Lecture notes to the course

Numerical Methods I

Clemens Kirisits

February 10, 2017

Preface

These lecture notes are intended as a written companion to the course "Numerical Methods I" taught in October and November 2016 at the University of Vienna. The course is part of an interdisciplinary masters programme called *Computational Science*. As such it targets an audience with diverse backgrounds. The course is supposed to acquaint the students with a basic knowledge of numerical analysis and scientific computing.

For the creation of these notes the following references have proven extremely helpful. They are highly recommended for further reading.

- *Matrix Computations*, Gene H. Golub and Charles F. Van Loan
- *Grundlagen der Numerischen Mathematik und des Wissenschaftlichen Rechnens* by Martin Hanke-Bourgeois
- *Accuracy and Stability of Numerical Algorithms*, Nicholas Higham
- *Princeton Companion to Applied Mathematics*, Nicholas Higham (ed.)
- *Introduction to Scientific Computing*, Charles F. Van Loan
- *Numerical Linear Algebra*, Lloyd N. Trefethen and David Bau, III

In addition I have used the lecture notes to the course *Numerische Mathematik* held by Winfried Auzinger in winter term 2005/06 at TU Wien.

Clemens Kirisits

Contents

Preface iii

1 Basic Concepts of Numerical Analysis 1
- 1.1 Floating Point Arithmetic . . . 2
- 1.2 Cancellation . . . 6
- 1.3 Condition of a Problem . . . 7
- 1.4 Condition Number of a Matrix . . . 9
- 1.5 Stability of an Algorithm . . . 11

2 Numerical Linear Algebra 17
- 2.1 Triangular Systems . . . 18
  - 2.1.1 Forward Substitution . . . 18
  - 2.1.2 Back Substitution . . . 21
- 2.2 LU Factorization . . . 22
  - 2.2.1 Gaussian Elimination . . . 22
  - 2.2.2 Pivoting . . . 28
- 2.3 Cholesky Factorization . . . 33
  - 2.3.1 SPD Matrices . . . 34
  - 2.3.2 Symmetric Gaussian Elimination . . . 34
  - 2.3.3 Algorithm . . . 37
- 2.4 QR Factorization . . . 38
  - 2.4.1 Gram-Schmidt . . . 39
  - 2.4.2 Modified Gram-Schmidt . . . 42
  - 2.4.3 Householder's Method . . . 43
  - 2.4.4 Comparison of Algorithms for Solving $Ax = b$ . . . 47
- 2.5 Linear Least Squares Problems . . . 48
  - 2.5.1 Numerical Solution . . . 49

3 Interpolation 51
- 3.1 Polynomial Interpolation . . . 51
  - 3.1.1 The General Problem . . . 52
  - 3.1.2 Error Estimates . . . 55
- 3.2 Spline Interpolation . . . 57
  - 3.2.1 Linear Splines . . . 58
  - 3.2.2 Cubic Splines . . . 60

3.3 Trigonometric Interpolation 63
3.3.1 Fast Fourier Transform 65

4 Numerical Integration 69
4.1 Newton-Cotes Formulas 70
4.1.1 Error estimates 72
4.2 Composite Rules 73
4.3 Gauss Quadrature 74

Appendix on Linear Algebra 77

Chapter 1

Basic Concepts of Numerical Analysis

Numerical analysis is the study of algorithms for solving problems of mathematical analysis. Its importance and usefulness are best understood by regarding it as a subdiscipline of applied mathematics. In the words of mathematician Garrett Birkhoff "mathematics becomes applied when it is used to solve real-world problems." Frequently, solving a problem in applied mathematics includes (some of) the following steps.

Modelling. A real-world problem is translated into a mathematical one. For example, the distribution of heat in a certain material can be modelled by a partial differential equation called heat equation.

Analysis. The mathematical problem is analysed. This step often involves the question of well-posedness: Does the problem have a unique solution that depends continuously on the data?

Approximation. Very often, solutions to mathematical problems cannot be computed directly. The problem must be approximated by a simpler one. Discretization, i.e. replacing continuous objects by discrete counterparts, is an important example of an approximation process.

Algorithm. The reduced problem is broken down into an algorithm: a sequence of simple steps that can be followed through mechanically.

Software. Software must be written so that the algorithm can be executed on a computer.

Validation. If it is possible to make measurements of the real-world phenomenon, then these measurements can be used to validate the mathematical model by comparing them to the computer output.

Prediction. Finally, the model can be used to gain new insights about the original problem that would be too costly or even impossible to obtain otherwise.

Numerical analysis mainly occupies the steps *Approximation* and *Algorithm*,[1] thereby bridging the gap between mathematical problems and computers. In summary, numerical methods address the questions (i) how mathematical problems can be turned into a form solvable by a computer, and (ii) how numerical solutions can be computed efficiently and accurately.

Suppose, while validating your model for a particular problem, you realize that your computer output is significantly wrong. For the sake of the argument assume that your mathematical model is accurate, that your algorithm is mathematically correct and your software is a faithful implementation of the algorithm. In this case the observed errors might have the following sources.

Data uncertainty. Many algorithms are fed on data produced by real-world measurements or by previous computations. In both cases the data can only be accurate to a certain degree.

Discretization. Since discretization is a kind of approximation, it typically introduces errors. These errors are sometimes called truncation errors.

Roundoff. On computers real numbers are replaced by floating point numbers. Therefore, both data input and internal computations lead to roundoff errors.

While awareness of data uncertainty is always necessary, the focus of numerical analysis usually is on controlling truncation and roundoff errors.

1.1 Floating Point Arithmetic

In the following we will often want to assess the quality of an approximation $\tilde{x} = x + \Delta x = x(1 + \delta x)$ to a real number x. Two very common measures are the *absolute error* of $\tilde{x}$

$$|x - \tilde{x}| = |\Delta x|$$

and the *relative error* of $\tilde{x}$

$$\frac{|x - \tilde{x}|}{|x|} = \frac{|\Delta x|}{|x|} = |\delta x|.$$

Designing and analysing good algorithms requires knowledge of how computers handle numbers. The following example is borrowed from *Matrix Computations* by Gene H. Golub and Charles F. Van Loan.

Example 1.1 (3-digit calculator)**.** Suppose you have a very simple calculator

[1]However, writing software is an important aspect of numerical analysis as well, since it allows you, for instance, to compare the theoretical properties of an algorithm with its practical behaviour.

which represents numbers in the following way:

$$x = \pm 0.d_1 d_2 d_3 \times 10^e, \quad \text{where} \begin{cases} d_1 & \in \{1, \dots, 9\} \\ d_2 & \in \{0, \dots, 9\} \\ d_3 & \in \{0, \dots, 9\} \\ e & \in \{-9, \dots, 9\}. \end{cases}$$

A few observations are in order:

- The first digit d_1 does not take the value 0 for reasons of uniqueness. If we allowed $d_1 = 0$, then we could write certain numbers in more than one way, e.g. $74 = 0.74 \times 10^2 = 0.074 \times 10^3$.
- Clearly, zero must be representable. However, since $d_1 \neq 0$, we need an exception to do so, for example 0.00×10^0.
- Our calculator can only represent finitely many numbers: $2 \times 9 \times 10 \times 10 \times 19 + 1 = 34201$.
- In particular, there is a largest number (0.999×10^9) and a smallest positive number (0.100×10^{-9}). Notice that with the convention $d_1 \neq 0$ we have "lost" the even smaller numbers of the form $0.0d_2d_3 \times 10^{-9}$.
- The calculator's precision depends on the number of digits, which in our case is three. In order to represent the number 123456, for example, the toy calculator cannot do better than 0.123×10^6. The resulting relative error is of order 10^{-3}.
- Similarly, results of calculations generally will have to be rounded in order to fit the 3-digit format. For instance, $(0.123 \times 10^2) * (0.456 \times 10^3) = 5608.8 \approx 0.561 \times 10^4$.
- The set of representable numbers is not equispaced. Instead, the spacing increases by a power of 10 at every power of 10. Between $0.100 \times 10^{e+1}$ and $0.100 \times 10^{e+2}$ the spacing is 10^{e-1}.

In the example above we have seen a particular instance of a *normalized floating point number system.* A general floating point number system is a subset $\mathbb{F}$ of $\mathbb{R}$ whose nonzero elements have the form

$$x = \pm \Big(\frac{d_1}{\beta^1} + \cdots + \frac{d_t}{\beta^t} \Big) \times \beta^e. \tag{1.1}$$

Here, $\beta \in \{2, 3, \dots\}$ is the *base* and $t \in \mathbb{N}$ the *precision* of $\mathbb{F}$. In contrast to β and t, the exponent e is not fixed but can take any integer value in the *exponent range* $e_{\min} \leq e \leq e_{\max}$. Similarly, each *digit* d_i can vary in the set $\{0, \dots, \beta - 1\}$. If the number system is *normalized*, then $d_1 \neq 0$. The number zero does not have a normalized representation. The gap between 1 and the next largest floating point number is called *machine epsilon* $\epsilon_M = \beta^{1-t}$. The *unit roundoff*

$u = \frac{1}{2}\epsilon_M$ gives a worst case bound on the relative error when approximating real numbers by floating point numbers. The following theorem summarizes some of the properties of a normalized floating point number system.

Theorem 1.1. *For $\mathbb{F} = \mathbb{F}(\beta, t, e_{\min}, e_{\max})$ the following statements are true.*

1. *Every $x \in \mathbb{F} \setminus \{0\}$ has a unique representation* (1.1).
2. *The largest and smallest positive elements of $\mathbb{F}$, respectively, are given by*
$$\begin{aligned} x_{\max} &= \beta^{e_{\max}}(1 - \beta^{-t}), \\ x_{\min} &= \beta^{e_{\min}-1}. \end{aligned}$$
3. *The spacing between adjacent floating point numbers in the range $[\beta^{e-1}, \beta^e]$ is $\epsilon_M \beta^{e-1}$.*
4. *Every real number x satisfying $|x| \in [x_{\min}, x_{\max}]$ can be approximated by a $y \in \mathbb{F}$ such that the relative error stays below the unit roundoff. That is,*
$$\min_{y \in \mathbb{F}} \frac{|x - y|}{|x|} \leq u.$$

Proof. Exercise. □

For every floating point number system $\mathbb{F}$ we define a *rounding* function $\mathrm{fl} : \mathbb{R} \to \mathbb{F}$ in the following way. It maps zero to zero and every other real number to its best respective representation (1.1) without restriction on the exponent. If the number to be rounded is the exact midpoint of two adjacent floating point numbers, then we need a rule telling us which one to choose. Of course, we are mainly interested in the cases when $\mathrm{fl}(x) \in \mathbb{F}$. When $|\mathrm{fl}(x)| < x_{\min}$, we say that $\mathrm{fl}(x)$ *underflows*. When $|\mathrm{fl}(x)| > x_{\max}$, then it *overflows*. The following statement is an immediate consequence of item 4 in Thm. 1.1.

Corollary 1.1. *For every $x \in [-x_{\max}, -x_{\min}] \cup \{0\} \cup [x_{\min}, x_{\max}]$ there is a $\delta \in [-u, u]$ such that*
$$\mathrm{fl}(x) = (1 + \delta)x. \tag{1.2}$$

Note that δ is just the relative error of $\mathrm{fl}(x)$. Assuming that the rounding function is correctly implemented on our computer, equation (1.2) basically tells us what happens to a real number when we feed it into our computer (as long as there is no over- or underflow). This is the first step towards analysing the effects of rounding errors in numerical algorithms. The next step is to quantify the error committed during an arithmetic operation with floating point numbers. How large the error of a floating point operation, a *flop*, is depends on the way it is implemented on your computer. A good design principle is formulated in the following assumption.

Assumption 1.1. Let $\circ$ stand for any of the four basic arithmetic operations $(+,-,\times,/)$ and denote by $\circledcirc$ its floating point analogue. For all $x, y \in \mathbb{F}$ satisfying

$$x \circ y \in [-x_{\max}, -x_{\min}] \cup \{0\} \cup [x_{\min}, x_{\max}]$$

we have

$$x \circledcirc y = \mathrm{fl}(x \circ y). \tag{1.3}$$

This assumption is essentially a mathematical definition of the basic floating point operations. It states that the result of any such operation (that does not lead to over- or underflow) should be the same as the rounded result of the *exact* operation. On a machine that satisfies both (1.2) and (1.3) floating point operations are exact up to a relative error of size at most u.

Normalized systems $\mathbb{F}$ can be extended by *subnormal numbers*, also called *denormalized numbers.* These numbers are characterized by having minimal exponent $e = e_{\min}$ and $d_1 = 0$. The extended floating point number system $\hat{\mathbb{F}} = \hat{\mathbb{F}}(\beta, t, e_{\min}, e_{\max})$ is thus given by

$$\hat{\mathbb{F}} = \mathbb{F} \cup \Big\{ \pm \Big(\frac{d_2}{\beta^2} + \cdots + \frac{d_t}{\beta^t}\Big) \times \beta^{e_{\min}} : 0 \le d_i \le \beta - 1 \Big\}.$$

The smallest positive subnormal number is $\beta^{e_{\min}-t}$, which usually is much smaller than $x_{\min}$. Note that in contrast to normalized numbers, subnormal numbers are equidistant. Because of this, they do not satisfy equation (1.2). Instead the relative approximation error of $\mathrm{fl}(x)$ grows larger the smaller $|x|$ becomes.

Example 1.2 (IEEE standard)**.** Nowadays, on most computers floating point arithmetic is in conformance with the so-called IEEE 754 standard which supports two standard floating number systems: the *single format*

$$\hat{\mathbb{F}}(2, 24, -125, 128)$$

and the *double format*

$$\hat{\mathbb{F}}(2, 53, -1021, 1024).$$

This standard also satisfies Assumption 1.1 with rounding to even, i.e. $d_t = 0$.

Example 1.3 (Non-associativity of floating point arithmetic)**.** We return to our toy calculator. In accordance with Assumption 1.1 we have

$$((1000 \oplus 4) \oplus 4) = \mathrm{fl}(\mathrm{fl}(1000 + 4) + 4) = \mathrm{fl}(1000 + 4) = 1000.$$

If we change the order of addition, then

$$(1000 \oplus (4 \oplus 4)) = \mathrm{fl}(1000 + \mathrm{fl}(4 + 4)) = \mathrm{fl}(1000 + 8) = 1010.$$

Similar examples can be constructed for the other three operations. From this simple observation we can already conclude the following fundamental fact: Mathematically equivalent algorithms in general do not lead to the same results!

1.2 Cancellation

Cancellation is maybe the most common reason for numerical instability. It happens when we compute the difference between two almost equal floating point numbers. Leading digits are cancelled out, while trailing digits, which are more likely to be erroneous, become significant.

Example 1.4. Consider the following computation

$$(236 + 2340) - 2560 = 2576 - 2560 = 16.$$

Performing the same computation on the 3-digit calculator we get

$$(236 \oplus 2340) \ominus 2560 = 2580 \ominus 2560 = 20,$$

which has a relative error of 25%. It is very important to realize that the reason for this is not poor accuracy of floating point subtraction. On the contrary, the subtraction is done exactly! But it brings into prominence the error of the previous addition, which was not exact. Recalling Example 1.3 we can avoid the error amplification by changing the order of addition and subtraction

$$(236 \oplus (2340 \ominus 2560) = 236 \oplus (-220) = 16.$$

Let us look at cancellation in a more general way. Let $\tilde{x} = x(1 + \delta x)$ be a contaminated version of x and $\tilde{y} = y(1 + \delta y)$ a contaminated version of y. We would like to compute $z = x - y$. The relative error of the approximation $\tilde{z} = \tilde{x} - \tilde{y}$ can be bounded in the following way

$$\frac{|\tilde{z} - z|}{|z|} = \frac{|x\delta x - y\delta y|}{|x - y|} \leq \max(|\delta x|, |\delta y|)\frac{|x| + |y|}{|x - y|}. \tag{1.4}$$

This inequality is sharp, that is, there are real numbers $x, y, \delta x, \delta y$ with $x \neq y$ such that the inequality is an equality. Thus, in the worst case the relative error of $\tilde{z}$ is basically a large multiple of the original relative errors $\delta x, \delta y$. Note that the fraction on the right hand side is large, if $|x - y| \ll |x| + |y|$, i.e. if x and y are very close in a relative sense, which is exactly when cancellation occurs.

The reasoning above also shows that cancellation is unproblematic, if we can be sure that the numbers are error-free ($\delta x = \delta y = 0$). Cancellation (with erroneous data) can also be neglected, if it does not really influence the final result of a computation. We illustrate this point with an example.

Example 1.5. We extend the previous example by an addition with a large number:

$$((236 + 2340) - 2560) + 10^8 = (2576 - 2560) + 10^8 = 16 + 10^8.$$

With 3-digit precision we get

$$((236 \oplus 2340) \ominus 2560) \oplus 10^8 = (2580 \ominus 2560) \oplus 10^8 = 10^8,$$

and the relative error is negligible.

Example 1.6 (Roots of quadratic polynomials)**.** Consider the problem of computing the roots of the quadratic polynomial x^2+px+q. The standard formula

$$x_{1,2} = -p/2 \pm \sqrt{p^2/4 - q}$$

contains two possible sources of cancellation. First, if $q \approx 0$, then one of the two solutions (depending on the sign of p) is affected. This problem can be eliminated by first computing the other solution, i.e. the one which is larger in absolute value, and then using Vieta's formula $x_1 x_2 = q$ to compute the smaller one. Second, cancellation can occur when $q \approx p^2/4$. This happens for polynomials with almost identical roots. This case is more severe, as there is no mathematical trick to overcome the loss of accuracy.

Unavoidable cancellations often indicate an ill-conditioned problem. Conditioning is the topic of Section 1.3.

1.3 Condition of a Problem

The condition of a problem describes its behaviour under perturbations. A problem is *ill-conditioned*, if small perturbations of the input can lead to drastic changes of the output. On the other hand, if all small perturbations of the input lead only to small changes in the output, the problem is well-conditioned.

Abstractly, we can view a problem as a function $f : X \to Y$, which must be evaluated at $x \in X$. Here, X is the space of inputs and Y the space of outputs, both of which are assumed to be normed vector spaces. Using this notation we say the problem of evaluating f at x is *ill-conditioned with respect to the absolute error*, if there is a small perturbation $\tilde{x} \in X$ of x, for example $\tilde{x} = \mathrm{fl}(x)$, such that

$$\|f(\tilde{x}) - f(x)\| \gg \|\tilde{x} - x\|. \tag{1.5}$$

The relation $\gg$ stands for "much larger than." Frequently, relative errors are more appropriate than absolute ones.[2] We call a problem *ill-conditioned with respect to the relative error*, if instead

$$\frac{\|f(\tilde{x}) - f(x)\|}{\|f(x)\|} \gg \frac{\|\tilde{x} - x\|}{\|x\|}. \tag{1.6}$$

Clearly, words like "small" and "large" are not mathematically precise and depend very much on the context. However, a left-hand side several orders of magnitude larger than the right-hand side very often indicates an ill-conditioned problem.

For differentiable functions an obvious measure of sensitivity with respect to a change in the argument is the derivative f'. We therefore define the *absolute* and *relative condition numbers* of the problem $x \mapsto f(x)$ as

$$\kappa_{\text{abs}}(x) = \|f'(x)\| \tag{1.7}$$

[2]Recall that the function fl introduces relative errors!

and

$$\kappa_{\mathrm{rel}}(x) = \frac{\|f'(x)\|\|x\|}{\|f(x)\|}, \tag{1.8}$$

respectively. A problem is well-conditioned, if its condition number is small. For the relative condition number this can mean that it is of order 10^2 or below.

Note that we have not defined the expression $\|f'(x)\|$ or what differentiability means for a function between general normed spaces X and Y. This shall not concern us any further. It will suffice to know that for $f : \mathbb{R}^n \to \mathbb{R}^m$ we replace $\|f'(x)\|$ in (1.7) and (1.8) with $\|J(x)\|$, where

$$J = \left(\frac{\partial f_i}{\partial x_j}\right)_{ij} \in \mathbb{R}^{m\times n}$$

is the *Jacobian matrix* of f, and the norm in this case is the *induced matrix norm.*

Example 1.7 (Division by two)**.** We consider the problem of dividing a number by two. Thus, $f(x) = x/2$ and $f'(x) = 1/2$ for all $x \in \mathbb{R}$. The relative condition number is given by

$$\kappa_{\mathrm{rel}}(x) = \frac{\frac{1}{2}|x|}{|\frac{x}{2}|} = 1,$$

which indicates a well-conditioned problem.

Example 1.8 (Cancellation)**.** Consider the function $f : \mathbb{R}^2 \to \mathbb{R}, (x, y) \mapsto x-y$. The Jacobian is given by $(1, -1)$. Endowing $\mathbb{R}^2$ with the 1-norm, the induced norm of J equals 1. Hence

$$\kappa_{\mathrm{rel}}(x, y) = \frac{\|J(x, y)\|\|(x, y)\|_1}{|f(x)|} = \frac{|x| + |y|}{|x - y|},$$

which is large for $x \approx y$. We conclude that the subtraction of nearly equal numbers is ill-conditioned with respect to the relative error. Revisiting our brief analysis from (1.4) we see that the behaviour observed there is captured well by the concept of conditioning. The amplifying factor turns out to be the relative condition number.

Example 1.9 (Quadratic polynomials revisited)**.** Examples 1.6 and 1.8 strongly suggest that computing the roots of a polynomial with $p^2/4 \approx q$ is an ill-conditioned problem. We avoid the tedious computation of κ_{rel} in this case. Instead we only consider the polynomial $x^2 - 2x + 1 = (x - 1)^2$ and show that a large change in the roots can be caused by much smaller perturbation of the coefficients.

Replacing the double root $x_{1,2} = 1$ of the original polynomial by $x_{1,2} = 1 \pm \Delta x$ we calculate

$$(x - 1 + \Delta x)(x - 1 - \Delta x) = x^2 - 2x + 1 - (\Delta x)^2.$$

Reading this equation backwards, we see that a perturbation of q by $(\Delta x)^2$ changes the roots by Δx. However, if $(\Delta x)^2 \ll 1$ then $|\Delta x| \gg (\Delta x)^2$ and both (1.5) and (1.6) are satisfied. Note that absolute and relative conditioning are basically equivalent for this problem, because input $(p, q) = (-2, 1)$ and output $(x_1, x_2) = (1, 1)$ are of the same order.

1.4 Condition Number of a Matrix

Consider the problem of multiplying $x \in \mathbb{R}^n$ with a matrix $A \in \mathbb{R}^{m\times n}$. The Jacobian of the map $f : x \mapsto Ax$ is A itself. Thus

$$\kappa_{\text{rel}}(x) = \frac{\|A\|\|x\|}{\|Ax\|}.$$

Now assume that $m = n$ and that A is invertible. Using the definition of induced matrix norm for the inverse A^{-1} we can obtain a sharp upper bound for $\kappa_{\text{rel}}(x)$ which is independent of x

$$\frac{\|A\|\|x\|}{\|Ax\|} \le \|A\| \sup_{x\in\mathbb{R}^n} \frac{\|x\|}{\|Ax\|} = \|A\| \sup_{y\in\mathbb{R}^n} \frac{\|A^{-1}y\|}{\|y\|} = \|A\|\|A^{-1}\| =: \kappa.$$

The number $\kappa = \kappa(A)$ is called the *condition number of* A.

Now consider the problem of solving the linear system $Ax = b$ for given $b \in \mathbb{R}^n$. Mathematically this is equivalent to computing $f : b \mapsto A^{-1}b$. As before

$$\kappa_{\text{rel}}(b) = \frac{\|A^{-1}\|\|b\|}{\|A^{-1}b\|} \le \|A^{-1}\|\|A\| = \kappa.$$

Thus, the condition number of A not only controls how perturbations in x affect $b = Ax$, but also how perturbations in the right hand side b affect the solution x of the linear system $Ax = b$.

Finally, we want to find out how changes in A affect the solution x. So we consider the problem $f : A \mapsto A^{-1}b$. Let ΔA be an infinitesimal perturbation of A. Then there must be an infinitesimal Δx such that

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

Expanding the left hand side, using $Ax = b$ and neglecting the product $\Delta A\Delta x$ we get

$$A\Delta x + \Delta Ax = 0$$

or equivalently

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

Now we take norms on both sides and multiply with $\|A\|$. This yields

$$\|\Delta x\|\|A\| = \|A^{-1}\Delta Ax\|\|A\| \le \|A^{-1}\|\|\Delta A\|\|x\|\|A\|$$

and upon division by $\|\Delta A\|\|x\|$

$$\frac{\|\Delta x\|\|A\|}{\|\Delta A\|\|x\|} \leq \kappa.$$

Since $\|\Delta x\|/\|\Delta A\|$ is essentially $\|f'(A)\|$, the left-hand side is nothing but the relative condition number. Again it can be shown that the inequality is sharp. In summary, the condition number of A controls the sensitivity of three problems: $x \mapsto b$, $b \mapsto x$ and $A \mapsto x$. Hence its central role in numerical linear algebra.

Using the properties of the induced matrix norm (see the appendix), the condition number of A equals

$$\kappa(A) = \frac{\max_{\|x\|=1} \|Ax\|}{\min_{\|x\|=1} \|Ax\|} \geq 1$$

Geometrically, it measures how much A distorts the unit circle, and by the linearity of A, the whole space.

Example 1.10 (Condition number for the spectral norm.)**.** Since every induced matrix norm depends on the chosen vector norms, so does the corresponding condition number. For the spectral norm (cf. the appendix) we get the ratio of largest to smallest singular value

$$\kappa_2(A) = \|A\|_2\|A^{-1}\|_2 = \frac{\sigma_{\max}(A)}{\sigma_{\min}(A)}.$$

If A is a self-adjoint matrix, then this fraction in fact equals the ratio of eigenvalues

$$\kappa_2(A) = \frac{\lambda_{\max}(A)}{\lambda_{\min}(A)}.$$

If Q is an orthogonal or unitary matrix, then all its singular values equal one and $\kappa_2(Q) = 1$. Therefore, orthogonal and unitary matrices are the "best-conditioned" types of matrices. Also, multiplication of a matrix with an orthogonal or unitary matrix from the left or right does not change its condition number, cf. (14):

$$\kappa_2(UAV) = \|UAV\|_2\|V^*A^{-1}U^*\|_2 = \|A\|_2\|A^{-1}\|_2.$$

Example 1.11. Consider the system $Ax = b$ with matrix $A = \operatorname{diag}(1, \epsilon)$ and right hand side $b = (1, \epsilon)^\top$ for some $0 < \epsilon \ll 1$. The solution is given by $x = (1, 1)^\top$. The matrix A is self-adjoint and therefore $\kappa_2 = 1/\epsilon \gg 1$, which indicates a very ill-conditioned problem.

