

Numerical Methods for Computational Science and Engineering

by Prof. Dr. Ralf Hiptmair, SAM, ETH Zurich

revised and modified by Prof. Dr. Rima Alaifari

(with contributions from Prof. P. Arbenz and Dr. V. Gradinaru)

Lecture Notes
Autumn Term 2018

Version of January 21, 2019



Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

SAM

Contents

1	Computing with Matrices and Vectors	1
1.1	Numerics & Error Analysis	1
1.1.1	Representation of numbers	2
1.1.2	Roundoff errors	4
1.1.3	Floating Point Operations	5
1.1.4	Estimating Roundoff Errors	6
1.2	Fundamentals	10
1.2.1	Notations	10
1.2.2	Classes of matrices	12
1.3	Software and Libraries	13
1.3.1	EIGEN	14
1.3.2	(Dense) Matrix storage formats	19
1.4	Computational effort	21
1.4.1	Asymptotic complexity	21
1.4.2	Cost of basic operations	23
1.5	Cancellation	23
1.6	Numerical stability	30
2	Direct Methods for Linear Systems of Equations	33
2.1	Existence and uniqueness of solutions	33
2.1.1	Solution of an LSE as a <i>problem</i>	34
2.1.2	Sensitivity of linear systems	35
2.2	Gaussian Elimination	38
2.2.1	Alternative way: LU-Decomposition	42
2.2.2	Using LU-factorization to solve a linear system of equations	43

2.3	Exploiting structure when solving Linear Systems	45
2.3.1	Block elimination	46
2.3.2	Low-rank perturbation/modification of an LSE	48
2.4	Sparse Linear Systems	51
2.4.1	Sparse matrix storage formats	51
2.4.2	Sparse matrices in EIGEN	53
2.4.3	Direct solution of sparse LSEs	55
3	Direct Methods for Linear Least Squares Problems	59
3.1	Least Squares Solutions	61
3.1.1	Normal Equations	62
3.1.2	Generalized Solutions & Moore-Penrose Pseudoinverse	63
3.2	Normal Equation Methods	65
3.3	Orthogonal Transformation Methods	68
3.3.1	QR-Decomposition	70
3.3.2	Householder Transformations	75
3.3.3	QR-Based Solver for Linear Least Squares Problems	80
3.4	Singular Value Decomposition (SVD)	81
3.4.1	Theory	81
3.4.2	SVD in EIGEN	84
3.4.3	Generalized solutions by SVD	85
3.4.4	SVD-Based Optimization and Approximation	86
4	Filtering Algorithms	95
4.1	Discrete Convolutions	96
4.1.1	Discrete finite linear time-invariant causal channels/filters	96
4.1.2	Transmission through LT-FIR filters	98
4.1.3	Discrete convolution	102
4.1.4	Circulant matrices	104
4.2	Discrete Fourier Transform (DFT)	106
4.2.1	Eigenvalues and eigenvectors of circulant matrices	106
4.2.2	Discrete Convolution via DFT	109
4.2.3	Fast Fourier Transform (FFT)	110
4.2.4	Frequency filtering via DFT	112
4.2.5	Two-dimensional DFT	116
5	Data Interpolation in 1D	121
5.1	Introduction	121
5.1.1	Piecewise linear interpolation	123
5.1.2	The general interpolation problem	125
5.2	Global Polynomial Interpolation	127
5.2.1	Lagrange Interpolation	128
5.2.2	Polynomial interpolation algorithms	129
5.2.3	Newton basis	131
5.2.4	Approximation of functions by interpolating polynomials	133
5.2.5	Chebyshev Interpolation	138

