \right\|_\infty = \left\| \frac{\begin{pmatrix} |x_1| \\ \varepsilon|x_2| \end{pmatrix}}{\left|\begin{pmatrix} x_1 \\ \varepsilon x_2 \end{pmatrix}\right|} \right\|_\infty = \max\left\{ \frac{|x_1|}{|x_1|}, \frac{|x_2|}{|x_2|} \right\} = 1.$$

So the componentwise condition number is actually ideal, reflecting our initial intuition.

Let us look at another example. Consider the multiplication of two real numbers, defined by

$$f\colon \mathbb{R}^2 \to \mathbb{R}, \qquad f(x,y) = xy.$$

The Jacobian is given by

$$Df(x,y) = \begin{pmatrix} y & x \end{pmatrix}.$$

Hence, the normwise condition number in the discrete maximum norm yields

$$\kappa_r = \frac{\|Df(x,y)\|_\infty \left\| \begin{pmatrix} x \\ y \end{pmatrix} \right\|_\infty}{|f(x,y)|} = \frac{(|x|+|y|)\max\{|x|,|y|\}}{|xy|}$$

$$= \begin{cases} \dfrac{|x|^2+|x||y|}{|xy|} = \dfrac{|x|}{|y|} + 1 & \text{for} \quad |x| > |y|, \\ \dfrac{|y|^2+|x||y|}{|xy|} = \dfrac{|y|}{|x|} + 1 & \text{for} \quad |x| \le |y|. \end{cases}$$

This means that the normwise condition number implies that multiplication is only well conditiond if $|x| \approx |y|$. However, if the absolute values of both factors differ by a lot, then the normwise condition number becomes large. On the other hand, the componentwise condition number is given by

$$\kappa_r^c = \left\| |Df(x)| \left|\begin{pmatrix} x \\ y \end{pmatrix}\right| \Big/ |f(x,y)| \right\|_\infty = \frac{2|x||y|}{|xy|} = 2,$$

implying that in the componentwise sense multiplication is well-conditioned regardless of the magnitude of x and y. So which condition number mirrors the

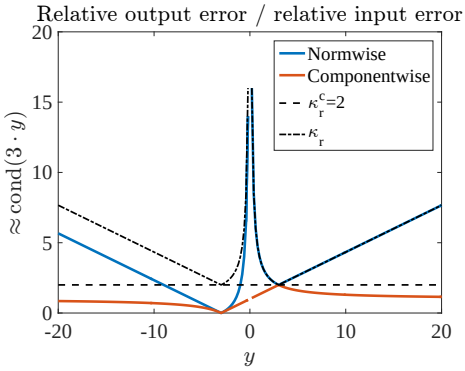


Figure 2.1: Comparison between relative output error over relative input error for multiplication $x \cdot y$ with $x = 3$ and $\varepsilon_1 = \varepsilon_2 = 10^{-8}$, using normwise (blue) and componentwise (red) errors.

"correct" behavior? Suppose we perturb some $x, y \in \mathbb{R}$ of possibly quite different magnitudes by $\tilde{x} = x(1 + \varepsilon_1)$ and $\tilde{y} = y(1 + \varepsilon_2)$ for some $\varepsilon_1, \varepsilon_2 \in \mathbb{R}$. Then the output error is given by

$$\frac{xy - \tilde{x}\tilde{y}}{xy} = \varepsilon_1 + \varepsilon_2 + \varepsilon_1\varepsilon_2 = \frac{\tilde{x} - x}{x} + \frac{\tilde{y} - y}{y} + \left(\frac{\tilde{x} - x}{x}\right)\left(\frac{\tilde{y} - y}{y}\right).$$

This means that the multiplication error is directly bounded by the errors in the input, independently from any size difference between x and y. So the componentwise condition number wins. Figure 2.1 demonstrates that even the componentwise condition number could be improved to accurately reflect the ratio of relative output to relative input error.

Even though the componentwise condition number is a very precise tool, in practice it might be difficult more difficult to compute than the normwise condition number.

Demos

- `MultiplicationConditionNumbers.m`

2.3 Stability

Well-posedness and the condition number(s) only apply to the mathematical problem. When solving the mathematical problem numerically on a computer, we need to be aware that due to floating point arithmetic rounding errors are introduced. For example, the irrational number π is represented in double precision by these 16 digits:

`3.141592653589793.`

Again, the question is how does the *algorithm* propagate these rounding errors? This is addressed by the concept of stability. There are several different concept of stability. We focus on one concept related to the error in the output and one related to the error in the input.

Analogously, to the mathematical problem, we will denote the algorithm with $\tilde{f} : V \to W$, i.e. another map between the same input and output space as for the mathematical problem. The absolute and relative *forward errors* are defined by

$$\|F(x) - \tilde{F}(x)\|_W \qquad \text{and} \qquad \frac{\|F(x) - \tilde{F}(x)\|_W}{\|F(x)\|_W},$$

respectively. The absolute and relative *backward errors* are given by

$$\beta \qquad \text{and} \qquad \frac{\beta}{\|x\|_V}$$

where

$$\beta = \inf \left\{ \|x - \tilde{x}\|_V \mid \tilde{F}(x) = F(\tilde{x}) \right\}.$$

The backward error is the error between the original input x and some perturbed input $\tilde{x}$. This perturbation is determined by asking which perturbed input one needs to supply to the exact problem to obtain the same output like the algorithm. This question can possibly not be answered in a unique way. Hence, we take the infimum of all such possible input errors.

Now an algorithm is called *forward stable* if the forward error divided by the condition number is "small". It is called *backward stable* if the backward error is small for all inputs x.

2.4 Punchline

So in short, while well-posedness and the condition number belong to the underlying mathematical problem (and are thus independent from the numerical methods ones uses to solve it), stability is inherent to the choice of the algorithm.

3 Floating point arithmetic

3.1 The first principle of floating point arithmetic

It is a well-known fact that there are infinitely many real numbers in the interval $[1, 2]$. One might argue this is a mathematical approximation to a world which is often perceived as continuous – when in fact it isn't, consisting of discrete atoms and molecules. At sea level a cubic meter of air contains roughly $2.5 \cdot 10^{25}$ molecules, i.e. roughly 10^9 molecule per meter. There is no hope to store infinitely many points on a computer. The IEEE-754 double precision floating point standard [13] approximates the interval $[1, 2]$ by exactly $2^{52} \approx 4.5 \cdot 10^{15}$ evenly spaced points, namely

$$1, 1 + 2^{-52}, 1 + 2 \cdot 2^{-52}, 1 + 3 \cdot 2^{-52}, \ldots, 2. \tag{3.1}$$

This means that computers work on scale a million times finer than the physical world. So the discrete set of numbers that computers use to approximate $[1, 2]$ is unbelievably fine.

The interval $[2, 4]$ is approximated again by 2^{52} points, namely (3.1) multiplied by two,

$$2, 2 + 2^{-51}, 2 + 2 \cdot 2^{-51}, 2 + 3 \cdot 2^{-51}, \ldots, 4.$$

Thus since the interval length has increased, also gaps have grown. In general, the interval $[2^j, 2^{j+1}]$ is represented by (3.1) multiplied by 2^j. Figure 3.1 shows the distribution of floating numbers schematically.

Figure 3.1: Schematic distribution of floating point numbers

The advantage of increasing the gaps for larger intervals is that these gaps are in a relative sense never larger than the so-called *machine precision*

$$\varepsilon_M = 2^{-52} \approx 2.2204 \cdot 10^{-16}.$$

This leads us directly to the *first principle of floating point arithmetic*. Suppose we have some real number $x \in \mathbb{R}$ which is bounded away from negative and positive infinity in such a way that we can approximate it with a finite double precision floating point number. We obtain this nearest neighbor floating point approximation denoted with $fl(x)$ via rounding. There are different rounding modes (round to nearest, up, down, toward zero). The most common method is *round to nearest, ties to even* or *bankers' rounding*, which would round 3.5 as well as 4.5 to 4. Assuming this rounding mode, the IEEE-754 standard guarantees that the relative error between x and its floating point approximation is always bounded by half of the relative gaps

$$\left| \frac{x - fl(x)}{x} \right| \leq \frac{\varepsilon_M}{2} =: u, \tag{3.2}$$

where u is often referred to as the *unit round-off*. Equivalently, we can reformulate the principle to saying there exists some ε with $|\varepsilon| \leq u$ such that

$$fl(x) = x(1 + \varepsilon).$$

This means that the nearest floating point neighbor of any real number $x \in \mathbb{R}$ which is bounded away from the infinities is exact within a factor $1 + \varepsilon$.

3.2 Composition of a floating point number

Let us focus on the internal mechanics of double precision floating point numbers. Any double-precision floating point number is represented by

$$\text{sign} \cdot \text{mantissa} \cdot 2^{\text{exponent-1023}}. \tag{3.3}$$

It uses 64 bits of computer memory, where the sign occupies one bit, the mantissa 52 and the exponent 11 bits. Usually, the sign, mantissa and exponent are given as binary numbers since this is the natural language of a computer. However, sometimes to shorted the notation we prefer decimal numbers. To avoid confusion we use the base as a subindex so that for example

$$11_2 = 3_{10} \qquad \text{and} \qquad 11_{10} = 1011_2.$$

If the first digit in the mantissa is one, for example

$$\text{mantissa} = 1.111010\ldots$$

the floating point number is called *normalized*. Usually one prefers to change the order of magnitude of the floating point number by adjusting the exponent. However, in order to also be able to represent numbers very close to zero, one allows *denormalized* floating pointing number, where the mantissa begins with a zero,

$$\text{mantissa} = 0.111010\ldots.$$

In any case the digit left of the binary point is not stored separately and all 52 bits are used for the fractional part of the mantissa. For this reason it is often referred to as the *hidden bit*. Since the exponent occupies 11 bits of memory, two of which are reserved, there are a total of $2^{11} - 2 = 2046$ possible values for the exponent. One introduces an offset (or bias) of 1023 to be able to represent small numbers as well. Thus the biased (decimal) range of the exponent is

$$-1022, -1021, \ldots, 0, 1, \ldots, 1023.$$

The computer stores a floating point number in the order sign, exponent and mantissa. So for example $1 = 1 \cdot 2^0$ is represented by

$$\texttt{0 01111111111 0000000000000000000000000000000000000000000000000000}_2.$$

The exponent 00000000000_2 is reserved to represent either a signed zero for vanishing mantissa $= 0$ or denormalized numbers for non-vanishing mantissa $\neq 0$. Furthermore, the exponent 11111111111_2 is reserved to represent either the

infinities ($\pm$`inf`) for mantissa $= 0$ or not a number (`nan`) for mantissa $\neq 0$. This is quite a useful feature as when writing code

```
exp(3000), exp(-3000), 0/0
```

result in

```
inf, 0, nan.
```

The first output is an example of *overflow*, the second one an example for *underflow*. These exceptions are carefully defined in the IEEE standard. Since the set of floating point numbers is a finite, there exist smallest/largest numbers as well as normalized numbers closest to zero. One can easily verify that these numbers are given by $\pm 1.79769 \cdot 10^{308}$ and $\pm 2^{-1022} \approx \pm 2.22507 \cdot 10^{-308}$.

3.3 The second principle of floating point arithmetic

The beauty of the IEEE-754 standard is that it not just properly defines floating point numbers, rounding modes and exception handling. It also dictates upper error bounds when using basic arithmetic operations on a computer. This is leads us to the *second principle of floating point arithmetic*.

Denote with $\oplus, \ominus, \odot, \oslash$, the computer implementations of addition, subtraction, multiplication and division. It is clear that even if x and y are representable as exact floating point numbers, this is not necessarily true for the result of an arithmetic operation. Take $2^2 + 2^{-52}$, for example, which would be rounded in floating point arithmetic to 4. Hence, the second principle of floating point arithmetic demands that the computer implementations shall give the same result as first adding the two floating point numbers x to y in exact arithmetic and rounding afterwards, i.e.

$$\begin{aligned} x \oplus y &= fl(x + y), & x \ominus y &= fl(x - y), \\ x \odot y &= fl(x \cdot y), & x \oslash y &= fl(x/y). \end{aligned}$$

From these properties, we instantly derive with (3.2) the second principle of floating point arithmetic

$$x \circledast y = (x * y)(1 + \delta) \tag{3.4}$$

for $* \in \{+, -, \cdot, /\}$ some δ which satisfies $|\delta| \leq u$. The second principle of floating point arithmetic shows that the worst relative error that basic operations introduce is again on the order of the unit round-off. This is the best we can hope for. A word of warning: Of course this does not imply that repeated applications of these operations may not accumulate. For example, in Matlab

```
1e5*sin(pi) = 1.224646799147353e-11.
```

Similarly, it is dangerous to subtract numbers of the same size. The result of the following calculation

```
1.000000000001 - 1 = 1.000088900582341e-12
```

is accurate only up to five digits. This loss in precision is called *cancellation*, which is however not the result of an unstable algorithm but of the ill-conditioning of the underlying mathematical problem (in this case the subtraction of numbers which are almost of the same size). It is equally dangerous to add two numbers of greatly varying size as the calculation

```
sqrt(1e-16 + 1) - 1 = 0.
```

demonstrates.

The advantage of the scientific notation (3.3) as opposed to fixed-point notation is obvious. Suppose we use a fixed point notation with 5 integer and 6 fractional digits in decimal arithmetic. Then

π	is approximated by	00003.141592,
$\pi/10000$	is approximated by	00000.000314,
10000π	is approximated by	31415.926535,

showing that the amount of significant digits depends on the order of magnitude of the number.

The interested reader may find a lot more information on floating point arithmetic in the books by Overton [22] and Higham [17]. Online there are also IEEE calculators which are fun to play around with [5].

Demo

- FloatingPointArithmetic.m

4 Interpolation

4.1 What is interpolation?

Interpolation is the art of approximating discrete data by continuous functions. This is best explained with an example: Suppose we are given the following discrete data set

x	0	2.1429	4.2857	6.4286	8.5714	10.7143	12.8571	15
y	0	0.1504	-0.0470	0.0034	0.0101	-0.0083	0.0017	0.0029

.

Before we do anything else we should visualize this data set, the red dots in Figure 4.1.

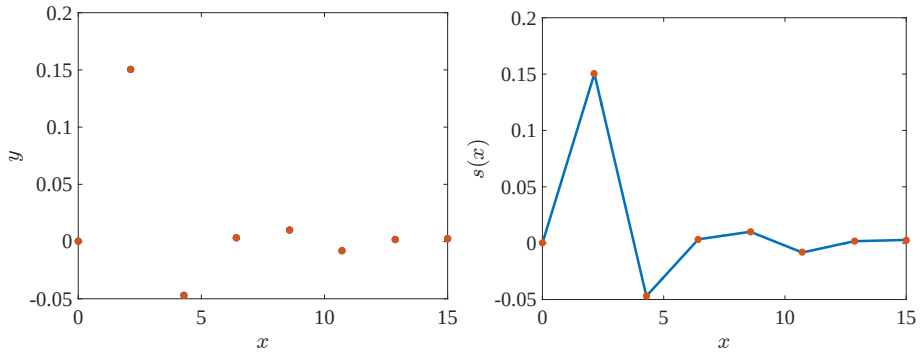


Figure 4.1: Visualization of the data set (left) and linear spline (right) through eight data points ("connect the dots")

The idea is now to find a continuous function which *interpolates* (passes through) these red dots. The advantages of such a function are obvious: We can evaluate it in the gaps between nodes or even use it to integrate or differentiate *discrete* data. It is by no means clear how the "correct" interpolant should look like. The final six data points seem to follow some pattern, which the second one clearly disobeys.

There are now several possibilities. For example, one might construct an interpolant by "connecting the dots". Mathematically, this corresponds to a linear spline [6]. The result is shown in Figure 4.1.

Possibly, one prefers a smoother approximation of the data sets. It is possible to use radial basis functions [20, 28, 11, 4] to interpolate data sets (here narrow Gaussians) like in Figure 4.2. In fact very many other types of radial basis functions can be used but we will not go into detail here.

Instead we will focus on a very particular class, namely polynomial interpolants. Figure 4.2 shows the interpolation polynomial of seventh order. In the following section, we will indeed see that there is only one of such polynomials. Polynomials have the advantage that they can be easily differentiated and integrated. This idea is indeed exploited in the context of the numerical solution of ordinary differential equation as well as numerical integration (quadrature).

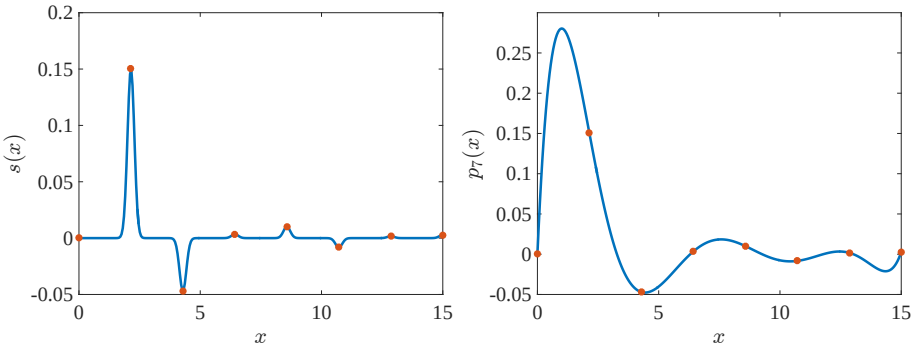


Figure 4.2: RBF interpolant with narrow Gaussians (left) and polynomial interpolant (right) through eight data points

Demos

1. `DataForInterpolants.m`
2. `PolynomialInterpolation.m`
3. `SplineInterpolation.m`
4. `RBFInterpolation.m`

4.2 Polynomial interpolation

We will focus first on polynomial interpolation, a well-established subject in numerical analysis. Let us introduce some notation. We denote with

$$\mathcal{P}_n := \left\{ a_n x^n + a_{n-1} x^{n-1} + \dots a_1 x + a_0 \mid a_i \in \mathbb{R} \text{ for } 0 \le i \le n \right\}$$

the set of all polynomials of degree not larger than $n \in \mathbb{N}_0$. Thus for example $x^3 + 2 \in \mathcal{P}_3 \subseteq \mathcal{P}_{3000}$. In $\mathcal{P}_n$, we can prescribe then $n+1$ free parameters $a_n, \dots, a_0$. We determine these from the *interpolation conditions*

$$p_n(x_i) = y_i \tag{4.1}$$

for $0 \le i \le n$ and $p_n \in \mathcal{P}_n$. The pairwise distinct *interpolation nodes* x_i shall lie in some interval $[a, b]$. The corresponding *data values* y_i are real numbers, though most of the theory generalized to complex numbers as well. We adhere to this notation for the rest of the chapter. Sometimes we assume that the data values are sampled from some unknown function $f \in C[a, b]$, i. e.

$$p_n(x_i) = f(x_i).$$

The conditions (4.1) abstractly describe what we had demanded previously graphically: that the function p_n passes through the data points y_i.

As an example, we briefly look at the important special case of linear interpolation polynomials. In this case, it can easily be seen that

$$p_1(x) = \frac{x - x_0}{x_1 - x_0} y_1 + \frac{x - x_1}{x_0 - x_1} y_0 = \frac{y_1 - y_0}{x_1 - x_0} x + \frac{x_1 y_0 - x_0 y_1}{x_1 - x_0} \tag{4.2}$$

passes through the data (x_0, y_0) and (x_1, y_1). It is easy to see that any interpolation polynomial satisfying (4.1) for given nodes and data values has to be unique.

Theorem 3 (Uniqueness of the interpolation polynomial)**.** *The interpolation polynomial p_n satisfying (4.1) is unique.*

Proof. Suppose there are two interpolation polynomials p and q in $\mathcal{P}_n$ which satisfy (4.1). Then the polynomial $r := p - q \in \mathcal{P}_n$ has $n + 1$ zeros,

$$r(x_i) = p(x_i) - q(x_i) = y_i - y_i = 0,$$

for $0 \leq i \leq n$. However, since r has maximal degree n it cannot have more than n zeros – unless it is the zero polynomial. This implies that $p = q$. Hence, the interpolation polynomial is unique. □

The proof shows only uniqueness, it is not clear whether it is actually always possible to find such a polynomial. Of course one can ask easily questions which do not have a solution, e.g. find an $x \in \mathbb{R}$ which satisfies

$$x + 2 = x + 3.$$

Existence (without doubt) and uniqueness (most often) are very important features that any interpolant should satisfy. Incidentally, also the other interpolation methods introduced at the beginning of the chapter provably satisfy both properties.

For this reason, we present another proof which also includes the existence of the polynomial interpolant. The key idea is to show the uniqueness and existence of the interpolation polynomial with the help of linear algebra.

Theorem 4 (Uniqueness and existence of the interpolation polynomial)**.** *The interpolation polynomial p_n satisfying (4.1) exists and is unique.*

Proof. For $p_n(x) = \sum_{j=0}^{n} a_j x^j \in \mathcal{P}_n$, the interpolation conditions (4.1) can be restated as a linear system of the form

$$\begin{pmatrix} 1 & x_0 & x_0^2 & \dots & x_0^n \\ 1 & x_1 & x_1^2 & \dots & x_1^n \\ 1 & x_2 & x_2^2 & \dots & x_2^n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \dots & x_n^n \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_n \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}. \tag{4.3}$$

The matrix is called the *Vandermonde* matrix and we denote it with V_n. If it is nonsingular, we obtain the existence of a unique coefficient vector $(a_0, \dots, a_n)^T \in \mathbb{R}^{n+1}$, which implies that the corresponding interpolation polynomial p_n is unique.

We show the nonsingularity of the Vandermonde matrix by computing its determinant, which is given by

$$\det(V_n) = \prod_{0 \leq i < j \leq n} (x_j - x_i). \tag{4.4}$$

The notation means that for V_3 for example, we obtain

$$\det(V_3) = (x_3 - x_2)(x_3 - x_1)(x_3 - x_0)(x_2 - x_1)(x_2 - x_0)(x_1 - x_0).$$

This particular structure of the determinant implies, we are done since the nodes are assumed to be pairwise distinct (thus the determinant is nonzero, the Vandermonde matrix is nonsingular, the coefficient vector is unique and so is the interpolant).