Indeed, a perturbation of the right-hand side with $\Delta b = (0, \epsilon)$ leads to a solution $\tilde{x} = A^{-1}(b + \Delta b) = (1, 2)^\top$ with relative error of about 70 percent, while the relative error of b is only ϵ. Similarly, a very small perturbation of A with $\Delta A = \operatorname{diag}(0, \epsilon)$ leads to a solution $\tilde{x} = (A + \Delta A)^{-1}b = (1, 0.5)^\top$ having a large relative error.

Note that the oversensitive behaviour of solutions is not caused by the fact that $\det A = \epsilon \approx 0$. The matrix $A = \operatorname{diag}(1, 1/\epsilon)$ with $\det A \gg 0$ would lead to comparably bad results. This argumentation also shows that, at least for our purposes, the determinant is not a useful measure of closeness to singularity.

1.5 Stability of an Algorithm

Let $\tilde{y}$ be an approximation to the solution $y = f(x)$ of the problem $x \mapsto f(x)$. The relative and absolute errors of $\tilde{y}$ are called *forward errors* of $\tilde{y}$. Forward errors of reasonable magnitude can be hard to obtain sometimes. There is, however, another way to measure the quality of $\tilde{y}$. Suppose there is a perturbation of x, for which $\tilde{y}$ is the exact solution, i.e. $\tilde{y} = f(x + \Delta x)$. Then $\|\Delta x\|$ is called the *absolute backward error.* If there is more than one such Δx, then we choose the one which is smallest in norm. As usual the *relative backward error* is the absolute one divided by $\|x\|$.

The condition number of f controls the relationship between forward and backward errors. For simplicity let $f : \mathbb{R} \to \mathbb{R}$ be a twice continuously differentiable function. As above denote by Δx the backward error of $\tilde{y}$. By Taylor's theorem there is a number ξ between x and $x + \Delta x$ such that

$$\tilde{y} - y = f(x + \Delta x) - f(x) = f'(x)\Delta x + \frac{f''(\xi)}{2}(\Delta x)^2.$$

Dropping the quadratic term in Δx and taking absolute values we obtain

$$|\tilde{y} - y| \approx \kappa_{\text{abs}}(x)|\Delta x|.$$

Division by $|y|$ gives the corresponding relation for the relative errors

$$\frac{|\tilde{y} - y|}{|y|} \approx \kappa_{\text{rel}}(x)\frac{|\Delta x|}{|x|}. \tag{1.9}$$

We have the following rule of thumb:

- If a problem is ill-conditioned, then computed solutions $\tilde{y}$ can have a large forward error even though the backward error is small.
- On the other hand, computed solutions of an extremely well-conditioned problem ($\kappa_{\text{rel}}(x) \ll 1$) can have small forward but very large backward errors.
- Finally, forward and backward errors of computed solutions to a well-conditioned problem are often comparable.

Examples 1.14, 1.15 and 1.16 below illustrate these three cases.

Example 1.12 (Cancellation)**.** Compare (1.9) to what we have shown in equation (1.4) and Example 1.8 for subtracting nearly equal numbers. In the worst case the forward error equalled the backward error $\max(|\delta x|, |\delta y|)$ times the relative condition number.

It is convenient to denote methods or algorithms for solving a problem f by $\tilde{f}$, where $\tilde{f}$ acts between the same spaces as f. If, for every x, $\tilde{f}$ produces results with a small backward error, it is called *backward stable.* Put differently, for every x, there is a small Δx such that $f(x + \Delta x) = \tilde{f}(x)$. One way to interpret this definition is this: A backward stable algorithm produces the right result for almost the right problem.

A weaker notion of stability is the following. An algorithm $\tilde{f}$ is called *stable*, if for every x there are small Δx, Δy such that $f(x + \Delta x) = \tilde{f}(x) + \Delta y$. Again this can be read in a catchy way: A stable algorithm produces almost the right result to almost the right problem. Stability is weaker than backward stability in the sense that every backward stable algorithm is also stable.

As usual, the meaning of words like "small" depends on the context. However, a reasonable bound for the backward error of a backward stable algorithm is

$$\frac{\|\Delta x\|}{\|x\|} \leq Cu, \tag{1.10}$$

where $C > 0$ is a moderate constant and u is the unit roundoff. Since in general the input to algorithms must be assumed to be erroneous, a relative input error much smaller than u can hardly be expected. With essentially the same argumentation that led to (1.9), we can obtain a bound on the forward error of a backward stable algorithm

$$\frac{\|\tilde{f}(x) - f(x)\|}{\|f(x)\|} = \frac{\|f(x + \Delta x) - f(x)\|}{\|f(x)\|} \approx \kappa_{\text{rel}}(x)\frac{\|\Delta x\|}{\|x\|} \lessapprox Cu\kappa_{\text{rel}}(x). \tag{1.11}$$

Since (1.11) measures the accuracy of the algorithm $\tilde{f}$, estimates of this type are sometimes called *accuracy estimates.*

When analysing errors of a given algorithm $x \mapsto \tilde{f}(x)$ one typically replaces every floating point operation with an exact operation times a $(1 + \delta)$ term, where $|\delta| \leq u$. This is in accordance with Assumption 1.1. First order Taylor approximations can then be used to linearize nonlinear expressions:

$$g(\delta) \approx g(0) + g'(0)\delta.$$

For $g(\delta) = 1/(1 - \delta)$ we would get

$$\frac{1}{1 - \delta} \approx 1 + \delta.$$

Example 1.13 (Backward stability of basic floating point operations)**.** Consider the problem $f : (x_1, x_2) \mapsto x_1 + x_2$ for two real numbers x_1, x_2. Our "algorithm" in this case is $\tilde{f}(x_1, x_2) = \text{fl}(x_1) \oplus \text{fl}(x_2)$. Invoking (1.2) and (1.3) we compute

$$\begin{aligned}
\tilde{f}(x_1, x_2) &= \text{fl}(x_1) \oplus \text{fl}(x_2) \\
&= x_1(1 + \delta_1) \oplus x_2(1 + \delta_2) \\
&= [x_1(1 + \delta_1) + x_2(1 + \delta_2)](1 + \delta_3) \\
&= \underbrace{x_1(1 + \delta_1 + \delta_3 + \delta_1\delta_3)}_{\tilde{x}_1} + \underbrace{x_2(1 + \delta_2 + \delta_3 + \delta_2\delta_3)}_{\tilde{x}_2} \\
&= f(\tilde{x}_1, \tilde{x}_2)
\end{aligned}$$

where all δ_i are smaller than the unit roundoff in absolute value. Thus the algorithm gives the right answer for a perturbed input with relative error

$$|\delta_i + \delta_j + \delta_i\delta_j| \leq 2u + u^2 \approx 2u,$$

which is reasonably small. Therefore floating point addition is backward stable. However, in spite of backward stability, if $x_1 \approx -x_2$, the forward errors might still be large!

Backward stability of $\ominus, \otimes, \oslash$ can be shown analogously.

Example 1.14 (An extremely well-conditioned problem)**.** Consider $f : x \mapsto 1 + x$ for $x \approx 0$. Since it does not change the outcome, we neglect rounding of the input, i.e. we assume $x \in \mathbb{F}$:

$$\tilde{f}(x) = 1 \oplus x = (1+x)(1+\delta) = 1 + x(1 + \delta + \delta/x),$$

where $|\delta| \leq u$. Notice that for very small x the fraction δ/x will be much larger than δ. Therefore the relative backward error $\delta + \delta/x$ cannot be considered small. We conclude that $\tilde{f} : x \mapsto 1 \oplus x$ is not backward stable. However, since

$$\tilde{f}(x) = 1 + x + \delta + x\delta = f(x(1+\delta)) + \delta,$$

it is stable. The forward errors are small as well

$$\frac{\tilde{f}(x) - f(x)}{f(x)} = \frac{\delta + x\delta}{1+x} = \frac{\delta}{1+x} + \delta\frac{x}{1+x} \approx \delta.$$

This situation can be explained by the fact that $f : x \mapsto 1 + x$ is an extremely well-conditioned problem for x very close to zero: $\kappa_{\text{rel}}(x) = |x|/|1+x| \ll 1$. Relative errors in x are dampened heavily and thus have almost no effect on $f(x)$. This example shows that backward stability is not always a reasonable goal.

Example 1.15 (A well-conditioned problem)**.** Consider the problem $f : x \mapsto \log(1+x)$ for $x \approx 0$ which has condition number

$$\kappa_{\text{rel}}(x) = \frac{x}{(1+x)\log(1+x)} \approx \frac{1}{1+x} \approx 1,$$

since $\log(1+x) \approx x$ for x close to zero. We assume $x \in \mathbb{F}$ and that we have an exact implementation of the logarithm. Recalling that $\log(ab) = \log a + \log b$ we get

$$\begin{aligned}
\tilde{f}(x) &= \log(1 \oplus x) \\
&= \log[(1+\delta)(1+x)] \\
&= \log(1+\delta) + \log(1+x) \\
&= \log(1+x)(1 + \delta/\log(1+x))
\end{aligned}$$

for some $|\delta| \leq u$. We have a relative forward error of $\delta/\log(1+x)$ which is very large for $x \approx 0$. For estimating the relative backward error ϵ we make the ansatz $\tilde{f}(x) = f(x(1+\epsilon))$, that is,

$$\log[(1+\delta)(1+x)]) = \log[1 + x(1+\epsilon)].$$

Cancelling the logarithms and solving for ϵ yields $\epsilon = \delta + \delta/x$ which shows a lack of backward stability.

As we have pointed out in discussing (1.9) well-conditioned problems often lead to computed solutions with similar forward and backward errors. This is also the case for this problem. The issue with $x \mapsto \log(1 \oplus x)$ is very similar to cancellation. The rounding error introduced by the addition is magnified by the logarithm, even though it is evaluated exactly. Notice that the problem $y \mapsto \log y$ is ill-conditioned for $y \approx 1$.

There is, however, a mathematically equivalent rewriting

$$\log(1+x) = 2\,\mathrm{artanh}\,(x/(x+2)),$$

which avoids numerical instabilities. There are many other examples where an algebraically equivalent rewriting leads to stable algorithms.

Example 1.16 (An ill-conditioned problem)**.** Consider $f : x \mapsto 1 - x^2$ for $|x| \approx 1$. In this case the relative condition number $\kappa_{\mathrm{rel}}(x) = 2x^2/|1-x^2|$ is very large, which indicates an ill-conditioned problem. For the sake of simplicity assume that our algorithm performs the subtraction exactly. Then

$$\tilde{f}(x) = 1 - x \otimes x = 1 - x^2(1+\delta)$$

for $\delta \leq u$ and the relative forward error explodes

$$\frac{\tilde{f}(x) - f(x)}{f(x)} = \frac{-x^2\delta}{1-x^2} \approx \frac{-\delta}{1-x^2}.$$

For estimating the backward error we again equate $\tilde{f}(x)$ with $f(x(1+\epsilon))$ leading to

$$1 - x^2(1+\delta) = 1 - (x(1+\epsilon))^2.$$

Solving for ϵ gives $\epsilon = \sqrt{1+\delta} - 1$ which has a first order approximation of $\delta/2$. Thus, the relative backward error satisfies $|\epsilon| \lesssim u/2$ from which we deduce backward stability of $\tilde{f}$.

This situation is similar to the one encountered in Example 1.4. A rounding error introduced in an otherwise unproblematic computation is elevated by subsequent subtraction even if this subtraction is performed exactly.

We conclude this chapter with general advice for designing stable algorithms. Nicholas Higham, in his standard reference *Accuracy and Stability of Numerical Algorithms*, lists the following guidelines (among others):

1. Avoid subtraction of quantities contaminated by error. Also, try to minimize the size of intermediate results relative to the size of the final result. If intermediate quantities are very large, then the final result might be affected by cancellation.

2. Look for mathematically equivalent reformulations of your problem.

3. Avoid overflow and underflow.

Chapter 2

Numerical Linear Algebra

This chapter's focus is on direct methods for systems of linear equations $Ax = b$ where $A \in \mathbb{K}^{n\times n}$ is regular. A *direct method* is an algorithm that produces a solution in a finite number of steps. In contrast, *iterative methods* in principle take infinitely many steps and have to be terminated at some point. The latter will be dealt with in the follow-up course Numerical Methods II.

Our main tool in constructing direct methods for the problem $Ax = b$ are *matrix factorizations* or *decompositions.* Their basic idea is the following:

1. Find "simple" matrices $B, C \in \mathbb{K}^{n\times n}$ such that $A = BC$.

2. Solve $By = b$ for y.

3. Solve $Cx = y$ for x.

Then, the vector x solves $Ax = b$, because $Ax = BCx = By = b$.

When using a matrix factorization approach to solve $Ax = b$, in the worst case a relative error in b will be amplified by a factor of $\kappa(B)$ to produce a new error in y. Similarly, the relative error in y can be magnified by $\kappa(C)$. In total, a relative error in b propagates to the final result x with an amplifying factor of $\kappa(B)\kappa(C)$. Thus, solving $Ax = b$ via the factorization $A = BC$ is only a good idea, if the product of condition numbers $\kappa(B)\kappa(C)$ is not much larger than $\kappa(A)$.

A general rule in numerical analysis says that one should avoid explicit computation of A^{-1}. First, it is superfluous, if only $x = A^{-1}b$ is needed. Another important reason is that computation of A^{-1} is slower and less stable than Gaussian elimination, for instance. In addition, many matrices arising in practice are very large but *sparse*, meaning that most of their entries are zero. The inverse of a sparse matrix, however, is not sparse in general and would require a large amount of space to store.

The structure and presentation of the topics covered in this chapter follows, more or less closely, the corresponding lectures in *Numerical Linear Algebra* by Trefethen and Bau.

2.1 Triangular Systems

2.1.1 Forward Substitution

Let $L = (\ell_{ij}) \in \mathbb{K}^{n\times n}$ be a regular lower triangular matrix, that is, $\ell_{ij} = 0$ for $i < j$. For a given right hand side $b \in \mathbb{K}^n$ we consider the linear system of equations $Lx = b$, which takes the form

$$\begin{array}{ccccccccc}
\ell_{11}x_1 & & & & & = & b_1 \\
\ell_{21}x_1 & + & \ell_{22}x_2 & & & = & b_2 \\
\vdots & \vdots & & \ddots & & \vdots & \vdots \\
\ell_{n1}x_1 & + & \cdots & + & \ell_{nn}x_n & = & b_n.
\end{array}$$

There is an obvious procedure for solving such a system in n steps. We start by solving the first equation for x_1 yielding $x_1 = b_1/\ell_{11}$. Next, we plug this solution into the second equation and obtain x_2. More generally, in the k-th step we plug the $k-1$ previously obtained unknowns $x_1, \dots, x_{k-1}$ into the k-th equation an solve for x_k. In fact, x_k can be expressed explicitly in terms of $x_1, \dots, x_{k-1}$ in a simple way

$$x_k = \Big(b_k - \sum_{j=1}^{k-1} \ell_{kj}x_j\Big)/\ell_{kk}. \tag{2.1}$$

The procedure of calculating x according to (2.1) is called *forward substitution:*[1]

Algorithm 2.1: Forward Substitution

```
x(1) = b(1)/L(1,1);
for j = 2:n
    x(j) = (b(j) - L(j,1:j-1)*x(1:j-1))/L(j,j);
end
```

Theorem 2.1. *If $L \in \mathbb{K}^{n\times n}$ is a regular lower triangular matrix, then for every $b \in \mathbb{K}^n$ forward substitution computes the unique solution of $Lx = b$.*

Proof. The procedure outlined above could only break down, if $\ell_{kk} = 0$ for some k. However, a triangular matrix is regular, if and only if all its diagonal entries are nonzero. Recall that for every (upper or lower) triangular matrix the determinant equals the product of its diagonal entries. Therefore, if L is invertible, forward substitution cannot fail. □

Theorem 2.2. *Forward substitution requires n^2 flops.*

[1]Assuming a basic familiarity with Matlab, in these lecture notes we will always write algorithms using Matlab syntax.

Proof. According to (2.1) the k-th step of forward substitution consists of one division, $k-1$ multiplications and $k-1$ subtractions, in total $1+2(k-1) = 2k-1$ operations. Thus, the number of flops involved in the whole procedure equals

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

In the summation above we used the fact that $\sum_{k=1}^{n} k = n(n+1)/2$. □

Is forward substitution backward stable? That is, can we assert that an algorithm implementing (2.1) produces results $\tilde{x}$ solving a system $\tilde{L}\tilde{x} = \tilde{b}$ which is an only slightly perturbed version of the exact problem $Lx = b$? The following theorem gives a positive answer.

Theorem 2.3. *Forward substitution is backward stable. That is, for all regular upper triangular matrices $L \in \mathbb{F}^{n\times n}$ and $b \in \mathbb{F}^n$ forward substitution implemented on a machine satisfying (1.2) and (1.3) will produce a result $\tilde{x} \in \mathbb{F}^n$ which solves a perturbed system $\tilde{L}\tilde{x} = b$ with componentwise errors*

$$\frac{|\ell_{ij} - \tilde{\ell}_{ij}|}{|\ell_{ij}|} \le nu + \text{h.o.t.}, \tag{2.2}$$

if $\ell_{ij} \neq 0$. If $\ell_{ij} = 0$, then also $\tilde{\ell}_{ij} = 0$.

A few remarks are in order before we prove this statement. First, the u in the estimate above is the unit roundoff as introduced in the previous chapter and h.o.t. stands for higher order terms in u. Second, note that $\tilde{x}$ is asserted to solve a system with *exact* right hand side b.

Proof. We give a detailed proof only for the case $n = 3$. This should convince the reader of the theorem's validity for arbitrary $n \in \mathbb{N}$.

The first component of $\tilde{x}$ is given by

$$\tilde{x}_1 = b_1 \oslash \ell_{11} = \frac{b_1}{\ell_{11}}(1+\epsilon),$$

where ϵ is a number which, according to Assumption 1.1, satisfies $|\epsilon| \le u$. The way the theorem is stated we want to interpret all appearing errors as perturbations in L. (b is exact!) In the case of $\tilde{x}_1$ this means that we have to move the error term $1+\epsilon$ to the denominator. If we define $\epsilon' = -\epsilon/(1+\epsilon)$, then $1+\epsilon' = 1/(1+\epsilon)$ and we get

$$\tilde{x}_1 = \frac{b_1}{\ell_{11}(1+\epsilon')}.$$

Taylor expansion of ϵ' (as a function of ϵ) around zero shows that $\epsilon' = -\epsilon$ plus higher order terms. Therefore $|\epsilon'| \le u + \text{h.o.t.}$ Note how this ϵ'-trick allows us to shift error terms from numerator and denominator and vice versa.

For the second unknown we get

$$\tilde{x}_2 = (b_2 \ominus \ell_{21} \otimes \tilde{x}_1) \oslash \ell_{22} = \frac{[b_1 - \ell_{21}\tilde{x}_1(1+\epsilon_1)]\,(1+\epsilon_2)}{\ell_{22}}(1+\epsilon_3),$$

where again $|\epsilon_i| \le u$ for $i = 1, 2, 3$. Defining ϵ_2' and ϵ_3' as above, we can interpret all three error terms as perturbations in matrix coefficients

$$\tilde{x}_2 = \frac{b_1 - \ell_{21}\tilde{x}_1(1+\epsilon_1)}{\ell_{22}(1+\epsilon_2')(1+\epsilon_3')}.$$

The third component is slightly more involved. If we subtract from left to right (recall that order matters), we have

$$\begin{aligned}\tilde{x}_3 &= ((b_3 \ominus \ell_{31} \otimes \tilde{x}_1) \ominus \ell_{32} \otimes \tilde{x}_2) \oslash \ell_{33} \\ &= \frac{[(b_3 - \ell_{31}\tilde{x}_1(1+\epsilon_4))\,(1+\epsilon_5) - \ell_{32}\tilde{x}_2(1+\epsilon_6)]\,(1+\epsilon_7)}{\ell_{33}}(1+\epsilon_8).\end{aligned}$$

The errors ϵ_4 and ϵ_6 can already be interpreted as perturbations in ℓ_{31} and ℓ_{32}, respectively. Employing the ϵ'-trick we can move the terms $1+\epsilon_7$ and $1+\epsilon_8$ into the denominator yielding

$$\tilde{x}_3 = \frac{(b_3 - \ell_{31}\tilde{x}_1(1+\epsilon_4))\,(1+\epsilon_5) - \ell_{32}\tilde{x}_2(1+\epsilon_6)}{\ell_{33}(1+\epsilon_7')(1+\epsilon_8')}.$$

It only remains to take care of ϵ_5. We divide numerator and denominator by $1+\epsilon_5$ and switch to $1+\epsilon_5'$ twice:

$$\tilde{x}_3 = \frac{(b_3 - \ell_{31}\tilde{x}_1(1+\epsilon_4)) - \ell_{32}\tilde{x}_2(1+\epsilon_5')(1+\epsilon_6)}{\ell_{33}(1+\epsilon_5')(1+\epsilon_7')(1+\epsilon_8')}.$$

If we make the following definitions

$$\begin{aligned}\tilde{\ell}_{11} &= \ell_{11}(1+\epsilon'), \\ \tilde{\ell}_{21} &= \ell_{21}(1+\epsilon_1), \\ \tilde{\ell}_{22} &= \ell_{22}(1+\epsilon_2')(1+\epsilon_3'), \\ \tilde{\ell}_{31} &= \ell_{31}(1+\epsilon_4), \\ \tilde{\ell}_{32} &= \ell_{32}(1+\epsilon_5')(1+\epsilon_6), \\ \tilde{\ell}_{33} &= \ell_{33}(1+\epsilon_5')(1+\epsilon_7')(1+\epsilon_8'),\end{aligned}$$

what we have shown so far is that $\tilde{L}\tilde{x} = b$, where $\tilde{L}$ is the lower triangular matrix with entries $\tilde{\ell}_{ij}$. In other words, the computed solution $\tilde{x}$ solves a system with exact right hand side b and perturbed matrix.

It remains to estimate the backward errors. Let $\delta\ell_{ij} = (\ell_{ij} - \tilde{\ell}_{ij})/\ell_{ij}$ denote the relative componentwise errors of the perturbed matrix. From the definitions of the $\tilde{\ell}_{ij}$ above it follows directly that

$$\begin{bmatrix} |\delta\ell_{11}| & & \\ |\delta\ell_{21}| & |\delta\ell_{22}| & \\ |\delta\ell_{31}| & |\delta\ell_{32}| & |\delta\ell_{33}| \end{bmatrix} \le u \begin{bmatrix} 1 & & \\ 1 & 2 & \\ 1 & 2 & 3 \end{bmatrix} + \text{h.o.t.}$$

Since the largest error occurs for ℓ_{33}, we have shown that

$$|\delta\ell_{ij}| \leq |\delta\ell_{33}| \leq 3u + \text{h.o.t},$$

which proves the theorem because $n = 3$. □

Remark 2.1. Note that Theorem 2.3 assumes the entries of L and b to be floating point numbers. It can be adapted to the more general case when $L \in \mathbb{K}^{n\times n}$ and $b \in \mathbb{K}^n$.

Remark 2.2. Theorem 2.3 gives us a componentwise bound on the backward error. From such a bound we can derive a *normwise* one in the following way. We can pick, for instance, the induced 1-norm and use equation (12) from the appendix to get

$$\|L - \tilde{L}\|_1 = \max_{1\leq j\leq n} \sum_{i=1}^{n} |\ell_{ij} - \tilde{\ell}_{ij}| \lesssim \max_{1\leq j\leq n} \sum_{i=1}^{n} nu|\ell_{ij}| = nu\|L\|_1.$$

Here, the relation $\lesssim$ signifies that we omitted the higher order terms from (2.2). Thus we get the normwise bound

$$\frac{\|\tilde{L} - L\|_1}{\|L\|_1} \lesssim nu. \tag{2.3}$$

Note that the same bound holds for the induced ∞-norm. Now we can use the fact that all matrix norms are equivalent to obtain a similar result for any other matrix norm: Let $\|\cdot\|$ be an arbitrary matrix norm. Then there are two positive numbers C_1 and C_2 such that

$$C_1\|A\| \leq \|A\|_1 \leq C_2\|A\|$$

for all $A \in \mathbb{K}^{n\times n}$ (compare eq. (9) in the appendix). Combining this with the first estimate above yields

$$C_1\|L - \tilde{L}\| \leq \|L - \tilde{L}\|_1 \lesssim nu\|L\|_1 \leq nuC_2\|L\|$$

and therefore

$$\frac{\|\tilde{L} - L\|}{\|L\|} \lesssim nu\frac{C_2}{C_1}.$$

2.1.2 Back Substitution

For an upper triangular matrix $U \in \mathbb{K}^{n\times n}$, $u_{ij} = 0$ for $i > j$, the associated system of linear equations takes the form

$$\begin{array}{ccccccccc}
u_{11}x_1 & + & \cdots & + & \cdots & + & u_{1n}x_n & = & b_1 \\
 & & u_{22}x_2 & + & \cdots & + & u_{2n}x_n & = & b_2 \\
 & & & & \ddots & & \vdots & \vdots & \vdots \\
 & & & & & & u_{nn}x_n & = & b_n.
\end{array}$$

There is a procedure for solving such systems which is completely analogous to forward substitution. The only difference is that you now start with the last equation and iterate through all equations from bottom to top. Accordingly, it is called *back substitution.* Theorems 2.1, 2.2 and 2.3 have obvious analogues for back substitution.

2.2 LU Factorization

Given a regular matrix A the aim of LU factorization is to find a lower triangular matrix L and an upper triangular matrix U such that $A = LU$. Having found such a decomposition, the system $Ax = b$ can be solved in two simple steps: first forward substitution for $Ly = b$ and then backward substitution for $Ux = y$.

2.2.1 Gaussian Elimination

LU factorizations can be found using a well-known algorithm from linear algebra: *Gaussian elimination.* To see how this works, let us go through an example.[2]

Example 2.1. Consider the following matrix

$$A = \begin{bmatrix} 2 & 1 & 1 & 0 \\ 4 & 3 & 3 & 1 \\ 8 & 7 & 9 & 5 \\ 6 & 7 & 9 & 8 \end{bmatrix}.$$

For a 4×4 matrix Gaussian elimination consists of three steps. In the k-th step ($k = 1, 2, 3$) one uses row operations to introduce zeros in the k-th column below the main diagonal. After the last step A has been transformed into an upper triangular matrix.