5.3	Piecewise polynomial interpolation	150
5.3.1	Piecewise polynomial Lagrange interpolation	150
5.3.2	Spline interpolation	154
6	Numerical Quadrature	159
6.1	Quadrature Formulas	160
6.1.1	Quadrature by approximation schemes	161
6.2	Polynomial Quadrature Formulas	163
6.2.1	Newton-Cotes formulas	164
6.3	Gauss Quadrature	166
6.3.1	Quadrature error and best approximation error	172
6.4	Composite Quadrature	176
6.4.1	The Composite trapezoidal and Composite Simpson rule	176
6.4.2	Errors and orders	178
7	Numerical Methods for ODEs	181
7.1	Initial value problems for ordinary differential equations	181
7.1.1	Terminology and notations related to ODEs	181
7.1.2	Modeling with ordinary differential equations: Examples	182
7.1.3	Initial value problems	185
7.1.4	Domain of definition of solutions of IVPs	187
7.1.5	Evolution operators	188
7.2	Polygonal Approximation Methods	190
7.3	General single step methods	194
7.3.1	Discrete evolution operators	195
7.3.2	Consistent single step methods	196
7.3.3	Convergence of single step methods	198
7.4	Higher order Single-step methods (Runge-Kutta Methods)	206
7.4.1	General form of Runge-Kutta methods	208
7.4.2	Consistency conditions for Runge-Kutta methods	211
7.5	Stability of Numerical Methods for ODEs	212
7.5.1	Absolute Stability	215
7.5.2	A-stability	223
7.5.3	Systems of linear ordinary differential equations	224
7.6	Stiff Initial Value Problems	226
8	Iterative Methods for Non-Linear Systems of Equations	231
8.1	Fixed Point Iterations in 1D	234
8.1.1	Algorithm for Root-Finding with Quadratic Convergence	239
8.1.2	Secant Method	240
8.2	Nonlinear Systems of Equations	241
8.2.1	Fixed Point Iterations in \mathbb{R}^n	243
8.2.2	Newton's Method in Higher Dimensions	247
8.2.3	Damped Newton Method	251
8.2.4	Quasi-Newton Method	253
8.3	Unconstrained Optimization	254
8.3.1	Optimization with a Differentiable Objective Function	256

Contents

8.3.2	Optimization with a Convex Objective Function	256
8.3.3	Optimization Algorithms	257
Index		261
Bibliography		261

Computing with Matrices and Vectors

1.1 Numerics & Error Analysis

In numerical methods, we leave the discrete world of `int`'s and `long`'s to describe real-world quantities by `float`'s or `double`'s. This transition brings challenges, as the real-world quantities are in general only approximated by finite representations on the computer.

Note. Computers cannot compute “properly” in \mathbb{R} : numerical computations may not respect the laws of analysis and linear algebra!

The reason why computers must fail to execute exact computations with real numbers is clear:

Computer = finite automaton \longrightarrow can handle only finitely many numbers, not \mathbb{R}

Essential property. \mathbb{M} , the set of machine numbers, is a finite, discrete subset of \mathbb{R} . Therefore, roundoff errors (*ger.:* Rundungsfehler) are inevitable.

The set of machine numbers \mathbb{M} cannot be closed under elementary arithmetic operations $+$, $-$, \cdot , $/$, that is, when performing the addition, multiplication, etc., of two machine numbers, the result may not belong to \mathbb{M} . The results of elementary operations with operands in \mathbb{M} have to be mapped back to \mathbb{M} , by an operation called *rounding*.

Let's look at the following simple example.

Code Snippet 1.1: Machine Arithmetic Example

```
10 #include <iostream>
11 using namespace std;
12
13 double a = 1.0;
14 double b = a/9.0;
15 if(a==b*9.0) cout << "They are equal";
16 else cout << "They are not equal";
17 /*
```

The output will be "They are not equal", because of the roundoff error which occurs in each computation. The exact query `==` can be changed to an approximate equality, as in the code below, which will print "They are equal":

Code Snippet 1.2: Machine Arithmetic Example

```
10 #include <limits>
11 #include <iostream>
12 using namespace std;
13
14 double a = 1.0;
15 double b = a/9.0;
16 if(fabs(a-b*9.0)<numeric_limits<double>::epsilon)
17     cout << "They are equal";
18 else cout << "They are not equal";
19 /*
```

1.1.1 Representation of numbers

The most straightforward way to store a fractional number is using a fixed decimal point system.