In order to show (4.4), we first note that the determinant is a homogeneous polynomial of order $n(n+1)/2$ since by Laplace's formula every summand in the determinant is a product of a zeroth power of some variable, times the first power of another one and so on. (A homogeneous polynomial is a polynomial whose nonvanishing terms all have the same degree, e.g. $x^3 + 5x^2y$ is a homogeneous polynomial of degree 3.) Thus, the determinant of the Vandermonde is a homogeneous polynomial of order $0+1+\ldots+n = n(n+1)/2$. For $x_i = x_j$ with $i \neq j$ two rows in the matrix would agree, leading to a vanishing determinant. Thus, the determinant includes a factor of the form $(x_j - x_i)$. Doing this for all possible combinations implies that the determinant is of the form

$$\det(V_n) = q \prod_{0 \leq i < j \leq n} (x_j - x_i),$$

where q is some remainder polynomial. However, since the left-hand side is also a homogeneous polynomial of degree $n(n+1)/2$, we deduce that q is a constant. In order to determine it, we realize that the coefficient of the leading diagonal $\prod_{j=0}^{n} x_j^j$ in the determinant (due to Laplace's formula) and the right-hand side after expansion is in both cases one. Hence, also $q = 1$. □

The linear system (4.3) is ill-conditioned. However, there are other (better conditioned) ways to obtain the interpolant. The trick is to use different bases for the space $\mathcal{P}_n$. This will also remedy the fact that the previous existence proof was not constructive – we have not actually constructed an interpolation polynomial.

Demos

1. `PolynomialInterpolationVandermonde.m`

4.3 Two disguises of the same interpolation polynomial

There are two well-known representations of the unique interpolation polynomial named after *Lagrange* and *Newton*. They result from a change of basis. The proof of Theorem 4 used the standard monomial basis which is rarely the best choice.

4.3.1 The Lagrange and barycentric representations

We define the Lagrange basis polynomials

$$\ell_j(x) := \prod_{\substack{i=0 \\ i \neq j}}^{n} \frac{x - x_i}{x_j - x_i} = \frac{(x - x_0)}{(x_j - x_0)} \cdots \frac{(x - x_{j-1})}{(x_j - x_{j-1})} \frac{(x - x_{j+1})}{(x_j - x_{j+1})} \cdots \frac{(x - x_n)}{(x_j - x_n)}. \tag{4.5}$$

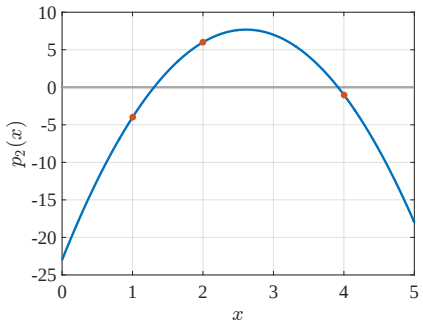


Figure 4.3: Polynomial interpolant p_2 through three data points.

They can be considered as another basis of the space of polynomials $\mathcal{P}_n$ (apart from the standard monomial one) due to the useful properties that $\ell_j(x_j) = 1$ and $\ell_j(x_i) = 0$ for $j \neq i$. Both equations can be summarized with the *Kronecker delta* as follows

$$\ell_j(x_i) = \delta_{ij} := \begin{cases} 1 & \text{if } i = j, \\ 0 & \text{if } i \neq j. \end{cases} \tag{4.6}$$

Then the interpolation polynomial satisfying (4.1) is given by

$$p_n(x) = \sum_{j=0}^{n} y_j \ell_j(x) \tag{4.7}$$

since the interpolation polynomial is unique and from (4.6) we see that

$$p_n(x_i) = \sum_{j=0}^{n} y_j \ell_j(x_i) = \sum_{j=0}^{n} y_j \delta_{ij} = y_i$$

for $0 \leq i \leq n$. Note that our assumptions of pairwise distinct interpolation nodes implies that the basis polynomials are always well-defined (we do not divide by zero).

Maybe it's best to explain this way to interpolate with a picture. Suppose we want to find the interpolant p_2 through the data points $(1, -4), (2, 6)$ and $(4, -1)$ shown in Figure 4.3. In Figure 4.4 we show the Lagrange approach of doing this: First find the basis functions that is one at one node and vanishes for the others, then scale these basis polynomials with the given y_j data and finally add these scaled basis polynomials.

In the proof of Theorem 4, we used a monomial basis for the polynomial space $\mathcal{P}_n$. If we applied the interpolation conditions (4.1) to the Lagrange representation (4.7), the matrix in the corresponding linear system becomes due to (4.6) the identity matrix. So solving this linear system incurs no computational cost – instead it is transferred to the computation of the Lagrange basis polynomials.

Incidentally, explicitly stating the interpolation polynomial in its Lagrange representation (4.7) gives another proof of its existence.

Though it has theoretical advantages to define the interpolation polynomial via (4.7), there are at least three commonly mentioned disadvantages [2]:

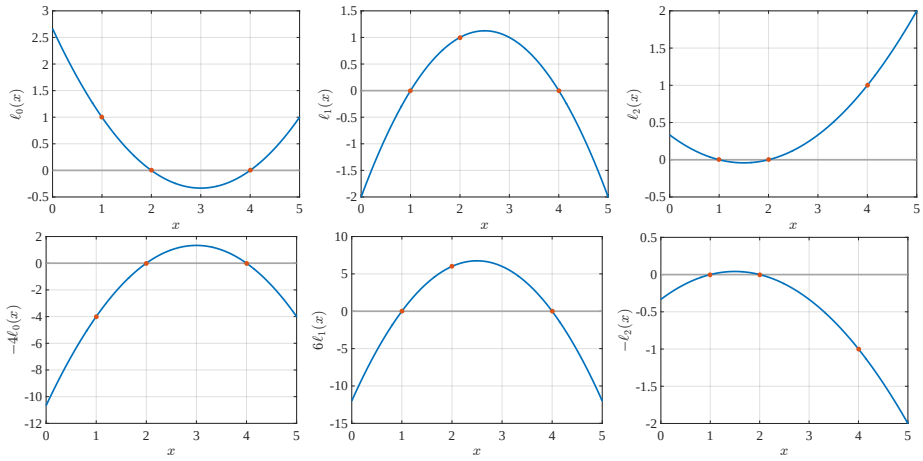


Figure 4.4: Basis polynomials (first row) and their scaled version (second row).

1. Each evaluation of the interpolant via (4.7) takes $\mathcal{O}(n^2)$ operations:

   - $2n(n+1)$ subtractions, $2(n-1)(n+1)$ multiplications and $n(n+1)$ divisions to compute all Lagrange basis polynomials $\ell_j(x)$ for $0 \leq j \leq n$.
   - $n+1$ additions as well $n+1$ multiplications to assemble p_n via (4.7).

   Note the n^2 terms will dominate for large degrees n, so we can neglect the other ones.

2. There are no "storable" quantities independent of x in (4.7). Suppose we have already computed an interpolant through the data $(x_0, y_0), \ldots, (x_n, y_n)$ and evaluate it at 3. Later we decide to add another data pair (x_{n+1}, y_{n+1}). To evaluate $p_{n+1}(3)$, we need to redo all the calculations!

3. The computation is numerically unstable.

For this reason, we rewrite the Lagrange representation of the interpolant. Defining the *nodal function*

$$\ell(x) = (x - x_0)(x - x_1)\ldots(x - x_n) \tag{4.8}$$

and the *barycentric weights*

$$w_j = \frac{1}{\prod_{\substack{i=0 \\ i \neq j}}^{n} (x_j - x_i)} \tag{4.9}$$

for $0 \leq j \leq n$, we can rewrite the Lagrange basis polynomials to

$$\ell_j(x) = \ell(x) \frac{w_j}{x - x_j}.$$

Note that since $\ell(x)$ is independent from the summation index, we obtain

$$p_n(x) = \ell(x) \sum_{j=0}^{n} y_j \frac{w_j}{x - x_j} \tag{4.10}$$

for $x \neq x_i$ with $0 \leq i \leq n$. This formula is often referred to as *first form of the barycentric interpolation formula* [25, 2]. The barycentric weights are now storable quantities which are independent of the variable x. Once having computed the interpolant in $\mathcal{O}(n^2)$ operations, we can evaluate it in $\mathcal{O}(n)$ operations – thus gaining an order of magnitude compared to before. Moreover, the barycentric weights are independent of the data y_i, meaning once we have them, we can interpolate different data on the same set of nodes very efficiently.

Updating the interpolant is now also possible. Adding another node to (4.9) requires

- $n+1$ subtractions and $n+1$ divisions for dividing each w_j for $0 \leq j \leq n$ with $x_j - x_{n+1}$.
- Computing w_{n+1} with another $n+1$ subtractions and divisions.

Updating is hence possible in $\mathcal{O}(n)$ operations. There is a more symmetric variant of the barycentric interpolation formula. Interpolating the constant function 1 and using (4.10), we obtain

$$1 = \sum_{j=0}^{n} \ell_j(x) = \ell(x) \sum_{j=0}^{n} \frac{w_j}{x - x_j}.$$

Dividing (4.10) with this expression, yields after canceling $\ell(x)$ the *second form of the barycentric interpolation formula*

$$p_n(x) = \frac{\displaystyle\sum_{j=0}^{n} y_j \frac{w_j}{x - x_j}}{\displaystyle\sum_{j=0}^{n} \frac{w_j}{x - x_j}} \tag{4.11}$$

for $x \neq x_i$ with $0 \leq i \leq n$. It is possible to show that the first barycentric formula is unconditionally stable and the second one is stable too if the nodes are appropriately clustered [18].

4.3.2 The Newton representation

The barycentric interpolation formulas helped to ease the computational drawbacks of Lagrange representation of the interpolation polynomial. The Newton representation

$$p_n(x) = c_0 + c_1(x - x_0) + \ldots + c_n(x - x_0)(x - x_1)\ldots(x - x_{n-1}) \tag{4.12}$$

is another possibility to efficiently compute the interpolant. Here the c_i are some coefficients which yet have to be determined. Similarly, to the Lagrange basis polynomials, we introduce the Newton basis polynomials

$$n_0(x) = 1 \qquad \text{and} \qquad n_j(x) = \prod_{i=0}^{j-1} (x - x_i) \tag{4.13}$$

for $1 \leq j \leq n$. Then (4.12) can be shortened to

$$p_n(x) = \sum_{j=0}^{n} c_j n_j(x).$$

Note that even though it might seem that both representations (4.7) and (4.12) are different, the uniqueness of the interpolation polynomial guarantees that they must agree. They are really two disguises of the same polynomial. The coefficients c_i can be determined from (4.1) by solving the linear system

$$\begin{aligned}
p_n(x_0) &= c_0 &&= y_0 \\
p_n(x_1) &= c_0 + c_1(x_1 - x_0) &&= y_1 \\
&\vdots &&\vdots \\
p_n(x_n) &= c_0 + c_1(x_n - x_0) + \ldots + c_n(x_n - x_0)(x_n - x_1)\ldots(x_n - x_{n-1}) &&= y_n
\end{aligned}$$

which in matrix form looks like

$$\begin{pmatrix}
1 & 0 & & \ldots & 0 \\
1 & x_1 - x_0 & 0 & \ldots & 0 \\
1 & x_2 - x_0 & (x_2 - x_0)(x_2 - x_1) & & 0 \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
1 & x_n - x_0 & (x_n - x_0)(x_n - x_1) & \ldots & \prod_{j=0}^{n-1}(x_n - x_j)
\end{pmatrix}
\begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_n \end{pmatrix}
=
\begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{pmatrix}.$$

Note that the change of basis has significantly reduced the computational cost of solving the linear system compared to (4.3). This time the matrix is lower-triangular which implies that we have to solve the unknown coefficients iteratively from

$$\sum_{j=0}^{i} c_j n_j(x_i) = y_i$$

for $0 \leq i \leq n$. After rearranging and noting that $n_i(x_i) \neq 0$, we obtain

$$c_i = \frac{1}{n_i(x_i)} \left[y_i - \big(1, n_1(x_i), \ldots, n_{i-1}(x_i)\big) \begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_{i-1} \end{pmatrix} \right],$$

which costs for $1 \leq i \leq n$

- one division and
- $2(i-1)$ multiplications: $i-1$ multiplications from the scalar product ($1 \cdot c_0$ does not count) and another $i-1$ multiplications from computing $n_i(x_i)$ which we do in such a way that we do not forget the lower dimensional Newton polynomials evaluated at x_i.

Thus in total, we have n divisions and $\sum_{i=1}^{n} 2(i-1) = n(n-1)$ multiplications. There is a cheaper algorithm which only needs $n(n-1)/2$ divisions, which we discuss next.

Divided differences

We introduce the *divided differences* recursively by defining

$$f[x_i] = y_i$$

for $0 \leq i \leq n$ as well as

$$f[x_i, \ldots, x_{i+k}] = \frac{f[x_{i+1}, \ldots, x_{i+k}] - f[x_i, \ldots, x_{i+k-1}]}{x_{i+k} - x_i}$$

for $0 \leq i < i + k \leq n$. Inductively, it can be shown that then

$$\begin{aligned} c_0 &= f[x_0] \\ c_1 &= f[x_0, x_1] \\ c_2 &= f[x_0, x_1, x_2] \\ &\vdots \\ c_n &= f[x_0, x_1, x_2, \ldots, x_n]. \end{aligned}$$

We can collect the divided differences in the following tableau

$$\begin{array}{ccccccc} & & & x_0 & y_0 = f[x_0] & & \\ & & x_1 - x_0 & x_1 & y_1 = f[x_1] & f[x_0, x_1] & \\ & x_2 - x_0 & x_2 - x_1 & x_2 & y_2 = f[x_2] & f[x_1, x_2] & f[x_0, x_1, x_2] \\ x_3 - x_0 & x_3 - x_1 & x_3 - x_2 & x_3 & y_3 = f[x_3] & f[x_2, x_3] & f[x_1, x_2, x_3] \end{array} \quad f[x_0, x_1, x_2, x_3]$$

.

The coefficients we are looking for are in the shaded boxes. We finish this discussion with an example. Suppose we would want to find the parabola which passes through the points $(-1, 2), (0, 1)$ and $(2, 2)$. We compute

$$\begin{array}{cccccc} & & -1 & 2 & & \\ & 1 & 0 & 1 & -1 & \\ 3 & 2 & 2 & 2 & 0.5 & 0.5 \end{array} \quad .$$

Thus, the interpolation polynomial is given by $p_2(x) = 2 - (x+1) + 1/2(x+1)x$. It obviously passes through the data points. An advantage of the divided differences is that a given data set can easily be expanded by an additional point without having to recompute the whole table. One only needs to add a row. A similar advantage we had seen for the barycentric interpolation formulas.

We provide an overview of the different representations.

representation	evaluation	updating	stability	advantage
Lagrange (4.7)	$\mathcal{O}(n^2)$	$--$	no	theoretical tool
Barycentric (4.10) & (4.11)	$\mathcal{O}(n)$	$\mathcal{O}(n)$	yes [18]	w_j independent of y_i
Newton (4.12)	$\mathcal{O}(n)$	$\mathcal{O}(n)$	yes [17]	well-studied

Often one only needs to evaluate the interpolation polynomial at certain points but it is not necessary to compute the whole polynomial. This leads us to the next section.

Neville's algorithm

Similar to the divided differences we may define polynomials recursively (not to be confused with the interpolation polynomial) by setting

$$p_i(x) = y_i$$

for $0 \leq i \leq n$ and

$$p_{i,\ldots,i+k}(x) = \frac{(x - x_i)p_{i+1,\ldots,i+k}(x) - (x - x_{i+k})p_{i,\ldots,i+k-1}(x)}{x_{i+k} - x_i}$$

for $0 \leq i < i + k \leq n$. If

$$p_{i+1,\ldots,i+k} \text{ interpolates } (x_{i+1}, y_{i+1}), \ldots, (x_{i+k}, y_{i+k})$$

and

$$p_{i,\ldots,i+k-1} \text{ interpolates } (x_i, y_i), \ldots, (x_{i+k-1}, y_{i+k-1})$$

then the polynomial on the right-hand side

$$p_{i,\ldots,i+k} \text{ interpolates } (x_i, y_i), \ldots, (x_{i+k}, y_{i+k}).$$

With a similar divided difference type of scheme we can compute the value of an (unknown) interpolation polynomial at some point x via the tableau

$$\begin{array}{ccccccccc}
 & & & x - x_0 & x_0 & y_0 = p_0(x) & & & \\
 & & x_1 - x_0 & x - x_1 & x_1 & y_1 = p_1(x) & p_{0,1}(x) & & \\
 & x_2 - x_0 & x_2 - x_1 & x - x_2 & x_2 & y_2 = p_2(x) & p_{1,2}(x) & p_{0,1,2}(x) & \\
x_3 - x_0 & x_3 - x_1 & x_3 - x_2 & x - x_3 & x_3 & y_3 = p_3(x) & p_{2,3}(x) & p_{1,2,3}(x) & p_{0,1,2,3}(x)
\end{array}$$

For example, suppose we want to evaluate the parabola that passes through the points $(1, 1), (2, 2)$ and $(4, 1)$ at $x = 3$. Then the scheme

$$\begin{array}{ccccccc}
 & & 3 - x_i & x_i & y_i & & \\
\hline
 & & 2 & 1 & 1 & & \\
 & 1 & 1 & 2 & 2 & \frac{2 \cdot 2 - 1 \cdot 1}{2 - 1} = 3 & \\
3 & 2 & -1 & 4 & 1 & \frac{1 \cdot 1 + 1 \cdot 2}{4 - 2} = \frac{3}{2} & \frac{2 \cdot 3/2 + 1 \cdot 3}{3} = 2
\end{array}$$

yields that the interpolation polynomial at $x = 3$ takes the value $p(3) = 2$.

Horner's method

We show how (4.12) can be evaluated efficiently. Evaluating a polynomial

$$p(x) = a_n x^n + \ldots + a_1 x + a_0$$

carelessly may incur a cost of

- $n-1$ multiplications for the monomials $x^2, \ldots, x^n$,
- n coefficient multiplications,
- n additions/subtractions.

Rewriting the polynomial to

$$p(x) = (\ldots(a_n x + a_{n-1})x + \ldots)x + a_0$$

results into an algorithm which only needs

- n multiplications
- n additions/subtractions.

The savings are drastic and noticeable when computing manually. Suppose we want to evaluate the polynomial

$$p(x) = -x^4 + 2x^3 + 3x^2 - 2x + 5$$

at $x = 3$. Doing it without any simplifications one calculates

$$p(3) = -3^4 + 2 \cdot 3^3 + 3 \cdot 3^2 - 2 \cdot 3 + 5 = -81 + 54 + 27 - 6 + 5 = -1,$$

which is very error prone. However, Horner's method makes it easy. Collecting the coefficients in a table, we compute

a_i	-1	2	3	-2	5
$x = 3$	-1	-1	0	-2	-1

.

Demos

1. `LagrangeInterpolation.m`
2. `LagrangeInterpolationBarycentric.m`
3. `HornersMethod.m`

4.4 Polynomial interpolation is well-posed

Apart from existence and uniqueness polynomial interpolation (in 1D at least) satisfies also the continuous dependence on the given data (x_i, y_i) for $0 \leq i \leq n$. Hence, it is well-posed in the sense of Hadamard.

Theorem 5 (Well-posedness of polynomial interpolation)**.** *Polynomial interpolation is well-posed.*

Proof. Due to Theorem 4, it only remains to show the continuous dependence on the data. We do this by showing that the absolute condition number is finite. Suppose we have two interpolants

$$p_n(x) = \sum_{j=0}^{n} f(x_j)\ell_j(x) \quad \text{and} \quad \tilde{p}_n(x) = \sum_{j=0}^{n} \tilde{f}(x_j)\ell_j(x)$$

through the data $(x_i, f(x_i))$ and $(x_i, \tilde{f}(x_i))$ for $0 \leq i \leq n$, respectively. Here we assumed that the data is generated from two unknown continuous functions $f, \tilde{f} \in C[a,b]$. Of course, the interpolation nodes shall lie in the interval $[a,b]$. Strictly speaking it is not necessary for the proof to assume that the y data is generated from these functions. However, it helps to interpret the linear operator

$$P_n \colon C[a,b] \to \mathcal{P}_n, \qquad P_n f = p_n(x) = \sum_{j=0}^{n} f(x_j)\ell_j(x) \tag{4.14}$$

which assigns every function f its corresponding interpolation polynomial as a linear projection ($P_n p_n = p_n$). We will call this projection the *interpolation operator*. By Theorem 4, we know that this mapping is bijective. If the interpolation operator was bounded, we could write

$$\|P_n\tilde{f} - P_n f\|_\infty \leq \|P_n\|_\infty \|\tilde{f} - f\|_\infty \tag{4.15}$$

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

$$\|P_n\|_\infty = \sup_{0\neq f\in C[a,b]} \frac{\|P_n f\|_\infty}{\|f\|_\infty} =: \Lambda_n \tag{4.16}$$

is the induced operator norm, which is called *Lebesgue constant* Λ_n. The problem is that up to now it is not clear whether the operator norm is finite (in which case the operator would be bounded). Moreover, we point out that $\kappa_a = \|P_n\|_\infty$ as one can see by comparing (4.15) with the definition of the absolute condition number (2.1). So proving the well-posedness of polynomial interpolation boils down to showing that $\Lambda_n < \infty$.

To see this we compute

$$\begin{aligned}\|P_n f\|_\infty &= \max_{x\in[a,b]} \left| \sum_{j=0}^{n} f(x_j)\ell_j(x) \right| \leq \max_{x\in[a,b]} \sum_{j=0}^{n} |f(x_j)||\ell_j(x)| \\ &\leq \max_{x\in[a,b]} |f(x)| \max_{x\in[a,b]} \sum_{j=0}^{n} |\ell_j(x)| = \|f\|_\infty \max_{x\in[a,b]} \sum_{j=0}^{n} |\ell_j(x)|\end{aligned}$$

which implies

$$\Lambda_n = \|P_n\|_\infty \leq \max_{x\in[a,b]} \sum_{j=0}^{n} |\ell_j(x)| < \infty.$$

Thus, the Lebesgue constant is actually finite (though dependent on degree n and the interpolation nodes).

□

The final inequality in the proof can be tightened. We introduce the *Lebesgue function*

$$\lambda_n(x) = \sum_{j=0}^{n} |\ell_j(x)|. \tag{4.17}$$

It's largest value in $[a, b]$ is precisely the Lebesgue constant. We assume it becomes maximal for some $x^* \in [a, b]$ such that

$$\Lambda_n = \sum_{j=0}^{n} |\ell_j(x^*)| = \lambda_n(x^*).$$

Then we choose some $f \in C[a, b]$ with $\|f\|_\infty = 1$ such that $f(x_j) = \text{sign}(\ell_j(x^*)) \in \{-1, 1\}$. Then

$$|P_n f(x^*)| = \left| \sum_{j=0}^{n} f(x_j)\ell_j(x^*) \right| = \sum_{j=0}^{n} f(x_j)\ell_j(x^*) = \sum_{j=0}^{n} |\ell_j(x^*)| = \Lambda_n.$$

Hence, the supremum in (4.16) is attained and

$$\Lambda_n = \|P_n\|_\infty = \max_{x \in [a,b]} \sum_{j=0}^{n} |\ell_j(x)|.$$

So in summary, the Lebesgue constant Λ_n, the absolute (operator) condition number κ_a as well as the norm of the interpolation operator agree!

