Numerical Analysis

Lecture Notes

Prof. Daniel Kressner
EPFL / MATH / ANCHP

Spring 2025

May 27, 2025

- The following sources were used to prepare these lecture notes:

  1. Buffa, Annalisa. Numerical Analysis. Class notes, EPFL, 2020.
  2. Deuflhard, Peter; Hohmann, Andreas. Numerical analysis in modern scientific computing. [1]
  3. Driscoll, Tobin A.; Braun Richard J. Fundamentals of Numerical Computation. SIAM, 2022. See `https://fncbook.com/`
  4. Golub, Gene H.; Van Loan, Charles F. Matrix computations. [2]
  5. Higham, Nicholas J. Accuracy and stability of numerical algorithms. [3].
  6. Hiptmair, Ralf. Numerik für CSE. Lecture Notes, ETH Zürich, 2007.
  7. Jeltsch, Rolf. Numerische Mathematik. Vorlesungsskript, ETH Zürich, 1997.
  8. Quarteroni, Alfio; Sacco, Riccardo; Saleri, Fausto, Méthodes numériques. [5]

  Please do *not* understand this list as a literature recommendation; the material contained in these lecture notes and discussed during the lecture/exercise is entirely sufficient to prepare for the exam.

- Sections and parts marked by a $\star$ represent supplementary material, which is *not* part of the exam.

- If you spot errors (most likely, there will be plenty), it is a nice gesture to your colleagues if you put them on the Ed Discussion Board. These lecture notes will be constantly updated and errors will be corrected.

Chapter 1

Representation of numbers

> *Since none of the numbers which we take out from logarithmic and trigonometric tables admit of absolute precision, but all are to a certain extent approximate only, the results of all calculations performed by the aid of these numbers can only be approximately true.*
> *It may happen, that in special cases the effect of the errors of the tables is so augmented that we may be obliged to reject a method, otherwise the best, and substitute another in its place.*
> — Carl F. Gauss, *Theoria Motus (1809)*[1]

> Matlab*'s creator Dr. Cleve Moler used to advise foreign visitors not to miss the country's two most awesome spectacles: the Grand Canyon, and meetings of IEEE p754.*
> — William M. Kahan[2]

The aim of this chapter is to understand how numbers are represented in computers. While the set of real numbers is infinite, computers can only represent and work with a finite subset. Therefore, we need to understand which numbers are representable and how the operations are performed in this set of representable real numbers. This topic has regained importance during the last years with the advent of GPUs and TPUs for scientific computing and machine learning. In particular, TPUs are designed to perform a high volume of low precision computation and one cannot expect high accuracy.

Example 1.1 The following expression for Euler's number e is known from Analysis:

$$e = \lim_{n\to\infty} \left(1 + \frac{1}{n}\right)^n .$$

One therefore expects that $e_n = \left(1 + \frac{1}{n}\right)^n$ yields increasingly good approximations to e as n increases. In *exact* arithmetic this is indeed true. On the computer,

[1] Translated and quoted in Higham (2002).
[2] See `http://www.cs.berkeley.edu/~wkahan/ieee754status/754story.html`.

roundoff error affects the accuracy of computations and the *computed* value $\hat{e}_n$ behaves quite differently:

PYTHON

```
# Approximation of e, numpy package needed to include e
import numpy as np
for i in range(1,16):
    n = 10.0 ** i; en = (1 + 1/n) ** n
    print('10^%2d %20.15f %20.15f' % (i,en,en-np.e))
```

n	Computed $\hat{e}_n$	Error $\hat{e}_n - e$
10^1	2.593742460100002	-0.124539368359044
10^2	2.704813829421529	-0.013467999037517
10^3	2.716923932235520	-0.001357896223525
10^4	2.718145926824356	-0.000135901634689
10^5	2.718268237197528	-0.000013591261517
10^6	2.718280469156428	-0.000001359302618
10^7	2.718281693980372	-0.000000134478673
10^8	2.718281786395798	-0.000000042063248
10^9	2.718282030814509	0.000000202355464
10^{10}	2.718282053234788	0.000000224775742
10^{11}	2.718282053357110	0.000000224898065
10^{12}	2.718523496037238	0.000241667578192
10^{13}	2.716110034086901	-0.002171794372145
10^{14}	2.716110034087023	-0.002171794372023
10^{15}	3.035035206549262	0.316753378090216

Initially, the accuracy gets better as n increases, as expected. However, at around $n = 10^8$, the accuracy stagnates and even *gets worse* when n continues to increase; for $n = 10^{15}$ not even a single (decimal) digit of e is computed correctly. ◇

Example 1.2 From Analysis, we know that the Taylor series for the exponential function converges for every $x \in \mathbb{R}$:

$$e^x = \sum_{k=0}^{\infty} \frac{x^k}{k!} = 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + \dots.$$

In practice, one can of course only compute a partial sum

$$s_i(x) = \sum_{k=0}^{i} \frac{x^k}{k!}.$$

The remainder of the Taylor expansion admits the expression

$$e^x - s_i(x) = \frac{e^\xi x^{i+1}}{(i+1)!}$$

for some $\xi \in \mathbb{R}$ strictly between 0 and x (that is, $\xi \in (0, x)$ for $x > 0$ and $\xi \in (x, 0)$ for $x < 0$). If we choose i such that $|x|^{i+1}/(i+1)! \leq \mathsf{tol} \cdot |s_i(x)|$ is satisfied for a (small) chosen tolerance tol then for negative x it holds that

$$|e^x - s_i(x)| \leq \frac{|x|^{i+1}}{(i+1)!} \leq \mathsf{tol} \cdot |s_i(x)| \approx \mathsf{tol} \cdot e^x.$$

x	Computed $\hat{s}_i(x)$	$\exp(x)$	$\frac{\lvert \exp(x)-\hat{s}_i(x)\rvert}{\exp(x)}$
-20	5.6218844674e-09	2.0611536224e-09	1.727542676201181
-18	1.5385415977e-08	1.5229979745e-08	0.*010*205938187564
-16	1.1254180496e-07	1.1253517472e-07	0.*000*058917020257
-14	8.3152907681e-07	8.3152871910e-07	0.*000000*430176956
-12	6.1442133148e-06	6.1442123533e-06	0.*000000*156480737
-10	4.5399929556e-05	4.5399929762e-05	0.*000000000*4544414
-8	3.3546262817e-04	3.3546262790e-04	0.*000000000*788902
-6	2.4787521758e-03	2.4787521767e-03	0.*000000000*333306
-4	1.8315638879e-02	1.8315638889e-02	0.*000000000*530694
-2	1.3533528320e-01	1.3533528324e-01	0.*000000000*273603
0	1.0000000000e+00	1.0000000000e+00	0.*000000000000000*
2	7.3890560954e+00	7.3890560989e+00	0.*000000000*479969
4	5.4598149928e+01	5.4598150033e+01	0.*000000001*923058
6	4.0342879295e+02	4.0342879349e+02	0.*000000001*344248
8	2.9809579808e+03	2.9809579870e+03	0.*000000002*102584
10	2.2026465748e+04	2.2026465795e+04	0.*000000002*143800
12	1.6275479114e+05	1.6275479142e+05	0.*000000001*723845
14	1.2026042798e+06	1.2026042842e+06	0.*000000003*634135
16	8.8861105010e+06	8.8861105205e+06	0.*000000002*197990
18	6.5659968911e+07	6.5659969137e+07	0.*000000003*450972
20	4.8516519307e+08	4.8516519541e+08	0.*000000004*828738

Table 1.1: Numerical approximation of e^x by the truncated Taylor series $s_i(x)$, where i has been chosen such that the relative error is (approximately) bounded by $\mathsf{tol} = 10^{-8}$ in *exact* arithmetic.

In turn, the *relative error* $|e^x - s_i(x)|/|e^x|$ is approximately bounded by tol. For positive x a similar bound can be found by a refined analysis of the remainder (EFY=Exercise For You).

PYTHON

```
def expeval(x, tol):
    #Approximation of e^x
    s = 1; k = 1
    term = 1
    while (abs(term)>tol*abs(s)):
        term = term * x / k
        s = s + term
        k = k + 1
    return s
```

Table 1.1 shows the results for $\mathsf{tol} = 10^{-8}$. In contrast to the theoretical results, the relative error is above the imposed (approximate) bound $\mathsf{tol} = 10^{-8}$ for very negative x. ◇

One goal of this chapter is to better understand (and avoid if possible) the phenomena observed in Examples 1.1 and 1.2. To achieve this, we first need to discuss the representation and approximation of real numbers on computers.

For more interesting stories of what has gone (terribly) wrong in the world because of roundoff errors, see [4].

1.1 Representation of real numbers

The first step in representing a real number on a computer is to express it in terms of its digits with respect to a base $\beta \geq 2$. The following theorem, which we state without proof, gives the so called normalized representation or scientific notation of a number.

Theorem 1.3 *Given a* **base** $2 \leq \beta \in \mathbb{N}$*, every nonzero* $x \in \mathbb{R}$ *can be represented as*

$$x = \pm\beta^e \left(\frac{d_1}{\beta} + \frac{d_2}{\beta^2} + \frac{d_3}{\beta^3} + \cdots \right) \tag{1.1}$$

with the **digits** $d_1, d_2, \ldots, \in \{0, 1, 2, \ldots, \beta - 1\}$*, where* $d_1 \neq 0$*, and the* **exponent** $e \in \mathbb{Z}$*.*

The representation (1.1) becomes unique if we additionally require that there is an infinite subset $\mathbb{N}_1 \subset \mathbb{N}$ such that $d_k \neq \beta - 1$ for all $k \in \mathbb{N}_1$. Otherwise, $x = +10^1 \cdot 0.1$ and $x = +10^0 \cdot 0.999\ldots$ would be two different representations of the same number. However, this aspect will be irrelevant for us; on a computer we can only work with a finite number of digits anyway.

System	β	Digits
Decimal	10	$0, 1, 2, \ldots, 9$
Binary	2	$0, 1$
Octal	8	0,1,2,...,7
Hexadecimal	16	$0, 1, 2, \ldots, 9$, A,B,C,D,E,F

Les doits ou les poings*

In daily life we mostly use the decimal system ($\beta = 10$). But there are exceptions, $\beta = 12$ also plays a role (hours, months, dozen / douzaine). Also, some tribes used their toes for counting as well ($\beta = 20$). Some regions of the world also use $4, 8$ or 16, see `https://en.wikipedia.org/wiki/Numeral_system`. For obvious reasons, almost every computer operates with a binary system ($\beta = 2$) and the hexadecimal system is only used for representing binary numbers more conveniently. A computer developed in Moscow at the end of the 1950ies was based on the ternary system ($\beta = 3$) but it was not a huge success.

To compute the representation of Theorem 1.3 for given $x \in \mathbb{R} \setminus \{0\}$, the following algorithm can be used:

Determine $e \in \mathbb{Z}$ such that $x \in \beta^e \tilde{x}$ with $\beta^{-1} \leq \tilde{x} < 1$. Set $j = 1$.
while $\tilde{x} \neq 0$ **do**
  Set $x = \beta\tilde{x}$.
  Decompose $x = d_j + \tilde{x}$ such that $d_j \in \mathbb{N}, 0 \leq d_j \leq \beta - 1$ and $0 \leq \tilde{x} < 1$.
  $j \leftarrow j + 1$.
end while

The binary system has some unexpected consequences. As we will see below, the decimal number 0.2 *cannot* be represented by a finite binary fraction, that is,

the algorithm above does not terminate. To avoid this effect, some specialized applications in the finance industry use the decimal system. In most cases, this is emulated by software and, consequently, quite slow. Hardware processors, which directly support decimal operations, are rare; an example was the IBM Power6 processor.

Theorem 1.4 *Consider a nonzero rational number $x = p/q$, where $p, q \in \mathbb{Z}$ have no common divisor. Then x has a finite representation in base $\beta \geq 2$ if and only if each of the prime factors of the denominator q divides β.*

Example 1.5 In base $\beta = 10$, a rational number has a finite representation if and only if it can be written as $\pm\frac{p}{2^n 5^m}$ for some $p, n, m \in \mathbb{N}$. This corresponds to the known fact that in base 10, if the divisor has a factor that is neither 2 nor 5, then the Euclidean division does not finish and becomes periodic.

In base $\beta = 2$ instead, to have a finite representation, a number needs to be written as $\pm\frac{p}{2^n}$ for some $p, n \in \mathbb{N}$. For instance, $x = \frac{1}{5} = (0.2)_{10}$ does not have a finite representation in base 2. This can be checked by applying the algorithm above:

$$\begin{aligned} x &= 2^{-2}0.8 && \Rightarrow e = -2, \tilde{x} = 0.8 \\ x &= 2\tilde{x} = 1.6 && \\ x &= 1 + 0.6 && \Rightarrow d_1 = 1, \tilde{x} = 0.6 \\ x &= 2\tilde{x} = 1.2 && \Rightarrow d_2 = 1, \tilde{x} = 0.2 \end{aligned}$$

After two loops we are again at 0.2 and the algorithm becomes cyclic. Therefore, $\frac{1}{5} = (0.00110011\ldots)_2 = \left(0.\overline{0011}\right)_2$. ◇

1.2 Floating point numbers on computers

Computers can only store and calculate with finitely many numbers $\mathbb{F} \subset \mathbb{R}$. Apart from specialized devices, such as audio decoding hardware devices, nearly all processors operate with floating point numbers.

Definition 1.6 (Floating point numbers $\mathbb{F}(\beta, t, e_{\min}, e_{\max})$) *Consider $\beta \in \mathbb{N}$, $\beta \geq 2$ (**base**), $t \in \mathbb{N}$ (**length of mantissa/significand**), and $e_{\min} < 0 < e_{\max}$ with $e_{\min}$, $e_{\max} \in \mathbb{Z}$ (**range of exponent**). Then the set $\mathbb{F} = \mathbb{F}(\beta, t, e_{\min}, e_{\max}) \subset \mathbb{R}$ is defined as*

$$\mathbb{F} := \left\{ \pm\beta^e \left(\frac{d_1}{\beta} + \frac{d_2}{\beta^2} + \cdots + \frac{d_t}{\beta^t} \right) : \begin{array}{l} d_1, \ldots, d_t \in \{0, \ldots, \beta - 1\}, \\ d_1 \neq 0, \\ e \in \mathbb{Z},\ e_{\min} \leq e \leq e_{\max}. \end{array} \right\} \cup \{0\}.$$

Example 1.7 The floating point number $+2^3(0.1011)_2 = (5.5)_{10}$ belongs to $\mathbb{F}(2, 4, -1, 4)$, but neither to $\mathbb{F}(2, 3, -1, 4)$, nor to $\mathbb{F}(2, 4, -2, 2)$.

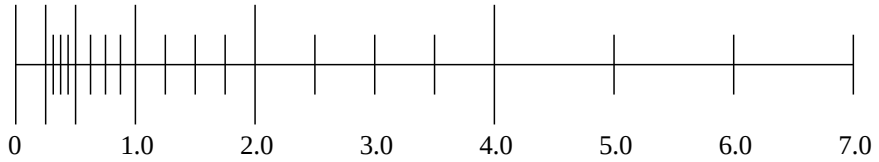
Let us consider the set $\mathbb{F} = \mathbb{F}(10, 3, -2, 2)$, that is, $\beta = 10$, $t = 3$, and $-2 \le e \le 2$. Then:

$$\begin{aligned} 23.4 &= +10^2\,(0.234) && \in \mathbb{F} \\ -53.8 &= -10^2\,(0.538) && \in \mathbb{F} \\ 3.141 &= +10^1\,(0.3141) && \notin \mathbb{F} \quad (4 > 3 \text{ significant digits}) \\ 3.1 &= +10^1\,(0.310) && \in \mathbb{F} \end{aligned}$$

◇

It is important to keep in mind that the elements of $\mathbb{F}$ are not uniformly distributed on the real line.

Example: Nonnegative floating point numbers for $\beta = 2, t = 3, e_{\min} = -1, e_{\max} = 3$:



Lemma 1.8 *For $\mathbb{F} = \mathbb{F}(\beta, t, e_{\min}, e_{\max})$ one has*

$$x_{\min}(\mathbb{F}) := \min\{x \in \mathbb{F} : x > 0\} = \beta^{e_{\min}-1}, \tag{1.2}$$

$$x_{\max}(\mathbb{F}) := \max\{x \in \mathbb{F}\} \qquad = \beta^{e_{\max}}(1 - \beta^{-t})\,. \tag{1.3}$$

Proof. In view of Definition 1.6, the verification of (1.2)–(1.3) comes down to determining the range of the mantissa. Consider $x \in \mathbb{F}$ with mantissa $d = \sum_{k=1}^{t} d_k \beta^{-k}$. Because of $d_1 \neq 0$ we have

$$\begin{array}{ccc} \beta^{-1} \le d & \le \displaystyle\sum_{k=1}^{t} \beta^{-k}(\beta - 1) = 1 - \beta^{-t} \\ \uparrow & \uparrow & \\ d_1 \ge 1 & d_k \le \beta - 1\,. & \end{array}$$

The smallest positive number is obtained by choosing the mantissa $d_1 = 1, d_2 = 0, d_3 = 0, \ldots$ and the exponent $e = e_{\min}$. The largest number is obtained by choosing the mantissa $d_1 = \beta - 1, d_2 = \beta - 1, d_3 = \beta - 1, \ldots$ and the exponent $e = e_{\max}$. □

Lemma 1.9 *The distance between a floating point number x satisfying $x_{\min}(\mathbb{F}) < |x| < x_{\max}(\mathbb{F})$ and the nearest floating point number is* at least $\beta^{-1}\epsilon_M|x|$ *and* at most $\epsilon_M|x|$, *where $\epsilon_M = \beta^{1-t}$ is the distance of* 1 *to the next larger floating point number.*

Proof. Without loss of generality, we may assume that $x \in \mathbb{F}$ is positive and that $e = 0$, which implies that $\beta^{-1} \le x \le 1 - \beta^{-t}$. The next larger floating point number

x_+ is obtaining by adding 1 to the last digit d_t of the mantissa, which corresponds to adding β^{-t} to x. In turn, the relative distance between x and x_+ satisifies

$$\frac{x_+ - x}{x} \leq \beta^{-t}\beta = \epsilon_M, \quad \frac{x_+ - x}{x} \geq \frac{\beta^{-t}}{1-\beta^{-t}} \geq \beta^{-t} = \epsilon_M/\beta.$$

In the same manner, analogous bounds are shown for the relative distance between x and the next smaller number x_-. □

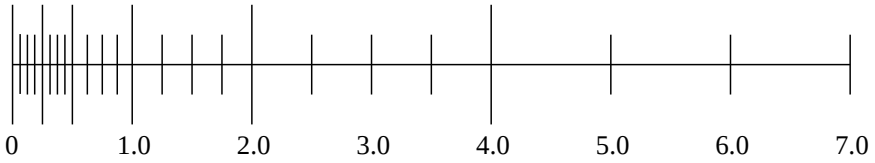
Lemma 1.9 demonstrates that the length of the mantissa determines the relative accuracy of $\mathbb{F}$; the exponent bounds $e_{\min}$, $e_{\max}$ *determine the range of* $\mathbb{F}$ *but not its accuracy.* The number ϵ_M of Lemma 1.9 is usually called *machine precision.*

Subnormal numbers*

As visible in the picture above, there is a "gap" of $\beta^{e_{\min}-1}$ between 0 and the smallest nonnegative floating point number. This gap is caused by the normalization $d_1 \neq 0$ of the mantissa when $e = e_{\min}$. The gap is bridged by adding the so called **subnormal** numbers:

$$\widehat{\mathbb{F}} := \mathbb{F} \cup \left\{ \pm\beta^{e_{\min}} \left(\frac{d_2}{\beta^2} + \cdots + \frac{d_t}{\beta^t} \right) : \ d_2, \ldots, d_t \in \{0, \ldots, \beta - 1\} \right\}, \tag{1.4}$$

where one excludes the case $d_2 = \cdots = d_t = 0$. In other words, one adds numbers with minimal exponent $e_{\min}$ and $d_1 = 0$.
Example: $\widehat{\mathbb{F}}$ for $\beta = 2, t = 3, e_{\min} = -1, e_{\max} = 3$:



Two important remarks:

1. Lemma 1.9 does *not* hold for subnormal numbers, which have a lower relative accuracy.

2. While subnormal numbers are supported by most processors; the computations can slow down significantly because processors are often not optimized to work with subnormal numbers.

In the following, we will ignore subnormal numbers to simplify the discussion.

IEEE standard

IEEE 754 describes the standard for storing and operating with floating point numbers. It also describes the treatment of exceptions ($1/0, \infty \cdot \infty, \infty - \infty, \ldots$). These are the two most widely used formats in base $\beta = 2$:

Name	Size	Mantissa	Exponent	$x_{\min}$	$x_{\max}$
Single precision	32 bits	24 bits	8 bits	10^{-38}	10^{+38}
Double precision	64 bits	53 bits	11 bits	10^{-308}	10^{+308}

Double precision corresponds to $\mathbb{F}(2, 53, -1022, 1023)$ and is the standard on central processing units (CPUs) . One bit of the mantissa is used for the sign. On the other hand, d_1 is *not* saved because the normalization always implies $d_1 = 1$ for $\beta = 2$. In turn, there are $t = 53$ bits for the mantissa. The exponent field is an (unsigned) integer from 0 to 2047; shifted such that it represents the range -1022 to $+1023$ (the fields all 0s and all 1 are reserved for special numbers).

In PYTHON all operations with real numbers are executed in double precision by default. Variables in single precision are generated with the command `numpy.float32()` from the `numpy` package.

PYTHON

```
import sys
sys.float_info.min              # 2.2251e-308
sys.float_info.max              # 1.7977e+308
1 / 0                           # Divide by zero error
3 * float('inf')                # inf
-1 / 0                          # Divide by zero error
0 / 0                           # Divide by zero error
float('inf') - float('inf')     # nan
```

Somewhat surprisingly, and not in accordance with the IEEE 754 standard[3], PYTHON throws an error when a division by zero occurs, instead of returning a signed ∞. To avoid this, one can use `float64` from numpy. For example, `1/numpy.float64(0)` returns `inf` (as it should).

In machine learning, precision is much less important compared to typical applications in scientific computing. On the other hand, speed and memory are key, which makes the bfloat16 format a popular choice that is utilized in many CPUs, GPUs, and AI processors.

Name	Size	Mantissa	Exponent	$x_{\min}$	$x_{\max}$
bfloat16	16 bits	8 bits	8 bits	10^{-38}	10^{+38}
FP8	8 bits	4 bits	4 bits	10^{-2}	240

The use of FP8, which is not standardized yet, is partly responsible for the success of Deepseek.[4] The narrow range of numbers makes it quite difficult to work with.

1.3 Rounding

A real number $x \in \mathbb{R}$ (think of, $\sqrt{2}$ or π) can, in general, not be represented exactly in the computer. The process of approximating x by a number in a floating point

[3] See also `https://wusun.name/blog/2017-12-18-python-zerodiv/` for a discussion.

[4] See `https://verticalserve.medium.com/how-deepseek-optimized-training-fp8-framework-74e3667a2d4a`.

number in $\mathbb{F} = \mathbb{F}(\beta, t, e_{\min}, e_{\max})$ is called **rounding**. More concretely, rounding is a function

$$\mathsf{fl} : r(\mathbb{F}) \to \mathbb{F}$$

which maps $x \in r(\mathbb{F})$ to the[5] nearest element in $\mathbb{F}$. Here, $r(\mathbb{F})$ denotes the range of $\mathbb{F}$:

$$r(\mathbb{F}) := \{x \in \mathbb{R} : x_{\min}(\mathbb{F}) \le |x| \le x_{\max}(\mathbb{F})\};$$

see also Lemma 1.8.

The map fl is not uniquely determined when x is exactly in the middle between two floating point numbers (this *not* a rare event when $\beta = 2$). There are several strategies to break the tie in this situation. According to the IEEE 754 standard, fl chooses the element for which the last digit of the mantissa is even (for $\beta = 2$ this is zero). The following example illustrates this convention.

Example 1.10 Let $\beta = 10$, $t = 3$. Then $\mathsf{fl}(0.9996) = 1.0$, $\mathsf{fl}(0.3345) = 0.334$, $\mathsf{fl}(0.3355) = 0.336$. ◇

The following theorem provides a useful and tight estimate of the relative error of fl.

Theorem 1.11 *For every $x \in r(\mathbb{F})$ there exists $\delta \equiv \delta(x) \in \mathbb{R}$ such that*

$$\mathsf{fl}(x) = x(1+\delta), \quad |\delta| \le u,$$

where $u = u(\mathbb{F}) := \frac{1}{2}\epsilon_M = \frac{1}{2}\beta^{1-t}$.

Proof. Without loss of generality, we may assume $x > 0$. According to Theorem 1.3 we can represent x as follows:

$$x = \mu \cdot \beta^{e-t}, \quad \beta^{t-1} \le \mu < \beta^t.$$

This shows that x is between the adjacent floating point numbers $y_1 = \lfloor \mu \rfloor \beta^{e-t}$, $y_2 = \lceil \mu \rceil \beta^{e-t}$, and hence $\mathsf{fl}(x) \in \{y_1, y_2\}$. Because x is rounded to the nearest floating point number, we have $|\mathsf{fl}(x) - x| \le |y_2 - y_1|/2 = \beta^{e-t}/2$, where the last equality follows from the definition of y_1, y_2. This implies that

$$\left| \frac{\mathsf{fl}(x) - x}{x} \right| \le \frac{\frac{\beta^{e-t}}{2}}{\mu \cdot \beta^{e-t}} \le \frac{1}{2}\beta^{1-t} = u,$$

establishing the claim of the theorem. □

[5] More precisely, one would need to write *a* nearest element.

The quantity $u = u(\mathbb{F}) := \frac{1}{2}\beta^{1-t}$ of Theorem 1.11 is called **unit roundoff** and features prominently in every round-off error analysis.

Example 1.12
Double $u = 2^{-53} \approx 1.11 \times 10^{-16}$
Single $u = 2^{-24} \approx 5.96 \times 10^{-8}$
In PYTHON u is computed by `sys.float_info.epsilon/2` and `numpy.finfo(numpy.float32).eps/2`, respectively.

The following variation of Theorem 1.11 is sometimes useful.

Theorem 1.13 *For every $x \in r(\mathbb{F})$ there exists $\delta \in \mathbb{R}$ such that*

$$\mathsf{fl}(x) = \frac{x}{1+\delta}, \quad |\delta| < u.$$

Proof. EFY. □

It is important to keep in mind that the quantity δ in Theorems 1.11 and 1.13 depends on x. In a round-off error analysis one does not operate with its explicit value but only with the property that $|\delta| < u$.

1.4 Elementary operations in $\mathbb{F}$ and round-off error

We now let '$\circ$' denote any of the four elementary binary operations:

$$\circ \in \{+, -, *, /\}.$$

The set $\mathbb{R}$ is closed under these operations, that is, with the exception of division by zero, $\circ$ maps two real numbers again to a real number. This closedness property does not hold for $\mathbb{F}$. To map the result back to $\mathbb{F}$, one needs to round. It would be reasonable to require a result of an elementary operation on a computer to be equal to what would have been obtained from *first computing the result exactly and then rounding it* afterwards. In practice, a slightly weaker requirements is imposed, which follows from combining the "first computing exactly then rounding" paradigm with Theorem 1.11.

Definition 1.14 (Standard model of rounding) *For every $x, y \in r(\mathbb{F})$ there exists $\delta \in \mathbb{R}$ such that*

$$\mathsf{fl}(x \circ y) = (x \circ y)(1+\delta), \quad |\delta| \le u, \quad \circ \in \{+, -, *, /\}. \tag{1.5}$$

In analogy to Theorem 1.13, an alternative to (1.5) is to require

$$\mathsf{fl}(x \circ y) = \frac{x \circ y}{1+\delta'}, \quad |\delta'| \le u, \quad \circ \in \{+, -, *, /\}. \tag{1.6}$$

It is important to remark, once more, that the quantities δ, δ' above depend on the inputs x, y and, in particular, they may also depend on the order of x, y. Although $\mathsf{fl}(x \circ y) = \mathsf{fl}(y \circ x)$ often holds in practice, commutativity is not and cannot be taken for granted when operating in floating point arithmetic. More importantly, associativity is frequently violated - even under the stronger "first computing exactly then rounding" paradigm.

Example 1.15 Let $\beta = 10$, $t = 2$. Then

$$\begin{aligned} & \mathsf{fl}(\mathsf{fl}(70+74)+74) = \mathsf{fl}(140+74) = 210 \\ \neq\ & \mathsf{fl}(70+\mathsf{fl}(74+74)) = \mathsf{fl}(70+150) = 220 \end{aligned}$$

and

$$\begin{aligned} & \mathsf{fl}(\mathsf{fl}(110-99)-10) = \mathsf{fl}(11-10) = 1 \\ \neq\ & \mathsf{fl}(110+\mathsf{fl}(-99-10)) = \mathsf{fl}(110-110) = 0. \end{aligned}$$

$\diamond$

The property (1.5) is also desirable for elementary functions $f \in \{\exp, \sin, \cos, \tan, \ldots\}$:

$$\mathsf{fl}(f(x)) = f(x)(1+\delta), \quad |\delta| \leq u. \tag{1.7}$$

The development and implementation of a numerical method that computes $f(x)$ in finite precision arithmetic such that the result satisfies (1.7) is by no means trivial. It is very difficult predict how many digits of $f(x)$ need to computed accurately such that the result obtained after rounding satisfies (1.7).[6]

1.5 Round-off error analysis

The standard model of rounding allows us to estimate the propagation of roundoff errors in a computation. As an important example let us consider the computation of the inner product of two vectors $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$, that is,

$$\mathbf{x}^{\mathsf{T}}\mathbf{y} = \sum_{k=1}^{n} x_k y_k \ .$$

For the partial sums $s_i = \sum_{k=1}^{i} x_k y_k$, $i = 1, 2, ...$, evaluated in floating point arithmetic $\mathbb{F}$ according to the standard model, one obtains

$$\begin{aligned} \widehat{s}_1 &= \mathsf{fl}(x_1 y_1) = x_1 y_1 (1+\delta_1), \\ \widehat{s}_2 &= \mathsf{fl}(\widehat{s}_1 + x_2 y_2) = \big(\widehat{s}_1 + x_2 y_2 (1+\delta_2)\big)(1+\delta_3) \\ &= x_1 y_1 (1+\delta_1)(1+\delta_3) + x_2 y_2 (1+\delta_2)(1+\delta_3), \end{aligned}$$

[6] In the literature this effect is called *Table Maker's Dilemma*; see also `http://perso.ens-lyon.fr/jean-michel.muller/Intro-to-TMD.htm`.

where $|\delta_i| \leq u$ holds for all δ_i. To simplify the notation, it is common practice to drop indices and let δ denote an arbitrary real number with $|\delta| \leq u$. Using this convention, one gets

$$\begin{aligned} \widehat{s}_3 &= \mathsf{fl}(\widehat{s}_2 + x_3 y_3) = \big(\widehat{s}_2 + x_3 y_3(1+\delta)\big)(1+\delta) \\ &= x_1 y_1 (1+\delta)^3 + x_2 y_2 (1+\delta)^3 + x_3 y_3 (1+\delta)^2. \end{aligned}$$

By induction, we obtain

$$\widehat{s}_n = x_1 y_1 (1+\delta)^n + x_2 y_2 (1+\delta)^n + x_3 y_3 (1+\delta)^{n-1} + \cdots + x_n y_n (1+\delta)^2. \quad (1.8)$$

The following lemma can be used to simplify this expression.

Lemma 1.16 *Let $|\delta_i| \leq u(\mathbb{F})$ for $i = 1, \ldots, n$ and $n < 1/u(\mathbb{F})$. Then there exists $\theta_n \in \mathbb{R}$ such that*

$$\prod_{i=1}^{n} (1+\delta_i) = 1 + \theta_n, \quad \textit{where} \quad |\theta_n| \leq \frac{nu(\mathbb{F})}{1 - nu(\mathbb{F})} =: \gamma_n(\mathbb{F}). \quad (1.9)$$

Proof. The proof proceeds by induction. For $n = 1$, the statement trivially holds. Induction hypothesis: The statement holds for $n-1$ with $n \geq 2$.

Induction step: Using the induction hypothesis, one obtains

$$\prod_{i=1}^{n} (1+\delta_i) = (1+\delta_n)(1+\theta_{n-1}) =: 1 + \theta_n \quad , \quad \theta_n := \delta_n + (1+\delta_n)\theta_{n-1},$$

$$|\theta_n| \leq u + (1+u)\frac{(n-1)u}{1-(n-1)u} \quad = \quad \frac{nu}{1-(n-1)u} \leq \gamma_n \, .$$

□

Applying Lemma 1.16 to (1.8) gives

$$\widehat{s}_n = x_1 y_1 (1+\theta_n) + x_2 y_2 (1+\theta'_n) + x_3 y_3 (1+\theta_{n-1}) + \cdots + x_n y_n (1+\theta_2), \quad (1.10)$$

where $|\theta_j| \leq \gamma_j$ and $|\theta'_n| \leq \gamma_n$. Using monotonicity, $0 < \gamma_j \leq \gamma_n$, one therefore obtains

$$\widehat{s}_n = (\mathbf{x} + \triangle\mathbf{x})^{\mathsf{T}} \mathbf{y} = \mathbf{x}^{\mathsf{T}} (\mathbf{y} + \triangle\mathbf{y}), \quad |\triangle\mathbf{x}| \leq \gamma_n |\mathbf{x}|, \quad |\triangle\mathbf{y}| \leq \gamma_n |\mathbf{y}|, \quad (1.11)$$

where $|\mathbf{x}|$ is the vector with elements $|x_i|$ and inequalities between vectors and matrices are understood componentwise. Let us remark that the bounds in (1.11) are – in contrast to (1.10) – independent of the order of summation.

The result (1.11) is an example for what is called a **backward error**. It states that the computed result (inner product) is the exact inner product of slightly perturbed input data ($\mathbf{x}$ and $\mathbf{y}$). Since the inputs are usually perturbed by roundoff error anyway (for example when storing the data as floating point numbers), a small backward error is a very desirable property. It implies that the algorithm attains an accuracy (nearly) at the level of the accuracy of the input data. The actual

error in the computed result is called **forward error**. From (1.11) one obtains the following bound on the forward error:

$$\left|\mathbf{x}^{\mathsf{T}}\mathbf{y} - \widehat{s}_n\right| \leq \gamma_n \sum_{i=1}^{n} |x_i y_i| = \gamma_n |\mathbf{x}|^{\mathsf{T}} |\mathbf{y}|. \tag{1.12}$$

If $|\mathbf{x}^{\mathsf{T}}\mathbf{y}| \approx |\mathbf{x}|^{\mathsf{T}}|\mathbf{y}|$ then the relative error $\left|\mathbf{x}^{\mathsf{T}}\mathbf{y} - \mathsf{fl}(\mathbf{x}^{\mathsf{T}}\mathbf{y})\right| / |\mathbf{x}^{\mathsf{T}}\mathbf{y}|$ is small. If, on the other hand, $|\mathbf{x}^{\mathsf{T}}\mathbf{y}| \ll |\mathbf{x}|^{\mathsf{T}}|\mathbf{y}|$ then one cannot expect a small relative error.

1.6 Cancellation

Numerical cancellation happens when two numbers that are nearly equal *and* already affected by roundoff error are subtracted from each other. Cancellation is the number one reason to look for when errors get massively amplified in the course of a computation.

Example 1.17 Consider

$$f(x) = \frac{1 - \cos(x)}{x^2}\,, \quad x = 1.2 \times 10^{-5}.$$

Rounding $\cos(x)$ to 10 decimal digits gives

$$\widehat{c} = 0.9999\ 9999\ 99\,,$$

and

$$1 - \widehat{c} = 0.0000\ 0000\ 01\,.$$

Therefore $(1 - \widehat{c})/x^2 = 10^{-10}/1.44 \times 10^{-10} = 0.6944\ldots$. However, since we know that $0 \leq f(x) < \frac{1}{2}$ for $x \neq 0$ it is obvious that the computed result is completely wrong. ◇

To see the general picture, let us consider $a, b \in \mathbb{R}$ and

$$\widehat{a} = \mathsf{fl}(a) = a(1 + \delta_a), \quad \widehat{b} = \mathsf{fl}(b) = b(1 + \delta_b)$$

with $|\delta_a| \leq u$, $|\delta_b| \leq u$. Then $x = a - b$ and $\widehat{x} = \widehat{a} - \widehat{b}$ satisfy

$$\left|\frac{x - \widehat{x}}{x}\right| = \left|\frac{-a\delta_a + b\delta_b}{a - b}\right| \leq u\ \frac{|a| + |b|}{|a - b|} \tag{1.13}$$

The relative error in the computation $x = a - b$ is large when

$$|a - b| \ll |a| + |b|.$$

Let us emphasize that it is crucial in the discussion above that a and b are already affected by roundoff error. The subtraction itself is carried out *without roundoff error* for $a \approx b$; it just amplifies the existing errors.

Roundoff errors can also cancel each other, that is, a very inaccurate intermediate result does not necessarily lead to very inaccurate final result. This effect is demonstrated by the following example.

x	Alg. 1	Alg. 2
10^{-3}	*1.0005*236	*1.0005002*
10^{-4}	*1.000*1659	*1.0000499*
10^{-5}	*1.00*13580	*1.0000050*
10^{-6}	0.9536743	*1.0000005*
10^{-7}	*1.*1920929	*1.0000001*

Table 1.2: Results of Algorithms 1 and 2 in single precision. Correctly computed digts are italic.