Fixed Point Representation

$$0.\underbrace{73125}_{\text{digits of mantissa}} \cdot \underbrace{10}_{\text{base}}^{12} \leftarrow \text{exponent} \in \mathbb{Z}$$

- Range: 10^{-k} to 10^l , $l, k \in \mathbb{Z}$
- Digits: $k + l + 1$ where k digits appear after the decimal point
- Advantage: simple implementation of arithmetic operations, for example $a + b = (a \cdot 10^k + b \cdot 10^k) \cdot 10^{-k}$
- Disadvantage: precision issue, for example $k = 1$, the operation 0.1×0.1 will output 0

This system can be used in cases that favor time over accuracy (e.g. some GPU systems). However, the “default” representation used is the *Floating Point Representation*.

Floating Point Representation

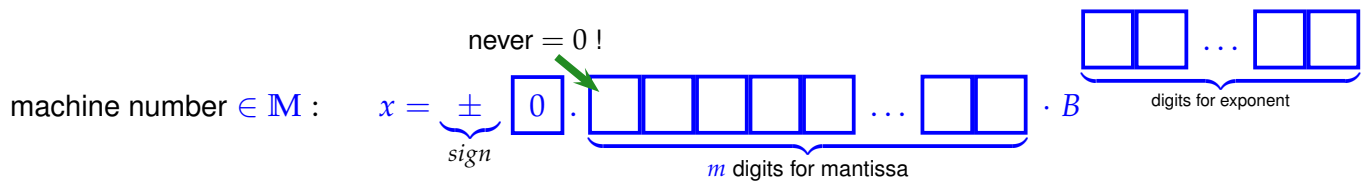
In various applications it is of utmost importance to have a reliable system which is sufficiently flexible for frequent changes of scales. For example, in chemistry the scale of computation could easily go from 10^{-31} to 10^{24} . Therefore, in many applications, a unified representation is desirable.

Definition 1.1.1 (Machine numbers/floating point numbers). Given:

- Basis $B \in \mathbb{N} \setminus \{1\}$
- exponent range $\{e_{\min}, \dots, e_{\max}\}$, $e_{\min}, e_{\max} \in \mathbb{Z}$ and $e_{\min} < e_{\max}$
- number $m \in \mathbb{N}$ of digits (for mantissa)

the corresponding set of *machine numbers* is:

$$\mathbb{M} := \{d \cdot B^E : d = i \cdot B^{-m}, i = B^{m-1}, \dots, B^m - 1, E \in \{e_{\min}, \dots, e_{\max}\}\}.$$



IEEE 754/IEC 559 standard for machine numbers

The standardisation of machine numbers is important because it ensures that the same numerical algorithm, executed on different computers will produce the same result.

The IEEE 754 standard includes 5 basic formats.

- 3 binary formats
 - binary32: single
 - binary64: double
 - binary128: quadruple
- 2 decimal formats
 - decimal64: double
 - decimal128: quadruple

The two most common formats are binary32 and binary64 which have the following parameters:

1 Computing with Matrices and Vectors

single precision : $m = 24^*$, $E \in \{-125, \dots, 128\} \rightarrow 4$ bytes

double precision : $m = 53^*$, $E \in \{-1021, \dots, 1024\} \rightarrow 8$ bytes

*: including 1 bit indicating the sign

Special cases in IEEE standard:

$E = e_{\max}$, $M \neq 0 \triangleq NaN = \text{Not a number} \rightarrow$ exception

$E = e_{\max}$, $M = 0 \triangleq Inf = \text{Infinity} \rightarrow$ overflow

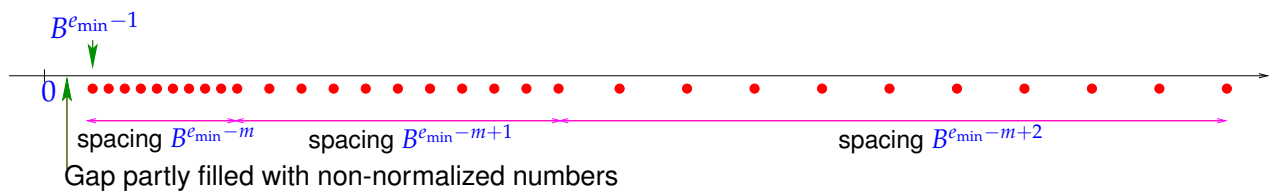
$E = 0 \triangleq$ Non-normalized numbers \rightarrow underflow