4.5 Conditioning of polynomial interpolation

When discussing the conditioning of polynomial interpolation some confusion can easily arise. It is paramount to keep two types of conditioning apart:

1. the conditioning of the data values and
2. the conditioning of the coefficients.

Theorem 5 not only showed the well-posedness of polynomial interpolation but also explicitly stated the absolute condition number for the data values. This condition number measures the sensitivity of the interpolant on the input data as stated in (4.14). It does not depend on *how* the interpolant is computed. However, it depends on the degree of the interpolant as well as the interpolation nodes. Figure 4.5 shows that this (at least for equidistant nodes) may lead to

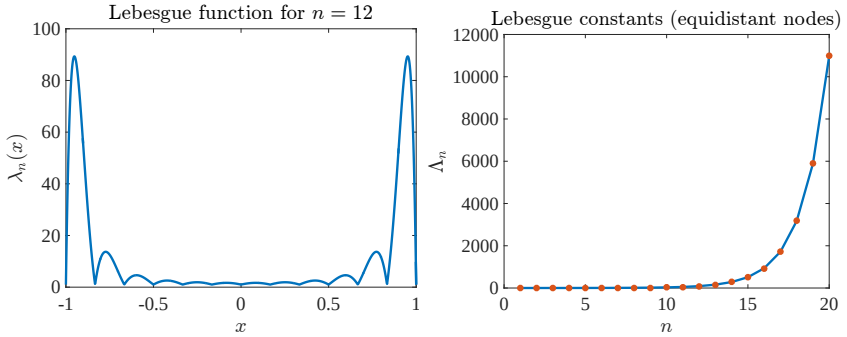


Figure 4.5: Lebesgue function for $n = 12$ (left) and Lebesgue constants (right) for equidistant nodes in the interval $[-1, 1]$

serious difficulties. The picture on the left shows the Lebesgue function (4.17) for $n = 12$. The picture on the right shows how the Lebesgue function grows when the polynomial degree increases. In fact, it can be shown that for equidistant nodes and large n

$$\Lambda_n \sim \frac{2^{n+1}}{e\, n \log n}. \tag{4.18}$$

That is, the Lebesgue constant grows exponentially. This implies that for larger polynomial degree polynomial interpolation in equidistant nodes becomes very fast ill-conditioned. We will remedy this later by studying nonuniform nodes.

On the other hand, the conditioning of the coefficients highly depends on how we construct the interpolant. In other words: How sensitively do the coefficients of the interpolant depend on the input data. Suppose we have some basis $b_0, \ldots, b_n$ for the polynomial space $\mathcal{P}_n$. The interpolation conditions $p_n(x_i) = \sum_{j=0}^{n} a_j b_j(x_i) = f(x_i)$ for $0 \le i \le n$ lead to a linear system

$$Ba = d \iff a = B^{-1}d$$

for $B = (b_j(x_i))_{0 \le i,j \le n}, a = (a_0, \ldots, a_n)^T$ and $d = (f(x_0), \ldots, f(x_n))^T$. Comparing equation on the right with (4.14), we see that the input is still the data but this time the output consists of the coefficients. Which basis we choose has now a direct impact on the condition (number). We summarize the three cases we have studied:

basis	B	type	conditioning
Monomial	$B = V_n$	Vandermonde matrix	ill
Lagrange (4.5)	$B = (\ell_j(x_i)) = I_n$	identity matrix	perfect
Newton (4.13)	$B = (n_j(x_i))$	lower triangular matrix	acceptable

Even though (4.3) allows in theory to compute the coefficients for any $n \in \mathbb{N}_0$, the numerical solution will easily become corrupted for $n > 8$ since solving the Vandermonde system is ill-conditioned (the entries in the matrix vary greatly, we will study more details concerning the conditioning of linear systems in Chapter 7). As we know now this is not a problem of the mathematical problem we want to solve (finding an interpolant) but only of the strategy we have chosen (using a monomial basis). However, as we have seen there are better conditioned ways to obtain the interpolant, using the other two bases for the space $\mathcal{P}_n$. We only point out once more that one needs to careful when constructing the Lagrange basis to not generate problems with respect to efficiency and stability.

Demos

1. `PlotLebesgueConstants.m`

4.6 Error analysis and node distribution

After interpolating function values on a set of interpolation nodes we would like to study the overall maximal error one can expect.

Suppose we want to interpolate the function $f(x) = e^{-x}$ at the nodes $x = -1/2$ and $x = 1/2$. We can then ask what the maximal error between f and p_1 is in the interval $[-1/2, 1/2]$. Due to (4.2), we know that

$$p_1(x) = (e^{1/2} - e^{-1/2})x + \frac{1}{2}(e^{1/2} + e^{-1/2}).$$

The error between interpolant and error looks like this:

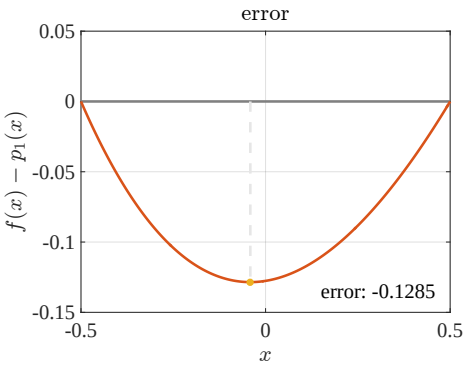


Figure 4.6: Error between $f(x) = e^{-x}$ and its linear interpolant

For a different function, the error will most surely be different, too. So how can we give a bound for the error in terms of the function itself? Since we interpolate, the error must include both factors $(x - x_0)$ and $(x - x_1)$, i.e.

$$f(x) - p_1(x) = (x - x_0)(x - x_1)h(x)$$

where h is some unknown function. Differentiating twice yields

$$\begin{aligned}\frac{d^2}{dx^2}(f(x) - p_1(x)) &= \frac{d^2}{dx^2}(x - x_0)(x - x_1)h(x) \\ &= 2h(x) + (x - x_0)(x - x_1)h'(x)\end{aligned}$$

Since p_1 is a first-order polynomial its second derivative vanishes. For values ξ near x_0 or x_1, the second term on the right-hand side vanishes such that

$$h(\xi) \approx \frac{1}{2}f''(\xi).$$

We can generalize this argument.

Theorem 6. *Let f be an $n+1$ times continuously differentiable function and let p_n be the interpolation polynomial through the points $(x_0, f(x_0)), \ldots, (x_n, f(x_n))$ such that the pairwise distinct nodes lie in the interval $[a, b]$. Then there exists some $\xi = \xi(x) \in [a, b]$ such that*

$$f(x) - p_n(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!}\prod_{i=0}^{n}(x - x_i).$$

Proof. We define the error function

$$r_n(x) = f(x) - p_n(x)$$

as well as

$$g(t) = r_n(t) - \frac{r_n(x)}{\ell(x)}\ell(t)$$

where ℓ is defined as in (4.8). We notice that $t = x$ is a root of g. Moreover, since the pairwise distinct interpolation nodes x_i are roots of the error r_n, the function g mush have at least $n+2$ zeros. Rolle's theorem[1] implies that after differentiating $n+1$ times

$$g^{(n+1)}(t) = r_n^{(n+1)}(t) - \frac{r_n(x)}{\ell(x)}\ell^{(n+1)}(t) = f^{(n+1)}(t) - \frac{r_n(x)}{\ell(x)}(n+1)! \tag{4.19}$$

has some root $\xi = \xi(x) \in [a,b]$. In the above calculation we have used for the final equality just like in the previous example that a polynomial of degree n vanishes after differentiating it $n+1$ times, i.e. $r_n^{(n+1)}(t) = f^{(n+1)}(t)$ as well as the fact that only the leading order term of ℓ, namely x^{n+1}, contributes after differentiation. Inserting the root into (4.19) yields after rearrangement

$$f(x) - p_n(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!}\ell(x)$$

which proves the claim.

□

The previous theorem shows several things: The distribution of the nodes as well as the smoothness of the function one wants to interpolate can make the error smaller or larger. Usually it is enough to find an acceptable error bound. With Theorem 6 we immediately derive the bound

$$\max_{x\in[a,b]} |f(x) - p_n(x)| \leq \max_{\xi\in[a,b]} \frac{|f^{(n+1)}(\xi)|}{(n+1)!} \max_{x\in[a,b]} \left|\prod_{i=0}^{n}(x - x_i)\right|. \tag{4.20}$$

Noting the factorial in the denominator which grows rapidly one might be inclined to ask: Does the error on the left-hand side tend to zero for any function f as the degree n of the interpolation polynomial grows large? Unfortunately, not. Runge [24] presented in 1901 a famous counterexample. For the *Runge function*

$$f\colon [-1,1] \to \mathbb{R}, \qquad f(x) = \frac{1}{1+25x^2}$$

the interpolants on *equidistant nodes* do not converge for increasing n as can be seen in Figure 4.7.

Two effects outweigh the beneficial factorial in the denominator. First, the equidistant node distribution leads to a relatively large absolute value of the nodal function ℓ. Denoting the mesh width with h, it can be bounded by

$$\max_{x\in[a,b]} \left|\prod_{i=0}^{n}(x - x_i)\right| \leq \frac{1}{4}h^{n+1}n!.$$

[1] Any real-valued differentiable function that attains equal values at two distinct points must have a point where the first derivative vanishes.

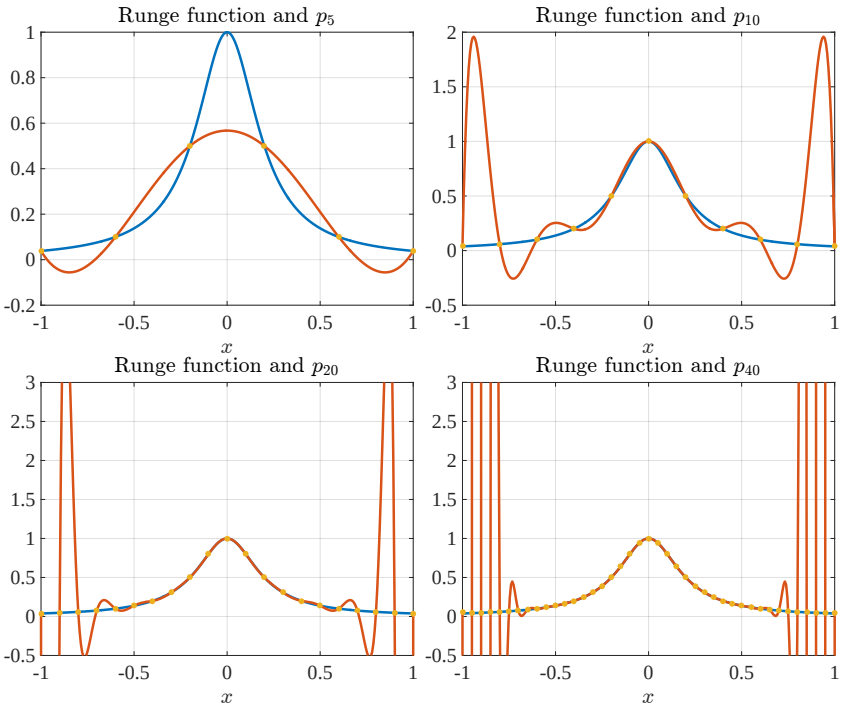


Figure 4.7: The Runge function (blue) and interpolants (red) on equidistant nodes

Thus on uniform meshes we have derived the error bound

$$\max_{x\in[a,b]} |f(x) - p_n(x)| \leq \frac{1}{4(n+1)} h^{n+1} \max_{\xi\in[a,b]} |f^{(n+1)}(\xi)|.$$

Second, the derivatives of the Runge functions increase fast for growing n near the boundaries of the interval. So fast that the bound on the right-hand side does not tend to zero. Of course this does not say anything about the error on the left-hand side. However, the plots clearly suggest that also the maximum error itself does not converge.

One way to improve the bound on the error is to use piecewise polynomials (splines). Another way is to minimize the distribution of nodes in such a way that the maximum norm of the nodal function $\max_{x\in[a,b]} |\ell(x)|$ is minimized. Surprisingly, there is an answer to this question.

One can show that the *Chebyshev nodes* in $[-1, 1]$ defined by

$$x_j = \cos\left(\frac{2j+1}{2(n+1)}\pi\right)$$

with $0 \leq j \leq n$ minimize the maximum norm of the nodal function, yielding

$$\max_{x\in[-1,1]} |\ell(x)| = \frac{1}{2^n}.$$

Also the Lebesgue constant grows now only logarithmically

$$\tfrac{2}{\pi}\log(n+1) + a < \Lambda_n < \tfrac{2}{\pi}\log(n+1) + 1$$

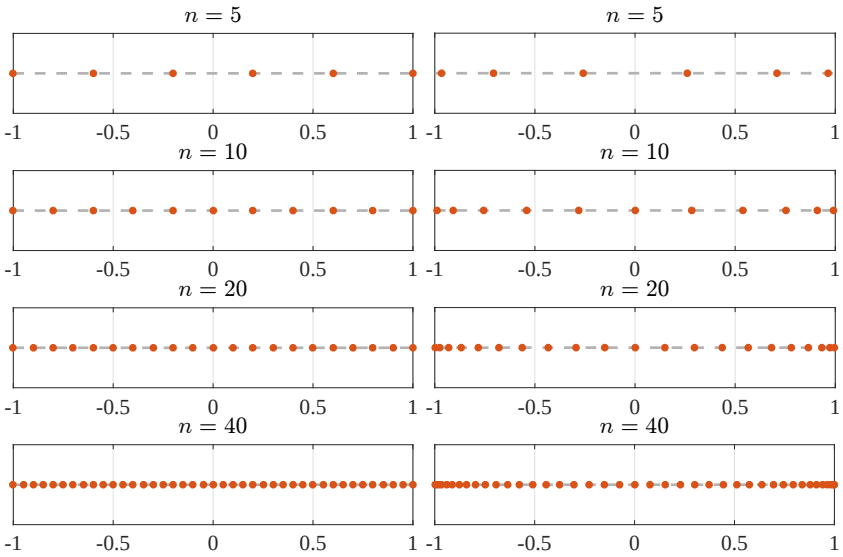


Figure 4.8: Equidistant nodes (left) and Chebyshev nodes (right)

for $a \approx 0.9625$. Even though the Chebyshev nodes are only defined in the interval $[-1, 1]$, we can easily transform them to general intervals of the form $[a, b]$ via

$$h(x) = a + \frac{b-a}{2}(1+x).$$

The visual comparison with equidistant nodes in Figure 4.8 shows that the Chebyshev nodes cluster near the boundaries, i.e. exactly where previously the problems appeared. Figure 4.9 shows the improvement after using Chebyshev nodes. The unwanted oscillations are gone.

It would also be nice to have a measure for the quality (the largest possible error) of an interpolant without having to rely on specific choices for f. Here again the Lebesgue constant comes in handy. Suppose we have a best approximating polynomial p_n^* in $\mathcal{P}_n$ to some function $f \in C[a, b]$, i.e.

$$\|f - p_n^*\|_\infty \leq \|f - q\|_\infty \tag{4.21}$$

for all $q \in \mathcal{P}_n$. Here $\|\cdot\|_\infty$ denotes again the continuous maximum norm. Note that p_n^* is the *closest* polynomial to f in this norm. Among all polynomial of degree at most n, the polynomial p_n^* has the smallest error. The Weierstraß theorem[2] guarantees the existence of such a best approximation. In general, however, it is not known how it looks like. It is important to understand that this best approximating polynomial is *not* necessarily the interpolating polynomial $p_n = P_n f$ for some given nodes. Here P_n is again the interpolation operator defined in (4.14). The following theorem says that we can get close if the Lebesgue constant $\Lambda_n = \|P_n\|_\infty$ is small.

Theorem 7 (Near best approximation)**.** *Let $f \in C[a, b]$ be interpolated by $p_n = P_n f$. Denote with p_n^* the best approximation in the sense of (4.21). Then*

$$\|f - p_n\|_\infty \leq (1 + \Lambda_n)\|f - p_n^*\|_\infty.$$

[2]Any continuous function can be uniformly approximated by polynomials.

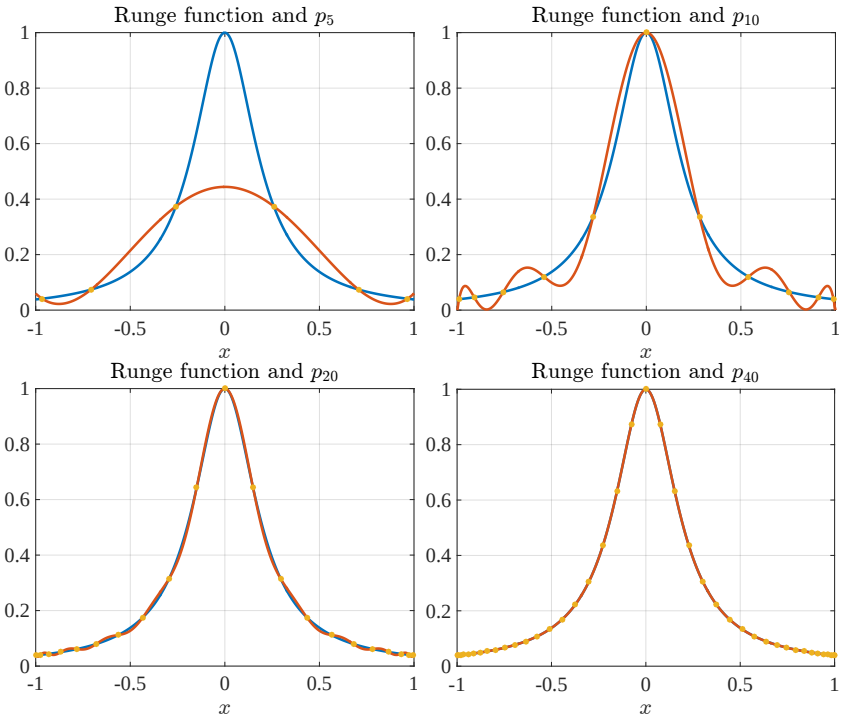


Figure 4.9: The Runge function (blue) and interpolants (red) on Chebychev nodes

Proof. By the triangle inequality, the fact that every polynomial is its own interpolant, the linearity of the interpolation operator and Theorem 5, we obtain

$$\begin{aligned}\|f - p_n\|_\infty &\leq \|f - p_n^*\|_\infty + \|p_n^* - p_n\|_\infty \\ &= \|f - p_n^*\|_\infty + \|P_n p_n^* - P_n f\|_\infty \\ &= \|f - p_n^*\|_\infty + \|P_n(p_n^* - f)\|_\infty \\ &\leq \|f - p_n^*\|_\infty + \|P_n\|_\infty \|p_n^* - f\|_\infty \\ &= (1 + \Lambda_n)\|f - p_n^*\|_\infty.\end{aligned}$$

□

Thus, while a large Lebesgue constant does not necessarily imply that the maximum error is small, a small Lebesgue constant does! In Figure 4.10 we compare the Lebesgue functions for equidistant nodes to the ones for Chebyshev nodes. The result is impressive. For equidistant nodes we see the exponential growth already discussed in (4.18). The growth of the Lebesgue function on Chebyshev nodes is much more moderate. In fact, it can be shown to be logarithmic [19]. Chebfun [10] is a new interesting software package which uses ideas from barycentric Lagrange interpolation on Chebyshev nodes, making it possible to approximate function to very high precision – effectively giving the user the feeling to compute with functions rather than vectors. More theoretical aspects are covered in [15].

Demos

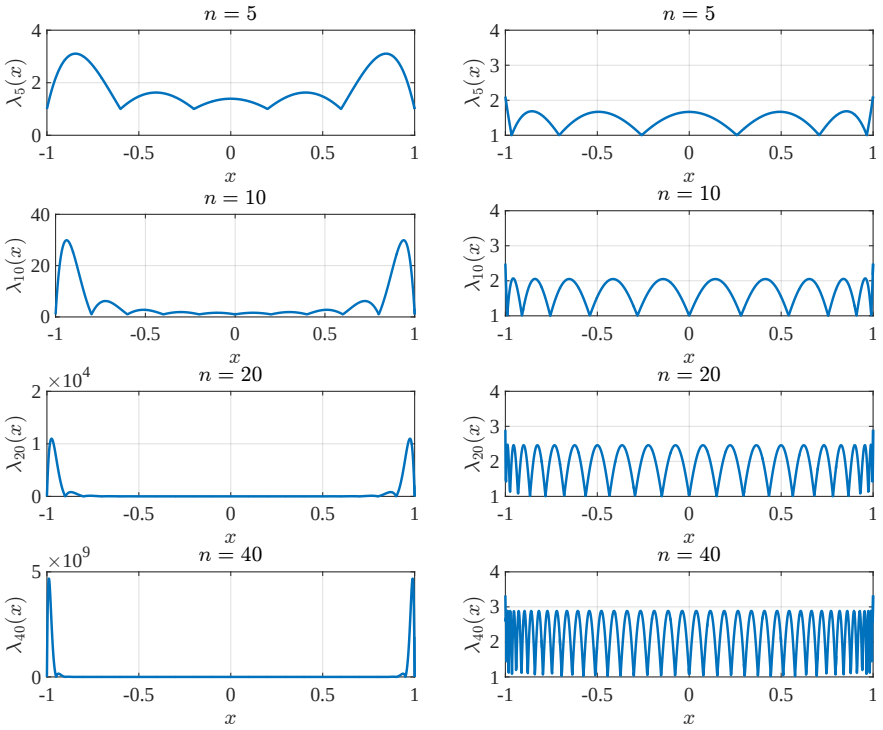


Figure 4.10: Lebesgue function for equidistant nodes (left) and Chebyshev nodes (right)

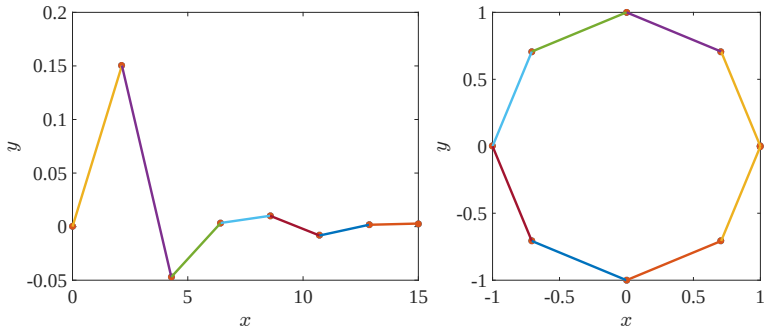


Figure 4.11: A linear spline consists of a sequence of linear polynomials and can even interpolate parametrized curves.