First, we subtract two times the first row from the second, four times the first row from the third, and three times the first row from the fourth. These row operations can be realized by multiplying A from the left with the matrix

$$L_1 = \begin{bmatrix} 1 & & & \\ -2 & 1 & & \\ -4 & & 1 & \\ -3 & & & 1 \end{bmatrix}.$$

Thus the resulting matrix is given by

$$L_1 A = \begin{bmatrix} 2 & 1 & 1 & 0 \\ & 1 & 1 & 1 \\ & 3 & 5 & 5 \\ & 4 & 6 & 8 \end{bmatrix}.$$

[2]This example is borrowed from *Numerical Linear Algebra* by Trefethen and Bau.

We proceed to the second column and subtract three times the second row from the third and four times the second row from the fourth. Using matrix multiplications we can write the result as

$$L_2L_1A = \begin{bmatrix} 1 & & & \\ & 1 & & \\ & -3 & 1 & \\ & -4 & & 1 \end{bmatrix} L_1A = \begin{bmatrix} 2 & 1 & 1 & 0 \\ & 1 & 1 & 1 \\ & & 2 & 2 \\ & & 2 & 4 \end{bmatrix}.$$

Now, after subtracting the third column from the fourth we obtain an upper triangular matrix U.

$$L_3L_2L_1A = \begin{bmatrix} 1 & & & \\ & 1 & & \\ & & 1 & \\ & & -1 & 1 \end{bmatrix} L_2L_1A = \begin{bmatrix} 2 & 1 & 1 & 0 \\ & 1 & 1 & 1 \\ & & 2 & 2 \\ & & & 2 \end{bmatrix} = U.$$

By inverting the matrices L_k we can write A as

$$A = (L_3L_2L_1)^{-1}U = L_1^{-1}L_2^{-1}L_3^{-1}U.$$

If the product $L_1^{-1}L_2^{-1}L_3^{-1}$ happens to be lower triangular, then we have found an LU factorization of A. The inverses are given by

$$L_1^{-1} = \begin{bmatrix} 1 & & & \\ 2 & 1 & & \\ 4 & & 1 & \\ 3 & & & 1 \end{bmatrix}, \quad L_2^{-1} = \begin{bmatrix} 1 & & & \\ & 1 & & \\ & 3 & 1 & \\ & 4 & & 1 \end{bmatrix}, \quad L_3^{-1} = \begin{bmatrix} 1 & & & \\ & 1 & & \\ & & 1 & \\ & & 1 & 1 \end{bmatrix}.$$

And their product is indeed lower triangular

$$L = L_1^{-1}L_2^{-1}L_3^{-1} = \begin{bmatrix} 1 & & & \\ 2 & 1 & & \\ 4 & 3 & 1 & \\ 3 & 4 & 1 & 1 \end{bmatrix}.$$

Summarizing, we have found the following factorization of A

$$\underbrace{\begin{bmatrix} 2 & 1 & 1 & 0 \\ 4 & 3 & 3 & 1 \\ 8 & 7 & 9 & 5 \\ 6 & 7 & 9 & 8 \end{bmatrix}}_{A} = \underbrace{\begin{bmatrix} 1 & & & \\ 2 & 1 & & \\ 4 & 3 & 1 & \\ 3 & 4 & 1 & 1 \end{bmatrix}}_{L} \underbrace{\begin{bmatrix} 2 & 1 & 1 & 0 \\ & 1 & 1 & 1 \\ & & 2 & 2 \\ & & & 2 \end{bmatrix}}_{U}.$$

General Formulas

A few observations are in order concerning the previous example. First, the inverses L_k^{-1} differ from L_k only in the signs of their subdiagonal entries. Second, the product $L_1^{-1}L_2^{-1}L_3^{-1}$ has a very simple structure as well: It collects the subdiagonal entries of all the L_k^{-1} into one identity matrix. For these two reasons the product $L_1^{-1}L_2^{-1}L_3^{-1}$ turns out to be lower triangular and an LU factorization has been obtained. That this is not a coincidence is shown below.

Let $A \in \mathbb{K}^{n\times n}$ be a regular matrix and denote by $A^{(k)}$ be the system matrix at the beginning of step k of Gaussian elimination. That is, $A^{(1)} = A$. Define the numbers

$$\ell_{jk} = \frac{a_{jk}^{(k)}}{a_{kk}^{(k)}}$$

for $j = k+1, \dots, n$. Then, the row operation matrix L_k is given by

$$L_k = \begin{bmatrix} 1 & & & & & \\ & \ddots & & & & \\ & & 1 & & & \\ & & -\ell_{k+1,k} & 1 & & \\ & & \vdots & & \ddots & \\ & & -\ell_{n,k} & & & 1 \end{bmatrix}.$$

If we define the vector

$$\ell_k = (0, \dots, 0, \ell_{k+1,k}, \dots, \ell_{n,k})^\top \in \mathbb{K}^n,$$

we can write the matrix as

$$L_k = I - \ell_k e_k^*, \tag{2.4}$$

where e_k is the k-th canonical basis vector. With these definitions at hand we can prove in full generality the two observations made in the previous paragraph.

Lemma 2.1. *The inverse of L_k is given by*

$$L_k^{-1} = I + \ell_k e_k^*.$$

Proof. We have to show that $L_k(I + \ell_k e_k^*) = (I + \ell_k e_k^*)L_k = I$. Using (2.4) we compute

$$(I - \ell_k e_k^*)(I + \ell_k e_k^*) = (I + \ell_k e_k^*)(I - \ell_k e_k^*) = I - \ell_k e_k^* \ell_k e_k^*.$$

But $e_k^* \ell_k = 0$, since the k-th entry of ℓ_k is zero, and therefore $\ell_k e_k^* \ell_k e_k^* = 0$. □

Lemma 2.2. *The product $L_k^{-1}L_{k+1}^{-1}$ equals $I + \ell_k e_k^* + \ell_{k+1}e_{k+1}^*$.*

Proof. We use the previous lemma and calculate

$$L_k^{-1}L_{k+1}^{-1} = (I + \ell_k e_k^*)(I + \ell_{k+1}e_{k+1}^*) = I + \ell_k e_k^* + \ell_{k+1}e_{k+1}^* + \ell_k e_k^* \ell_{k+1} e_{k+1}^*.$$

As before, the last term vanishes, because $e_k^* \ell_{k+1} = 0$. □

For the product of all $n-1$ matrices we get with essentially the same argumentation

$$L_1^{-1}\cdots L_{n-1}^{-1} = I + \sum_{k=1}^{n-1}(\ell_k e_k^*) = \begin{bmatrix} 1 & & & \\ \ell_{21} & 1 & & \\ \vdots & \ddots & \ddots & \\ \ell_{n1} & \cdots & \ell_{n,n-1} & 1 \end{bmatrix} = L.$$

Finally, recall that U is just the result of successive row operations, which can be written as

$$U = L_{n-1}\cdots L_1 A.$$

Algorithm

Having found general formulas for L and U we can write down an algorithm that, for a given matrix A, computes these factors. The algorithm's title will become clear in section 2.2.2.

Algorithm 2.2: Gaussian Elimination without Pivoting

```
1 L = eye(n);
2 U = A;
3 for k = 1:n-1
4     for j = k+1:n
5         L(j,k) = U(j,k)/U(k,k);
6         U(j,k:n) = U(j,k:n) - L(j,k)*U(k,k:n);
7     end
8 end
```

Note how the matrices L_k are never formed explicitly. The row operations can be performed without them and the values ℓ_{jk} are written directly into an identity matrix. (The Matlab command `eye(n)` creates an $n \times n$ identity matrix.) Actually, not even explicit forming of L and U is necessary, as their entries could be stored into A.

How many flops does Algorithm 2.2 require? There are three for-loops: two obvious ones plus a hidden one in line 6 where two vectors are subtracted. Therefore, we should expect a figure which is cubic in n. The following theorem supports this claim. Instead of giving the exact number of operations required, it only gives an asymptotic count as $n \to \infty$. This means that lower order terms are discarded, since for large n only the highest order terms are significant.

Theorem 2.4. *To leading order Gaussian elimination requires $\frac{2}{3}n^3$ operations.*

Proof. The vector `U(j,k:n)` has $n-k+1$ entries. Thus, there are $2n-2k+3$ arithmetic operations ($n-k+1$ multiplications, $n-k+1$ subtractions, and one division) at each step of the inner loop. The total number of operations required equals

$$\sum_{k=1}^{n-1}\sum_{j=k+1}^{n}(2n-2k+3) \approx \sum_{k=1}^{n-1}\sum_{j=k+1}^{n} 2(n-k).$$

Since $n-k$ does not depend on the inner summation index j and the inner sum has $n-(k+1)+1 = n-k$ summands, we have

$$\sum_{k=1}^{n-1}\sum_{j=k+1}^{n} 2(n-k) = \sum_{k=1}^{n-1} 2(n-k)^2 = 2\sum_{k=1}^{n-1} n^2 - 4\sum_{k=1}^{n-1} nk + 2\sum_{k=1}^{n-1} k^2.$$

For computing the second and third sums we use the formulas

$$\sum_{i=1}^{m} i = \frac{1}{2}m(m+1) \qquad \text{and} \qquad \sum_{i=1}^{m} i^2 = \frac{1}{6}m(m+1)(2m+1),$$

respectively, and obtain

$$2n^2(n-1) - 2n(n-1)n + \frac{1}{3}(n-1)n(2n-1).$$

Only keeping the highest order terms we end up with $2n^2 - 2n^2 + 2n^3/3$. □

Remark 2.3. Recall that forward substitution, and for reasons of symmetry also backward substitution, requires only n^2 operations. Thus, the work involved in solving $Ax = b$ via LU factorization and subsequent forward and back substitution is dominated by the factorization step.

Existence and Uniqueness

There are some basic questions about LU decomposition that we have not addressed so far: Can every regular matrix be decomposed into a product LU? If not, which matrices can and which cannot? Assuming that an LU factorization exists, is it unique?

Clearly, Algorithm 2.2 is not well-defined for every regular matrix A. If, for instance, $a_{11} = 0$, then in line 5 a division by zero will occur. Or, more generally, if for any k the diagonal entry `U(k,k)` turns out to be zero, then the algorithm will break down. However, `U(k,k)` being zero means that the $k \times k$ submatrix

$$(a_{ij})_{i,j=1}^{k} = \begin{bmatrix} a_{11} & \cdots & a_{1k} \\ \vdots & \ddots & \vdots \\ a_{k1} & \cdots & a_{kk} \end{bmatrix}$$

of the original matrix A is singular. Why? Because a sequence of row operations have produced a row consisting of all zeros. Conversely, if all submatrices $(a_{ij})_{i,j=1}^{k}$ are regular, then all `U(k,k)` will be nonzero. Hence the algorithm does not fail, but instead produces two matrices L and U as desired. Thus we have found a sufficient condition for existence of an LU factorization. In fact, the condition can be shown to be necessary as well.

Theorem 2.5. *A regular matrix $A \in \mathbb{K}^{n\times n}$ can be written $A = LU$ with $L \in \mathbb{K}^{n\times n}$ lower triangular and $U \in \mathbb{K}^{n\times n}$ upper triangular, if and only if all submatrices $(a_{ij})_{i,j=1}^{k}, k = 1, \ldots, n$, are regular. In this case there is a unique* normalized LU factorization*, that is, $\ell_{jj} = 1$ for $j = 1, \ldots, n$.*

This theorem can also be interpreted in the following way: The matrices for which Algorithm 2.2 succeeds are exactly those which possess an LU factorization.

Below we give two types of matrices which by Thm. 2.5 always admit an LU factorization.

Example 2.2. A matrix $A \in \mathbb{R}^{n\times n}$ is *positive definite*, if $(Ax)^* x > 0$ for all nonzero $x \in \mathbb{R}^n$. Positive definite matrices are always regular. In addition, every submatrix $(a_{ij})_{i,j=1}^k$ of a positive definite matrix is again positive definite and therefore regular.

Example 2.3. A matrix $A \in \mathbb{R}^{n\times n}$ is *strictly diagonally dominant*, if

$$|a_{jj}| > \sum_{k\neq j} |a_{jk}|$$

for all $j = 1, \dots, n$. Such matrices are again always regular. Every submatrix $(a_{ij})_{i,j=1}^k$ of a strictly diagonally dominant matrix is again strictly diagonally dominant, hence regular.

Instability of Gaussian Elimination

Interestingly, nonexistence of an LU factorization is related to instability of Gaussian elimination. This connection can be illustrated conveniently by means of an example.

Example 2.4. Consider the regular matrix

$$A = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}.$$

We have pointed out already that, since $a_{11} = 0$, Algorithm 2.2 cannot be used to compute an LU factorization of A. Moreover, Theorem 2.5 tells us that this is not the algorithm's fault, because A does not have an *LU* factorization.

Now replace the zero by a very small number $\epsilon > 0$. Then the resulting matrix A_ϵ does have an LU factorization. It is given by

$$\underbrace{\begin{bmatrix} \epsilon & 1 \\ 1 & 1 \end{bmatrix}}_{A_\epsilon} = \underbrace{\begin{bmatrix} 1 & 0 \\ \epsilon^{-1} & 1 \end{bmatrix}}_{L} \underbrace{\begin{bmatrix} \epsilon & 1 \\ 0 & 1-\epsilon^{-1} \end{bmatrix}}_{U}$$

For the sake of simplicity, assume that ϵ^{-1} so large that $1 - \epsilon^{-1}$ cannot be represented exactly on our machine and that $\mathrm{fl}(1-\epsilon^{-1})$ happens to be equal to $-\epsilon^{-1}$. Consequently, running Algorithm 2.2 on this machine will produce the factors

$$\tilde{L} = \begin{bmatrix} 1 & 0 \\ \epsilon^{-1} & 1 \end{bmatrix}, \quad \tilde{U} = \begin{bmatrix} \epsilon & 1 \\ 0 & -\epsilon^{-1} \end{bmatrix}.$$

While the errors in $\tilde{L}$ and $\tilde{U}$ might seem acceptable, their product

$$\tilde{L}\tilde{U} = \begin{bmatrix} \epsilon & 1 \\ 1 & 0 \end{bmatrix}$$

is entirely different from A_ϵ. In other words, the forward error is small, but the backward error is not. Therefore a solution of $\tilde{L}\tilde{U}x = b$ will in general differ considerably from a solution of $A_\epsilon x = b$. Choosing $b = (1, 0)^\top$, for instance, we get $x = (0, 1)^\top$, but the actual solution is $A_\epsilon^{-1} b \approx (-1, 1)^\top$. The underlying problem is that the factorization of A_ϵ was *not* backward stable.

Another way of looking at this lack of stability is in terms of condition numbers. At the very beginning of Chapter 2 on page 17 we argued that a matrix factorization approach $A = BC$ is only reasonable if $\kappa(A) \approx \kappa(B)\kappa(C)$. However, this rule of thumb can be severely violated by the LU factorization. In particular it is violated for A_ϵ: The matrices A_ϵ, L, U have inverses

$$A_\epsilon^{-1} = \frac{1}{\epsilon - 1}\begin{pmatrix} 1 & -1 \\ -1 & \epsilon \end{pmatrix}, \quad L^{-1} = \begin{bmatrix} 1 & 0 \\ -\epsilon^{-1} & 1 \end{bmatrix}, \quad U^{-1} = \frac{1}{\epsilon - 1}\begin{bmatrix} 1 - \epsilon^{-1} & -1 \\ 0 & \epsilon \end{bmatrix},$$

from which we easily compute their condition numbers

$$\begin{aligned} \kappa_\infty(A_\epsilon) &= \|A_\epsilon\|_\infty \|A_\epsilon^{-1}\|_\infty = 2\frac{2}{1-\epsilon} \approx 4, \\ \kappa_\infty(L) &= \|L\|_\infty \|L^{-1}\|_\infty = (1 + \epsilon^{-1})(1 + \epsilon^{-1}) \approx \epsilon^{-2}, \\ \kappa_\infty(U) &= \|U\|_\infty \|U^{-1}\|_\infty = (\epsilon^{-1} - 1)\frac{\epsilon^{-1}}{1-\epsilon} \approx \epsilon^{-2}. \end{aligned}$$

2.2.2 Pivoting

There is a surprisingly simple remedy to the situation encountered in Example 2.4. Let us try to pinpoint what exactly caused the large condition numbers. Applying Gaussian elimination to A_ϵ the first thing to do is to compute

$$\ell_{21} = \frac{a_{\epsilon,21}}{a_{\epsilon,11}} = \frac{1}{\epsilon}.$$

This number turned out to be huge and directly affected $\|L\|_\infty$. However, this can be avoided by simply interchanging the rows of A_ϵ first. Recall that, when solving a linear system $Ax = b$, interchanging rows is fine as long as we interchange the corresponding entries of b as well. So, if we apply the elimination step to

$$PA_\epsilon = \begin{bmatrix} & 1 \\ 1 & \end{bmatrix}\begin{bmatrix} \epsilon & 1 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ \epsilon & 1 \end{bmatrix},$$

instead of A_ϵ, then $\ell_{21} = \epsilon$ and $\|L\|_\infty \approx 1$. This is the basic idea of pivoting.

In step k of Gaussian elimination, we use the (k, k) element of the system matrix $a_{kk}^{(k)}$ to introduce zeros in the k-th column

$$\begin{bmatrix} * & * & * & * \\ & a_{kk}^{(k)} & * & * \\ & * & * & * \\ & * & * & * \end{bmatrix} \longrightarrow \begin{bmatrix} * & * & * & * \\ & a_{kk}^{(k)} & * & * \\ & 0 & * & * \\ & 0 & * & * \end{bmatrix}.$$

This element is usually called the *pivot*. We have seen above that this elimination is impossible, if the pivot is zero, or problematic, if it is much smaller in absolute value than one of the entries below. However, instead of $a_{kk}^{(k)}$ we could equally use any element $a_{ik}^{(k)}, i \geq k$, to introduce zeros in column k. Ideally we would like to choose that element $a_{ik}^{(k)}$ which is largest in absolute value. In addition, in order to keep the triangular structure we can interchange rows so that $a_{ik}^{(k)}$ moves into the main diagonal. For example,

$$\begin{bmatrix} * & * & * & * \\ & * & * & * \\ & a_{ik}^{(k)} & * & * \\ & * & * & * \end{bmatrix} \longrightarrow \begin{bmatrix} * & * & * & * \\ & a_{ik}^{(k)} & * & * \\ & * & * & * \\ & * & * & * \end{bmatrix} \longrightarrow \begin{bmatrix} * & * & * & * \\ & a_{ik}^{(k)} & * & * \\ & 0 & * & * \\ & 0 & * & * \end{bmatrix}.$$

This strategy is called *partial pivoting*. Using matrix products we can write the result of $n - 1$ steps of Gaussian elimination with partial pivoting in the following way.

$$L_{n-1}P_{n-1} \cdots L_1 P_1 A = U,$$

where the P_j are permutation matrices.

More generally, we could look not only at the k-th column for the largest element but at all entries $a_{ij}^{(k)}$, where both $i \geq k$ and $j \geq k$, and then interchange rows and columns in order to move it into the (k, k) position. This more expensive strategy is called *complete pivoting*. In practice, however, partial pivoting is often sufficient.

Example 2.5. We consider the same 4×4 matrix as before

$$A = \begin{bmatrix} 2 & 1 & 1 & 0 \\ 4 & 3 & 3 & 1 \\ 8 & 7 & 9 & 5 \\ 6 & 7 & 9 & 8 \end{bmatrix}$$

and apply Gaussian elimination with partial pivoting. The largest element in the first column is a_{31}. Therefore we interchange rows three and one. This is achieved by multiplying A from the left with an identity matrix that has rows one and three interchanged

$$P_1 A = \begin{bmatrix} & & 1 & \\ & 1 & & \\ 1 & & & \\ & & & 1 \end{bmatrix} A = \begin{bmatrix} 8 & 7 & 9 & 5 \\ 4 & 3 & 3 & 1 \\ 2 & 1 & 1 & 0 \\ 6 & 7 & 9 & 8 \end{bmatrix}.$$

Next we eliminate

$$L_1P_1A = \begin{bmatrix} 1 & & & \\ -\frac{1}{2} & 1 & & \\ -\frac{1}{4} & & 1 & \\ -\frac{3}{4} & & & 1 \end{bmatrix} P_1A = \begin{bmatrix} 8 & 7 & 9 & 5 \\ & -\frac{1}{2} & -\frac{3}{2} & -\frac{3}{2} \\ & -\frac{3}{4} & -\frac{5}{4} & -\frac{5}{4} \\ & \frac{7}{4} & \frac{9}{4} & \frac{17}{4} \end{bmatrix}.$$

Since $\frac{7}{4}$ is greater than both $-\frac{3}{4}$ and $-\frac{1}{2}$ in absolute value, we interchange rows two and four before eliminating

$$\underbrace{\begin{bmatrix} 1 & & & \\ & 1 & & \\ & \frac{3}{7} & 1 & \\ & \frac{2}{7} & & 1 \end{bmatrix}}_{L_2} \underbrace{\begin{bmatrix} 1 & & & \\ & & & 1 \\ & & 1 & \\ & 1 & & \end{bmatrix}}_{P_2} L_1P_1A = \begin{bmatrix} 8 & 7 & 9 & 5 \\ & \frac{7}{4} & \frac{9}{4} & \frac{17}{4} \\ & & -\frac{2}{7} & \frac{4}{7} \\ & & -\frac{6}{7} & -\frac{2}{7} \end{bmatrix}.$$

Finally, to obtain an upper triangular matrix we interchange the third and fourth rows and then introduce a zero in position $(4, 3)$.

$$\underbrace{\begin{bmatrix} 1 & & & \\ & 1 & & \\ & & 1 & \\ & & -\frac{1}{3} & 1 \end{bmatrix}}_{L_3} \underbrace{\begin{bmatrix} 1 & & & \\ & 1 & & \\ & & & 1 \\ & & 1 & \end{bmatrix}}_{P_3} L_2P_2L_1P_1A = \begin{bmatrix} 8 & 7 & 9 & 5 \\ & \frac{7}{4} & \frac{9}{4} & \frac{17}{4} \\ & & -\frac{6}{7} & -\frac{2}{7} \\ & & & \frac{2}{3} \end{bmatrix} = U.$$

In the end we have obtained $L_3P_3L_2P_2L_1P_1A = U$. Now, the natural question is whether this gives us an LU factorization of A. In short the answer is no, because the product $L_3P_3L_2P_2L_1P_1$ in general is far from lower triangular, and so is its inverse. However, it is easy to check that

$$L_3P_3L_2P_2L_1P_1 = L_3'L_2'L_1'P_3P_2P_1,$$

where L_j' is equal to L_j up to a permutation of subdiagonal entries. More precisely, the L_j' are given by

$$L_3' = L_3, \quad L_2' = P_3L_2P_3^{-1}, \quad L_1' = P_3P_2L_1P_2^{-1}P_3^{-1}.$$

For example,

$$L_2' = P_3L_2P_3^{-1} = \begin{bmatrix} 1 & & & \\ & 1 & & \\ & & & 1 \\ & & 1 & \end{bmatrix} \begin{bmatrix} 1 & & & \\ & 1 & & \\ & \frac{3}{7} & 1 & \\ & \frac{2}{7} & & 1 \end{bmatrix} \begin{bmatrix} 1 & & & \\ & 1 & & \\ & & & 1 \\ & & 1 & \end{bmatrix}$$
$$= \begin{bmatrix} 1 & & & \\ & 1 & & \\ & \frac{2}{7} & 1 & \\ & \frac{3}{7} & & 1 \end{bmatrix}.$$

Note that the product of lower triangular matrices $L_3'L_2'L_1'$ is again lower triangular and so is their inverse. Similarly the product of permutation matrices is again a permutation matrix. We therefore define $L \coloneqq (L_3'L_2'L_1')^{-1}$ and $P \coloneqq P_3P_2P_1$ and return to the factorization of A

$$U = L_3P_3L_2P_2L_1P_1A = L_3'L_2'L_1'P_3P_2P_1A = L^{-1}PA,$$

or equivalently

$$\underbrace{\begin{bmatrix} & & 1 & \\ & & & 1 \\ & 1 & & \\ 1 & & & \end{bmatrix}}_{P}\underbrace{\begin{bmatrix} 2 & 1 & 1 & 0 \\ 4 & 3 & 3 & 1 \\ 8 & 7 & 9 & 5 \\ 6 & 7 & 9 & 8 \end{bmatrix}}_{A} = \underbrace{\begin{bmatrix} 1 & & & \\ \frac{3}{4} & 1 & & \\ \frac{1}{2} & -\frac{2}{7} & 1 & \\ \frac{1}{4} & -\frac{3}{7} & \frac{1}{3} & 1 \end{bmatrix}}_{L}\underbrace{\begin{bmatrix} 8 & 7 & 9 & 5 \\ & \frac{7}{4} & \frac{9}{4} & \frac{17}{4} \\ & & -\frac{6}{7} & -\frac{2}{7} \\ & & & \frac{2}{3} \end{bmatrix}}_{U}$$

Thus, we have found an LU factorization not of A but of PA, which is a row-permuted version of A. Besides, note how all subdiagonal entries of L are ≤ 1 in absolute value because of the pivoting strategy.

General Formulas and Algorithm

Let A be a regular $n \times n$ matrix. Then, after $n-1$ steps of Gaussian elimination with partial pivoting we can write the resulting upper triangular matrix U in the following way

$$U = L_{n-1}P_{n-1}\cdots L_1P_1A.$$

If we define for every $k = 1, \ldots, n-1$ the modified row operation matrices as

$$L_k' = P_{n-1}\cdots P_{k+1}L_kP_{k+1}^{-1}\cdots P_{n-1}^{-1},$$

we can rewrite the above expression for U

$$U = \underbrace{L_{n-1}'\cdots L_1'}_{=L^{-1}}\underbrace{P_{n-1}\cdots P_1}_{=P}A.$$

Since the L_k' have the same structure as the L_k, they are just as easily multiplied and inverted.

We are now in a position to write down the algorithm. For the sake of readability we use pseudocode instead of correct Matlab syntax.

Algorithm 2.3: Gaussian Elimination with Partial Pivoting

```
L=I;
U=A;
P=I;
for k = 1:n-1
    select i >= k to maximize |U(i,k)|
    interchange U(k,k:n) and U(i,k:n)
    interchange L(k,1:k-1) and L(i,1:k-1)
    interchange P(k,:) and P(i,:)
    for j = k+1:n
        L(j,k) = U(j,k)/U(k,k);
        U(j,k:n) = U(j,k:n) - L(j,k)*U(k,k:n);
    end
end
```

A few remarks are in order regarding actual implementation of this algorithm. First, there is no need to represent P as a matrix. The same can be achieved more efficiently using a permutation vector, that is, a vector which is initialized as $p = (1, 2, \dots, n)$ and whose entries are subsequently interchanged according to P. In addition, as was the case for Algorithm 2.2, the matrices L, U are actually superfluous, since their entries can be stored directly into A.