Example 1.18 (Computation of $(e^x - 1)/x$ for $x \to 0+$) We aim at computing

$$f(x) = (e^x - 1)/x = \sum_{i=0}^{\infty} \frac{x^i}{(i+1)!} \tag{1.14}$$

in IEEE single precision by two different methods:

PYTHON
```
# Algorithm 1
import numpy as np
if x == 0:
    f = 1
else:
    f = ( np.exp(x) - 1 ) / x;
```

PYTHON
```
# Algorithm 2
import numpy as np
y = np.exp(x)
if y == 1:
    f = 1
else:
    f = (y - 1) / np.log(y);
```

The obtained results are shown in Table 1.2.[7] To understand why Algorithm 1 yields much worse accuracy, we consider $x = 9.0 \times 10^{-8}$ and assume that the implementations of the elementary functions $\exp(\cdot)$ and $\log(\cdot)$ satisfy the standard model (1.7). The first 9 decimal digits of the exact result are

$$\frac{e^x - 1}{\log e^x} = 1.00000005\,.$$

Algorithm 1 yields

$$\mathsf{fl}\Big(\frac{\mathsf{fl}(\mathsf{fl}(e^x) - 1)}{x}\Big) = \mathsf{fl}\Big(\frac{1.19209290 \times 10^{-7}}{9.\overline{0} \qquad \times 10^{-8}}\Big) = 1.32454766\,;$$

In contrast, Algorithm 2 yields

$$\mathsf{fl}\Big(\frac{\mathsf{fl}(\widehat{y} - 1)}{\mathsf{fl}(\log \widehat{y})}\Big) = \mathsf{fl}\Big(\frac{1.19209290 \times 10^{-7}}{1.19209282 \times 10^{-7}}\Big) = 1.00000006\,.$$

[7] It can be difficult to reproduce these results, especially when working in compiled languages. Some processors work internally (registers) with higher-precision 80-bit arithmetic; so the results may depend on whether the registers are flushed, which in turn depends on the compiler, etc. The behavior might also change when using a debugger or displaying intermediate results, leading to the notorious heisenbugs.

One observes that Algorithm 2 computes, because of cancellation, a very inaccurate result for the numerator $e^x - 1 = 9.00000041 \times 10^{-8}$ and the denominator $\log e^x = 9 \times 10^{-8}$; but most of the roundoff error vanishes during the division!

The described phenomenon can be explained by a roundoff error analysis of Algorithm 2. We have $\widehat{y} = e^x(1+\delta)$, $|\delta| \le u$. If $\widehat{y} = 1$, it follows that

$$e^x(1+\delta) = 1 \Longleftrightarrow x = -\log(1+\delta) = -\delta + \delta^2/2 - \delta^3/3 + \dots ,$$

and hence

$$\widehat{f} = \mathsf{fl}\big(1 + x/2 + x^2/6 + \dots\big|_{x=-\delta+O(\delta^2)}\big) = 1\,. \tag{1.15}$$

If $\widehat{y} \neq 1$, it follows that

$$\widehat{f} = \mathsf{fl}\big((\widehat{y}-1)/\log\widehat{y}\big) = \frac{(\widehat{y}-1)(1+\delta_1)}{\log\widehat{y}(1+\delta_2)}\,(1+\delta_3), \quad |\delta_i| \le u\,. \tag{1.16}$$

Defining $v := \widehat{y} - 1$, we obtain

$$\begin{aligned} g(\widehat{y}) := \frac{\widehat{y}-1}{\log\widehat{y}} &= \frac{v}{\log(1+v)} = \frac{v}{v - v^2/2 + v^3/3 - \dots} \\ &= \frac{1}{1 - v/2 + v^2/3 - \dots} = 1 + \frac{v}{2} + O(v^2)\,. \end{aligned}$$

For small x, we have $y \approx 1$ and

$$g(\widehat{y}) - g(y) \approx \frac{\widehat{y}-y}{2} \approx \frac{e^x\delta}{2} \approx \frac{\delta}{2} \approx g(y)\,\frac{\delta}{2}.$$

By (1.16),

$$\Big|\frac{\widehat{f}-f}{f}\Big| \le 3.5\,u\,. \tag{1.17}$$

While $\widehat{y} - 1$ and $\log\widehat{y}$ are very inaccurate, the quantity $(\widehat{y}-1)/\log\widehat{y}$ is a very accurate approximation of $(y-1)/\log y$ at $y = 1$, because $g(y) := (y-1)/\log y$ varies "slowly" close to 1. ◇

Chapter 2

Numerical integration

This chapter is concerned with algorithms for approximating a definite integral

$$\int_a^b f(x)\,\mathrm{d}x\,, \tag{2.1}$$

for a function $f : [a, b] \to \mathbb{R}$ and specific values of a, b. In Calculus courses it is common to calculate simple integrals like $\int_0^1 e^x\,\mathrm{d}x$ or $\int_0^\pi \cos(x)\,\mathrm{d}x$ using a closed-form primitive for f obtained from a table of integrals or computer algebra systems like Maple, Mathematica, the Symbolic Math Toolbox in MATLAB (which is based on MuPAD) or simply ask WolframAlpha / ChatGPT questions like "What is the primitive of $\exp(-x^2)$?" When trying to compute more complicated expressions like $\int_0^1 e^{x^2}\,\mathrm{d}x$ or $\int_0^\pi \cos(x^2)\,\mathrm{d}x$, one quickly realizes the limitations of such approaches. Indeed, in practice, it is usually *not* possible to find a closed form expression for a primitive. Still, evaluating very complicated definite integrals is a common problem arising in many areas of science and engineering. In these cases, one needs to resort to numerical methods (and understand their limitations).

The goal of this chapter is to introduce and analyze the most popular numerical methods for approximating definite integrals. In the following, unless otherwise stated, we will assume that f is (at least) continuous on $[a, b]$ and hence the integral is well defined.

2.1 A first glimpse at polynomial interpolation

A common principle to derive numerical integration methods is to interpolate f by a polynomial and obtain an approximation by computing the definite integral for this polynomial. We will therefore first take a brief excursion to the world of polynomial interpolation. We will discuss interpolation in more detail in the next chapter.

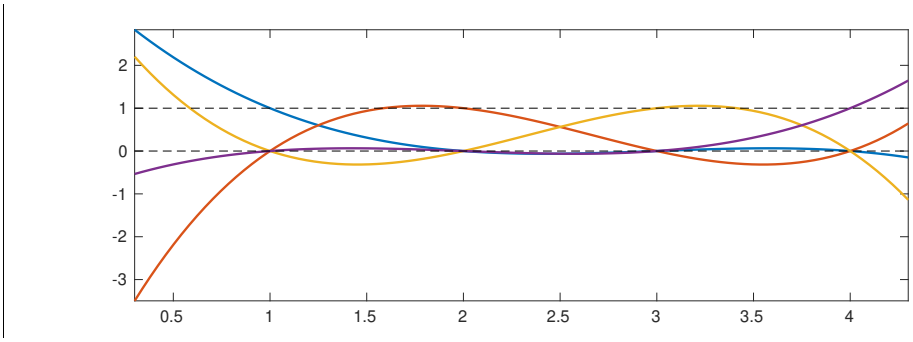


Figure 2.1: The basis of 4 Lagrange polynomials for the nodes $1, 2, 3, 4$.

We let $\mathbb{P}_n$ denote the vector space of real polynomials of degree at most n:

$$\mathbb{P}_n := \operatorname{span}\{x^i \,:\, i = 0, \ldots, n\} = \Big\{ \sum_{i=0}^{n} c_i x^i \,:\, c_i \in \mathbb{R} \Big\}.$$

Given interpolation data (x_j, f_j) with $x_j \in \mathbb{R}$ and $f_j \in \mathbb{R}$ for $j = 0, \ldots, n$, the task of polynomial interpolation is to find a polynomial $p_n \in \mathbb{P}_n$ such that

$$p_n(x_j) = f_j, \qquad j = 0, \ldots, n. \tag{2.2}$$

For $n = 0$ (constant p_0), $n = 1$ (linear p_1), and $n = 2$ (quadratic p_2), it is quite intuitive to see that (2.2) has a unique solution if and only if the **interpolation nodes x_j are pairwise distinct**. To verify this statement for general n, we first need to define a suitable basis for $\mathbb{P}_n$. The monomial basis $\{1, x, x^2, \ldots, x^n\}$ appears to be the canonical choice, but it is actually not well suited for interpolation on the real line because it turns (2.2) into an extremely ill-conditioned linear system. Later on, We will learn more about the impediments of ill-conditioning.

Instead of monomials, we will use the following polynomials whose choice depends on the interpolation nodes.

Definition 2.1 *Given $n+1$ pairwise distinct nodes $x_0, \ldots, x_n \in \mathbb{R}$, the polynomials $\ell_j \in \mathbb{P}_n$ defined by*

$$\ell_j(x) := \prod_{i=0, i\neq j}^{n} \frac{x - x_i}{x_j - x_i}, \qquad j = 0, \ldots, n, \tag{2.3}$$

are called **Lagrange polynomials**.

The following relation is an important property of Lagrange polynomials:

$$\ell_j(x_i) = \delta_{ij} := \begin{cases} 1 & \text{if } i = j, \\ 0 & \text{otherwise.} \end{cases} \tag{2.4}$$

Because $x_0, \ldots, x_n \in \mathbb{R}$ are pairwise distinct, this implies that $\{\ell_0, \ldots, \ell_n\}$ is a linearly independent family and, consequently, it is a basis of $\mathbb{P}_{n+1}$ because $\mathbb{P}_{n+1}$ has dimension $n+1$. Moreover, it also proves the existence and uniqueness of interpolating polynomials.

Theorem 2.2 *Given interpolation data (x_j, f_j), $j = 0, \ldots, n$, with pairwise distinct interpolation nodes $x_0, \ldots, x_n$, the polynomial*

$$p_n(x) := \sum_{j=0}^{n} f_j \ell_j(x) \tag{2.5}$$

is the unique polynomial of degree at most n that satisfies the interpolation conditions (2.2).

Proof. Because $\ell_j \in \mathbb{P}_n$, it follows that the polynomial p_n defined in (2.5) is also in $\mathbb{P}_n$. Moreover, it follows from (2.4) that p_n satisfies the interpolation conditions (2.2).

It remains to show the uniqueness of the solution. For this purpose, let $\tilde{p}_n \in \mathbb{P}_n$ denote another polynomial satisfying (2.2). Then $q_n := p_n - \tilde{p}_n$ is a polynomial of degree at most n. By definition, $q_n(x_j) = 0$ for $j = 0, \ldots, n$, which shows that q_n has $n+1$ has pairwise distinct zeros. According to the fundamental theorem of algebra, the only polynomial of degree at most n having $n+1$ pairwise distinct zeros is the zero polynomial $q_n \equiv 0$. Hence, $\tilde{p}_n \equiv p_n$. ∎

One important application of interpolation is to replace a complicated function f by a polynomial. The following theorem provides an expression for the interpolation error if f is sufficiently smooth.

Theorem 2.3 *Let $x_0 < x_1 < \cdots < x_n$ and let $p_n \in \mathbb{P}_n$ denote the interpolating polynomial satisfying $p_n(x_j) = f(x_j)$, $j = 0, \ldots, n$, for some function*

$$f \in C^{n+1}([x_0, x_n]).$$

Given $x_\star \in [x_0, x_n]$, there exists $\xi \in [x_0, x_n]$ such that

$$E_n[f](x_\star) := f(x_\star) - p_n(x_\star) = \frac{f^{(n+1)}(\xi)}{(n+1)!}\, \omega_{n+1}(x_\star)\,, \tag{2.6}$$

where

$$\omega_{n+1}(x) := (x - x_0)(x - x_1) \cdots (x - x_n) \in \mathbb{P}_{n+1}\,. \tag{2.7}$$

Proof. If $x_\star$ equals one of the interpolation nodes x_j then $E_n[f](x_\star) = E_n[f](x_j) = 0$. Hence, we may assume that $x_\star \neq x_j$. For $x \in I := [x_0, x_n]$ we define the function

$$g(x) := E_n[f](x) - \omega_{n+1}(x)\, E_n[f](x_\star)/\omega_{n+1}(x_\star)\,. \tag{2.8}$$

Because $f \in C^{n+1}([x_0, x_n])$ and $\omega_{n+1} \in \mathbb{P}_{n+1}$ it follows that $g \in C^{n+1}([x_0, x_n])$. Moreover, g has at least $n+2$ distinct zeros in $[x_0, x_n]$ because

$$g(x_j) = E_n[f](x_j) - \omega_{n+1}(x_j)\, E_n[f](x_\star)/\omega_{n+1}(x_\star) = 0, \quad j = 0, \dots, n,$$
$$g(x_\star) = E_n[f](x_\star) - \omega_{n+1}(x_\star)\, E_n[f](x_\star)/\omega_{n+1}(x_\star) = 0\,.$$

By Rolle's theorem, $g' = \frac{dg}{dx}$ has at least $n+1 = n+2-1$ zeros in $[x_0, x_n]$. Continuing this argument, $g^{(i)} = \frac{d^i g}{dx^i}$ has at least $n+2-i$ zeros for $i = 1, \dots, n+1$. Hence, $g^{(n+1)}$ has at least one zero, which we denote by ξ. Because $E_n^{(n+1)}[f](x) = f^{(n+1)}(x)$ and $\omega_{n+1}^{(n+1)}(x) \equiv (n+1)!$, it follows from (2.8) that

$$0 = g^{(n+1)}(\xi) = f^{(n+1)}(\xi) - (n+1)!\, E_n[f](x_\star)/\omega_{n+1}(x_\star)\,.$$

Rearranging this relation yields (2.6). □

Barycentric representation* The straightforward use of the interpolating polynomial $p_n(x)$ in the representation (2.5) would require $O(n^2)$ operations for *each* evaluation of p_n. This complexity can be reduced by *first* computing the following quantities:

$$\lambda_j = \frac{1}{(x_j - x_0)\cdots(x_j - x_{j-1})(x_j - x_{j+1})\cdots(x_j - x_n)}, \quad j = 0, \dots, n.$$

This allows us to write

$$\ell_j(x) = \omega_{n+1}(x)\frac{\lambda_j}{x - x_j},$$

with $\omega_{n+1}(x)$ defined as in (2.7). It follows that

$$p_n(x) = \sum_{j=0}^{n} f_j \ell_j(x) = \omega_{n+1}(x) \sum_{j=0}^{n} \frac{\lambda_j f_j}{x - x_j}. \tag{2.9}$$

Now, only $O(n)$ operations are needed for each operation (after λ_j has been computed). We can simplify this even further. When considering the interpolation of the constant function 1, the relation (2.9) reduces to

$$1 = \omega_{n+1}(x) \sum_{j=0}^{n} \frac{\lambda_j}{x - x_j}.$$

Inserted into (2.9), this yields the **barycentric interpolation formula**

$$p_n(x) = \sum_{j=0}^{n} \frac{\lambda_j f_j}{x - x_j} \Bigg/ \sum_{j=0}^{n} \frac{\lambda_j}{x - x_j}.$$

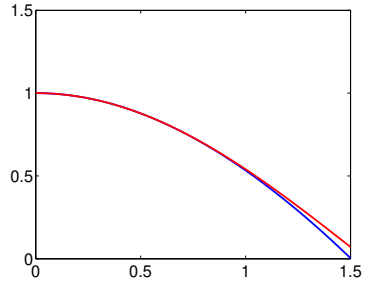
Example Let $x_0 = 0, x_1 = 0.2, x_2 = 0.4, x_3 = 0.6$ and $f_j = \cos(x_j)$. Then the parameters of the barycentric interpolation formula are given by

$$\begin{aligned}
\lambda_0 &= \frac{1}{(x_0 - x_1)(x_0 - x_2)(x_0 - x_3)} = -20.8333, \\
\lambda_1 &= \frac{1}{(x_1 - x_0)(x_1 - x_2)(x_1 - x_3)} = 62.5000, \\
\lambda_2 &= \frac{1}{(x_2 - x_0)(x_2 - x_1)(x_2 - x_3)} = -62.5000, \\
\lambda_3 &= \frac{1}{(x_2 - x_0)(x_2 - x_1)(x_2 - x_3)} = 20.8333.
\end{aligned}$$

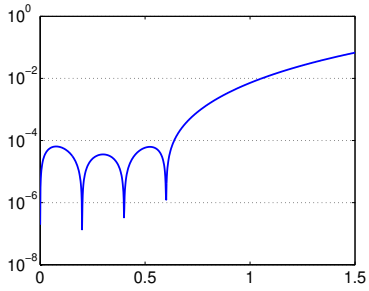
The interpolating polynomial in the Lagrange representation is given by

$$\begin{aligned}
p_3(x) &= y_0\ell_0(x) + y_1\ell_1(x) + y_2\ell_2(x) + y_3\ell_3(x) \\
&= \quad y_0\lambda_0(x - x_1)(x - x_2)(x - x_3) + y_1\lambda_1(x - x_0)(x - x_2)(x - x_3) \\
&\quad + y_2\lambda_2(x - x_0)(x - x_1)(x - x_3) + y_3\lambda_3(x - x_0)(x - x_1)(x - x_2) \\
&= -20.8333(x - x_1)(x - x_2)(x - x_3) + 61.2542(x - x_0)(x - x_2)(x - x_3) \\
&\quad - 57.5663(x - x_0)(x - x_1)(x - x_3) + 17.1945(x - x_0)(x - x_1)(x - x_2).
\end{aligned}$$

cos (red) und $p_3(x)$ (blue)



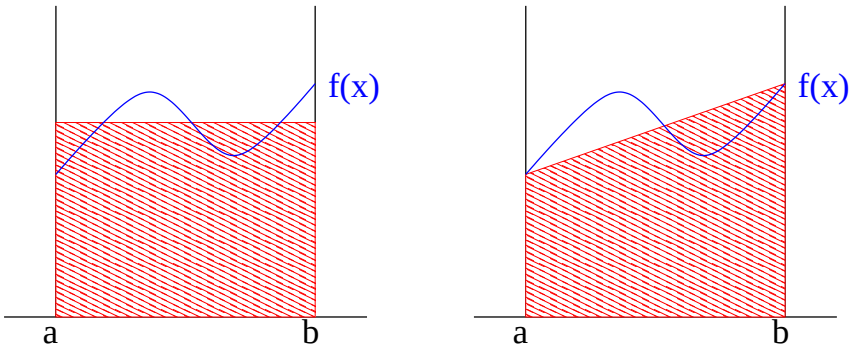
Interpolation error



2.2 Newton-Cotes formulae

We now come back to the idea of approximating the definite integral (2.1) by integrating an interpolating polynomial. If the interpolation nodes are uniformly distributed, the quadrature rule resulting from this approach is called a *Newton-Cotes formula*. In the following, we first treat in detail the three classic cases before discussing the general case.

1. **Midpoint rule**
   The function f is interpolated by a constant polynomial $p_0 \in \mathbb{P}_0$ in the middle of the interval:
   $$p_0(x) \equiv f\Big(\frac{a+b}{2}\Big).$$



$Q^{(0)}_{[a,b]}[f] = f((a+b)/2)(b-a)$ $\quad$ $Q^{(1)}_{[a,b]}[f] = I_{[a,b]}[f] \approx \frac{1}{2}(f(a)+f(b))(b-a)$

Figure 2.2: Illustration of midpoint and trapezoidal rule for approximating the definite integral $\int_a^b f(x)\,\mathrm{d}x$.

The resulting approximation of the integral is given by

$$\int_a^b f(x)\,\mathrm{d}x \approx \int_a^b p_0(x)\,\mathrm{d}x = (b-a)\cdot f\Big(\frac{a+b}{2}\Big) =: Q^{(0)}_{[a,b]}[f],$$

see Figure 2.2 for an illustration.

2. **Trapezoidal rule**
The linear polynomial $p_1 \in \mathbb{P}_1$ interpolating f at the two end-points of the interval $[a,b]$ (that is, $p_1(a) = f(a)$ and $p_1(b) = f(b)$ holds) is given by

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

The resulting approximation of the integral is the *trapezoidal rule* given by

$$\begin{aligned}\int_a^b f(x)\,\mathrm{d}x &\approx \int_a^b p_1(x)\,\mathrm{d}x = f(a)\int_a^b \frac{x-b}{a-b}\,\mathrm{d}x + f(b)\int_a^b \frac{x-a}{b-a}\,\mathrm{d}x\\ &= f(a)(b-a)\int_0^1 y\,\mathrm{d}y + f(b)(b-a)\int_0^1 y\,\mathrm{d}y\\ &= (b-a)\Big(\frac{1}{2}f(a)+\frac{1}{2}f(b)\Big) =: Q^{(1)}_{[a,b]}[f].\end{aligned}$$

3. **Simpson rule**
Let $p_2 \in \mathbb{P}_2$ be the quadratic polynomial interpolating f at the mid- and end-points:

$$p_2(a) = f(a), \quad p_2(x_1) = f(x_1), \quad p_2(b) = f(b), \qquad x_1 = \frac{a+b}{2}.$$

This polynomial is given by

$$p_2(x) = \frac{(x-x_1)(x-b)}{(a-x_1)(a-b)}f(a) + \frac{(x-a)(x-b)}{(x_1-a)(x_1-b)}f(x_1) + \frac{(x-a)(x-x_1)}{(b-a)(b-x_1)}f(b).$$

To compute the definite integral of p_2, we again use the variable substitution $y = \frac{x-a}{b-a}$ to obtain

$$\int_a^b \frac{(x-x_1)(x-b)}{(a-x_1)(a-b)}\,\mathrm{d}x = (b-a)\int_0^1 \frac{(x-1/2)(x-1)}{1/2}\,\mathrm{d}y = \frac{1}{6}(b-a).$$

Similarly, one computes

$$\int_a^b \frac{(x-a)(x-b)}{(x_1-a)(x_1-b)}\,\mathrm{d}x = \frac{4}{6}(b-a),\ \int_a^b \frac{(x-a)(x-x_1)}{(b-a)(b-x_1)}\,\mathrm{d}x = \frac{1}{6}(b-a).$$

In turn, the resulting approximation of the integral is the *Simpson rule* given by

$$\int_a^b f(x)\,\mathrm{d}x \approx (b-a)\Big(\frac{1}{6}f(a) + \frac{4}{6}f\Big(\frac{a+b}{2}\Big) + \frac{1}{6}f(b)\Big) =: Q^{(2)}_{[a,b]}[f].$$

The calculations above can, in fact, be somewhat simplified by first performing the variable substitution

$$\int_a^b f(x)\,\mathrm{d}x = (b-a)\int_1^0 \tilde{f}(y)\,\mathrm{d}y, \quad \tilde{f}(y) := f(a+(b-a)y) \tag{2.10}$$

and then applying interpolation to $\tilde{f}$ on the interval $[0,1]$. This procedure yields the same Simpson rule.

For the general case, we choose $n+1$ interpolation nodes

$$a \le x_0 < x_1 \le \cdots < x_n \le b.$$

We recall the Lagrange representation of the interpolating polynomial:

$$p_n(x) = \sum_{j=0}^n f(x_j)\ell_j(x), \qquad \ell_j(x) = \prod_{\substack{i=0\\ i\neq j}}^n \frac{x-x_i}{x_j-x_i}\,.$$

The definite integral of p_n yields the approximation

$$\begin{aligned} \int_a^b f(x)\,\mathrm{d}x \approx Q^{(n)}_{[a,b]}[f] \quad &:= \quad \int_a^b p_n(x)\,\mathrm{d}x = \int_a^b \sum_{j=0}^n f(x_j)\ell_j(x)\ \mathrm{d}x \\ &= \quad \sum_{j=0}^n f(x_j)\int_a^b \ell_j(x)\ \mathrm{d}x = \sum_{j=0}^n \alpha_j f(x_j), \end{aligned} \tag{2.11}$$

n		ξ_j	$\alpha_j/(b-a)$	Error
0	Midpoint rule	$\frac{1}{2}$	1	$\frac{1}{24}(b-a)^3 f^{(2)}(\xi)$
1	Trap. rule	$0, 1$	$\frac{1}{2}, \frac{1}{2}$	$-\frac{1}{12}(b-a)^3 f^{(2)}(\xi)$
2	Simpson rule	$0, \frac{1}{2}, 1$	$\frac{1}{6}, \frac{4}{6}, \frac{1}{6}$	$-\frac{1}{90}\left(\frac{b-a}{2}\right)^5 f^{(4)}(\xi)$
3	$\frac{3}{8}$ rule	$0, \frac{1}{3}, \frac{2}{3}, 1$	$\frac{1}{8}, \frac{3}{8}, \frac{3}{8}, \frac{1}{8}$	$-\frac{3}{80}\left(\frac{b-a}{3}\right)^5 f^{(4)}(\xi)$
4	Milne rule	$0, \frac{1}{4}, \frac{1}{2}, \frac{3}{4}, 1$	$\frac{7}{90}, \frac{32}{90}, \frac{12}{90}, \frac{32}{90}, \frac{7}{90}$	$-\frac{8}{945}\left(\frac{b-a}{4}\right)^7 f^{(6)}(\xi)$

Table 2.1: Newton-Cotes formulae ($x_j = a + \xi_j(b-a)$).

where the scalars $\alpha_j := \int_a^b \ell_j(x)\,\mathrm{d}x$ are called the **weights** of the quadrature rule $Q^{(n)}_{[a,b]}[f]$. Using the substitution (2.10), it can be seen that $\alpha_j/(b-a)$ is a constant not depending on a, b or f. As mentioned in the beginning, if the points are uniformly distributed,

$$x_j = a + jh, \quad j = 0, \ldots, n, \quad h = \frac{b-a}{n},$$

for $n \geq 1$ then $Q^{(n)}_{[a,b]}[f]$ is called a (closed) Newton-Cotes formula. Table 2.1 shows the quadrature points $x_j = a + \xi_j(b-a)$ und weights for $n \leq 4$. The fourth column contains the quadrature error, which will be discussed in the next section.

2.3 Order and error analysis

The order of a quadrature rule is determined by its ability to integrate polynomials up to a certain degree exactly.

Definition 2.4 *A quadrature rule $Q_{[a,b]}$ has* **order $s+1$**, *if*

$$\int_a^b p_s(x)\,\mathrm{d}x = Q_{[a,b]}[p_s], \qquad \forall p_s \in \mathbb{P}_s.$$

Every commonly used quadrature rule $Q_{[a,b]}$ is a linear operator and, hence, Definition 2.4 is equivalent to verifying that

$$Q_{[0,1]}[1] = 1,\ Q_{[0,1]}[x] = \frac{1}{2},\ \ldots,\ Q_{[0,1]}[x^s] = \frac{1}{s+1}.$$

By construction, the Newton-Cotes formula $Q^{(n)}_{[a,b]}[f]$ has order *at least* $n+1$ because polynomials of degree up to n are interpolated and, hence, integrated exactly. It is easy to verify that the mid-point rule has, in fact, one order higher, order 2. The

trapezoidal rule also has order 2 and the Simpson rule has order 4, again one order higher.

The order has a pronounced influence on the quadrature error $\int_a^b f\,\mathrm{d}x - Q_{[a,b]}[f]$ if f is sufficiently smooth and the interval $[a,b]$ is small. To see this intuitively, consider the (truncated) Taylor expansion of f at a

$$f(x) = t_s(x) + \frac{f^{(s+1)}(\xi_x)}{(s+1)!}(x-a)^{s+1}, \qquad t_s \in \Pi_s.$$

Because t_s is integrated exactly, the quadrature error is determined by the second term. The absolute value of the second term is $O(h^{s+1})$ for $h := b - a \to 0$. Integrating it over an interval of length h gives an integration error of $O(h^{s+2})$. Finer estimates can be obtained by applying Theorem 2.3. The following theorem and proof establish the quadrature error for $n = 0$ and $n = 1$; the other entries in Table 2.1 can be established in an analogous fashion.

Theorem 2.5 *Let $f \in C^2[a,b]$.*

i) *There is $\xi \in [a,b]$ such that the error of the midpoint rule satisfies*

$$\int_a^b f(x)\,\mathrm{d}x - (b-a)\, f\Big(\frac{a+b}{2}\Big) = \frac{(b-a)^3}{24}\, f''(\xi).$$

ii) *There is $\xi \in [a,b]$ such that the error of the trapezoidal rule satisfies*

$$\int_a^b f(x)\,\mathrm{d}x - \frac{b-a}{2}\Big[f(a)+f(b)\Big] = -\frac{(b-a)^3}{12}\, f''(\xi).$$

Proof. ii) We first prove the second part, because it follows directly from Theorem 2.3:

$$f(x) - p_1(x) = \frac{f''(\xi_x)}{2}(x-a)(x-b).$$

for some $\xi_x \in [a,b]$ (depending on x). Integrating both sides yields

$$\int_a^b f(x)\,\mathrm{d}x - Q^{(1)}_{[a,b]}[f] = \int_a^b \frac{f''(\xi_x)}{2}(x-a)(x-b)\,\mathrm{d}x. \tag{2.12}$$

Let

$$c = \frac{\int_a^b \frac{f''(\xi_x)}{2}(x-a)(x-b)\;\mathrm{d}x}{\frac{1}{2}\int_a^b (x-a)(x-b)\;\mathrm{d}x} = \frac{\int_a^b f''(\xi_x)(x-a)(x-b)\;\mathrm{d}x}{-\frac{1}{6}(b-a)^3}.$$

Because $(x-a)(x-b) \le 0$ for every x in $[a,b]$, it follows that

$$\min_{x\in[a,b]} f''(x) \le c \le \max_{x\in[a,b]} f''(x).$$

Since f'' is continuous, the intermediate value theorem shows that there exists $\xi \in [a,b]$ (not depending on x) such that $c = f''(\xi)$. Inserting this into (2.12) completes the proof of part ii).

i) Let $x_0 = \frac{a+b}{2}$. Using Taylor expansion at x_0, it follows that

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

for some ξ_x between x_0 and x. In turn, the quadrature error satisfies

$$\int_a^b f(x)\,\mathrm{d}x - Q^{(0)}_{[a,b]}[f] = f'(x_0)\int_a^b (x - x_0)\,\mathrm{d}x + \frac{1}{2}\int_a^b f''(\xi_x)(x - x_0)^2\,\mathrm{d}x.$$

Note that the first term vanishes because $\int_a^b (x - x_0)\,\mathrm{d}x = 0$. Because $(x - x_0)^2 \geq 0$, we can apply the intermediate value theorem as in part ii) to conclude that there exists $\xi \in [a, b]$ such that

$$\frac{1}{2}\int_a^b f''(\xi_x)(x - x_0)^2\,\mathrm{d}x = \frac{1}{2}f''(\xi)\int_a^b (x - x_0)^2\,\mathrm{d}x = \frac{1}{24}f''(\xi)(b - a)^3.$$

□

Table 2.1 might suggest that it is a good idea to go even further and use very large for n (on relatively small intervals). However, for $n = 8$ and larger some of the weights become negative, which leads to numerical cancellation and limits the usefulness of such high-order Newton-Cotes formulae.

2.4 Composite Newton-Cotes formulae

As mentioned above, the gains in accuracy by increasing the order of Newton-Cotes formulae are limited due to the influence of roundoff error. A more effective approach to increasing accuracy is to partition the interval $[a, b]$ in N subintervals and approximate $I_{[a,b]}[f]$ by applying a quadrature rule to each subinterval.

Notation: *From now on, x_i, $i = 0, \ldots, N$, denote the boundaries of the subintervals, N denotes the number of subintervals, and n denotes (as before) the maximum polynomial degree used in the integration of subintervals.*

A uniform partition of $[a, b]$ into N subintervals corresponds to

$$x_j = a + jh, \qquad j = 0, \ldots, N, \qquad h = \frac{b - a}{N}.$$

Applying the quadrature rule $Q^{(n)}_{[x_i,x_{i+1}]}[f]$ to each subinterval und summing the obtained values yields the corresponding **composite** quadrature rule:

$$Q^{(n)}_h[f] := \sum_{i=0}^{N-1} Q^{(n)}_{[x_i,x_{i+1}]}[f]. \tag{2.13}$$

For $n = 1$ we obtain the **composite trapezoidal rule**:

$$Q^{(1)}_h[f] = \sum_{i=0}^{N-1} \frac{x_{i+1} - x_i}{2}\,[f(x_i) + f(x_{i+1})] = \frac{h}{2}\left[f(a) + 2\sum_{i=1}^{N-1} f(x_i) + f(b)\right].$$

For $n = 1$ we obtain the **composite Simpson rule**:

$$\begin{aligned} Q_h^{(2)}[f] &= \sum_{i=0}^{N-1} \frac{x_{i+1} - x_i}{6} \left[f(x_i) + 4f\Big(\frac{x_i + x_{i+1}}{2}\Big) + f(x_{i+1}) \right] \\ &= \frac{h}{6} \left[f(a) + 2 \sum_{i=1}^{N-1} f(x_i) + 4 \sum_{i=0}^{N-1} f\Big(\frac{x_i + x_{i+1}}{2}\Big) + f(b) \right]. \end{aligned}$$

Theorem 2.6 *For the composite trapezoidal and Simpson rules it holds that*

$$\big| I_{[a,b]}[f] - Q_h^{(1)}[f] \big| \leq \frac{h^2}{12}(b-a) \max_{x \in [a,b]} |f''(x)|, \quad f \in C^2[a,b],$$

$$\big| I_{[a,b]}[f] - Q_h^{(2)}[f] \big| \leq \frac{h^4}{2880}(b-a) \max_{x \in [a,b]} |f^{(4)}(x)|, \quad f \in C^4[a,b].$$

Proof. Applying Theorem 2.5 to each subinterval yields

$$\begin{aligned} \big| I_{[a,b]}[f] - Q_h^{(1)}[f] \big| &= \Big| \sum_{i=0}^{N-1} \frac{(x_{i+1} - x_i)^3}{12} f''(\xi_i) \Big| \\ &\leq \sum_{i=0}^{N-1} \frac{h^3}{12} |f''(\xi_i)| \leq \frac{h^2}{12}(b-a) \max_{x \in [a,b]} |f''(x)|. \end{aligned}$$

The proof for the Simpson rule proceeds analogously. ∎

Example 2.7 Figure 2.3 shows the implementations and the errors obtained when applying the composite trapezoidal and Simpson rules to approximating

$$\int_0^\pi \sin(x) \, \mathrm{d}x = 2. \tag{2.14}$$

As expected, the error of the (composite) Simpson rule converges much faster to zero than the error of the trapezoidal rule. The asymptotic behavior is clearly $O(h^4)$ and $O(h^2)$, as predicted by Theorem 2.6. As a consequence, the Simpson rule needs much fewer function evaluations to attain the same accuracy.

If f is not smooth, the advantage of the Simpson rule becomes less important. To see this, consider

$$\int_0^1 \sqrt{x} \, \mathrm{d}x = \frac{2}{3}. \tag{2.15}$$

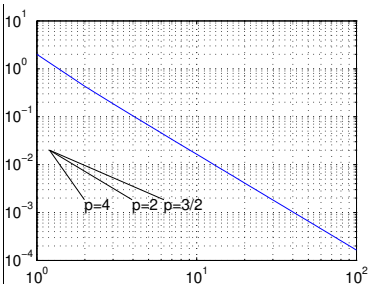
Figure 2.4 shows the error of the composite trapezoidal and Simpson rules. For both rules, the asymptotic behavior of the error is $O(h^p)$ with $p = 3/2$ (but with a smaller constant for the Simpson rule). ⋄

PYTHON
```
import numpy as np
import matplotlib.pyplot as plt

def trapez(fun, a, b, n):
  x = np.linspace(a,b,n+1)
  vecfun = np.vectorize(fun)
  f = vecfun(x)
  f[0] = f[0]/2
  f[-1] = f[-1]/2
  T = (b-a) * sum(f) / n
  return T

nn = 100; err = np.zeros(nn)
for n in range(nn):
  err[n] = np.abs(2 -
  trapez(np.sin, 0, np.pi, n + 1))
plt.loglog(range(1,nn+1), err)
```

PYTHON
```
def simpson(fun,a,b,n):
  x = np.linspace(a,b,2*n+1)
  vecfun = np.vectorize(fun)
  f = vecfun(x)
  end = len(f) - 1
  f[1:end: 2] = 4*f[1:end:2]
  f[2:end-1:2] = 2*f[2:end-1:2]
  T = (b-a) * sum(f) / n / 6
  return T
```



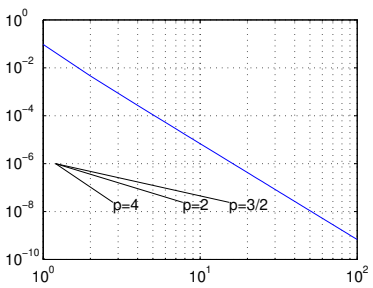


Figure 2.3: Errors of composite Simpson rule (left) and trapezoidal rule (right) vs. $N = O(h^{-1})$ when approximating smooth integral (2.14).

The **composite midpoint rule**,

$$Q_h^{(0)}[f] = \sum_{i=0}^{N-1} (x_{i+1} - x_i) f\Big(\frac{x_i + x_{i+1}}{2}\Big)$$

is of interest for functions with singularities at the interval boundaries; see Section 2.6.2 below.

2.5 Gauss formulae

In the Newton-Cotes formulae, we have chosen the interpolation points to be uniformly distributed. In this section, we modify these points in order to obtain a quadrature rule of higher order.

Remark 2.8 *There is no quadrature rule* (2.11) *with* $n+1$ *points* $x_0, \ldots, x_n$ *of order higher than* $2n+2$. *If* $Q^{(n)}[p]$ *was such a quadrature rule it would be exact*

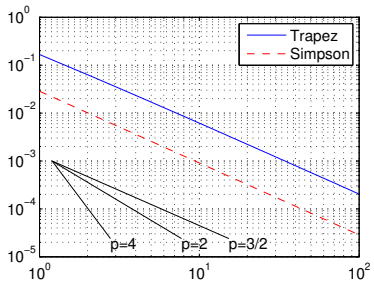


Figure 2.4: Errors of composite Simpson and trapezoidal rules vs. $N = O(h^{-1})$ when approximating non-smooth integral (2.15).

for the polynomial

$$p(x) = \prod_{i=0}^{n} (x - x_i)^2 \in \mathbb{P}_{2n+2}\,,$$

which leads to the contradiction

$$0 < \int_a^b p(x)\,dx = Q^{(n)}[p] = 0\,.$$

In the following, we construct quadrature rules that attain the highest possible order according to Remark 2.8, the so called **Gauss formulae**. As an additional benefit, these formulae always have nonnegative weights and therefore avoid the numerical problems associated with high orders for the Newton-Cotes formulae.