1. `PlotLinearInterpolationPolynomial.m`
2. `PlotRungeFunction.m`
3. `PlotNodeDistribution.m`
4. `PlotRungeFunctionCheb.m`
5. `LebesgueFunction.m`
6. `ChebfunExample.m`

4.7 Spline interpolation

In the introduction of this chapter we already mentioned splines as an alternative method to use for interpolation. A spline is a piecewise polynomial function. We will briefly linear and cubic splines.

4.7.1 Linear splines

The key idea behind linear spline interpolation could not be simpler: Take the data and connect with straight lines. The introductory picture is reproduced in Figure 4.11 highlighting the different components. It can even be used to interpolate data from parametrized curves. Interpolation polynomials cannot achieve this since the nodes have to be distinct. How did we compute it? Again we assume that we have given $n+1$ data points $(x_0, y_0), \ldots, (x_n, y_n)$ where the nodes are ordered $x_0 \leq x_1 \leq \ldots \leq x_n$. The interval $[x_i, x_{i+1}]$ can be mapped to $[0, 1]$ via

$$t(x) = \frac{x - x_i}{x_{i+1} - x_i}. \tag{4.22}$$

Hence, we define the line segment between the data points (x_i, y_i) and (x_{i+1}, y_{i+1}) by

$$Y_i \colon [0,1] \to \mathbb{R}, \qquad Y_i(t) = a_i + b_i t$$

for $0 \leq i \leq n-1$ and some coefficients a_i and b_i that still need to be determined in such a way that

$$Y_i(0) = y_i \qquad \text{and} \qquad Y_i(1) = y_{i+1}. \tag{4.23}$$

Hence, in particular $Y_{i-1}(1) = y_i = Y_i(0)$. But with (4.23) it is immediately clear that

$$a_i = y_i \qquad \text{and} \qquad b_i = y_{i+1} - y_i$$

for $0 \leq i \leq n-1$. That's it! We even do not have to solve a linear system. In fact, the above equations directly imply uniqueness and existence. We define similar to before

$$\ell_j(x) = \begin{cases} \dfrac{x - x_{j-1}}{x_j - x_{j-1}} & \text{if} \quad x \in [x_{j-1}, x_j] \\ \dfrac{x_{j+1} - x}{x_{j+1} - x_j} & \text{if} \quad x \in [x_j, x_{j+1}] \\ 0 & \text{otherwise} \end{cases} \tag{4.24a}$$

for $1 \leq j \leq n-1$ as well as

$$\ell_0(x) = \begin{cases} \dfrac{x_1 - x}{x_1 - x_0} & \text{if} \quad x \in [x_0, x_1] \\ 0 & \text{otherwise} \end{cases} \tag{4.24b}$$

and

$$\ell_n(x) = \begin{cases} \dfrac{x - x_{n-1}}{x_n - x_{n-1}} & \text{if} \quad x \in [x_{n-1}, x_n] \\ 0 & \text{otherwise} \end{cases}. \tag{4.24c}$$

These "hat" functions fulfill a similar role like the Lagrange basis polynomials (4.5) as they also satisfy $\ell_j(x_i) = \delta_{ij}$. We realize that due to uniqueness the linear spline interpolant can be written as

$$s_n(x) := \sum_{j=0}^{n} f(x_j)\ell_j(x).$$

Here we have assumed again that $y_j = f(x_j)$ for some function $f \in C[a,b]$. Another difference to polynomial interpolation becomes apparent: Unlike the Lagrange basis polynomials (4.5), the hat functions (4.24) only contribute locally to the interpolant in the sense that for $x \in (x_j, x_{j+1})$ only ℓ_j and ℓ_{j+1} need to be computed the other hat functions vanish by construction.

Defining again a linear operator S_n from the continuous functions to the space of all linear splines by $S_n f = s_n$, we deduce similar to before

$$\|S_n f\|_\infty \leq \|f\|_\infty \max_{x \in [a,b]} \sum_{j=0}^{n} |\ell_j(x)|.$$

Again equality is attained in the same way as for the polynomial interpolation operator and we define the Lebesgue constant as the induced operator norm

$$\Lambda_{n,1}^s := \|S_n\|_\infty = \max_{x\in[a,b]} \sum_{j=0}^{n} |\ell_j(x)| < \infty$$

where this time the ℓ_j correspond to the hat functions (4.24a), (4.24b) or (4.24c). For these function we can also define the Lebesgue function via (4.17). Due to the special structure of the hat functions, we have

$$1 = \sum_{j=0}^{n} |\ell_j(x)| = \max_{x\in[a,b]} \sum_{j=0}^{n} |\ell_j(x)| = \Lambda_{n,1}^s.$$

We have proven the following theorem.

Theorem 8. *Linear spline interpolation is well-posed and well-conditioned for any node distribution.*

Applying our error result for the interpolation polynomial to linear interpolants gives us the following error estimate.

Theorem 9. *Let $f \in C^2[a,b]$, then*

$$\|f - s_n\|_{\infty,[a,b]} \leq \frac{1}{8} h_{\max}^2 \|f''\|_{\infty,[a,b]},$$

where $h_{\max} := \max_{0\leq i\leq n-1} |x_{i+1} - x_i|$.

Proof. Take some arbitrary $x \in [a,b]$ and suppose it lies in the interval $[x_j, x_{j+1}]$. Then since the spline interpolant consists of localized hat functions

$$|f(x) - s_n(x)| = \left| f(x) - f(x_j)\frac{x_{j+1} - x}{x_{j+1} - x_j} - f(x_{j+1})\frac{x - x_j}{x_{j+1} - x_j} \right|.$$

By Theorem 6, we find some $\xi = \xi(x) \in [x_j, x_{j+1}]$ such that

$$\begin{aligned} |f(x) - s_n(x)| &\leq \frac{1}{2}\left|f''(\xi)(x - x_j)(x - x_{j+1})\right| \\ &\leq \frac{1}{8}(x_{j+1} - x_j)^2 \|f''\|_{\infty,[x_j,x_{j+1}]} \leq \frac{1}{8} h_{\max}^2 \|f''\|_{\infty,[a,b]}. \end{aligned}$$

Taking the maximum over all $x \in [a,b]$ yields the claim. □

4.7.2 Cubic splines

Though beautiful from a mathematical point of view and easy to implement linear splines will not produce smooth interpolants. In general, they are not differentiable. Cubic splines allow third-order polynomials to connect the data and match the pieces in such a way that the overall interpolant is smooth.

Following [1], let's assume that a cubic polynomial

$$Y_i \colon [0,1] \to \mathbb{R}, \qquad Y_i(t) = a_i + b_i t + c_i t^2 + d_i t^3.$$

connects neighboring points (x_i, y_i) and (x_{i+1}, y_{i+1}) for $0 \leq i \leq n-1$ and some coefficients a_i, b_i, c_i and d_i that need to be determined. For simplicity we assume that $x_{i+1} - x_i = 1$ for $0 \leq i \leq n-1$.

In theory we would need to solve $4n$ equations to obtain 4 coefficients for n segments. However, the problem can be considerably simplified in terms of derivative values. Of course we want again the ith polynomial to match the endpoints, i.e.

$$y_i = Y_i(0) = a_i \qquad \text{and} \qquad y_{i+1} = Y_i(1) = a_i + b_i + c_i + d_i \tag{4.25}$$

for $0 \leq i \leq n-1$. These $2n$ equations directly imply continuity of the cubic spline at interior nodes, $Y_i(1) = y_{i+1} = Y_{i+1}(0)$. However, on each segment thus far we have only two equations for four unknowns. We will now impose additional conditions for the first and second derivatives.

We introduce the $2n$ definitions

$$D_i := Y_i'(0) = b_i \qquad \text{and} \qquad D_{i+1} := Y_i'(1) = b_i + 2c_i + 3d_i, \tag{4.26}$$

for $0 \leq i \leq n-1$. Both definitions do not contradict each other if we assume that the derivatives agree at *interior* nodes

$$Y_i'(1) = D_{i+1} = Y_{i+1}'(0) \tag{4.27}$$

for each $0 \leq i \leq n-2$. Combining (4.25) and (4.26) in terms of the (unknown) derivatives yields

$$\begin{aligned} a_i &= y_i, \\ b_i &= D_i, \\ c_i &= 3(y_{i+1} - y_i) - 2D_i - D_{i+1}, \\ d_i &= 2(y_i - y_{i+1}) + D_i + D_{i+1}. \end{aligned} \tag{4.28}$$

It remains to solve for the derivative values D_i. For interior nodes, we require now on top of matching end points (4.25) and matching first derivatives (4.27) also that the second derivatives agree, i.e.

$$Y_i''(1) = Y_{i+1}''(0) \tag{4.29}$$

for $0 \leq i \leq n-2$. This choice implies

$$2c_i + 6d_i = 2c_{i+1}.$$

When substituting (4.28) into this equation, we obtain

$$2\Big[3(y_{i+1} - y_i) - 2D_i - D_{i+1}\Big] + 6\Big[2(y_i - y_{i+1}) + D_i + D_{i+1}\Big] = 2\Big[3(y_{i+2} - y_{i+1}) - 2D_{i+1} - D_{i+2}\Big]$$

and after simplifications we obtain $n-1$ equations,

$$D_i + 4D_{i+1} + D_{i+2} = 3(y_{i+2} - y_i)$$

for $0 \leq i \leq n-2$. On each boundary segment (4.25), i.e.

$$Y_0(0) = y_0 \qquad \text{and} \qquad Y_{n-1}(1) = y_n,$$

determines one coefficient and the neighboring interior segment another two. Thus, in total we are two conditions short – one for either boundary segment. There are several different possibilities [6] now to obtain two additional equations. We will focus here on *natural cubic splines*, demanding that the spline becomes linear at the boundary nodes by requiring the second derivatives to vanish,

$$Y_0''(0) = 0 \qquad \text{and} \qquad Y_{n-1}''(1) = 0.$$

This condition implies

$$2c_0 = 0 \qquad \text{and} \qquad 2c_{n-1} + 6d_{n-1} = 0$$

or equivalently

$$2D_0 + D_1 = 3(y_1 - y_0) \qquad \text{and} \qquad D_{n-1} + 2D_n = 3(y_n - y_{n-1}).$$

Thus we need to solve the tridiagonal linear system

$$\begin{pmatrix} 2 & 1 & & & & \\ 1 & 4 & 1 & & & \\ & 1 & 4 & 1 & & \\ & & \ddots & \ddots & \ddots & \\ & & & 1 & 4 & 1 \\ & & & & 1 & 2 \end{pmatrix} \begin{pmatrix} D_0 \\ D_1 \\ D_2 \\ \vdots \\ D_{n-1} \\ D_n \end{pmatrix} = \begin{pmatrix} 3(y_1 - y_0) \\ 3(y_2 - y_0) \\ 3(y_3 - y_1) \\ \vdots \\ 3(y_n - y_{n-2}) \\ 3(y_n - y_{n-1}) \end{pmatrix}.$$

Once this system is solved one only needs to substitute the solution into (4.28). It is also worth pointing out that even though originally we needed to find $4n$ coefficients the linear system above consists only of $n+1$ equations. Furthermore, for such a *diagonally dominant* matrix where the magnitude of the diagonal entry in every row is larger than or equal to the sum of the magnitudes of all the other (non-diagonal) entries, it can be shown that it is invertible [27]. Thus there exists a unique natural cubic spline. It has lots of other beautiful properties which the interested reader may find in [6].

Figure 4.12 and Figure 4.13 show that spline interpolation on equidistant nodes works well.

Demos

1. `SplineInterpolationLinear.m`
2. `PlotRungeFunctionLinearSpline.m`
3. `PlotRungeFunctionCubicSpline.m`

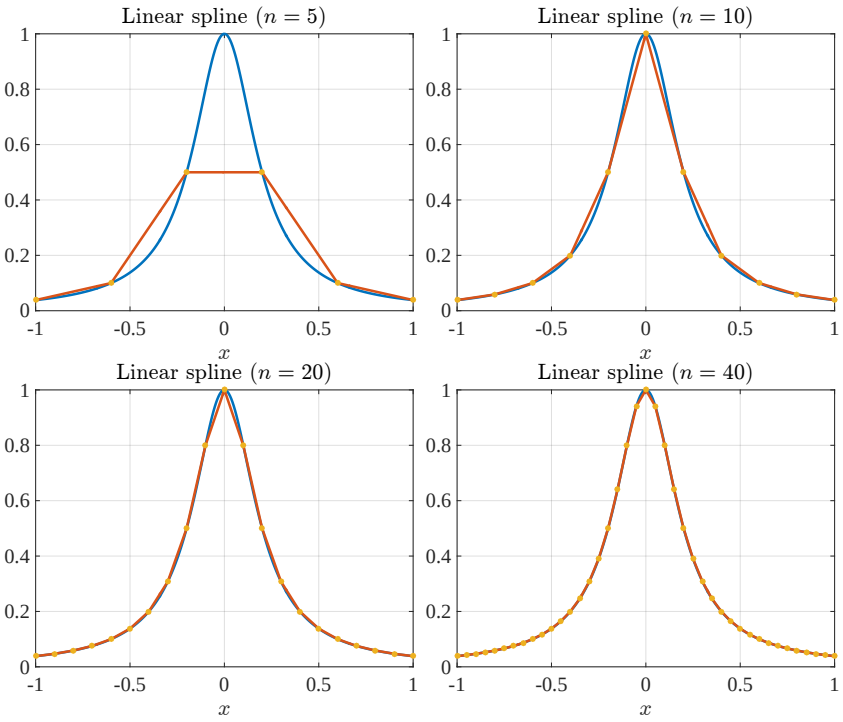


Figure 4.12: The Runge function (blue) and linear spline interpolants (red) on equidistant nodes

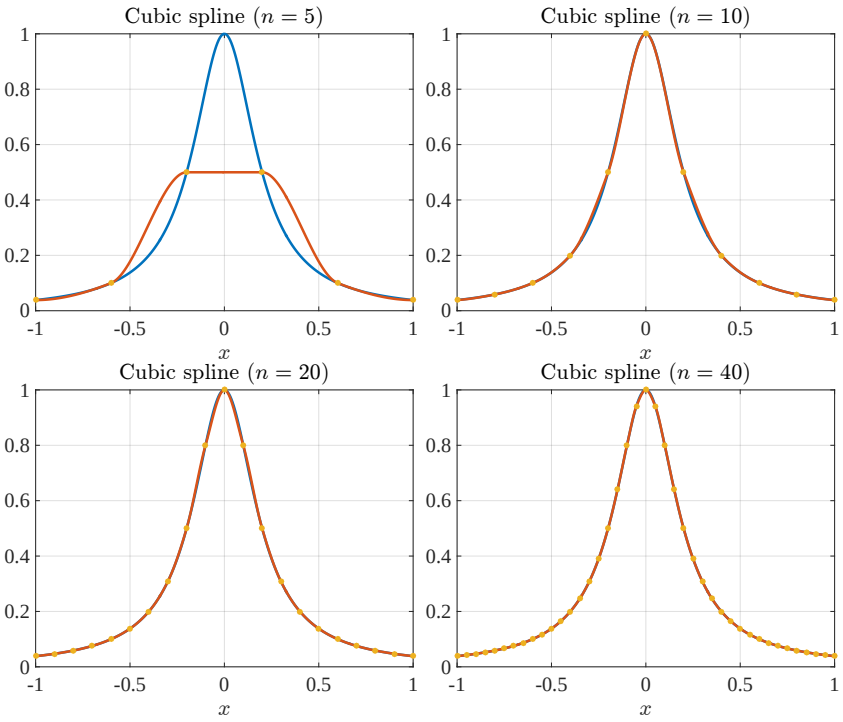


Figure 4.13: The Runge function (blue) and natural cubic spline interpolants (red) on equidistant nodes

5 Numerical integration aka quadrature

In this chapter, we discuss how to numerically approximate integrals of the form

$$\int_a^b f(x)dx \tag{5.1}$$

for some integrable function $f\colon [a,b] \to \mathbb{R}$. In the literature this process is referred to as *numerical integration* or *quadrature*. By the fundamental theorem of calculus, we know the definite integral exists and is unique. Before we jump into the discussion, we collect some important features. Similar to the interpolation operator, we can define the *integral operator* I as the mapping

$$I\colon L_1[a,b] \to \mathbb{R}, \qquad If := \int_a^b f(x)dx.$$

Theorem 10 (Properties of the integral operator)**.** *The integration operator satisfies the following properties:*

1. *The integration operator is linear.*
2. *The integration operator preserves nonnegative functions:* $0 \le f$ *implies* $0 \le If$*.*
3. *The integration operator is monotonous:* $f \le g$ *implies* $If \le Ig$*.*
4. *The absolute and relative condition numbers of* I *with respect to the* L_1 *norm,* $\|f\|_{L_1} = \int_a^b |f(x)|dx$*, are given by*

$$\kappa_a = 1 \qquad \text{and} \qquad \kappa_r = \frac{\|f\|_{L_1}}{|If|} = \frac{I|f|}{|If|} \tag{5.2}$$

Proof. The first three properties can be found in introductory analysis text books. For the final claim we have by the second property

$$|If| = \left| \int_a^b f(x)dx \right| \le \int_a^b |f(x)|\, dx = \|f\|_{L_1}. \tag{5.3}$$

Equality is obtained for any positive function on $[a,b]$. Thus, the induced operator norm (which is equal to κ_a due to the first property), we have

$$\kappa_a = \|I\| = 1.$$

Now we compute

$$\frac{|If - I\tilde{f}|}{|If|} = \frac{|\int_a^b \big(f(x) - \tilde{f}(x)\big)dx|}{|If|} \le \frac{\|f\|_{L_1}}{|If|} \frac{\|f - \tilde{f}\|_{L_1}}{\|f\|_{L_1}}$$

where we have used (5.3). □

The relative condition number (5.2) can become very large if the integral of the absolute value of the function becomes a lot larger than the integral itself. This may happen with functions that rapidly change between positive and negative values. This intrinsic ill-conditioning of integrating such highly oscillatory functions can be interpreted as the continuous analogue of cancellation when adding/subtracting two numbers of nearly equal size but opposite sign.

5.1 Midpoint, trapezoidal and Simpson's rules

Even though for let's say piecewise continuous functions the definite integral (5.1) is guaranteed to exist, this does not mean we can always compute it. Consider for example the integrand

$$f(x) = \log(\log(x)) - \log(\log(2)) \qquad \text{in} \quad [2, 5]. \tag{5.4}$$

There exists no antiderivative in terms of elementary functions. Yet the integrand in Figure 5.1 (blue) looks smooth. We can even be tempted to very roughly get

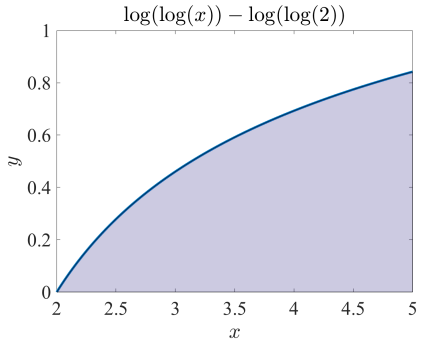


Figure 5.1: The integrand $f(x) = \log(\log(x)) - \log(\log(2))$.

an idea for how large the integral should be. If we approximate the integral with a triangle we see that its value has to be larger than $\approx 3 \cdot 0.8/2 = 1.2$. Using the `quad` function in Matlab we get

```
1.622564946017777
```

but have no idea how accurate this is.

Our goal is now to approximate the definite integral numerically in such a way that we get an idea for the error. We can achieve this by replacing the function f with its interpolation polynomial,

$$\int_a^b f(x) \approx \int_a^b p_n(x) dx,$$

thus we can use our results from Chapter 4. The right-hand side, a polynomial, is then very easy to integrate. In this section, we will assume that the nodes are uniformly distributed. Sometimes including the interval boundaries, sometimes not. Of course, we have already learned that we need to be careful when the degree of the interpolation polynomial becomes large.

5.1.1 Midpoint rule

The simplest approach would be to replace the integrand (5.1) with a constant interpolation polynomial p_0. It is not quite clear what a uniform grid with just one point is. The best choice would be to put it in the middle of the interval, resulting in the *midpoint rule*, sometimes also referred to as *rectangle method*,

$$\int_a^b f(x)dx \approx Q_0 f := \int_a^b p_0(x)dx = \int_a^b f\left(\frac{a+b}{2}\right)dx = (b-a)f\left(\frac{a+b}{2}\right).$$

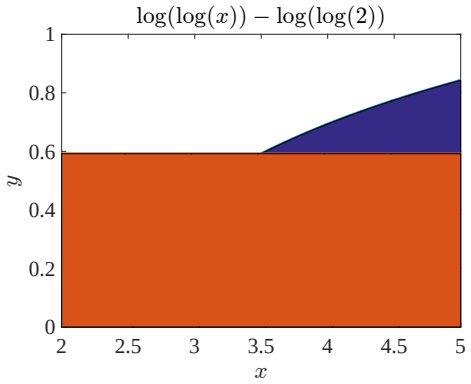


Figure 5.2: The integrand $f(x) = \log(\log(x)) - \log(\log(2))$ approximated via the midpoint rule.

The midpoint rule is interpreted graphically in Figure 5.2 and approximates the integral of (5.4) with

`1.775593222222878.`

This formula integrates by construction all constants exactly. It turns out, it also integrates linear polynomials exactly! Consider some arbitrary linear polynomial $q(x) = mx + d$ for $m, d \in \mathbb{R}$, then

$$\begin{aligned} Iq = \int_a^b q(x)dx &= \frac{1}{2}m(b^2 - a^2) + d(b-a) = (b-a)\Big(m\frac{a+b}{2} + d\Big) \\ &= (b-a)q\Big(\frac{a+b}{2}\Big) = Q_0 q. \end{aligned}$$

Graphically, this can be verified for $f(x) = 2x - 3$ in Figure 5.3.