To leading order Algorithm 2.3 requires the same amount of floating point operations as Algorithm 2.2, that is, $2n^3/3$. In order to determine the pivot in step k, one has to look at $n-k$ entries. Thus, in total the partial pivoting strategy leads to additional cost of $\sum_{k=1}^{n-1}(n-k)$ operations, which is quadratic in n and therefore negligible compared to the already cubic cost. With complete pivoting, however, in every step $(n-k)^2$ entries must be examined, thus leading to an additional cost which is cubic in n and not negligible anymore.

Finally, concerning existence of a factorization $PA = LU$ we have the following result.

Theorem 2.6. *For every regular matrix $A \in \mathbb{K}^{n\times n}$ Gaussian elimination with partial pivoting produces a normalized LU factorization of PA, where $P \in \mathbb{K}^{n\times n}$ is a permutation matrix.*

Stability

Stability analysis of Gaussian elimination is a complicated matter, which is why we will not go into details here. The gist is, however, that Gaussian elimination with partial pivoting is unstable in theory but perfectly stable in practice. Put differently, there are certain matrices for which it behaves in an unstable way, but these matrices are extremely rare in practice.

Theorem 2.7. *Let $A \in \mathbb{K}^{n\times n}$ be a regular matrix. Suppose A has normalized LU factorization $A = LU$, and that $\tilde{L}, \tilde{U}$ are the factors computed by Algorithm 2.2 on a machine satisfying* (1.2) *and* (1.3). *Then there is a matrix $\Delta A \in \mathbb{K}^{n\times n}$ such that*

$$\tilde{L}\tilde{U} = A + \Delta A \quad \textit{and} \quad \|\Delta A\| \le C\|L\|\|U\|u$$

for a moderate constant $C > 0$.

This theorem is to be interpreted in the following way. If $\|L\|\|U\| \approx \|A\|$, then $\|\Delta A\|/\|A\| \lesssim Cu$. In this case Gaussian elimination without pivoting is backward stable. Otherwise it is not. However, in Example 2.4 we have seen that $\|L\|$ and $\|U\|$ can become arbitrarily large. Therefore, Algorithm 2.2 is not backward stable.

For Gaussian elimination with partial pivoting there is a similar theorem stating that $\tilde{L}\tilde{U} = \tilde{P}A + \Delta A$, where $\tilde{P}$ is the permutation matrix produced by the algorithm. In addition, we know that $\|L\|$ is small because of the pivoting strategy. In this case the estimate on $\|\Delta A\|$ implies that

$$\frac{\|\Delta A\|}{\|A\|} \leq \frac{C\|L\|\|U\|u}{\|A\|} \approx Cu\frac{\|U\|}{\|A\|}.$$

Thus, for Gaussian elimination with partial pivoting to be backward stable the ratio $\|U\|/\|A\|$ must remain moderate. For virtually all matrices A encountered in practice this is the case and therefore Algorithm 2.3 can be called backward stable in practice. Yet there are matrices where the ratio does become very large (see below). This is the reason why Gaussian elimination with partial pivoting is not backward stable in theory.

Example 2.6. Consider the following matrix

$$A = \begin{bmatrix} 1 & & & 1 \\ -1 & 1 & & 1 \\ -1 & -1 & 1 & 1 \\ -1 & -1 & -1 & 1 \end{bmatrix}.$$

Gaussian elimination with (or without) partial pivoting leads to the factors

$$L = \begin{bmatrix} 1 & & & \\ -1 & 1 & & \\ -1 & -1 & 1 & \\ -1 & -1 & -1 & 1 \end{bmatrix}, \qquad U = \begin{bmatrix} 1 & & & 1 \\ & 1 & & 2 \\ & & 1 & 4 \\ & & & 8 \end{bmatrix}.$$

Partial pivoting leads to the same factors as no pivoting at all, since no row interchanges will occur for this particular A. Choosing the ∞-norm, for instance, the ratio mentioned in the previous paragraph equals

$$\frac{\|U\|_\infty}{\|A\|_\infty} = \frac{8}{4} = 2.$$

In general, however, for an $n \times n$ matrix with the same pattern as A the largest element of U will equal 2^{n-1}. Therefore the ratio equals $2^{n-1}/n$ which for growing n quickly becomes a huge number.

2.3 Cholesky Factorization

Self-adjoint positive definite matrices arise naturally in many physical systems. They have factorizations $A = R^*R$, where R is upper triangular. In contrast to LU factorization only one matrix must be constructed. Accordingly, Cholesky factorizations can be computed twice as fast. The standard algorithm for doing so can be viewed as a symmetric variant of Gaussian elimination that operates on rows and columns at the same time.

2.3.1 SPD Matrices

A matrix $A \in \mathbb{K}^{n\times n}$ is an *SPD matrix*, if it is self-adjoint and positive definite. That is, $A^* = A$ and $x^*Ax > 0$ for all nonzero $x \in \mathbb{K}^n$. Note that the diagonal entries of such a matrix are positive real numbers, because $a_{jj} = e_j^* A e_j > 0$.

Lemma 2.3. *Let $A \in \mathbb{K}^{n\times n}$ be an SPD matrix and let $B \in \mathbb{K}^{n\times m}$, $n \geq m$, have full rank. Then B^*AB is an SPD matrix.*

Proof. The matrix B^*AB is self-adjoint, because

$$(B^*AB)^* = B^*A^*B^{**} = B^*AB.$$

Let $x \in \mathbb{K}^m$ be a nonzero vector. Then Bx is again nonzero, since B is of full rank. Therefore

$$x^*B^*ABx = (Bx)^*ABx > 0.$$

□

An important consequence of Lemma 2.3 is that every principal submatrix of an SPD matrix is again SPD. An $m \times m$ submatrix of A is called *principal submatrix*, if it is obtained from A by deleting any $n-m$ columns and the same $n-m$ rows. Principal submatrices can be written as $(a_{ij})_{i,j\in J}$ where J is any subset of $\{1, \dots, n\}$. In matrix notation principal matrices can be written as follows. Denote by I_J the matrix that results from the n-dimensional identity matrix by deleting all those columns whose indices are not in the set J. Then $(a_{ij})_{i,j\in J} = I_J^* A I_J$. By the previous lemma this matrix is SPD, if A is.

2.3.2 Symmetric Gaussian Elimination

Let $A \in \mathbb{K}^{n\times n}$ be an SPD matrix. Using block notation we can write it as

$$\begin{bmatrix} a & w^* \\ w & K \end{bmatrix},$$

where $a > 0$, $w \in \mathbb{K}^{n-1}$ and $K \in \mathbb{K}^{n-1\times n-1}$. One step of Gaussian elimination gives

$$L_1 A = \begin{bmatrix} 1 & 0 \\ -w/a & I \end{bmatrix} \begin{bmatrix} a & w^* \\ w & K \end{bmatrix} = \begin{bmatrix} a & w^* \\ 0 & K - ww^*/a \end{bmatrix}.$$

Next, instead of proceeding to the second column (as we would, were we computing an LU factorization), we apply a symmetric elimination step on the columns. That is, using column operations we eliminate the w^* in the first row of L_1A. Due to the self-adjointness of A, the matrix that performs this step is just L_1^*, which has to be multiplied from the right. Therefore, we get

$$L_1 A L_1^* = \begin{bmatrix} a & w^* \\ 0 & K - ww^*/a \end{bmatrix} \begin{bmatrix} 1 & -w^*/a \\ 0 & I \end{bmatrix} = \begin{bmatrix} a & 0 \\ 0 & K - ww^*/a \end{bmatrix}.$$

Inverting the two matrices L_1 and L_1^* we have the following factorization of A

$$A = \underbrace{\begin{bmatrix} 1 & 0 \\ w/a & I \end{bmatrix}}_{L_1^{-1}} \underbrace{\begin{bmatrix} a & 0 \\ 0 & K - ww^*/a \end{bmatrix}}_{\tilde{A}_1} \underbrace{\begin{bmatrix} 1 & w^*/a \\ 0 & I \end{bmatrix}}_{L_1^{-*}}.$$

This is almost one step of Cholesky factorization. For reasons that will become clear soon, we would like the $(1,1)$ entry of $\tilde{A}_1$ to equal 1. Note that $\tilde{A}_1$ can be decomposed in the following way

$$\tilde{A}_1 = \begin{bmatrix} \sqrt{a} & 0 \\ 0 & I \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & K - ww^*/a \end{bmatrix} \begin{bmatrix} \sqrt{a} & 0 \\ 0 & I \end{bmatrix}.$$

Recall that $a = a_{11} > 0$ and therefore $\sqrt{a}$ is again a real number, in particular a positive one. Plugging this product back into the previous factorization of A we finally arrive at

$$A = \underbrace{\begin{bmatrix} \sqrt{a} & 0 \\ w/\sqrt{a} & I \end{bmatrix}}_{R_1^*} \underbrace{\begin{bmatrix} 1 & 0 \\ 0 & K - ww^*/a \end{bmatrix}}_{A_1} \underbrace{\begin{bmatrix} \sqrt{a} & w^*/\sqrt{a} \\ 0 & I \end{bmatrix}}_{R_1}. \tag{2.5}$$

If the $(1,1)$ entry of $K - ww^*/a$ is a positive real number, we can eliminate the second row and column of A_1 by applying the above procedure to $K - ww^*/a$. This would give us a factorization $A_1 = R_2^* A_2 R_2$ and in combination with (2.5) we would get $A = R_1^* R_2^* A_2 R_2 R_1$. Continuing this process, after n steps we would eventually end up with

$$A = \underbrace{R_1^* \cdots R_n^*}_{R^*} I \underbrace{R_n \cdots R_1}_{R} = R^* R,$$

where $r_{jj} > 0$ for $j = 1, \ldots, n$. Such a factorization is called *Cholesky factorization.*

So, is the $(1,1)$ element of $K - ww^*/a$ positive? The answer is yes, because it is a diagonal entry of $A_1 = R_1^{-*} A R_1^{-1}$, which by Lemma 2.3 is an SPD matrix. With the same is reasoning it follows that for every k the (k,k) entry of A_{k-1} must be a positive number. The process we have described so far cannot fail and in fact produces a unique matrix R for every given SPD matrix A. Thus, we have the following theorem.

Theorem 2.8. *Every SPD matrix $A \in \mathbb{K}^{n \times n}$ has a unique Cholesky factorization. That is, there is exactly one upper triangular matrix $R \in \mathbb{K}^{n \times n}$ such that*

$$A = R^* R$$

and $r_{jj} > 0$ for $j = 1, \ldots, n$.

Example 2.7. Consider the SPD matrix

$$A = \begin{bmatrix} 4 & 2 & 0 \\ 2 & 3 & 3 \\ 0 & 3 & 9 \end{bmatrix}.$$

First, we subtract $\frac{1}{2}$ times the first row from the second and obtain

$$L_1 A = \begin{bmatrix} 1 & & \\ -\frac{1}{2} & 1 & \\ & & 1 \end{bmatrix} \begin{bmatrix} 4 & 2 & 0 \\ 2 & 3 & 3 \\ 0 & 3 & 9 \end{bmatrix} = \begin{bmatrix} 4 & 2 & 0 \\ 0 & 2 & 3 \\ 0 & 3 & 9 \end{bmatrix}.$$

Next we apply the same operations to the columns of $L_1 A$, that is, we subtract $\frac{1}{2}$ times the first column from the second

$$L_1 A L_1^* = \begin{bmatrix} 4 & 2 & 0 \\ 0 & 2 & 3 \\ 0 & 3 & 9 \end{bmatrix} \begin{bmatrix} 1 & -\frac{1}{2} & \\ & 1 & \\ & & 1 \end{bmatrix} = \begin{bmatrix} 4 & & \\ & 2 & 3 \\ & 3 & 9 \end{bmatrix}.$$

Now we invert the matrices L_1 and L_1^* and write A as a product of three matrices

$$\begin{aligned} A &= \begin{bmatrix} 1 & & \\ \frac{1}{2} & 1 & \\ & & 1 \end{bmatrix} \begin{bmatrix} 4 & & \\ & 2 & 3 \\ & 3 & 9 \end{bmatrix} \begin{bmatrix} 1 & \frac{1}{2} & \\ & 1 & \\ & & 1 \end{bmatrix} \\ &= \underset{R_1^*}{\begin{bmatrix} 2 & & \\ 1 & 1 & \\ & & 1 \end{bmatrix}} \underset{A_1}{\begin{bmatrix} 1 & & \\ & 2 & 3 \\ & 3 & 9 \end{bmatrix}} \underset{R_1}{\begin{bmatrix} 2 & 1 & \\ & 1 & \\ & & 1 \end{bmatrix}}. \end{aligned}$$

We continue with the matrix A_1 and subtract $3/2$ times the second row from the third row, as well as $3/2$ times the second column from the column. Using matrix notation this yields

$$L_2 A_1 L_2^* = \begin{bmatrix} 1 & & \\ & 1 & \\ & -\frac{3}{2} & 1 \end{bmatrix} \begin{bmatrix} 1 & & \\ & 2 & 3 \\ & 3 & 9 \end{bmatrix} \begin{bmatrix} 1 & & \\ & 1 & -\frac{3}{2} \\ & & 1 \end{bmatrix} = \begin{bmatrix} 1 & & \\ & 2 & \\ & 0 & 9 \end{bmatrix}.$$

But now A_1 can be expressed as

$$\begin{aligned} A_1 &= \begin{bmatrix} 1 & & \\ & 1 & \\ & \frac{3}{2} & 1 \end{bmatrix} \begin{bmatrix} 1 & & \\ & 2 & \\ & & 9 \end{bmatrix} \begin{bmatrix} 1 & & \\ & 1 & \frac{3}{2} \\ & & 1 \end{bmatrix} \\ &= \underset{R_2^*}{\begin{bmatrix} 1 & & \\ & \sqrt{2} & \\ & \frac{3}{\sqrt{2}} & 1 \end{bmatrix}} \underset{A_2}{\begin{bmatrix} 1 & & \\ & 1 & \\ & & \frac{9}{2} \end{bmatrix}} \underset{R_2}{\begin{bmatrix} 1 & & \\ & \sqrt{2} & \frac{3}{\sqrt{2}} \\ & & 1 \end{bmatrix}}. \end{aligned}$$

In order to finish the Cholesky factorization we only have to decompose

$$A_2 = \underset{R_3^*}{\begin{bmatrix} 1 & & \\ & 1 & \\ & & \frac{3}{\sqrt{2}} \end{bmatrix}} \underset{R_3}{\begin{bmatrix} 1 & & \\ & 1 & \\ & & \frac{3}{\sqrt{2}} \end{bmatrix}}.$$

The factor R is now the product of the matrices R_1, R_2 and R_3

$$R = \begin{bmatrix} 1 & & \\ & 1 & \\ & & \frac{3}{\sqrt{2}} \end{bmatrix} \begin{bmatrix} 1 & & \\ & \sqrt{2} & \frac{3}{\sqrt{2}} \\ & & 1 \end{bmatrix} \begin{bmatrix} 2 & 1 & \\ & 1 & \\ & & 1 \end{bmatrix} = \begin{bmatrix} 2 & 1 & \\ & \sqrt{2} & \frac{3}{\sqrt{2}} \\ & & \frac{3}{\sqrt{2}} \end{bmatrix}.$$

2.3.3 Algorithm

Let $A \in \mathbb{K}^{n\times n}$ be an SPD matrix. After k steps of symmetric Gaussian elimination, we have reduced A to

$$A_k = \begin{bmatrix} I & \\ & S \end{bmatrix},$$

where I is a $k \times k$ identity matrix and S is an $(n-k)\times(n-k)$ SPD matrix which can be written as

$$S = \begin{bmatrix} s & w^* \\ w & \tilde{S} \end{bmatrix},$$

with $s > 0$. The $(k+1)$-th step now consists of two parts. One is computation of R_{k+1} and the other one is updating A_k into A_{k+1}. The general formulas are

$$R_{k+1} = \begin{bmatrix} I & & \\ & \sqrt{s} & w^*/\sqrt{s} \\ & & I \end{bmatrix},$$

$$A_{k+1} = \begin{bmatrix} I & \\ & \tilde{S} - ww^*/\sqrt{s} \end{bmatrix}.$$

Note how the one nontrivial row of R_{k+1} is just the first row of S divided by $\sqrt{s}$.

We can now write down the algorithm in Matlab syntax. It takes as input an $n \times n$ SPD matrix A and returns the upper triangular matrix R.

Algorithm 2.4: Cholesky Factorization

```
R = zeros(n,n);
for k=1:n
    for j from k+1:n
        A(j,j:n) = A(j,j:n) - A(k,j)/A(k,k) * A(k,j:n);
    end
    R(k:k:n) = A(k,k:n)/sqrt(A(k,k));
end
```

As usual, the new matrix R need not be formed explicitly. Its entries can be written directly into the input array A.

Theorem 2.9. *To leading order Algorithm 2.4 requires $n^3/3$ operations.*

Proof. The work for the whole algorithm is dominated by the inner for-loop, which requires

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

In total we get

$$\begin{aligned}
2\sum_{k=1}^{n} \sum_{j=k+1}^{n} (n-j) &= 2\sum_{k=1}^{n} \sum_{j=1}^{n-k} (n-(j+k)) \\
&= 2\sum_{k=1}^{n} \left[(n-k)(n-k+1) - \frac{1}{2}(n-k)(n-k+1) \right] \\
&\approx \sum_{k=1}^{n} (n^2 - 2nk + k^2) \\
&= n^3 - 2n\frac{n(n+1)}{2} + \frac{1}{6}n(n+1)(2n+1) \\
&\approx \frac{n^3}{3}.
\end{aligned}$$

□

When computing Cholesky-factorizations there is no need for pivoting, because Algorithm 2.4 is always stable. The problems encountered for the LU factorization, where the factors U might have very large norms compared to A, cannot occur here. In fact, one can show that

$$\|R\|_2 = \|R^*\|_2 = \|A\|_2^{\frac{1}{2}}.$$

Theorem 2.10. *Algorithm 2.4 is backward stable. More precisely, for every SPD matrix $A \in \mathbb{K}^{n\times n}$ Algorithm 2.4 implemented on a machine satisfying* (1.2) *and* (1.3) *produces an upper triangular matrix $\tilde{R}$ such that*

$$\tilde{R}^*\tilde{R} = A + \Delta A,$$

where the error matrix $\Delta A \in \mathbb{K}^{n\times n}$ satisfies

$$\frac{\|\Delta A\|}{\|A\|} \le Cu, \quad \textit{for some } C > 0.$$

2.4 QR Factorization

In this section we do not restrict our attention to square matrices, but instead consider $A \in \mathbb{K}^{m\times n}$, $m \geq n$. For such matrices there are two different types of QR factorizations, a reduced and a full one. The *reduced QR factorization*

$$A = \hat{Q}\hat{R}$$

expresses A as a product of an $m \times n$ matrix $\hat{Q}$ with orthonormal columns and an upper triangular matrix $\hat{R} \in \mathbb{K}^{n\times n}$. For the *full QR factorization*

$$A = QR$$

the matrix $Q \in \mathbb{K}^{m\times m}$ is orthogonal/unitary and $R \in \mathbb{K}^{m\times n}$ is generalized upper triangular, that is, it is not necessarily square but still $r_{ij} = 0$ for $i > j$. The relation between these two factorization can be visualized as follows

$$A = \underbrace{\begin{bmatrix}\hat{Q} & \cdots\end{bmatrix}}_{Q} \underbrace{\begin{bmatrix}\hat{R} \\ 0\end{bmatrix}}_{R}.$$

The matrix Q contains an additional $m - n$ orthonormal vectors so that its columns form an orthonormal basis of $\mathbb{K}^m$. However, these additional columns do not contribute to the product QR since they are multiplied with the block of zeros at the bottom of R.

2.4.1 Gram-Schmidt

The QR factorization can be understood in terms of a well-known algorithm from linear algebra: the Gram-Schmidt process. See the section *Orthogonality* in the Appendix.

First, let us write out a reduced QR factorization using column vectors

$$\underbrace{\left[\begin{array}{c|c|c} a_1 & \cdots & a_n \end{array}\right]}_{A} = \underbrace{\left[\begin{array}{c|c|c} q_1 & \cdots & q_n \end{array}\right]}_{\hat{Q}} \underbrace{\begin{bmatrix} r_{11} & \cdots & r_{1n} \\ & \ddots & \vdots \\ & & r_{nn} \end{bmatrix}}_{\hat{R}}.$$

Writing out separate equations for every column of A yields the following system

$$\begin{aligned} a_1 &= r_{11} q_1 \\ a_2 &= r_{12} q_1 + r_{22} q_2 \\ &\vdots \\ a_n &= r_{1n} q_1 + r_{2n} q_2 + \cdots + r_{nn} q_n. \end{aligned} \tag{2.6}$$

Since every column of A is expressed as a linear combination of certain columns of $\hat{Q}$, we have that

$$a_k \in \operatorname{span}\{q_1, \ldots, q_k\}, \quad k = 1, \ldots, n.$$

Assume for the moment that A has full rank. This implies that also $\hat{R}$ has full rank and therefore all diagonal entries r_{kk} are nonzero. In particular the top left $k \times k$ submatrix of $\hat{R}$ is invertible so that we can express q_k as linear combinations of the first k columns of A. Thus,

$$q_k \in \operatorname{span}\{a_1, \ldots, a_k\}, \quad k = 1, \ldots, n,$$

and therefore the columns of A and $\hat{Q}$ span the same spaces

$$\operatorname{span}\{a_1, \dots, a_k\} = \operatorname{span}\{q_1, \dots, q_k\}, \quad k = 1, \dots, n. \tag{2.7}$$

We can now reformulate the problem of finding a reduced QR factorization for a given full rank matrix $A \in \mathbb{K}^{m\times n}$: Given n linearly dependent vectors $a_1, \dots, a_n \in \mathbb{K}^m$, find n orthonormal vectors $q_1, \dots, q_n \in \mathbb{K}^m$ satisfying (2.7). But this is exactly what the Gram-Schmidt process does! Thus, we already know an algorithm to compute the q_k.

What about the r_{ij}? Let us first express the q_k in (2.6)

$$\begin{aligned} q_1 &= \frac{a_1}{r_{11}} \\ q_2 &= \frac{1}{r_{22}}\left(a_2 - r_{12}q_1\right) \\ &\vdots \\ q_n &= \frac{1}{r_{nn}}\left(a_n - r_{1n}q_1 - \dots - r_{n-1,n}q_{n-1}\right). \end{aligned}$$

Now we only have to compare with the Gram-Schmidt process, see eq. (8) in the Appendix, to find that

$$r_{ij} = \begin{cases} q_i^* a_j, & i < j \\ \left\|a_j - r_{1j}q_1 - \dots - r_{j-1,j}q_{j-1}\right\|_2, & i = j. \end{cases}$$

Having found general formulas for both q_k and r_{ij} we can write down an algorithm which for a given full rank matrix $A \in \mathbb{R}^{m\times n}$ computes a reduced QR factorization.

Algorithm 2.5: Classical Gram-Schmidt

```
R = zeros(n);
Q = zeros(m,n);
V = A;
for j=1:n
    for i = 1:j-1
            R(i,j) = Q(:,i)' * A(:,j);
            V(:,j) = V(:,j) - R(i,j) * Q(:,i);
    end
    R(j,j) = norm(V(:,j));
    Q(:,j) = V(:,j)/R(j,j);
end
```

Theorem 2.11. *Every $A \in \mathbb{K}^{m\times n}$, $m \geq n$, has both a reduced and a full QR factorization.*

Proof. If A has full rank, then Algorithm 2.5 cannot break down, because $r_{jj} \neq 0$, thus producing a reduced QR factorization. Extend $\hat{Q}$ to an orthonormal basis and append $m-n$ zero rows to $\hat{R}$ to obtain a full QR factorization.

If A does not have full rank, then $r_{jj} = 0$ for some j. In this case set q_j equal to any normalized vector orthogonal to all previously computed q_i and proceed to $j+1$. □

Theorem 2.12. *Every $A \in \mathbb{K}^{m\times n}$, $m \geq n$, with full rank has a unique QR factorization with $r_{jj} > 0$ for all $j = 1, \dots, n$.*

Proof. Since A has full rank, the algorithm does not fail. However, we could set $r_{jj} = -\|\cdots\|$ and obtain a different set of orthonormal vectors. Upon fixing these signs, the factorization becomes unique. □

Theorem 2.13. *To leading order Algorithm 2.5 requires $2mn^2$ flops.*

Proof. As usual the work is dominated by the inner for loop. There are $2m$ multiplications, $m-1$ additions and m subtractions. We compute

$$\sum_{j=1}^{n}\sum_{i=1}^{j-1}(4m-1) \approx 4m\sum_{j=1}^{n}\sum_{i=1}^{j-1}1 = 4m\sum_{j=1}^{n}(j-1) = 4m\sum_{j=1}^{n-1} j = 4m\frac{n(n-1)}{2}.$$

□

Unfortunately, the classical Gram-Schmidt process is well-known for its numerical instability. We illustrate its behaviour using a simple example.

Example 2.8. Let $\epsilon > 0$ be a small number and consider the matrix

$$\begin{bmatrix} 1 & 1 & 1 \\ \epsilon & & \\ & \epsilon & \\ & & \epsilon \end{bmatrix}.$$

Below we will go through the steps of the classical Gram-Schmidt process using the approximation $1 + \epsilon^2 \approx 1$.

First, since $r_{11} = \|a_1\|_2 = \sqrt{1+\epsilon^2} \approx 1$ we have $q_1 = a_1$.

Next, we compute $r_{12} = q_1^* a_2 = 1$ and $a_2 - r_{12}q_1 = (0, -\epsilon, \epsilon, 0)^\top$. This vector has length $r_{22} = \sqrt{2}\epsilon$ and therefore

$$q_2 = \frac{1}{r_{22}}\begin{bmatrix} 0 \\ -\epsilon \\ \epsilon \\ 0 \end{bmatrix} = \frac{1}{\sqrt{2}}\begin{bmatrix} 0 \\ -1 \\ 1 \\ 0 \end{bmatrix}.$$

Finally, we have $r_{13} = q_1^* a_3 = 1$ and $r_{23} = q_2^* a_3 = 0$. The vector $a_3 - r_{13}q_1 - r_{23}q_2 = (0, -\epsilon, 0, \epsilon)$ has again norm $r_{33} = \sqrt{2}\epsilon$. This gives

$$q_3 = \frac{1}{r_{33}}\begin{bmatrix} 0 \\ -\epsilon \\ 0 \\ \epsilon \end{bmatrix} = \frac{1}{\sqrt{2}}\begin{bmatrix} 0 \\ -1 \\ 0 \\ 1 \end{bmatrix}.$$

Now, the three vectors q_1, q_2 and q_3 should be orthonormal. While their lengths are fine ($\|q_1\|_2 \approx 1$ and $\|q_2\|_2 = \|q_3\|_2 = 1$), the two vectors q_2 and q_3 are not even close to orthogonal: $q_2^* q_3 = 1/2$.