To construct Gauss formulae, we have to choose the quadrature points $x_0, \ldots, x_n$ such that $Q^{(n)}$ exactly integrates all polynomials of degree at most $2n+1$. Because of linearity, it suffices to verify this property for any basis $\mathbb{P}_{2n+1}$. When choosing the monomial basis this leads to the $2n+2$ (nonlinear) equations

$$\int_{-1}^{1} x^k\,\mathrm{d}x = \alpha_0 x_0^k + \cdots + \alpha_n x_n^k, \quad k = 0, \ldots, 2n+1. \tag{2.16}$$

Here and in the following, we assume for simplicity that $[a, b] = [-1, 1]$. The solution of the nonlinear equations (2.16) by hand or computer algebra becomes quickly infeasible for larger n. There is a much simpler and more elegant approach, which will be discussed in the following. Polynomial division provides the starting point of this approach.

Theorem 2.9 (Polynomial division) *Let $p \in \mathbb{P}_{2n+1}$ and $q \in \mathbb{P}_{n+1}$. There exist unique polynomials $h \in \mathbb{P}_n$ and $r \in \mathbb{P}_n$ such that $p = hq + r$.*

For integrating a polynomial $p \in \mathbb{P}_{2n+1}$ exactly, it must hold that

$$\begin{aligned} 0 &= \int_{-1}^{1} p(x)\,\mathrm{d}x - \sum_{j=0}^{n} \alpha_j p(x_j) \\ &= \int_{-1}^{1} h(x)q(x)\,\mathrm{d}x - \sum_{j=0}^{n} \alpha_j h(x_j)q(x_j) + \left(\int_{-1}^{1} r(x)\,\mathrm{d}x - \sum_{j=0}^{n} \alpha_j r(x_j) \right) \end{aligned} \tag{2.17}$$

Because $r \in \mathbb{P}_n$, the third term (in brackets) is always zero when using an interpolating quadrature rule with $n+1$ points. For addressing the first two terms, we will make a clever choice of q via Legendre polynomials.

Definition 2.10 *The* **Legendre polynomial** *(of order $n+1$) is the polynomial $q_{n+1} \in \mathbb{P}_{n+1}$ satisfying*

$$\int_{-1}^{1} q_{n+1}(x)h(x)\,\mathrm{d}x = 0 \ \forall h \in \mathbb{P}_n, \quad q_{n+1}(1) = 1. \tag{2.18}$$

Defining the L_2 inner product

$$\langle p, q \rangle = \int_{-1}^{1} p(x)q(x)\,\mathrm{d}x$$

on the vector space $\mathbb{P}_{n+1}$, the first condition in (2.18) states that q_{n+1} is orthogonal to the subspace $\mathbb{P}_n$. In turn, this shows that Legendre polynomials can be constructed by applying the Gram-Schmidt procedure to the monomial basis 1, x, …, x^{n+1}. The next result provides a more direct characterization of Legendre polynomials.

Theorem 2.11 *The polynomial q_k defined by*

$$q_k(x) = c_k \frac{\mathrm{d}^k}{\mathrm{d}x^k} \left[(x^2-1)^k \right], \qquad c_k := \frac{1}{2^k k!}, \tag{2.19}$$

is the kth Legendre polynomial.

Proof. First, we note that $q_k \in \mathbb{P}_k$, since it is obtained by computing k derivatives of a polynomial of degree $2k$. Also, it is straightforward to see that $q_k(1) = 1$. To show that q_k is orthogonal to every $g \in \mathbb{P}_{k-1}$ we proceed via integration by parts:

$$\begin{aligned} c_k \int_{-1}^{1} \frac{d^k}{dx^k} \left[(x^2-1)^k \right] g(x)\ \mathrm{d}x = &- c_k \int_{-1}^{1} \frac{d^{k-1}}{dx^{k-1}} \left[(x^2-1)^k \right] \frac{d}{dx} g(x)\ \mathrm{d}x \\ &+ c_k \left[\frac{d^{k-1}}{dx^{k-1}} (x^2-1)^k g(x) \right]_{-1}^{1} \end{aligned}$$

Note that the second term vanishes because $(x^2-1)^k$ has a zero of multiplicity k at ± 1. To treat the first term, we again integrate by parts:

$$-c_k \int_{-1}^{1} \frac{d^{k-1}}{dx^{k-1}} \left[(x^2-1)^k\right] \frac{d}{dx} g(x)\ \mathrm{d}x = c_k \int_{-1}^{1} \frac{d^{k-2}}{dx^{k-2}} \left[(x^2-1)^k\right] \frac{d^2}{dx^2} g(x)\ \mathrm{d}x - c_k \left[\frac{d^{k-2}}{dx^{k-2}} (x^2-1)^k \frac{d}{dx} g(x)\right]_{-1}^{1}.$$

Once again, the second term vanishes. Continuing this procedure and integrating by parts k times, we obtain

$$c_k \int_{-1}^{1} \frac{d^k}{dx^k} \left[(x^2-1)\right]^k g(x)\ \mathrm{d}x = (-1)^k c_k \int_{-1}^{1} \left(x^2-1\right) \frac{d^k}{dx^k} g(x)\ \mathrm{d}x.$$

Since g has degree at most $\mathbb{P}_{k-1}$ its kth derivative vanishes and, hence,

$$c_k \int_{-1}^{1} \frac{d^k}{dx^k} \left[(x^2-1)\right]^k g(x)\ \mathrm{d}x = 0.$$

□

Theorem 2.11 implies a recurrence relation, which is convenient for computing Legendre polynomials.

Theorem 2.12 *The Legendre polynomials $q_0, q_1, \ldots$ satisfy the three-term recurrence relation*

$$q_{n+1}(x) = \frac{2n+1}{n+1} x q_n(x) - \frac{n}{n+1} q_{n-1}(x), \quad q_0(x) = 1, \quad q_1(x) = x.$$

Proof. EFY. □

According to Theorem 2.12, the first five Legendre polynomials are given by

$$\begin{aligned} q_0(x) &= 1 \\ q_1(x) &= x \\ q_2(x) &= \frac{1}{2}(3x^2-1) \\ q_3(x) &= \frac{1}{2}(5x^3-3x) \\ q_4(x) &= \frac{1}{8}(35x^4-30x^2+3) \\ q_5(x) &= \frac{1}{8}(63x^5-70x^3+15x). \end{aligned}$$

Choosing $q = q_{n+1}$, the first term in (2.17) vanishes. Choosing $x_0, \ldots, x_{n+1}$ as zeros of q_{n+1} then the second term vanishes as well.

Theorem 2.13 *The Legendre polynomial q_{n+1} has $n+1$ simple zeros in $(-1,1)$.*

Proof. Let us define the set

$$N := \{\lambda \in (-1,1) : \lambda \text{ is a zero of } \textit{odd} \text{ multiplicity of } q_{n+1}\}$$

and

$$\begin{aligned} h(x) &:= 1 && \text{for } N = \emptyset, \quad \text{and} \\ h(x) &:= \prod_{i=1}^{m} (x - \lambda_i) && \text{for } N = \{\lambda_1, \dots, \lambda_m\}. \end{aligned}$$

Then $q_{n+1} \cdot h \in \mathbb{P}_{n+m+1}$ is real and all its roots in $(-1,1)$ have even order. In particular, it has *no* change of sign in $(-1,1)$ and therefore

$$(q_{n+1}, h) = \int_{-1}^{1} q_{n+1}(x)\, h(x)\, \mathrm{d}x \neq 0\,.$$

If $m \leq n$ this contradicts $q_{n+1} \perp \mathbb{P}_n$. In turn, $m > n$ and q_{n+1} has at least $n+1$ zeros in $(-1,1)$. By the fundamental lemma of algebra, q_{n+1} has *exactly* $n+1$ zeros in $(-1,1)$. ∎

The following theorem summarizes our observations.

Theorem 2.14 *Let $x_0, \dots, x_n \in (-1,1)$ be the zeros of the Legendre polynomials q_{n+1} let $\ell_0, \dots, \ell_n$ be the corresponding Lagrange polynomials. Choosing $\alpha_j = \int_{-1}^{1} \ell_j(x)\,\mathrm{d}x$ the* **Gauss formula**

$$Q^{(n)}[f] = \alpha_0 f(x_0) + \cdots + \alpha_n f(x_n)$$

has order $2n+2$.

For $n = 1$ and $n = 2$ we obtain the quadrature rules

$$\begin{aligned} Q^{(1)}[f] &= f(-\sqrt{1/3}) + f(\sqrt{1/3}), \\ Q^{(2)}[f] &= \frac{1}{9}\big\{5f(-\sqrt{3/5}) + 8f(0) + 5f(\sqrt{3/5})\big\}. \end{aligned}$$

Because of

$$0 < \int_{-1}^{1} \ell_i(x)^2\, dx = \sum_{j=0}^{n} \alpha_j \underbrace{\ell_i(x_j)^2}_{\delta_{ij}} = \alpha_i,$$

it follows that the weights are always positive and we avoid the numerical instability associated with high-order Newton-Cotes formulae. For general n, one can obtain the weights from the following linear system of equations:

$$\sum_{j=0}^{n} \alpha_j x_j^i = \int_{-1}^{1} x^i\, \mathrm{d}x = \frac{1}{i+1}(1 - (-1)^{i+1}), \quad i = 0, \dots, n.$$

The computation of the zeros of q_{n+1} is more difficult; the following result yields a simple implementation.[8]

Theorem 2.15 (Golub/Welsh) *The zeros $x_0, \ldots, x_n$ of q_{n+1} are the eigenvalues of the matrix*

$$J = \begin{pmatrix} 0 & b_1 & & \\ b_1 & 0 & \ddots & \\ & \ddots & \ddots & b_n \\ & & b_n & 0 \end{pmatrix},$$

where

$$b_j = \frac{j}{\sqrt{4j^2 - 1}}.$$

PYTHON

```
import numpy as np
def gaussQuad(n):
    b = np.arange(1,n+1)
    b = b / np.sqrt(4*b*b-1)
    J = np.diag(b,-1) + np.diag(b,1)
    x, ev = np.linalg.eigh(J) # eigh stands for symmetric EVP
    a = np.array(2*(ev[0,:]*ev[0,:]))
    return x,a
```

The following theorem establishes an expression for the error of the Gauss formulae, which highlights (once more) their advantages compared to Newton-Cotes formulae.

Theorem 2.16 (Error of Gauss quadrature) *For $f \in C^{2n+2}[-1,1]$ there exists $\xi \in (-1,1)$ such that*

$$\begin{aligned} R^{(n)}[f] := Q^{(n)}[f] - \int_{-1}^{1} f(x)\ \mathrm{d}x &= \frac{f^{(2n+2)}(\xi)}{(2n+2)!} \int_{-1}^{1} \prod_{j=0}^{n} (x - x_j)^2\ \mathrm{d}x \\ &= \frac{2^{2n+3}[(n+1)!]^4}{(2n+3)[(2n+2)!]^3}\ f^{(2n+2)}(\xi). \end{aligned}$$

Example 2.17 For the trapezoidal rule, Theorem 2.5 yields an error of the form $2/3\, f''(\xi)$ on the interval $[a,b] = [-1,1]$. For the Gauss quadrature for $n = 1$ (which requires the same number of function evaluations), Theorem 2.16 establishes an error of the form $1/135\, f^{(4)}(\xi)$. ◇

By a linear transformation, one obtains Gauss quadrature rules on an arbitrary

[8] See `https://pi.math.cornell.edu/~ajt/papers/QuadratureEssay.pdf` for a nice account of the history of methods for computing Gauss quadrature nodes/weights.

interval $[a, b]$. For $n = 1$ and $n = 2$ one obtains

$$Q^{(1)}[f] = \frac{b-a}{2}\left[f\big(c-\tilde{h}\sqrt{1/3}\big) + f\big(c+\tilde{h}\sqrt{1/3}\big)\right],$$

$$Q^{(2)}[f] = \frac{b-a}{18}\left[5f\big(c-\tilde{h}\sqrt{3/5}\big) + 8f\big(c\big) + 5f\big(c+\tilde{h}\sqrt{3/5}\big)\right],$$

with $c = \frac{b+a}{2}$ and $\tilde{h} = \frac{b-a}{2}$. Setting $x_j = a+jh$ for $j = 0, \ldots, N$ and $h = (b-a)/N$), the corresponding composite rules are given by

$$Q_h^{(1)}[f] = \frac{h}{2}\sum_{j=0}^{N-1}\left[f(x_j + h') + f(x_{j+1} - h')\right]$$

with $h' = (\frac{1}{2} - \frac{1}{2\sqrt{3}})\, h \sim 0.2113249\, h$, and

$$Q_h^{(2)}[f] = \frac{h}{18}\sum_{j=0}^{N-1}\left[5f(x_j + h') + 8f\big(x_j + \tfrac{1}{2}\, h\big) + 5f(x_{j+1} - h')\right]$$

with $h' = (\frac{1}{2} - \frac{1}{2}\sqrt{\frac{3}{5}})\, h \sim 0.1127012\, h$.

2.6 Miscellaneous*

2.6.1 Periodic functions

The composite trapezoidal rule has excellent convergence properties for (smooth) periodic functions. For example, consider the function

$$f(x) = \frac{1}{\sqrt{1 - a\sin(x-1)}}, \tag{2.20}$$

which is periodic on the interval $[0, 2\pi]$. Für $a = 1$, the function has a singularity at $2\pi/4 + 1 \approx 2.57$.

In the left part of Figure 2.5, the interval is chosen such that it matches a period of the function. Note that the x axis is *not* scaled logarithmically; the quadrature error converges *exponentially fast* to 0 as N increases. The speed of convergence clearly depends on a; as a gets very close 1 one suffers from the non-smoothness of the function for $a = 1$. In the right part of Figure 2.5 the interval does not match the period of the function. Note that the x axis is scaled logarithmically. In this situation, the quadrature error converges much more slowly and in accordance with the result of Theorem 2.6. On the other hand, choosing a close to 1 has a less dramatic impact on the convergence speed.

The fast convergence of the composite trapezoidal rule for a periodic function is connected to favorable properties of the Fourier expansion, which will be discussed later on.

For a periodic function, the composite trapezoidal rule is the method of choice. It would do more harm than good to choose higher order quadrature rules.

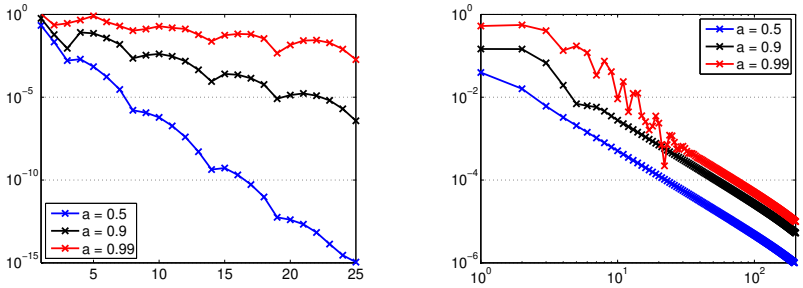


Figure 2.5: Quadrature error vs. N for composite trapezoidal rule applied to integrating f from (2.20) for $a \in \{0.5, 0.9, 0.99\}$ on two different intervals.

2.6.2 Singular integrals

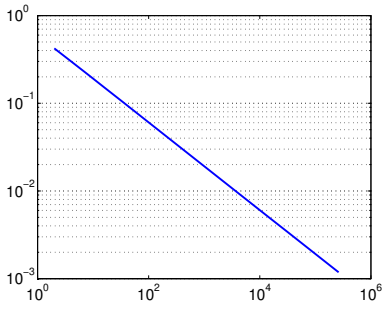
Functions with singularities (leading to singular/improper integrals) can also be integrated numerically, but some care is needed in the choice of quadrature rule. For example, the composite midpoint rule converges for

$$\int_0^1 \frac{1}{\sqrt{x}}\,\mathrm{d}x = 2.$$

PYTHON

```
import numpy as np
import matplotlib.pyplot as plt
def sing(x):
    return 1/np.sqrt(x)
def midpoint(fun,a,b,n):
    h = (b-a)/n;
    x = np.arange(a+h/2,b-h/2+h,h)
    fun_vec = np.vectorize(fun)
    f = fun_vec(x)
    return h * sum(f)
nn = 2**np.arange(1,19,1)
err = []
for n in nn:
    err.append(abs(2 -
    midpoint(sing,0,1,n) ))
plt.loglog(nn,err)
```

The following figure shows the observed error vs. N:



The observed convergence order $1/2$ is not very satisfying; halving the error requires to increase the number of points by a factor four. More effective approaches to singularities are variable transformation techniques or adaptive quadrature.

2.6.3 Two-dimensional integrals

In practice, one is often interested in integrals in 2D, 3D or, more generally, on domains in $\mathbb{R}^d$. By domain decomposition and transformation one typically reduces such problems to standard domains like the unit cube. To illustrate the development of quadrature rules for such standard domains we discuss the unit square and the unit triangle.

Unit square. Consider the integral

$$\int_0^1 \int_0^1 f(x,y)\,\mathrm{d}x\,\mathrm{d}y. \tag{2.21}$$

Let

$$Q_m[g] = \sum_{i=0}^{m} \alpha_i g(x_i)$$

be a quadrature rule for $\int_0^1 g(x)\ \mathrm{d}x$. Denoting

$$F(y) = \int_0^1 f(x,y)\ \mathrm{d}x,$$

we obtain from the application of Q_m to the x and y variables the following product quadrature rule $Q^{(m\times m)}_{[0,1]\times[0,1]}[f]$:

$$\begin{aligned}
\int_0^1 \int_0^1 f(x,y)\ \mathrm{d}x\ \mathrm{d}y &= \int_0^1 F(y)\ \mathrm{d}y \approx \sum_{j=0}^{m} \alpha_j F(x_j) \\
&= \sum_{j=0}^{m} \alpha_j \int_0^1 f(x,x_j)\ \mathrm{d}x \approx \sum_{j=0}^{m} \alpha_j \sum_{i=0}^{m} \alpha_i f(x_i,x_j) \\
&= \sum_{i,j=0}^{m} \alpha_i \alpha_j f(x_i,x_j) =: Q_{m\times m}[f].
\end{aligned}$$

For the hypercube in $\mathbb{R}^d$ the approach above would result in m^d terms. Hence, the cost grows very quickly with d. This so called *curse of dimensionality* can sometimes be countered with *Monte Carlo methods* or *sparse grids*.

Unit triangle. The approach for the unit square has no meaningful extension to triangles. Instead one aims at finding quadrature rule which exactly integrate $x^{k_1}y^{k_2}$, $k_1+k_2 \le M$ for a certain integer M. Typical examples for the unit triangle $\{(x,y) : x+y \le 1\}$:

1. $Q[f] = \frac{1}{2} f(1/3,1/3)$,
2. $Q[f] = \frac{1}{6}\big[f(0,0)+f(1,0)+f(0,1)\big]$,
3. $Q[f] = \frac{1}{6}\big[f(1/2,0)+f(0,1/2)+f(1/2,1/2)\big]$,

4. $Q[f] = \frac{1}{6}\big[f(1/6, 1/6) + f(2/3, 1/6) + f(1/6, 2/3)\big]$.

One verifies that 1 and 2 exactly integrate $1, x, y$ ($M = 1$), while 3 and 4 exactly integrate $1, x, y, xy, x^2, y^2$ ($M = 2$).

Chapter 3

Polynomial interpolation

We are getting back to the topic of (polynomial) interpolation from Section 2.1. First, a short summary of what we learned in that section. Given interpolation data (x_j, y_j) with $x_j \in \mathbb{R}$ and $y_j \in \mathbb{R}$ for $j = 0, \ldots, n$, we have shown that there is a unique polynomial p_n of degree at most n such that

$$p_n(x_j) = y_j, \quad j = 0, \ldots, n,$$

if (and only if) the interpolation nodes x_j are pairwise distinct. The Lagrange representation of p_n is given by

$$p_n(x) = \sum_{j=0}^{n} y_j \ell_j(x), \quad \ell_j(x) := \prod_{i=0, i\neq j}^{n} \frac{x - x_i}{x_j - x_i}. \tag{3.1}$$

In the following, we let $\|\cdot\|_\infty$ denote the uniform (or L^∞) norm of a function $g : [a, b] \to \infty$, that is, $\|g\|_\infty := \sup_{x\in[a,b]} |g(x)|$. Then Theorem 2.3 implies the error bound

$$\|f - p_n\|_\infty \leq \frac{1}{(n+1)!} \|\omega_{n+1}\|_\infty \|f^{(n+1)}\|_\infty \tag{3.2}$$

when $y_j = f(x_j)$ for an $n + 1$ times continuously differentiable function f. Apart from properties of the function f, the norm of $\omega_{n+1}(x) = (x-x_0)(x-x_1)\cdots(x-x_n)$ on $[a, b]$ also enters this error bound.

By the Weierstrass approximation theorem, for any *continuous* function f there is a sequence of polynomials $\tilde{p}_n$ such that $\|f-\tilde{p}_n\|_\infty$ converges to zero as $n \to \infty$. This, however, does *not* necessarily hold for the polynomials p_n defined in (3.1), even when assuming that f is infinitely often differentiable. To see this, consider the function $f(x) = 1/\left(1 + 25x^2\right)$ and equidistant nodes on $[-1, 1]$, that is, $x_j = -1 + 2j/n$ for $j = 0, \ldots, n$. As n increases, the accuracy of the polynomial interpolation deteriorates at the neighborhoods of the end points of the interval; see Figure 3.1. In particular, the $\|f - p_n\|_\infty$ *grows* (quite quickly) instead of converging to zero. This problem has been observed for the first time in 1901 by Runge and, for this reason, is called *Runge phenomenon*. Both, the growth of the derivatives of f and

the strong oscillations of ω_{n+1} near the end points of the interval contribute to this phenomenon.

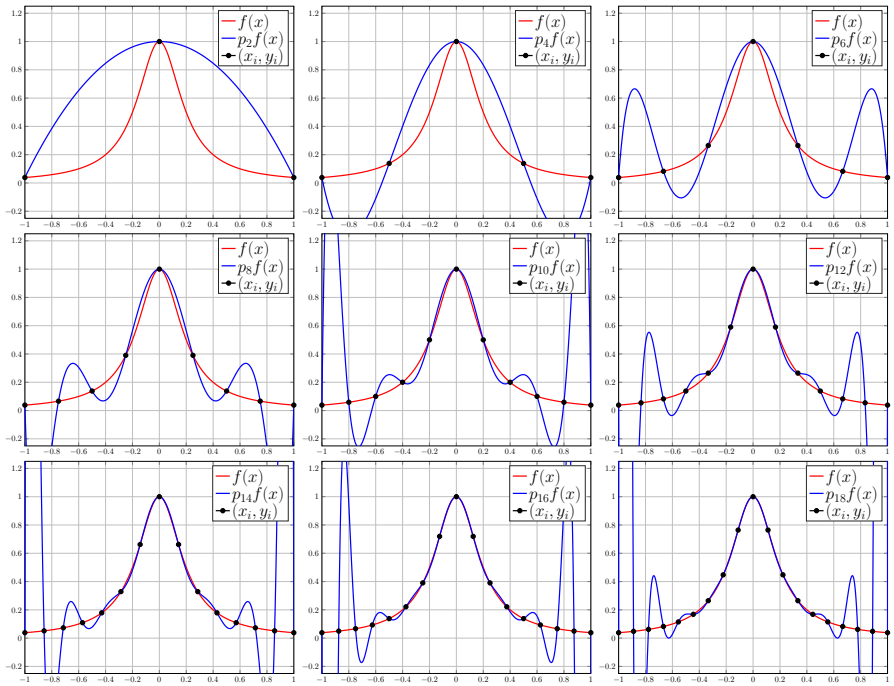


Figure 3.1: The Runge phenomenon for $f(x) = \frac{1}{1+25x^2}$.

Much of this chapter is concerned with finding better interpolation nodes that avoid the Runge phenomenon for sufficiently nice functions. We will also discuss links to stability and best approximation.

3.1 Chebyshev nodes

Motivated by the error *bound* (3.2) we now aim at determining interpolation nodes $x_0, \ldots, x_n$ that minimize

$$\|\omega_{n+1}\|_\infty = \max_{x\in[a,b]} |(x - x_0)(x - x_1)\cdots(x - x_n)|.$$

By an affine linear transformation of the interval and the interpolation nodes, we may assume without loss of generality that $[a, b] = [-1, 1]$. Minimizing $\|\omega_{n+1}\|_\infty$ directly by brute force would be a daunting task. We therefore approach it indirectly via Chebyshev polynomials.

Definition 3.1 *Given $n \in \mathbb{N}$, the nth* Chebyshev polynomial *is defined as*

$$T_n(x) = \cos(n \arccos x) \qquad \forall\ x \in [-1, 1].$$

Note that for every $n \in \mathbb{N}$ and $x \in [-1,1]$ we have $|T_n(x)| \leq 1$. Moreover, in spite of the unusual definition, it will follow from Lemma 3.2 below that $T_n \in \mathbb{P}_n$ for every $n \in \mathbb{N}$. By computing the first few examples, we obtain indeed:

$$\begin{aligned} n = 0: &\quad T_0(x) = \cos(0 \arccos x) = \cos(0) = 1, \\ n = 1: &\quad T_1(x) = \cos(1 \arccos x) = x, \\ n = 2: &\quad T_2(x) = \cos(2 \arccos x) = 2\cos^2(\arccos x) - 1 = 2x^2 - 1. \end{aligned}$$

Theorem 3.2 *The Chebyshev polynomials satisfy the following* recurrence relation*:*

$$T_0(x) = 1, \quad T_1(x) = x, \quad T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x) \quad n \in \mathbb{N}, n \geq 1.$$

Proof. For every $n \in \mathbb{N}$, $n \geq 1$, the following trigonometric identity holds:

$$\cos\left((n+1)\,\varphi\right) + \cos\left((n-1)\,\varphi\right) = 2\cos\varphi\cos\left(n\varphi\right) \qquad \forall\ \varphi \in \mathbb{R}.$$

Setting $\varphi = \arccos x$, this implies

$$T_{n+1}(x) + T_{n-1}(x) = 2T_1(x)T_n(x) = 2xT_n(x) \qquad \forall\ x \in [-1,1],$$

which completes the proof. □

Note that the result of Theorem 3.2 also implies that the leading coefficient of T_{n+1} is 2^n.

Lemma 3.3 *The roots of T_n are*

$$x_k = \cos\left(\frac{(2k+1)\,\pi}{2n}\right), \qquad k = 0, \ldots, n-1,$$

which are called Chebyshev nodes*.*

Proof. A direct calculation yields, for every $k \in \mathbb{N}$, $0 \leq k \leq n-1$,

$$T_n\left(x_k\right) = \cos\left(n \arccos\cos\left(\frac{(2k+1)\,\pi}{2n}\right)\right) = \cos\left(\frac{(2k+1)\,\pi}{2}\right) = 0.$$

These are all the roots of T_n because, by Theorem 3.2, T_n is a (nonzero) polynomial of degree n. □

Lemma 3.3 shows that T_n has n real, pairwise distinct roots in the open interval $(-1,1)$. Moreover, it can be seen from Figure 3.2 that the roots tend to cluster at the end points of $[-1,1]$.[9]

By looking at the extremal properties of T_n, we are getting closer to our goal of optimizing $\|\omega_{n+1}\|_\infty$.

[9] EFY: Show that the Chebyshev nodes are the real parts of points *uniformly* distributed on the upper part of the unit circle.

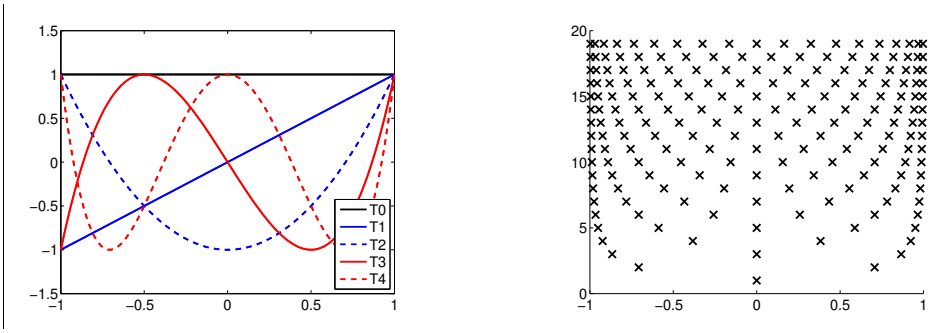


Figure 3.2: *Left:* Chebyshev polynomials $T_0, \ldots, T_4$. *Right:* Roots of Chebyshev polynomials $T_1, \ldots, T_{20}$.

Lemma 3.4 *For $n \geq 1$ the Chebyshev polynomial T_n takes alternatingly the values $+1$ and -1 exactly $n+1$ times:*

$$T_n\left(\cos\left(\frac{k\pi}{n}\right)\right) = (-1)^k, \qquad \forall\ k = 0, \ldots, n.$$

Proof. By differentiation, it is easy to show that T_n attains at the points $\cos\left(\frac{k\pi}{n}\right)$, $k = 0, \ldots n$, a local minimum for k odd and a local maximum for k even. Since, by construction, $\|T_n\|_\infty \leq 1$, these are global extrema. ∎

We are now ready to solve our optimization problem.

Lemma 3.5 *Among all polynomials of degree $n+1$ with leading coefficient 1, the rescaled Chebyshev polynomial $\tilde{T}_{n+1} := 2^{-n} T_{n+1}$ minimizes the uniform norm on $[-1, 1]$.*

Proof. From Lemma 3.4 we have that $\tilde{T}_{n+1}$ alternates at the values $+2^{-n}$ and -2^{-n} exactly $n+2$ times. We assume, in contradiction to the statement of the lemma, that there exists $q \in \mathbb{P}_{n+1}$ with leading coefficient 1 such that $\|q\|_\infty < 2^{-n}$ and define $p(x) := q(x) - 2^{-n} T_{n+1}(x)$ for every $x \in [-1, 1]$. Note that $p \in \mathbb{P}_n$ since the terms x^{n+1} cancel. Note, also, that p is nonzero because otherwise $\|q\|_\infty = 2^{-n}$. Moreover, p changes sign in each interval (z_i, z_{i+1}) for $i = 0, \ldots, n$, where $z_0, \ldots, z_{n+1}$ are the points where T_{n+1} attains its extreme values. In turn, by the intermediate value theorem and continuity of polynomials, p admits $n+1$ distinct roots, which is not possible for a nonzero polynomial of degree n. ∎

The result of Lemma 3.5 tells us that choosing the Chebyshev nodes (that is, the roots of T_{n+1}) as interpolation nodes leads to the ω_{n+1} of smallest uniform norm. Mapping the Chebyshev nodes to a general interval $[a, b]$, we arrive at the following result.

Theorem 3.6 *The expression* $\max_{x\in[a,b]} |(x-x_0)\dots(x-x_n)|$ *is minimized for the interpolation nodes*

$$x_k = \frac{a+b}{2} + \frac{b-a}{2}\cos\left(\frac{(2k+1)\,\pi}{2n+2}\right), \qquad k=0,\dots,n.$$

For this choice of interpolation nodes, the interpolating polynomial p_n *satisfies*

$$\|f-p_n\|_\infty \le \frac{1}{2^n(n+1)!}\left(\frac{b-a}{2}\right)^{n+1}\|f^{(n+1)}\|_\infty.$$

Proof. The proof proceeds, similarly to what has been used in Chapter 2, by an affine linear mapping from the reference interval to the interval of interest. In this case, the reference interval is $[-1,1]$ and the interval of interest is $[a,b]$, and the mapping takes the form

$$\varphi : [-1,1] \to [a,b], \qquad \varphi : x \mapsto \frac{a+b}{2} + \frac{b-a}{2}x.$$

Note that this map is invertible, with the inverse map given by

$$\varphi^{-1} : [a,b] \to [-1,1], \qquad \varphi^{-1} : y \mapsto \frac{2}{b-a}\left(y - \frac{a+b}{2}\right).$$

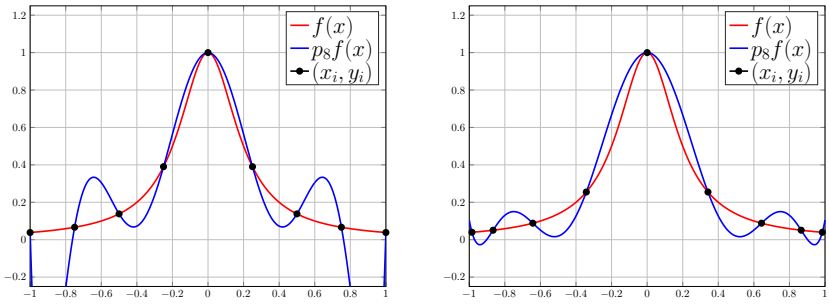
We can use φ to map interpolation nodes $\tilde{x}_0,\dots,\tilde{x}_n \in [-1,1]$, to the interval $[a,b]$: $x_k = \varphi(\tilde{x}_k)$ for $k=0,\dots,n$. Setting $\tilde{x} = \varphi^{-1}(x) \in [-1,1]$, we obtain

$$\begin{aligned}
\omega_{n+1}(x) &= (x-x_0)(x-x_1)\cdots(x-x_n)\\
&= (\varphi(\tilde{x})-\varphi(\tilde{x}_0))(\varphi(\tilde{x})-\varphi(\tilde{x}_1))\cdots(\varphi(\tilde{x})-\varphi(\tilde{x}_n))\\
&= \left(\frac{b-a}{2}\right)^{n+1} \underbrace{(\tilde{x}-\tilde{x}_0)(\tilde{x}-\tilde{x}_1)\cdots(\tilde{x}-\tilde{x}_n)}_{=:\tilde{\omega}_{n+1}(\tilde{x})}.
\end{aligned}$$

By Lemma 3.5, we know that the maximum norm of $\tilde{\omega}_{n+1}$ on $[-1,1]$ is minimized by choosing $\tilde{x}_k = \cos((2k+1)\pi/(2n+2))$, the roots of the Chebyshev polynomial T_{n+1}. The relation above shows that the maximum norm of ω_{n+1} on $[a,b]$ is minimized by setting $x_k = \varphi(\tilde{x}_k) = \varphi(\cos((2k+1)\pi/(2n+2))$, which proves the first part of the theorem. The second part follows directly from the error bound (3.2):

$$\begin{aligned}
\|f-p_n\|_\infty &\le \frac{1}{(n+1)!}\|\omega_{n+1}\|_\infty\|f^{(n+1)}\|_\infty\\
&= \frac{1}{(n+1)!}\left(\frac{b-a}{2}\right)^{n+1}\|\tilde{\omega}_{n+1}\|_\infty\|f^{(n+1)}\|_\infty\\
&= \frac{1}{2^n(n+1)!}\left(\frac{b-a}{2}\right)^{n+1}\|f^{(n+1)}\|_\infty,
\end{aligned}$$

where the last inequality uses $\tilde{\omega}_{n+1}(\tilde{x}) = 2^{-n} T_{n+1}(\tilde{x})$. □



(a) Interpolating on equidistant points, $n = 8$, $f(x) = \frac{1}{1+25x^2}$.

(b) Interpolating on Chebyshev nodes, $n = 8$, $f(x) = \frac{1}{1+25x^2}$.

Figure 3.3: Comparison of polynomial interpolation with equidistant and Chebyshev nodes.

Figure 3.3 confirms that Chebyshev nodes compare favorably with equidistant nodes for the Runge function.

It can be shown that the error $\|f - p_n\|_\infty$ converges to zero for any Lipschitz function f when choosing Chebyshev nodes. For a real analytic function f, the error converges exponentially fast, that is, it is bounded by a constant times ρ^{-n}, where $\rho > 1$ depends on the domain of analyticity of f. We highly recommend Trefethen's book [6] and the associated Chebfun software package for many more fascinating facts and uses of Chebyshev polynomials.[10]

3.2 Sensitivity and best uniform approximation

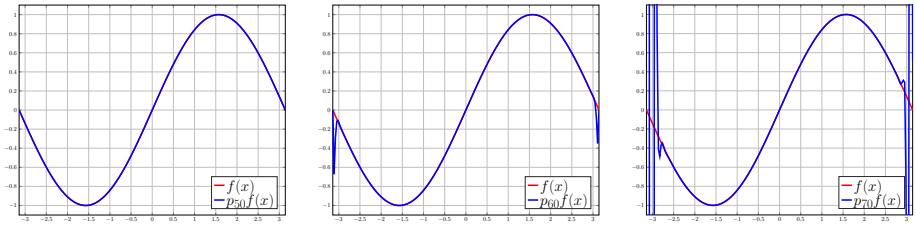
Building and evaluating interpolating polynomials is subject to roundoff error on a computer. Figures 3.4 and 3.5 show the results of interpolating $f(x) = \sin(x)$ on $[-\pi, \pi]$. Theoretically, for both equidistant and Chebyshev nodes, we expect that $\|f - p_n\|_\infty$ converges to zero as n increases. Instead, Figure 3.4 shows "wiggles" for large n in the case of equidistant nodes. These wiggles are due to roundoff error caused by numerical instabilities when using equidistant nodes.

Roundoff error already occurs when evaluating and storing the function values $y_i = f(x_i)$ on the computer. Let us investigate the effect of this error on the interpolating polynomial. For this purpose, suppose that

$$\hat{f}(x_i) = f(x_i)(1 + \delta_i), \qquad \text{where} \qquad |\delta_i| \leq \epsilon.$$

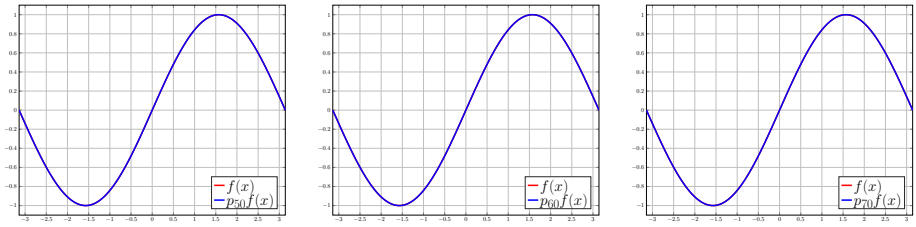
For elementary functions, we know that $\epsilon = u$, the unit roundoff. For more complex functions, this can be larger, but often one still has $\epsilon \approx 10^{-16}$. The interpolating

[10] EFY: Download Chebfun / ApproxFun / pychebfun and play with it!



(a) $n = 50$ (b) $n = 60$ (c) $n = 70$

Figure 3.4: Interpolation of $f(x) = \sin(x)$ on equidistant nodes.