Let us discuss the error of midpoint rule. For $f \in C^2[a,b]$ with $x_0 = \frac{a+b}{2}$, we find some $\xi = \xi(x) \in [a,b]$ such that

$$f(x) = f(x_0) + f'(x_0)(x - x_0) + \frac{1}{2}f''(\xi)(x - x_0)^2.$$

That is, we find

$$\begin{aligned} \int_a^b \Big(f(x) - p_0(x)\Big)dx &= \int_a^b \Big(f'(x_0)(x - x_0) + \frac{1}{2}f''(\xi(x))(x - x_0)^2\Big)dx \\ &= \int_{x_0}^b f''(\xi(x))(x - x_0)^2 dx \end{aligned}$$

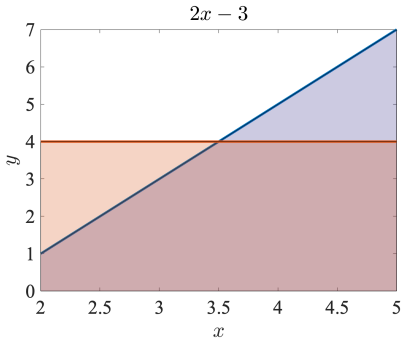


Figure 5.3: The integrand $f(x) = 2x - 3$ approximated via the midpoint rule.

where we have used that an odd function over a symmetric interval vanishes and an even function over a symmetric interval can be computed by considering only one half of the integration domain. Incidentally, this formula also shows that linear polynomials are integrated exactly. The error can then be bounded by

$$\begin{aligned}\left|\int_a^b \Big(f(x) - p_0(x)\Big)dx\right| &\leq \max_{\xi\in[a,b]} |f''(\xi)| \int_{x_0}^b (x - x_0)^2 dx \\ &= \frac{1}{3}(b - x_0)^3 \max_{\xi\in[a,b]} |f''(\xi)| \\ &= \frac{(b-a)^3}{24} \max_{\xi\in[a,b]} |f''(\xi)| = \frac{h^3}{24} \max_{\xi\in[a,b]} |f''(\xi)|\end{aligned} \tag{5.5}$$

with $h = b - a$. Of course, we could replace the constant polynomial p_0 with other function values, such as $f(a)$ or $f(b)$. These choices lead to the left point and right point rule, respectively, depicted in Figure 5.4. Whereas the former grossly underestimates the integral, the latter overestimates it. This would be true for any strictly increasing function.

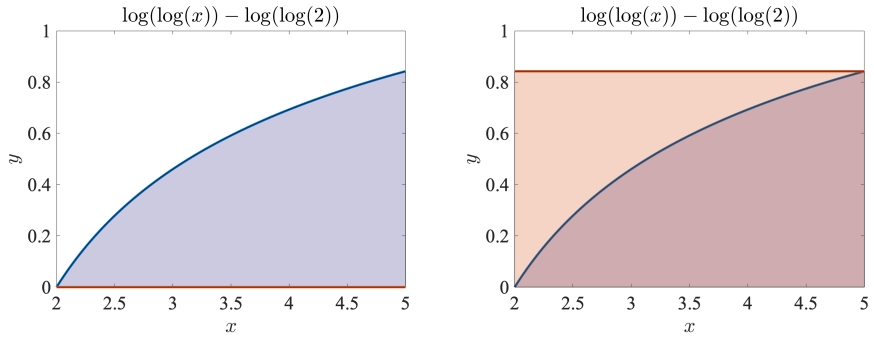


Figure 5.4: The integrand $f(x) = \log(\log(x)) - \log(\log(2))$ approximated via left point (left) and right point (right) rule.

5.1.2 Trapezoidal rule

A next slightly more sophisticated choice would be use a first-order interpolation polynomial p_1 at the points $x_0 = a$ and $x_1 = b$ as well as the corresponding function values. By (4.2), we know that

$$p_1(x) = \frac{x-a}{b-a} f(b) + \frac{x-b}{a-b} f(a).$$

This leads to the *trapezoidal rule*

$$\int_a^b f(x)dx \approx Q_1 f := \int_a^b p_1(x)dx = \frac{b-a}{2}\Big(f(a) + f(b)\Big).$$

This quadrature rule approximates the area under the curve with a trapzoid – hence the name. For $f \in C^2[a,b]$, we can derive an error estimate of the form

$$\left| \int_a^b \Big(f(x) - p_1(x)\Big)dx \right| \leq \frac{(b-a)^3}{12} \max_{\xi\in[a,b]} |f''(\xi)| = \frac{h^3}{12} \max_{\xi\in[a,b]} |f''(\xi)| \qquad (5.6)$$

with $h = b-a$, see also the next section. The above inequality means the simpler midpoint rule has an error bound (5.5) which is half of the analogous bound for the trapezoidal rule. This does *not* imply that the error for trapezoidal rule is higher – only the bound on the error is.

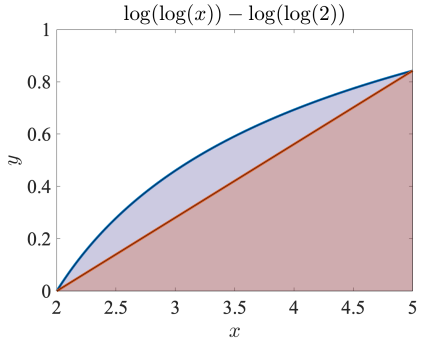


Figure 5.5: The integrand $f(x) = \log(\log(x)) - \log(\log(2))$ approximated via trapezoidal rule.

The trapezoidal rule is interpreted graphically in Figure 5.5 and approximates the integral of (5.4) with

```
1.263596873863162
```

5.1.3 Simpson's rule

We can increase the polynomial degree even more, considering the second-order interpolation polynomial p_2 at the points $x_0 = a$, $x_1 = \frac{a+b}{2}$ and $x_2 = b$ as well as the corresponding function values. It takes the form

$$p_2(x) = \frac{x-x_1}{x_0-x_1}\frac{x-x_2}{x_0-x_2} f(x_0) + \frac{x-x_0}{x_1-x_0}\frac{x-x_2}{x_1-x_2} f(x_1) + \frac{x-x_0}{x_2-x_0}\frac{x-x_1}{x_2-x_1} f(x_2).$$

Integrating this polynomial leads to the *Simpson's rule*,

$$\int_a^b f(x)dx \approx Q_2 f := \int_a^b p_2(x)dx = \frac{b-a}{6}\left(f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right).$$

Simpson's rule has a similar feature like the midpoint rule. By construction, it integrates second-order polynomials exactly. However, one can show this also extends to third-order polynomials. This is due to the following observation.

Theorem 11. *Let p_2 be the interpolation polynomial of some function $f \in C[a,b]$ with nodes $x_0 = a$, $x_1 = \frac{a+b}{2}$ and $x_2 = b$ and p_3 the interpolation polynomial to f on the same nodes as well as some pairwise distinct $z \in (a,b)$. Then*

$$\int_a^b p_3(x)dx = \int_a^b p_2(x)dx.$$

Proof. Using Newton's representation of the interpolation polynomial, we have

$$p_3(x) = p_2(x) + f[x_0, x_1, x_2, z](x - x_0)(x - x_1)(x - x_2),$$

which implies

$$\begin{aligned}\int_a^b p_3(x)dx &= \int_a^b p_2(x)dx + f[x_0, x_1, x_2, z]\int_a^b (x - x_0)(x - x_1)(x - x_2)dx \\ &= \int_a^b p_2(x)dx + f[x_0, x_1, x_2, z]\int_{-\frac{b-a}{2}}^{\frac{b-a}{2}} \left(y + \frac{b-a}{2}\right) y \left(y - \frac{b-a}{2}\right) dy.\end{aligned}$$

Due to symmetry, however, the final integral vanishes (odd function over symmetric interval). □

For $f \in C^4[a,b]$, we derive the error bound

$$\left|\int_a^b \Big(f(x) - p_2(x)\Big)dx\right| \leq \frac{(b-a)^5}{2880}\max_{\xi\in[a,b]}|f^{(4)}(\xi)| = \frac{h^5}{90}\max_{\xi\in[a,b]}|f^{(4)}(\xi)| \quad (5.7)$$

for $h = \frac{b-a}{2}$. Simpson's rule approximates the integral of (5.4) with

```
1.604927772769639.
```

Figure 5.6 (left) shows how it approximates the integral.

5.1.4 Simpson's 3/8 rule

Finally, for the interpolation polynomial p_3 on the nodes $x_0 = a$, $x_1 = \frac{2a+b}{3}$, $x_2 = \frac{a+2b}{3}$ and $x_3 = b$, we obtain *Simpson's 3/8 formula*,

$$\begin{aligned}\int_a^b f(x)dx \approx Q_3 f &:= \int_a^b p_3(x)dx \\ &= \frac{b-a}{8}\left(f(a) + 3f\left(\frac{2a+b}{3}\right) + 3f\left(\frac{a+2b}{3}\right) + f(b)\right).\end{aligned}$$

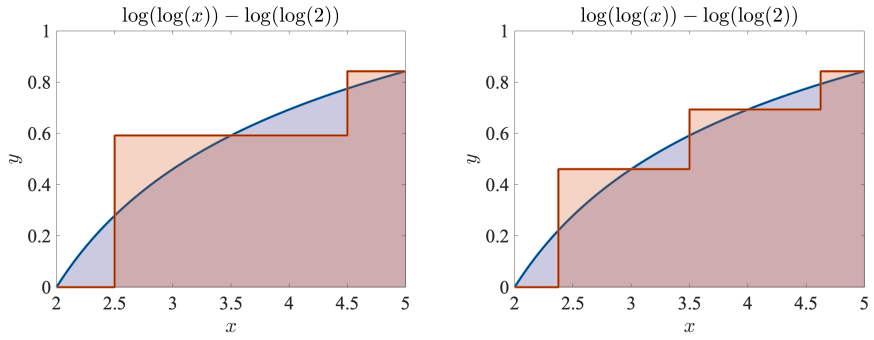


Figure 5.6: The integrand $f(x) = \log(\log(x)) - \log(\log(2))$ approximated via Simpson's rule (left) and Simpson's 3/8 rule (right).

For $f \in C^4[a,b]$, we derive the error bound

$$\left| \int_a^b \Big(f(x) - p_3(x) \Big) dx \right| \leq \frac{(b-a)^5}{2160} \max_{\xi \in [a,b]} |f^{(4)}(\xi)| = \frac{3h^5}{80} \max_{\xi \in [a,b]} |f^{(4)}(\xi)|$$

for $h = \frac{b-a}{3}$. Simpson's 3/8 rule approximates the integral of (5.4) with

```
1.613820638318888
```

Figure 5.6 (right) shows how it approximates the integral.

Demos

1. `LogLog.m`

5.2 Newton-Cotes formulas

The quadrature rules introduced in the previous section belong to the class of Newton-Cotes formulas. In general, we have equally spaced nodes

$$a \leq x_0 < x_1 < \ldots < x_n \leq b$$

and corresponding data values

$$f(x_0), \ldots, f(x_n)$$

sampled from the integrand in (5.1). We then find the interpolation polynomial

$$p_n(x) = \sum_{j=0}^{n} f(x_j) \ell_j(x)$$

through the set of nodes and data values. Here the ℓ_j denote the Lagrange basis polynomials (4.5). In this setting, the general *Newton-Cotes formulas*, i.e. numerical approximations to (5.1), are given by

$$\int_a^b p_n(x) dx = (b-a) \sum_{j=0}^{n} f(x_j) \frac{1}{(b-a)} \int_a^b \ell_j(x) dx = (b-a) \sum_{j=0}^{n} f(x_j) w_j, \quad (5.8)$$

with the *Newton-Cotes quadrature weights*

$$w_j = \frac{1}{(b-a)} \int_a^b \ell_j(x)dx \tag{5.9}$$

for $0 \leq j \leq n$.

If the set of nodes includes the endpoints a and b, the Newton-Cotes formula are called *closed*. If not, they are called *open*. With this terminology, the midpoint rule is the simplest example of an open Newton-Cotes formula. The trapezoidal, Simpson's and Simpson's 3/8 rule correspond to the closed Newton-Cotes formulas for $n = 1, 2, 3$, respectively. Since the grid is assumed to be uniform, the points are given by

$$x_i = ih + x_0 \tag{5.10}$$

with $h = \frac{b-a}{n}$ and $0 \leq i \leq n$ for closed Newton-Cotes formulas and $h = \frac{b-a}{n+2}$ and $1 \leq i \leq n-1$ for open Newton-Cotes formulas.

Besides, the midpoint rule, open Newton-Cotes formulas are of less interest in practice. Already for $n > 1$, they may have positive and negative weights, increasing the likelihood of cancellation. Closed Newton-Cotes formulas may have negative weights for $n > 7$. For this reason we focus on closed Newton-Cotes formulas.

5.2.1 Errors

We will discuss the general error between the definite integral (5.1) and the closed Newton-Cotes formulas.

Theorem 12 (Error for closed Newton-Cotes formulas)**.** *Let $f \in C^{n+1}[a,b]$, $n \in \mathbb{N}_0$ and p_n be the interpolation polynomial through $(x_0, f(x_0)), \ldots, (x_n, f(x_n))$ where the quadrature nodes are given by (5.10) for $h = \frac{b-a}{n}$ and $0 \leq i \leq n$. Then, there exists some $\xi = \xi(x) \in [a,b]$ such that*

$$\int_a^b \Big(f(x) - p_n(x)\Big)dx = \frac{1}{(n+1)!} \int_a^b f^{(n+1)}(\xi(x)) \prod_{i=0}^{n}(x - x_i)dx.$$

Proof. This follows directly from Theorem 6. □

This theorem can be considerably refined.

Theorem 13. *Under the same assumptions of Theorem 12, there exists for odd $n \geq 3$ some $\mu \in [a,b]$ such that*

$$\int_a^b \Big(f(x) - p_n(x)\Big)dx = \frac{1}{(n+1)!} f^{(n+1)}(\mu) \int_a^b \prod_{i=0}^{n}(x - x_i)dx.$$

If $f \in C^{n+2}[a,b]$, then there exists for even $n \geq 2$ some $\mu \in [a,b]$ such that

$$\int_a^b \Big(f(x) - p_n(x)\Big)dx = \frac{1}{(n+2)!} f^{(n+2)}(\mu) \int_a^b x \prod_{i=0}^{n}(x - x_i)dx.$$

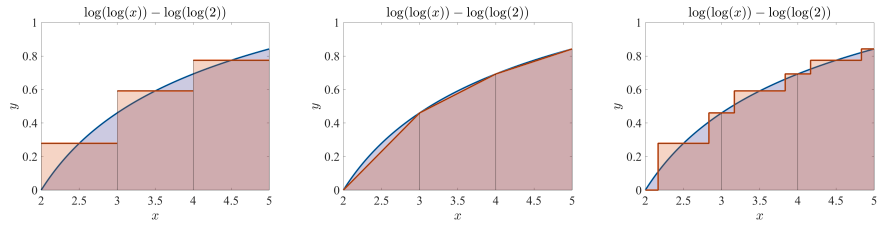


Figure 5.7: Composite rules for $m = 3$: midpoint, trapezoidal and Simpson's rule (from left to right).

Proof. A proof of this can be found in [21, Section 6.2]. □

The error estimates in the previous section follow exactly from this theorem. Additionally, it shows that the special property of Simpson's rule exactly integrating third-order polynomials – even though it was only constructed to be exact for second-order polynomials ($n = 2$) – is true for any closed Newton Cotes formula of even degree.

5.3 Composite Newton-Cotes formulas

We already mentioned that increasing the polynomial degree (on equidistant) nodes may lead to cancellation. Additionally, one can show that not for every f

$$\lim_{n\to\infty} \left| \int_a^b \Big(f(x) - p_n(x) \Big) dx \right| = 0.$$

For this reason, we study *composite formulas*. The idea is to chop the integral $[a, b]$ into m smaller intervals of equal length $H = \frac{b-a}{m}$ and use a low-degree Newton-Cotes formula on each subdivision. In particular, we discuss the composite midpoint, trapezoidal and Simpson's rule. Figure 5.7 visualizes three different composite rules.

The *composite midpoint rule* is given by

$$M_H f = H \sum_{k=0}^{m-1} f(x_k)$$

with $x_k = a + \frac{1}{2}(2k+1)H$ for $0 \le k \le m-1$. The red dots in the following picture show where the integrand needs to be evaluated. The gray lines indicate the subinterval boundaries.

The *composite trapezoidal rule* is given by

$$T_H f = H \left(\frac{1}{2} f(x_0) + \sum_{k=1}^{m-1} f(x_k) + \frac{1}{2} f(x_m) \right)$$

with $x_k = a + kH$ for $0 \le k \le m$. This time the integrand needs to be evaluated at the boundaries of the sub intervals.

The *composite Simpson's rule* is given by

$$S_H f = \frac{H}{6}\left(f(x_0) + 2\sum_{k=1}^{m-1} f(x_{2k}) + 4\sum_{k=0}^{m-1} f(x_{2k+1}) + f(x_{2m})\right)$$

with $x_k = a + \frac{1}{2}kH$ for $0 \le k \le 2m$. Again, we show where the integrand needs to be evaluated.

The errors for the composite formulas follow directly from the error estimates (5.5), (5.6) and (5.7). For $H = \frac{b-a}{m}$ one obtains

$$\left|\int_a^b f(x)dx - M_H f\right| \le \frac{b-a}{24}H^2 \max_{\xi\in[a,b]} |f''(\xi)|,$$
$$\left|\int_a^b f(x)dx - T_H f\right| \le \frac{b-a}{12}H^2 \max_{\xi\in[a,b]} |f''(\xi)|,$$
$$\left|\int_a^b f(x)dx - S_H f\right| \le \frac{b-a}{2880}H^4 \max_{\xi\in[a,b]} |f^{(4)}(\xi)|.$$

That is the global error from a composite quadrature rule is of order lower in terms of H than the local ones in terms of $b-a$. We demonstrate the error estimate for Simpson's rule – the other ones follow in the same fashion. Denote with $Q_2^{[a,b]}$ the (local) Simpson's quadrature approximation to (5.1) and with $p_2^{[a,b]}$ the corresponding interpolation polynomial. We compute

$$\begin{aligned}\left|\int_a^b f(x)dx - S_H f\right| &= \left|\sum_{i=0}^{m-1}\int_{x_{2i}}^{x_{2i+2}} f(x)dx - \frac{H}{6}\Big\{f(x_{2i} + 4f(x_{2i+1}) + f(x_{2i+2})\Big\}\right| \\ &= \left|\sum_{i=0}^{m-1}\int_{x_{2i}}^{x_{2i+2}} f(x)dx - Q_2^{[x_{2i},x_{2i+2}]} f\right|.\end{aligned}$$

Using the definition of Simpson's rule, the triangle inequality and (5.7), we find

$$\begin{aligned}\left|\int_a^b f(x)dx - S_H f\right| &= \left|\sum_{i=0}^{m-1}\int_{x_{2i}}^{x_{2i+2}} \Big(f(x) - p_2^{[x_{2i},x_{2i+2}]}(x)\Big)dx\right| \\ &\le \sum_{i=0}^{m-1}\int_{x_{2i}}^{x_{2i+2}} \left|f(x) - p_2^{[x_{2i},x_{2i+2}]}(x)\right| dx \\ &\le \sum_{i=0}^{m-1} \frac{(x_{2i+2} - x_{2i})^5}{2880} \max_{\xi\in[a,b]} |f^{(4)}(\xi)| \\ &= \frac{H^4}{2880} \max_{\xi\in[a,b]} |f^{(4)}(\xi)| \sum_{i=0}^{m-1} x_{2i+2} - x_{2i},\end{aligned}$$

which yields the desired result since the final sum is a telescoping series.

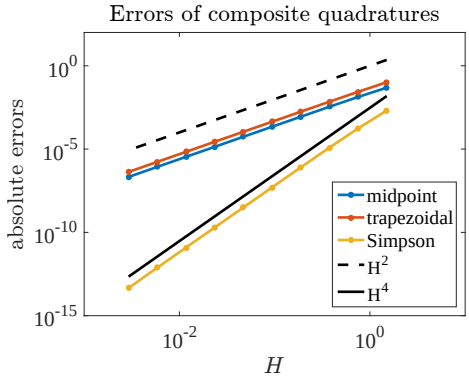


Figure 5.8: Comparison of convergence orders for different quadrature rules for the integrand $f(x) = \log(\log(x)) - \log(\log(2))$. The errors are computed with respect to a fine reference solution.

Demos

1. `PlotNodeDistributionCompositeRules.m`
2. `CompositeMidpoint.m`
3. `CompositeTrapezoidal.m`
4. `CompositeSimpsons.m`
5. `PlotQuadratureErrors.m`

5.4 Adaptive formulas

It is advisable to use finer subdivisons only when necessary. This motivates an adaptive approach, refining only in certain problematic regions. Matlab's `quad` command implements such an adaptive approach for Simpson's rule.

6 Nonlinear problems

A linear world would be pretty boring. Fortunately, most of the problems appearing in nature are *nonlinear*, yielding more interesting but also more complicated solutions. Obviously, this calls for special solution techniques which we would like to study in this chapter. We focus mostly on nonlinear root finding or fixed point problems. However, some of the techniques (in particular Newton's method) can be applied to more general nonlinear problems.

We consider the following problem. For some function $F\colon D \subseteq \mathbb{R}^d \to \mathbb{R}^d$ find a point x^* such that

$$F(x^*) = 0. \tag{6.1}$$

Obviously this *root finding problem* can equivalently be written as a *fixed point problem*: Find a point x^* such that

$$F(x^*) = x^*. \tag{6.2}$$

Also (nonlinear) systems of equations can be transformed into a rootfinding problem. Before we look at numerical solution techniques, we discuss when we can expect to find fixed points.

6.1 Fixed point theorems

A major tool to show the existence of fixed points are so called *fixed point theorems*. We will focus here on fixed point theorems due to the mathematicians Stefan Banach and Luitzen Brouwer. Both can be stated in more generality. The following theorem is given in terms of metric spaces. Note that any normed vector space is also a metric space.

Theorem 14 (Banach fixed point theorem)**.** *Let (X, d) be a complete metric space. If a function $F\colon X \to X$ is a contraction that is there exists a constant $0 \leq L < 1$ such that*

$$d(F(x), F(y)) \leq L d(x, y) \tag{6.3}$$

then there exists a unique fixed point x^*, satisfying

$$F(x^*) = x^*.$$

Moreover, the fixed point can be computed as follows: Start with an arbitrary point $x_0 \in X$ and set

$$x_{n+1} = F(x_n) \tag{6.4}$$

for $n \in \mathbb{N}$, then x_n tends to x^* as n tends to infinity.

Proof. The proof is along the lines of [23]. For $x, y \in X$ the triangle inequality and the contraction property (6.3) yield

$$\begin{aligned} d(x,y) &\leq d(x,F(x)) + d(F(x),F(y)) + d(F(y),y) \\ &\leq d(x,F(x)) + Ld(x,y) + d(F(y),y) \end{aligned}$$

which yields after rearrangement the contraction inequality

$$d(x,y) \leq \frac{d(F(x),x) + d(F(y),y)}{1-L}. \tag{6.5}$$

Not the denominator cannot become zero as $L < 1$. This inequality implies that if x and y are two fixed points then the right-hand side vanishes. Hence $x = y$, establishing uniqueness of the fixed point. To prove existence of a fixed point, we will show that the sequence of points defined via (6.4) converges to *a* point which will turn out to be *the* fixed point.