$E = 0$, $M = 0 \triangleq$ number 0

Code Snippet 1.3: Querying characteristics of double numbers \rightarrow GITLAB

```
10 #include <limits>
11 #include <iostream>
12 #include <iomanip>
13
14 using namespace std;
15
16 int main() {
17     cout << std::numeric_limits<double>::is_iec559 << endl
18     << std::defaultfloat << numeric_limits<double>::min() << endl
19     << std::hexfloat << numeric_limits<double>::min() << endl
20     << std::defaultfloat << numeric_limits<double>::max() << endl
21     << std::hexfloat << numeric_limits<double>::max() << endl;
22 }
23 /*
```

Remark. Machine numbers are not evenly spaced! Gaps are bigger for large number:



1.1.2 Roundoff errors

Example 1.1.1: Input Errors and Roundoff Errors

The following computations would always result in 0, if done in exact arithmetic.

Code Snippet 1.4: Demonstration of roundoff errors \rightarrow GITLAB

```
9 #include <iostream>
10 int main() {
```

```

11  std::cout.precision(15);
12  double a = 4.0/3.0, b = a-1, c = 3*b, e = 1-c;
13  std::cout << e << std::endl;
14  a = 1012.0/113.0; b = a-9; c = 113*b; e = 5+c;
15  std::cout << e << std::endl;
16  a = 83810206.0/6789.0; b = a-12345; c = 6789*b; e = c-1;
17  std::cout << e << std::endl;
18  }
19  /*

```

Output:

```

1  2.22044604925031e-16
2  6.75015598972095e-14
3  -1.60798663273454e-09

```

Can you devise a similar calculation, whose result is even farther off zero? Apparently the rounding that inevitably accompanies arithmetic operations in \mathbb{M} can lead to results that are far away from the true result.

For the discussion of errors introduced by rounding we need important notions.

Definition 1.1.2 (Absolute and relative error). Let $\tilde{x} \in \mathbb{C}$ be an approximation of $x \in \mathbb{C}$. Then its *absolute error* is given by

$$\epsilon_{\text{abs}} := |x - \tilde{x}| ,$$

and its *relative error* is defined as

$$\epsilon_{\text{rel}} := \frac{|x - \tilde{x}|}{|x|} .$$

Definition 1.1.3 (Number of Correct Digits). The *number of correct* (significant, valid) *digits* of an approximation \tilde{x} of $x \in \mathbb{C}$ is defined through the relative error:

$$\text{If } \epsilon_{\text{rel}} := \frac{|x - \tilde{x}|}{|x|} \leq 10^{-\ell} , \text{ then } \tilde{x} \text{ has } \ell \text{ correct digits, } \ell \in \mathbb{N}_0 .$$

1.1.3 Floating Point Operations

We may think of the elementary binary operations $+, -, *, /$ in \mathbb{M} as comprising of two steps:


- ❶ Compute the exact result of the operation.
- ❷ Perform rounding of the result of ❶ to map it back to \mathbb{M} .

Definition 1.1.4 (Correct rounding). Correct rounding (“rounding up”) is given by the function

$$\text{rd} : \begin{cases} \mathbb{R} & \rightarrow & \mathbb{M} \\ x & \mapsto & \max \arg \min_{\tilde{x} \in \mathbb{M}} |x - \tilde{x}| . \end{cases}$$

(Recall that $\operatorname{argmin}_x F(x)$ is the set of arguments of a real valued function F that makes it attain its (global) minimum.)

Of course, ❶ above is not possible in a strict sense, but the effect of both steps can be realised and yields a floating point realization of $\star \in \{+, -, \cdot, /\}$.

 **Notation:** Write $\tilde{\star}$ for the floating point realization of $\star \in \{+, -, \cdot, /\}$:

Then ❶ and ❷ may be summed up into

$$\text{For } \star \in \{+, -, \cdot, /\}: \quad x \tilde{\star} y := \operatorname{rd}(x \star y) .$$

Remark (Breakdown of Associativity). As a consequence of rounding, addition $\tilde{+}$ and multiplication $\tilde{\cdot}$, as implemented on computers, fail to be associative. They will usually be commutative, though this is not guaranteed.