(a) $n = 50$ (b) $n = 60$ (c) $n = 70$

Figure 3.5: Interpolating of $f(x) = \sin(x)$ on Chebyshev nodes.

polynomials for f and $\hat{f}$ are given by

$$p_n(x) = \sum_{i=0}^{n} f(x_i)\,\ell_i(x), \qquad \hat{p}_n(x) = \sum_{i=0}^{n} \hat{f}(x_i)\,\ell_i(x). \tag{3.3}$$

respectively. Then for every $x \in [a, b]$ we have that

$$\begin{aligned} |p_n(x) - \hat{p}_n(x)| &= \Big| \sum_{i=0}^{n} \big(f(x_i) - \hat{f}(x_i)\big)\ell_i(x) \Big| \\ &\leq \sum_{i=0}^{n} \epsilon |f(x_i)|\,|\ell_i(x)| \leq \epsilon \|f\|_\infty \sum_{i=0}^{n} |\ell_i(x)|. \end{aligned} \tag{3.4}$$

Definition 3.7 *The quantity*

$$\Lambda_n := \max_{x\in[a,b]} \sum_{i=0}^{n} |\ell_i(x)|$$

is called Lebesgue constant *associated with the interpolation nodes* $x_0, \ldots, x_n$.

Table 3.1: Lebesgue constant Λ_n for equidistant and Chebyshev nodes on $[-1, 1]$.

n	equidistant	Chebyshev
5	3.106	2.104
10	29.89	2.489
15	512.05	2.728
20	10986.53	2.901

By the discussion above, we have

$$\|p_n - \hat{p}_n\|_\infty \leq \epsilon \Lambda_n \|f\|_\infty .$$

Hence, Λ_n measures the sensitivity of the interpolating polynomial with respect to perturbations in the interpolation data. Table 3.1 shows that the Lebesgue constant grows much more rapidly for equidistant nodes than for Chebyshev nodes, explaining the effect observed in Figure 3.4. It has been shown that

$$\Lambda_n \sim \frac{2^n}{e(n-1)\ln n} \quad n \to +\infty$$

for equidistant nodes and

$$\Lambda_n \sim \frac{2}{\pi} \ln n \quad n \to +\infty \tag{3.5}$$

for Chebyshev nodes. There is a dramatic difference; exponential vs. logarithmic growth!

The Lebesgue constant also measures how far one is away from the best uniform approximation of a function.

Theorem 3.8 *Let $f \in C^0([a, b])$ and consider the interpolating polynomial p_n for f on interpolation nodes $x_0, \ldots, x_n \in [a, b]$. Then*

$$\inf_{q \in \mathbb{P}_n} \|f - q\|_\infty \leq \|f - p_n\|_\infty \leq (1 + \Lambda_n) \inf_{q \in \mathbb{P}_n} \|f - q\|_\infty.$$

Proof. The first inequality holds trivially because $p_n \in \mathbb{P}_n$. We first note the trivial fact that the interpolation of a polynomial p of degree at most n by a polynomial of degree n is the polynomial p itself. This implies

$$\begin{aligned} f(x) - p_n(x) &= f(x) - q(x) + q(x) - p_n(x) \\ &= f(x) - q(x) + \sum_{i=0}^{n} (q(x_i) - p_n(x_i))\ell_i(x). \end{aligned}$$

for any $q \in \mathbb{P}_n$. Hence,

$$\|f - p_n\|_\infty \leq \|f - q\|_\infty + \|f - q\|_\infty \Big\| \sum_{j=0}^{n} |\ell_j(x)| \Big\|_\infty = (1 + \Lambda_n)\|f - q\|_\infty.$$

Because this holds for arbitrary $q \in \mathbb{P}_n$, this completes the proof. □

Using Theorem 3.8 and (3.5), we see that Chebyshev interpolation attains nearly the best uniform approximation. In practice, being nearly best is usually sufficient. If one wants to achieve the truly best approximation in the L^∞ norm, one needs to resort to the so called Remez algorithm.

3.3 Best approximation in L^2 norm*

We now consider the best approximation in the L^2 norm on $[-1,1]$:

$$\|u\|_2 := \Big(\int_{-1}^{1} |u|^2 \, \mathrm{d}t \Big)^{1/2} \tag{3.6}$$

for a function $u : [-1,1] \to \infty$. More specifically, we aim at solving the following minimization problem. Given $f \in C^0([-1,1])$, determine

$$p^* = \operatorname*{arg\,min}_{q_n \in \mathbb{P}_n} \|f - q_n\|_2^2 . \tag{3.7}$$

What makes (3.7) fundamentally different (and simpler) compared to best uniform approximation is that $\|\cdot\|_2$ is induced by the L^2 inner product

$$(u,v)_2 = \int_{-1}^{1} u(t)v(t) \, \mathrm{d}t.$$

It is instructive to frame (3.7) in an abstract setting: Let V be a (possibly infinite-dimensional) real vector space with an inner product $(\cdot,\cdot)_V$. Letting U be a finite-dimensional subspace of V, we consider for given $v \in V$ the approximation problem

$$u^* = \operatorname*{arg\,min}_{u \in U} \|v - u\|_V^2. \tag{3.8}$$

Here, $\|\cdot\|_V^2$ denotes the norm induced by the inner product, that is, $\|w\|_V^2 = (w,w)_V$. We let v_U denote the orthogonal projection of $v \in V$ onto U, that is, the unique vector $v_U \in U$ in the decomposition

$$v = v_U + v_\perp, \quad v_U \in U, \quad (v_\perp, u)_V = 0 \ \forall u \in U. \tag{3.9}$$

Then for any vector $u \in U$ it holds that

$$\|v-u\|_V^2 = \|v_\perp + (v_u - u)\|_V^2 = \|v_\perp\|_V^2 + 2(v_\perp, v_u - u) + \|v_u - u\|_V^2 = \|v_\perp\|_V^2 + \|v_u - u\|_V^2.$$

The last expression is minimized by setting $u = v_u$. This yields the following theorem.

Theorem 3.9 *The unique solution to the minimization problem* (3.8) *is the orthogonal projection of v onto U.*

An explicit expression for $u^* = v_U$ is obtained when choosing an orthonormal basis $u_0, u_1, \ldots, u_n$ of U, where $\dim U =: n+1$. Then

$$v_U = \sum_{k=0}^{n} (u_k, v) u_k. \tag{3.10}$$

To see this, we need to verify that (3.9) is satisfied for $v_\perp = v - v_U$. First, it directly follows that $v_u \in U$. Second, for every u_j orthonormality implies

$$(v_\perp, u_j) = (v - v_U, u_j) = (v, u_j) - \sum_{k=0}^{n} (u_k, v)(u_k, u_j) = (v, u_j) - (v, u_j) = 0.$$

Hence, $v_\perp$ is orthogonal to every $u \in U$.

To apply Theorem 3.8 and (3.10) to polynomial approximation on $[-1, 1]$, we recall that, by Definition 2.10, the Legendre polynomials $q_0, \ldots, q_n$ form an *orthogonal* basis of Π_n in the L^2 inner product. It can be shown that

$$\|q_k\|_2^2 = \frac{2}{2k+1}.$$

Hence the scaled Legendre polynomials $\tilde{q}_0, \ldots, \tilde{q}_n$ with $\tilde{q}_k := \sqrt{\frac{2k+1}{2}} q_k$ form an *orthonormal* basis of Π_n. As a corollary of Theorem 3.9 we now obtain the solution of (3.7).

Corollary 3.10 *Given $f \in C^0([-1, 1])$, the approximation problem* (3.7) *is solved by*

$$p^*(x) = \sum_{k=0}^{n} (\tilde{q}_k, f)_2 \tilde{q}_k,$$

with the scaled Legendre polynomials $\tilde{q}_0, \ldots, \tilde{q}_n$.

In order to construct p^* we need to compute

$$(\tilde{q}_k, f)_2 = \int_{-1}^{1} f \tilde{q}_k \, \mathrm{d}t, \quad k = 0, \ldots, n.$$

In general, one cannot calculate these quantities exactly . Instead, one can employ a Gauss quadrature with $n+1$ points in order to approximate them.

3.4 A note on piece-wise interpolation*

Similar to composite numerical quadrature, piece-wise interpolation partitions the interval into subintervals and applies polynomial interpolation to every subinterval. For $n = 1$ (that is, piece-wise linear interpolation) this will yield a continuous, polygonal curve when choosing on each subinterval the end points as interpolation nodes. For larger n, the additional degrees of freedom are *not* used to interpolate additional nodes in each subinterval but to achieve smoothness at the end points of subintervals. For $n = 3$, this yields the concept of (cubic) splines, which are two times differentiable on the whole interval. Constructing such splines involves the solution of linear systems; the details of the construction are beyond the scope of this lecture.

Chapter 4

Linear Systems – Small Matrices

The solution of a **linear system of equations**

$$A\mathbf{x} = \mathbf{b}$$

with a square invertible matrix $A \in \mathbb{R}^{n\times n}$ and a right-hand side vector $\mathbf{b} \in \mathbb{R}^n$ is one of the most frequent tasks in numerical analysis. The matrix A and the vectors $\mathbf{x}$, $\mathbf{b}$ have the entries

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & & \vdots \\ \vdots & \vdots & & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & \cdots & a_{nn} \end{pmatrix}, \qquad \mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ \vdots \\ x_n \end{pmatrix}, \qquad \mathbf{b} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ \vdots \\ b_n \end{pmatrix}.$$

Example 4.1 Consider the following system of 3 linear equations:

$$\begin{array}{rrrcr} 2x_1 & -2x_2 & +4x_3 & = & 6 \\ -5x_1 & +6x_2 & -7x_3 & = & -7 \\ 3x_1 & +2x_2 & +x_3 & = & 9 \end{array}$$

Using the definition of the matrix-vector product, this can be written in matrix-vector form as

$$\begin{pmatrix} 2 & -2 & 4 \\ -5 & 6 & -7 \\ 3 & 2 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 6 \\ -7 \\ 9 \end{pmatrix}.$$

Example 4.2 Figure 4.1 shows a hydraulic network[11] of 10 pipelines. It is fed by a water reservoir having a constant pressure of $p = 10$ bar. Here and in the following, pressure values refer to the difference between the real pressure and the atmospheric pressure. The flow rate Q_j (in m^3/s) of the jth pipeline is proportional

[11] Example taken from [A. Quarteroni, F. Saleri, and P. Gervasi. Springer, 2010].

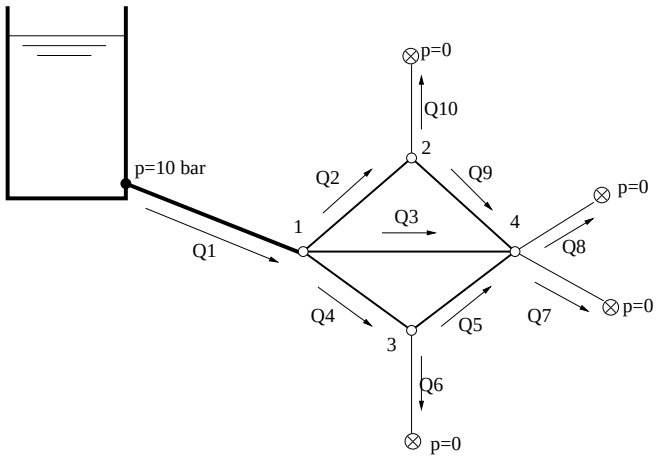


Figure 4.1: Hydraulic network from Example 4.2.

to the length L_j (in m) of the pipeline and the pressure difference Δp_j at both ends of the pipeline:

$$Q_j = k_j L_j \Delta p_j \tag{4.1}$$

The constant k_j denotes the hydraulic resistance (in m/(bar s)), which depends on the shape of the pipe and the fluid viscosity. It is assumed that the water flows from outlets (marked by $\otimes$ in the figure) at atmospheric pressure, and hence $p = 0$ at the exterior nodes of the network. To determine the pressure at the internal nodes $1, 2, 3, 4$, we can use that the flow rates at each internal node must sum up to zero. Denoting these pressures by $\mathbf{p} = [p_1, p_2, p_3, p_4]^T$, this implies for node 1:

$$\begin{aligned} & Q_1 - Q_2 - Q_3 - Q_4 = 0 \\ \overset{(4.1)}{\Longrightarrow} \quad & k_1 L_1 (p_1 - 10) - k_2 L_2 (p_2 - p_1) - k_4 L_4 (p_3 - p_1) - k_3 L_3 (p_4 - p_1) = 0 \\ \Longrightarrow \quad & -10 k_1 L_1 = -(k_1 L_1 + k_2 L_2 + k_3 L_3 + k_4 L_4) p_1 + k_2 L_2 p_2 + k_4 L_4 p_3 + k_3 L_3 p_4 \end{aligned}$$

To proceed, we choose concrete values for k_j and L_j as follows.

pipeline	k_j	L_j	pipeline	k_j	L_j	pipeline	k_j	L_j
1	0.01	20	2	0.005	10	3	0.005	14
4	0.005	10	5	0.005	10	6	0.002	8
7	0.002	8	8	0.002	8	9	0.005	10
10	0.002	8						

Inserting these values into the equation above gives

$$-2 = -0.37\, p_1 + 0.05\, p_2 + 0.05\, p_3 + 0.07\, p_4.$$

Similarly, linear equations can be derived for the other internal nodes $2, 3, 4$. In summary, this yields the linear system $A\mathbf{p} = \mathbf{b}$ with

$$A = \begin{pmatrix} -0.370 & 0.050 & 0.050 & 0.070 \\ 0.050 & -0.116 & 0 & 0.050 \\ 0.050 & 0 & -0.116 & 0.050 \\ 0.070 & 0.050 & 0.050 & -0.202 \end{pmatrix}, \qquad \mathbf{b} = \begin{pmatrix} -2 \\ 0 \\ 0 \\ 0 \end{pmatrix}.$$

The solution of this linear system will be presented in Example 4.11. ◇

4.1 Triangular Matrices

Before coming to the solution of a general linear system $A\mathbf{x} = \mathbf{b}$, we first consider two special cases for the matrix A.
A **lower triangular matrix** A is a square matrix satisfying

$$a_{ij} = 0 \qquad \text{for all } i, j \text{ with } i < j.$$

An **upper triangular matrix** A is a square matrix satisfying

$$a_{ij} = 0 \qquad \text{for all } i, j \text{ with } i > j.$$

Often, we will denote lower triangular matrices with the letter L and upper triangular matrices with the letter U. The definitions imply the shapes

$$L = \begin{pmatrix} \ell_{11} & 0 & \cdots & \cdots & 0 \\ \ell_{21} & \ell_{22} & 0 & \cdots & \\ \ell_{31} & \ell_{32} & \ddots & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & 0 \\ \ell_{n1} & \ell_{n2} & \cdots & \ell_{nn-1} & \ell_{nn} \end{pmatrix}, \qquad U = \begin{pmatrix} u_{11} & u_{12} & \cdots & \cdots & u_{1n} \\ 0 & u_{22} & \cdots & \cdots & u_{2n} \\ 0 & 0 & \ddots & & \vdots \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 & u_{nn} \end{pmatrix}.$$

Pictorially:

$L =$  , $U =$  .

The solution of linear systems with (lower or upper) triangular matrices is quite simple. For example, consider

$$\begin{pmatrix} 1 & 2 & -1 \\ 0 & 2 & 1 \\ 0 & 0 & 2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 2 \\ 3 \\ 2 \end{pmatrix} \qquad \Leftrightarrow \qquad \begin{aligned} x_1 + 2x_2 - x_3 &= 2 \\ 2x_2 + x_3 &= 3 \\ 2x_3 &= 2 \end{aligned}$$

The last equation can be immediately solved: $x_3 = 2/2 = 1$. Inserting this into the second equation gives

$$2x_2 + 2x_3 = 3 \quad \Rightarrow \quad x_2 = \frac{1}{2}(3 - x_3) = 1.$$

Finally, inserting $x_2 = 1$ and $x_3 = 1$ into the first equation gives

$$x_1 + 2x_2 - x_3 = 2 \quad \Rightarrow \quad x_1 = 2 - 2x_2 + x_3 = 1.$$

This process of eliminating the variables $x_n, x_{n-1}, \ldots$ is called *backward substitution*. Similarly, a linear system with a lower triangular matrix can be solved by eliminating the variables $x_1, x_2, \ldots$. This process is called *forward substitution*.

For general triangular matrices, forward and backward substitution are given by the following two algorithms.

Algorithm 4.3
Forward substitution

Input: Invertible lower triangular $L \in \mathbb{R}^{n\times n}$, $\mathbf{b} \in \mathbb{R}^n$.

Output: Solution $\mathbf{x}$ of $L\mathbf{x} = \mathbf{b}$.

for $i = 1, 2, \ldots, n$ **do**

$$x_i := \frac{1}{\ell_{ii}} \left(b_i - \sum_{k=1}^{i-1} \ell_{ik} x_k \right)$$

end for

Algorithm 4.4
Backward substitution

Input: Invertible upper triangular $U \in \mathbb{R}^{n\times n}$, $\mathbf{b} \in \mathbb{R}^n$.

Output: Solution $\mathbf{x}$ of $U\mathbf{x} = \mathbf{b}$.

for $i = n, n-1, \ldots, 1$ **do**

$$x_i := \frac{1}{u_{ii}} \left(b_i - \sum_{k=i+1}^{n} u_{ik} x_k \right)$$

end for

It is a good exercise to convince yourself that the right-hand side of the assignment in both algorithms only contains terms that are already known.

Remark 4.5 *In* PYTHON*, linear systems can be solved with the commands*

`numpy.linalg.solve(A, b)` *or* `scipy.linalg.solve(A, b)`

for a square, invertible matrix A. However, these functions do not automatically check whether A is triangular and, therefore, they are unnecessarily slow in such situations. To solve triangular linear systems, one should call `scipy.linalg.solve` *with the option* `assume_a` *set to 'upper triangular' or 'lower triangular'. Alternatively, one can also call* `scipy.linalg.solve_triangular`*.*

Let us perform a complexity analysis for Algorithm 4.3. The cost of an algorithm is determined by the number of elementary operations $+, -, *, /$ and elementary function evaluations. Each operation / evaluation is counted as one **flop** (floating point operation). The ith loop of Algorithm 4.3 performs 1 division, $i-1$ multiplications, and $i-1$ additions/subtractions; a total of $2i-1$ flops. Therefore, the total cost of Algorithm 4.3 is given by

$$\sum_{i=1}^{n} (2i - 1) = n^2 \text{ flops.} \tag{4.2}$$

The cost of Algorithm 4.4 is the same.

4.2 LU factorization

Knowing that a linear system with a triangular matrix is considerably simple to solve, we now try to reduce a general system to this case. To be more precise, we

will use a variant of **Gaussian elimination** to write the matrix A as a product of triangular matrices:

$$A = LU = \text{◺} \cdot \text{◹} \tag{4.3}$$

Once we know such a factorization, we can solve the linear system $A\mathbf{x} = \mathbf{b}$ with forward and backward substitution. If we introduce the auxiliary vector $\mathbf{y} = U\mathbf{x}$ then $\mathbf{b} = A\mathbf{x} = LU\mathbf{x} = L(U\mathbf{x}) = L\mathbf{y}$. Hence, we can solve $(LU)\mathbf{x} = \mathbf{b}$ in two steps:

1. solve $L\mathbf{y} = \mathbf{b}$ for $\mathbf{y}$ with Algorithm 4.3;
2. solve $U\mathbf{x} = \mathbf{y}$ for $\mathbf{x}$ with Algorithm 4.4.

The LU factorization of a matrix $A \in \mathbb{R}^{n\times n}$ proceeds in $n-1$ steps, by eliminating column-by-column the entries of A below the diagonal.

Example 4.6 We illustrate the computation of an LU factorization for the matrix from Example 4.1:

$$A = \begin{pmatrix} 2 & -2 & 4 \\ -5 & 6 & -7 \\ 3 & 2 & 1 \end{pmatrix}$$

In Step 1, we eliminate the entries below the first diagonal entry, by adding $5/2\times$ row 1 to row 2, and subtracting $3/2\times$ row 1 from row 3. As a result, we obtain the modified matrix

$$A^{(1)} := \begin{pmatrix} 2 & -2 & 4 \\ 0 & 1 & 3 \\ 0 & 5 & -5 \end{pmatrix}.$$

The crucial observation is that this step can be written as a matrix-matrix multiplication:

$$A^{(1)} := L_1 A = L_1 \begin{pmatrix} 2 & -2 & 4 \\ -5 & 6 & -7 \\ 3 & 2 & 1 \end{pmatrix}, \quad \text{with} \quad L_1 = \begin{pmatrix} 1 & 0 & 0 \\ 5/2 & 1 & 0 \\ -3/2 & 0 & 1 \end{pmatrix}.$$

In Step 2, we eliminate the remaining entry 5 in $A^{(1)}$ below the second diagonal element. This can be achieved by subtracting $5\times$ row 2 from row 3, leading to

$$A^{(2)} := \begin{pmatrix} 2 & -2 & 4 \\ 0 & 1 & 3 \\ 0 & 0 & -20 \end{pmatrix}$$

Again, this can be written as a matrix-matrix product

$$A^{(2)} := L_2 A^{(1)} = L_2 \begin{pmatrix} 2 & -2 & 4 \\ 0 & 1 & 3 \\ 0 & 5 & -5 \end{pmatrix}, \quad \text{with} \quad L_2 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -5 & 1 \end{pmatrix}.$$

Setting $U := A^{(2)}$, the two steps above can be summarized as

$$A = LU, \quad \text{with} \quad L = L_1^{-1} L_2^{-1}.$$

It turns out that – due to their very special structure – the inverses of L_1 and L_2 can be simply obtained by negating the elements below the diagonal:

$$L_1^{-1} = \begin{pmatrix} 1 & 0 & 0 \\ -5/2 & 1 & 0 \\ 3/2 & 0 & 1 \end{pmatrix}, \qquad L_2^{-1} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 5 & 1 \end{pmatrix}.$$

This can be easily verified by checking $L_1 L_1^{-1} = I_3$ and $L_2 L_2^{-1} = I_3$. Moreover, again due the very special structure, the product $L_1^{-1} L_2^{-1}$ is simply obtained by collecting all nonzero sub-diagonal elements:

$$L = L_1^{-1} L_2^{-1} = \begin{pmatrix} 1 & 0 & 0 \\ -5/2 & 1 & 0 \\ 3/2 & 5 & 1 \end{pmatrix}$$

Hence,

$$A = LU, \quad \text{with} \quad L = \begin{pmatrix} 1 & 0 & 0 \\ -5/2 & 1 & 0 \\ 3/2 & 5 & 1 \end{pmatrix}, \quad U = \begin{pmatrix} 2 & -2 & 4 \\ 0 & 1 & 3 \\ 0 & 0 & -20 \end{pmatrix}.$$

which is the LU factorization of A. ◇

The procedure from Example 4.6 easily extends to a general matrix A. *Before* Step k, the modified matrix A takes the form

$$A^{(k-1)} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & \cdots & \cdots & a_{1n} \\ & a_{22}^{(1)} & a_{23}^{(1)} & \cdots & \cdots & a_{2n}^{(1)} \\ & & \ddots & \ddots & & \vdots \\ & & & a_{kk}^{(k-1)} & \cdots & a_{kn}^{(k-1)} \\ & & & \vdots & & \vdots \\ & & & a_{nk}^{(k-1)} & \cdots & a_{nn}^{(k-1)} \end{pmatrix}. \tag{4.4}$$

(For Step 1, we formally set $A^{(0)} := A$.) For performing Step k, the coefficients

$$\ell_{ik} := \frac{a_{ik}^{(k-1)}}{a_{kk}^{(k-1)}}, \qquad i = k+1, \ldots, n,$$

are computed. Of course, this is only possible if the so called **pivot element** $a_{kk}^{(k-1)}$ is nonzero. We will come back to this limitation below, in Section 4.3.

The multiplication of the matrix

$$L_k := \begin{pmatrix} 1 & & & & & \\ & \ddots & & & & \\ & & 1 & & & \\ & & -\ell_{k+1,k} & 1 & & \\ & & \vdots & & \ddots & \\ & & -\ell_{n\,k} & & & 1 \end{pmatrix} \tag{4.5}$$

with $A^{(k-1)}$ performs Step k and eliminates all entries below the $(k-1)$th diagonal entry. The whole procedure is then repeated with the resulting matrix

$$A^{(k)} = L_k A^{(k-1)}. \tag{4.6}$$

After $n-1$ steps, we obtain

$$A^{(n-1)} = L_{n-1} L_{n-2} \cdots L_1 A^{(0)} = L_{n-1} L_{n-2} \cdots L_1 A.$$

This can be rewritten as

$$LU = A,$$

where

$$L := L_1^{-1} L_2^{-1} \cdots L_{n-1}^{-1}, \qquad U = A^{(n-1)}.$$

Note that U is upper triangular by construction. The factors L_k^{-1} of the matrix L are given by

$$L_k^{-1} = \begin{pmatrix} 1 & & & & & \\ & \ddots & & & & \\ & & 1 & & & \\ & & \ell_{k+1,k} & 1 & & \\ & & \vdots & & \ddots & \\ & & \ell_{n\,k} & & & 1 \end{pmatrix}, \tag{4.7}$$

which can be verified by checking $L_k^{-1} L_k = I$. Due to the special structure of these factors, the sub-diagonal entries of L are obtained from collecting the subdiagonal entries of all L_k^{-1}:

$$L = L_1^{-1} L_2^{-1} \cdots L_{n-1}^{-1} = \begin{pmatrix} 1 & & & & & \\ \ell_{21} & \ddots & & & & \\ \ell_{31} & \ddots & 1 & & & \\ \vdots & & \ell_{k+1,k} & 1 & & \\ \vdots & & \vdots & \ddots & \ddots & \\ \ell_{n1} & \cdots & \ell_{n\,k} & \cdots & \ell_{n,n-1} & 1 \end{pmatrix}. \tag{4.8}$$

The procedure above is summarized in the following algorithm.

Algorithm 4.7 (Abstract form of *LU* factorization)
Input: Invertible matrix $A \in \mathbb{R}^{n \times n}$.
Output: LU factorization $A = LU$ with $U = A^{(n-1)}$ and L as in (4.8).

$A^{(0)} := A$
for $k = 1, \ldots, n-1$ **do**
Determine matrix L_k (see (4.5)) by computing the coefficients

$$\ell_{ik} := \frac{a_{ik}^{(k-1)}}{a_{kk}^{(k-1)}}, \qquad i = k+1, \ldots, n.$$

Set $A^{(k)} := L_k A^{(k-1)}$.
end for

Not every invertible matrix A has an LU factorization, that is $A = LU$ with L lower triangular with ones and the diagonal and U upper triangular. The following theorem characterizes which ones have.

Theorem 4.8 *Let $A \in \mathbb{R}^{n\times n}$ be invertible. Then A has an LU factorization if and only all leading principal submatrices*

$$A_k = \begin{pmatrix} a_{11} & a_{12} & \cdots & \cdots & a_{1k} \\ a_{21} & a_{22} & \cdots & \cdots & a_{2k} \\ \vdots & \vdots & \ddots & & \vdots \\ \vdots & \vdots & & \ddots & \vdots \\ a_{k1} & a_{k2} & \cdots & \cdots & a_{kk} \end{pmatrix}, \quad k = 1, \ldots, n-1,$$

are also invertible.

Proof. Given an LU factorization $A = LU$, we partition

$$L = \begin{pmatrix} L_{11} & 0 \\ L_{12} & L_{22} \end{pmatrix}, \quad U = \begin{pmatrix} U_{11} & U_{12} \\ 0 & U_{22} \end{pmatrix}, \quad L_{11}, U_{11} \in \mathbb{R}^{k\times k}.$$

Then

$$A_k = L_{11}U_{11} \quad \Rightarrow \quad \det(A_k) = \det(L_{11})\det(U_{11}) = \det(U_{11}),$$

where we used that L has ones on the diagonal. Because $0 \neq \det(A) = \det(U) = \prod_{i=1}^{n} u_{ii}$, it follows that $\det(U_{11}) = \prod_{i=1}^{k} u_{ii} \neq 0$ and hence A_k is invertible.

To show the other direction we use the construction above of the LU factorization. For Algorithm 4.7 to succeed we need to have $a_{kk}^{(k-1)} \neq 0$. Suppose that the first $k-1$ steps of Algorithm 4.7 have succeeded (that is, $a_{11} \neq 0, a_{22}^{(1)} \neq 0 \ldots, a_{k-1,k-1}^{(k-2)} \neq 0$). Then

$$A_k = \begin{pmatrix} 1 & & & \\ l_{21} & \ddots & & \\ \vdots & \ddots & 1 & \\ l_{k1} & \cdots & l_{k,k-1} & 1 \end{pmatrix} \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1k} \\ & a_{22}^{(1)} & \cdots & a_{2n}^{(1)} \\ & & \ddots & \vdots \\ & & & a_{kk}^{(k-1)} \end{pmatrix}.$$

Hence it follows from $0 \neq \det(A_k) = a_{11}a_{22}^{(1)} \cdots a_{kk}^{(k-1)}$ that $a_{kk}^{(k-1)} \neq 0$. Therefore the kth step of Algorithm 4.7 succeeds as well and the claim follows by induction. ∎

To come up with a reasonable implementation of Algorithm 4.7, the matrix-matrix multiplications need to be replaced, as an explicit multiplication would be much too expensive. Moreover, to save memory, we will operate directly on the matrix A instead of creating temporary matrices $A^{(1)}, A^{(2)}, \ldots$.

Algorithm 4.9 (*LU* factorization)
Input: Invertible matrix $A \in \mathbb{R}^{n\times n}$.
Output: Factors L, U of LU factorization $A = LU$.

Set $L := I_n$.
for $k = 1, \ldots, n-1$ **do**
  for $i = k+1, \ldots, n$ **do**
    $\ell_{ik} \leftarrow \dfrac{a_{ik}}{a_{kk}}$
    for $j = k+1, \ldots, n$ **do**
      $a_{ij} \leftarrow a_{ij} - \ell_{ik} a_{kj}$
    end for
  end for
end for
Set U to upper triangular part of A.

PYTHON

```
import numpy as np
def mylu(A):
    n = A.shape[0]
    L = np.eye(n)
    for k in range(n):
        L[k+1:n, k] = A[k+1:n, k] / A[k,k]
        A[k+1:n, k+1:n] = A[k+1:n, k+1:n]
                        - np.outer(L[k+1:n, k], A[k, k+1:n])
    U = np.triu(A)
    return L, U
```

The complexity of Algorithm 4.9 is given by

$$\sum_{k=1}^{n-1} \big(1 + 2(n-k)\big)(n-k) = \frac{2}{3}n^3 - \frac{1}{2}n^2 - \frac{1}{6}n = \frac{2}{3}n^3 + O(n^2) \text{ flops.}$$

In summary, a linear system $A\mathbf{x} = \mathbf{b}$ is solved with the following procedure.

Algorithm 4.10 (Solution of $A\mathbf{x} = \mathbf{b}$ with LU factorization)

1. Compute LU factorization of $A = LU$ with Algorithm 4.9.
2. Solve $L\mathbf{y} = \mathbf{b}$ by forward substitution (Algorithm 4.3).
3. Solve $U\mathbf{x} = \mathbf{y}$ by backward substitution (Algorithm 4.4).

Example 4.11 We apply Algorithm 4.10 to Example 4.2:

```
import numpy as np, scipy as sp
A = np.array([[-0.370,  0.050,  0.050,  0.070],
```

```
              [ 0.050, -0.116,  0.000,  0.050],
              [ 0.050,  0.000, -0.116,  0.050],
              [ 0.070,  0.050,  0.050, -0.202]])
b = np.array([[-2], [0], [0], [0]])
L, U = mylu(A)
y = sp.linalg.solve_triangular(L, b, lower=True, unit_diagonal=True)
x = sp.linalg.solve_triangular(U, y)
```

This produces the output

```
x =
    8.1172
    5.9893
    5.9893
    5.7779
```

Of course, we could also have obtained the solution by directly calling `linalg.solve` from NumPy or SciPy. ◇

4.3 LU factorization with pivoting

Algorithm 4.9 will clearly fail whenever it encounters a zero pivot element. For example,

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

has a zero diagonal entry in the first position and immediately leads to a division by zero in Algorithm 4.9, although A is an invertible matrix. This situation is easy to spot, but it is important to keep in mind that – except for the first step – the pivot elements are computed in the course of the algorithm and it is in general impossible to "see" whether a matrix might lead to zero pivot elements.

The situation for the matrix A above is easy to resolve: We simply exchange rows 1 and 3 before attempting to compute the LU factorization. This corresponds to the multiplication of A with a permutation matrix[12]:

$$\widetilde{A} = PA = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & -1 \\ 0 & 1 & 1 \end{pmatrix}, \quad \text{with} \quad P = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}.$$

The LU factorization of $\widetilde{A}$ is then given by

$$\widetilde{A} = PA = LU \quad \text{with} \quad L = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix}, \quad U = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & -1 \\ 0 & 0 & 2 \end{pmatrix}.$$

[12] A permutation matrix has exactly one entry 1 in each row and column, and is otherwise zero.

There is an even more important reason to perform such permutations. Consider a slight modification of A:

$$A = \begin{pmatrix} 10^{-16} & 1 & 1 \\ 0 & 1 & -1 \\ 1 & 0 & 0 \end{pmatrix}$$

Then Algorithm 4.9 does not fail and Python returns

```
L =
   1.0000e+00            0            0
            0   1.0000e+00            0
   1.0000e+16  -1.0000e+16   1.0000e+00
U =
   1.0000e-16   1.0000e+00   1.0000e+00
            0   1.0000e+00  -1.0000e+00
            0            0  -2.0000e+16
```

Moreover, solving the linear system for the right-hand side $b = [2,\ 2,\ 1]^T$ with these triangular factoriuation gives the "solution"

```
x =
     0
     2
     0
```

However, this is **totally wrong**; the exact solution is $x = [1, 2, 0]^T$ (up to roundoff error 10^{-16}). What happened? The very large entries in L and U gave rise to massive numerical cancellation, which eventually destroyed the numerical accuracy of the solution completely. The large entries are due to the very small pivot element 10^{-16} in the first step of the LU factorization. To avoid such small pivot elements, we perform **a row permutation in each step of the LU factorization such that the entry of largest magnitude in the active column below the diagonal becomes the pivot element.** The realization of this idea is illustrated by the following example.

Example 4.12 Let

$$A = \begin{pmatrix} 1 & 2 & 2 \\ 2 & -7 & 2 \\ 1 & 24 & 0 \end{pmatrix}.$$

According to the idea above, we exchange the first and second rows by a permutation:

$$\widetilde{A}^{(0)} := P_1 A = \begin{pmatrix} 2 & -7 & 2 \\ 1 & 2 & 2 \\ 1 & 24 & 0 \end{pmatrix}, \quad \text{with} \quad P_1 = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

We now apply the usual elimination step to $\widetilde{A}^{(0)}$, resulting in $\ell_{21} = 0.5$, $\ell_{31} = 0.5$, and

$$A^{(1)} = \begin{pmatrix} 2 & -7 & 2 \\ 0 & 5.5 & 1 \\ 0 & 27.5 & -1 \end{pmatrix}.$$

Since $a_{32}^{(1)}$ is larger than $a_{22}^{(1)}$, we have to exchange rows 2 and 3 before applying the next step of LU factorization:

$$\widetilde{A}^{(1)} := P_2 A^{(1)} = \begin{pmatrix} 2 & -7 & 2 \\ 0 & 27.5 & -1 \\ 0 & 5.5 & 1 \end{pmatrix}, \quad \text{with} \quad P_2 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}.$$

The usual elimination step applied to $\widetilde{A}^{(1)}$, results in $\ell_{32} = \frac{5.5}{27.5} = 0.2$, and

$$U = A^{(2)} = \begin{pmatrix} 2 & -7 & 2 \\ 0 & 27.5 & -1 \\ 0 & 0 & 1.2 \end{pmatrix},$$

Moreover,

$$L = \begin{pmatrix} 1 & 0 & 0 \\ 0.5 & 1 & 0 \\ 0.5 & 0.2 & 1 \end{pmatrix}.$$

It remains to collect the permutations:

$$P = P_2 P_1 = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix}.$$

This can and should be performed without explicit multiplication by applying the row exchange described by P_2 to the rows of P_1. In summary, we obtain the following **LU factorization with pivoting**:

$$PA = LU,$$

where P is a permutation matrix and L/U are lower/upper triangular matrices. $\diamond$

The extension of the procedure outlined in Example 4.12 to general matrices is given by the following algorithm.

Algorithm 4.13 (*LU* factorization with pivoting)
Input: Invertible matrix $A \in \mathbb{R}^{n\times n}$.
Output: Factors P, L, U of LU factorization with pivoting $PA = LU$.

for $k = 1, \dots, n-1$ **do**
  Search $i \in \{k, \dots, n\}$ such that $|a_{ik}| \geq |a_{i'k}|$ for all $i' \in \{k, \dots, n\}$
  Define P_k as permutation matrix that exchanges rows i and k.
  Exchange rows: $A \leftarrow P_k A$
  for $i = k+1, \dots, n$ **do**
    $a_{ik} \leftarrow \dfrac{a_{ik}}{a_{kk}}$
    for $j = k+1, \dots, n$ **do**
      $a_{ij} \leftarrow a_{ij} - a_{ik}a_{kj}$
    end for
  end for
end for
Set U to upper triangular part of A.
Set L to lower triangular part of A and diagonal entries 1.
Set $P := P_{n-1}P_{n-2}\cdots P_2 P_1$.

In PYTHON, Algorithm 4.13 is performed by the function

```
P,L,U = scipy.linalg.lu(A)
```

If implemented well, the cost of Algorithm 4.13 is essentially identical with Algorithm 4.9. An algorithm for solving a linear system $A\mathbf{x} = \mathbf{b}$ can be obtained from a slight modification of Algorithm 4.10:

1. Compute LU factorization with pivoting $PA = LU$ with Algorithm 4.13.
2. Set $\tilde{\mathbf{b}} := P\mathbf{b}$ and solve $L\mathbf{y} = \tilde{\mathbf{b}}$ by forward substitution (Algorithm 4.3).
3. Solve $U\mathbf{x} = \mathbf{y}$ by backward substitution (Algorithm 4.4).

4.4 Symmetric positive definite matrices*

Matrices from applications often have additional properties that facilitate the solution of the associated linear systems. A quite common property is positive definiteness.