Convergence follows from the fact that the sequence (6.4) for any starting value x_0 is Cauchy. To see this, we write via (6.4)

$$F^n(x_0) = F^{n-1}(x_1) \ldots = F(x_{n-1}) = x_n.$$

Inductively, by the contraction property (6.3), we find

$$d(F^n(x), F^n(y)) \leq L^n d(x,y).$$

Now replacing x and y by $F^n(x_0)$ and $F^m(x_0)$ in the contraction inequality (6.5) gives

$$\begin{aligned} d(F^n(x_0), F^m(x_0)) &\leq \frac{d(F(F^n(x_0)), F^n(x_0)) + d(F(F^m(x_0)), F^m(x_0))}{1-L} \\ &= \frac{d(F^n(F(x_0)), F^n(x_0)) + d(F^m(F(x_0)), F^m(x_0))}{1-L} \\ &\leq \frac{L^n d(F(x_0), x_0) + L^m d(F(x_0), x_0)}{1-L} \\ &= \frac{L^n + L^m}{1-L} d(F(x_0), x_0). \end{aligned} \tag{6.6}$$

Since $L < 1$, the right-hand side vanishes in the limit for n, m tending to infinity, proving that the sequence $\{x_n = F^n(x_0)\}$ is Cauchy.

A Cauchy sequence in a complete space converges to a point in itself, say $x_c := \lim_{n\to\infty} x_n \in X$. It remains to show that $x_c = x^*$. However, this is immediately clear from

$$d(F(x_c), x_{n+1}) = d(F(x_c), F(x_n)) \leq Ld(x_c, x_n),$$

implying after taking the limit the first equality (the second holds by construction) in

$$F(x_c) = \lim_{n\to\infty} x_n = x_c.$$

Thus x_c must be the unique fixed point $x_c = x^*$.

□

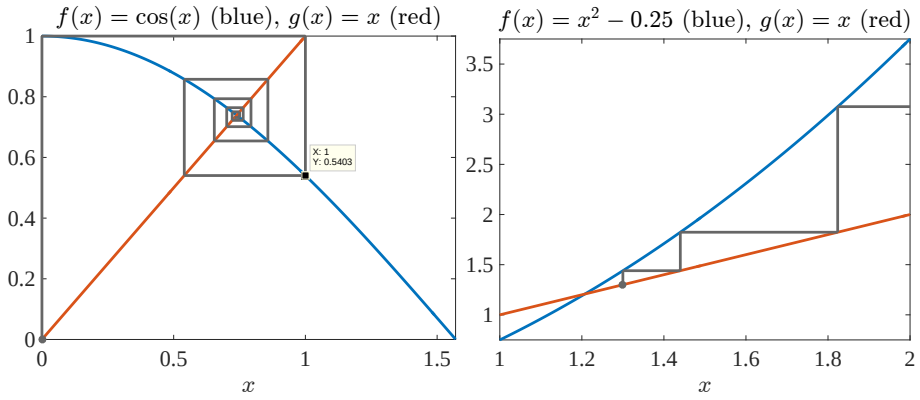


Figure 6.1: Example for a convergent and divergent fixed point iteration

We point out that the a-priori speed of convergence can be established from (6.6) by letting $m \to \infty$ we deduce

$$d(x_n, x^*) \leq \frac{L^n}{1-L} d(x_1, x_0).$$

So if L is close to one, the convergence might be relatively slow.

The contraction constant L is a special type of Lipschitz constant. For a differentiable function $f\colon \mathbb{R} \to \mathbb{R}$ it can be thought of as the largest absolute value of its derivative. Figure 6.1 shows an example when the fixed point iteration converges – and one where it does not.

If M is a closed subset of the complete metric space (X, d) then (M, d) is also complete. The second fixed point theorem is surprisingly simple to state, intuitively clear in 1D but not that easy to show in its full generality. We refer to [3].

Theorem 15 (Brouwer fixed point theorem)**.** *Let V be a nonempty compact convex subset of $\mathbb{R}^d$ and $F\colon V \to V$ a continuous function. Then F has at least one fixed point.*

Proof. [12, Corollary 1.1.1]. □

Finally, we qualify the speed of the convergence to either a fixed point or a root. Suppose a sequence of iterates $x_0, x_1, x_2, \ldots$ converges to some x^*. We introduce some nomenclature for the rate or speed at which this may happen. We say such a sequence of iterates *converges with order* $p \geq 1$ to the value x^* if

$$\|x_{n+1} - x^*\| \leq C \|x_n - x^*\|^p$$

for some $C > 0$, $n \in \mathbb{N}$ and some norm $\|\cdot\|$. If $p = 1$, then we demand that $C < 1$ in order to allow more interesting (converging) sequences than just the trivial one $x_n = x^*$ for all $n \in \mathbb{N}$. Such a behavior is called *linear* convergence. If $p = 2$, then we say the convergence is *quadratic*.

6.2 One dimensional techniques

In this section we introduce a couple of techniques to solve one-dimensional root-finding problems of the form (6.1) for some continuous function $f\colon [a, b] \to \mathbb{R}$.

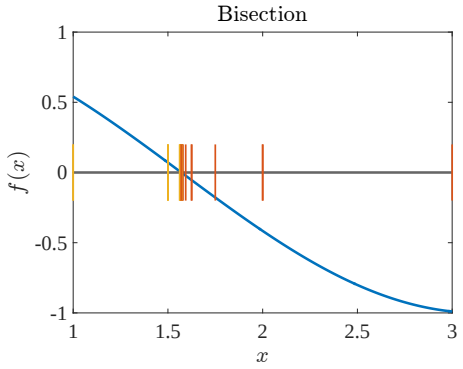


Figure 6.2: Bisection for $f(x) = \cos(x)$ in the interval $[1, 4]$. The red lines indicate how the right boundary is shifted while the yellow boundary indicates how the left boundary is shifted.

If $f(a)$ and $f(b)$ have different signs, then we know by the intermediate value theorem that there exists at least one zero in $[a, b]$.

We judge the quality of the different schemes with respect to the following criteria:

- the smoothness of the function (do we need more than continuity?),
- guaranteed convergence (do we always find a zero?),
- the order of convergence (how fast do we find the zero?).

6.2.1 Bisection

One of the most basic solution techniques is the *bisection*. We evaluate the function in the middle of the current interval and adjust the boundaries of the interval in such a way that we can guarantee that the zero lies in the new interval.

```
1 while |f(midpoint)| ≥ tol do
2 |   midpoint = (right+left)/2;
3 |   if sign(f(left)f(midpoint)) = −1 then
4 |   |   right = midpoint;
5 |   else
6 |   |   left = midpoint;
7 |   end
8 end
```

Algorithm 1: Bisection

This algorithm is explained visually in Figure 6.2. By construction the bisection method is guaranteed to (globally) converge if the target function f changes its sign at the initial interval boundaries. It suffices if f is continuous. Technically, the bisection method does not yield a sequence of values which decrease with some order p since the error does not decrease monotonically as can be seen in Figure 6.3, where we look for the root $\pi/2$ of $f(x) = \cos(x)$ in the interval $[1, 3]$. However, the error is bounded by

midpoint
$x_2 = 2.000000000000000$
$x_3 = 1.500000000000000$
$x_4 = 1.750000000000000$
$x_5 = 1.625000000000000$
$x_6 = 1.562500000000000$
$x_7 = 1.593750000000000$
$x_8 = 1.578125000000000$
$x_9 = 1.570312500000000$
$x_{10} = 1.574218750000000$
$x_{11} = 1.572265625000000$
$x_{12} = 1.571289062500000$
$x_{13} = 1.570800781250000$
$x_{14} = 1.570556640625000$
$x_{15} = 1.570678710937500$
$x_{16} = 1.570739746093750$
$x_{17} = 1.570770263671875$
$x_{18} = 1.570785522460938$
$x_{19} = 1.570793151855469$
$x_{20} = 1.570796966552734$

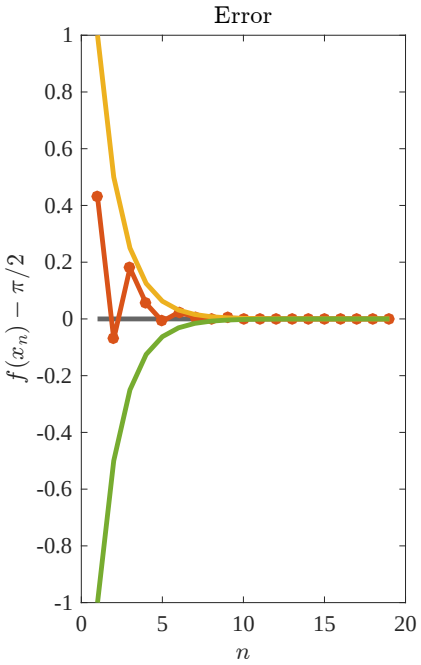


Table 6.1 & Figure 6.3: Iterates (left) and errors (right) for bisection method applied to $f(x) = \cos(x)$ in the interval $[x_0, x_1] = [1, 3]$. The yellow and green line show the guaranteed error bounds from above and below.

$$|x_n - x^*| \leq \frac{1}{2^n}(b - a)$$

for $n \in \mathbb{N}$. So we can think of the method as linearly convergent with rate $C = \frac{1}{2}$ which is compared to other methods very slow.

6.2.2 Newton's method

A very popular method to compute roots is *Newton's method*. To be able to apply it, we need the function f to be continuously differentiable. It works by finding the roots of a sequence of linearizations. Suppose we have found an estimate for a root x_n, then we construct linearize our (typically) nonlinear function via the tangent line

$$y(x) = f'(x_n)(x - x_n) + f(x_n), \tag{6.7}$$

i. e. the best linear approximation at $f(x_n)$. Newton's method defines the new iterate x_{n+1} as the root of the tangent line

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{6.8}$$

for $n \in \mathbb{N}_0$. Its popularity is mostly due to two reasons: It can be generalized to higher dimensions as we will see later and "near" the root it converges

quadratically. In order to get started we need to choose a starting guess x_0. This choice is crucial to obtain quadratic convergence as the following theorem shows.

Theorem 16 (Quadratic convergence of Newton's method)**.** *Let $f\colon \mathbb{R} \to \mathbb{R}$ be twice continuously differentiable. Denote with x^* the only root in the Interval $I = \left[x^* - |x^* - x_0|, x^* + |x^* - x_0|\right]$, where x_0 is the starting guess for Newton's method. Furthermore let*

1. *$f'(x) \neq 0$ for $x \in I$,*
2. *$|x^* - x_0| < \dfrac{1}{K}$ with $K := \dfrac{1}{2}\sup_{x\in I}\left|\dfrac{f''(x)}{f'(x)}\right|$.*

Then Newton's method (6.8) converges quadratically.

Proof. Taylor's theorem implies that there exists some ξ between x^* and $x \in I$ such that

$$0 = f(x^*) = f(x) + f'(x)(x^* - x) + \frac{1}{2}f''(\xi)(x^* - x)^2$$

or rearranging and dividing by $f'(x) \neq 0$

$$x - \frac{f(x)}{f'(x)} - x^* = \frac{1}{2}\frac{f''(\xi)}{f'(x)}(x - x^*)^2.$$

Now we can estimate

$$\left|x - \frac{f(x)}{f'(x)} - x^*\right| \leq K|x - x^*|^2$$

for all $x \in I$. Replacing x with x_n for $n \in \mathbb{N}_0$ and using (6.8), we find

$$|x_{n+1} - x^*| \leq K|x_n - x^*|^2$$

thus establishing the second order convergence *if* the series converges. Only then we could guarantee that $x_n \in I$ since the starting guess x_0 is in the same interval. However, the convergence follows from the (inductive) observation that

$$K|x_n - x^*| \leq (K|x_{n-1} - x^*|)^2 \leq \ldots \leq (K|x_0 - x^*|)^{2^n}.$$

Our assumption implies $K|x_0 - x^*| < 1$. In other words $x_n - x^*$ is a zero sequence and x_n converges to x^*. □

We point out that weaker assumptions on f are possible to show quadratic convergence. As can already be seen in the proof the continuity of f'' is only needed locally. In fact it even suffices if f' only satisfies some Lipschitz condition [8]. For us, however, the more important take-home message is that Newton's method converges locally quadratically.

Despite its popularity there are several cases where Newton's method can fail:

- Newton's method may enter a cycle. There can be at least to reasons: Either because the derivative at the root does not exist as for any starting guess x_0 and the function

$$f(x) = \operatorname{sign}(x)\sqrt{|x|}$$

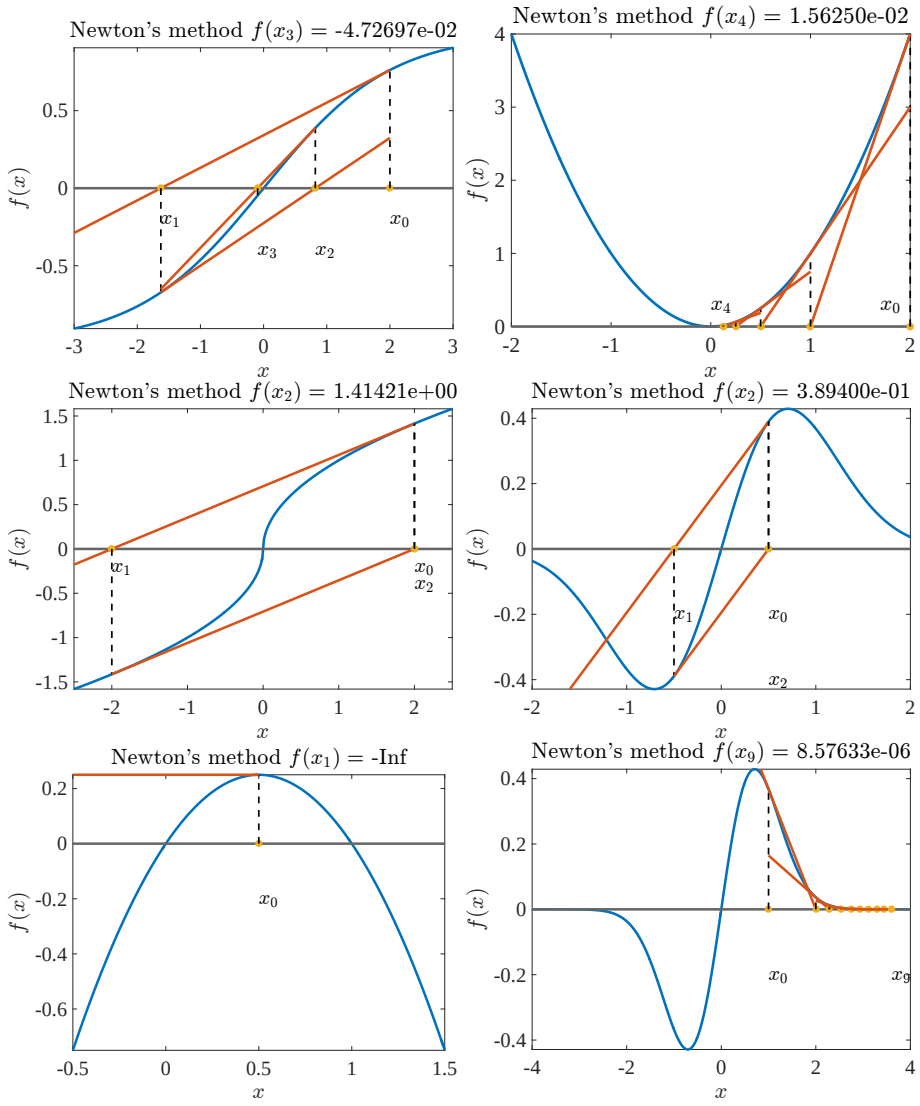


Figure 6.4: Newton's method for several different functions. The top row shows convergence for $f(x) = \tanh(x)$ and $f(x) = x^2$ with starting value $x_0 = 2$. The middle row shows that for certain combinations of functions and starting values (here $f(x) = \text{sign}(x)\sqrt{|x|}$) with $x_0 = 2$ and $f(x) = xe^{-x^2}$ with $x_0 = 0.5$). In the last row two other problems are shown: If the derivative is zero at a certain point, there is no intersection with the x axis. For starting values values larger than $\frac{1}{\sqrt{2}}$, the function $f(x) = xe^{-x^2}$ is an example where Newton's method does not converge.

or because we are not close enough to the root as for $x_0 = 0.5$ and the function

$$f(x) = xe^{-x^2}.$$

- Newton's method may diverge. This can happen if the derivative does not exist at the root. For example for $f(x) = x^{1/3}$, Newton's method diverges from the root at $x = 0$:

$$x_{n+1} = x_n - \frac{x_n^{1/3}}{1/3\, x_n^{-2/3}} = -2x_n.$$

  It may also diverge if we start too far from the root. Consider for example $x_0 > 1/\sqrt{2}$ and

$$f(x) = xe^{-x^2}.$$

- Newton's method may be slower than quadratic. For example if the derivative is zero at the root. For $f(x) = x^2$, Newton's method converges but only linearly since

$$x_{n+1} = \frac{x_n}{2}.$$

  Something similar can happen if there is no second derivative at the root, e.g. $f(x) = x + x^{4/3}$.

- Finally, we might also choose a bad starting guess where the tangent does not intersect the x axis. A very simple example would be $f(x) = x(1-x)$ and $x_0 = 0.5$.

These problematic cases are shown in Figure 6.4.

```
while |f(x_n)| ≥ tol_1 do
    if |f'(x_n)| ≤ tol_2 then
        break;
    else if n ≥ N then
        break;
    x_{n+1} = x_n − f(x_n)/f'(x_n);
end
```

Algorithm 2: Newton's method including two safeguard criteria

6.2.3 Damped Newton's method

If the standard Newton's method runs into difficulties, one may decrease the correction by some factor $t_{k(n)} \in (0, 1]$ such that the new iterate is given by

$$x_{n+1} = x_n - t_{k(n)} \frac{f(x_n)}{f'(x_n)}.$$

The value $t_{k(n)}$ is chosen for each n from a zero sequence $1 = t_0, t_1, \ldots, t_{i_{\max}}$ such that

$$\left| f\left(x_n - t_{k(n)} \frac{f(x_n)}{f'(x_n)} \right) \right|$$

is minimal among all numbers in the sequence. This technique is called *line search*. The hope is that then

$$\left| f\left(x_n - t_{k(n)} \frac{f(x_n)}{f'(x_n)} \right) \right| < |f(x_n)|,$$

even if that is not the case for the standard Newton's method where $t_{k(n)} = 1$. This version of Newton's method is often referred to as *damped Newton's method* [7, 9]. By adjusting the correction, one can often obtain convergence when the standard Newton's method (6.8) fails. Of course this comes at a cost: The convergence is usually not quadratic anymore. In practice one often considers for $i = 0, 1, 2, \ldots, i_{\max}$ either of the two sequences

- $t_i = \frac{1}{2^i}$,
- $t_i = \frac{1}{2^{\frac{i(i+1)}{2}}}$

and chooses e. g. $i_{\max} = 5$.

while $|f(x_n)| \geq \mathrm{tol}_1$ **do**
  if $|f'(x_n)| \leq \mathrm{tol}_2$ **then**
    break;
  else if $n \geq N$ **then**
    break;
  $k(n) = \operatorname{argmin} \left\{ 0 \leq i \leq i_{\max} : \left| f\left(x_n - t_i \frac{f(x_n)}{f'(x_n)} \right) \right| \right\}$;
  $x_{n+1} = x_n - t_{k(n)} \frac{f(x_n)}{f'(x_n)}$;
end

Algorithm 3: Damped Newton's method for some zero sequence $1 = t_0, \ldots, t_{i_{\max}}$.

6.2.4 Secant method

The *secant method* can be interpreted as an approximation of Newton's method, where the tangent line (6.7) is replaced with a secant line

$$y(x) = \frac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}}(x - x_n) + f(x_n),$$

or the derivative with a finite difference approximation. The new iterate is the zero of secant line

$$x_{n+1} = x_n - f(x_n) \frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})} = \frac{x_{n-1} f(x_n) - x_n f(x_{n-1})}{f(x_n) - f(x_{n-1})}. \tag{6.9}$$

There are a couple of notable differences between Newton's method and the secant method: We do not need to know the derivative f' but one needs two iterates x_0, x_1 to start the method. Similarly to Newton's method the root may not be in the interval confined by two successive iterates and the method is not guaranteed to converge if both iterates are "too far" from the zero as can be seen in Figure 6.5.

while $|f(x_n)| \geq \text{tol}$ **do**
  if $n \geq N$ **then**
    break;
  $x_{n+1} = x_n - f(x_n)\dfrac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})}$;
end

Algorithm 4: Secant method

However, if it converges, it converges superlinearly under appropriate conditions.

Theorem 17. *Suppose $f \in C^2[a,b]$ and the secant method converges to a simple root, i. e. $f'(x^*) \neq 0$. If also $f''(x^*) \neq 0$ then the sequence produced by (6.9) converges with order $p = \frac{1+\sqrt{5}}{2} \approx 1.618$.*

6.2.5 Regula falsi or false position method

The *regula falsi* or *false position method* overcomes some of the shortcomings of the secant method by combining it with bisection, see Algorithm 5.

The main conceptual difference to the secant method is the following: The regula falsi method updates *both* interval boundaries by checking that the root always lies within two successive iterates (Figure 6.6) – unlike the secant method which only updates the zeros of the secants. Since the interval boundaries converge to each other, the regula falsi converges to the (simple) root.

Demos

- `Bisection.m`
- `NewtonsMethod.m`
- `DampedNewtonsMethod.m`
- `SecantMethod.m`
- `RegulaFalsi.m`

6.3 Several dimensional problems

We can generalize some of the previous ideas to multidimensional root finding problems

$$F(x^*) = 0$$

for some function $F\colon D \subseteq \mathbb{R}^d \to \mathbb{R}^d$.

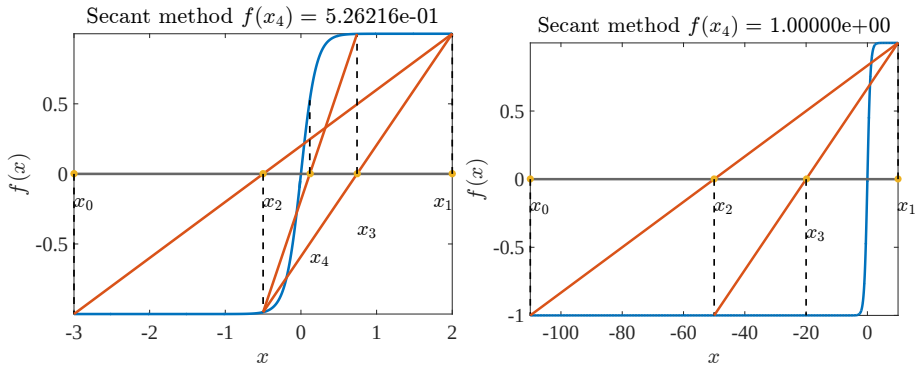


Figure 6.5: Secant method for $f(x) = \tanh(x)$ in the interval $[-3, 2]$ on the left and for the same function in the interval $[-110, 10]$ on the right. The secant method converges in the former setting and does not converge for the latter (in floating point arithmetic $x_4 =$ `inf`).

while $|f(\text{newpoint})| \geq \text{tol}$ **do**
  if $n \geq N$ **then**
    break;
  newpoint = left $- f(\text{left}) \dfrac{\text{left} - \text{right}}{f(\text{left}) - f(\text{right})}$;
  if $\text{sign}\big(f(\text{left}) f(\text{newpoint})\big) = -1$ **then**
    right = newpoint;
  else
    left = newpoint;
end

Algorithm 5: Regula falsi method

6.3.1 Multidimensional Newton's method

Newton's method can be generalized with the help of the Jacobian $DF(x)$. Then in several dimensions (6.8) becomes

$$x_{n+1} = x_n - \big(DF(x_n)\big)^{-1} F(x_n).$$

It is possible to avoid the costly computation of the inverse of the Jacobian with the following two-step process:

1. Solve the (linear) system

   $$DF(x_n)\Delta x_n = -F(x_n)$$

   for the Newton update $\Delta x_n = x_{n+1} - x_n$.