2.4.2 Modified Gram-Schmidt

There is a slight modification to Algorithm 2.5 that improves matters. The inner for loop of the classical Gram-Schmidt process computes the following expression

$$a_j - (q_1^* a_j) q_1 - \cdots - (q_{j-1}^* a_j) q_{j-1} = \Big(I - \sum_{i=1}^{j-1} q_i q_i^* \Big) a_j,$$

which is the projection of a_j onto the orthogonal complement of $\operatorname{span}\{q_1, \ldots, q_k\}$. The basic idea is that this sum of projections can be written as a product.

Lemma 2.4. *Let $q_1, \ldots, q_k \in \mathbb{K}^m$ be a set of orthonormal vectors. Then*

$$I - \sum_{i=1}^{k} q_i q_i^* = (I - q_k q_k^*) \cdots (I - q_1 q_1^*) \,.$$

Proof. This statement can be proven by induction. The case $k = 2$ should clarify the general idea:

$$(I - q_2 q_2^*)\,(I - q_1 q_1^*) = I - q_1 q_1^* - q_2 q_2^* - q_2 q_2^* q_1 q_1^* = I - q_1 q_1^* - q_2 q_2^*,$$

because $q_1^* q_2 = 0$. □

Implementing this change is surprisingly simple. In Algorithm 2.5 we only have to replace the `A` in line 6 with a `V`:

```
R(i,j) = Q(:,i)' * V(:,j);
```

In contrast to the classical Gram-Schmidt, the auxiliary variable `V` is now updated in every iteration of the inner for-loop. In order to show how this change can lead to smaller errors, we revisit the previous example.

Example 2.9. For q_1 and q_2 we get the same results as before. Therefore $q_1 = a_1$ and

$$q_2 = \frac{1}{\sqrt{2}} \begin{bmatrix} 0 \\ -1 \\ 1 \\ 0 \end{bmatrix}.$$

Next, we have $r_{13} = q_1^* v_3^{(0)} = q_1^* a_3 = 1$. The first update of the auxiliary variable is $v_3^{(1)} = v_3^{(0)} - r_{13} q_1 = (0, -\epsilon, 0, \epsilon)^\top$. Now, $r_{23} = q_2^* v_3^{(1)} = \epsilon/\sqrt{2}$ and the second and last update of v_3 is given by

$$v_3^{(2)} = v_3^{(1)} - r_{23} q_2 = \epsilon(0, -1/2, -1/2, 1)^\top.$$

This vector has length $r_{33} = \epsilon\sqrt{6}/2$ and therefore

$$q_3 = \frac{v_3^{(2)}}{r_{33}} = \frac{1}{\sqrt{6}} \begin{bmatrix} 0 \\ -1 \\ -1 \\ 2 \end{bmatrix}.$$

Now the inner products $q_i^* q_j$, $i \neq j$, stay below $\epsilon/\sqrt{2}$ in absolute value.

However, while the modified Gram-Schmidt process in general performs better than the classical one, it also suffers from a significant so-called *numerical loss of orthogonality,* especially when A is close to rank-deficient. The following example[3] illustrates this behaviour and applies to both the classical and the modified Gram-Schmidt, since for 2×2 matrices they are equivalent.

Example 2.10. Consider the matrix

$$A = \begin{bmatrix} 0.70000 & 0.70711 \\ 0.70001 & 0.70711 \end{bmatrix}.$$

On a machine with a precision of 5 digits r_{11} is computed to be $r_{11} = 0.98996$. The first column of Q equals

$$q_1 = a_1/r_{11} = \begin{bmatrix} 0.70710 \\ 0.70711 \end{bmatrix}.$$

Next, $r_{12} = 1.0000$ and finally, by means of cancellation, the rounding errors become prevalent

$$a_2 - r_{12}q_1 = \begin{bmatrix} 0.00001 \\ 0.00000 \end{bmatrix}, \quad q_2 = \begin{bmatrix} 1.0000 \\ 0.0000 \end{bmatrix}.$$

Clearly, q_1 and q_2 are far from orthogonal.

2.4.3 Householder's Method

Dissatisfied with our previous two algorithms for computing QR factorizations, we consider a third one: the Householder method. Its basic idea is to transform A into generalized upper triangular form R via multiplication from the left with orthogonal/unitary matrices

$$\underbrace{Q_n \cdots Q_1}_{Q^*} A = R.$$

As a product of orthogonal/unitary matrices Q^* is itself orthogonal/unitary and so is its inverse

$$Q = Q^{**} = Q_1^* \cdots Q_n^*,$$

[3]This example is taken from *Numerical Linear Algebra* by Trefethen and Bau.

and therefore $A = QR$ is a full QR factorization.

The matrix Q_k should be such that it introduces zeros in the k-th column below the main diagonal while leaving previously introduced zeros untouched, for example

$$\underbrace{\begin{bmatrix} * & * & * \\ * & * & * \\ * & * & * \\ * & * & * \end{bmatrix}}_{A} \xrightarrow{Q_1} \underbrace{\begin{bmatrix} * & * & * \\ & * & * \\ & * & * \\ & * & * \end{bmatrix}}_{Q_1 A} \xrightarrow{Q_2} \underbrace{\begin{bmatrix} * & * & * \\ & * & * \\ & & * \\ & & * \end{bmatrix}}_{Q_2 Q_1 A} \xrightarrow{Q_3} \underbrace{\begin{bmatrix} * & * & * \\ & * & * \\ & & * \\ & & \end{bmatrix}}_{Q_3 Q_2 Q_1 A = R}.$$

If Q_k is to leave the first $k-1$ columns unchanged while being an orthogonal/unitary matrix, it should have the following block structure

$$Q_k = \begin{bmatrix} I & \\ & H_k \end{bmatrix}, \tag{2.8}$$

where I is the $(k-1)$-dimensional identity matrix and H_k is another $(m-k+1) \times (m-k+1)$ orthogonal/unitary matrix. On the other hand, the matrix we are operating on in the k-th step should look like this

$$Q_{k-1} \cdots Q_1 A = \begin{bmatrix} U & B \\ & C \end{bmatrix},$$

where U is a $(k-1)\times(k-1)$ upper triangular matrix and B, C have no particular structure. Therefore the result of step k reads as follows

$$\underbrace{\begin{bmatrix} I & \\ & H_k \end{bmatrix}}_{Q_k} \underbrace{\begin{bmatrix} U & B \\ & C \end{bmatrix}}_{Q_{k-1} \cdots Q_1 A} = \underbrace{\begin{bmatrix} U & B \\ & H_k C \end{bmatrix}}_{Q_k \cdots Q_1 A}.$$

The matrix H_k should introduce zeros in the first column c_1 of C, more precisely it should turn the first column of C into a multiple of $e_1 = (1, 0, \ldots, 0)^\top$

$$H_k c_1 = s e_1,$$

where $s \in \mathbb{K}$. However, H_k is orthogonal/unitary and therefore norm-preserving, which implies that

$$\|c_1\|_2 = \|H_k c_1\|_2 = \|s e_1\|_2 = |s|.$$

Therefore, the scalar s is not arbitrary, but must equal $\sigma \|c_1\|_2$, where $|\sigma| = 1$. If $\mathbb{K} = \mathbb{R}$, then there are two possibilities for s. If, however, $\mathbb{K} = \mathbb{C}$, then there is a whole circle of possibilities.

Supposing that we have fixed a value for σ, there is more than one orthogonal/unitary transformation H_k that maps the vector c_1 to $\sigma \|c_1\|_2 e_1$. The Householder method selects H_k as the matrix that reflects across the hyperplane orthogonal to the vector

$$v = c_1 - \sigma \|c_1\|_2 e_1. \tag{2.9}$$

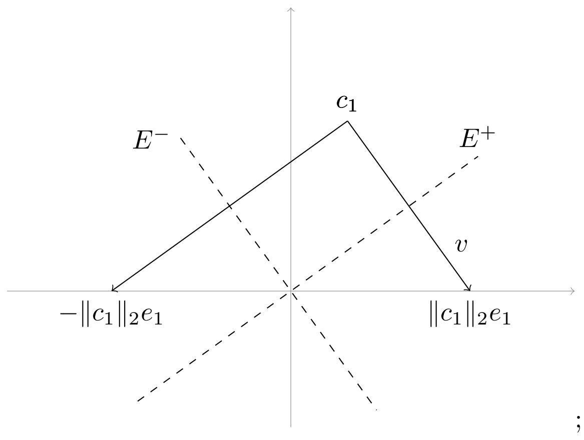

Figure 2.1: 2D sketch of the Householder reflection.

See Fig. 2.1 for a two-dimensional sketch of two such reflections.

The question of how to find this reflection remains. First, we could try to construct the matrix that does not reflect accross E^+, say, but projects onto it. Recall (or see the section *Projections* in the appendix) that the orthogonal projection that projects onto the hyperplane orthogonal to v, i.e. onto the orthogonal complement of span $\{v\}$, is given by

$$I - \frac{v}{\|v\|_2}\frac{v^*}{\|v\|_2} = I - \frac{vv^*}{v^*v}.$$

Now, if we want to reflect accross E^+, we just have to go twice as far into the same direction. Therefore,

$$H_k = I - 2\frac{vv^*}{v^*v} \tag{2.10}$$

is the matrix we have been looking for. It is called *Householder reflection.*

The final question we need to answer is how to choose the σ in (2.9). The guiding principle for this decision is numerical stability. More specifically, in order to avoid cancellation in (2.9) we should choose σ such that v is as large as possible in norm. Therefore

$$\sigma = -\frac{c_{11}}{|c_{11}|}. \tag{2.11}$$

If $c_{11} = 0$, we can choose σ arbitrarily, e.g. equal to one. In this case cancellation cannot occur.

Formulas (2.8), (2.9), (2.10) and (2.11) completely define the orthogonal transformation Q_k which we want to apply in step k of the Householder method. Thus we are in a position to write down the algorithm. The following Matlab code takes as input an $m \times n$ matrix A where $m \geq n$.

Algorithm 2.6: Householder Method

```
V = zeros(m,n);
for k = 1:n
    V(1:m-k+1,k) =  A(k:m,k);
    if A(k,k) ~= 0
        sig = sign(A(k,k));
    else
        sig = 1;
    end
    V(1,k) = V(1,k) + sig*norm(A(k:m,k));
    V(1:m-k+1,k) = V(1:m-k+1,k)/norm(V(:,k));
    A(k:m,k:n) = A(k:m,k:n) - ...
        ... 2*V(1:m-k+1,k)*(V(1:m-k+1,k)'*A(k:m,k:n));
end
R=A;
```

Note that none of the matrices Q_k or Q have been constructed explicitly, but only the vectors v. It takes additional work to construct Q, but often this is not necessary. First, recall that

$$Q^* = Q_n \cdots Q_1 \qquad \text{and} \qquad Q = Q_1^* \cdots Q_n^* = Q_1 \cdots Q_n, \tag{2.12}$$

the last equality being due to the fact that the Householder reflections H_k are self-adjoint and therefore the Q_k are as well. Now, if we needed to compute the matrix-vector product Q^*b for some $b \in \mathbb{K}^m$ we could do so without forming Q^* with the following lines of code.

Algorithm 2.7: Calculation of Q^*b

```
for k = 1:n
    b(k:m) = b(k:m) - ...
        ... 2*V(1:m-k+1,k)*(V(1:m-k+1,k)'*b(k:m));
end
```

Or, if we wanted to compute Qx for some $x \in \mathbb{K}^m$:

Algorithm 2.8: Calculation of Qx

```
for k = n:-1:1
    x(k:m) = x(k:m) - ...
        ... 2*V(1:m-k+1,k)*(V(1:m-k+1,k)'*x(k:m));
end
```

Note how, because of (2.12), the index `k` increases in the first snippet of code but decreases in the second one.

Theorem 2.14. *To leading order Algorithm 2.6 requires $2mn^2 - \frac{2}{3}n^3$ operations.*

Proof. The proof is very similar to previous ones about operation counting. We therefore omit it. □

Theorem 2.15. *The Householder method is backward stable. More precisely, for every $A \in \mathbb{K}^{m\times n}$ with $m \geq n$, Algorithm 2.6 implemented on a machine satisfying* (1.2) *and* (1.3) *computes factors $\tilde{Q}$ and $\tilde{R}$ satisfying*

$$\tilde{Q}\tilde{R} = A + \Delta A,$$

where the error matrix ΔA is such that

$$\frac{\|\Delta A\|}{\|A\|} \leq Cu \quad \textit{for some } C > 0.$$

As stated the theorem above is sloppy. Recall that Algorithm 2.6 does *not* compute $\tilde{Q}$. Instead it explicitly avoids computation of the orthogonal matrix as this would mean additional work. How should the theorem be interpreted then? The algorithm *does* compute vectors $\tilde{v}$ that define the Householder reflections. Therefore, $\tilde{Q}$ is the exactly unitary matrix satisfying $\tilde{Q} = \tilde{Q}_1 \cdots \tilde{Q}_n$ and the $\tilde{Q}_k$ are the matrices defined *mathematically* by equations (2.8) through (2.11).

2.4.4 Comparison of Algorithms for Solving $Ax = b$

Suppose A is a regular matrix. In Chapter 2 we have so far encountered three matrix factorizations. Combined with forward and/or back substitution each of them gives rise to a different method for solving a system of linear equations $Ax = b$.

LU factorization

1. Factorize $PA = LU$
2. Solve $Ly = Pb$ for y via forward substitution
3. Solve $Ux = y$ for x via back substitution

Cholesky factorization

1. Factorize $A = R^*R$
2. Solve $R^*y = b$ for y via forward substitution
3. Solve $Rx = y$ for x via back substitution

QR factorization

1. Factorize $A = QR$
2. Compute the vector $y = Q^*b$
3. Solve $Rx = y$ for x via back substitution

Cholesky factorization can only be used if A is an SPD matrix. In this case, however, it is the method of choice. It is always backward stable and is the fastest of three methods, only requiring $n^3/3$ operations. If A is not SPD, then LU factorization is the standard method. It is backward stable in practice and requires $2n^3/3$ operations.

If A is ill-conditioned, then QR factorization is a good choice. Recall Example 1.10 where we have shown that the condition number κ_2 is invariant under orthogonal/unitary transformations. Therefore $\kappa_2(A) = \kappa_2(QR) = \kappa_2(R)$. This means that the QR approach does not lead to additional error amplification. (Also recall our brief discussion about condition numbers at the beginning of Chapter 2 on page 17.) The Householder method is a backward stable algorithm for computing QR factorizations. For square matrices it requires $4n^3/3$ operations and is therefore twice as costly as LU factorization. In the next section we will encounter another application for the QR factorization.

2.5 Linear Least Squares Problems

Linear least squares problems are a fundamental class of problems with a wide range of applications throughout the mathematical sciences. In finite dimensions it can be stated as follows. Let $m \geq n$. For given matrix $A \in \mathbb{K}^{m\times n}$ and right hand side $b \in \mathbb{K}^m$ find a vector $x \in \mathbb{K}^n$ that minimizes $\|Ax - b\|_2$. Note that minimizing $\|Ax - b\|_2$ is equivalent to minimizing $\|Ax - b\|_2^2$. Hence the name *least squares.*

For $m > n$ the overdetermined system $Ax = b$ does not have a solution in general. It only does, if $b \in \operatorname{ran} A$, which is an n-dimensional subspace of $\mathbb{K}^m$. Therefore, minimizing the norm of the *residual* $Ax-b$ is, in some sense, the best one can do. Of course, the choice of the 2-norm is arbitrary and different norms will lead to different solutions. The 2-norm gives rise to a solution $x \in \mathbb{K}^n$ such that Ax is the closest point (in the Euclidean sense) in $\operatorname{ran} A$ to b.

The following theorem gives two equivalent characterizations of solutions to the linear least squares problem as stated above.

Theorem 2.16. *Let $m \geq n$, $A \in \mathbb{K}^{m\times n}$ and $b \in \mathbb{K}^m$. A vector $x \in \mathbb{K}^n$ solves the linear least squares problem, that is, it satisfies*

$$\|Ax - b\|_2 \leq \|Ay - b\|_2, \quad \textit{for all } y \in \mathbb{K}^n, \tag{2.13}$$

if and only if it solves the normal equations

$$A^*Ax = A^*b, \tag{2.14}$$

or equivalently

$$Ax = Pb, \tag{2.15}$$

where P is the orthogonal projection onto the range of A. The solution x is unique, if and only if A has full rank.

Proof. We first show that x is a solution, iff (2.15) is satisfied. Second, we prove equivalence of (2.14) and (2.15). Finally, we address the uniqueness of x.

First of all we need a generalization of Pythagoras' theorem: Let $x, y \in \mathbb{K}^m$ be a pair of orthogonal vectors, that is, $x^*y = 0$. Then

$$\|x + y\|_2^2 = (x + y)^*(x + y) = x^*x + x^*y + y^*x + y^*y = \|x\|_2^2 + \|y\|_2^2.$$

That is, the sum of squared norms equals the squared norm of the sum. Next, we use this identity to rewrite the squared norm of the residual

$$\|Ax - b\|_2^2 = \|Ax - Pb + Pb - b\|_2^2 = \|Ax - Pb\|_2^2 + \|Pb - b\|_2^2. \qquad (2.16)$$

Why does Pythagoras' theorem apply here? We need to verify that the two vectors $Ax - Pb$ and $Pb - b$ are orthogonal. The first vector $Ax - Pb \in \operatorname{ran} A = \operatorname{ran} P$, since P by definition maps onto the range of A. For the second vector we have $Pb - b = (P - I)b \in \operatorname{ran}(I - P)$. Recall (or see the section *Projections* in the appendix) that the range of an orthogonal projection is always orthogonal to the range of its complementary projection $I - P$. Therefore, the splitting of the norm in (2.16) is justified. We rewrite

$$\|Ax - b\|_2^2 = \|Ax - Pb\|_2^2 + \|Pb - b\|_2^2 \geq \|Pb - b\|_2^2,$$

since obviously $\|Ax - Pb\|_2^2 \geq 0$. This inequality provides us with a lower bound on $\|Ax - b\|_2^2$ that is independent of x. If there is an x such that $\|Ax - b\|_2^2$ attains this lower bound, then this x must be a solution of the least squares problem. However, $\|Ax - b\|_2^2$ attaining the lower bound $\|Pb - b\|_2^2$ is equivalent to $Ax = Pb$, which always has a solution, since $Pb \in \operatorname{ran} A$. Thus we have shown that (2.13) and (2.15) are equivalent.

Another way of writing the equation $Ax = Pb$ is $P(Ax - b) = 0$. This means that the residual vector $Ax - b$ lies in the kernel of P. However, since P is the orthogonal projection onto the range of A we have

$$\ker P \perp \operatorname{ran} P = \operatorname{ran} A.$$

Thus, the residual $Ax - b$ is orthogonal to the range of A. Since the range of A is nothing but the column space of A, we can express this orthogonality as

$$a_j^*(Ax - b) = 0, \quad \text{for } j = 1, \ldots, n,$$

where a_j is the j-th column of A. Yet another way of writing this is

$$A^*(Ax - b) = 0,$$

which is just the system of normal equations (2.14).

Finally, x is the unique solution of the least squares problem, if and only if (2.14) has a unique solution. But this is equivalent to the matrix A^*A being regular, which is again equivalent to A having full rank. □

2.5.1 Numerical Solution

There are several ways to solve the linear least squares problem numerically. One possibility is to exploit the fact that A^*A is an SPD matrix, if A has full rank, and to solve the normal equations using Cholesky factorization. The main steps for this approach are:

1. Compute A^*A and A^*b

2. Compute the Cholesky factorization $A^*A = R^*R$
3. Solve $R^*y = A^*b$ via forward substitution
4. Solve $Rx = y$ via back substitution

The work for this approach is dominated by the computation of the matrix A^*A, which requires mn^2 operations, and the Cholesky factorization. To leading order the total work required is therefore $mn^2 + n^3/3$. Solving the normal equations is fast, but the computation of the matrix A^*A can lead to numerical instabilities.

Another possibility is to use QR factorization. Here, we can exploit the fact that, if A has reduced QR factorization $A = \hat{Q}\hat{R}$, then the orthogonal projection P is given by $P = \hat{Q}\hat{Q}^*$. In this case the system (2.15) simplifies to

$$Ax = Pb \quad \Leftrightarrow \quad \hat{Q}\hat{R}x = \hat{Q}\hat{Q}^*b \quad \Leftrightarrow \quad \hat{R}x = \hat{Q}^*b.$$

The main algorithmic steps are therefore:

1. Compute a reduced QR factorization $A = \hat{Q}\hat{R}$
2. Compute the vector $\hat{Q}^*b$
3. Solve $\hat{R}x = \hat{Q}^*b$ via back substitution

The work for this approach is dominated by the QR factorization, which, if the Householder method is used, requires $\sim 2mn^2 - 2n^3/3$ operations. Solving the least squares problem using the QR factorization is slower than the previous approach. It does, however, avoid computation of A^*A and is therefore more stable in general.

Chapter 3

Interpolation

One way to think of interpolation is as a special case of data fitting. Suppose you are given n data points $(x_1, y_1), \dots, (x_n, y_n) \in \mathbb{K}^2$. *Data fitting* is the problem of finding a (simple) function $\phi : \mathbb{K} \to \mathbb{K}$ that captures the trend of the data. This could be achieved by minimizing a cost function such as $E[\phi] = \sum_i |\phi(x_i) - y_i|^2$ over a set Φ of of admissible functions. The choice of cost function and which functions are admissible depends a lot on the considered application. For example, if the data are oscillatory, then trigonometric functions ϕ might be a good choice. On the other hand, if the data display a decay behaviour, then functions of the form $\phi(x) = ae^{-\lambda x} + be^{-\mu x}$ might be appropriate.

Data interpolation is a special type of data fitting, where you require the function ϕ to exactly pass through all the data points, that is,

$$\phi(x_i) = y_i \quad \text{for } i = 1, \dots, n.$$

In this chapter we will consider interpolation with three types of functions: polynomials, splines, and trigonometric polynomials.

3.1 Polynomial Interpolation

At the heart of polynomial interpolation lies the following well-known theorem.

Theorem 3.1. *Let $x_0 < \cdots < x_n \in \mathbb{R}$ and $y_0, \dots, y_n \in \mathbb{R}$. Then there is a unique interpolating polynomial $p : \mathbb{R} \to \mathbb{R}$ of degree at most n, that is,*

$$p(x_i) = y_i \quad \textit{for } i = 0, \dots, n.$$

Before we discuss the general problem of polynomial interpolation let us go through a simple example.

Example 3.1 (Interpolating four points with a cubic polynomial.)**.** Consider the data points $(-2, 10)$, $(-1, 4)$, $(1, 6)$ and $(2, 3)$. The previous theorem tells us that there is a unique cubic polynomial

$$p(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3$$

satisfying the four equations

$$p(-2) = 10, \quad p(-1) = 4, \quad p(1) = 6, \quad p(2) = 3.$$

These are four equations in the four unknowns a_0, a_1, a_2, a_3. In matrix-vector notation they can be written as

$$\begin{bmatrix} 1 & -2 & 4 & -8 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & 1 & 1 \\ 1 & 2 & 4 & 8 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} 10 \\ 4 \\ 6 \\ 3 \end{bmatrix}.$$

Since the matrix is regular, we have a unique solution

$$\begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} = \frac{1}{12} \begin{bmatrix} 54 \\ 23 \\ 6 \\ -11 \end{bmatrix}.$$

3.1.1 The General Problem

In the rest of this section we assume to be given $n+1$ real data points

$$(x_0, y_0), \ldots, (x_n, y_n) \in \mathbb{R}^2,$$

where the x_i are pairwise distinct. We want to find the unique polynomial p of degree less or equal to n

$$p(x) = a_0 + a_1 x + \cdots + a_n x^n \tag{3.1}$$

such that

$$p(x_i) = y_i \quad \text{for } i = 0, \ldots, n.$$

Since p is completely determined by its coefficients, the polynomial interpolation problem now consists in finding the $n+1$ real numbers $a_i \in \mathbb{R}$. Plugging the $n+1$ data points into the general formula for $p(x)$ leads to the following system of $n+1$ linear equations

$$\begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^n \\ 1 & x_1 & x_1^2 & \cdots & x_1^n \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^n \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{bmatrix}.$$

The $(n+1) \times (n+1)$ matrix arising in this way is called *Vandermonde matrix.* The polynomial interpolation problem has a unique solution, if and only if the Vandermonde matrix is regular. From the previous theorem we can infer that it must be regular, but how to see this?

Lemma 3.1. *The Vandermonde matrix* $V = (x_i^j)_{i,j=0}^n$ *is regular.*

Proof. A square matrix A is regular, if the zero vector $x = 0$ is the only vector satisfying $Ax = 0$. (See the theorem in the *Invertibility* section of the appendix.) We use this argument to show that V is regular.

Suppose $c \in \mathbb{R}^{n+1}$ is a coefficient vector satisfying $Vc = 0$. An equivalent way of writing this is

$$\sum_{i=0}^{n} c_i x_j^i = 0 \quad \text{for } j = 0, \ldots, n.$$

But this means that the polynomial

$$p(x) = c_0 + c_1 x + \cdots + c_n x^n,$$

which is of degree $\leq n$, has $n + 1$ zeros. (Recall that the x_j are pairwise distinct.) This contradicts the Fundamental Theorem of Algebra, which states that p cannot have more than n zeros, unless p is the zero polynomial. Thus p must be the zero polynomial, or equivalently $c = 0$. $\square$

In summary, the polynomial interpolation problem reduces to a system of linear equations, which always has a unique solution. Therefore, at least from a theoretical perspective, we have solved the polynomial interpolation problem. In practice, however, it turns out to be more intricate, since Vandermonde matrices tend to have very large condition numbers.

Above, in equation (3.1), we have chosen to represent the polynomial p as a linear combination of the "canonical" basis $\{1, x, x^2, \ldots, x^n\}$. Yet there are many other bases in which polynomials can be expressed and the choice of basis strongly influences the condition number of the resulting Vandermonde matrix.