Definition 4.14 *A symmetric matrix $A \in \mathbb{R}^{n\times n}$ is called*

1. **positive definite**, *if* $\mathbf{x}^\mathsf{T} A\mathbf{x} > 0 \qquad \forall \mathbf{x} \in \mathbb{R}^n \backslash \{\mathbf{0}\}$;
2. **positive semi-definite**, *if* $\mathbf{x}^\mathsf{T} A\mathbf{x} \geq 0 \qquad \forall \mathbf{x} \in \mathbb{R}^n$.

A symmetric matrix is positive definite (semi-definite) if all its eigenvalues are positive (non-negative). The following theorem collects some simpler *necessary conditions for positive definiteness.*

Theorem 4.15 *Let $A \in \mathbb{R}^{n\times n}$ be symmetric positive definite. Then the following statements hold:*

1. *A is invertible;*
2. *$a_{ii} > 0$ for all $i \in \{1, \ldots, n\}$;*
3. *$|a_{ij}| < \frac{1}{2}(a_{ii} + a_{jj})$ for $i \neq j$ and hence $\max_{ij} |a_{ij}| = \max_i a_{ii}$;*
4. *if $X \in \mathbb{R}^{n\times n}$ is invertible, then $X^\mathsf{T}AX$ is again symmetric positive definite.*

Setting $A = I$, it follows from Theorem 4.15.4 that $X^\mathsf{T}X$ is symmetric positive for every invertible matrix X. The converse is also true: every symmetric positive matrix can be written in the form $X^\mathsf{T}X$.

Theorem 4.16 *Let $A \in \mathbb{R}^{n\times n}$ be symmetric positive definite. Then there exists an upper triangular matrix $R \in \mathbb{R}^{n\times n}$ such that $A = R^\mathsf{T}R$.*

The factorization $A = R^\mathsf{T}R$ from Theorem 4.16, with an upper triangular matrix R having positive diagonal entries, is called **Cholesky factorization**. Note that this is a special case of an LU factorization $A = LU$, with $L = R^\mathsf{T}$ and $U = R$. In fact, the Cholesky factorization could be extracted from an LU factorization. However, the following algorithm is more direct and more efficient.

Algorithm 4.17 (Cholesky factorization)
Input: Symmetric positive matrix $A \in \mathbb{R}^{n\times n}$.
Output: Cholesky factor R such that $A = R^\mathsf{T}R$.

for $j = 1, \ldots, n$ **do**
  for $i = 1, \ldots, j-1$ **do**
    $r_{ij} = \left(a_{ij} - \sum_{k=1}^{i-1} r_{ki}r_{kj}\right)/r_{ii}$
  end for
  $r_{jj} = \sqrt{a_{jj} - \sum_{k=1}^{j-1} r_{kj}^2}$
end for

Algorithm 4.17 requires $n^3/3 + O(n^2)$ flops and is therefore half as expensive as computing the LU factorization. This is due to the fact that only one factor instead of two need to be computed.

Remark 4.18 *Algorithm 4.17 will break down if the matrix A is not positive definite. In this case, the expression $a_{jj} - \sum_{k=1}^{j-1} r_{kj}^2$ will become negative at one point of the algorithm and no (real) square root can be determined.*

Note that Algorithm 4.17 does not perform any pivoting/permutations. It turns out that this is not necessary; the Cholesky factorization of a symmetric positive definite matrix can always be computed in a numerically reliable way without pivoting.

Recap on norms

Before we continue, we will recall basic definitions and facts about vector and matrix norms, which will be important for the rest of this chapter and next chapter. A (vector) norm on $\mathbb{R}^n$ is a real-valued function $\|\cdot\| : \mathbb{R}^n \to \mathbb{R}$ that satisfies the following three axioms:

Triangle inequality: $\|\mathbf{x}+\mathbf{y}\| \leq \|\mathbf{x}\| + \|\mathbf{y}\|$ for all $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$;

Homogeneity: $\|\alpha\mathbf{x}\| \leq |\alpha|\|x\|$ for all $\alpha \in \mathbb{R}$, $\mathbf{x} \in \mathbb{R}^n$;

Positivity: $\|\mathbf{x}\| \geq 0$ for all $\mathbf{x} \in \mathbb{R}^n$ and $\|\mathbf{x}\| = 0$ if and only if $\mathbf{x} = 0$.

The most common examples are the Euclidean norm $\|\mathbf{x}\|_2 = \sqrt{x_1^2 + \cdots + x_n^2}$, the 1-norm $\|\mathbf{x}\|_1 = |x_1| + \cdots + |x_n|$, and the ∞-norm $\|\mathbf{x}\|_\infty = \max\{|x_i| : i = 1, \ldots, n\}$. On the other hand, the function $\|\mathbf{x}\|_0 = \#\{i : x_i \neq 0\}$, where $\#$ denotes the cardinality of a set, is *not* a norm.

A matrix norm on $\mathbb{R}^{m\times n}$ is a real-valued function $\|\cdot\| : \mathbb{R}^{m\times n} \to \mathbb{R}$ that satisfies the vector norm axioms on the vector space $\mathbb{R}^{m\times n} \cong \mathbb{R}^{m\cdot n}$:

Triangle inequality: $\|A+B\| \leq \|A\| + \|B\|$ for all $A, B \in \mathbb{R}^{m\times n}$;

Homogeneity: $\|\alpha A\| \leq |\alpha|\|A\|$ for all $\alpha \in \mathbb{R}$, $A \in \mathbb{R}^{m\times n}$;

Positivity: $\|A\| \geq 0$ for all $A \in \mathbb{R}^{m\times n}$ and $\|A\| = 0$ if and only if $A = 0$.

In order to be useful for the analysis of algorithms, a matrix norm should also be sub-multiplicative. More precisely, a family of matrix norms $\|\cdot\|$ on $\mathbb{R}^{m\times n}$, defined for all $m, n \in \mathbb{N}$, is called *sub-multiplicative* if

$$\|AB\| \leq \|A\|\|B\|, \quad \forall A \in \mathbb{R}^{m\times n}, B \in \mathbb{R}^{n\times p},\ m, n, p \in \mathbb{N}.$$

The most simple way to define a matrix norm is to just take a vector norm on $\mathbb{R}^{m\times n} \cong \mathbb{R}^{m\cdot n}$. For example, the Euclidean norm gives rise to the Frobenius norm

$$\|A\|_F = \Big(\sum_{i=1}^{n}\sum_{j=1}^{m} a_{ij}^2\Big)^{1/2}$$

One can show that the Frobenius norm is sub-multiplicative. In contrast, the maximum norm

$$\|A\|_{\max} = \max\{|a_{ij}| : i = 1, \ldots, n,\ j = 1, \ldots, m\}$$

is a matrix norm but it is *not* sub-multiplicative.

Another popular way to define a matrix norm on $\mathbb{R}^{m\times n}$ is the *operator norm* induced by a vector norm $\|\cdot\|$ on $\mathbb{R}^m, \mathbb{R}^n$:

$$\|A\| := \sup_{\substack{\mathbf{x}\in\mathbb{R}^n\\ \mathbf{x}\neq 0}} \frac{\|A\mathbf{x}\|}{\|\mathbf{x}\|} = \sup_{\substack{\mathbf{x}\in\mathbb{R}^n\\ \|\mathbf{x}\|=1}} \|A\mathbf{x}\|$$

One easily verifies the three matrix norm axioms. Additionally, any operator norm is sub-multiplicative because

$$\begin{aligned}\|AB\| &= \sup_{\substack{\mathbf{x}\in\mathbb{R}^n\\ \mathbf{x}\neq 0}} \frac{\|AB\mathbf{x}\|}{\|\mathbf{x}\|} = \sup_{\substack{\mathbf{x}\in\mathbb{R}^n\\ \mathbf{x}\neq 0}} \frac{\|AB\mathbf{x}\|}{\|\mathbf{x}\|}\frac{\|B\mathbf{x}\|}{\|B\mathbf{x}\|} = \sup_{\substack{\mathbf{x}\in\mathbb{R}^n\\ \mathbf{x}\neq 0}} \frac{\|AB\mathbf{x}\|}{\|B\mathbf{x}\|}\frac{\|B\mathbf{x}\|}{\|\mathbf{x}\|}\\ &\leq \sup_{\substack{\mathbf{x}\in\mathbb{R}^n\\ \mathbf{x}\neq 0}} \frac{\|AB\mathbf{x}\|}{\|B\mathbf{x}\|} \sup_{\substack{\mathbf{x}\in\mathbb{R}^n\\ \mathbf{x}\neq 0}} \frac{\|B\mathbf{x}\|}{\|\mathbf{x}\|} \leq \sup_{\substack{\mathbf{y}\in\mathbb{R}^n\\ \mathbf{y}\neq 0}} \frac{\|A\mathbf{y}\|}{\|\mathbf{y}\|} \cdot \sup_{\substack{\mathbf{x}\in\mathbb{R}^n\\ \mathbf{x}\neq 0}} \frac{\|B\mathbf{x}\|}{\|\mathbf{x}\|} = \|A\|\|B\|.\end{aligned}$$

The usefulness of an operator norm in practice also depends on how easy it is to determine the supremum. For the following three situations, the operator norm can be computed explicitly

Spectral norm: The operator norm induced by the Euclidean norm $\|\cdot\| \equiv \|\cdot\|_2$ is called spectral norm and satisfies $\|A\|_2 = \sigma_1(A)$, where $\sigma_1(\cdot)$ denotes the largest singular value of a matrix.

Maximum column sum: The operator norm induced by the 1-norm $\|\cdot\| \equiv \|\cdot\|_1$ can be shown to satisfy

$$\|A\|_1 = \max\Big\{\sum_{i=1}^{m} |a_{ij}| \colon j = 1,\ldots,n\Big\},$$

that is, the maximum 1-norm of the columns of A.

Maximum row sum: The operator norm induced by the ∞-norm $\|\cdot\| \equiv \|\cdot\|_\infty$ can be shown to satisfy

$$\|A\|_\infty = \max\Big\{\sum_{j=1}^{n} |a_{ij}| \colon i = 1,\ldots,m\Big\},$$

that is, the maximum 1-norm of the rows of A.

By definition, *any* operator norm satisfies $\|I_n\| = 1$, where I_n is the $n\times n$ identity matrix. This shows that the Frobenius norm cannot be expressed as an operator norm because $\|I_n\|_F = \sqrt{n}$.

4.5 Error analysis: Conditioning

Solving linear systems on the computer in finite precision arithmetic is subject to roundoff error. Already storing the matrix A and the vector $\mathbf{b}$ will usually cause

some error. To verify the accuracy of a *computed* (approximate) solution $\widehat{\mathbf{x}}$ one often computes the norm of the **residual** $\mathbf{r} = \mathbf{b} - A\widehat{\mathbf{x}}$. One major philosophy in numerical analysis is *backward error analysis*, to view a computed solution of a given linear system as the *exact solution of a perturbed linear system*. The norm of the residual corresponds to the minimal perturbation.

Theorem 4.19 *Let $\widehat{\mathbf{x}} \neq 0$ be an approximate solution of the linear system $A\mathbf{x} = \mathbf{b}$. Then*

$$\min\{\|\triangle A\|_2 : \ (A + \triangle A)\widehat{\mathbf{x}} = \mathbf{b}\} = \frac{\|\mathbf{r}\|_2}{\|\widehat{\mathbf{x}}\|_2},$$

where $\mathbf{r} = \mathbf{b} - A\widehat{\mathbf{x}}$.

Proof. The relation $(A + \triangle A)\widehat{\mathbf{x}} = \mathbf{b}$ yields

$$\|\triangle A\|_2\|\widehat{\mathbf{x}}\|_2 \geq \|\triangle A\widehat{\mathbf{x}}\|_2 = \|\mathbf{r}\|_2,$$

and hence $\min\{\|\triangle A\|_2 : \ (A + \triangle A)\widehat{\mathbf{x}} = \mathbf{b}\} \geq \|\mathbf{r}\|_2/\|\widehat{\mathbf{x}}\|_2$. To establish equality, we still need an admissible perturbation $\triangle_0 A$ with $\|\triangle_0 A\|_2 = \|\mathbf{r}\|_2/\|\widehat{\mathbf{x}}\|_2$. This is given by

$$\triangle_0 A = \frac{1}{\|\widehat{\mathbf{x}}\|_2^2}\mathbf{r}\,\widehat{\mathbf{x}}^{\mathsf{T}},$$

because $(A + \triangle_0 A)\widehat{\mathbf{x}} = A\widehat{\mathbf{x}} + \mathbf{r} = \mathbf{b}$. ∎

When storing the entries of A in double precision one causes an error of order $10^{-16}\|A\|_2$, which in turn leads to a residual norm $\|\mathbf{r}\|_2$ of order $10^{-16}\|A\|_2\|x\|_2$ even if the rest of the computations was carried out exactly. For this reason, the best we can reasonably hope from a numerical algorithm for solving linear system is that it achieves $\|r\|_2 \sim 10^{-16}\|A\|_2\|x\|_2$. Such algorithms are called *backward stable*. However, this does *not* necessarily imply a small error $\|\widehat{\mathbf{x}} - \mathbf{x}\|_2$ of the solution itself.

Example 4.20 Let $A = (a_{ij})$ with $a_{ij} = \dfrac{1}{i + j - 1}$ be the $n \times n$ **Hilbert matrix** and $b = (1, \ldots, 1)^{\mathsf{T}}$. We compute the solution of $A\mathbf{x} = \mathbf{b}$ with \ and compare it with the "exact" solution, which is computed in 100 decimal digits arithmetic, using the mpmath library for performing floating-point arithmetic with arbitrary precision.

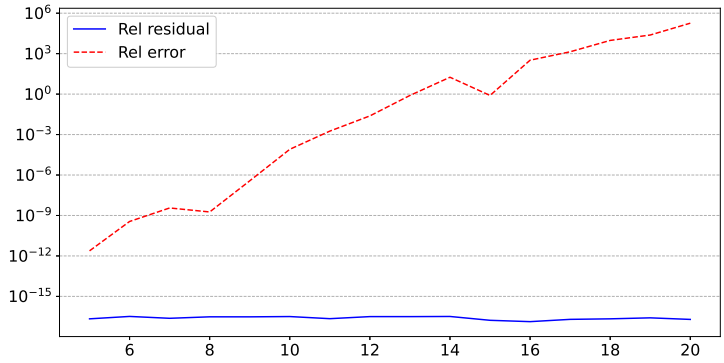
PYTHON

```
import matplotlib.pyplot as plt
import numpy as np, scipy.linalg as spla
from mpmath import hilbert, lu_solve, mp, ones

mp.dps = 100, ns = range(5, 21)
xrel, res = [], []
for n in ns:
    Anp, Amp = spla.hilbert(n), hilbert(n)
    rhsnp, rhsmp = np.ones(n), ones(n, 1)
    xnp, xmp = spla.solve(Anp, rhsnp), lu_solve(Amp, rhsmp)
    res.append(spla.norm(Anp @ xnp - rhsnp) / spla.norm(xnp))
```

```
    xrel.append(float(spla.norm(xnp - xmp)) / spla.norm(xnp))

plt.semilogy(ns, res, "b", ns, xrel, "r--")
plt.legend(["Rel residual", "Rel error"])
plt.grid(axis="y", which="major", linestyle="--")
plt.show()
```



The red dashed line shows the relative error of the solution $\widehat{\mathbf{x}}$ *computed in double precision* and the blue solid line shows the relative norm of the residual as n increases. The error of $\widehat{\mathbf{x}}$ grows rapidly, despite the fact that it is the solution of a very slightly perturbed linear system, according to Theorem 4.19. ◇

4.5.1 Condition of a function

To explain the phenomenon observed in Example 4.20 we will first consider the sensitivity analysis in a more abstract framework. A computation can be viewed as a function $f : \text{Input} \mapsto \text{Output}$. We assume that the admissible inputs are in a finite dimensional vector space V_i with norm $\|\cdot\|_{V_i}$. The Outputs are in a vector space V_o with norm $\|\cdot\|_{V_o}$. A computation is the evaluation $f(x) \in V_o$ for some admissible $x \in V_i$ (x can be a matrix; $V_i = \mathbb{R}^{n\times n}$). Because of roundoff error the representation of x is corrupted by an error $\triangle x$. We now want to understand how much this error is potentially increased when evaluating the function, that is, when considering $f(x+\triangle x)$ instead of $f(x)$. For this purpose we assume that f is defined and two times continuously differentiable in ab open ball containing $x \in V_i$. Using Taylor expansion, we have

$$x \mapsto f(x), \quad x + \triangle x \mapsto f(x + \triangle x) = f(x) + f'(x)\,\triangle x + O(\|\triangle x\|_{V_i}^2)$$

as $\triangle x \to 0$. The norm of the differential f' at x, that is,

$$\|f'(x)\|_{\mathcal{L}(V_i,V_o)} := \max\{\|f'(x)\,\triangle x\|_{V_o} : \|\triangle x\|_{V_i} = 1\}$$

measures the **absolute sensitivity** or *condition* of $f(x)$ with respect to a (small) perturbation $\triangle x$ of x.

Often, it is more reasonable to consider the **relative error**, which is independent of scaling of x or $f(x)$: For $f(x) \neq 0$, $x \neq 0$, $\triangle x \to 0$ we have

$$\frac{\|f(x+\triangle x)-f(x)\|_{V_o}}{\|f(x)\|_{V_o}} = \frac{\|f'(x)\triangle x\|_{V_o}+O(\|\triangle x\|_{V_i}^2)}{\|f(x)\|_{V_o}} \implies$$

$$\underbrace{\frac{\|f(x+\triangle x)-f(x)\|_{V_o}}{\|f(x)\|_{V_o}}}_{\text{rel. error in } f} \leq \Bigg(\frac{\|f'(x)\|_{\mathcal{L}(V_i,V_o)}}{\|f(x)\|_{V_o}}\,\|x\|_{V_i}\Bigg)\underbrace{\frac{\|\triangle x\|_{V_i}}{\|x\|_{V_i}}}_{\text{rel. error in } x} + O(\|\triangle x\|_{V_i}^2).$$

Definition 4.21 *The (relative)* **condition of the function** $x \mapsto f(x)$ *at* x *is*

$$\operatorname{cond}(f,x) := \frac{\|f'(x)\|_{\mathcal{L}(V_i,V_o)}}{\|f(x)\|_{V_o}}\,\|x\|_{V_i}\,.$$

As an example, consider the subtraction of two numbers $a, b \in \mathbb{R}$. Then $V_i = \mathbb{R}^2$ (with norm $\|\cdot\|_1$) and $V_o = \mathbb{R}$. Moreover, $f : V_i \to V_o$ with $f : (a,b) \mapsto a-b$ and

$$\|f'(a_0,b_0)\|_{\mathcal{L}(V_i,V_o)} = \left\|\left(\frac{\partial f}{\partial a}|_{(a_0,b_0)}, \frac{\partial f}{\partial b}|_{(a_0,b_0)}\right)\right\|_\infty = \|(1,-1)\|_\infty = 1.$$

Hence the relative condition of f at (a_0,b_0) is

$$\operatorname{cond}(f,(a_0,b_0)) = \frac{|a_0|+|b_0|}{|a_0-b_0|}.$$

As expected, this is large when $a_0 \approx b_0$.

4.5.2 Condition number of a matrix

In the following, $\|\cdot\| : \mathbb{R}^{n\times n} \to \mathbb{R}$ denotes the operator norm induced by a vector norm $\|\cdot\| : \mathbb{R}^n \to \mathbb{R}$ (e.g., the spectral norm induced by the Euclidean norm). The following proposition studies the relative condition of matrix inversion, that is, the $f(X) = X^{-1}$ at an invertible matrix $X = A$.

Proposition 4.22 (Sensitivity of matrix inversion) *Let $A \in \mathbb{R}^{n\times n}$ be invertible. If $\|A^{-1}\triangle A\| < 1$ then $A+\triangle A$ is invertible and*

$$\underbrace{\frac{\|(A+\triangle A)^{-1}-A^{-1}\|}{\|A^{-1}\|}}_{\text{rel. error in } A^{-1}} \leq \frac{\|A\|\,\|A^{-1}\|}{1-\|A^{-1}\triangle A\|}\underbrace{\frac{\|\triangle A\|}{\|A\|}}_{\text{rel. error in } A}.$$

Proof. For $\|X\| < 1$, the so called **Neumann series**

$$\sum_{k=0}^{\infty} X^k = I + X + X^2 + \cdots = I + Y, \quad Y := \sum_{k=1}^{\infty} X^k \tag{4.9}$$

converges because $\| \sum_{k=0}^{\infty} X^k \| \leq \sum_{k=0}^{\infty} \|X\|^k = 1/(1 - \|X\|)$. This series is the inverse of $I - X$ because

$$(I - X)(I + Y) = (I - X)\Big(\sum_{k=0}^{\infty} X^k\Big) = \sum_{k=0}^{\infty} X^k - \sum_{k=1}^{\infty} X^k = I.$$

Moreover,

$$\|Y\| = \Big\|X \sum_{k=0}^{\infty} X^k\Big\| \leq \frac{\|X\|}{1 - \|X\|}.$$

Now we can write

$$A + \triangle A = A(I + A^{-1}\triangle A).$$

Setting $X = -A^{-1}\triangle A$, the discussion above shows that $(I + A^{-1}\triangle A)$ is invertible when $\|A^{-1}\triangle A\| < 1$ with the inverse given by $I + Y$ for some Y with $\|Y\| \leq \|A^{-1}\triangle A\|/(1 - \|A^{-1}\triangle A\|)$. In turn $A + \triangle A$ is also invertible and

$$\|(A+\triangle A)^{-1} - A^{-1}\| = \|(I+Y)A^{-1} - A^{-1} \leq \|Y\|\|A^{-1}\| \leq \|A^{-1}\|^2 \frac{\|\triangle A\|}{1 - \|A^{-1}\triangle A\|}.$$

□

Proposition 4.22 shows that $\|A\|\,\|A^{-1}\|$ governs the condition of matrix inversion. This quantity is called **condition number** of A:

$$\kappa(A) := \|A\|\,\|A^{-1}\|. \tag{4.10}$$

Note that the condition number depends on the operator norm $\|\cdot\|$. We write $\kappa_2(A) = \|A\|_2\,\|A^{-1}\|_2$.

PYTHON

```
from numpy import linalg as LA
LA.cond(A) # condition number of A in the spectral norm
LA.cond(A,p) # condition number of A in the p-norm with p = 1,2,inf
LA.cond(A,'fro') # condition number of A in the Frobenius norm
```

4.5.3 Sensitivity of linear systems

As a direct consequence of Proposition 4.22 we obtain the following result on the sensitivity of the solution of a linear system to perturbations in A and $\mathbf{b}$.

Theorem 4.23 *Let $A \in \mathbb{R}^{n\times n}$ be invertible and consider $\triangle A \in \mathbb{R}^{n\times n}$ satisfying $\|A^{-1}\triangle A\| < 1$. Then*

$$(A + \triangle A)\widehat{\mathbf{x}} = \mathbf{b} + \triangle\mathbf{b} \tag{4.11}$$

has a unique solution and

$$\frac{\|\widehat{\mathbf{x}} - \mathbf{x}\|}{\|\mathbf{x}\|} \leq \frac{\kappa(A)}{1 - \|A^{-1}\triangle A\|}\Big(\frac{\|\triangle A\|}{\|A\|} + \frac{\|\triangle\mathbf{b}\|}{\|\mathbf{b}\|}\Big). \tag{4.12}$$

Proof. Using the notation from the proof of Proposition 4.22 we have

$$\widehat{\mathbf{x}} = (A + \triangle A)^{-1}(\mathbf{b} + \triangle\mathbf{b}) = (I + Y)(\mathbf{x} + A^{-1}\triangle\mathbf{b}).$$

Hence, $\widehat{\mathbf{x}} - \mathbf{x} = Y\mathbf{x} + (I + Y)A^{-1}\triangle\mathbf{b}$ and, in turn,

$$\begin{aligned}
\|\widehat{\mathbf{x}} - \mathbf{x}\| &\leq \|Y\|\,\|\mathbf{x}\| + (1 + \|Y\|)\|A^{-1}\|\|\triangle\mathbf{b}\| \\
&\leq \frac{\|A^{-1}\triangle A\|}{1 - \|A^{-1}\triangle A\|}\|\mathbf{x}\| + \frac{1}{1 - \|A^{-1}\triangle A\|}\|A^{-1}\|\|\triangle\mathbf{b}\| \\
&\leq \frac{\kappa(A)}{1 - \|A^{-1}\triangle A\|}\Big(\frac{\|\triangle A\|}{\|A\|} + \frac{\|\triangle b\|}{\|A\|\|\mathbf{x}\|}\Big)\|\mathbf{x}\|
\end{aligned}$$

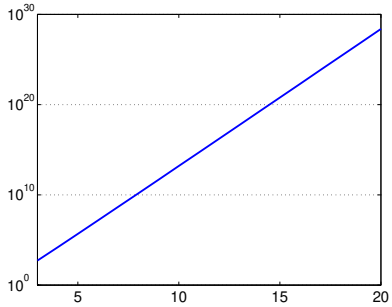
The proof is completed using $\|A\|\|\mathbf{x}\| \geq \|\mathbf{b}\|$. ∎

Remark 4.24 *The inequality* (4.12) *can be weakened to the asymptotic bound*

$$\frac{\|\widehat{\mathbf{x}} - \mathbf{x}\|}{\|\mathbf{x}\|} \leq \kappa(A)\,\Big(\frac{\|\triangle A\|}{\|A\|} + \frac{\|\triangle\mathbf{b}\|}{\|\mathbf{b}\|}\Big) + O(\epsilon^2)$$

for $\epsilon = \max\{\|\triangle A\|, \|\triangle\mathbf{b}\|\} \to 0$.

This analysis explains the strong growth of the error of $\widehat{\mathbf{x}}$ in Example 4.20. As n increases, the condition number of the Hilbert matrix increases exponentially fast.[13] The following figure shows the condition number for the spectral norm vs n.



[13] See Beckermann, B. The condition number of real Vandermonde, Krylov and positive definite Hankel matrices. Numer. Math. 85 (2000), no. 4, 553–577.

4.6 Error analysis: Solution by LU factorization*

We have now understood the effect of error in the data (matrix A and vector b) on the quality of the solution. This effect is independent of any algorithm; in fact, no algorithm can reasonably undo the effect of rounding A and b. Therefore, the best we can hope for is that our algorithm does not introduce a much larger error. This leads to the idea of *backward error analysis*; relate the error conducted during the algorithm back to the original data and hope that it is not much larger than what happened due to rounding anyway.

In the following, we will provide the analysis of the forward/backward substitution step and only state the result for the solution via the LU decomposition. In the following lemma, $|A|$ will denote the matrix obtained from A by replacing each entry by its absolute value. Comparison between matrices is understood elementwise. Also, we recall the quantity $\gamma_n \equiv \gamma_n(\mathbb{F}) := \frac{nu(\mathbb{F})}{1-nu(\mathbb{F})} = nu(\mathbb{F}) + O(u(\mathbb{F})^2)$ defined in Lemma 1.16.

Lemma 4.25 *Let $\widehat{\mathbf{x}}$ denote the solution computed by Algorithm 4.3 in floating point arithmetic $\mathbb{F}$. Then there exists a matrix $\triangle L$ with*

$$|\triangle L| \leq \gamma_n |L|,$$

such that

$$(L + \triangle L)\widehat{\mathbf{x}} = \mathbf{b}.$$

Proof. To simplify the notation, we let θ_i denote an undetermined generic variable satisfying $|\theta_i| \leq \gamma_i$. Using the models (1.5) and (1.6) of floating point arithmetic, the first loop of Algorithm 4.3 satisfies

$$\ell_{11}(1+\theta_1)\widehat{x}_1 = b_1,$$

and the second loop satisfies

$$\ell_{22}(1+\theta_2)\widehat{x}_2 = b_2 - \ell_{21}\widehat{x}_1(1+\theta_1)\ .$$

More generally, in the ith loop we have

$$\ell_{ii}(1+\theta_2)\widehat{x}_i = b_i - \ell_{i1}\widehat{x}_1(1+\theta_{i-1}) - \ell_{i2}\widehat{x}_2(1+\theta_{i-2}) - \cdots - \ell_{i,i-1}\widehat{x}_{i-1}(1+\theta_1)\ .$$

Setting $\triangle\ell_{ii} := \ell_{ii}\theta_2$ and $\triangle\ell_{ik} := \ell_{ik}\theta_{i-k}$ yields the result. ∎

An immediate consequence of the result of Lemma 4.25 is that

$$\|\triangle L\|_2 \leq c_L u \|L\|_2$$

for some constant c_L mildly growing with n. An analogous result holds, of course, for Algorithm 4.4. More importantly, this also holds for $Ax = b$; the solution $\hat{x}$

computed by the LU factorization (with or without pivoting) satisfies $(A+\triangle A)\hat{x} = b$ for some $\triangle A$ with

$$\|\triangle A\|_2 \leq c_A u \|L\|_2 \|U\|_2$$

for some constant c_A mildly growing with n; see [3]. It is important to observe that $\|L\|_2\|U\|_2$ can be significantly larger than $\|A\|$. When using pivoting (that is, Algorithm 4.13) then all entries of L are bounded by one in magnitude, while the norm of U is critically determined whether the algorithm encounters small pivot elements. Even with pivoting, there are examples for which the pivot elements can become very small (much smaller than $\|A^{-1}\|$) but this is a rare event; see [3] for more details.

Chapter 5

Linear Systems – Large Matrices

In practice, one often encounters linear systems with large sparse matrices. It is not uncommon to meet a $100,000 \times 100,000$ matrix with only about 10^6 nonzero entries. When A is tridiagonal (or, more generally, banded) it was already discussed in the exercises that the solution of such large linear systems is still feasible through an LU factorization. However, in practice one often encounters more complicated sparsity patterns for which it is not possible to carry out the LU factorization without excessive cost. In the following, we discuss several iterative methods that only involve matrix-vector products with sparse matrices.

5.1 Jacobi and Gauss-Seidel methods

The Jacobi method is the simplest iterative method for $A\mathbf{x} = \mathbf{b}$. Its idea consists of considering the jth equation and considering all but the jth variable fixed. This computation is performed repeatedly for $j = 1, \ldots, n$. For example, for $n = 3$, this corresponds to rewriting $A\mathbf{x} = \mathbf{b}$ as

$$\begin{aligned} x_1 &= (b_1 - a_{12}x_2 - a_{13}x_3)/a_{11} \\ x_2 &= (b_2 - a_{21}x_1 - a_{23}x_3)/a_{22} \\ x_3 &= (b_3 - a_{31}x_1 - a_{32}x_2)/a_{33}\,. \end{aligned}$$

Let $\mathbf{x}^{(0)}$ be given[14] then the Jacobi method is the recursion defined by

$$x_i^{(k+1)} := \left(b_i - \sum_{j=1}^{i-1} a_{ij}\, x_j^{(k)} - \sum_{j=i+1}^{n} a_{ij}\, x_j^{(k)} \right) \Big/ a_{ii}, \quad i = 1, \ldots, n. \tag{5.1}$$

One obvious possibility for improvement of (5.1) is to use the newest available

[14] The starting vector $\mathbf{x}^{(0)}$ is often set to zero or chosen randomly.

information, that is to use $x_j^{(k+1)}$ instead of $x_j^{(k)}$ for $j < i$:

$$x_i^{(k+1)} := \left(b_i - \sum_{j=1}^{i-1} a_{ij}\, x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij}\, x_j^{(k)} \right) \Big/ a_{ii}\,. \tag{5.2}$$

◇

For the purpose of analysis, it is better to rewrite (5.1) and (5.2) in the form of matrix operations. We write

$$A = L + D + U \tag{5.3}$$

where

$$D = \begin{pmatrix} a_{11} & & 0 \\ & \ddots & \\ 0 & & a_{nn} \end{pmatrix},\ L = \begin{pmatrix} 0 & & & 0 \\ a_{21} & \ddots & & \\ & & \ddots & \\ a_{n1} & & a_{n,n-1} & 0 \end{pmatrix},\ U = \begin{pmatrix} 0 & a_{12} & & a_{1n} \\ & \ddots & & \vdots \\ & & \ddots & a_{n-1,n} \\ & 0 & & 0 \end{pmatrix}.$$

Jacobi method: From (5.1) it follows that

$$\begin{aligned} D\mathbf{x}^{(k+1)} &= \mathbf{b} - L\mathbf{x}^{(k)} - U\mathbf{x}^{(k)} \iff \\ \mathbf{x}^{(k+1)} &= D^{-1}\big(\mathbf{b} - L\mathbf{x}^{(k)} - U\mathbf{x}^{(k)}\big) \\ &= \underbrace{-D^{-1}(L+U)}_{B^J}\ \mathbf{x}^{(k)} + D^{-1}\,\mathbf{b} \\ &= \qquad B^J \qquad\ \mathbf{x}^{(k)} + \mathbf{f}\,. \end{aligned} \tag{5.4}$$

Gauss-Seidel method: From (5.2) it follows that

$$\begin{aligned} & \mathbf{x}^{(k+1)} = D^{-1}\big(\mathbf{b} - L\mathbf{x}^{(k+1)} - U\mathbf{x}^{(k)}\big) && \iff \\ (D+L)\ & \mathbf{x}^{(k+1)} = -U\mathbf{x}^{(k)} + \mathbf{b} && \iff \\ & \mathbf{x}^{(k+1)} = -(D+L)^{-1}\,U\mathbf{x}^{(k)} + (D+L)^{-1}\mathbf{b} && \iff \\ & \mathbf{x}^{(k+1)} = B^{\mathrm{GS}}\,\mathbf{x}^{(k)} + \mathbf{f} \end{aligned} \tag{5.5}$$

with

$$B^{\mathrm{GS}} = -(D+L)^{-1}\,U, \quad \mathbf{f} = (D+L)^{-1}\,\mathbf{b}.$$

5.2 Splitting methods

Both, the Jacobi and Gauss-Seidel methods are splitting methods. Consider some splitting of A:

$$A = P - N, \qquad P, N \in \mathbb{R}^{n\times n}, \tag{5.6}$$

where P is usually called *preconditioner* and it is assumed that it is relatively easy to solve linear systems with P. Given such a splitting, we reformulate $A\mathbf{x} = \mathbf{b}$ as the fixed point equation

$$\mathbf{x} = B\mathbf{x} + \mathbf{f}, \quad B := P^{-1}N. \tag{5.7}$$

The corresponding fixed point iteration is given by

$$\mathbf{x}^{(k+1)} = B\mathbf{x}^{(k)} + \mathbf{f}, \tag{5.8}$$

with $\mathbf{f} = P^{-1}\mathbf{b}$. Equivalently, this corresponds to solving the linear system

$$P\mathbf{x}^{(k+1)} = N\mathbf{x}^{(k)} + \mathbf{b}$$

in each step.

It is easy to check that (5.8) corresponds to the Jacobi and Gauss-Seidel methods when choosing $P = D$ (diagonal preconditioner) and $P = D + L$, respectively.

For analyzing the convergence of (5.8), we let $\rho(A)$ denote the spectral radius of A, that is,

$$\rho(A) := \max\{|\lambda| \colon \lambda \in \mathbb{C} \text{ is an eigenvalue of } A\}.$$

Lemma 5.1 *Let $A \in \mathbb{R}^{n\times n}$. Then $A^k \to 0$ for $k \to \infty$ if and only if $\rho(A) < 1$.*

Proof. EFY. □

By subtracting $\mathbf{x} = B\mathbf{x} + \mathbf{f}$ from (5.8), we obtain the error recurrence

$$\mathbf{e}^{(k+1)} = B\mathbf{e}^{(k)}, \qquad k = 0, 1, 2, ..., \qquad \text{where} \quad \mathbf{e}^{(k)} := \mathbf{x}^{(k)} - \mathbf{x}. \tag{5.9}$$

We have $\mathbf{x}^{(k)} \to \mathbf{x}$ if and only if $\|\mathbf{e}^{(k)}\| = \|\mathbf{x}^{(k)} - \mathbf{x}\| \to 0$. Because of

$$\mathbf{e}^{(k)} = B\mathbf{e}^{(k-1)} = B^2\mathbf{e}^{(k-2)} = \ldots = B^k\mathbf{e}^{(0)}$$

we obtain convergence for *every* starting vector $\mathbf{x}^{(0)} \in \mathbb{R}^n$, if $B^k \to 0$ for $k \to \infty$. By Lemma 5.1, this holds if and only if the spectral radius $\rho(B)$ is smaller than 1.

Proving statements about the spectral radius is usually very difficult. Instead one uses the well-known fact that the spectral radius is bounded by any operator norm. We will illustrate this principle by showing that the Jacobi method converges for strictly diagonally dominant matrices.

Definition 5.2 *A matrix $A \in \mathbb{R}^{n\times n}$ is called* **strictly diagonally dominant by rows** *if*

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

and **strictly diagonally dominant by columns** *if*

$$|a_{ii}| > \sum_{\substack{j=1\\ j\neq i}}^{n} |a_{ji}|, \quad i = 1, \ldots, n\,.$$

Theorem 5.3 *Let A be strictly diagonally dominant by rows or columns. Then the Jacobi method converges.*

Proof. We will prove the statement when A is strictly diagonally dominant by rows; the column case is handled analogously. We recall that the iteration matrix for the Jacobi method is given by $B^J = -D^{-1}(L+U)$ and hence

$$\|B^J\|_\infty = \max_{i=1,\dots,n} \sum_{\substack{j=1 \\ j\neq i}}^{n} |a_{ij}| \,/\, |a_{ii}| < 1\,.$$

By the discussion above, this implies $\rho(B^J) \leq \|B^J\|_\infty < 1$ and hence the Jacobi method converges. □

Theorem 5.3 also holds for the Gauss-Seidel method but the proof is more difficult. Moreover, one can show that the Gauss-Seidel method converges for every symmetric positive definite matrix A.