1.1.4 Estimating Roundoff Errors

Let us denote by EPS the largest relative error (see Definition 1.1.2) incurred through rounding:

$$\text{EPS} := \max_{x \in I \setminus \{0\}} \frac{|\operatorname{rd}(x) - x|}{|x|} ,$$

where $I = [\min(\mathbb{M}_+), \max(\mathbb{M}_+)]$ is the range of positive machine numbers, and $\mathbb{M}_+ := \{x \in \mathbb{M}: x > 0\}$ is the set of strictly positive machine numbers.

For machine numbers according to definition 1.1.1, EPS can be computed from the defining parameters B (base) and m (length of mantissa) [1, p. 24]:

$$\text{EPS} = \frac{1}{2} B^{1-m} .$$

However, when studying roundoff errors, we do not want to delve into the intricacies of the internal representation of machine numbers. This can be avoided by just using a single bound for the relative error due to rounding, and, thus, also for the relative error potentially suffered in each elementary operation.

Assumption 1.1.1 (“Axiom” of roundoff analysis). *There is a small positive number EPS, the machine precision, such that, for the elementary arithmetic operations $\star \in \{+, -, \cdot, /\}$ and “hard-wired” functions $* f \in \{\exp, \sin, \cos, \log, \dots\}$, the following holds*

$$x \tilde{\star} y = (x \star y)(1 + \delta) \quad , \quad \tilde{f}(x) = f(x)(1 + \delta) \quad \forall x, y \in \mathbb{M} ,$$

with $|\delta| < \text{EPS}$.

Note. The relative roundoff errors of elementary steps in a program are bounded by the machine precision.

*this is an ideal, which may not be accomplished even by modern CPUs

Example 1.1.2: Machine Precision for IEEE Standard

In C++ we can get the machine precision as following:

Code Snippet 1.5: Finding out EPS in C++ → GITLAB

```

9  #include <iostream>
10 #include <limits> // get various properties of arithmetic types
11 int main() {
12     std::cout.precision(15);
13     std::cout << std::numeric_limits<double>::epsilon() << std::endl;
14 }
15 /*

```

Output:

```

1  2.22044604925031e-16

```

Knowing the machine precision can be important for checking the validity of computations or coding termination conditions for iterative approximations.

Example 1.1.3: Adding EPS to 1

Code Snippet 1.6: 1 + EPS

```

14 cout.precision(25);
15 double eps = numeric_limits<double>::epsilon();
16 cout << fixed << 1.0 + 0.5*eps << endl
17     << 1.0 - 0.5*eps << endl
18     << (1.0 + 2/eps) - 2/eps << endl;
19 /*

```

Output:

```

1  1.000000000000000000000000000000
2  0.99999999999999998889776975
3  0.000000000000000000000000000000

```

EPS is the smallest positive number $\in \mathbb{M}$ for which $1 \tilde{+} \text{EPS} \neq 1$ (in \mathbb{M}).

Note. We have to worry about roundoff errors because of accumulation and/or amplification.

Remark (Testing Equality With Zero). Since results of numerical computations are almost always polluted by roundoff errors, tests like `if (x == 0)` are pointless and even dangerous (if x contains the result of a numerical computation).

Instead test: `if (abs(x) < eps*s)`, with some positive number s which is small compared to $|x|$.

Overflow and Underflow

overflow : $|\text{result of an elementary operation}| > \max\{\mathbb{M}\}$

IEEE standard: `Inf`

underflow : $0 < |\text{result of an elementary operation}| < \min\{\mathbb{M}_+\}$

IEEE standard: Use subnormal numbers

Subnormal numbers are introduced to fill the gap between 0 and $\min\{\mathbb{M}_+\}$. This is done by allowing general digit strings for the mantissa (leading entry does not have to be non-zero). The axiom of roundoff analysis (Assumption 1.1.1) does *not hold* once subnormal numbers are encountered:

Code Snippet 1.7: Demonstration of over-/underflow → GITLAB

```

9  #include <iostream>
10 #define _USE_MATH_DEFINES
11 #include <cmath>
12 #include <limits>
13 using namespace std;
14 int main() {
15     cout.precision(15);
16     double min = numeric_limits<double>::min();
17     double res1 = M_PI*min/123456789101112;
18     double res2 = res1*123456789101112/min;
19     cout << res1 << endl << res2 << endl;
20 }
21 /*

```

Output:

```

1  5.68175492717434e-322
2  3.15248510554597

```

Underflow and overflow should be avoided.