Lemma 3.2. *Let $n \in \mathbb{N}$ and denote by $\mathbb{P}_n$ the set of all polynomials $p : \mathbb{R} \to \mathbb{R}$ of degree $\leq n$.*

1. *The set $\mathbb{P}_n$ is a vector space of dimension $n + 1$.*

2. *The* monomials $\{1, x, x^2, \ldots, x^n\}$ *form a basis of $\mathbb{P}_n$.*

3. *The* Newton polynomials $\{\omega_0(x), \omega_1(x), \ldots, \omega_n(x)\}$ *form a basis of $\mathbb{P}_n$, where*
$$\omega_j(x) = \begin{cases} 1, & j = 0, \\ \displaystyle\prod_{k=0}^{j-1} (x - x_k), & j = 1, \ldots, n. \end{cases}$$

4. *The* Lagrange polynomials $\{L_0(x), L_1(x), \ldots, L_n(x)\}$ *form a basis of $\mathbb{P}_n$, where*
$$L_j(x) = \begin{cases} 1, & j = 0, \\ \displaystyle\prod_{\substack{k=0 \\ k \neq j}}^{n} \frac{x - x_k}{x_j - x_k}, & j = 1, \ldots, n. \end{cases}$$

The statement "$\{q_0(x), \dots, q_n(x)\}$ forms a basis of $\mathbb{P}_n$" means we can write every polynomial of order $\leq n$ as a linear combination of the q_i. Also note that, in contrast to the monomial basis, the Newton and Lagrange polynomials depend on the interpolation points x_i.

Example 3.2. Consider the following interpolation points

$$\begin{aligned}(x_0, y_0) &= (1, 1),\\ (x_1, y_1) &= (2, 4),\\ (x_2, y_2) &= (3, 9).\end{aligned}$$

The associated Newton polynomials

$$\begin{aligned}\omega_0(x) &= 1,\\ \omega_1(x) &= x - x_0 = x - 1,\\ \omega_2(x) &= (x - x_0)(x - x_1) = (x - 1)(x - 2)\end{aligned}$$

form a basis of $\mathbb{P}_2$. Therefore, the unique quadratic interpolating polynomial can be written as

$$\begin{aligned}p(x) &= a_0\omega_0(x) + a_1\omega_1(x) + a_2\omega_2(x)\\ &= a_0 + a_1(x - 1) + a_2(x - 1)(x - 2).\end{aligned}$$

Plugging in the three interpolation points leads to the following system

$$\begin{bmatrix}1 & & \\ 1 & 1 & \\ 1 & 2 & 2\end{bmatrix}\begin{bmatrix}a_0\\ a_1\\ a_2\end{bmatrix} = \begin{bmatrix}1\\ 4\\ 9\end{bmatrix},$$

which has solution $a = (1, 3, 1)^\top$ and therefore $p(x) = x^2$.

On the other hand, the Lagrange polynomials associated to the interpolation points (x_i, y_i) are given by

$$\begin{aligned}L_0(x) &= \frac{x - x_1}{x_0 - x_1}\frac{x - x_2}{x_0 - x_2} = \frac{x-2}{1-2}\frac{x-3}{1-3} = \frac{1}{2}(x-2)(x-3),\\ L_1(x) &= \frac{x - x_0}{x_1 - x_0}\frac{x - x_2}{x_1 - x_2} = \frac{x-1}{2-1}\frac{x-3}{2-3} = -(x-1)(x-3),\\ L_2(x) &= \frac{x - x_0}{x_2 - x_0}\frac{x - x_1}{x_2 - x_1} = \frac{x-1}{3-1}\frac{x-2}{3-2} = \frac{1}{2}(x-1)(x-2).\end{aligned}$$

They also form a basis of $\mathbb{P}_2$, which means we can write

$$p(x) = b_0L_0(x) + b_1L_1(x) + b_2L_2(x).$$

This time we obtain the system

$$\begin{bmatrix}1 & & \\ & 1 & \\ & & 1\end{bmatrix}\begin{bmatrix}b_0\\ b_1\\ b_2\end{bmatrix} = \begin{bmatrix}1\\ 4\\ 9\end{bmatrix}.$$

The system is very easy to solve this time. We have $b = y$ and therefore

$$p(x) = \sum_{i=0}^{2} y_i L_i(x) = x^2.$$

Remark 3.1. In the previous example the Vandermonde matrix for the Newton basis turned out to be lower triangular. In fact, this is always the case, since the Newton polynomials have the property

$$\omega_j(x_i) = 0 \quad \text{for } i < j.$$

Similarly, the Lagrange polynomials satisfy

$$L_j(x_i) = \delta_{ij},$$

and therefore they always let you write down the interpolating directly in terms of the y_i without any additional calculations required

$$p(x) = \sum_{i=0}^{n} y_i L_i(x). \tag{3.2}$$

We summarize the polynomial interpolation problem with the following theorem.

Theorem 3.2. *Let $n \in \mathbb{N}$, $x_0 < \cdots < x_n \in \mathbb{R}$ and $y_0, \ldots, y_n \in \mathbb{R}$. Suppose that $\{q_0, \ldots, q_n\}$ is a basis of $\mathbb{P}_n$. Then the unique interpolating polynomial p is given by*

$$p(x) = \sum_{i=0}^{n} a_i q_i(x),$$

where the vector of coefficients $(a_0, \ldots, a_n)^\top$ solves the regular system

$$\begin{bmatrix} q_0(x_0) & \cdots & q_n(x_0) \\ \vdots & & \vdots \\ q_0(x_n) & \cdots & q_n(x_n) \end{bmatrix} \begin{bmatrix} a_0 \\ \vdots \\ a_n \end{bmatrix} = \begin{bmatrix} y_0 \\ \vdots \\ y_n \end{bmatrix}.$$

3.1.2 Error Estimates

Frequently, the y_i happen to be the values of some function f, that is,

$$y_i = f(x_i) \quad \text{for } i = 0, \ldots, n.$$

In this case a natural question to ask is how well the interpolating polynomial p approximates f at points $x \neq x_i$.

Before we state our main result we introduce the space of k times continuously differentiable functions. Let $[a, b]$ be a closed interval and $k \in \mathbb{N} \cup 0$. Then we denote by $C^k[a, b]$ the space of all functions $f : [a, b] \to \mathbb{R}$ that are

k times differentiable and whose k-th derivative $f^{(k)}$ is continuous. If $k = 0$, then $C^k[a,b]$ denotes the space of continuous functions. By $C^\infty[a,b]$ we denote the space of infinitely differentiable functions on $[a,b]$. Clearly, if $m \geq k$, then $C^m[a,b] \subset C^k[a,b]$. For $f \in C^0[a,b]$ the following norm

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

is always finite.

Theorem 3.3. *Let $n \in \mathbb{N}$ and $a \leq x_0 < \cdots < x_n \leq b$. Suppose that $f \in C^{n+1}[a,b]$ and that $p \in \mathbb{P}_n$ is the unique interpolating polynomial, that is,*

$$p(x_k) = f(x_k), \quad \text{for } k = 0,\dots,n.$$

Then,

$$|p(x) - f(x)| \leq \frac{\|f^{(n+1)}\|_\infty |\omega_{n+1}(x)|}{(n+1)!}$$

for all $x \in [a,b]$, where $\omega_{n+1}(x) = \prod_{k=0}^n (x - x_k)$ is the $(n+1)$-th Newton polynomial.

An immediate consequence of this theorem is that, if $f \in C^\infty[a,b]$ has uniformly bounded derivatives, then the sequence of interpolating polynomials converges in the ∞-norm to f. Interestingly, the choice of interpolation points does not matter as long as their number increases steadily.

Corollary 3.1. *Let $f \in C^\infty[a,b]$ and assume that there exists an $M \geq 0$ such that*

$$\|f^{(n)}\|_\infty \leq M \quad \text{for all } n \in \mathbb{N}.$$

For every $n \in \mathbb{N}$ let $a \leq x_0^{(n)} < \cdots < x_n^{(n)} \leq b$ be a set of $n+1$ interpolation points and let $p_n \in \mathbb{P}_n$ be the unique polynomial interpolating f at these points, that is,

$$p_n\big(x_k^{(n)}\big) = f\big(x_k^{(n)}\big), \quad \text{for } k = 0,\dots,n.$$

Then,

$$\lim_{n\to\infty} \|p_n - f\|_\infty = 0.$$

Proof. By the previous theorem we have for every $n \in \mathbb{N}$ that

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

for all $x \in [a,b]$. Since the Newton polynomial satisfies

$$|\omega_{n+1}(x)| = \prod_{k=0}^n |x - x_k^{(n)}| \leq \prod_{k=0}^n |b-a| = (b-a)^n.$$

we get

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

But the fraction on the right hand side goes to zero as $n \to \infty$, because the factorial $(n+1)!$ grows much faster than the geometric sequence $(b-a)^n$. $\square$

If the assumption that f has uniformly bounded derivatives is not met, then the approximation quality of the interpolating polynomials can be arbitrarily poor. In fact, it is not difficult to find an example where $\|p_n - f\|_\infty \to \infty$.

Example 3.3 (Runge's phenomenon.)**.** Consider the function

$$f(x) = \frac{1}{1+x^2}$$

on the interval $[a,b] = [-5,5]$. It is infinitely differentiable, but the ∞-norms of its derivatives grow very fast. In particular, there is no number $M \ge 0$ such that $\|f^{(n)}\|_\infty \le M$ for all $n \in \mathbb{N}$. The polynomials p_n interpolating f at the equidistant points

$$x_k^{(n)} = a + k\frac{b-a}{n},$$

do *not* converge to f, but instead display oscillations near the interval boundaries. In fact the oscillations grow stronger as n increases. This is called *Runge's phenomenon.*

This example shows that polynomial interpolation for a large number of equidistant points should be avoided in general. But then the question is what to do in practice, if the number of interpolation points is large? There are several options. One of them is discussed in the next section.

3.2 Spline Interpolation

Instead of computing one interpolating polynomial *globally*, that is, for all interpolations points at once, one can divide the interval $[a,b]$ into several subintervals and then interpolate with a low degree polynomial on each subinterval. The resulting interpolating function is a *piecewise* polynomial, also called spline. The presentation of the topics covered in this section is along the lines of the corresponding chapter in *Grundlagen der Numerischen Mathematik und des Wissenschaftlichen Rechnens* by Hanke-Bourgeois.

Let $\Delta = (x_0, \dots, x_n)$ be a set of pairwise distinct points in $[a,b]$ such that $x_0 = a$, $x_n = b$, and $x_i < x_{i+1}$ for $0 \le i \le n-1$. We call Δ a *partition* of $[a,b]$. A *spline of order* $p \in \mathbb{N}$ is a function $s \in C^{p-1}[a,b]$ such that the restriction of s to each subinterval $[x_i, x_{i+1}]$ is a polynomial of degree $\le p$, in symbols,

$$s|_{[x_i,x_{i+1}]} \in \mathbb{P}_p \quad \text{for } i = 1 \dots n-1.$$

The set of all splines of order p on Δ is denoted by $\mathbb{S}^p(\Delta)$. Clearly, $\mathbb{P}_p \subset \mathbb{S}^p(\Delta)$, since every polynomial of degree p is also a spline of order p.

Lemma 3.3. *Let $n, p \in \mathbb{N}$ and Δ a partition of $[a,b]$ as defined above.*

1. *The set $\mathbb{S}^p(\Delta)$ is a vector space of dimension $n+p$.*
2. *Let $0 \leq k \leq p-1$. If $s \in \mathbb{S}^p(\Delta)$, then $s^{(k)} \in \mathbb{S}^{p-k}(\Delta)$. That is, the k-th derivative of a p-th order spline is a spline of order $p-k$.*

In these notes we consider interpolation with two types of splines. In Section 3.2.1 we treat the simplest case of *linear splines* ($p = 1$) and afterwards, in Section 3.2.2, we discuss interpolation using *cubic splines* ($p = 3$).

3.2.1 Linear Splines

Let Δ be a partition of $[a,b]$ and $y_0, \dots, y_n \in \mathbb{R}$. We want to find a linear interpolating spline, that is, a function $s \in \mathbb{S}^1(\Delta)$ satisfying

$$s(x_i) = y_i, \quad \text{for } i = 0, \dots, n. \tag{3.3}$$

According to Lemma 3.3 the space $\mathbb{S}^1(\Delta)$ has dimension $n+1$. In other words $n+1$ conditions are required to uniquely determine a linear spline. On the other hand, the interpolation problem (3.3) imposes just $n+1$ conditions on the unknown spline. This suggests that the linear spline interpolation problem has a unique solution.

Consider the subinterval $[x_i, x_{i+1}]$. On this subinterval we simply have to find the linear interpolating polynomial. We can, for instance, use equation (3.2) to express it in terms of the locally computed Lagrange polynomials

$$\ell_i(x) = y_i \frac{x - x_{i+1}}{x_i - x_{i+1}} + y_{i+1} \frac{x - x_i}{x_{i+1} - x_i}.$$

Having found a formula for all the local interpolants ℓ_i, we can write down the globally interpolating linear spline in the following way

$$s(x) = \begin{cases} \ell_0(x), & x \in [x_0, x_1], \\ \vdots & \\ \ell_{n-1}(x), & x \in [x_{n-1}, x_n]. \end{cases}$$

Another convenient way to express linear splines is in the basis of *hat functions.* The hat function Λ_i, $0 \leq i \leq n$, is defined as the unique linear spline satisfying

$$\Lambda_i(x_j) = \delta_{ij} \quad \text{for } j = 0, \dots, n.$$

Theorem 3.4. *For a given partition Δ of $[a,b]$ and values $y_0, \dots, y_n \in \mathbb{R}$. The unique interpolating linear spline is given by*

$$s(x) = \sum_{i=0}^{n} y_i \Lambda_i(x).$$

As for polynomials we can ask the following question. Suppose that $y_i = f(x_i)$ for some function $f : [a, b] \to \mathbb{R}$. How well does the piecewise linear interpolant s approximate f with respect to the ∞-norm? An answer can be derived directly from Theorem 3.3.

Corollary 3.2. *Let $f \in C^2[a, b]$ and denote by $s \in \mathbb{S}^1(\Delta)$ the linear spline that interpolates f at some partition Δ of $[a, b]$. Denote by $h_i = x_{i+1} - x_i$ the length of the i-th subinterval. Then*

$$\|s - f\|_\infty \le \frac{\|f''\|_\infty}{8} \max_{0 \le i \le n-1} h_i^2.$$

Proof. Applying Thm. 3.3 to the subinterval $[x_i, x_{i+1}]$ gives the estimate

$$|s(x) - f(x)| \le \frac{|\omega_2(x)|}{2} \max_{y \in [x_i, x_{i+1}]} |f''(y)|,$$

which holds for every $x \in [x_i, x_{i+1}]$. The Newton polynomial

$$\omega_2(x) = (x - x_i)(x - x_{i+1})$$

is a quadratic polynomial and therefore attains its extremum at the midpoint $(x_i + x_{i+1})/2$ of the subinterval

$$|\omega_2(x)| \le |\omega_2\left((x_i + x_{i+1})/2\right)| = \frac{h_i^2}{4}.$$

Thus we get

$$\begin{aligned} |s(x) - f(x)| &\le \frac{h_i^2}{8} \max_{y \in [x_i, x_{i+1}]} |f''(y)| \\ &\le \frac{1}{8} \|f''\|_\infty \max_{0 \le i \le n-1} h_i^2. \end{aligned}$$

□

The maximal error between f and s is controlled by the second derivative of f and the length of the longest subinterval. If we steadily decrease this length by adding more and more interpolation points, the error can be made arbitrarily small.

Corollary 3.3. *Let $f \in C^2[a, b]$. For every $n \in \mathbb{N}$ let $\Delta^{(n)}$ be a partition of $[a, b]$ consisting of $n + 1$ points. Denote the length of the longest subinterval of $\Delta^{(n)}$ by $h^{(n)}$ and assume that $h^{(n)} \to 0$ as $n \to \infty$. Let $s_n \in \mathbb{S}^1(\Delta^{(n)})$ be the linear spline interpolating f at $\Delta^{(n)}$. Then,*

$$\lim_{n \to \infty} \|s_n - f\|_\infty = 0.$$

Compare this result to Corollary 3.1. While in Cor. 3.1 the interpolation points can be chosen arbitrarily, the assumptions on the function f are much more restrictive.

3.2.2 Cubic Splines

Now, for a given partition $\Delta = (x_0, \dots, x_n)$ of $[a, b]$ and values $y_0, \dots, y_n \in \mathbb{R}$ we want to find a cubic spline $s \in \mathbb{S}^3(\Delta)$ such that $s(x_i) = y_i$ for all i. Recall Lemma 3.3. If Δ consists of $n+1$ points, then the space of all cubic splines $\mathbb{S}^3(\Delta)$ has dimension $n+3$. This suggests that, in contrast to the linear case, a cubic interpolating polynomial will not be uniquely determined by the $n+1$ interpolation conditions.

Every cubic spline s is a piecewise cubic polynomial that is two times continuously differentiable. According to Lemma 3.3, the second derivative of s is a linear spline and can therefore be expressed in the basis of hat functions

$$s''(x) = \sum_{i=0} \gamma_i \Lambda_i(x). \tag{3.4}$$

The coefficients γ_i are called the *moments* of the cubic spline s. Below we will make use of the following abbreviations

$$s_i' = s'(x_i), \qquad h_i = x_i - x_{i-1}.$$

The next lemma gives us an explicit formula for the cubic interpolation spline.

Lemma 3.4. *Assume $s \in \mathbb{S}^3(\Delta)$ satisfies the interpolation conditions*

$$s(x_i) = y_i \quad \textit{for} \quad i = 0, \dots, n.$$

Then, for $x \in [x_i, x_{i-1}]$ we have

$$s(x) = y_i + (x - x_i)s_i' + \gamma_i \frac{(x - x_i)^2}{2} + \frac{\gamma_i - \gamma_{i-1}}{h_i} \frac{(x - x_i)^3}{6}. \tag{3.5}$$

Proof. Let $x \in [x_i, x_{i-1}]$. We can write

$$\begin{aligned} s(x) - s(x_i) &= \int_{x_i}^{x} s'(t)\,dt \\ &= s'(t)(t - x)\Big|_{x_i}^{x} - \int_{x_i}^{x} s''(t)(t - x)\,dt, \end{aligned}$$

where we have used the integration by parts formula

$$\int_{x_i}^{x} u'(t)v(t)\,dt = u(t)v(t)\Big|_{x}^{x_i} - \int_{x_i}^{x} u(t)v'(t)\,dt$$

with $u(t) = t - x$ and $v(t) = s(t)$. Therefore $s(x)$ is given by

$$\begin{aligned} s(x) &= s(x_i) + s'(x_i)(x_i - x) - \int_{x_i}^{x} s''(t)(t - x)\,dt \\ &= y_i + (x_i - x)s_i' - \int_{x_i}^{x} s''(t)(t - x)\,dt. \end{aligned}$$

Next, we compute the integral on the right hand side. On $[x_i, x_{i-1}]$ the second derivative s'' is given by

$$\begin{aligned} s''(t) &= \gamma_{i-1}\Lambda_{i-1}(t) + \gamma_i \Lambda_i(t) \\ &= \gamma_{i-1}\frac{x_i - t}{x_i - x_{i-1}} + \gamma_i \frac{t - x_{i-1}}{x_i - x_{i-1}}, \end{aligned}$$

which, as can be easily verified, equals

$$= \frac{\gamma_i - \gamma_{i-1}}{h_i}(t - x_i) + \gamma_i.$$

Plugging this into the integral above we get

$$\begin{aligned} \int_{x_i}^{x} s''(t)(t-x)\,dt &= \frac{\gamma_i - \gamma_{i-1}}{h_i}\int_{x_i}^{x}(t-x_i)(t-x)\,dt + \gamma_i \int_{x_i}^{x}(t-x)\,dt \\ &= \frac{\gamma_i - \gamma_{i-1}}{h_i}\frac{(x-x_i)^3}{6} - \gamma_i \frac{(x_i - x)^2}{2}, \end{aligned}$$

which together with the above expression for $s(x)$ finishes the proof. □

Equation (3.5) completely determines the interpolating spline, as soon as we know the moments γ_i and the values of the first derivatives s_i'. The latter can be expressed in terms of the former by evaluating $s(x)$ at $x = x_{i-1}$ according to (3.5)

$$\begin{aligned} y_{i-1} &= y_i - h_i s_i' + \gamma_i \frac{h_i^2}{2} + (\gamma_{i-1} - \gamma_i)\frac{h_i^2}{6} \\ &= y_i - h_i s_i' + h_i^2\left(\frac{\gamma_i}{3} + \frac{\gamma_{i-1}}{6}\right). \end{aligned} \tag{3.6}$$

Now, a simple rearrangement of terms yields a formula for s_i' in which the only unknowns are γ_i and γ_{i-1}. Thus it remains to determine the moments.

Lemma 3.5. *The vector of moments $\gamma = (\gamma_0, \ldots, \gamma_n)^\top$ associated to the interpolating spline $s \in \mathbb{S}^3(\Delta)$ solves the system $A\gamma = d$, where $A \in \mathbb{R}^{(n-1)\times(n+1)}$ and $d \in \mathbb{R}^{n-1}$ are given by*

$$A = \frac{1}{6}\begin{bmatrix} h_1 & 2(h_1+h_2) & h_2 & & & \\ & h_2 & 2(h_2+h_3) & \ddots & & \\ & & \ddots & \ddots & h_{n-1} & \\ & & & h_{n-1} & 2(h_{n-1}+h_n) & h_n \end{bmatrix},$$

$$d = -\begin{bmatrix} -h_1^{-1} & h_1^{-1}+h_2^{-1} & -h_2^{-1} & & & \\ & -h_2^{-1} & h_2^{-1}+h_3^{-1} & \ddots & & \\ & & \ddots & \ddots & -h_{n-1}^{-1} & \\ & & & -h_{n-1}^{-1} & h_{n-1}^{-1}+h_n^{-1} & -h_n^{-1} \end{bmatrix}\begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{bmatrix}.$$

Proof. Rewrite (3.6) in the following way

$$-\frac{y_i - y_{i-1}}{h_i} = -s_i' + \gamma_i \frac{h_i}{3} + \gamma_{i-1}\frac{h_i}{6}.$$

Now subtract from this the same equation but with i replaced by $i+1$

$$\frac{y_{i+1} - y_i}{h_{i+1}} - \frac{y_i - y_{i-1}}{h_i} = s_{i+1}' - s_i' + \gamma_i \frac{h_i}{3} + \gamma_{i-1}\frac{h_i}{6} - \gamma_{i+1}\frac{h_{i+1}}{3} - \gamma_i \frac{h_{i+1}}{6}, \quad (3.7)$$

which holds for $1 \le i \le n-1$.

Next we want to get rid of the difference $s_{i+1}' - s_i'$. We first differentiate (3.5) with respect to x yielding

$$s'(x) = s_i' + \gamma_i(x - x_i) + \frac{\gamma_i - \gamma_{i-1}}{h_i}\frac{(x-x_i)^2}{2}.$$

Since this equation holds for every $x \in [x_{i-1}, x_i]$, we can plug in x_{i-1} for x. After a slight rearrangement of terms we arrive at

$$s_i' - s_{i-1}' = \frac{h_i}{2}(\gamma_i + \gamma_{i-1}).$$

Now we can replace the difference $s_{i+1}' - s_i'$ in (3.7) and after simplification of the appearing terms we finally end up with

$$\frac{y_{i+1} - y_i}{h_{i+1}} - \frac{y_i - y_{i-1}}{h_i} = \gamma_{i-1}\frac{h_i}{6} + \gamma_i \frac{h_i + h_{i+1}}{3} + \gamma_{i+1}\frac{h_{i+1}}{6}.$$

Again this equation holds for $1 \le i \le n-1$. Rewrite all these equations in matrix-vector form we end up with the system $A\gamma = d$ as asserted. □

The system $A\gamma = d$ is underdetermined, as there are $n+1$ unknowns but only $n-1$ equations. Therefore, two more conditions are needed in order to completely determine the moments. This confirms our reasoning from the beginning of this section. Additional conditions are usually imposed on s at the boundary of the interval $[a, b]$. A common choice for such boundary conditions are the so-called *natural boundary conditions*

$$s''(a) = s''(b) = 0.$$

Cubic splines that satisfy natural boundary conditions are called *natural cubic splines.*

Directly from the definition of moments (3.4) it follows that the moments γ_0 and γ_n of a natural cubic spline must vanish. In this case the system $A\gamma = d$ from Lemma 3.5 simplifies to

$$\frac{1}{6}\begin{bmatrix} 2(h_1+h_2) & h_2 & & \\ h_2 & 2(h_2+h_3) & \ddots & \\ & \ddots & \ddots & h_{n-1} \\ & & h_{n-1} & 2(h_{n-1}+h_n) \end{bmatrix} \begin{bmatrix} \gamma_1 \\ \gamma_2 \\ \vdots \\ \gamma_{n-1} \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_{n-1} \end{bmatrix}. \quad (3.8)$$

Theorem 3.5. *Let Δ be a partition of $[a,b]$ and $y_0,\dots,y_n \in \mathbb{R}$. Then there is a unique natural cubic spline $s \in \mathbb{S}^3(\Delta)$ such that $s(x_i) = y_i$ for $i = 0,\dots,n$.*

Proof. The system (3.8) is regular and therefore uniquely determines the moments, which in turn uniquely determine the interpolating cubic spline s via (3.5) and (3.6). □

Example 3.4. Let Δ be a partition consisting of $n+1$ equidistant points. That is, $x_i - x_{i-1} = h_i = h$. Fix a $j \in \{0,\dots,n\}$. The moments of the unique natural cubic spline that interpolates the values $y_i = \delta_{ij}$ solve the system

$$\frac{h}{6}\begin{bmatrix} 4 & 1 & & \\ 1 & 4 & \ddots & \\ & \ddots & \ddots & 1 \\ & & 1 & 4 \end{bmatrix}\begin{bmatrix} \gamma_1 \\ \gamma_2 \\ \vdots \\ \gamma_{n-1} \end{bmatrix} = -\frac{1}{h}\begin{bmatrix} -1 & 2 & -1 & & & \\ & -1 & 2 & -1 & & \\ & & \ddots & \ddots & \ddots & \\ & & & -1 & 2 & -1 \end{bmatrix} e_{j+1},$$

where as usual $e_{j+1} \in \mathbb{R}^{n+1}$ is the zero vector with a one at position $j+1$.

3.3 Trigonometric Interpolation

In this section we mainly deal with complex-valued functions. While the topic of trigonometric interpolation could be treated using only real-valued functions as well, the notation becomes much more tedious. Therefore, in all of Section 3.3 the symbol i denotes the complex unit $\sqrt{-1}$.