5.3 Richardson method

The Richardson method for approximating the solution of $A\mathbf{x} = \mathbf{b}$ takes the form

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha\, P^{-1}\, \mathbf{r}^{(k)}\,, \tag{5.10}$$

where $\alpha > 0$ is an acceleration parameter, P is a preconditioner (that is easy to solve linear systems with), and $\mathbf{r}^{(k)}$ is the residual $\mathbf{x}^{(k)}$ defined by

$$\mathbf{r}^{(k)} = \mathbf{b} - A\mathbf{x}^{(k)}. \tag{5.11}$$

We again get an error recurrence of the form (5.9), with the iteration matrix $B = I - \alpha P^{-1}A$. Note that the Jacobi and Gauss-Seidel methods can be viewed as special cases of the Richardson method *if* $\alpha = 1$. However, in contrast to the Jacobi and Gauss-Seidel methods, the Richardson method can always be made convergent by choosing α appropriately. When A, P are symmetric positive definite then $P^{-1}A$ has positive real eigenvalues (Proof EFY) and the following result can be established.

Theorem 5.4 *Given an invertible preconditioner P, assume that $P^{-1}A$ has positive real eigenvalues*

$$\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_n > 0\,.$$

Then the Richardson method (5.10) *converges if and only if* $0 < \alpha < 2/\lambda_1$.
The choice $\alpha = \alpha_{\text{opt}} := 2/(\lambda_1 + \lambda_n)$ *minimizes the spectral radius of B, that is,*

$$\rho_{\text{opt}} = \min_{\alpha>0} |\,\rho(B_\alpha)| = \frac{\lambda_1 - \lambda_n}{\lambda_1 + \lambda_n}\,. \tag{5.12}$$

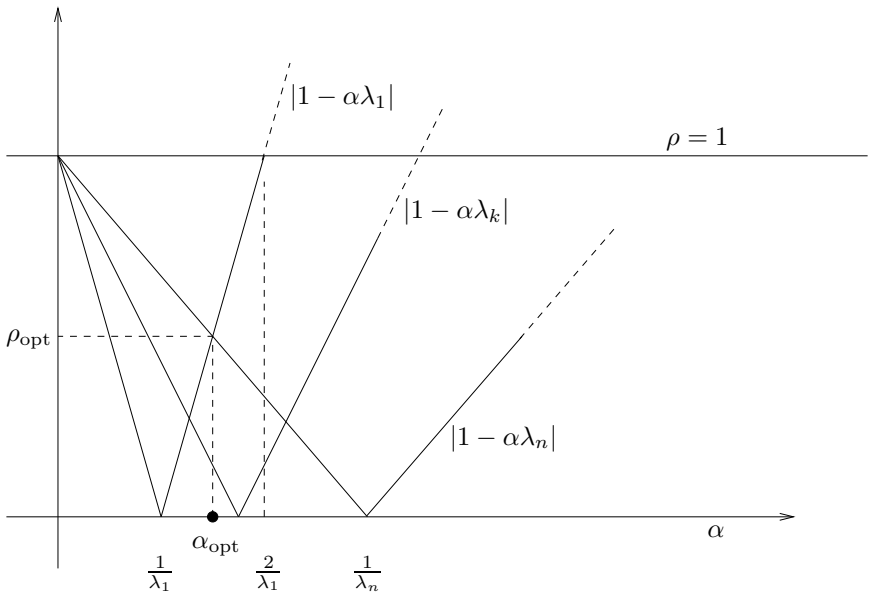


Figure 5.1: Spectral radius of B_α as a function of the eigenvalues of $P^{-1}A$.

Proof. The eigenvalues of B_α are given by $\lambda_i(B_\alpha) = 1 - \alpha\lambda_i$ for $i = 1, \ldots, n$. Hence, (5.10) converges if and only if $|\lambda_i(B_\alpha)| < 1$ for $i = 1, \ldots, n$, or, equivalently, if $0 < \alpha < 2/\lambda_1$. It follows (see Figure 5.1) that $\rho(B_\alpha)$ is minimal for $1-\alpha\lambda_n = \alpha\lambda_1 - 1$, which is satisfied when choosing $\alpha = 2/(\lambda_1 + \lambda_n)$. ∎

Remark 5.5 When A, P are symmetric positive definite we set $\kappa_\lambda(P^{-1}A) := \lambda_1/\lambda_n$. For $P = I$, this is the standard condition number $\kappa_2(A)$ of A. Otherwise, $\kappa_\lambda(P^{-1}A) = \kappa_2(P^{-1/2}AP^{-1/2})$ (proof EFY). By this definition,

$$\rho_{\text{opt}} = \frac{\lambda_1/\lambda_n - 1}{\lambda_1/\lambda_n + 1} = \frac{\kappa_\lambda(P^{-1}A) - 1}{\kappa_\lambda(P^{-1}A) + 1} \tag{5.13}$$

It follows that the convergence rate of the Richardson method depends only on $\kappa_\lambda(P^{-1}A)$.

This explains the wording "preconditioner" for P. The construction of cheap and effective preconditioners is an art by itself. First choices are D (the diagonal part of A) or incomplete LU/Cholesky decompositions of A.

5.4 Gradient method

The optimal choice of α in the Richardson method depends on the eigenvalues of A, which are usually unknown (and can be more expensive to compute than solving the linear system!). We will now take an "optimization perspective" of the Richardson

method for symmetric positive definite A (with $P = I$ for simplicity), which allows choose α optimally in each step, without knowledge of the eigenvalues of A.

Theorem 5.6 *Let A be symmetric positive definite. Then $\mathbf{x}$ is the solution of $A\mathbf{x} = \mathbf{b}$ if and only if it solves the optimization problem*

$$\mathbf{x} = \arg\min_{\mathbf{y}\in\mathbb{R}^n} \Phi(\mathbf{y}), \quad \textit{with} \quad \Phi(\mathbf{y}) := \frac{1}{2}\mathbf{y}^\top A\mathbf{y} - \mathbf{y}^\top \mathbf{b}. \tag{5.14}$$

Proof. For $\triangle\mathbf{y} \in \mathbb{R}^n$ we consider

$$\begin{aligned} \Phi(\mathbf{y}+\triangle\mathbf{y}) - \Phi(\mathbf{y}) &= \frac{1}{2}\triangle\mathbf{y}^\top A\mathbf{y} + \frac{1}{2}\mathbf{y}^\top A\triangle\mathbf{y} - \triangle\mathbf{y}^\top\mathbf{b} + O(\|\triangle\mathbf{y}\|_2^2) \\ &= \triangle\mathbf{y}^\top(A\mathbf{y}-\mathbf{b}) + O(\|\triangle\mathbf{y}\|_2^2). \end{aligned} \tag{5.15}$$

By the uniqueness of the Taylor expansion, it follows that $A\mathbf{y} - \mathbf{b}$ is the gradient of Φ at $\mathbf{y}$.

If $\mathbf{x}$ solves (5.14) then the gradient of Φ at $\mathbf{x}$ is zero, that is, $A\mathbf{x} - \mathbf{b}$.[15] In the other direction, if $\mathbf{x}$ is solution of the linear system, then

$$\begin{aligned} \Phi(\mathbf{y}) &= \Phi(\mathbf{x} + (\mathbf{y}-\mathbf{x})) \\ &= \frac{1}{2}\mathbf{x}^T A\mathbf{x} - \mathbf{x}^T\mathbf{b} + (\mathbf{y}-\mathbf{x})^T(A\mathbf{x}-\mathbf{b}) + \frac{1}{2}(\mathbf{y}-\mathbf{x})^T A(\mathbf{y}-\mathbf{x}) \\ &= \Phi(\mathbf{x}) + \frac{1}{2}(\mathbf{y}-\mathbf{x})^T A(\mathbf{y}-\mathbf{x}), \end{aligned}$$

and since $\frac{1}{2}(\mathbf{y}-\mathbf{x})^T A(\mathbf{y}-\mathbf{x}) \geq 0$, it follows that $\Phi(\mathbf{y}) \geq \Phi(\mathbf{x})$ and hence $\mathbf{x}$ minimizes Φ. ∎

The idea behind the gradient method is to proceed in every step in the negative direction of the gradient. Let $\mathbf{x}^{(k)}$ denote the kth iterate of the method. The $(k+1)$th iterate is obtained by setting

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha_k \mathbf{p}^{(k)}.$$

The search direction $\mathbf{p}^{(k)}$ is chosen such that it minimizes the first-order term $(\mathbf{p}^{(k)})^\top(A\mathbf{y}-\mathbf{b})$ in the Taylor expansion (5.15) among all vectors of the same 2-norm. By the Cauchy-Schwarz inequality the best choice is

$$\mathbf{p}^{(k)} = -\nabla\Phi\big(\mathbf{x}^{(k)}\big) = \mathbf{b} - A\mathbf{x}^{(k)} =: \mathbf{r}^{(k)}. \tag{5.16}$$

[15] One could complete the proof with the observation that f is strictly convex and, hence, a zero gradient characterizes the (global) minimum. We include a direct proof of the other direction for illustration.

The step size α_k is chosen such that $\Phi\big(\mathbf{x}^{(k)} + \alpha_k \mathbf{p}^{(k)}\big) = \Phi\big(\mathbf{x}^{(k+1)}\big)$ is minimal among all choices of α, that is,

$$\begin{aligned} 0 &= \frac{\mathrm{d}}{\mathrm{d}\alpha} \Phi\Big(\mathbf{x}^{(k)} + \alpha \mathbf{p}^{(k)}\Big)\Big|_{\alpha=\alpha_k} \\ &= \frac{\mathrm{d}}{\mathrm{d}\alpha}\left[\frac{1}{2}\big\langle \mathbf{x}^{(k)} + \alpha \mathbf{r}^{(k)}, A(\mathbf{x}^{(k)} + \alpha \mathbf{r}^{(k)})\big\rangle - \big\langle \mathbf{x}^{(k)} + \alpha \mathbf{r}^{(k)}, \mathbf{b}\big\rangle\right]\Big|_{\alpha=\alpha_k} \\ &= -\big\langle \mathbf{r}^{(k)}, \mathbf{r}^{(k)}\big\rangle + \alpha_k \big\langle \mathbf{r}^{(k)}, A\mathbf{r}^{(k)}\big\rangle, \end{aligned} \tag{5.17}$$

or, equivalently,

$$\alpha_k = \frac{\big\langle \mathbf{r}^{(k)}, \mathbf{r}^{(k)}\big\rangle}{\big\langle A\mathbf{r}^{(k)}, \mathbf{r}^{(k)}\big\rangle}.$$

In summary, the gradient method reads as follows: Given $\mathbf{x}^{(0)} \in \mathbb{R}^n$, let $\mathbf{r}^{(0)} = \mathbf{b} - A\mathbf{x}^{(0)}$. Then for all $k \geq 0$,

$$\begin{cases} \alpha_k = \frac{\big\langle \mathbf{r}^{(k)}, \mathbf{r}^{(k)}\big\rangle}{\big\langle A\mathbf{r}^{(k)}, \mathbf{r}^{(k)}\big\rangle}, \\ \mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha_k \mathbf{r}^{(k)}, \\ \mathbf{r}^{(k+1)} = \mathbf{r}^{(k)} - \alpha_k A\mathbf{r}^{(k)}. \end{cases}$$

The gradient method is thus a variant of the Richardson method (with $P = I$) for which the acceleration parameter is chosen adaptively in every iteration.

Figure 5.2 gives a visual representation of the gradient method. In particular, it seems that each direction $\mathbf{p}^{(k+1)} = \mathbf{r}^{(k+1)}$ for $k \geq 0$ is orthogonal to the descent direction at the previous iteration, $\mathbf{p}^{(k)} = \mathbf{r}^{(k)}$. Indeed, from equation (5.17),

$$0 = -\big\langle \mathbf{r}^{(k)}, \mathbf{r}^{(k)}\big\rangle + \alpha_k \big\langle \mathbf{r}^{(k)}, A\mathbf{r}^{(k)}\big\rangle = \big\langle \mathbf{r}^{(k)}, -\mathbf{r}^{(k)} + \alpha_k A\mathbf{r}^{(k)}\big\rangle = -\big\langle \mathbf{r}^{(k)}, \mathbf{r}^{(k+1)}\big\rangle. \tag{5.18}$$

Therefore, $\big\langle \mathbf{r}^{(k)}, \mathbf{r}^{(k+1)}\big\rangle = 0$. However, in general, $\big\langle \mathbf{r}^{(k)}, \mathbf{r}^{(k+2)}\big\rangle \neq 0$. Indeed, in Figure 5.2, any two directions $\mathbf{r}^{(k)}$ and $\mathbf{r}^{(k+2)}$ are parallel.

Using the Kantorovich inequality, it can be shown that the gradient method converges at the rate (5.13), the convergence rate of the Richardson iteration with optimally chosen α.

5.5 The method of conjugate gradients (CG)

For ill-conditioned matrices, the gradient method makes little progress because the search directions $\mathbf{p}^{(0)}, \mathbf{p}^{(1)}, \mathbf{p}^{(2)}, \ldots$ are too similar across several iterations. For $n = 2$ this can be nicely illustrated by the "zigzag" behavior of the gradient method.

We continue to assume that $A \in \mathbb{R}^{n\times n}$ is symmetric positive definite. The idea of the CG method is to chose search directions that are orthogonal to each other in the inner product induced by A: $\langle \mathbf{y}, \mathbf{z}\rangle_A := \mathbf{y}^\top A\mathbf{z} = \langle \mathbf{y}, A\mathbf{z}\rangle = \langle A\mathbf{y}, \mathbf{z}\rangle$. In the first

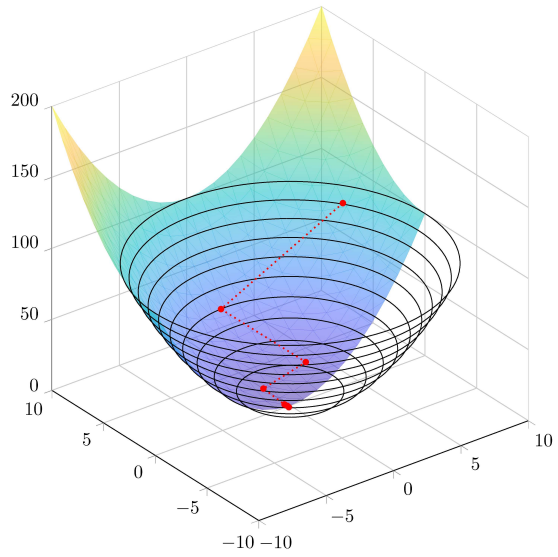


Figure 5.2: Illustration of the gradient method. In red are the descent directions $\mathbf{p}^{(k)}$. The red dots represent the intermediate solutions $\mathbf{x}^{(k)}$, $k \geq 0$. The ellipsoids represent some level curves of Φ; note that they are centered on the exact solution $\mathbf{x}$, and that the descent directions arrive tangentially towards them.

iterate, we still choose $\mathbf{p}^{(0)} = \mathbf{r}^{(0)} = \mathbf{b} - A\mathbf{x}^{(0)}$, as in the gradient method, and set $\mathbf{x}^{(1)} := \mathbf{x}^{(0)} + \alpha_0 \mathbf{p}^{(0)}$. To choose the next search direction, we set

$$\mathbf{p}^{(1)} = \mathbf{r}^{(1)} - \beta_0 \mathbf{p}^{(0)},$$

where the parameter β_0 is chosen such that $\mathbf{p}^{(1)}$ is orthogonal (conjugate) to $\mathbf{p}^{(0)}$. This corresponds to one step of the Gram-Schmidt process in the A-inner product:

$$\beta_0 = \frac{\langle \mathbf{r}^{(1)}, \mathbf{p}^{(0)} \rangle_A}{\langle \mathbf{p}^{(0)}, \mathbf{p}^{(0)} \rangle_A}.$$

More generally, we choose

$$\mathbf{p}^{(k+1)} = \mathbf{r}^{(k+1)} - \sum_{i=0}^{k} \frac{\langle \mathbf{r}^{(k+1)}, \mathbf{p}^{(i)} \rangle_A}{\langle \mathbf{p}^{(i)}, \mathbf{p}^{(i)} \rangle_A} \mathbf{p}^{(i)}, \tag{5.19}$$

As for the gradient method, for all $k \geq 0$, $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha_k \mathbf{p}^{(k)}$, and α_k is chosen so that $\Phi\left(\mathbf{x}^{(k)} + \alpha_k \mathbf{p}^{(k)}\right) = \Phi\left(\mathbf{x}^{(k+1)}\right)$ is minimal. By following the exact same steps as in (5.17), we obtain:

$$\alpha_k = \frac{\langle \mathbf{p}^{(k)}, \mathbf{r}^{(k)} \rangle}{\langle \mathbf{p}^{(k)}, A\mathbf{p}^{(k)} \rangle}.$$

The crucial observation, which makes CG efficient for larger k, is that most terms in (5.19) vanish.

Lemma 5.7 *With the notation introduced above, assume that $\alpha_i \neq 0$ for $i = 0, \ldots, k$. Then*

$$\langle \mathbf{r}^{(k+1)}, \mathbf{p}^{(0)} \rangle_A = \cdots = \langle \mathbf{r}^{(k+1)}, \mathbf{p}^{(k-1)} \rangle_A = 0.$$

Proof. We start by noting that $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha_k \mathbf{p}^{(k)}$ implies the residual recursion

$$\mathbf{r}^{(k+1)} = \mathbf{r}^{(k)} - \alpha_k A \mathbf{p}^{(k)}. \tag{5.20}$$

Step 1: Alternative formula for α_k. Using the definition of α_k, it follows from (5.20) that

$$\langle \mathbf{r}^{(k+1)}, \mathbf{p}^{(k)} \rangle = \langle \mathbf{r}^{(k)}, \mathbf{p}^{(k)} \rangle - \alpha_k \langle \mathbf{p}^{(k)}, \mathbf{p}^{(k)} \rangle_A = 0.$$

Because $\{\mathbf{p}^{(0)}, \ldots, \mathbf{p}^{(k)}\}$ is an A-orthogonal basis, we have for $i < k$ that

$$\langle \mathbf{r}^{(k+1)}, \mathbf{p}^{(i)} \rangle = \langle \mathbf{r}^{(k)}, \mathbf{p}^{(i)} \rangle + \alpha_k \langle \mathbf{p}^{(k)}, \mathbf{p}^{(i)} \rangle_A = \langle \mathbf{r}^{(k)}, \mathbf{p}^{(i)} \rangle,$$

and we can conclude inductively that

$$\langle \mathbf{r}^{(k+1)}, \mathbf{p}^{(i)} \rangle = 0, \quad i = 0, \ldots, k. \tag{5.21}$$

EFY: Show that this relation implies

$$\alpha_k = \frac{\langle \mathbf{r}^{(k)}, \mathbf{r}^{(k)} \rangle}{\langle \mathbf{p}^{(k)}, A\mathbf{p}^{(k)} \rangle}. \tag{5.22}$$

Step 2: Orthogonality of residuals. Note that

$$\langle \mathbf{r}^{(k)}, \mathbf{p}^{(k)} \rangle_A = \langle \mathbf{p}^{(k)}, \mathbf{p}^{(k)} \rangle_A.$$

For $k = 0$, this follows from $\mathbf{p}^{(0)} = \mathbf{r}^{(0)}$. For $k > 0$, this follows from applying the A-inner product with $p^{(k+1)}$ to both sides of (5.19), exploiting A-orthogonality, and shift $k + 1$ to k. Together with (5.22) it follows from (5.20) that

$$\langle \mathbf{r}^{(k+1)}, \mathbf{r}^{(k)} \rangle = \langle \mathbf{r}^{(k)}, \mathbf{r}^{(k)} \rangle - \alpha_k \langle \mathbf{r}^{(k)}, A\mathbf{p}^{(k)} \rangle = \langle \mathbf{r}^{(k)}, \mathbf{r}^{(k)} \rangle - \alpha_k \langle \mathbf{p}^{(k)}, A\mathbf{p}^{(k)} \rangle = 0.$$

For $i < k$, it follows from (5.19) that $\langle \mathbf{p}_k, \mathbf{r}_i \rangle_A = 0$. We conclude from (5.20) that

$$\langle \mathbf{r}^{(k+1)}, \mathbf{r}^{(i)} \rangle = 0, \quad i < k.$$

In other words, $\{\mathbf{r}^{(0)}, \ldots, \mathbf{r}^{(k+1)}\}$ is an orthogonal basis.

Step 3: Conclusion. Using once more (5.20) we obtain

$$\langle \mathbf{r}^{(k+1)}, \mathbf{p}^{(i)} \rangle_A = \langle \mathbf{r}^{(k+1)}, A\mathbf{p}^{(i)} \rangle = \frac{1}{\alpha_k} \langle \mathbf{r}^{(k+1)}, \mathbf{r}^{(i)} - \mathbf{r}^{(i+1)} \rangle = 0.$$

□

The result of Lemma 5.7 allows us to rewrite (5.19) as

$$\mathbf{p}^{(k+1)} = \mathbf{r}^{(k+1)} - \beta_k \mathbf{p}^{(k)}, \quad \beta_k = \frac{\langle \mathbf{r}^{(k+1)}, \mathbf{p}^{(k)} \rangle_A}{\langle \mathbf{p}^{(k)}, \mathbf{p}^{(k)} \rangle_A}.$$

In summary, the CG method reads as follows. Given $\mathbf{x}^{(0)} \in \mathbb{R}^n$, let $\mathbf{r}^{(0)} = \mathbf{b} - A\mathbf{x}^{(0)}$ and $\mathbf{p}^{(0)} = \mathbf{r}^{(0)}$. Then for all $k \geq 0$,

$$\begin{cases} \alpha_k = \frac{\langle \mathbf{p}^{(k)}, \mathbf{r}^{(k)} \rangle}{\langle \mathbf{p}^{(k)}, A\mathbf{p}^{(k)} \rangle}; \\ \mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha_k \mathbf{p}^{(k)}; \\ \mathbf{r}^{(k+1)} = \mathbf{r}^{(k)} - \alpha_k A\mathbf{p}^{(k)}; \\ \beta_k = \frac{\langle \mathbf{r}^{(k+1)}, A\mathbf{p}^{(k)} \rangle}{\langle \mathbf{p}^{(k)}, A\mathbf{p}^{(k)} \rangle}; \\ \mathbf{p}^{(k+1)} = \mathbf{r}^{(k+1)} - \beta_k \mathbf{p}^{(k)}. \end{cases}$$

Remark 5.8

- For CG to be well-defined, $\mathbf{p}^{(k)}$ has to be different from 0 at each iteration $k \geq 0$. But if for some $k \geq 0$, $\mathbf{p}^{(k)} = 0$, then $\mathbf{x}^{(k)} = \mathbf{x}$.
- At each iteration, the CG method considers a descent direction that is linearly independent from (since A-orthogonal to) the previous descent directions, and minimizes the quadratic form Φ in this new descent direction.

Theorem 5.9 *Let $A \in \mathbb{R}^{n \times n}$ be a symmetric positive definite matrix. Then the CG method yields after at most n iterations the exact solution (assuming exact arithmetic).*

Proof. As discussed above, in the unlikely case that $\mathbf{p}^{(k)} = 0$ for $k \leq n-1$ then CG has found already the exact solution. Otherwise, $\left\{\mathbf{p}^{(0)}, \mathbf{p}^{(1)}, \ldots, \mathbf{p}^{(n-1)}\right\}$ forms an A-orthogonal basis of $\mathbb{R}^n$. Because of (5.21), the vector $\mathbf{r}^{(n)}$ is orthogonal to the space $\operatorname{span}\left\{\mathbf{p}^{(0)}, \mathbf{p}^{(1)}, \ldots, \mathbf{p}^{(n-1)}\right\} = \mathbb{R}^n$. Consequently, $\mathbf{r}^{(n)} = \mathbf{0}$, which implies $\mathbf{x}^{(n)} = \mathbf{x}$. ∎

Theorem 5.9 is misleading because in practice one never wants to run n iterations of CG. Instead, one hopes to stop the method much earlier, as soon as the error is below a certain tolerance. The following theorem shows that the error of CG decreases quickly (and thus CG can be stopped after a few iterations) if the condition number of A is not too high.

Theorem 5.10 *Let $A \in \mathbb{R}^{n \times n}$ be symmetric positive definite and consider the linear system $A\mathbf{x} = \mathbf{b}$. For $k \geq 0$, let $\mathbf{e}^{(k)} := \mathbf{x}^{(k)} - \mathbf{x} \in \mathbb{R}^n$, where $\mathbf{x}^{(k)}$ is the k-th iterate of CG. Then,*

$$\left\|\mathbf{e}^{(k)}\right\|_A \leq 2\frac{C^k}{1 + C^{2k}}\left\|\mathbf{e}^{(0)}\right\|_A, \quad \textit{with} \quad C := \frac{\sqrt{\kappa_2(A)} - 1}{\sqrt{\kappa_2(A)} + 1}.$$

Proof. See Theorem 3.1.1 in [A. Greenbaum. Iterative methods for solving linear systems. SIAM, 1987]. ∎

Chapter 6

Regression and Least Squares

Regression refers to the problem of finding a function that best fits a set of data points. That is, given m data points $(t_1, y_1), (t_2, y_2), \ldots, (t_m, y_m)$, one is looking for a function f that best approximates $f(t_i) \approx y_i$, where "best" has to be mathematically defined and corresponds to an optimization problem. Unlike interpolation, the number of data points in *classical* regression is larger than the number of parameters in the function used to fit the data[16], and thus in general, one cannot (or does not even want to) obtain $f(t_i) = y_i$ for all $i = 1, \ldots, n$. Examples of regression problems are illustrated in Figure 6.1.

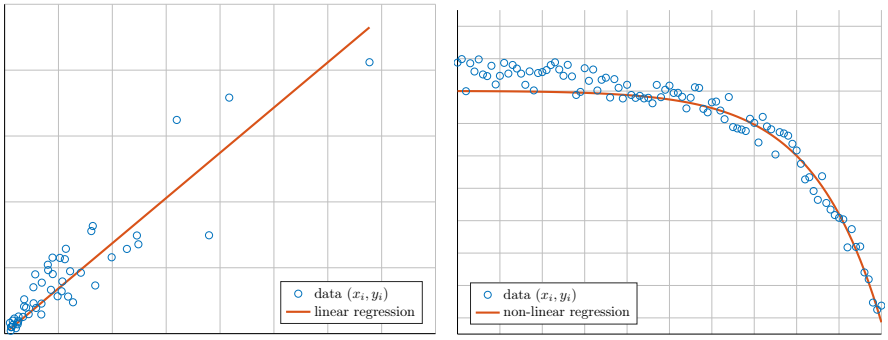
6.1 Regression: Introduction

To carry out regression, we need to specify two ingredients: The class of functions that is believed to fit the data well and the error that quantifies the fit.

6.1.1 Model function

In *parametric* regression, one considers a class of functions containing a model function that depends on n parameters. Typically, $n \ll m$ in order to avoid overfitting. Some examples are presented below; we call x_i, $i = 1, \ldots, n$, the parameters determined by fitting the function to the given data, t the problem variable, and g the model function.

[16] In regimes (statistical/machine learning) not covered in this lecture, the number of parameters is often (much) larger than the number of data points. In this case, the problem needs to be regularized, either explicitly by adding penalties/constraints or implicitly by terminating optimization methods early.



(a) Linear regression (b) Non-linear regression

Figure 6.1: Examples of regression.

- Linear Regression: g depends (affine) linearly on the *parameters*

$$\begin{aligned} g(t) &= x_1 + x_2 t \\ g(t) &= x_1 + x_2 t + x_3 t^2 \\ g(t) &= x_1 + x_2 x + \ldots + x_n t^{n-1} \\ g(t) &= x_1 e^t + x_2 e^{-t} \end{aligned} \tag{6.1}$$

- Nonlinear Regression:

$$\begin{aligned} g(t) &= x_1 + x_2 e^{x_3 t} && \text{asymptotic regression function} \\ g(t) &= \frac{x_1 t}{x_2 + t} && \text{Michaelis-Menten function} \\ g(t) &= \frac{x_1}{1 + x_2 e^{-x_3 t}} && \text{logistic function} \end{aligned} \tag{6.2}$$

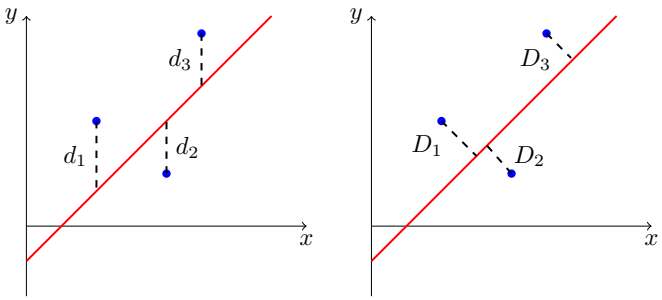
It is the application that determines *a priori* which model function to use, either by inspecting the data or by theoretical considerations. For example, in Figure 6.1a, the data looks to be approximately linear, so one would be inclined to use (6.1), while theoretical linear relationships can be given by Hooke's law, Ohm's law, or Fick's law for example. Nonlinear functions can also come from theoretical considerations; for example, the model function (6.2) is called logistic function (leading to logistic regression) because it is the solution of the logistic equation given by

$$\frac{\mathrm{d}g}{\mathrm{d}t} = rg(1-g) \quad \text{in } \mathbb{R}^+; \qquad f(0) = \alpha,$$

for some $r, \alpha > 0$.

6.1.2 Error function

The error function, also called objective or loss function, is used to assess the quality of the approximation of the data, which in turn drives the parameter selection.



(a) Vertical distance. (b) Euclidean distance.

Figure 6.2: Examples of different measures for the error.

For measuring the error at individual data points t_i, one can for example use vertical distances

$$d_i := |g(t_i) - y_i|\,, \quad i = 1, \dots, m, \tag{6.3}$$

as in the left plot of Figure 6.2, or Euclidean distances as in the right plot of Figure 6.2. One disadvantage of the Euclidean distance is that it depends on the parameters, which complicates fitting. For example, when considering the linear function (6.1), one can show that the Euclidean distance is

$$D_i := \frac{d_i}{\sqrt{1 + x_1^2}}, \quad i = 1, \dots, m. \tag{6.4}$$

In the following, we will therefore focus on vertical distances.

By choosing a norm on $\mathbb{R}^m$, one combines the m individual distances in a single quantity for measuring the error. The most common choices are:

1. Maximum error:

$$\max_{i=1,\dots,m} d_i = \max_{i=1,\dots,m} |g(t_i) - y_i| = \|\mathbf{g} - \mathbf{y}\|_\infty\,, \tag{6.5}$$

   where $\mathbf{g} := \big(g(t_1), \dots, g(t_m)\big)$ and $\mathbf{y} := (y_1, \dots, y_m)$.

2. Error in 1-norm:

$$\sum_{i=1}^{m} d_i = \sum_{i=1}^{m} |g(t_i) - y_i| = \|\mathbf{g} - \mathbf{y}\|_1\,. \tag{6.6}$$

3. (Squared) Error in 2-norm:

$$\sum_{i=1}^{m} d_i^2 = \sum_{i=1}^{m} |g(t_i) - y_i|^2 = \|\mathbf{g} - \mathbf{y}\|_2^2\,. \tag{6.7}$$

   When this norm is used, the resulting regression method is called least-squares. This is the most common choice because the error function has the advantage

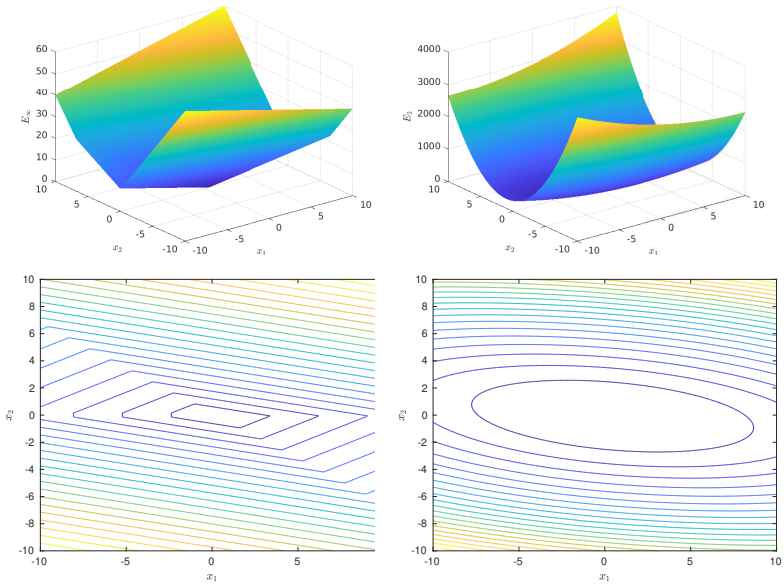


Figure 6.3: Examples of maximum (left) and 2-norm (right) error functions, respectively, both as surfaces and contour plots, for an example of linear regression.

of being differentiable if $g(t_i)$ is a differentiable function of the parameters. It also has a meaninful statistical intepretation.

Example 6.1 *Consider the linear regression defined in (6.1), and the maximum and 2-norm error functions. Figure 6.3 gives the contour plots of these error functions in an example. We can easily see the smoothness of the 2-norm while the maximum norm results in edges and corners. Note that the parameters that minimize E_2 are (usually) not equal to the ones that minimize E_∞.*

6.2 Linear least-squares

In this section, we will focus on the error function given by the 2-norm, defined in (6.7). We assume that the model function g depends linearly on the parameters to be fitted to the data, that is we can write

$$g(t) = x_1\phi_1(t) + x_2\phi_2(t) + \ldots + x_n\phi_n(t), \tag{6.8}$$

where the functions ϕ_j, $j = 1, \ldots, n$, are given functions, and x_j, $j = 1, \ldots, n$, are the parameters. For example, if $g(t) = x_1 + x_2 t$, then $\phi_1(t) = 1$ and $\phi_2(t) = t$. It will always be assumed that $n < m$, that is, there are more data points than unknown parameters.

The parameters $x_1, \ldots, x_n$ are determined by minimizing the error

$$f(\mathbf{x}) := \sum_{i=1}^{m} \left(g(t_i) - y_i\right)^2 = \|A\mathbf{x} - \mathbf{y}\|_2^2, \tag{6.9}$$

where

$$A := \begin{pmatrix} \phi_1(t_1) & \phi_2(t_1) & \cdots & \phi_n(t_1) \\ \phi_1(t_2) & \phi_2(t_2) & \cdots & \phi_n(t_2) \\ \vdots & \vdots & \ddots & \vdots \\ \phi_1(t_m) & \phi_2(t_m) & \cdots & \phi_n(t_m) \end{pmatrix}, \quad \mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}, \quad \mathbf{y} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{pmatrix}.$$

In the following, we will study two different ways of finding the values of the parameters $\mathbf{x}$ that minimize (6.9): the normal equations and the QR factorization. We will assume that the matrix A has full rank n, that is, its columns are linearly independent.

6.3 Normal equations

We start by computing the gradient of f at $\mathbf{x}$ by perturbing $\mathbf{x}$:

$$\begin{aligned} f(\mathbf{x}+\mathbf{h}) - f(\mathbf{x}) &= \|A(\mathbf{x}+\mathbf{h}) - \mathbf{y}\|_2^2 - \|A\mathbf{x} - \mathbf{y}\|_2^2 \\ &= \big(A(\mathbf{x}+\mathbf{h}) - \mathbf{y}\big)^\top \big(A(\mathbf{x}+\mathbf{h}) - \mathbf{y}\big) - \big(A\mathbf{x} - \mathbf{y}\big)^\top \big(A\mathbf{x} - \mathbf{y}\big) \\ &= 2\mathbf{h}^\top A^\top A\mathbf{x} - 2\mathbf{h}^\top A^\top \mathbf{y} + O(\|\mathbf{h}\|_2^2). \end{aligned}$$

Hence,

$$\nabla f(\mathbf{x}) = 2(A^\top A\mathbf{x} - A^\top \mathbf{y})$$

and thus the gradient is zero if and only if the so called **normal equations** are satisfied:

$$A^\top A\mathbf{x} = A^\top \mathbf{y}. \tag{6.10}$$

Lemma 6.2 *If $A \in \mathbb{R}^{m\times n}$ has rank n then $A^\top A$ is symmetric positive definite.*

Proof. EFY. ∎

As a consequence of Lemma 6.2, the matrix $A^\top A$ is invertible and (6.10) has a unique solution. Since zero gradient is a necessary condition, this implies that zero gradient is also a sufficient condition for being a global minimizer of f.[17]

Theorem 6.3 *Suppose that $A \in \mathbb{R}^{m\times n}$ has rank n. Then $\mathbf{x}$ is a minimizer of $f(\mathbf{x}) = \|A\mathbf{x} - \mathbf{y}\|_2^2$ if and only if it solves the normal equations* (6.10).

The linear system (6.10) can be solved using the Cholesky factorization of $A^\mathsf{T}A$, leading to Algorithm 6.4.

[17] Another way of seeing this is to note that f is (strictly) convex.

Computation of $A^\mathsf{T}A$	$n(n+1)m$
Cholesky factorization of $A^\mathsf{T}A$	$\frac{1}{3}n^3$
Computation of $A^\mathsf{T}\mathbf{y}$	$2mn$
Forward substitution	n^2
Backward substitution	n^2

Table 6.1: Number of floating point operations for each step of Algorithm 6.4.

Algorithm 6.4 (Least-squares via normal equations)

1. $C := A^\mathsf{T}A$,
2. Compute Cholesky factor R of C using Alg. 4.17 (that is, $R^\mathsf{T}R = C$).
3. $\mathbf{b} := A^\mathsf{T}\mathbf{y}$.
4. Solve $R^\mathsf{T}\mathbf{z} = \mathbf{b}$ using forward substitution (Alg. 4.3).
5. Solve $R\mathbf{x} = \mathbf{z}$ using backward substitution (Alg. 4.4).

The computational complexity of Algorithm 6.4 is summarized in Table 6.1. The computation of $A^\mathsf{T}A$ exploits that $A^\mathsf{T}A$ is symmetric. One observes that for $m \gg n$ the computations of $A^\mathsf{T}A$ and $A^\mathsf{T}\mathbf{y}$ dominate the cost.

Remark 6.5 The normal equations (6.10) have a simple geometric interpretation. From

$$A^\mathsf{T}\left(\mathbf{b} - A\mathbf{x}\right) = 0$$

it follows that the residual $\mathbf{r} = \mathbf{y} - A\mathbf{x}$ is orthogonal to the columns of A, that is, the the residual $\mathbf{r}$ is *normal* to the span of A. It is instructive to connect this interpretation to the discussion in Section 3.3.