Example 1.1.4: Avoiding overflow

A simple example showing how to avoid overflow during the computation of the norm of a 2D vector [1, Ex. 2.9]:

$$r = \sqrt{x^2 + y^2}.$$

Straightforward evaluation: there is an overflow when $|x| > \sqrt{\max |\mathbb{M}|}$ or $|y| > \sqrt{\max |\mathbb{M}|}$.

Whereas for

$$r = \begin{cases} |x| \sqrt{1 + \left(\frac{y}{x}\right)^2} & , \text{ if } |x| \geq |y| , \\ |y| \sqrt{1 + \left(\frac{x}{y}\right)^2} & , \text{ if } |y| > |x| , \end{cases}$$

there is no overflow.

Note. So far, we have only discussed roundoff errors. Other error sources exist such as discretization error, modelling error, measurement error.

Note. Relative or absolute errors are in general not computable since the true solution will be unknown. Therefore, we have to use other means to estimate the errors such as:

1. Worst case estimates
2. Compute backward error

Example 1.1.5: LSE $Ax = b$

A good example for explaining the backward error is a linear system of equations of the form $Ax = b$.

Suppose we want to solve for x in $Ax = b$ and we have obtained an approximate solution x_{app} . We can define the corresponding right-hand side

$$b_{app} := Ax_{app} .$$

Then, with x_{ex} being the exact solution to $Ax = b$, we can introduce two error terms

$$\begin{aligned} \text{Forward} & : & x_{ex} - x_{app} , \\ \text{Backward} & : & b_{ex} - b_{app} . \end{aligned}$$

While in practice, x_{ex} and hence the forward error is unknown, the backward error can be easily computed.

In practice: Stop when backward error $Ax_{app} - b$ is small.

However: Small backward error \nRightarrow small forward error.

In the case of a linear system of equations, the condition number of the system matrix indicates the reliability of the backward error: If the condition number (see Definition 1.1.5 below) is small, then a small backward error guarantees a small forward error. But for large condition numbers, the forward error can be large even if the backward error is very small.

Definition 1.1.5 (Condition number of a matrix). Suppose $\mathbf{A} \in \mathbb{R}^{n,n}$ is a matrix with largest singular value σ_{\max} and smallest singular value σ_{\min} . Then the condition number can be calculated as follows:

$$\text{cond}(\mathbf{A}) = \frac{\sigma_{\max}}{\sigma_{\min}}.$$

1.2 Fundamentals

1.2.1 Notations

We now introduce some basic notation used throughout this course. Notations in textbooks may be different, beware!

 Notation for generic field of numbers: \mathbb{K}

In this course, \mathbb{K} will designate either \mathbb{R} (real numbers) or \mathbb{C} (complex numbers); complex arithmetic [2, Sect. 2.5] plays a crucial role in many applications, for instance in signal processing.

Vector notations

- Vectors are n -tuples ($n \in \mathbb{N}$) with components in \mathbb{K} .

vector = one-dimensional array (of real/complex numbers)

 Notation for vectors: small **bold** symbols: $\mathbf{a}, \mathbf{b}, \dots, \mathbf{x}, \mathbf{y}, \mathbf{z}$

- Default in this lecture: vectors are *column vectors*

$\begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} \in \mathbb{K}^n$	$\begin{bmatrix} x_1 & \cdots & x_n \end{bmatrix} \in \mathbb{K}^{1,n}$
column vector	row vector

$\mathbb{K}^n \triangleq$ vector space of *column vectors* with n components in \mathbb{K} .

Remark. Unless stated otherwise, in mathematical formulas vector components are indexed starting from 1.