A *trigonometric polynomial of degree* $\leq n-1$ is a function $p : [0, 2\pi] \to \mathbb{C}$ which can be written as

$$p(x) = \sum_{j=0}^{n-1} c_j e^{ijx},$$

where $c_j \in \mathbb{C}$. We denote the set of all trigonometric polynomials of degree less or equal to $n-1$ by Π_{n-1}.

Lemma 3.6. *The set Π_{n-1} is a vector space of dimension n.*

Theorem 3.6. *Let $x_0 < \cdots < x_{n-1} \in [0, 2\pi]$ and $y_0,\dots,y_{n-1} \in \mathbb{C}$. There is a unique $p \in \Pi_{n-1}$ satisfying*

$$p(x_j) = y_j \quad \text{for } j = 0,\dots n-1. \tag{3.9}$$

For the rest of this section we only consider equidistant points

$$x_k = \frac{2\pi k}{n}, \quad k = 0,\dots,n-1.$$

In this case the resulting linear system will turn out to have a very special structure. Denote by

$$w_n = e^{\frac{2\pi i}{n}}$$

the *n-th root of unity.* Note that for $k = 0, \dots, n-1$ we have

$$w_n^k = \left(e^{\frac{2\pi i}{n}}\right)^k = e^{\frac{2\pi k i}{n}} = e^{ix_k}$$

and that $w_n^n = 1$. The interpolation conditions (3.9) can now be written as

$$p(x_k) = \sum_{j=0}^{n-1} c_j e^{ijx_k} = \sum_{j=0}^{n-1} c_j w_n^{jk} = y_k, \quad \text{for } k = 0, \dots n-1,$$

or in matrix-vector notation

$$F_n c = y, \tag{3.10}$$

with the *Fourier matrix* $F_n = (w_n^{jk})_{j,k=0}^{n-1} \in \mathbb{C}^{n \times n}$. This system is very easy to solve, once we have collected some properties of F_n.

Lemma 3.7. *The Fourier matrix is symmetric, that is, $F_n^\top = F_n$ and $\frac{1}{\sqrt{n}} F_n$ is unitary.*

Proof. The matrix F_n is symmetric, because $w_n^{jk} = w_n^{kj}$. For unitarity we have to show

$$\left(\frac{1}{\sqrt{n}} F_n\right)^* \frac{1}{\sqrt{n}} F_n = \frac{1}{n} F_n^* F_n = I.$$

Let v_k be the k-th column of F_n. Then

$$v_k^* v_k = \sum_{j=0}^{n-1} \overline{w_n^{jk}} w_n^{jk} = \sum_{j=0}^{n-1} |w_n^{jk}|^2 = \sum_{j=0}^{n-1} 1 = n.$$

On the other hand, if $\ell \neq k$, then

$$v_k^* v_\ell = \sum_{j=0}^{n-1} \overline{w_n^{jk}} w_n^{j\ell} = \sum_{j=0}^{n-1} w_n^{-jk} w_n^{j\ell} = \sum_{j=0}^{n-1} \left(w_n^{(\ell-k)}\right)^j,$$

which is a geometric sum. We use the well-known formula

$$\sum_{j=0}^{n-1} a^j = \frac{1-a^n}{1-a}$$

to get

$$v_k^* v_\ell = \frac{1 - w_n^{n(\ell-k)}}{1 - w_n^{\ell-k}} = 0,$$

because $w_n^n = 1$ and $w_n^{\ell-k} \neq 1$. Thus we have shown $F_n^* F_n = nI$. □

Theorem 3.7. *The solution to the trigonometric interpolation problem* (3.10)*, that is, the vector of coefficients $c = (c_0, \dots, c_{n-1})^\top \in \mathbb{C}^n$ of the interpolating polynomial is given by*

$$c = \frac{1}{n} \overline{F}_n y.$$

Proof. Since the matrix F_n is unitary up to a scaling factor, it is in particular invertible. Therefore the solution of (3.10) is given by $c = F_n^{-1} y$. It remains to show that $F_n^{-1} = \frac{1}{n}\bar{F}_n$. From the previous lemma we know that

$$\left(\frac{1}{\sqrt{n}} F_n\right)^{-1} = \left(\frac{1}{\sqrt{n}} F_n\right)^{*} = \frac{1}{\sqrt{n}} \overline{F}_n^{\top} = \frac{1}{\sqrt{n}} \overline{F}_n .$$

Combining this equality with the fact that $\left(\frac{1}{\sqrt{n}} F_n\right)^{-1} = \sqrt{n} F_n^{-1}$ completes the proof. □

3.3.1 Fast Fourier Transform

The function

$$\mathcal{F}_n : \mathbb{C}^n \to \mathbb{C}^n, \quad y \mapsto \mathcal{F}_n(y) = \overline{F}_n y.$$

is called *discrete Fourier transform* in analogy to its continuous counterpart. Note that the literature is not consistent in regards to this definition. Some authors include a factor of $\frac{1}{n}$ or $\frac{1}{\sqrt{n}}$ in the definition of $\mathcal{F}_n$.[1]

According to Thm. 3.7 solving the trigonometric interpolation involves two main tasks:

1. assembly of the matrix $\overline{F}_n$,
2. matrix-vector multiplication $\overline{F}_n y$.

For general matrices and vectors of dimension n the number of flops required for both tasks is quadratic in n. However, for reasons of periodicity F_n only has n different entries $w_n^0, \dots, w_n^{n-1}$. Moreover, using the *Fast Fourier Transform* (FFT) the matrix-vector multiplication can be realized with essentially $n \log_2 n$ operations. This is a significant speedup. For example, if $n = 2^{10} = 1024$, then $n \log_2 n \approx 10^4$ while $n^2 \approx 10^6$.

At the heart of the FFT lies, again, a matrix factorization. We start with an example.

Example 3.5. Consider the case of $n = 4$ interpolation points. The 4-th root of unity is $w_4 = e^{\frac{\pi i}{2}} = i$ and the Fourier matrix is given by

$$F_4 = \begin{bmatrix} w_4^0 & w_4^0 & w_4^0 & w_4^0 \\ w_4^0 & w_4^1 & w_4^2 & w_4^3 \\ w_4^0 & w_4^2 & w_4^4 & w_4^6 \\ w_4^0 & w_4^3 & w_4^6 & w_4^9 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{bmatrix}.$$

It turns out that F_4 can be factorized in the following way

$$F_4 = \begin{bmatrix} 1 & & 1 & \\ & 1 & & i \\ 1 & & -1 & \\ & 1 & & -i \end{bmatrix} \begin{bmatrix} 1 & 1 & & \\ 1 & -1 & & \\ & & 1 & 1 \\ & & 1 & -1 \end{bmatrix} \begin{bmatrix} 1 & & & \\ & & 1 & \\ & 1 & & \\ & & & 1 \end{bmatrix} \tag{3.11}$$

[1] In what follows we will not make explicit use of the function $\mathcal{F}_n$. We mainly mention it for the sake of completeness.

The matrix on the right is a permutation matrix P. When applied to a vector it puts the entries with even indices ahead of the entries with odd indices. Note, however, that in this section we start indexing with 0 instead of 1. Therefore, the matrix P maps $(c_0, c_1, c_2, c_3)^\top$ to $(c_0, c_2, c_1, c_3)^\top$.

The matrix in the middle, in fact, contains two copies of F_2:

$$\begin{bmatrix} 1 & 1 & & \\ 1 & -1 & & \\ & & 1 & 1 \\ & & 1 & -1 \end{bmatrix} = \begin{bmatrix} w_2^0 & w_2^0 & & \\ w_2^0 & w_2^1 & & \\ & & w_2^0 & w_2^0 \\ & & w_2^0 & w_2^1 \end{bmatrix} = \begin{bmatrix} F_2 & \\ & F_2 \end{bmatrix}.$$

Finally, the leftmost matrix in the factorization (3.11) can be generalized to higher dimensions by realizing that

$$\begin{bmatrix} 1 & & 1 & \\ & 1 & & i \\ 1 & & -1 & \\ & 1 & & -i \end{bmatrix} = \begin{bmatrix} I & D_2 \\ I & -D_2 \end{bmatrix},$$

where I is the 2×2 identity matrix and $D_2 = \operatorname{diag}(1, i) = \operatorname{diag}(w_4^0, w_4^1)$.

Therefore, the following sequence of steps leads to the same result as direct multiplication of F_4 with a vector $c = (c_0, c_1, c_2, c_3)^\top$.

1. Rearrange the entries of c so that the ones with even indices are ahead of those with odd indices.

$$\begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{bmatrix} \mapsto \begin{bmatrix} c_0 \\ c_2 \\ c_1 \\ c_3 \end{bmatrix} =: \begin{bmatrix} c_{\text{even}} \\ c_{\text{odd}} \end{bmatrix}$$

   Here we set $c_{\text{even}} = (c_0, c_2)^\top$ and $c_{\text{odd}} = (c_1, c_3)^\top$.

2. Multiply both the top and bottom halves of the resulting vector with F_2.

$$\begin{bmatrix} c_{\text{even}} \\ c_{\text{odd}} \end{bmatrix} \mapsto \begin{bmatrix} F_2 c_{\text{even}} \\ F_2 c_{\text{odd}} \end{bmatrix}$$

3. Multiply the bottom half of the resulting vector with D_2. Add the result to the top half to form the first two entries of the final result. Subtract it from the top half to form the last two entries of the final vector.

$$\begin{bmatrix} F_2 c_{\text{even}} \\ F_2 c_{\text{odd}} \end{bmatrix} \mapsto \begin{bmatrix} F_2 c_{\text{even}} + D_2 F_2 c_{\text{odd}} \\ F_2 c_{\text{even}} - D_2 F_2 c_{\text{odd}} \end{bmatrix} = F_4 c \tag{3.12}$$

The matrix factorization encountered in this example extends to general even $n \in \mathbb{N}$ in the following way.

Lemma 3.8. *Let $m \in \mathbb{N}$ and $n = 2m$. Then the $n \times n$ Fourier matrix F_n admits the following factorization*

$$F_n = \begin{bmatrix} I & D_m \\ I & -D_m \end{bmatrix} \begin{bmatrix} F_m & \\ & F_m \end{bmatrix} P_n, \tag{3.13}$$

where I is the $m \times m$ identity matrix, $D_m = \operatorname{diag}(w_n^0, \dots, w_n^{m-1})$ and P_n is the permutation matrix that maps every vector $c = (c_0, \dots, c_{n-1})^\top \in \mathbb{C}^n$ to

$$P_n c = (c_0, c_2, \dots, c_{n-2}, c_1, c_3, \dots, c_{n-1})^\top.$$

If in the previous theorem m is again an even number, $m = 2k$ say, we can factorize both copies of F_m in (3.13) and obtain

$$\begin{bmatrix} F_m & \\ & F_m \end{bmatrix} = \begin{bmatrix} I & D_k & & \\ I & -D_k & & \\ & & I & D_k \\ & & I & -D_k \end{bmatrix} \begin{bmatrix} F_k & & & \\ & F_k & & \\ & & F_k & \\ & & & F_k \end{bmatrix} \begin{bmatrix} P_m & \\ & P_m \end{bmatrix},$$

which yields a new factorization of the original Fourier matrix F_n

$$F_n = \begin{bmatrix} I & D_m \\ I & -D_m \end{bmatrix} \begin{bmatrix} I & D_k & & \\ I & -D_k & & \\ & & I & D_k \\ & & I & -D_k \end{bmatrix} \begin{bmatrix} F_k & & & \\ & F_k & & \\ & & F_k & \\ & & & F_k \end{bmatrix} \begin{bmatrix} P_m & \\ & P_m \end{bmatrix} P_n.$$

More generally, if $n = 2^p$, then we can perform this recursive decomposition p times, and compute the matrix-vector product $F_n c$ accordingly. This is the main idea behind the FFT. The following theorem shows that this approach pays off in terms of number of operations.

Theorem 3.8. *Let $p \in \mathbb{N}$, $n = 2^p$ and $c \in \mathbb{C}^n$. Computation of the product $F_n c$ as outlined above requires $\frac{3}{2} n \log_2 n$ flops.*

Proof. Using the recursive factorization approach the Fourier matrix F_n can be written as a product of $2p + 1$ matrices, p of which are permutation matrices and do not require arithmetic operations. The matrix in the middle will look like

$$\begin{bmatrix} F_1 & & \\ & \ddots & \\ & & F_1 \end{bmatrix},$$

which is just an identity matrix, because $F_1 = 1$. Thus we only need to consider the remaining p matrices on the left hand-side. Multiplication with one of these matrices requires $\frac{n}{2}$ multiplications, because, first, the matrices D_j are diagonal and, second, they are multiplied with only one half of the entries of each vector; recall (3.12). In addition each of the remaining p matrices requires n additions. The total number of flops is therefore

$$p\big(\frac{n}{2} + n\big) = \frac{3}{2} np = \frac{3}{2} n \log_2 n.$$

□

So far we have only discussed how to efficiently multiply with F_n, which up to a scaling factor corresponds to computing the *inverse* discrete Fourier transform $\mathcal{F}_n^{-1}$. The reasoning above, however, is easily modified to apply to $\overline{F}_n$ as well, for instance, by taking the complex conjugate of equation (3.13).

Chapter 4

Numerical Integration

The aim of numerical integration is to approximate the numerical value of definite integrals. In these notes we restrict our attention to one-dimensional integrals over intervals of finite length. Thus, for $a < b \in \mathbb{R}$ and $f : [a, b] \to \mathbb{R}$, we want to approximate the integral

$$I[f] = \int_a^b f(x)\,dx.$$

Clearly, if F is an antiderivative of f, that is $F'(x) = f(x)$, then $I[f] = F(b) - F(a)$. This raises the question what the purpose of numerical integration is. First, there are functions whose antiderivatives are difficult or impossible to express as elementary functions. (An elementary function is a function that is a finite combination of arithmetic operations, trigonometric functions, exponentials, logarithms, and so on.) An example of a function whose antiderivative is not elementary is $f(x) = e^{-x^2}$. Second, even if an antiderivative of f can be computed, it might just be more efficient to numerically integrate f instead of evaluating F. Finally, in certain situations the integrand might not be known on the whole interval $[a, b]$ but only at a few points. While this makes determining an antiderivative impossible in general, numerical approximation of $I[f]$ might still be feasible.

In order to numerically integrate a given function f we consider *quadrature rules* $Q[f] \approx I[f]$. These will take the following general form

$$Q[f] = \sum_{i=0}^{n} w_i f(x_i)$$

with *weights* $w_i \in \mathbb{R}$ and *nodal points* $x_i \in [a, b]$. That is, $Q[f]$ is a linear combination of a finite number of function values.

Example 4.1 (Midpoint rule)**.** The simplest example of such a quadrature rule is maybe the *midpoint rule*

$$Q[f] = (b - a) f\Big(\frac{a + b}{2}\Big).$$

The midpoint rule approximates the area below the function graph with the area of a rectangle with sides $b-a$ and $f((a+b)/2)$. Therefore it is exact for constant functions $f(x)=c$. However, it is not hard to see that even linear polynomials $f(x)=kx+d$ are integrated exactly by the midpoint rule. Using our notation from Section 3.1, we can formulate this property of the midpoint rule as

$$Q[p]=I[p] \quad \text{for all} \quad p \in \mathbb{P}_1.$$

Example 4.2 (Trapezoidal rule)**.** Another example is the *trapezoidal rule*

$$Q[f]=\frac{b-a}{2}f(a)+\frac{b-a}{2}f(b).$$

Visually it approximates the integral $I[f]$ with the combined area of two rectangles, one with sides $(b-a)/2$ and $f(a)$, and one with $(b-a)/2$ and $f(b)$, or equivalently with that of a trapezoid with vertices $(a,0)$, $(b,0)$, $(a,f(a))$, and $(b,f(b))$. The trapezoidal rule, too, is exact for polynomials of degree one.

In general, we say a quadrature rule Q has *degree* $k \in \mathbb{N}$, if all $p \in \mathbb{P}_k$ are integrated exactly, that is,

$$Q[p]=I[p] \quad \text{for all} \quad p \in \mathbb{P}_k.$$

The next result tells us what maximal degree we can expect of a quadrature rule with $n+1$ nodal points.

Lemma 4.1. *The degree of*

$$Q[f]=\sum_{i=0}^{n} w_i f(x_i)$$

cannot be higher than $2n+1$.

Proof. We prove the assertion by constructing a $p \in \mathbb{P}_{2n+2}$ for which $Q[p] \neq I[p]$. Define

$$p(x)=\prod_{k=0}^{n}(x-x_k)^2,$$

where the x_k are just the nodal points of the quadrature rule. The polynomial p is obviously of degree $2(n+1)=2n+2$. Since it only consists of quadratic terms it is nonnegative. Hence, its integral must be positive $I[p]>0$. On the other hand p has been constructed in such a way that $Q[p]=0$. □

4.1 Newton-Cotes Formulas

Newton-Cotes formulas are a special class of quadrature rules. Their basic idea is to replace the integrand f by the polynomial p that interpolates f at the nodal points x_i, that is, $Q[f] := I[p]$. The polynomial p can be integrated exactly by taking the right linear combination of its values at the nodal points.

More specifically, let f be given together with a set of nodal points $x_0, \ldots, x_n$. From Section 3.1 we know that the unique polynomial $p \in \mathbb{P}_n$ satisfying $p(x_i) = f(x_i)$ for $i = 0, \ldots, n$ can be expressed in terms of the Lagrange polynomials

$$p(x) = \sum_{i=0}^{n} f(x_i) L_i(x).$$

The main argument behind Newton-Cotes formulas is that $I[p]$ should be close to $I[f]$, if p is close to f. Therefore

$$\begin{aligned} I[f] \approx I[p] &= \int_a^b p(x)\,dx = \int_a^b \Big(\sum_{i=0}^{n} f(x_i) L_i(x) \Big)\,dx \\ &= \sum_{i=0}^{n} f(x_i) \underbrace{\int_a^b L_i(x)\,dx}_{w_i}\,. \end{aligned} \tag{4.1}$$

Lemma 4.2. *The degree of a Newton-Cotes formula with* $n+1$ *nodal points, as defined by* (4.1)*, is at least* n*.*

Proof. Let $p \in \mathbb{P}_n$. Then the unique polynomial interpolating p is p itself. Therefore $I[p] = Q[p]$. □

If the nodal points are equispaced and such that $x_0 = a$ and $x_n = b$, that is,

$$x_i = i\frac{b-a}{n}, \quad \text{for } i = 0, \ldots, n,$$

then we have a so-called *closed Newton-Cotes formula.* The closed Newton-Cotes formula for $n = 1$ is the trapezoidal rule, which can be shown by computing the two weights w_0 and w_1 according to (4.1). For $n = 2$ we obtain *Simpson's rule*

$$Q[f] = \frac{b-a}{6}\Big[f(a) + 4f\Big(\frac{a+b}{2}\Big) + f(b)\Big].$$

Lemma 4.2 tells us that all quadratic polynomials are integrated exactly by Simpson's rule.

If a and b are not nodal points, then the resulting Newton-Cotes formula is called *open.* Such quadrature rules can be used for improper integrals like

$$\int_0^1 \frac{dx}{\sqrt{x}},$$

where the integrand cannot be evaluated at one of the endpoints, but the integral is finite nonetheless.

4.1.1 Error estimates

The error estimates that we have obtained for polynomial interpolation can be used to estimate the error

$$|Q[f] - I[f]|$$

of a Newton-Cotes formula Q.

Example 4.3 (Error estimate for the trapezoidal rule)**.** The trapezoidal rule Q as defined in Example 4.2 is the closed Newton-Cotes formula for $n = 1$. This means that

$$Q[f] = \frac{b-a}{2}\,[f(a) + f(b)] = I[p],$$

where p is the linear polynomial that interpolates f at the two points $x_0 = a$ and $x_1 = b$. By Thm. 3.3 we have the following error estimate for p

$$|p(x) - f(x)| \leq \frac{\|f''\|_\infty}{2}|x-a||x-b|,$$

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

$$\begin{aligned}
|Q[f] - I[f]| = |I[p] - I[f]| &= \Big|\int_a^b \big(p(x) - f(x)\big)\,dx\Big| \\
&\leq \int_a^b \big|p(x) - f(x)\big|\,dx \\
&\leq \frac{\|f''\|_\infty}{2}\int_a^b (x-a)(b-x)\,dx \\
&= \frac{\|f''\|_\infty}{12}(b-a)^3.
\end{aligned}$$

Example 4.4 (Error estimate for Simpson's rule)**.** A similar derivation shows that Simpson's rule satisfies

$$|Q[f] - I[f]| \leq \frac{\|f^{(4)}\|_\infty}{2880}(b-a)^5.$$

Note that the quadrature error of Simpson's rule is controlled by the fourth derivative of the integrand. If f is a cubic polynomial, then $f^{(4)} = 0$ and the quadrature error vanishes. This implies that the degree of Simpson's rule is actually three and not merely two (as guaranteed by Lemma 4.2).

More generally, it can be shown that a Newton-Cotes formula with $n + 1$ nodal points has degree

$$r = \begin{cases} n, & \text{if } n \text{ is odd,} \\ n+1, & \text{if } n \text{ is even.} \end{cases}$$

Theorem 4.1. *A Newton-Cotes formula Q of degree r satisfies*

$$|Q[f] - I[f]| \leq C \frac{\|f^{(r+1)}\|_\infty}{(r+1)!}(b-a)^{r+2} \tag{4.2}$$

for all $f \in C^{r+1}[a,b]$, where the value of the constant $C > 0$ depends on the distribution of the nodal points.

Not surprisingly, Newton-Cotes formulas suffer from the same problems as polynomial interpolation does: A large number of nodal points $n+1$ should be avoided, unless the derivatives of f can be guaranteed to have small ∞-norms. Fortunately, the very idea that overcomes this problem in the case of interpolation also works for numerical integration.

4.2 Composite Rules

Instead of using Newton-Cotes formulas with a large number of nodal points, it is often better to subdivide the domain of integration $[a,b]$ and use quadrature rules with fewer points on every subinterval. This leads to the definition of a *composite rule* $Q_N[f]$

$$I[f] = \int_a^b f(x)\,dx = \sum_{j=1}^N \int_{a_j}^{b_j} f(x)\,dx \approx \sum_{j=1}^N Q^{(j)}[f] =: Q_N[f],$$

where $a = a_1 < b_1 = a_2 < \cdots < b_{N-1} = a_N < b_N = b$. The following theorem shows how this subdivision strategy affects the quadrature error.

Theorem 4.2. *Let Q_N be a composite rule of length $N \in \mathbb{N}$. Suppose that every subinterval is of equal length $b_j - a_j = \frac{b-a}{N}$ and that $Q^{(j)}$ is a closed Newton-Cotes formula of degree $r \in \mathbb{N}$ for every $j = 1, \dots, N$. Then,*

$$|Q_N[f] - I[f]| \leq C \frac{\|f^{(r+1)}\|_\infty}{(r+1)!} \frac{(b-a)^{r+2}}{N^{r+1}}, \tag{4.3}$$

for every $f \in C^{r+1}[a,b]$.

Proof. Applying (4.2) to every subinterval we get

$$\begin{aligned}
|Q_N[f] - I[f]| &= \Big|\sum_{j=1}^{N} Q^{(j)}[f] - \int_a^b f(x)\,dx\Big| \\
&= \Big|\sum_{j=1}^{N} \Big(Q^{(j)}[f] - \int_{a_j}^{b_j} f(x)\,dx\Big)\Big| \\
&\le \sum_{j=1}^{N} \Big|\Big(Q^{(j)}[f] - \int_{a_j}^{b_j} f(x)\,dx\Big)\Big| \\
&\overset{(4.2)}{\le} \sum_{j=1}^{N} C\frac{(b_j-a_j)^{r+2}}{(r+1)!} \max_{x\in[a_j,b_j]} \Big|f^{(r+1)}(x)\Big| \\
&= \frac{C}{(r+1)!}\sum_{j=1}^{N} \frac{(b-a)^{r+2}}{N^{r+2}} \max_{x\in[a_j,b_j]} \Big|f^{(r+1)}(x)\Big| \\
&\le C\frac{(b-a)^{r+2}}{(r+1)!N^{r+2}} \sum_{j=1}^{N} \|f^{(r+1)}\|_\infty \\
&= C\frac{\|f^{(r+1)}\|_\infty}{(r+1)!}\frac{(b-a)^{r+2}}{N^{r+1}}.
\end{aligned}$$

□

Note that the quadrature error of the composite rule (4.3) is equal to the error of the non-composite Newton-Cotes formula (4.2) divided by N^{r+1}. We can state the result of Thm. 4.2 more succinctly by saying that there is a $B \ge 0$ such that

$$|Q_N[f] - I[f]| \le B\Big(\frac{1}{N}\Big)^{r+1} \tag{4.4}$$

for every $f \in C^{r+1}[a,b]$. Since B is independent of N, we can decrease the quadrature error arbitrarily much by simply increasing the number of subintervals N. When a composite rule Q_N satisfies estimate (4.4), we say that Q_N *converges with order* $r+1$.

The trapezoidal rule, for instance, has degree $r = n = 1$. Therefore, by Thm. 4.2 the composite trapezoidal rule converges with order $r+1 = 2$. According to Example 4.4 Simpson's rule has degree $r = n+1 = 3$. Its order of convergence is therefore $r+1 = 4$.

4.3 Gauss Quadrature

Lemma 4.1 states that the degree of a quadrature rule with $n+1$ nodal points cannot be higher than $2n+1$. This raises the question whether it can actually be equal to $2n+1$.

Example 4.5 (Two-point Gauss-Legendre rule[1])**.** Let $[a,b]=[-1,1]$ and consider a general quadrature rule with $n+1=2$ nodal points

$$\int_{-1}^{1} f(x)\,dx \approx w_0 f(x_0) + w_1 f(x_1). \tag{4.5}$$

We want to find values for the weights $w_0, w_1 \in \mathbb{R}$ and points $x_0, x_1 \in [-1,1]$ such that (4.5) becomes an equality for all cubic polynomials $f \in \mathbb{P}_3$. If we plug in the four monomials $1, x, x^2, x^3$, we get a system of four nonlinear equations in four unknowns

$$\begin{aligned} w_0 + w_1 &= 2 \\ w_0x_0 + w_1x_1 &= 0 \\ w_0x_0^2 + w_1x_1^2 &= \frac{2}{3} \\ w_0x_0^3 + w_1x_1^3 &= 0. \end{aligned}$$

Subtracting x_0^2 times the second equation from the fourth yields

$$w_0x_0^3 + w_1x_1^3 - w_0x_0^3 - w_1x_1x_0^2 = w_1x_1(x_1^2 - x_0^2) = 0. \tag{4.6}$$

The weight w_1 cannot equal zero, since then Q would be a 1-point rule. For the same reason we must have $x_0 \neq x_1$. Therefore $x_0 = -x_1$ must hold in order to satisfy (4.6). Plugging this into the second equation gives

$$w_0x_0 + w_1x_1 = w_0x_0 - w_1x_0 = x_0(w_0 - w_1) = 0,$$

which implies $w_0 = w_1$. From the first equation we now get $w_0 = w_1 = 1$. Finally, the third equation yields $x_0 = -\frac{1}{\sqrt{3}}$ and $x_1 = \frac{1}{\sqrt{3}}$.