2. Find the new iterate by adding the previous one to the update

   $$x_{n+1} = \Delta x_n + x_n.$$

Again it is possible to devise a damped version

$$x_{n+1} = x_n - t_{k(n)} \big(DF(x_n)\big)^{-1} F(x_n)$$

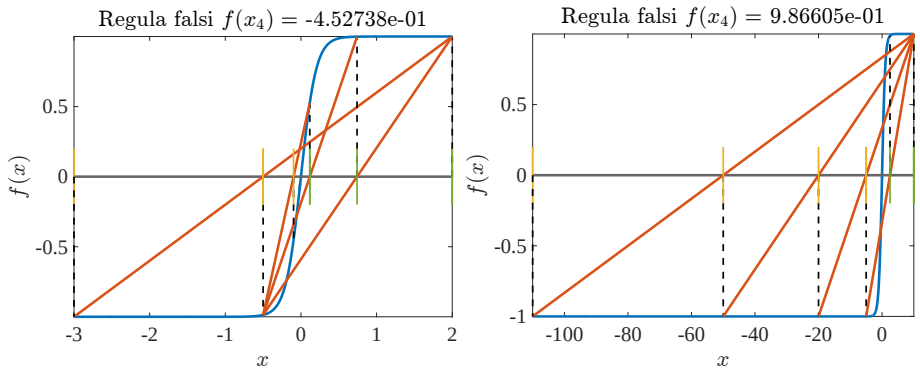


Figure 6.6: Regula falsi converges where the secant method does not. As before $f(x) = \tanh(x)$ in the interval $[-3, 2]$ on the left and for the same function in the interval $[-110, 10]$ on the right.

where $t_{k(n)} \in (0, 1]$ is chosen for each n via line search from the first couple of numbers of some zero sequence such that

$$\left\| F\left(x_n - t_{k(n)} \big(DF(x_n)\big)^{-1} F(x_n)\right)\right\|$$

is smallest.

while $\|F(x_n)\| \geq \text{tol}$ **do**
  if $n \geq N$ **then**
    break;
  $k(n) =$
    $\operatorname{argmin}\left\{0 \leq i \leq i_{\max} \colon \left\| F\left(x_n - t_i \big(DF(x_n)\big)^{-1} F(x_n)\right)\right\|\right\}$;
  $x_{n+1} = x_n - t_{k(n)} \big(DF(x_n)\big)^{-1} F(x_n)$;
end

Algorithm 6: Damped Newton's method for some zero sequence $1 = t_0, \ldots, t_{i_{\max}}$.

6.3.2 Gradient descent methods

It is also possible to employ so called *gradient descent methods* which are often used in the context of finding the optimal values of a function. Since the gradient of a function points in the direction of the steepest *ascent*, we can find a *minimum* of some function $G \colon D \subseteq \mathbb{R}^d \to \mathbb{R}$ by looking at the iterates

$$x_{n+1} = x_n - \gamma_n \nabla G(x_n) \tag{6.10}$$

for $n \in \mathbb{N}$ and some starting guess x_0. There are different ways to obtain the parameters γ_n, most commonly via some line search. In order to use this minimization strategy to solve a (nonlinear) system, we introduce the functions

$$h \colon \mathbb{R}^d \to \mathbb{R}, \qquad h(x) = \frac{1}{2} x^T x = \frac{1}{2} \|x\|_2^2,$$
$$G \colon \mathbb{R}^d \to \mathbb{R}, \qquad G(x) = (h \circ F)(x) = \frac{1}{2} \|F(x)\|_2^2.$$

The root x^* of the function F minimizes the function G. The chain rule implies

$$DG(x) = D(h \circ F)(x) = Dh(F(x))\, DF(x) = F(x)^T\, DF(x)$$

or in terms of gradients (obtained by transposition)

$$\nabla G(x) = DF(x)^T\, F(x).$$

So the gradient descent method (6.10) for the special choice of G can be rewritten to

$$x_{n+1} = x_n - \gamma_n DF(x_n)^T\, F(x_n).$$

Compared to Newton's method we do not need to solve a linear system for each iteration. However, the convergence is often considerably slower.

7 Linear systems

It is probably safe to say that the most fundamental problem in applied mathematics is finding the solution to a linear system

$$Ax = b \tag{7.1}$$

for a given matrix $A \in \mathbb{R}^{n\times n}$, a given right-hand side $b \in \mathbb{R}^n$ and an unknown vector $x \in \mathbb{R}^n$. For this reason there are many textbooks on the beautiful topic of numerical linear algebra [14, 27, 26]. After briefly repeating some well-known facts about linear systems, we will focus on the numerical solution of (7.1).

7.1 A little bit of theory

Let us first repeat some of the theory presented in introductory courses on linear algebra.

7.1.1 Solvability

There are several equivalent statements one can make to describe the unique solvability of (7.1). The following theorem collects the most important ones.

Theorem 18. *There is a unique solution to the linear system* (7.1) *if one of the following equivalent statements is fulfilled:*

- *A is invertible.*
- *The columns of A are linearly independent.*
- *The rows of A are linearly independent.*
- *The determinant does not vanish,* $\det(A) \neq 0$.
- *The matrix has full rank,* $\operatorname{rank}(A) = n$.
- *The matrix has a trivial null space,* $\ker(A) = \{0\}$.

7.1.2 Matrix norms

We discussed already general operator norms in Chapter 2. For convenience, we repeat the definition here. Suppose F is linear operator from one normed space V to another normed space W, then the operator norm is given by

$$\|F\|_{V,W} := \sup_{x\neq 0} \frac{\|F(x)\|_W}{\|x\|_V} = \sup_{\|x\|_V=1} \|F(x)\|_W. \tag{7.2}$$

Here we will give examples for specific norms and the case where the operator is given by a matrix $A \in \mathbb{R}^{n\times n}$. To define these norms the matrix A does not need to invertible – in fact it does not even need to be square.

For some $x \in \mathbb{R}^n$ the following vector norms on both spaces V and W,

ℓ_∞ or maximum norm: $\|x\|_\infty = \max_{1\le i\le n} |x_i|$,

ℓ_1 norm: $\|x\|_1 = \sum_{i=1}^n |x_i|$,

ℓ_2 norm: $\|x\|_2 = \left(\sum_{i=1}^n |x_i|^2\right)^{1/2}$

induce the corresponding matrix norms

maximum absolute row sum $\|A\|_\infty = \max_{1\le i\le m} \sum_{j=1}^n |a_{ij}|$,

maximum absolute column sum $\|A\|_1 = \max_{1\le j\le n} \sum_{i=1}^m |a_{ij}|$,

largest singular value $\|A\|_2 = \sqrt{\text{largest eigenvalue of } AA^T}$

for $A \in \mathbb{R}^{m\times n}$.

7.1.3 Condition numbers

By Theorem 2, we know that the relative condition number for solving the linear system (7.1) for some invertible square matrix is bounded by

$$\kappa_r \le \|A\|\|A^{-1}\| =: \text{cond}(A)$$

where $\text{cond}(A)$ is called the condition number of the matrix A. Note that the condition number is only an upper bound for the relative condition number κ_r. Often (though not always) it is nevertheless an accurate measure for the relative output error in x, when the right-hand side b is slightly perturbed. Though it seems that the computation of the condition number requires finding the inverse of the matrix. This computational burden can often be alleviated by lower bounds of the form

$$\|Ax\| \ge K\|x\|$$

for some $K > 0$.

A very important special case is the ℓ_2 condition number for symmetric matrices. Then the condition number becomes

$$\text{cond}(A) = \left|\frac{\lambda_{\max}(A)}{\lambda_{\min}(A)}\right|,$$

where $\lambda_{\max}(A)$ and $\lambda_{\min}(A)$ denote the largest and the smallest eigenvalue in absolute value of A.

7.2 Solving systems directly

There are two very common techniques to solve linear systems numerically: iteratively or directly. Here we will discuss the latter approach.

7.2.1 Forward and backward substitution

Before we start discussing discussing how to devise a general algorithm to solve the linear system (7.1), we start with two special cases. Suppose is either a

lower-triangular $A = L$ or an upper triangular $A = U$ matrix, where

$$L = \begin{pmatrix} \ell_{1,1} & 0 & \dots & \dots & 0 \\ \ell_{2,1} & \ell_{2,2} & 0 & \dots & 0 \\ \ell_{3,1} & \ell_{3,2} & \ddots & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & 0 \\ \ell_{n,1} & \ell_{n,2} & \dots & \ell_{n,n-1} & \ell_{n,n} \end{pmatrix}, \quad U = \begin{pmatrix} u_{1,1} & u_{1,2} & u_{1,3} & \dots & u_{1,n} \\ 0 & u_{2,2} & u_{2,3} & \dots & u_{2,n} \\ \vdots & 0 & \ddots & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & u_{n-1,n} \\ 0 & 0 & \dots & 0 & u_{n,n} \end{pmatrix}.$$

If all diagonal entries of L are equal to, the matrix L is called *unitriangular*. If the entries on the either diagonal are all equal to zero, the matrices are called *strictly* upper or lower triangular. The matrices L and U are invertible if and only if all diagonal entries are nonzero.

(Upper or lower) triangular matrices are quite robust with respect to several operations. In particular:

- A linear combination of triangular matrices is triangular.
- The product of two triangular matrices is triangular.
- The inverse of an invertible triangular matrix is triangular.

All of these conditions hold for upper and lower triangular matrices as well.

The most useful property perhaps is fact that triangular matrices can be solved rather quickly by *forward substitution*

$$x_j = \frac{1}{a_{jj}} \left(b_j - \sum_{k=1}^{j-1} a_{jk} x_k \right)$$

for $1 \leq j \leq n$ and lower triangular matrices as well as *back(ward) substitution*

$$x_j = \frac{1}{a_{jj}} \left(b_j - \sum_{k=j+1}^{n} a_{jk} x_k \right)$$

for $n \leq j \leq 1$ and upper triangular matrices.

7.2.2 Gaussian elimination

One strategy to solve the linear system (7.1) is to first bring it (usually) in upper triangular form and then use back substitution to solve the system. The matrices can be turned into upper triangular form via *Gaussian elimination*:

The way the algorithm is presented it overwrites the original matrix A and right-hand side b. So also the entries a_{ij} and b_i are constantly overwritten. This is somewhat unpractical if we would like to solve for several different right-hand sides.

Though this algorithm works for certain types of matrices, it does not work for all invertible matrices. Consider for example

$$\begin{pmatrix} 0 & 1 & 2 \\ -3 & 5 & 2 \\ 2 & 2 & 2 \end{pmatrix}.$$

```
for columns j = 1, ..., n − 1 do
    for rows i = j + 1, ..., n do
        b_i = b_i − (a_ij / a_jj) b_j;
        row i = row i − (a_ij / a_jj) row j;
    end
end
```

Algorithm 7: Gaussian elimination without pivoting

According to the algorithm, to manipulate the second row we would need to divide by zero. There are easy ways to fix this, which we discuss in the next section. However for certain classes of matrices such as strictly diagonally dominant ones for example, Gaussian elimination is guaranteed to work.

A matrix $A = (a_{ij}) \in \mathbb{R}^{n\times n}$ is called *strictly diagonally dominant*, if the absolute value of all diagonal entries a_{ii} is larger than the sum of the absolute values of the remaining row entries a_{ij}. In other words, if for all $1 \leq i \leq n$

$$\sum_{\substack{j=1\\ j\neq i}}^{n} |a_{ij}| < |a_{ii}|.$$

One can show that such matrices are automatically invertible.

Finally, we discuss the rough cost of Gaussian elimination. To introduce a zero in the first row underneath a_{11}, we need n multiplications and $n-1$ subtractions. Thus in total $2n-1$ operations. Repeating this for the remaining $n-2$ rows leads to a combined cost of

$$(n-1)(2n-1) = 2n^2 - 3n + 1 = \mathcal{O}(2n^2)$$

operations to introduce zeros in the first row. For the second row we would need roughly $\mathcal{O}(2(n-1)^2)$ operations. In total

$$2\sum_{k=1}^{n} k^2 = 2\frac{1}{6}n(n+1)(2n+1) = \mathcal{O}\left(\frac{2}{3}n^3\right).$$

The back solve is comparatively cheap. At step k, we would need k multiplications and $k-1$ subtractions, yielding a total cost of

$$\sum_{k=1}^{n}(k+k-1) = 2\sum_{k=1}^{n} k - n = n(n+1) - n = n^2$$

operations. Thus the total cost of solving a linear system via Gaussian eliminations is on the order of $\mathcal{O}\left(\frac{2}{3}n^3\right)$.

7.2.3 Gaussian elimination with pivoting

In order to overcome the problems of the simple Gaussian elimination algorithm in the previous section, we are going to adjust it slightly. Since the matrix is invertible there will be at least always one row where the current diagonal

element is not zero (otherwise there would be a zero column). So a simple adjustment would be to swap the rows until we find a diagonal element a_{jj} which is not zero. However, we will aim for an even stronger condition and choose the largest in absolute value. This choice has advantages from a stability point of view.

```
for columns j = 1, ..., n − 1 do
    find row i ≥ j such that |a_ij| = max_{k≥j} |a_kj|;
    if i ≠ j then
        swap rows j and i
    end
    for rows i = j + 1, ..., n do
        row i = row i − (a_ij / a_jj) row j;
        b_i = b_i − (a_ij / a_jj) b_j;
    end
end
```

Algorithm 8: Gaussian elimination with pivoting

The example form the previous section with right-hand side $b = (1, 2, 3)^T$ leads after application of Gaussian elimination with pivoting to the following *augmented* coefficient matrix

$$\left(\begin{array}{ccc|c} -3 & 5 & 2 & 2 \\ 0 & \frac{16}{3} & \frac{10}{3} & \frac{13}{3} \\ 0 & 0 & \frac{11}{8} & \frac{3}{16} \end{array}\right).$$

Demos

- GaussianEliminationWithPivoting.m

7.3 Three decompositions

In this section, we reinterpret Gaussian elimination as a factorization of the matrix A. We then proceed to discuss other useful factorizations.

7.3.1 LU decomposition

Consider

$$\begin{pmatrix} 2 & 3 & 1 \\ 2 & 7 & 0 \\ -2 & 5 & -2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 10 \\ 11 \\ -5 \end{pmatrix} \qquad \Longleftrightarrow \qquad Ax = b.$$

We can interpret the first step in Gaussian elimination as multiplying the system with a (nonsingular) lower triangular matrix Λ_1 as follows

$$\begin{pmatrix} 1 & 0 & 0 \\ -1 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} 2 & 3 & 1 \\ 2 & 7 & 0 \\ -2 & 5 & -2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ -1 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} 10 \\ 11 \\ -5 \end{pmatrix} \qquad \Longleftrightarrow \qquad \Lambda_1 A x = \Lambda_1 b.$$

This simplifies to

$$\begin{pmatrix} 2 & 3 & 1 \\ 0 & 4 & -1 \\ 0 & 8 & -1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 10 \\ 1 \\ 5 \end{pmatrix} \qquad \Longleftrightarrow \qquad \Lambda_1 A x = \Lambda_1 b.$$

Now the second step of Gaussian elimination (without pivoting) is equivalent to introducing a second (nonsingular) lower triangular matrix Λ_2 as follows:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -2 & 1 \end{pmatrix} \begin{pmatrix} 2 & 3 & 1 \\ 0 & 4 & -1 \\ 0 & 8 & -1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -2 & 1 \end{pmatrix} \begin{pmatrix} 10 \\ 1 \\ 5 \end{pmatrix} \qquad \Longleftrightarrow \qquad \Lambda_2 \Lambda_1 A x = \Lambda_2 \Lambda_1 b.$$

This finally gives

$$\begin{pmatrix} 2 & 3 & 1 \\ 0 & 4 & -1 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 10 \\ 1 \\ 3 \end{pmatrix} \qquad \Longleftrightarrow \qquad \Lambda_2 \Lambda_1 A x = \Lambda_2 \Lambda_1 b.$$

The matrix

$$U := \Lambda_2 \Lambda_1 A$$

leads to an upper triangular system which we can easily solve via backward substitution. We note that the inverse of the product of both lower triangular matrices

$$L := (\Lambda_2 \Lambda_1)^{-1} = \Lambda_1^{-1} \Lambda_2^{-1}$$

is also lower triangular. Interestingly, the inverses of the matrices Λ_1 and Λ_2 can be written down easily simply by switching the sign of the off-diagonal elements

$$\Lambda_1^{-1} = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ -1 & 0 & 1 \end{pmatrix} \qquad \text{and} \qquad \Lambda_2^{-1} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 2 & 1 \end{pmatrix}.$$

The product one obtains by filling up the unit matrix with subdiagonal entries from Λ_1^{-1} and Λ_2^{-1}, i. e.

$$L := (\Lambda_2 \Lambda_1)^{-1} = \Lambda_1^{-1} \Lambda_2^{-1} = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ -1 & 2 & 1 \end{pmatrix}.$$

Thus we have found the so-called *LU factorization* of the matrix A because

$$LU = (\Lambda_2 \Lambda_1)^{-1} \Lambda_2 \Lambda_1 A = A.$$

Of course one can easily generalize this process to arbitrary matrices:

$$LU = (\Lambda_{n-1} \dots \Lambda_1)^{-1} \Lambda_{n-1} \dots \Lambda_1 A = A.$$

Once an *LU factorization* is found solving the original linear system

$$Ax = LUx = b$$

is a two-step process by solving first

$$Ly = b \qquad \text{and then} \qquad Ux = y.$$

In general one can also include permutation matrices P to account for necessary pivoting. In this case the LU factorization is given by

$$PA = LU.$$

There are several advantages to the LU factorization:

- Computing the L and U matrices requires not more computational work than Gaussian elimination
- Storing L and U does not cost more than storing the original matrix A – note there are only ones on the diagonal of L.
- To solve the same coefficient matrix for several different right-hand sides one needs to compute the LU factorization only once.

Demos

- LUdecomposition.m

7.3.2 Cholesky decomposition

Suppose we have a-priori more information given on the linear system (7.1). In particular, suppose the matrix is symmetric and positive definite. That is

$$A = A^T \qquad \text{and} \qquad x^T Ax > 0$$

for all $x \neq 0$. We write the matrix A as follows

$$A = \begin{pmatrix} a_{11} & z^T \\ z & \widetilde{B}_1 \end{pmatrix},$$

for some matrix $\widetilde{B}_1 \in \mathbb{R}^{(n-1)\times(n-1)}$. Since the matrix A is positive definite all diagonal elements a_{ii} are positive. The first step in the LU factorization gives

$$\Lambda_1 A = \begin{pmatrix} a_{11} & z^T \\ 0 & B_1 \end{pmatrix}$$

for another matrix $B_1 \in \mathbb{R}^{(n-1)\times(n-1)}$ and a lower triangular matrix $\Lambda_1 \in \mathbb{R}^{n\times n}$. However due to symmetry, we find

$$\Lambda_1 A \Lambda_1^T = \begin{pmatrix} a_{11} & 0^T \\ 0 & A_1 \end{pmatrix}.$$

Here $A_1 \in \mathbb{R}^{(n-1)\times(n-1)}$ is again a symmetric and positive definite matrix and so we can repeat the process. Thus exploiting symmetry we can introduce lower triangular, yielding

$$\Lambda_{n-1} \dots \Lambda_1 A \Lambda_1^T \dots \Lambda_{n-1}^T = D,$$

where D is a diagonal matrix with positive entries. Setting again

$$L = (\Lambda_{n-1} \dots \Lambda_1)^{-1}$$

we thus have

$$A = LDL^T.$$

We can simplify this factorization even more: Since all diagonal entries in D are positive, we may as well take the squares of their roots, i. e. $d_i = \sqrt{d_i}\sqrt{d_i}$. This implies we can write

$$D = D^{1/2}D^{1/2},$$

where

$$D^{1/2} = \begin{pmatrix} \sqrt{d_1} & & 0 \\ & \ddots & \\ 0 & & \sqrt{d_n} \end{pmatrix}.$$

Setting $\widetilde{L} = LD^{1/2}$, we thus obtain the *Cholesky decomposition*

$$A = \widetilde{L}\widetilde{L}^T.$$

Since the upper triangular matrix is in this case directly linked to the lower one (by transposition), the computational cost is cut in half compared to previous LU decomposition.

7.3.3 QR decomposition

Another very common decomposition of a matrix is the so-called *QR decomposition*. For $A \in \mathbb{R}^{m\times n}$ it is given by

$$A = QR,$$

where $Q \in \mathbb{R}^{m\times m}$ is an orthogonal matrix (that is $Q^T = Q^{-1}$) and $R \in \mathbb{R}^{m\times n}$ is a generalized upper triangular matrix, in the sense that

$$r_{jk} = 0 \qquad \text{for} \qquad j > k,$$

where $1 \le j \le m$ and $1 \le k \le n$. Suppose we have a QR decomposition of a matrix A. Then $x \in \mathbb{R}^n$ is the solution to (7.1) if and only if $Rx = Q^Tb$.

We collect a couple of well-known facts about orthogonal matrices.

Theorem 19. *Let $Q \in \mathbb{R}^{m\times m}$ be orthogonal and denote with $\|\cdot\|_2$ the Euclidean norm. Then*

- *Q^T is orthogonal*
- *For $x \in \mathbb{R}^m$, we have $\|Qx\|_2 = \|x\|_2 = \|Q^Tx\|_2$.*
- *The product of two orthogonal matrices is orthogonal.*
- *$\|Q\|_2 = 1$*
- *If $A \in \mathbb{R}^{m\times m}$ is invertible, then $\kappa(QA) = \kappa(AQ) = \kappa(A)$.*
- *The columns of Q form an orthonormal basis for $\mathbb{R}^m$.*

Next we study how to actually compute a QR decomposition.