6.4 Method of Orthogonalization

The normal equations have a significant disadvantage; they are numerically unstable when $\kappa_2(A) \gg 1$.[18]

Example 6.6 For $m = n$ and $\phi_i(t) = t^i$ and uniformly distributed points t_i, we obtain the Vandermonde matrix

$$A = \begin{pmatrix} t_0^0 & t_0^1 & \cdots & t_0^{n-1} \\ t_1^0 & t_1^1 & \cdots & t_1^{n-1} \\ \vdots & \vdots & & \vdots \\ t_{n-1}^0 & t_{n-1}^1 & \cdots & t_{n-1}^{n-1}, \end{pmatrix}, \qquad t_i = i/(n-1).$$

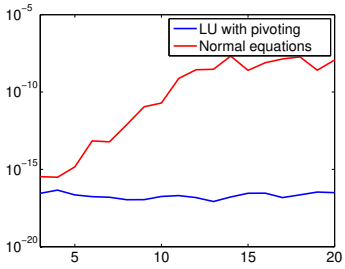
[18]The 2-norm condition number of a rectangular matrix is defined as the ration between the largest and the smallest singular value.

The linear system $A\mathbf{x} = \mathbf{y}$ with randomly chosen right-hand side $\mathbf{y}$ is solved using (a) the LU factorization with column pivoting and (b) the Cholesky factorization applied to the normal equations (6.10), For each computed solution $\widehat{\mathbf{x}}$ we measured the relative residual norm

$$\frac{\|\mathbf{r}\|_2}{\|A\|_2\|\widehat{\mathbf{x}}\|_2 + \|\mathbf{y}\|_2},$$

with

$$\mathbf{r} = \mathbf{y} - A\widehat{\mathbf{x}}.$$



It is clearly visible that the normal equations are numerically unstable; the relative residual is significantly larger than 10^{-16}. ◇

The effect observed in Example 6.6 is due to $\kappa(A^\mathsf{T}A) = \kappa(A)^2$ and hence this effect is unavoidable when working with normal equations. Note that the LU factorization has no meaningful extension to $m > n$ for least-squares problems and cannot be used as an alternative. Instead, we will use orthogonalization to reduce A to simpler form.

First, we consider the special case that the matrix $A \in \mathbb{R}^{m\times n}$ is already **upper triangular**, that is,

$$A = \begin{pmatrix} R \\ 0 \end{pmatrix} \tag{6.11}$$

for an upper triangular matrix $R \in \mathbb{R}^{n\times n}$. The vector $\mathbf{y} \in \mathbb{R}^m$ is partitioned analogously:

$$\mathbf{y} = \begin{pmatrix} \mathbf{y}_1 \\ \mathbf{y}_2 \end{pmatrix}, \qquad \mathbf{y}_1 \in \mathbb{R}^n, \quad \mathbf{y}_2 \in \mathbb{R}^{m-n}.$$

We compute

$$\begin{aligned} f(\mathbf{x}) &= \|\mathbf{y} - A\mathbf{x}\|_2^2 = \left\| \begin{pmatrix} \mathbf{y}_1 \\ \mathbf{y}_2 \end{pmatrix} - \begin{pmatrix} R \\ 0 \end{pmatrix} \mathbf{x} \right\|_2^2 \\ &= \left\| \begin{pmatrix} \mathbf{y}_1 - R\mathbf{x} \\ \mathbf{y}_2 - 0\mathbf{x} \end{pmatrix} \right\|_2^2 = \|\mathbf{y}_1 - R\mathbf{x}\|_2^2 + \|\mathbf{y}_2\|_2^2. \end{aligned}$$

If $R \in \mathbb{R}^{n\times n}$ is invertible then f is obviously minimized by the solution $\mathbf{x} \in \mathbb{R}^n$ of $R\mathbf{x} = \mathbf{y}_1$, which can be determined using backward substitution.

Using orthogonal matrices we transform $A \in \mathbb{R}^{m\times n}$ successively to the upper triangular form (6.11).

Theorem 6.7 *Consider $m \geq n$, $A \in \mathbb{R}^{m\times n}$ with rank$(A) = n$, $\mathbf{y} \in \mathbb{R}^m$. Let $Q \in \mathbb{R}^{m\times m}$ be orthogonal and $R \in \mathbb{R}^{n\times n}$ be upper triangular such that*

$$A = Q \begin{pmatrix} R \\ 0 \end{pmatrix}. \tag{6.12}$$

Partition $\widetilde{\mathbf{y}} = Q^\mathsf{T}\mathbf{y} \in \mathbb{R}^m$ as follows:

$$Q^\mathsf{T}\mathbf{y} = \begin{pmatrix} \widetilde{\mathbf{y}}_1 \\ \widetilde{\mathbf{y}}_2 \end{pmatrix}, \qquad \mathbf{y}_1 \in \mathbb{R}^n, \quad \mathbf{y}_2 \in \mathbb{R}^{m-n}.$$

Then the solution $\mathbf{x} \in \mathbb{R}^n$ of

$$R\mathbf{x} = \widetilde{\mathbf{y}}_1$$

minimizes $\|A\mathbf{x} - \mathbf{y}\|_2^2$.

Proof. Because the application of an orthogonal matrix Q^T does not change the Euclidean norm of a vector we obtain

$$\|\mathbf{y} - A\mathbf{x}\|_2 = \|Q^\mathsf{T}(\mathbf{y} - A\mathbf{x})\|_2 = \|Q^\mathsf{T}\mathbf{y} - Q^\mathsf{T}A\mathbf{x}\|_2 = \left\| \begin{pmatrix} \widetilde{\mathbf{y}}_1 \\ \widetilde{\mathbf{y}}_2 \end{pmatrix} - \begin{pmatrix} R \\ 0 \end{pmatrix} \mathbf{x} \right\|_2.$$

Hence the least-squares problem is equivalent to one in the upper triangular form discussed above. Because the rank of A is n, it follows that R is invertible, which completes the proof. □

It remains to actually compute the matrix Q from Theorem 6.7. A factorization of the form (6.12) is called **QR factorization**. In PYTHON, this factorization can be computed using `Q,R = numpy.linalg.qr(A,'complete')`. For $m \gg n$, having to compute and store the $m \times m$ matrix Q creates substantial overhead, which can be avoided. The so called **economic QR factorization** takes the form

$$A = Q_1 R,$$

where $Q_1 \in \mathbb{R}^{m\times n}$ satisfies $Q_1^\mathsf{T}Q_1 = I_n$ and $R \in \mathbb{R}^{n\times n}$ is upper triangular. This factorization completely suffices to solve the linear least-squares problem, which only requires to know R and $\widetilde{\mathbf{y}}_1 = Q_1^\mathsf{T}\mathbf{y}$. In PYTHON, this economic QR factorization is computed by `Q1,R = numpy.linalg.qr(A,'reduced')` with `numpy` package. Note that the default mode for `numpy.linalg.qr` is actually 'reduced', that is, the economic QR.

6.5 QR factorization via Gram-Schmidt

The Gram-Schmidt process is a simple way to compute the economic QR factorization.

Given $n \geq 1$ linearly independent vectors

$$\mathbf{a}_1, \ldots, \mathbf{a}_n \in \mathbb{R}^m,$$

the aim of the Gram-Schmidt process is to find n vectors

$$\mathbf{q}_1, \ldots, \mathbf{q}_n \in \mathbb{R}^m$$

such that $\|\mathbf{q}_i\|_2 = 1$ for $i = 1, \ldots, n$ and

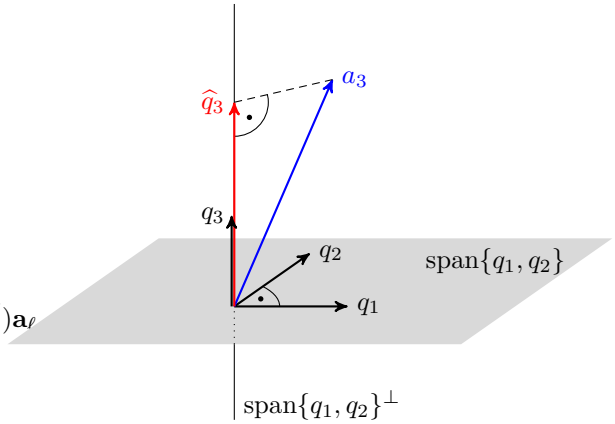
$$\operatorname{span}\{\mathbf{a}_1, \ldots, \mathbf{a}_\ell\} = \operatorname{span}\{\mathbf{q}_1, \ldots, \mathbf{q}_\ell\}, \quad \forall 1 \le \ell \le n. \tag{6.13}$$

Moreover, all vectors $\mathbf{q}_i$ are *mutually orthogonal*:

$$\mathbf{q}_i \perp \mathbf{q}_j, \quad i \neq j\,. \tag{6.14}$$

The idea of the Gram-Schmidt process is as follows. Suppose we have already computed $\ell - 1$ vectors $\mathbf{q}_1, \ldots, \mathbf{q}_{\ell-1}$ satisfying the conditions above. Then the ℓth vector $\mathbf{q}_\ell$ is obtained by projecting $\mathbf{a}_\ell$ onto the orthogonal complement of $\operatorname{span}\{\mathbf{q}_1, \ldots, \mathbf{q}_{\ell-1}\}$ and normalizing:

$$\begin{aligned} \widehat{\mathbf{q}}_\ell &= (I - (\mathbf{q}_1, \ldots, \mathbf{q}_{\ell-1})(\mathbf{q}_1, \ldots, \mathbf{q}_{\ell-1})^\mathsf{T})\mathbf{a}_\ell \\ \mathbf{q}_\ell &= \widehat{\mathbf{q}}_\ell / \|\widehat{\mathbf{q}}_\ell\|_2. \end{aligned}$$



This leads to the following algorithm.

Algorithm 6.8 (Gram-Schmidt)
Input: Linearly independent vectors $\mathbf{a}_1, \ldots, \mathbf{a}_n \in \mathbb{R}^m$.
Output: Orthonormal basis $\mathbf{q}_1, \ldots, \mathbf{q}_n$, such that (6.13) is satisfied.

for $\ell = 1, \ldots, n$ **do**
$\widehat{\mathbf{q}}_\ell := \mathbf{a}_\ell - \sum_{j=1}^{\ell-1} (\mathbf{q}_j^\mathsf{T} \mathbf{a}_\ell)\, \mathbf{q}_j$
$\mathbf{q}_\ell := \frac{\widehat{\mathbf{q}}_\ell}{\|\widehat{\mathbf{q}}_\ell\|_2}$
end for

PYTHON

```
% The columns of A containing a_1, ..., a_n are replaced by
%                             q_1, ..., q_n.
import numpy as np
for l in range(n):
    qlhat = np.zeros((n,1))
    if l!=0:
        sl = A[:,:l].T @ A[:,l]
        qlhat = A[:,l]-A[:,:l]@sl
    else:
        qlhat = A[:,l]
```

```
    rll = np.linalg.norm(qlhat)
    A[:,l] = qlhat / rll
```

The Gram-Schmidt process produces an economic QR factorization of the $m \times n$ matrix $A = (\mathbf{a}_1, \ldots, \mathbf{a}_n)$. This can be seen as follows. Let $r_{\ell\ell} := \|\widehat{\mathbf{q}}_\ell\|_2$ and $r_{j\ell} := q_j^\mathsf{T}\mathbf{a}_\ell$. Then Algorithm 6.8 implies the relation

$$\mathbf{a}_\ell = \widehat{\mathbf{q}}_\ell + \sum_{j=1}^{\ell-1} \langle \mathbf{q}_j, \mathbf{a}_\ell \rangle \, \mathbf{q}_j = \sum_{j=1}^{\ell} r_{j\ell}\mathbf{q}_j$$

for $\ell = 1, \ldots, n$. Collecting these relations into a single matrix yields

$$A = Q_1 R, \quad \text{with} \quad R = \begin{pmatrix} r_{11} & r_{12} & \cdots & r_{1n} \\ 0 & r_{22} & \cdots & r_{2n} \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & r_{nn} \end{pmatrix}. \tag{6.15}$$

But this happens to be an economic QR factorization, as

$$Q_1^\mathsf{T} Q_1 = \begin{pmatrix} \mathbf{q}_1^\mathsf{T}\mathbf{q}_1 & \mathbf{q}_1^\mathsf{T}\mathbf{q}_2 & \cdots & \mathbf{q}_1^\mathsf{T}\mathbf{q}_n \\ \mathbf{q}_2^\mathsf{T}\mathbf{q}_1 & \mathbf{q}_2^\mathsf{T}\mathbf{q}_2 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \mathbf{q}_{n-1}^\mathsf{T}\mathbf{q}_n \\ \mathbf{q}_n^\mathsf{T}\mathbf{q}_1 & \cdots & \mathbf{q}_n^\mathsf{T}\mathbf{q}_{n-1} & \mathbf{q}_n^\mathsf{T}\mathbf{q}_n \end{pmatrix} = \begin{pmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & 1 \end{pmatrix}$$

Remark 6.9 *Due to roundoff error, the basis produced by the Gram-Schmidt process may lose orthogonality to a large extent, especially if the columns of A are nearly linearly dependent. A simple but effective cure is to perform orthogonalization twice:*

PYTHON

```
import numpy as np
for l in range(n):
    qlhat = np.zeros((n,1))
    if l!=0:
        sl = A[:,:l].T @ A[:,l]
        A[:,l] = A[:,l] - A[:,:l] @ sl # Standard orth step.
        sl = A[:,:l].T @ A[:,l]
        qlhat = A[:,l] - A[:,:l] @ sl  # Extra orth step.
    else:
        qlhat = A[:,l]
    rll = np.linalg.norm(qlhat)
    A[:,l] = qlhat / rll
```

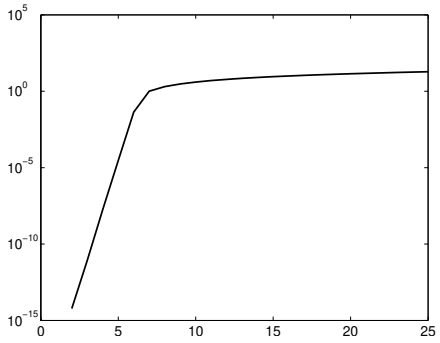
Example 6.10 Let us consider the $25 \times n$ matrix

$$A = \begin{pmatrix} 1 & t_1^1 & \cdots & t_1^n \\ 1 & t_2^1 & \cdots & t_2^n \\ \vdots & \vdots & & \vdots \\ 1 & t_{25}^1 & \cdots & t_{25}^n, \end{pmatrix}, \qquad t_i = (i-1)/24,$$

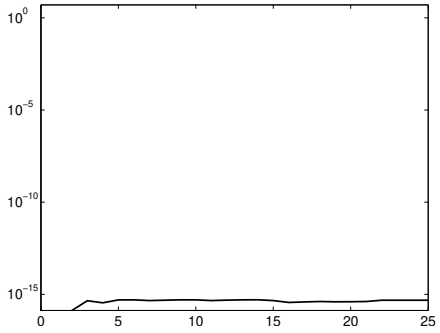
which arises when fitting a polynomial of degree n. We apply the Gram-Schmidt process (Algorithm 6.8) as well as the modification discussed in Remark 6.9. To measure the loss of orthogonality in the computed matrix $\widehat{Q}_1$, we compute

$$\|I_n - \widehat{Q}_1^{\mathsf{T}} \widehat{Q}_1\|_2.$$

Ideally, this quantity should not be much larger than 10^{-16}. The following two figures show this quantity for $n = 1, \ldots, 25$:



Gram-Schmidt
Algorithm 6.8



Improved Gram-Schmidt
Remark 6.9

It turns out that orthogonality is completely lost in the Gram-Schmidt process for $n = 7$ or larger. On the other hand, the improved Gram-Schmidt process maintains orthogonality nearly perfectly, with a loss of only $\approx 10^{-16}$. ◇

6.6 QR factorization via Householder reflectors⋆

The MATLAB command `qr` does not use Gram-Schmidt but a different approach for computing the QR factorization, which completely avoids any issue with

In the Householder based QR factorization one constructs the matrix Q as a composition of simple orthogonal matrices. There are two types of elementary matrices, Givens rotations and Householder reflectors, and both are suitable for constructing QR factorizations. We will focus our discussion on Householder reflectors.

6.6.1 Construction

Theorem 6.11 (Properties of Householder reflectors) *Let* $\mathbf{0} \neq \mathbf{v} \in \mathbb{R}^m$. *Then the* **Householder reflector**

$$Q := I_m - \frac{2}{\mathbf{v}^{\mathsf{T}}\mathbf{v}} \mathbf{v}\mathbf{v}^{\mathsf{T}}$$

has the following properties:

1. *Q is symmetric,*
2. *Q is orthogonal,*
3. *$Q^2 = I_m$.*

Proof. EFY. ∎

Remark 6.12 *Householder reflectors have a geometric interpretation. When applied to a vector, they correspond to reflecting the vector at the hyperplane $\{\mathbf{x} \in \mathbb{R}^m \,|\, \mathbf{x}^\mathsf{T}\mathbf{v} = 0\}$.*

A QR factorization of $A \in \mathbb{R}^{m\times n}$ is produced by successively reducing the columns of the matrix using Householder reflections. The following result is essential for this purpose.

Lemma 6.13 *Let $\mathbf{0} \neq \mathbf{a} \in \mathbb{R}^m$ and let $\mathbf{e}_1 \in \mathbb{R}^m$ denote the first unit vector. Let*

$$\alpha = \|\mathbf{a}\|_2 \quad \text{or} \quad \alpha = -\|\mathbf{a}\|_2.$$

(If $\mathbf{a} = \beta\mathbf{e}_1$ one uses $\alpha = -\|\mathbf{a}\|_2 = -|\beta|$.) Then with $\mathbf{v} := \mathbf{a} - \alpha\mathbf{e}_1$ it holds that

$$Q = I_m - \frac{2}{\mathbf{v}^\mathsf{T}\mathbf{v}}\mathbf{v}\mathbf{v}^\mathsf{T} \qquad \Rightarrow \qquad Q\mathbf{a} = \alpha\mathbf{e}_1. \tag{6.16}$$

Proof. The choice of α implies that $\mathbf{v} \neq \mathbf{0}$ and thus Q is well defined. The claim $Q\mathbf{a} = \alpha\mathbf{e}_1$ is verified by direct calculation. ∎

Remark 6.14 *In practice, one chooses $\alpha = -\text{sign}(a_1)\sqrt{\mathbf{a}^\mathsf{T}\mathbf{a}}$, $\mathbf{v} = (a_1 - \alpha, a_2, \ldots, a_k)^\mathsf{T}$ to avoid numerical cancellation*[19]*.*

Now, let $\mathbf{a}_1$ denote the first column of $A \in \mathbb{R}^{m\times n}$. In the first step of the Householder based QR factorization A we use Lemma 6.13 to construct a Householder reflector $Q^{(1)}$ such that

$$Q^{(1)}\mathbf{a}_1 = \begin{pmatrix} r_{11} \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

for some $r_{11} \in \mathbb{R}$. Applied to A we obtain

$$Q^{(1)}A = \left(\begin{array}{c|ccc} r_{11} & r_{12} & \cdots & r_{1n} \\ \hline 0 & & & \\ \vdots & & A^{(1)} & \\ 0 & & & \end{array}\right), \tag{6.17}$$

[19] We define $\text{sign}(x) = 1$ for $x \geq 0$ and $\text{sign}(x) = -1$ for $x < 0$

with $A^{(1)} \in \mathbb{R}^{(m-1)\times(n-1)}$.

In the next step, we repeat this procedure for the submatrix $A^{(1)}$ and embed the transformations as follows:

$$Q^{(2)} := \left(\begin{array}{c|ccc} 1 & 0 & \cdots & 0 \\ \hline 0 & & & \\ \vdots & & \widetilde{Q}^{(2)} & \\ 0 & & & \end{array}\right),$$

and

$$\left(\begin{array}{c|ccc} 1 & 0 & \cdots & 0 \\ \hline 0 & & & \\ \vdots & & \widetilde{Q}^{(2)} & \\ 0 & & & \end{array}\right)\left(\begin{array}{c|ccc} r_{11} & r_{12} & \cdots & r_{1n} \\ \hline 0 & & & \\ \vdots & & A^{(1)} & \\ 0 & & & \end{array}\right) = \left(\begin{array}{c|ccc} r_{11} & r_{12} & \cdots & r_{1n} \\ \hline 0 & & & \\ \vdots & & \widetilde{Q}^{(2)}A^{(1)} & \\ 0 & & & \end{array}\right).$$

Letting $\mathbf{a}_2$ denote the first column of $A^{(1)}$, we choose $\widetilde{Q}^{(2)}$ as a Householder reflector that maps $\mathbf{a}_2$ to a scalar multiple of $\mathbf{e}_1 \in \mathbb{R}^{m-1}$. Then

$$Q^{(2)}Q^{(1)}A = \left(\begin{array}{cc|ccc} r_{11} & r_{12} & r_{13} & \cdots & r_{1n} \\ 0 & r_{22} & r_{23} & \cdots & r_{2n} \\ \hline 0 & 0 & & & \\ \vdots & \vdots & & A^{(2)} & \\ 0 & 0 & & & \end{array}\right),$$

It is now clear how to continue this process until the matrix $Q^{(n)}Q^{(n-1)}\cdots Q^{(1)}A$ is in upper triangular form (if $m = n$ one can skip the last step, $Q^{(n)}$).

In summary, the QR factorization of $A \in \mathbb{R}^{n\times n}$ is given by

$$A = (Q^{(1)})^\mathsf{T}\cdots(Q^{(n)})^\mathsf{T}R = \underbrace{Q^{(1)}\cdots Q^{(n)}}_{=:Q}\begin{pmatrix} R \\ 0 \end{pmatrix},$$

where $R \in \mathbb{R}^{n\times n}$ us upper triangular and the orthogonal matrices $Q^{(k)} \in \mathbb{R}^{m\times m}$ take the following form:

$$Q^{(k)} = \left(\begin{array}{c|c} I_{k-1} & 0\cdots 0 \\ \hline 0 & \\ \vdots & I_{m-k+1} - \dfrac{2}{\mathbf{v}_k^\mathsf{T}\mathbf{v}_k}\mathbf{v}_k\mathbf{v}_k^\mathsf{T} \\ 0 & \end{array}\right), \quad \mathbf{v}_k \in \mathbb{R}^{m-k+1}. \tag{6.18}$$

Algorithm 6.15 summarizes the described procedure.

Algorithm 6.15 (QR factorization)
Input: Matrix $A \in \mathbb{R}^{m\times n}$.
Output: QR factorization $A = Q\begin{pmatrix} R \\ 0 \end{pmatrix}$ with Q orthogonal and R upper triangular.

$Q = I_m$
for $k = 1, \ldots, \min\{n, m-1\}$ **do**
  Determine (embedded) Householder reflector $Q^{(k)}$ (see (6.18)) such that the trailing $m-k$ entries of the kth column of $Q^{(k)}A$ are zero.
  Replace $A \leftarrow Q^{(k)}A$.
  Replace $Q \leftarrow Q^{(k)}Q$.
end for
Return R as the first n rows of A.

Example 6.16 For

$$A = \begin{pmatrix} -4 & -2-2\sqrt{6} & -6-3\sqrt{2}-\sqrt{6} \\ 0 & -2\sqrt{3} & 9-\sqrt{3} \\ -4\sqrt{2} & -2\sqrt{2}+2\sqrt{3} & 3-6\sqrt{2}+\sqrt{3} \end{pmatrix}$$

we compute its QR factorization using Algorithm 6.15. The first column of A is

$$\mathbf{a}_1 = \begin{pmatrix} -4 \\ 0 \\ -4\sqrt{2} \end{pmatrix}, \qquad \|\mathbf{a}_1\|_2 = \sqrt{16+16\cdot 2} = \sqrt{48} = 4\sqrt{3}.$$

For the corresponding Householder reflector, we compute $\alpha = -\operatorname{sign}(a_1)\|\mathbf{a}\|_2 = -\operatorname{sign}(-4)4\sqrt{3} = 4\sqrt{3}$ and hence

$$\mathbf{v}_1 = \mathbf{a}_1 - \alpha\mathbf{e}_1 = \begin{pmatrix} -4 \\ 0 \\ -4\sqrt{2} \end{pmatrix} - 4\sqrt{3}\begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} -4-4\sqrt{3} \\ 0 \\ -4\sqrt{2} \end{pmatrix}, \quad \|\mathbf{v}_1\|_2^2 = 16(6+2\sqrt{3}),$$

$$\begin{aligned} Q^{(1)} &= I_3 - \frac{2}{\|\mathbf{v}_1\|_2^2}\mathbf{v}_1\mathbf{v}_1^{\mathsf{T}} = I_3 - \frac{2}{16(6+2\sqrt{3})}\begin{pmatrix} -4-4\sqrt{3} \\ 0 \\ -4\sqrt{2} \end{pmatrix}\cdot\begin{pmatrix} -4-4\sqrt{3}, & 0, & -4\sqrt{2} \end{pmatrix} \\ &= I_3 - \frac{1}{3+\sqrt{3}}\begin{pmatrix} -1-\sqrt{3} \\ 0 \\ -\sqrt{2} \end{pmatrix}\cdot\begin{pmatrix} -1-\sqrt{3}, & 0, & -\sqrt{2} \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} - \frac{1}{\sqrt{3}(1+\sqrt{3})}\begin{pmatrix} 4+2\sqrt{3} & 0 & \sqrt{2}(1+\sqrt{3}) \\ 0 & 0 & 0 \\ \sqrt{2}(1+\sqrt{3}) & 0 & 2 \end{pmatrix} \\ &= \frac{1}{\sqrt{3}}\begin{pmatrix} -1 & 0 & -\sqrt{2} \\ 0 & \sqrt{3} & 0 \\ -\sqrt{2} & 0 & 1 \end{pmatrix}. \end{aligned}$$

Evidently, $Q^{(1)}$ is – as expected – an orthogonal matrix. We obtain

$$\begin{aligned} Q^{(1)}A &= \frac{1}{\sqrt{3}}\begin{pmatrix} -1 & 0 & -\sqrt{2} \\ 0 & \sqrt{3} & 0 \\ -\sqrt{2} & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} -4 & -2-2\sqrt{6} & -6-3\sqrt{2}-\sqrt{6} \\ 0 & -2\sqrt{3} & 9-\sqrt{3} \\ -4\sqrt{2} & -2\sqrt{2}+2\sqrt{3} & 3-6\sqrt{2}+\sqrt{3} \end{pmatrix} \\ &= \frac{1}{\sqrt{3}}\begin{pmatrix} 12 & 6 & 18 \\ 0 & -6 & -3+9\sqrt{3} \\ 0 & 6\sqrt{3} & 9+3\sqrt{3} \end{pmatrix} = \sqrt{3}\begin{pmatrix} 4 & 2 & 6 \\ 0 & -2 & -1+3\sqrt{3} \\ 0 & 2\sqrt{3} & 3+\sqrt{3} \end{pmatrix}. \end{aligned}$$

The first column $Q^{(1)}A$ equals $\alpha\mathbf{e}_1$, as desired.

We repeat this procedure for the submatrix

$$A^{(1)} = \sqrt{3}\begin{pmatrix} -2 & -1+3\sqrt{3} \\ 2\sqrt{3} & 3+\sqrt{3} \end{pmatrix}.$$

It is first column is

$$\mathbf{a}_2 = \begin{pmatrix} -2\sqrt{3} \\ 6 \end{pmatrix}, \qquad \|\mathbf{a}_2\|_2 = \sqrt{12+36} = 4\sqrt{3}.$$

In turn, $\alpha = -\mathrm{sign}(a_1)\|\mathbf{a}_2\|_2 = -\mathrm{sign}(-2\sqrt{3})\cdot 4\sqrt{3} = 4\sqrt{3}$ and for $\mathbf{v}_2 = \mathbf{a}_2 - \alpha\mathbf{e}_1$ we obtain

$$\mathbf{v}_2 = \begin{pmatrix} -2\sqrt{3} \\ 6 \end{pmatrix} - 4\sqrt{3}\begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} -6\sqrt{3} \\ 6 \end{pmatrix}, \qquad \|\mathbf{v}_2\|_2^2 = 144.$$

Therefore $\widetilde{Q}^{(2)}$ is given by

$$\begin{aligned} \widetilde{Q}^{(2)} &= I_2 - \frac{2}{\|\mathbf{v}\|_2^2}\mathbf{v}\cdot\mathbf{v}^{\mathsf{T}} = I_2 - \frac{2}{144}\cdot\begin{pmatrix} -6\sqrt{3} \\ 6 \end{pmatrix}\cdot\begin{pmatrix} -6\sqrt{3}, & 6 \end{pmatrix} \\ &= I_2 - \frac{1}{2}\cdot\begin{pmatrix} -\sqrt{3} \\ 1 \end{pmatrix}\cdot\begin{pmatrix} -\sqrt{3}, & 1 \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} - \frac{1}{2}\begin{pmatrix} 3 & -\sqrt{3} \\ -\sqrt{3} & 1 \end{pmatrix} = \frac{1}{2}\begin{pmatrix} -1 & \sqrt{3} \\ \sqrt{3} & 1 \end{pmatrix} \end{aligned}$$

and we obtain

$$\widetilde{Q}^{(2)}A^{(1)} = \frac{1}{2}\begin{pmatrix} -1 & \sqrt{3} \\ \sqrt{3} & 1 \end{pmatrix}\cdot\sqrt{3}\begin{pmatrix} -2 & -1+3\sqrt{3} \\ 2\sqrt{3} & 3+\sqrt{3} \end{pmatrix} = \frac{\sqrt{3}}{2}\begin{pmatrix} 8 & 4 \\ 0 & 12 \end{pmatrix}.$$

This already gives the upper triangular matrix R:

$$R = \sqrt{3}\begin{pmatrix} 4 & 2 & 6 \\ 0 & 4 & 2 \\ 0 & 0 & 6 \end{pmatrix}.$$

The orthogonal matrix Q^{T} with $Q^{\mathsf{T}}A = R$ is given by

$$\begin{aligned} Q^{\mathsf{T}} &= \left(\begin{array}{c|cc} 1 & 0 & 0 \\ \hline 0 & \multicolumn{2}{c}{\multirow{2}{*}{$\widetilde{Q}^{(2)}$}} \\ 0 & & \end{array}\right) Q^{(1)} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & -\frac{1}{2} & \frac{1}{2}\sqrt{3} \\ 0 & \frac{1}{2}\sqrt{3} & \frac{1}{2} \end{pmatrix}\cdot\frac{1}{\sqrt{3}}\begin{pmatrix} -1 & 0 & -\sqrt{2} \\ 0 & \sqrt{3} & 0 \\ -\sqrt{2} & 0 & 1 \end{pmatrix} \\ &= \frac{1}{2\sqrt{3}}\begin{pmatrix} -2 & 0 & -2\sqrt{2} \\ -\sqrt{6} & -\sqrt{3} & \sqrt{3} \\ -\sqrt{2} & 3 & 1 \end{pmatrix}. \end{aligned}$$

For verification we compute

$$\begin{aligned} QR &= \frac{1}{2\sqrt{3}} \begin{pmatrix} -2 & -\sqrt{6} & -\sqrt{2} \\ 0 & -\sqrt{3} & 3 \\ -2\sqrt{2} & \sqrt{3} & 1 \end{pmatrix} \cdot \sqrt{3} \begin{pmatrix} 4 & 2 & 6 \\ 0 & 4 & 2 \\ 0 & 0 & 6 \end{pmatrix} \\ &= \begin{pmatrix} -4 & -2-2\sqrt{6} & -6-3\sqrt{2}-\sqrt{6} \\ 0 & -2\sqrt{3} & 9-\sqrt{3} \\ -4\sqrt{2} & -2\sqrt{2}+2\sqrt{3} & 3-6\sqrt{2}+\sqrt{3} \end{pmatrix} = A. \end{aligned}$$

◇

It would be quite inefficient to implement Algorithm 6.15 literally. In particular, the matrix $Q^{(k)}$ is never formed but only applied implicitly via the vector $\mathbf{v}_k$. To apply $Q^{(k)}$ to matrix $\begin{pmatrix} B_1 \\ B_2 \end{pmatrix}$ with $B_1 \in \mathbb{R}^{(k-1)\times \ell}$ and $B_2 \in \mathbb{R}^{(m-k+1)\times \ell}$, one notes that

$$Q^{(k)} \begin{pmatrix} B_1 \\ B_2 \end{pmatrix} = \begin{pmatrix} B_1 \\ B_2 - \frac{2}{\mathbf{v}_k^\mathsf{T}\mathbf{v}_k} \mathbf{v}_k \mathbf{v}_k^\mathsf{T} B_2 \end{pmatrix}.$$

One first computes the vector $\mathbf{w} = \frac{2}{\mathbf{v}_k^\mathsf{T}\mathbf{v}_k} B_2^\mathsf{T}\mathbf{v}_k$ ($\approx 2(m-k+1)\ell$ flops) and then the rank-one update $B_2 - \mathbf{v}_k\mathbf{w}^\mathsf{T}$ ($\approx 2(m-k+1)\ell$ flops). Using the structure of A the operation $A \leftarrow Q^{(k)}A$ in Algorithm 6.15 only costs $4(m-k+1)(n-k+1)$ flops. Altogether the updates of A cost

$$4\sum_{k=1}^{n}(m-k+1)(n-k+1) \approx 2mn^2 - \frac{2}{3}n^3 \tag{6.19}$$

flops. The computation of Q is executed analogously.

6.6.2 Application to linear least-squares problems

In principle, Algorithm 6.15 together with Theorem 6.7 provide all the tools needed to solve least-squares problems. However, one can avoid the (expensive) computation of Q if one instead directly computes $Q^\mathsf{T}\mathbf{y}$; see Algorithm 6.17.

Algorithm 6.17 (Least-squares problems via QR factorization)
Input: Matrix $A \in \mathbb{R}^{m\times n}$ with $\operatorname{rank}(A) = n$, $\mathbf{y} \in \mathbb{R}^m$
Output: Solution $\mathbf{x} \in \mathbb{R}^n$ of least-square problem $\min \|A\mathbf{x} - \mathbf{y}\|_2^2$.

for $k = 1, \ldots, \min\{n, m-1\}$ **do**
Determine (embedded) Householder reflector $Q^{(k)}$ (see (6.18)) such that the trailing $m-k$ entries of the kth column of $Q^{(k)}A$ are zero.
Replace $A \leftarrow Q^{(k)}A$.
Replace $\mathbf{y} \leftarrow Q^{(k)}\mathbf{y}$.
end for
Partition $A = \begin{pmatrix} R \\ 0 \end{pmatrix}$ and $\mathbf{y} = \begin{pmatrix} \mathbf{y}_1 \\ \mathbf{y}_2 \end{pmatrix}$.
Compute $\mathbf{x} = R^{-1}\mathbf{y}_1$ using Algorithm 4.4.

Since all other costs are negligible, the overall cost of Algorithm 6.17 is given by the cost for updating A (see (6.19)), that is, $2mn^2 - \frac{2}{3}n^3$ flops. Compared with the approach via normal equations, see Table 6.1, the cost is at most a factor 2 larger.

Chapter 7

Fourier transform

7.1 Recap of Fourier analysis

This section recalls material from the Analysis IV course.

Definition 7.1 *Let $L > 0$. A function $f : \mathbb{R} \to \mathbb{R}$ or $f : \mathbb{R} \to \mathbb{C}$ is called* L-periodic *if*

$$f(x + L) = f(x) \qquad \forall\ x \in \mathbb{R}.$$

In principle, any function on a bounded interval, $f : [a, b) \to \mathbb{R}$, could be extended to an L-periodic function with $L = b - a$ by setting $f(x + kL) = f(x)$ for $x \in [a, b)$ and $k \in \mathbb{Z}$. However, this is not a very useful way to think about periodic functions, because such extensions will rarely turn out to be continuous, differentiable, ... on $\mathbb{R}$.

Clearly, the functions $\sin(x)$, $\cos(x)$, $e^{\mathrm{i}x}$ are 2π-periodic. In the following, we will assume that $L = 2\pi$ without loss of generality (after a suitable rescaling of x).

For a 2π-periodic function $f : \mathbb{R} \to \mathbb{R}$ we can identify f with its restriction on any interval of length 2π. Let us take, for instance, $[0, 2\pi]$. *In the following, we will simply say "periodic function $f : [0, 2\pi] \to \mathbb{R}$".*

From the Analysis IV course, it is known that the Fourier series

$$\sum_{k=-\infty}^{+\infty} c_k e^{\mathrm{i}kx}, \quad c_k = \frac{1}{2\pi} \int_0^{2\pi} f(x) e^{-\mathrm{i}kx}\, \mathrm{d}x, \qquad \forall\ k \in \mathbb{Z}. \tag{7.1}$$

converges to f under certain conditions in a certain sense. For example, if $f \in L^2(0, 2\pi)$ then the truncated functions

$$f_N(x) := \sum_{k=-N}^{N} c_k e^{\mathrm{i}kx} \tag{7.2}$$

converge to f in the L^2-norm.

Lemma 7.2 (Orthogonality relations) *For every $k, \ell \in \mathbb{Z}$,*

$$\langle e^{\mathrm{i}\ell\cdot}, e^{\mathrm{i}k\cdot}\rangle_{L^2} := \int_0^{2\pi} e^{-\mathrm{i}\ell x} e^{\mathrm{i}kx}\,\mathrm{d}x = \begin{cases} 0 & \text{if } k \neq \ell, \\ 2\pi & \text{if } k = \ell. \end{cases}$$

Proof. See Analysis IV. □

Lemma 7.2 implies that $\left(e^{\mathrm{i}kx}\right)_{k\in\mathbb{Z}}$ is an orthogonal basis of $L^2(0,2\pi)$. By the Plancherel / Parseval theorem[20], we obtain for $f \in L^2(0,2\pi)$ that

$$\|f\|_{L_2}^2 = 2\pi \sum_{k=-\infty}^{\infty} |c_k|^2,$$

with the Fourier coefficients c_k defined as in (7.1). As a consequence, approximating f with the truncated Fourier series f_N in (7.2) results in the approximation error

$$\|f - f_N\|_{L_2}^2 = 2\pi \sum_{|k|>N} |c_k|^2. \tag{7.3}$$

Note that f_N can be represented by the $2N+1$ complex numbers $c_{-N}, \ldots, c_N$, where $c_{-k} = \overline{c_k}$ (and $c_0 \in \mathbb{R}$). It is worth noting that f_N is still real; using Euler's formula one has for $x \in \mathbb{R}$ that

$$\begin{aligned} f_N(x) &= \sum_{k=-N}^{N} c_k e^{\mathrm{i}kx} = \sum_{k=-N}^{N} c_k(\cos(kx) + \mathrm{i}\sin(kx)) \\ &= c_0 + \sum_{k=1}^{N} (c_k + \overline{c_k})\cos(kx) + \mathrm{i}(c_k - \overline{c_k})\sin(kx) \\ &= \frac{a_0}{2} + \sum_{k=1}^{\infty} a_k \cos(kx) + b_k \sin(kx), \end{aligned} \tag{7.4}$$

where $a_0 = 2c_0$, $a_k = 2 \cdot \mathsf{Re}(c_k)$, $b_k = -2 \cdot \mathsf{Im}(c_k)$.