Thus, we have found the so-called *two-point Gauss-Legendre rule*

$$Q[f] = f\Big(\frac{-1}{\sqrt{3}}\Big) + f\Big(\frac{1}{\sqrt{3}}\Big),$$

which is exact for all polynomials of degree less than or equal to 3.

In general, the $(n+1)$*-point Gauss-Legendre rule*

$$Q[f] = \sum_{i=0}^{n} w_i f(x_i)$$

has maximal degree $2n+1$. Its weights and nodal points solve the $2n+2$ nonlinear equations

$$\sum_{i=0}^{n} w_i x_i^k = \int_{-1}^{1} x^k\,dx = \frac{1-(-1)^{k+1}}{k+1}, \quad k = 0, \dots, 2n+1.$$

[1]This example is taken from *Introduction to Scientific Computing* by Van Loan.

As above, the weights and nodes are usually computed for the interval $[-1,1]$ and tables containing their precise values can be found in the literature. If the domain of integration happens to be different from $[-1,1]$, one can transform the integral first using integration by substitution:

$$\int_a^b f(x)\,dx = \frac{b-a}{2}\int_{-1}^{1} f\left(\frac{a+b}{2} + x\frac{b-a}{2}\right)\,dx.$$

Finally, for the sake of completeness we mention that the $(n+1)$-point Gauss-Legendre rule satisfies the following error estimate

$$|Q[f] - I[f]| \leq \frac{(b-a)^{2n-3}\left[(n+1)!\right]^4}{(2n+3)\left[(2n+2)!\right]^3}\|f^{(2n+2)}\|_\infty.$$

Appendix on Linear Algebra

Linear dependence. Let $\mathbb{K} \in \{\mathbb{R}, \mathbb{C}\}$. We follow the usual convention that elements of the vector space $\mathbb{K}^n$ are identified with column vectors. A *subspace* of $\mathbb{K}^n$ is a subset $S \subset \mathbb{K}^n$ that is closed under addition and scalar multiplication. That is, for all $x, y \in S$ and scalars α, β the vector $\alpha x + \beta y$ lies again in S. Let $x_1, \dots, x_k \in \mathbb{K}^n$ be a set of vectors. Any sum of the form

$$\alpha_1 x_1 + \cdots + \alpha_k x_k$$

with $\alpha_i \in \mathbb{K}$ is called a *linear combination*. The set of all possible linear combinations is called *span* of $x_1, \dots, x_k$, in symbols

$$\operatorname{span}\{x_1, \dots, x_k\} = \{\alpha_1 x_1 + \cdots + \alpha_k x_k : \alpha_i \in \mathbb{K}\}.$$

The span of a set of vectors in $\mathbb{K}^n$ is always a subspace of $\mathbb{K}^n$. A set of vectors is called *linearly dependent*, if there is a nontrivial linear combination (not all $\alpha_i = 0$) that equals zero. If there is no such linear combination, the set is called *linearly independent*. Let S be a subspace of $\mathbb{K}^n$. A linearly independent set of vectors whose span equals S is called *basis of S*. The *dimension* of a space is the number of elements of any of its bases. For example, if the vectors $x_1, \dots, x_k$ are linearly independent, then they form a basis of their span, which in this case is k-dimensional.

Matrix products. Let $A = (a_{ij}) \in \mathbb{K}^{m \times n}$ be a matrix with m rows and n columns. For every $x \in \mathbb{K}^n$ the matrix-vector product $Ax = b \in \mathbb{K}^m$ is defined by

$$b_i = \sum_{j=1}^{n} a_{ij} x_j, \quad b = \begin{pmatrix} b_1 \\ \vdots \\ b_m \end{pmatrix}.$$

In this way, every matrix defines a function from $\mathbb{K}^n$ to $\mathbb{K}^m$

$$A : \mathbb{K}^n \to \mathbb{K}^m, x \mapsto Ax.$$

Every such function is *linear*, meaning that $A(\alpha x + \beta y) = \alpha Ax + \beta Ay$ for all $x, y \in \mathbb{K}^n$ and all $\alpha, \beta \in \mathbb{K}$. Conversely, it can be shown that every linear function between $\mathbb{K}^n$ and $\mathbb{K}^m$ can be represented by a matrix $A \in \mathbb{K}^{m \times n}$. Denote the j-th column of A by a_j. Then we can rewrite the matrix-vector product as $Ax = \sum_{j=1}^n x_j a_j$. Thus, Ax is a linear combination of the columns of A.

Let $A \in \mathbb{K}^{m\times n}$ and $B \in \mathbb{K}^{n\times k}$. The matrix-matrix product $AB = C \in \mathbb{K}^{m\times k}$ is defined by

$$c_{ij} = \sum_{\ell=1}^{n} a_{i\ell} b_{\ell j}, \quad 1 \le i \le m, \quad 1 \le j \le k.$$

Again denote the j-th column of B, C by b_j, c_j respectively. Then, c_j equals the matrix vector product Ab_j. In other words the j-th column of C is a linear combination of the columns of A where the coefficients are given by the entries of the j-th column of B. Matrix-matrix multiplication is associative and distributive over addition. It is *not* commutative.

Invertibility. The *range* of a matrix $A \in \mathbb{K}^{m\times n}$ is simply its range when regarded as a function from $\mathbb{K}^n$ to $\mathbb{K}^m$

$$\operatorname{ran} A = \{Ax : x \in \mathbb{K}^n\}.$$

The interpretation of the matrix-vector product as linear combination implies that $\operatorname{ran} A$ is equal to the span of the columns of A. This is the reason why $\operatorname{ran} A$ is sometimes called *column space* of A. The *row space* is defined analogously. It can be shown that the column and row spaces of a matrix always have the same dimension. This number is called *rank* of the matrix and denoted by $\operatorname{rank} A$. If $\operatorname{rank} A$ is maximal, that is, $\operatorname{rank} A = \min(m, n)$, then A is said to have *full rank.* The *kernel* or *nullspace* of a matrix is the set of all vectors that are mapped to zero by A

$$\ker A = \{x \in \mathbb{K}^n : Ax = 0\}.$$

The *identity* or *unit matrix* $I \in \mathbb{K}^{n\times n}$ is the matrix which has ones along its main diagonal and zeros elsewhere. It satisfies $IA = A$ for all matrices A with n rows and $BI = B$ for all matrices B with n columns. Let $A \in \mathbb{K}^{n\times n}$ be a square matrix. If there is another matrix B satisfying $AB = BA = I$, then A is called *invertible* or *regular.* In this case, the matrix B is unique, it is called the *inverse* of A and it is denoted by A^{-1}. The inverse of a product of two regular matrices is the reversed product of the inverse matrices: $(AB)^{-1} = B^{-1}A^{-1}$. A matrix which does not have an inverse is called *singular.* The following characterization of regular matrices is a standard result.

Theorem. *Let $A \in \mathbb{K}^{n\times n}$. The following statements are equivalent.*

1. *A is regular,*
2. *A is injective,*
3. *For every $b \in \mathbb{K}^n$ the linear system $Ax = b$ has exactly one solution.*
4. $\operatorname{ran} A = \mathbb{K}^n$,
5. $\operatorname{rank} A = n$,

6. $\ker A = \{0\}$,

7. $\det A \neq 0$.

Orthogonality. The *adjoint* or *conjugate transpose* of an $m \times n$ matrix A is the $n \times m$ matrix defined by $A^* = \bar{A}^\top$, where the bar means complex conjugation of all entries. If $A^* = A$, then A is called *self-adjoint.* In the complex case ($\mathbb{K} = \mathbb{C}$) A is also called *Hermitian.* If $\mathbb{K} = \mathbb{R}$ and $A^* = A$, then A is also called *symmetric.* The adjoint of a product of two matrices is the reversed product of the adjoint matrices: $(AB)^* = B^* A^*$.

Using the $*$-notation we define the *inner product* between two vectors $x, y \in \mathbb{K}^n$ as $x^* y$. (Note that every vector can be regarded as a single column matrix.) Also note that, if $\mathbb{K} = \mathbb{R}$, then the inner product is symmetric, whereas if $\mathbb{K} = \mathbb{C}$, then it is not. The inner product induces a norm on $\mathbb{K}^n$: $\|x\| = \sqrt{x^* x}$, the so-called *Euclidean norm,* which measures the length of vectors. The definition of a general norm together will be given below. The inner product allows one to measure the angle α between vectors as well

$$\cos\alpha = \frac{x^* y}{\|x\| \|y\|}.$$

Two vectors $x, y \in \mathbb{K}^n$ are *orthogonal*, if their inner product equals zero. The zero vector is orthogonal to every vector. A set of more than two nonzero vectors is called orthogonal, if they are all mutually orthogonal. A set of vectors $q_1, \ldots, q_\ell$ is called *orthonormal*, if it is orthogonal and all vectors have norm 1. This can be stated succinctly as $q_i^* q_j = \delta_{ij}$ for all i, j, where the *Kronecker delta* is defined by

$$\delta_{ij} = \begin{cases} 1, & i = j \\ 0, & i \neq j. \end{cases}$$

Sets of nonzero orthogonal vectors are linearly independent. For, if we suppose they are not, then

$$\alpha_1 x_1 + \cdots + \alpha_k x_k = 0,$$

where at least one α_j is nonzero. Multiplying both sides with x_j^* leads to $\|x_j\| = 0$, which is contradicts the assumption that all vectors are nonzero. Thus, every orthogonal set consisting of n vectors in $\mathbb{K}^n$ is a basis of $\mathbb{K}^n$. The best known example of an orthonormal basis is the *canonical basis* of $\mathbb{K}^n$: $e_1, \ldots, e_n$, where e_i consists only of zeros except for its i-th entry, which is one.

Orthonormal vectors are convenient to work with. One reason is that they allow us to easily decompose any vector into orthogonal components. To demonstrate this let $q_1, \ldots, q_k \in \mathbb{K}^n$ be an orthonormal set and $x \in \mathbb{K}^n$ an arbitrary vector. Then the vector

$$r = x - (q_1^* x) q_1 - \cdots - (q_k^* x) q_k \tag{7}$$

is orthogonal to all q_i. Multiply both sides with q_i^* to see this! Thus we have found a way to write x as a sum of $k + 1$ mutually orthogonal vectors. This decomposition suggests a procedure for turning a linearly independent set of vectors into an orthonormal set:

Let $x_1, \dots, x_\ell \in \mathbb{K}^n$ be linearly independent. We construct an orthonormal set $q_1, \dots, q_\ell$ inductively. Set $q_1 = x_1/\|x_1\|$. Then q_1 has norm one. To find the second vector we set

$$\tilde{q}_2 = x_2 - (q_1^* x_2) q_1.$$

By (7) this vector is orthogonal to q_1. After normalizing $q_2 = \tilde{q}_2/\|\tilde{q}_2\|$, the set $\{q_1, q_2\}$ is orthonormal. Now, suppose you have already found k orthonormal vectors, where $k \leq \ell - 1$. Then set

$$\tilde{q}_{k+1} = x_{k+1} - (q_1^* x_{k+1}) q_1 - \cdots - (q_k^* x_{k+1}) q_k \tag{8}$$

and normalize again to get q_{k+1}. After ℓ steps we have found an orthonormal set with the property

$$\operatorname{span}\{q_1, \dots, q_k\} = \operatorname{span}\{x_1, \dots, x_k\}$$

for all k between 1 and ℓ. This is the *Gram-Schmidt process.*

A square matrix $Q \in \mathbb{C}^{n \times n}$ is called *unitary*, if $Q^*Q = QQ^* = I$. Put differently, the inverse and the adjoint of Q coincide. A square matrix $Q \in \mathbb{R}^{n \times n}$ with the analogous property $Q^{-1} = Q^\top$ is called *orthogonal.* It is easy to see that the columns of Q form an orthonormal basis of $\mathbb{K}^n$: Denote the j-th column of Q by q_j. Then the (i,j) entry of Q^*Q equals $q_i^* q_j$. On the other hand the (i,j) entry of I equals δ_{ij}. Since $Q^*Q = I$, the columns of Q are an orthonormal basis of $\mathbb{K}^n$. An analogous argument for QQ^* instead of Q^*Q shows that the rows form an orthonormal basis as well.

Orthogonal and unitary matrices preserve the geometrical structure of the underlying space. Applying such a matrix to a pair of vectors does not change their inner product:

$$(Qy)^* Qx = y^* Q^* Q x = y^* x.$$

Consequently it does not change norms or angles either. Multiplying a vector with a 2×2 or 3×3 orthogonal matrix gives a rotated and/or reflected version of that vector.

Projections. A *projection (matrix)* is a square matrix that satisfies $P^2 x = Px$ for all x. Here, P^2 stands for the matrix-matrix product PP. Applying a projection matrix more than once to a vector does not change the result anymore. For example, identity matrices with an arbitrary number of ones deleted are projection matrices. If P is a projection, then

$$(I-P)^2 x = (I-P)(x - Px) = x - Px - Px + Px = (I-P)x$$

and therefore $I - P$ is a projection as well, the *complementary projection* to P. The complementary projection to $I - P$ is again P. Every projection decomposes $\mathbb{K}^n$ into the two subspaces $\operatorname{ran} P$ and $\ker P$ in the sense that

$$\operatorname{ran} P + \ker P = \mathbb{K}^n, \quad \text{while} \quad \operatorname{ran} P \cap \ker P = \emptyset.$$

This can be seen by writing $x = Px + (I - P)x$. Obviously, the first summand lies in $\operatorname{ran} P$, whereas the second one lies in $\operatorname{ran}(I - P)$. It remains to show that $\operatorname{ran}(I - P)$ actually equals $\ker P$. Assume that $y \in \operatorname{ran}(I - P)$. Then there is an x such that $Py = P(I - P)x = 0$. Thus $y \in \ker P$. Conversely, let $x \in \ker P$. Then $x = (I - P)x$ which shows $x \in \operatorname{ran}(I - P)$.

A projection is said to be *orthogonal,* if $\operatorname{ran} P$ is orthogonal to $\ker P$. This means that $x^*y = 0$ for all $x \in \ker P$ and $y \in \operatorname{ran} P$. An alternative characterization is this: A projection is orthogonal if and only if it is self-adjoint. The complementary projection to an orthogonal projection is again orthogonal, since $(I - P)^* = I^* - P^* = I - P$.

Every set of orthonormal vectors $q_1, \dots, q_k \in \mathbb{K}^n$ can be used to construct an orthogonal projection. Let $Q \in \mathbb{K}^{n\times k}$ be the matrix whose j-th column is q_j and define $P = QQ^*$. This matrix is an orthogonal projection, because

$$\begin{aligned} &P^2 = QQ^*QQ^* = QIQ^* = P, \quad \text{and} \\ &P^* = (QQ^*)^* = Q^{**}Q^* = P. \end{aligned}$$

The range of P is easily seen to be the span of $\{q_1, \dots, q_k\}$. Finally, notice that the j-th entry of Q^*x is q_j^*x and therefore, by the linear combination interpretation of the matrix-vector product, we have

$$Px = QQ^*x = \sum_{j=1}^{k} (q_j^*x)q_j.$$

This expression sheds new light on (7), which we can now rewrite as $r = (I-P)x$. The fact that r is orthogonal to all q_j can be deduced from what we have shown above. We have

$$r \in \operatorname{ran}(I - P) = \ker P \perp \operatorname{ran} P = \operatorname{span}\{q_1, \dots, q_k\},$$

where the symbol $\perp$ means “orthogonal to.”

Now we can also reformulate the Gram-Schmidt process using our new terminology. Suppose we have found k orthonormal vectors $q_1, \dots, q_k$. Arrange them as columns into a matrix $Q_k \in \mathbb{K}^{n\times k}$ and define the projection $P_k = Q_kQ_k^*$. Then

$$\tilde{q}_{k+1} = x_{k+1} - Q_kQ_k^*x_{k+1} = (I - P_k)x_{k+1},$$

and the normalization of $\tilde{q}_{k+1}$ is as before.

Norms. A *vector norm* is a function $\|\cdot\| : \mathbb{K}^n \to [0, +\infty)$ that assigns a length to each vector. It must satisfy the following three conditions. First, the zero vector is the only vector with length 0. Second, if a vector is multiplied by a scalar, then its length should scale accordingly. Third, the length of a sum of two vectors must never exceed the sum of their lengths. More precisely, for all $\alpha \in \mathbb{K}$ and $x, y \in \mathbb{K}^n$

1. $\|x\| = 0$ if and only if $x = 0$,
2. $\|\alpha x\| = |\alpha|\|x\|$,
3. $\|x + y\| \leq \|x\| + \|y\|$ (*triangle inequality*).

An important family of norms are the *p-norms*

$$\|x\|_p = (|x_1|^p + \cdots + |x_n|^p)^{\frac{1}{p}}, \quad p \in [1, +\infty),$$

and the *maximum norm*

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

The Euclidean norm, which we introduced above using the inner product, corresponds to the value $p = 2$. The p-norms satisfy the so-called *Hölder inequality*, which states that for all $x, y \in \mathbb{K}^n$

$$|x^* y| \leq \|x\|_p \|y\|_q,$$

where $p, q \in [1, +\infty]$ must be such that $1/p + 1/q = 1$. The fraction $1/{+\infty}$ is defined to be 0. The special case where $p = q = 2$ is called *Cauchy-Schwarz inequality*.

Two norms $\|\cdot\|$ and $\|\cdot\|'$ on $\mathbb{K}^n$ are called *equivalent*, if there are positive constants C_1 and C_2 such that the inequality

$$C_1 \|x\| \leq \|x\|' \leq C_2 \|x\| \tag{9}$$

holds for all $x \in \mathbb{K}^n$. It can be shown that *all* vector norms are equivalent. (The same holds true for matrix norms, see below.) For example, we have

$$\|x\|_\infty \leq \|x\|_2 \leq \sqrt{n}\|x\|_\infty$$

for all $x \in \mathbb{R}^n$. Because of this equivalence certain statements involving norms hold true independently of the choice of norm.

There are two ways of looking at $m \times n$ matrices. First, we can view them as elements of an mn-dimensional vector space. Second, we can interpret them as linear maps between $\mathbb{K}^n$ and $\mathbb{K}^m$. The first viewpoint leads to a definition analogous to the one for vectors: A *matrix norm* is a function $\|\cdot\| : \mathbb{K}^{m \times n} \to [0, +\infty)$ that satisfies, for all $\alpha \in \mathbb{K}$ and $A, B \in \mathbb{K}^{m \times n}$, the following conditions

1. $\|A\| = 0$ if and only if $A = 0$,
2. $\|\alpha A\| = |\alpha|\|A\|$,
3. $\|A + B\| \leq \|A\| + \|B\|$.

The second viewpoint leads to a definition of matrix norm that depends on the respective norms on $\mathbb{K}^n$ and $\mathbb{K}^m$: Fix norms on $\mathbb{K}^n$ and $\mathbb{K}^m$. Then the *induced matrix norm* of $A \in \mathbb{K}^{m \times n}$ is defined as

$$\|A\| = \sup_{x \neq 0} \frac{\|Ax\|}{\|x\|}. \tag{10}$$

Intuitively, the ratio $\|Ax\|/\|x\|$ measures the amount by which A has stretched the vector x. Thus, the number $\|A\|$ measures the maximal amount of stretching that the matrix A induces on $\mathbb{K}^n$.

Let us collect some properties of induced matrix norms. First, every induced matrix norm satisfies the three conditions above. Next, we can deduce directly from the definition that $\|Ax\| \le \|A\|\|x\|$ for all x. Consequently, we have for matrices $A \in \mathbb{K}^{m\times n}$ and $B \in \mathbb{K}^{n\times k}$

$$\|ABx\| \le \|A\|\|Bx\| \le \|A\|\|B\|\|x\|.$$

Dividing by $\|x\|$ and taking suprema on both sides shows the so-called *submultiplicativity* of induced matrix norms:

$$\|AB\| \le \|A\|\|B\|.$$

It is sometimes useful to realise that in (10) it is actually enough to take the supremum only over those $x \in \mathbb{K}^n$ with $\|x\| = 1$

$$\|A\| = \sup_{x\neq 0} \|A(x/\|x\|)\| = \sup_{\|x\|=1} \|Ax\|.$$

The induced norm of A^{-1} can be written as

$$\|A^{-1}\| = \sup_{x\neq 0} \frac{\|A^{-1}x\|}{\|x\|} = \sup_{y\neq 0} \frac{\|y\|}{\|Ay\|} = \left(\inf_{y\neq 0} \frac{\|Ay\|}{\|y\|}\right)^{-1} = \left(\inf_{\|y\|=1} \|Ay\|\right)^{-1}. \tag{11}$$

Finally, notice that all suprema and infima above are attained, which means that they are in fact maxima/minima.

The matrix norm induced by the p-norm on both $\mathbb{K}^n$ and $\mathbb{K}^m$ is denoted by a subscript p as well. That is,

$$\|A\|_p = \sup_{x\neq 0} \frac{\|Ax\|_p}{\|x\|_p}.$$

For $p = 1$ and $p = \infty$ this leads to

$$\|A\|_1 = \max_{1\le j\le n} \|a_j\|_1, \quad \text{and} \quad \|A\|_\infty = \max_{1\le i\le m} \|a_i^*\|_1. \tag{12}$$

respectively, where a_j denotes the j-th column and a_i^* the i-th row of A.

In order to explain the matrix norm that is induced by the Euclidean norm ($p = 2$) we need a few more definitions. A scalar $\lambda \in \mathbb{K}$ is called *eigenvalue* of $B \in \mathbb{K}^{n\times n}$, if there is a nonzero vector x such that $Bx = \lambda x$. The *spectral radius* of B is defined as

$$\rho(B) = \max\{|\lambda| : \lambda \text{ eigenvalue of } B\}.$$

The square roots of the eigenvalues of A^*A are called *singular values* of $A \in \mathbb{K}^{m\times n}$. Singular values are always nonnegative real numbers. This can be seen by multiplying the equation $A^*Ax = \lambda x$ from the left with x^* which leads to

$$\|Ax\|^2 = \lambda\|x\|^2.$$

Now it can be shown that the matrix norm induced by the Euclidean vector norm, that is,

$$\|A\|_2 = \max_{\|x\|_2=1} \|Ax\|_2. \tag{13}$$

equals the largest singular value of A, in symbols $\|A\|_2 = \sqrt{\rho(A^*A)}$. Hence its name *spectral norm.* Similarly, if A is regular, then $\|A^{-1}\|_2$ equals the reciprocal of the smallest singular value of A. Compare this statement to (11) and (13)!

One remarkable property of the spectral norm is that it is invariant under orthogonal or unitary transformations. Let U and V be two such matrices. Then

$$\begin{aligned} \|UAV\|_2^2 &= \max_{\|x\|_2=1} x^*V^*A^*U^*UAVx = \max_{\|x\|_2=1} \|AVx\|_2^2 \\ &= \max_{\|Vx\|_2=1} \|AVx\|_2^2 = \|A\|_2^2. \end{aligned} \tag{14}$$

Let us look at the spectral norm for two certain types of matrices: orthogonal/unitary ones and self-adjoint ones. The spectral norm of an orthogonal/unitary matrix equals one. For in this case A^*A is the identity matrix and all eigenvalues of the identity are one. If $A \in \mathbb{K}^{n\times n}$ is self-adjoint, then $\|A\|_2 = \rho(A)$. Due to the spectral theorem every self-adjoint matrix is *diagonalizable* by orthogonal/unitary matrices. This means that there is an orthogonal/unitary matrix U such that $A = U\Lambda U^*$, where $\Lambda = \operatorname{diag}(\lambda_1, \dots, \lambda_n)$ is a diagonal matrix and the λ_i are eigenvalues of A. Hence

$$\|A\|_2^2 = \rho(A^*A) = \rho(A^2) = \rho(U\Lambda^2U^*) = \max_{1\le i\le n} |\lambda_i|^2 = \rho(A)^2.$$

The most important matrix norm which is not an induced norm is the *Frobenius norm.* It corresponds to the Euclidean norm for vectors and is defined by

$$\|A\|_F = \Big(\sum_{ij} a_{ij}^2\Big)^{\frac{1}{2}}.$$

Linear systems of equations. A general system of m linear equations in n unknowns has the form

$$\begin{array}{cccccc} a_{11}x_1 + & \cdots & + a_{1n}x_n & = & b_1 \\ \vdots & \vdots & \vdots & & \vdots \\ a_{m1}x_1 + & \cdots & + a_{mn}x_n & = & b_m \end{array}$$

Using matrix-vector notation we can write this simply as $Ax = b$ with system matrix $A \in \mathbb{K}^{m\times n}$, vector of unknowns $x \in \mathbb{K}^n$ and right-hand side $b \in \mathbb{K}^m$. A system with more equations than unknowns ($m \geq n$) is called *overdetermined.* If there are less equations than unknowns ($m \leq n$), then the system is *underdetermined.* The $m \times (n+1)$ matrix that results from appending b to the right side of A is called *augmented matrix.* The Rouché-Capelli theorem states that the linear system $Ax = b$ has a solution, if and only if the augmented matrix has the same rank as A. If $\operatorname{rank}(A) = n$, then the solution is unique. Otherwise there are infinitely many solutions.

When solving a system of linear equations one often interchanges columns or rows or adds a multiple of a row to another. Such operations can be written as matrix-matrix multiplications. While row operations correspond to multiplication of the augmented matrix with another matrix from the left, column operations correspond to multiplication of A with another matrix from the right.

For example, interchanging columns k and ℓ can be realized by multiplying from the right with an identity matrix that has columns k and ℓ interchanged. Such a matrix is also called *permutation matrix*. Multiplying the same matrix from the left interchanges rows k and ℓ. Multiplying a row by $\alpha \in \mathbb{K}$ is realized by multiplication from the left with an identity matrix that has the corresponding 1 replaced wit α. Finally adding α times row k to row ℓ can be achieved by multiplying from the left with an identity that has an α inserted at position (ℓ, k).

In general, finding a matrix that performs a certain row operation on A is achieved by appending an identity matrix horizontally to A (i. e. to the left or right of A) and performing the row operation on the augmented matrix $(I \,|\, A)$ or $(A \,|\, I)$. The modified identity matrix is just the matrix that, if multiplied from the left with A, performs the desired row operation. For column operations the identity matrix must be appended vertically to A.