7.3.4 QR decomposition via Gram-Schmidt orthogonalization

One possibility is to apply Gram-Schmidt orthogonalization to the columns of $A \in \mathbb{R}^{m\times n}$. Suppose $m \geq n$ and write

$$A = (a_1, a_2, \ldots, a_n).$$

With Gram-Schmidt orthogonalization we can construct a family of orthonormal vectors $e_1, \ldots, e_n \in \mathbb{R}^m$ as follows:

$$\begin{aligned}
u_1 &:= a_1, & e_1 &:= \frac{u_1}{\|u_1\|_2} \\
u_2 &:= a_2 - (e_1^T a_2)e_1, & e_2 &:= \frac{u_2}{\|u_2\|_2} \\
u_3 &:= a_3 - (e_1^T a_3)e_1 - (e_2^T a_3)e_2, & e_3 &:= \frac{u_3}{\|u_3\|_2} \\
&\vdots & &\vdots \\
u_k &:= a_k - \sum_{j=1}^{k-1}(e_j^T a_k)e_j, & e_k &:= \frac{u_k}{\|u_k\|_2}.
\end{aligned}$$

In particular, the last line implies by orthonormality that

$$e_k^T a_k = e_k^T \left(u_k + \sum_{j=1}^{k-1}(a_k^T e_j)e_j \right) = e_k^T u_k = \|u_k\|_2 e_k^T e_k = \|u_k\|_2.$$

This allows us to write

$$a_k = u_k + \sum_{j=1}^{k-1}(e_j^T a_k)e_j = \|u_k\|_2 e_k + \sum_{j=1}^{k-1}(e_j^T a_k)e_j = \sum_{j=1}^{k}(e_j^T a_k)e_j$$

or

$$\begin{aligned}
a_1 &= (e_1^T a_1)e_1, \\
a_2 &= (e_1^T a_2)e_1 + (e_2^T a_2)e_2, \\
a_3 &= (e_1^T a_3)e_1 + (e_2^T a_3)e_2 + (e_3^T a_3)e_3, \\
&\vdots \\
a_k &= \sum_{j=1}^{k}(e_j^T a_k)e_j.
\end{aligned}$$

Thus in matrix form this yields the *reduced* QR decomposition

$$A = (a_1, a_2, \ldots, a_n) = (e_1, e_2, \ldots, e_n) \begin{pmatrix} e_1^T a_1 & e_1^T a_2 & \ldots & e_1^T a_n \\ 0 & e_2^T a_2 & \ldots & e_2^T a_n \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & e_n^T a_n \end{pmatrix} =: \hat{Q}\hat{R}.$$

By extending $e_1, \ldots, e_n$ to an orthonormal basis of $\mathbb{R}^m$, denoted with $e_1, \ldots, e_n, e_{n+1}, \ldots, e_m$, and inserting zeros in $\hat{R}$, we obtain the (full) QR decomposition

$$A = (a_1, a_2, \ldots, a_n) = (e_1, e_2, \ldots, e_m) \begin{pmatrix} e_1^T a_1 & e_1^T a_2 & \ldots & e_1^T a_n \\ 0 & e_2^T a_2 & \ldots & e_2^T a_n \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & e_n^T a_n \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \ldots & 0 \end{pmatrix} =: QR.$$

We point out that the full QR decomposition introduces just computational overhead. Algorithmically, the reduced one is fully sufficient.

The drawback of computing the QR decomposition via Gram-Schmidt is that it is numerically unstable.

7.3.5 QR decomposition via Householder transformation

We discuss another possibility to compute the QR factorization. The *Householder transformation* is given by the linear mapping described by the (Householder) matrix

$$H(w) = I - \frac{2}{w^T w} w w^T \in \mathbb{R}^{m \times m}.$$

Note that here the Euclidean inner product $w^T w$ produces a scalar. The outer product ww^T is an $m \times m$ matrix. For vectors $x = (x_1, \ldots, x_m), y = (y_1, \ldots, y_m) \in \mathbb{R}^m$ the *outer product* is defined by

$$xy^T = \begin{pmatrix} x_1 y_1 & x_1 y_2 & \ldots & x_1 y_m \\ x_2 y_1 & x_2 y_2 & \ldots & x_2 y_m \\ \vdots & \vdots & \ddots & \vdots \\ x_m y_1 & x_m y_2 & \ldots & x_m y_m \end{pmatrix}.$$

We can interpret the Householder transformation as a reflection along the $m-1$ dimensional space (a hyperplane)

$$w^\perp = \{x \in \mathbb{R}^m \mid w^T x = 0\}$$

for some vector $w \in \mathbb{R}^m$. Note that w itself does not lie in the hyperplane. If we apply the Householder transformation $H(w)$ to this vector, we see that it is reflected,

$$H(w)w = w - \frac{2}{w^T w} w(w^T w) = -w.$$

On the other hand, any element $y \in w^\perp$ in the hyperplane is left unaltered,

$$H(w)y = y - \frac{2}{w^T w} w(w^T y) = y.$$

If we think of the hyperplane as a mirror this is exactly what reflection would do: Any component orthogonal to the hyperplane (in front of the mirror) is

reflected and any component in the hyperplane (in the mirror) is not changed at all. Furthermore, since

$$\begin{aligned} H(w)H(w)^T &= \left(I - \frac{2}{w^T w} ww^T\right)\left(I - \frac{2}{w^T w} ww^T\right) \\ &= I - \frac{4}{w^T w} ww^T + \frac{4}{w^T w} w(w^T w)w^T \\ &= I \end{aligned}$$

we see that the Householder matrix is orthogonal.

Theorem 20. *Let $u \in \mathbb{R}^m$ be arbitrary. Then we can find some $w \in \mathbb{R}^m$ such that the corresponding Householder transformation reflects it to a multiple of the canonical unit vector e_1, i. e.*

$$H(w)u = \begin{pmatrix} \alpha \\ 0 \\ \vdots \\ 0 \end{pmatrix} =: v$$

with $\alpha = \pm\sqrt{u^T u}$.

Proof. Let $0 \neq \gamma \in \mathbb{R}$. Define $w := \gamma(u - v)$. By Theorem 19, we know that $u^T u = v^T v$. Then

$$\begin{aligned} w^T w &= \gamma^2 (u - v)^T (u - v) = \gamma^2 (u^T u - 2u^T v + v^T v) \\ &= \gamma^2 (u^T u - 2u^T v + u^T u) \\ &= 2\gamma u^T (\gamma(u - v)) \\ &= 2\gamma w^T u. \end{aligned}$$

So, we find

$$\begin{aligned} H(w)u &= \left(I - \frac{2}{w^T w} ww^T\right) u = u - \frac{2w^T u}{w^T w} w \\ &= u - \frac{1}{\gamma} w = u - (u - v) \\ &= v. \end{aligned}$$

□

The previous theorem now helps us to manipulate some matrix $A \in \mathbb{R}^{m \times n}$ in such a way that we obtain another matrix in upper triangular form. In the following $\times$ denotes some generic entry which may change from line to line.

To start denote with $u \in \mathbb{R}^m$ in Theorem 20 the first column of A. Then we find some $w_1 \in \mathbb{R}^m$ such that

$$H(w_1)A = \left(\begin{array}{c|ccc} \alpha_1 & \times & \dots & \times \\ \hline 0 & & & \\ \vdots & & A_1 & \\ 0 & & & \end{array}\right)$$

for some matrix $A_1 \in \mathbb{R}^{(m-1)\times(n-1)}$. Similarly, we can find some $\hat{w}_2 \in \mathbb{R}^{m-1}$ such that

$$H(\hat{w}_2)A_1 = \left(\begin{array}{c|ccc} \alpha_2 & \times & \dots & \times \\ \hline 0 & & & \\ \vdots & & A_2 & \\ 0 & & & \end{array}\right)$$

for some $A_2 \in \mathbb{R}^{(m-2)\times(n-2)}$. From this we deduce

$$\left(\begin{array}{c|ccc} 1 & 0 & \dots & 0 \\ \hline 0 & & & \\ \vdots & & H(\hat{w}_2) & \\ 0 & & & \end{array}\right) H(w_1)A = \left(\begin{array}{cc|ccc} \alpha_1 & \times & \times & \dots & \times \\ 0 & \alpha_2 & \times & \dots & \times \\ \hline 0 & 0 & & & \\ \vdots & \vdots & & A_2 & \\ 0 & 0 & & & \end{array}\right).$$

Setting

$$w_2 = \begin{pmatrix} 0 \\ \hat{w}_2 \end{pmatrix} \in \mathbb{R}^m,$$

we note that due to the definition of the Householder matrix, we have

$$H(w_2) = \left(\begin{array}{c|ccc} 1 & 0 & \dots & 0 \\ \hline 0 & & & \\ \vdots & & H(\hat{w}_2) & \\ 0 & & & \end{array}\right).$$

We can simply repeat this process to find n Householder matrices such that

$$Q^T A := H(w_n)\dots H(w_2)H(w_1)A = \begin{pmatrix} \alpha_1 & \times & \dots & \times \\ 0 & \alpha_2 & \dots & \times \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \alpha_n \\ 0 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \dots & 0 \end{pmatrix} =: R$$

We note that Q^T is orthogonal as the product and transposition of orthogonal matrices is still orthogonal.

Note if $m < n$ one can similarly construct a sequence of Householder matrices.

Thus we have constructively proven (twice) the following theorem.

Theorem 21. *For any matrix $A \in \mathbb{R}^{m\times n}$ there exists a QR decomposition.*

8 Linear least squares problem

In the previous chapter, we have considered the case when are given as many equations as unknowns. In practice, it often happens that we are given more equations than unknowns. Consider for example

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 &= b_1 \\ a_{21}x_1 + a_{22}x_2 &= b_2 \\ a_{31}x_1 + a_{32}x_2 &= b_3 \end{aligned}$$

or in general

$$Ax = b$$

for $A \in \mathbb{R}^{m\times n}$ for $m \geq n$ and $x \in \mathbb{R}^n$ and $b \in \mathbb{R}^m$ In general, this linear system will not have a solution as it may be overdetermined. However, we may ask for a *minimal* solution $x^* \in \mathbb{R}^n$ such that

$$\|Ax^* - b\|_2 \leq \|Ax - b\|_2 \tag{8.1}$$

for all $x \in \mathbb{R}^n$. This problem is known as the *linear least squares problem.*

8.1 Normal equations

Indeed (8.1) has a unique solution.

Theorem 22. *The unique solution of the least squares problem* (8.1)*, where A is assumed to have maximal rank, is given by the solution of* normal equations

$$A^TAx = A^Tb. \tag{8.2}$$

Proof. We compute

$$\begin{aligned} \|Ax - b\|_2^2 = (Ax - b)^T(Ax - b) &= (x^TA^T - b^T)(Ax - b) \\ &= x^T(A^TA)x - x^TA^Tb - b^TAx + b^Tb \\ &= x^T(A^TA)x - 2b^TAx + b^Tb, \end{aligned}$$

where we have used that $x^TA^Tb = b^TAx$. Componentwise we obtain

$$\|Ax - b\|_2^2 = \sum_{i,j=1}^{n} x_i(A^TA)_{ij}x_j - 2\sum_{i=1}^{m}\sum_{j=1}^{n} b_iA_{ij}x_j + \sum_{i=1}^{m} b_i^2.$$

We want to minimize this expression. A necessary condition is that the derivatives vanish. So we demand that

$$\begin{aligned} 0 = \frac{\partial}{\partial x_k}\|Ax - b\|_2^2 &= \sum_{j=1}^{n}(A^TA)_{kj}x_j + \sum_{i=1}^{n} x_i(A^TA)_{ik} - 2\sum_{i=1}^{n} A_{ik}b_i \\ = 2\sum_{j=1}^{n}(A^TA)_{kj}x_j - 2\sum_{j=1}^{n} A_{jk}b_j. \end{aligned}$$

The final line follows from the fact that A^TA is a symmetric matrix. So we can deduce

$$\sum_{j=1}^{n}(A^TA)_{kj}x_j = \sum_{j=1}^{n}A_{jk}b_j = \sum_{j=1}^{n}(A^T)_{kj}b_j,$$

which in matrix notation is given by

$$A^TAx = A^Tb.$$

A sufficient condition that ensures we have actually found the minimizer is that the Hessian is positive definite. We find

$$\frac{\partial^2}{\partial x_\ell \partial x_k}\|Ax-b\|_2^2 = 2(A^TA)_{k\ell}.$$

Thus

$$\mathrm{Hess}(\|Ax-b\|_2^2) = A^TA$$

where A^TA is positive definite (A has maximal rank). The positive definiteness of A^TA also implies its invertibility. Hence the solution to (8.2) and thus also (8.1) is unique. □

As an example, suppose we want to find a line $y(x) = ax + b$ through the points

x	-1	0	1	2
y	-2	0	1	3

Then we need to solve the normal equations

$$\begin{pmatrix} 6 & 2 \\ 2 & 4 \end{pmatrix}\begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} 9 \\ 2 \end{pmatrix}.$$

Finding a least squares line (fit) through data points is often referred to as *linear regression*. The

While the above approach gives us the unique solvability, it is numerically not ideal as the condition number $\mathrm{cond}_2(A^TA)$ is a lot larger than the condition number of the original problem $\mathrm{cond}_2(A)$. In fact, using that the inverse of a symmetric matrix is symmetric again, we obtain

$$\begin{aligned}
\mathrm{cond}_2(A^TA) &= \|A^TA\|_2\|(A^TA)^{-1}\|_2 \\
&= \sqrt{\lambda_{\max}\left((A^TA)^2\right)\lambda_{\max}\left((A^TA)^{-2}\right)} \\
&= \sqrt{\lambda_{\max}^2\left(A^TA\right)/\lambda_{\min}^2\left(A^TA\right)} \\
&= \lambda_{\max}\left(A^TA\right)/\lambda_{\min}\left(A^TA\right) \\
&= \|A\|_2^2\|A^{-1}\|_2^2 = \mathrm{cond}_2^2(A).
\end{aligned}$$

In that sense, it is fair to say that compared to the original problem the computational effort is squared. Since for orthogonal matrices Q, we have

$$\mathrm{cond}_2(Q^TQ) = \mathrm{cond}_2(I) = 1,$$

we will discuss numerical methods which rely on the QR decomposition next.

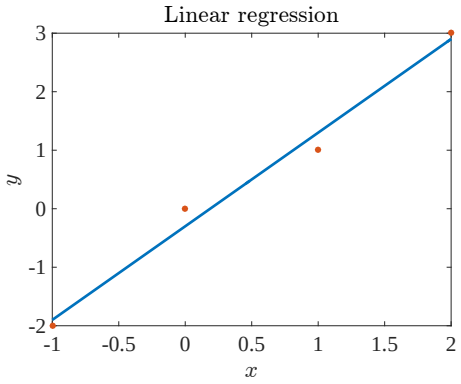


Figure 8.1: The linear line that produces the smallest ℓ_2 error in the sense of (8.1) through the data points.

Demos

- `LinearRegression.m`

8.2 Solving the normal equations with the QR decomposition

Suppose we have given a QR decomposition of the form $A = QR$, where $A \in \mathbb{R}^{m\times n}$ $(m \geq n)$ is given by the linear least squares problem (8.1), $Q \in \mathbb{R}^{m\times m}$ is orthogonal and $R \in \mathbb{R}^{m\times n}$ is a generalized upper triangular matrix, which is related to the upper triangular matrix $R_1 \in \mathbb{R}^{n\times n}$ via

$$R = \begin{pmatrix} \times & \times & \dots & \times \\ 0 & \times & \dots & \times \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \times \\ 0 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \dots & 0 \end{pmatrix} = \begin{pmatrix} & R_1 & \\ 0 & \dots & 0 \\ \vdots & \vdots & \vdots \\ 0 & \dots & 0 \end{pmatrix}.$$

The linear least squares problem (8.1) may then be solved as follows:

$$\begin{aligned} \min_{x\in\mathbb{R}^n} \|Ax - b\|_2 &= \min_{x\in\mathbb{R}^n} \|QRx - b\|_2 = \min_{x\in\mathbb{R}^n} \|Q^T(QRx - b)\|_2 \\ &= \min_{x\in\mathbb{R}^n} \|Rx - Q^T b\|_2. \end{aligned}$$

In the previous calculation, we have used that orthogonal matrices preserve lengths. Introducing the notation

$$Q^T b =: \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}$$

for some $b_1 \in \mathbb{R}^n$ and $b_2 \in \mathbb{R}^{m-n}$, we can even further simplify and obtain

$$\begin{aligned}\min_{x\in\mathbb{R}^n} \|Ax-b\|_2^2 &= \min_{x\in\mathbb{R}^n} \left\| \begin{pmatrix} R_1 \\ 0 \end{pmatrix} x - Q^T b \right\|_2^2 = \min_{x\in\mathbb{R}^n} \left\| \begin{pmatrix} R_1 \\ 0 \end{pmatrix} x - \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} \right\|_2^2 \\ &= \min_{x\in\mathbb{R}^n} \|R_1 x - b_1\|_2^2 + \|b_2\|_2^2\end{aligned}$$

If A has full rank then

$$R_1 x = b_1$$

is uniquely solvable as R_1 has full rank, too. Thus the unique solution of the linear least squares problem (8.1) is given by

$$x^* = R_1^{-1} b_1.$$

It produces the minimal value $\|b_2\|_2$.

What about the computational efficiency of this approach? Let us assume that A is square in order to analyze this (in other words $R = R_1$). In this case

$$\operatorname{cond}_2(R) = \|R\|_2 \|R^{-1}\|_2 = \|Q^T A\|_2 \|A^{-1} Q\|_2 = \|A\|_2 \|A^{-1}\|_2 = \operatorname{cond}_2(A). \tag{8.3}$$

For the penultimate equality we used $\|Q^T A\|_2 = \|A\|_2$ which follows from the fact that orthogonal matrices preserve lengths as well as $\|A^{-1}Q\|_2 = \|A^{-1}\|_2$, which follows from

$$\|A^{-1}Q\|_2 \leq \|A^{-1}\|_2 \|Q\|_2 = \|A^{-1}\|_2$$

and

$$\|A^{-1}\|_2 = \|A^{-1}QQ^T\|_2 \leq \|A^{-1}Q\|_2 \|Q^T\|_2 = \|A^{-1}Q\|_2.$$

So (8.3) indicates that the computational effort is similar to solving the original problem (8.1).

Bibliography

[1] R.H. Bartels, J.C. Beatty, and B.A. Barsky. *An Introduction to Splines for Use in Computer Graphics and Geometric Modeling*. Morgan Kaufmann Series in Comp. Morgan Kaufmann, 1987.

[2] Jean-Paul Berrut and Lloyd N. Trefethen. Barycentric Lagrange Interpolation. *SIAM Review*, 46(3):501–517, 2004.

[3] L. E. J. Brouwer. Über Abbildung von Mannigfaltigkeiten. *Mathematische Annalen*, 71(1):97–115, Mar 1911.

[4] Martin. D. Buhmann. *Radial Basis Functions: Theory and Implementations*, volume 12 of *Cambridge Monographs on Applied and Computational Mathematics*. Cambridge University Press, Cambridge, 2003.

[5] IEEE-754 Calculator. http://babbage.cs.qc.cuny.edu/IEEE-754/.

[6] Carl De Boor. *A Practical Guide to Splines*. Applied mathematical sciences. Springer, Berlin, 2001.

[7] P. Deuflhard. A modified Newton method for the solution of ill-conditioned systems of nonlinear equations with application to multiple shooting. *Numerische Mathematik*, 22(4):289–315, 1974.

[8] P. Deuflhard and A. Hohmann. *Numerische Mathematik 1: Eine algorithmisch orientierte Einführung*. De Gruyter Lehrbuch Series. De Gruyter, 2008.

[9] Peter Deuflhard. *Newton Methods for Nonlinear Problems: Affine Invariance and Adaptive Algorithms*. Springer, 2011.

[10] T. A Driscoll, N. Hale, and L. N. Trefethen. *Chebfun Guide*. Pafnuty Publications, 2014.

[11] Gregory E. Fasshauer. *Meshfree Approximation Methods with MATLAB*, volume 6 of *Interdisciplinary Mathematical Sciences*. World Scientific Publishing, Hackensack, 2007.

[12] M Florenzano. *General Equilibrium Analysis: Existence and Optimality Properties of Equilibria*. Kluwer Academic Pub, 2003.

[13] IEEE Standard for Floating-Point Arithmetic. https://doi.org/10.1109/ieeestd.2008.4610935.

[14] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. Johns Hopkins University Press, Baltimore, 3 edition, 1996.

[15] W. Hackbusch. *The Concept of Stability in Numerical Mathematics*. Springer Series in Computational Mathematics. Springer, 2014.

[16] Jacques Hadamard. Sur les problèmes aux dérivés partielles et leur signification physique. *Princeton University Bulletin*, 13:49–52, 1902.

[17] N. Higham. *Accuracy and Stability of Numerical Algorithms*. SIAM, second edition, 2002.

[18] Nicholas J. Higham. The numerical stability of barycentric Lagrange interpolation. *IMA Journal of Numerical Analysis*, 24(4):547–556, 2004.

[19] Bayram Ali Ibrahimoglu. Lebesgue functions and lebesgue constants in polynomial interpolation. *Journal of Inequalities and Applications*, 2016(1):93, Mar 2016.

[20] Armin Iske. *Multiresolution Methods in Scattered Data Modelling*, volume 37 of *Lecture Notes in Computational Science and Engineering*. Springer, Berlin, 2004.

[21] V.I. Krylov. *Approximate Calculation of Integrals*. Macmillan, 1962.

[22] M. Overton. *Numerical Computing with IEEE Floating Point Arithmetic*. SIAM, 2001.

[23] Richard S. Palais. A simple proof of the Banach contraction principle. *Journal of Fixed Point Theory and Applications*, 2(2):221–223, Dec 2007.

[24] Carl Runge. Über empirische Funktionen und die Interpolation zwischen äquidistanten Ordinaten. *Zeitschrift für Mathematik und Physik*, 46(224–243), 1901.

[25] H. Rutishauser. *Gleichungssysteme, Interpolation und Approximation*. Mathematische Reihe : Lehrbücher und Monographien aus dem Gebiete der exakten Wissenschaften. Springer, 1976.

[26] Lloyd N. Trefethen and David Bau. *Numerical Linear Algebra*. SIAM, 1997.

[27] R.S. Varga. *Matrix Iterative Analysis*. Springer Series in Computational Mathematics. Springer, 1999.

[28] Holger Wendland. *Scattered Data Approximation*, volume 17 of *Cambridge Monographs on Applied and Computational Mathematics*. Cambridge University Press, Cambridge, 2005.