Let us recall the definition of the most famous integral transform.

Definition 7.3 *Given $f \in L^1(0,2\pi)$, we define the* Fourier transform *of f as*

$$\hat{f}(\xi) := \frac{1}{2\pi} \int_0^{2\pi} f(x) e^{-\mathrm{i}\xi x}\,\mathrm{d}x \qquad \forall\ x \in \mathbb{R}. \tag{7.5}$$

Comparing (7.1) and (7.5), we observe that $c_k = \hat{f}(k)$ for every $k \in \mathbb{Z}$.

7.2 Regularity and Fourier coefficients*

As we have seen in (7.3), the size of the Fourier coefficients for $k \to \infty$ determines how well f can be approximated by truncated Fourier series. From the *Riemann-Lebesgue Lemma* we know that $\hat{f}(k) \to 0$ as $|k| \to +\infty$ for $f \in L^1(0,2\pi)$. If

[20] In Analysis IV, this theorem might have been presented for the period $L = 1$, for which the pre-factor 2π is not needed.

extra regularity on f is assumed then it is possible to say more about the speed of convergence of its Fourier coefficients.

Proposition 7.4 *Let f be 2π-periodic and $m+1$ times continuously differentiable on $\mathbb{R}$. Then*

$$|\hat{f}(k)| \leq C_m |k|^{-m-1} \qquad \forall\ k \in \mathbb{Z},$$

where $C_m := \frac{1}{2\pi}\int_0^{2\pi} |f^{(m+1)}(x)|\ \mathrm{d}x$.

Proof. Given $k \in \mathbb{Z}$, one computes

$$\hat{f}(k) = \frac{1}{2\pi}\int_0^{2\pi} f(x)e^{-\mathrm{i}kx}\ \mathrm{d}x = \frac{1}{2\pi\mathrm{i}k}\int_0^{2\pi} f'(x)e^{-\mathrm{i}kx}\ \mathrm{d}x = \frac{1}{\mathrm{i}k}\widehat{f'}(k),$$

where we used that the boundary term from the integration by parts vanishes because f and e^{-ikx} are 2π-periodic. By reiterating this procedure, it follows that

$$\hat{f}(k) = \frac{1}{(\mathrm{i}k)^{m+1}}\widehat{f^{(m+1)}}(k),$$

Because $f^{(m+1)}$ is continuous it is also in L^1. In turn, we can estimate

$$|\hat{f}(k)| \leq \frac{1}{2\pi|k|^{m+1}}\int_0^{2\pi} |f^{(m+1)}(x)|\ \mathrm{d}x,$$

which completes the proof. □

If f is infinitely often differentiable then it follows from Proposition 7.4 that $c_k = \hat{f}(k)$ decays faster than $|k|^{-m}$ for *any* $m \in \mathbb{N}$. This superpolynomial convergence is improved further under the stronger assumption that f is real analytic. In this case, using tools from complex/harmonic analysis, it can be shown that there exist constants $\rho > 1$ and $C > 0$ such that

$$|\hat{f}(k)| \leq C\rho^{-|k|} \qquad \forall\ k \in \mathbb{Z}. \tag{7.6}$$

7.3 The discrete Fourier transform (DFT)

The discrete Fourier transform (DFT) is a discrete analogue of the formula (7.5), that is, the transformation of a vector instead of a function.

Let us assume that the function f is sampled uniformly, that is, f is only known at the following set of n points in $[0, 2\pi]$:

$$x_j = \frac{2\pi j}{n}, \qquad y_j = f(x_j), \qquad \forall\ j = 0, 1, \ldots, n-1.$$

In analogy to (7.5), the *DFT* transforms these n function values into

$$z_k = \sum_{j=0}^{n-1} y_j \omega_n^{kj} \qquad \forall\ k = 0, \ldots, n-1, \tag{7.7}$$

where $\omega_n := e^{-\frac{2\pi \mathrm{i}}{n}}$ is an nth root of unity. Up to scaling, the DFT can be obtained by applying the composite trapezoidal rule to approximate (7.5) using the integration nodes $(x_j)_{j=0}^{n-1}$ (EFY).

Let us recall that $\left(\omega_n^{-k}\right)_{k=0}^{n-1}$ forms an Abelian group, the so called group of nth roots of unity. Another important property is the discrete analogue of the orthogonality relations shown in Lemma 7.2.

Lemma 7.5 (Discrete orthogonality relations) *For every $k, \ell = 0, 1, \ldots, n-1$, it holds that*

$$\sum_{j=0}^{n-1} \omega_n^{kj} \omega_n^{-\ell j} = \begin{cases} 0 & \text{if} \quad k \neq \ell, \\ n & \text{if} \quad k = \ell. \end{cases}$$

Proof. EFY. □

By defining the vectors $\mathbf{z} = \begin{pmatrix} z_0 & z_1 & \cdots & z_{n-1} \end{pmatrix}^\top$ and $\mathbf{y} = \begin{pmatrix} y_0 & y_1 & \cdots & y_{n-1} \end{pmatrix}^\top$, the DFT (7.7) can be expressed as the matrix-vector product

$$\mathbf{z} = F_n \mathbf{y},$$

with the so called *Fourier matrix*

$$F_n = \begin{pmatrix} 1 & 1 & \cdots & 1 \\ 1 & \omega^1 & \cdots & \omega^{n-1} \\ \vdots & \vdots & & \vdots \\ 1 & \omega^{n-1} & \cdots & \omega^{(n-1)(n-1)} \end{pmatrix} = (f_{kj})_{j,k=0}^{n-1}. \tag{7.8}$$

Note that it is custom to count vector/matrix indices from zero in the context of the DFT.

We now let a_{kl} denote the entry (j, k) (again counting from 0) of the matrix $F_n F_n^{\mathsf{H}}$ and obtain from Lemma 7.5 that

$$a_{k\ell} = \sum_{j=0}^{n-1} f_{kj} \overline{f_{\ell j}} = \sum_{i=0}^{n-1} \omega_n^{kj} \omega_n^{-\ell j} = \begin{cases} 0 & \text{if} \quad k \neq \ell, \\ n & \text{if} \quad k = \ell. \end{cases}$$

Hence, we get $F_n F_n^{\mathsf{H}} = n I_n$ or, in other words, $\frac{1}{\sqrt{n}} F_n$ is unitary! Therefore,

$$F_n^{-1} = \frac{1}{n} F_n^{\mathsf{H}} = \frac{1}{n} \overline{F}_n.$$

Given the transformed vector $\mathbf{z}$, this allows us to recover the data $\mathbf{y}$ from its DFT $\mathbf{z}$ using

$$\mathbf{y} = F_n^{-1} \mathbf{z} = \frac{1}{n} \overline{F}_n \mathbf{z}.$$

In terms of the entries, this Inverse Discrete Fourier Transform (IDFT) takes the form

$$y_k = \frac{1}{n} \sum_{j=0}^{n-1} z_j \omega_n^{-kj} \qquad \forall\ k = 0, \ldots, n-1.$$

Example 7.6 (Sound digitization and data compression) *For carrying out this compression in* PYTHON*, we first need to load a file (called audio.mat in the following code snippet) containing an audio signal:*

```
from scipy.io import loadmat
from IPython.display import Audio
audio =  loadmat("audio.mat") # Load audio signal stored in audio.mat
y = audio['y']
Fs = audio['Fs'][0][0]
y = y[:,0]
```

To play this audio signal:

```
Audio(y,rate = Fs)
```

The commands

```
import scipy.fft as fft
z = fft.fft(y);
```

compute the DFT of the audio samples contained in the vector $\mathbf{y}$*. From Figure 7.1, it is evident that the absolute values of the Fourier transform are small in large parts of the frequency range. We neglect these parts using*

```
ztilde = z * (abs(z)>30);
```

Using

```
ytilde = fft.ifft(ztilde)
Audio(ytilde,rate = Fs)
```

we compute the inverse transform and play the compressed signal. While the signal looks visually different, the sound does not seem to change too much.

One problem we neglected in the whole discussion above is that audio signals are usually not *periodic, which may lead to undesirable effects when blindly applying Fourier transforms. One way to fix this is to pad audio signals with zeros before performing the transform.* ◇

7.4 Resolving a mystery about the composite trapezoidal rule*

For a 2π-periodic function f we consider the approximation of the integral

$$\int_0^{2\pi} f(x)\,\mathrm{d}x.$$

Because of periodicity, the composite trapezoidal rule takes the form

$$Q_h^{(1)}[f] = \frac{2\pi}{N}\sum_{j=0}^{N-1} f(j/N)$$

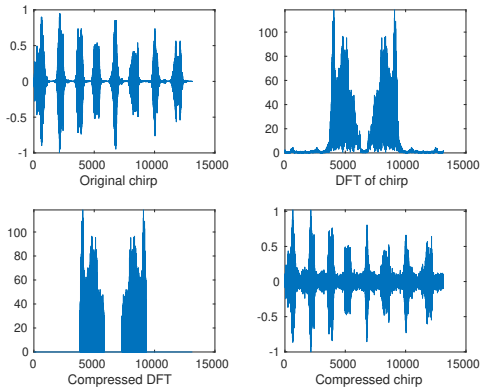


Figure 7.1: Compression of the chirp signal (from Matlab). Displayed is the original signal (top left), the absolute values of the DFT (top right), the compressed DFT (bottom left), and the compressed signal obtained from the IDFT of the compressed DFT (bottom right).

with $h = 2\pi/N$. Let us now consider the truncated Fourier expansion

$$f_{N-1}(x) := \sum_{k=-N+1}^{N-1} c_k e^{\mathrm{i}kx}.$$

On the one hand, we have

$$\int_0^{2\pi} e^{\mathrm{i}kx}\,\mathrm{d}x = \begin{cases} 0 & \text{for } k \neq 0, \\ 2\pi & \text{for } k = 0. \end{cases}$$

On the other hand, Lemma 7.5 yields

$$Q_h^{(1)}[e^{\mathrm{i}k\cdot}] = \frac{2\pi}{N} \sum_{j=0}^{N-1} e^{2\pi \mathrm{i} jk/N} = \begin{cases} 0 & \text{for } k = -N+1, \ldots, -1, 1, \ldots, N-1, \\ 2\pi & \text{for } k = 0. \end{cases}$$

Hence, the composite trapezoidal rule with $h = 2\pi/N$ integrates the truncated Fourier expansion f_{N-1} *exactly*! This yields the following error bound:

$$\begin{aligned} & \left| \int_0^{2\pi} f(x)\,\mathrm{d}x - Q_h^{(1)}[f] \right| \\ \leq\ & \left| \int_0^{2\pi} (f(x) - f_{N-1}(x))\,\mathrm{d}x - Q_h^{(1)}[f - f_{N-1}] \right| \\ \leq\ & 4\pi \sum_{|k| \geq N} |c_k|. \end{aligned}$$

For a real analytic 2π-periodic function, the result (7.6) shows that $|c_k|$ decays exponentially fast and, in turn, the error of the composite trapezoidal rule also converges exponentially fast to zero.

7.5 The fast Fourier transform (FFT)

In the following, we will describe a fast algorithm for performing the DFT (and, at the same time, the IDFT). Recall that $\omega_n = e^{-2\pi \mathrm{i}/n}$. Then

$$\omega_n^k = \omega_n^{k+n} \ \forall k \in \mathbb{Z}, \quad \omega_n^n = 1, \quad \omega_n^{n/2} = -1, \tag{7.9}$$

The most crucial property is the trivial relation $\omega_n^{2k} = \omega_{n/2}^k$ for even n, because it will allow us to relate entries from F_n to entries of the smaller matrix $F_{n/2}$.

Recall that the DFT is performed by multiplying a vector with the matrix $F_n = \left(\omega_n^{jk}\right)_{j,k=0}^{n-1}$. This matrix has a lot of structure, which can be exploited to accelerate matrix-vector multiplication. We illustrate this structure for $n = 6$. Then

$$F_6 = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & \omega_6 & \omega_6^2 & \omega_6^3 & \omega_6^4 & \omega_6^5 \\ 1 & \omega_6^2 & \omega_6^4 & \omega_6^6 & \omega_6^8 & \omega_6^{10} \\ 1 & \omega_6^3 & \omega_6^6 & \omega_6^9 & \omega_6^{12} & \omega_6^{15} \\ 1 & \omega_6^4 & \omega_6^8 & \omega_6^{12} & \omega_6^{16} & \omega_6^{20} \\ 1 & \omega_6^5 & \omega_6^{10} & \omega_6^{15} & \omega_6^{20} & \omega_6^{25} \end{pmatrix},$$

where $\omega_6 = e^{-2\pi \mathrm{i}/6} = e^{\pi \mathrm{i}/3}$. We reorder the rows such that the rows with even index appear first and then the rows with odd index. This can be achieved by defining

$$P_6 = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

and applying its transpose to F_6:

$$P_6^\mathsf{T} F_6 = \left(\begin{array}{ccc|ccc} 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & \omega_6^2 & \omega_6^4 & \omega_6^6 & \omega_6^8 & \omega_6^{10} \\ 1 & \omega_6^4 & \omega_6^8 & \omega_6^{12} & \omega_6^{16} & \omega_6^{20} \\ \hline 1 & \omega_6 & \omega_6^2 & \omega_6^3 & \omega_6^4 & \omega_6^5 \\ 1 & \omega_6^3 & \omega_6^6 & \omega_6^9 & \omega_6^{12} & \omega_6^{15} \\ 1 & \omega_6^5 & \omega_6^{10} & \omega_6^{15} & \omega_6^{20} & \omega_6^{25} \end{array}\right) = \left(\begin{array}{ccc|ccc} 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & \omega_6^2 & \omega_6^4 & 1 & \omega_6^2 & \omega_6^4 \\ 1 & \omega_6^4 & \omega_6^2 & 1 & \omega_6^4 & \omega_6^2 \\ \hline 1 & \omega_6 & \omega_6^2 & -1 & -\omega_6 & -\omega_6^2 \\ 1 & \omega_6^3 & \omega_6^6 & -1 & -\omega_6^3 & -\omega_6^6 \\ 1 & \omega_6^5 & \omega_6^4 & -1 & -\omega_6^5 & -\omega_6^4 \end{array}\right),$$

where we used the relation (7.9) multiple times. Because of $\omega_3^k = \omega_6^{2k}$ for $k \in \mathbb{Z}$, it follows that

$$P_6^\mathsf{T} F_6 = \left(\begin{array}{ccc|ccc} 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & \omega_3^1 & \omega_3^2 & 1 & \omega_3^1 & \omega_3^2 \\ 1 & \omega_3^2 & \omega_3^1 & 1 & \omega_3^2 & \omega_3^1 \\ \hline 1 & \omega_6 & \omega_6^2 & -1 & -\omega_6 & -\omega_6^2 \\ 1 & \omega_6\omega_3^1 & \omega_6^2\omega_3^2 & -1 & -\omega_6\omega_3^1 & -\omega_6^2\omega_3^2 \\ 1 & \omega_6\omega_3^2 & \omega_6^2\omega_3^1 & -1 & -\omega_6\omega_3^2 & -\omega_6^2\omega_3^1 \end{array}\right) = \begin{pmatrix} F_3 & F_3 \\ F_3\Omega_3 & -F_3\Omega_3 \end{pmatrix}$$

with

$$F_3 = \begin{pmatrix} 1 & 1 & 1 \\ 1 & \omega_3^1 & \omega_3^2 \\ 1 & \omega_3^2 & \omega_3^1 \end{pmatrix}, \quad \Omega_3 = \begin{pmatrix} 1 & & \\ & \omega_6 & \\ & & \omega_6^2 \end{pmatrix}.$$

This shows, for $n = 6$, that F_n is composed of four blocks $F_{n/2}$, combined with diagonal scaling and permutation. This relation generalizes to arbitrary even n.

Theorem 7.7 *For even $n \geq 2$, consider the permutation $\xi : \{0, \ldots, n-1\} \to \{0, \ldots, n-1\}$ with*

$$\xi : 0 \mapsto 0,\ 1 \mapsto 2,\ \ldots,\ \frac{n}{2} - 1 \mapsto n-2,\ \frac{n}{2} \mapsto 1,\ \frac{n}{2} + 1 \mapsto 3,\ \ldots,\ n-1 \mapsto n-1.$$

Let P_n be the permutation matrix belonging to this permutation, that is,

$$P_n = \begin{pmatrix} e_{\xi(0)} & e_{\xi(1)} & \cdots & e_{\xi(n-1)} \end{pmatrix},$$

where e_j is a vector of length n with entry j (for $j = 0, \ldots, n-1$) equal to 1 and all other entries equal to zero. Then

$$P_n^{\mathsf{T}} F_n = \begin{pmatrix} F_{n/2} & F_{n/2} \\ F_{n/2}\Omega_{n/2} & -F_{n/2}\Omega_{n/2} \end{pmatrix} = \begin{pmatrix} F_{n/2} & \\ & F_{n/2} \end{pmatrix} \begin{pmatrix} I_{n/2} & I_{n/2} \\ \Omega_{n/2} & -\Omega_{n/2} \end{pmatrix},$$

with

$$\Omega_{n/2} = \operatorname{diag}(\omega_n^0, \omega_n^1, \ldots, \omega_n^{n/2-1}).$$

Proof. This follows from a straightforward extension of the discussion above for $n = 6$, using relation (7.9). ∎

Theorem 7.7 can be used to perform a matrix-vector multiplication $F_n\mathbf{y}$ recursively. For this purpose, we partition the vector $\mathbf{y} = \begin{pmatrix} \mathbf{y}_1 \\ \mathbf{y}_2 \end{pmatrix}$ such that $\mathbf{y}_1, \mathbf{y}_2 \in \mathbb{C}^{n/2}$. According to Theorem 7.7 we can write $F_n\mathbf{y}$ as follows:

$$F_n\mathbf{y} = P_n \begin{pmatrix} F_{n/2} & \\ & F_{n/2} \end{pmatrix} \begin{pmatrix} I_{n/2} & I_{n/2} \\ \Omega_{n/2} & -\Omega_{n/2} \end{pmatrix} \begin{pmatrix} \mathbf{y}_1 \\ \mathbf{y}_2 \end{pmatrix} = P_n \begin{pmatrix} F_{n/2}(\mathbf{y}_1 + \mathbf{y}_2) \\ F_{n/2}\Omega_{n/2}(\mathbf{y}_1 - \mathbf{y}_2) \end{pmatrix}.$$

In other words, the multiplication with F_n can be reduced to two multiplications with $F_{n/2}$ and some cheap additional computations. Applying this recursion repeatedly, one arrives at the following algorithm when n is a power of two.[21]

[21] It is instructive to verify with a small example how `reshape` is used to effect the multiplication with P_n.

Algorithm 7.8
Input: Vector $\mathbf{y} \in \mathbb{C}^n$ where n is a power of 2.
Output: Matrix-vector product $\mathbf{z} = F_n\mathbf{y}$.

Partition $\mathbf{y} = \begin{pmatrix} \mathbf{y}_1 \\ \mathbf{y}_2 \end{pmatrix}$ with $\mathbf{y}_1, \mathbf{y}_2 \in \mathbb{C}^{n/2}$.
$\mathbf{z}_1 = \mathbf{y}_1 + \mathbf{y}_2$, $\mathbf{z}_2 = \Omega_{n/2}(\mathbf{y}_1 - \mathbf{y}_2)$.
Recursion: $\mathbf{z}_1 \leftarrow F_{n/2}\mathbf{z}_1$
Recursion: $\mathbf{z}_2 \leftarrow F_{n/2}\mathbf{z}_2$.
$\mathbf{z} = P_n \begin{pmatrix} \mathbf{z}_1 \\ \mathbf{z}_2 \end{pmatrix}$

PYTHON

```
import numpy as np
def myfft(y):
    n = y.shape[0]
    if (n == 1):  return y
    omega = np.exp(-2*np.pi*1j/n);
    mid   = int(n/2)
    z1    = y[0:mid]+y[mid:n];
    z2    = np.power(omega, range(0,mid))*(y[0:mid]-y[mid:n])
    z     = np.concatenate((myfft(z1), myfft(z2)))
    z     = np.reshape(np.reshape(z, (2,mid)), (n,), 'F') # permute
    return z
```

It is worth reflecting how the permutation is realized efficiently in the Python code.

The computational complexity of recursive algorithms can often be derived using the Master theorem (see Wikipedia). In the case of Algorithm 7.8, it is also quite easy to estimate the complexity directly. Let $A(n)$ denote the number of operations required by Algorithm 7.8 for a vector of length n. Ignoring the cost for computing the powers of ω_n (which can be computed beforehand and stored in a table), then[22] $A(n) = 5n + 2\,A(n/2)$. Recursive application of this formula yields

$$A(n) = 5n + 5n + 4\,A(n/4) = \cdots = 5kn + 2^k A(n/2^k).$$

Setting $k = \log_2 n$ we have $A(n/2^k) = A(n/n) = A(1) = 0$ and hence

$$A(n) = 5n \log_2 n.$$

Because this compares very favorably with the $O(n^2)$ complexity of general matrix-vector multiplication, one calls Algorithm 7.8 the Fast Fourier Transform (FFT).

Remark 7.9 *In Python, both NumPy and SciPy have functions implementing the FFT; the one by SciPy tends to be faster. The software package FFTW (Fastest Fourier Transform in the West, see* `http://www.fftw.org/`*) is a popular and well tuned implementation of the FFT. For $n = 2^k$ the algorithm by FFTW roughly corresponds to Algorithm 7.8. For $n \neq 2^k$ one needs to resort to other decompositions/factorizations of n; the asymptotic complexity remains $O(n \log_2 n)$ but the*

[22] A complex addition or subtraction counts 2 flops and a complex multiplication counts 6 flops.

constants can be significantly larger, especially if n has large prime factors. There is a Python wrapper for FFTW, called pyFFTW, which is more complicated to call. Once the package pyFFTW is installed, the FFT and inverse FFT can be compted as follows:

```
import pyfftw
#assume y is our signal
target_len = len(y)
a = pyfftw.empty_aligned(target_len, dtype='complex128')
b = pyfftw.empty_aligned(target_len, dtype='complex128')
#build a fft object to be called later
fft_object = pyfftw.FFTW(a, b)
a[:] = y
#call fft, both fft_a and b will be the result of fft
fft_a = fft_object()
c = pyfftw.empty_aligned(target_len, dtype='complex128')
#build an ifft object
ifft_object =  pyfftw.FFTW(b, c, direction='FFTW_BACKWARD')
#call ifft, both ifft_b and c will be the result of ifft
ifft_b = ifft_object()
```

7.6 Discrete cosine transform (DCT)*

We now assume that f is not only 2π-periodic but also even, that is,

$$f(x+2k\pi) = f(x), \qquad f(x) = f(-x), \qquad \forall k \in \mathbb{Z}, x \in \mathbb{R}.$$

It is then sufficient to restrict f to the interval $[-\pi, \pi]$. Its Fourier series, if convergent, reads as

$$f(x) = \frac{a_0}{2} + \sum_{k=1}^{\infty} a_k \cos(kx) \qquad \forall\ k \in \mathbb{Z}; \tag{7.10}$$

see (7.4). The following orthogonality relations hold:

Lemma 7.10 *For every $\ell, k \in \mathbb{Z}$*

$$\int_0^\pi \cos(\ell x)\cos(kx)\ \mathrm{d}x = \begin{cases} 0 & \text{if} \quad \ell \neq k, \\ \frac{\pi}{2} & \text{if} \quad \ell = k \neq 0, \\ \pi & \text{if} \quad \ell = k = 0. \end{cases}$$

The Fourier coefficients $(a_k)_{k\in\mathbb{Z}}$ can be recovered by multiplying (7.10) with $\cos(\ell x)$, integrating from 0 to π, and using Lemma 7.10, which yields

$$\int_0^\pi f(x)\cos(\ell x)\ \mathrm{d}x = \frac{a_0}{2}\int_0^\pi \cos(\ell x)\ \mathrm{d}x + \sum_{k=1}^{\infty} a_k \int_0^\pi \cos(kx)\cos(\ell x)\ \mathrm{d}x,$$

and thus

$$a_k = \frac{2}{\pi}\int_0^\pi f(x)\cos(kx)\ \mathrm{d}x \qquad \forall\ k \in \mathbb{N}. \tag{7.11}$$

Let us know suppose that f is known only at some discrete points in $[0, \pi]$:

$$x_j := \frac{(2j+1)\,\pi}{2N}, \qquad y_j := f(x_j) \qquad \forall\ j = 0, \dots, N-1. \tag{7.12}$$

The discrete counterpart of (7.10) becomes

$$y_j = \frac{z_0}{2} + \sum_{k=1}^{N-1} z_k \cos(kx_j), \qquad \forall\ j = 0, \dots, N-1, \tag{7.13}$$

and the following discrete orthogonality relations hold.

Lemma 7.11 *For all $\ell, k \in \mathbb{Z}$*

$$\sum_{j=0}^{N-1} \cos(\ell x_j)\cos(kx_j) = \begin{cases} 0 & \text{if} \quad \ell \neq k, \\ \frac{N}{2} & \text{if} \quad \ell = k \neq 0, \\ N & \text{if} \quad \ell = k = 0. \end{cases}$$

Proof. EFY. □

We now aim at computing $(z_k)_{k=0}^{N-1}$ such that (7.13) holds. Mimicking the procedure above in a discrete setting, we multiply (7.13) with $\cos(\ell x_j)$, summing from $j = 0$ to $j = N-1$ and employ Lemma 7.11, which yields

$$\sum_{j=0}^{N-1} y_j \cos(\ell x_j) = \frac{z_0}{2} \sum_{j=0}^{N-1} \cos(\ell x_j) + \sum_{k=1}^{N-1} z_k \sum_{j=0}^{N-1} \cos(kx_j)\cos(\ell x_j).$$

Thus,

$$z_k = \frac{2}{N} \sum_{j=0}^{N-1} y_j \cos(kx_j) \qquad \forall\ k = 0, \dots, N-1. \tag{7.14}$$

Definition 7.12 (Discrete Cosine Transform) $\left(\hat{f}_N(k)\right)_{k\in\mathbb{Z}}$ *is called the* discrete cosine transform (DCT) *of f with respect to the discretization $(x_j)_{j=0}^{N-1}$ defined in* (7.12).

Remark 7.13 *It is possible to interpret the DCT as the approximation of* (7.11)*, by using the composite midpoint quadrature rule.*

Definition 7.14 (Inverse Discrete Cosine Transform) *The sequence $(y_j)_{j\in\mathbb{Z}}$ is called* inverse discrete cosine transform (IDCT) *of f with respect to the discretization $(x_j)_{j=0}^{N-1}$.*

7.6.1 The JPEG: an image compression standard⋆

The light intensity measured by a camera is generally sampled over a rectangular array of picture elements called *pixels*. Let us consider an image consisting of M^2

pixels, such that each couple (i,j), for $i,j=0,\dots,M-1$, corresponds to a pixel. For the sake of simplicity of the discussion, let us focus on the case of black and white pictures. A BW picture can be thought as a function $Y : M \times M \to \{0,\dots,255\}$, $(i,j) \mapsto Y(i,j)$, where $Y(i,j)$ represents the gray level at the pixel (i,j). Thus, $M^2 \cdot 8$ bits (the number 255 is 11111111 in base 2) per pixel are needed in order to store a picture. In principle M may be very large: a typical high resolution color picture for the web contains on the order of one millions pixels. However, state-of-the-art techniques can compress typical images from 1/10 to 1/50 without visibly affecting image quality. One of the most popular procedures is indeed JPEG (N. Ahmed, T. Natarajan, K. R. Rao, 1974). Let us subdivide the image into 8×8 blocks. For each block $(Y_{i_k,j_\ell})_{k,\ell=0}^{7}$, where $(i_k)_{k=0}^{7}$, $(j_\ell)_{\ell=0}^{7} \subset \{0,\dots,M\}$ are subsequences of consecutive indices, we can apply the DCT, passing from the *spatial domain* to the *frequency domain*. In this way every 8×8 block of source image sample is effectively a discrete signal with 64 entries, which is a function of the two spatial dimensions, denoted for the sake of simplicity of the notation as $(Y_{i,j})_{i,j=0}^{N}$, with $N=7$. By analogy to (7.13), we want to find $(Z_{k,\ell})_{k,\ell=0}^{N-1}$ such that

$$Y_{i,j} = \sum_{k=0}^{N-1}\sum_{\ell=0}^{N-1} \tilde{Z}_{k,\ell}\cos(kx_i)\cos(x_j), \qquad i,j=0,\dots,N-1, \tag{7.15}$$

where, in order to compensate for the factor 1/2, we employ the notation $\tilde{Z}_{0,0} = Z_{0,0}/4$, $\tilde{Z}_{k,0} = Z_{k,0}/2$, $\tilde{Z}_{0,\ell} = Z_{0,\ell}/2$, $\tilde{Z}_{k,\ell} = Z_{k,\ell}$, $k,\ell \geq 1$. In Figure 7.2 the reader can see the representation of the 64 basis functions $(\cos(kx_i)\cos(\ell x_j))_{k,\ell=0}^{N-1}$ on a single 8×8 block. In particular, the columns correspond to the index k and the rows to the index ℓ, $k,\ell = 0,1,\dots,N=7$. Increasing k, respectively ℓ, corresponds to higher oscillations in the x-direction, respectively y-direction.

The partition $(x_j)_{j=0}^{N-1}$ of $[0,\pi]$ is the same as in (7.12). We multiply (7.15) by $\cos(kx_i)\cos(\ell x_j)$, sum over $i,j=0,\dots,N-1$ and use the discrete orthogonality relations (7.11). In this way we get the $2D$ counterpart of (7.14), that is

$$Z_{k,\ell} = \frac{4}{N^2}\sum_{i=0}^{N-1}\sum_{j=0}^{N-1} Y_{i,j}\cos(kx_i)\cos(\ell x_j), \qquad k,\ell=0,\dots,N-1. \tag{7.16}$$

The DCT takes the signal representing the block as an input and decomposes it into 64 orthogonal basis signals, each one of them corresponding to a particular *frequency*. The value of a frequency reflects the size and speed of a change as you can see from Figure 7.2. The output is the collection of 64 DCT coefficients, representing the *amplitudes* of these signals. The first coefficient, corresponding to the zero frequency in both spatial dimensions, is often called *DC* (*direct current*). The remaining 63 entries are called *AC* (*alternating currents*). The high frequencies represent the high contrast areas in the image, i.e. rapid changes in pixel intensity. Note that in a classic image there is a high continuity between pixel values. Hence it turns out that the numerically important AC coefficients can be found in the square 4×4 around the DC coefficient.

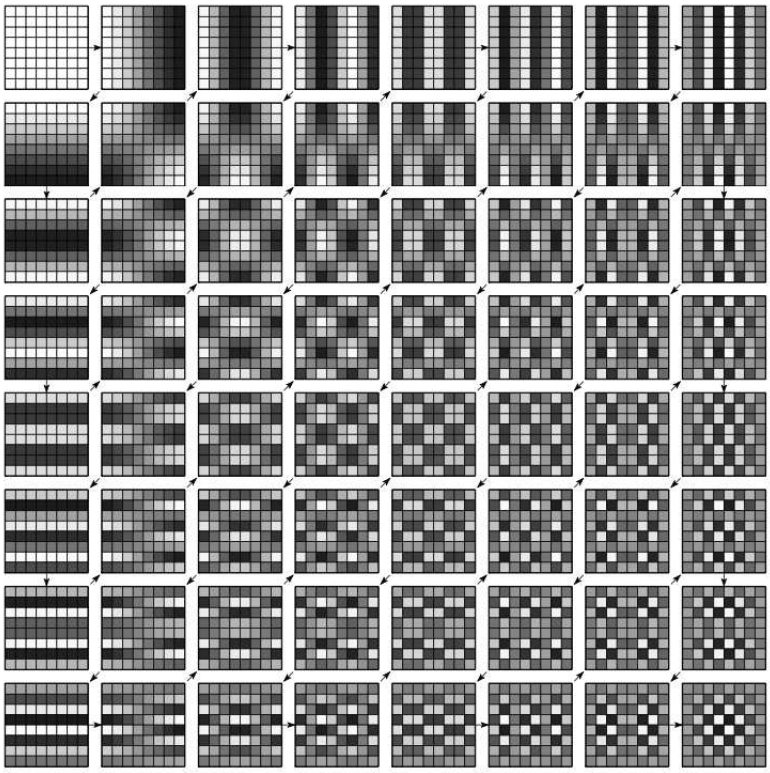


Figure 7.2: Representation of the basis functions $(\cos(kx_i)\cos(\ell x_j))_{k,\ell=0}^{N-1}$.

Once the DCT coefficients are obtained, we would like to numerically represent them with no greater precision than is necessary to achieve the desired image quality. This step is called *quantization* of the signal. Each of the 64 DCT coefficients is quantized according to a 8×8 matrix T called, *quantization matrix*, with integer entries between 1 and 255, which is specified by the user and is conceived to provide greater resolution to more perceptible frequency components on less perceptible ones. In formulas, the quantization step reads as

$$Z_{k,\ell} \mapsto \lfloor \frac{Z_{k,\ell}}{T_{k,\ell}} \rfloor, \qquad k, \ell = 0, \ldots, N-1. \tag{7.17}$$

Note that since the entries of T corresponding to the high frequencies are usually high and because we use the function "floor" in (7.17), the resulting high frequency coefficients will be zero. Let $\mathcal{Z}_N$ be the 8×8 matrix of coefficients after quantization. Let us go through its entries by following a *zig-zag path* (see Figure 7.3) in order to construct a vector of 64 coefficients. We just mention that this trick allows to further reduce the amount of information to be compressed of the image by placing low-frequency coefficients (more likely to be non-zero) before high frequency coefficients.

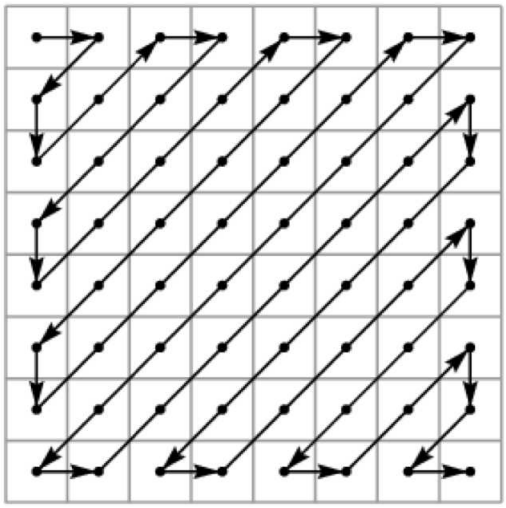
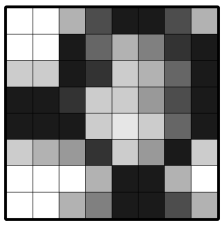
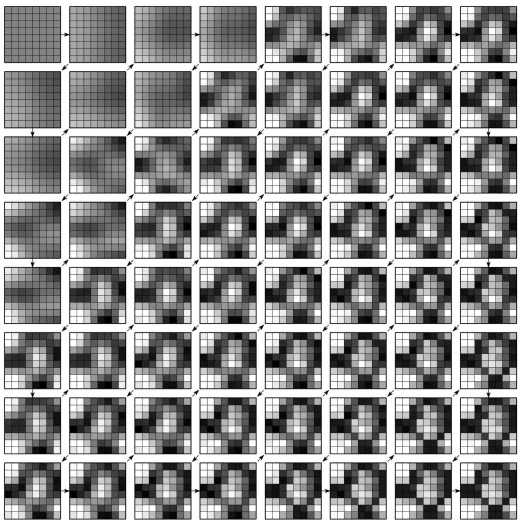


Figure 7.3: Zig-zag path used for the encoding of images in the JPEG.

The procedure described above can be reversed by applying the IDCT which takes the encoded coefficients and reconstructs the image signal by summing the basis signals. However, because of the quantization step, there is an inevitable loss of information. We have indeed introduced a numerical error and we made the whole procedure irreversible. That is why the JPEG is said to be a *lossy compression* technique. In Figure 7.4a we can see an 8×8 block from an image and in Figure 7.4b its decoding process that follows the zig-zag path.



(a) Target block.



(b) Decoding using the zig-zag procedure.

Figure 7.4: The reconstruction process of a sample 8×8 block from an image.

Bibliography

[1] Peter Deuflhard and Andreas Hohmann. *Numerical analysis in modern scientific computing*, volume 43 of *Texts in Applied Mathematics*. Springer-Verlag, New York, second edition, 2003.

[2] Gene H. Golub and Charles F. Van Loan. *Matrix computations*. Johns Hopkins Studies in the Mathematical Sciences. Johns Hopkins University Press, Baltimore, MD, fourth edition, 2013.

[3] Nicholas J. Higham. *Accuracy and stability of numerical algorithms*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, second edition, 2002.

[4] Thomas Huckle and Tobias Neckel. *Bits and bugs*, volume 29 of *Software, Environments, and Tools*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 2019.

[5] Alfio Quarteroni, Riccardo Sacco, and Fausto Saleri. *Méthodes numériques*. Springer-Verlag Italia, Milan, 2007. Algorithmes, analyse et applications. [Algorithms, analysis and applications], Translated from the 1998 Italian original by Jean-Frédéric Gerbeau.

[6] Lloyd N. Trefethen. *Approximation theory and approximation practice*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 2013.