- Transposing:
$$\begin{cases} \text{column vector} & \mapsto \text{row vector} \\ \text{row vector} & \mapsto \text{column vector} \end{cases}$$

$$\begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}^\top = [x_1 \cdots x_n] \quad , \quad [x_1 \cdots x_n]^\top = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}$$

✎ Notation for row vectors: $\mathbf{x}^\top, \mathbf{y}^\top, \mathbf{z}^\top$

- Addressing vector components:

✎ Notation:
$$\begin{aligned} \mathbf{x} = [x_1 \dots x_n]^\top &\rightarrow x_i, \quad i = 1, \dots, n \\ \mathbf{x} \in \mathbb{K}^n &\rightarrow (\mathbf{x})_i, \quad i = 1, \dots, n \end{aligned}$$

- Selecting sub-vectors:

✎ Notation:
$$\mathbf{x} = [x_1 \dots x_n]^\top \rightarrow (\mathbf{x})_{k:l} = (x_k, \dots, x_l)^\top, \quad 1 \leq k \leq l \leq n$$

- j -th unit vector: $\mathbf{e}_j = [0, \dots, 1, \dots, 0]^\top$, $(\mathbf{e}_j)_i = \delta_{ij}$, $i, j = 1, \dots, n$.

✎ Notation: *Kronecker symbol* $\delta_{ij} := 1$, if $i = j$; $\delta_{ij} := 0$, if $i \neq j$.

Notations and notions for matrices

- Matrices = two-dimensional arrays of real/complex numbers

$$\mathbf{A} := \begin{bmatrix} a_{11} & \dots & a_{1n} \\ \vdots & & \vdots \\ a_{m1} & \dots & a_{mn} \end{bmatrix} \in \mathbb{K}^{m,n}, \quad m, n \in \mathbb{N}.$$

vector space of $m \times n$ -matrices: ($m \triangleq$ number of *rows*, $n \triangleq$ number of *columns*)

✎ Notation: **bold CAPITAL** roman letters, e.g., $\mathbf{A}, \mathbf{S}, \mathbf{Y}$

$$\mathbb{K}^{n,1} \leftrightarrow \text{column vectors}, \quad \mathbb{K}^{1,n} \leftrightarrow \text{row vectors}$$

- Writing a matrix as a tuple of its columns or rows

$$\mathbf{c}_i \in \mathbb{K}^m, \quad i = 1, \dots, n \rightarrow \mathbf{A} = [\mathbf{c}_1, \mathbf{c}_2, \dots, \mathbf{c}_n] \in \mathbb{K}^{m,n},$$

$$\mathbf{r}_i \in \mathbb{K}^n, \quad i = 1, \dots, m \quad \rightarrow \quad \mathbf{A} = \begin{bmatrix} \mathbf{r}_1^\top \\ \vdots \\ \mathbf{r}_m^\top \end{bmatrix} \in \mathbb{K}^{m,n}.$$

- Addressing matrix entries & sub-matrices:

$$\mathbf{A} := \begin{bmatrix} a_{11} & \dots & a_{1n} \\ \vdots & & \vdots \\ a_{m1} & \dots & a_{mn} \end{bmatrix}$$


$\rightarrow \text{entry } (\mathbf{A})_{i,j} = a_{ij},$
 $\rightarrow i\text{-th row: } a_{i,:} = (\mathbf{A})_{i,:},$
 $\rightarrow j\text{-th column: } a_{:,j} = (\mathbf{A})_{:,j},$
 $\rightarrow \text{matrix block } (a_{ij})_{\substack{i=k,\dots,l \\ j=r,\dots,s}} = (\mathbf{A})_{k:l,r:s}, \quad \begin{matrix} 1 \leq k \leq l \leq m, \\ 1 \leq r \leq s \leq n. \end{matrix}$
 (sub-matrix)

- Transposed matrix:

$$\mathbf{A}^\top = \begin{bmatrix} a_{11} & \dots & a_{1n} \\ \vdots & & \vdots \\ a_{m1} & \dots & a_{mn} \end{bmatrix}^\top := \begin{bmatrix} a_{11} & \dots & a_{m1} \\ \vdots & & \vdots \\ a_{1n} & \dots & a_{nm} \end{bmatrix} \in \mathbb{K}^{n,m}.$$

- Adjoint matrix (Hermitian transposed):

$$\mathbf{A}^H := \begin{bmatrix} a_{11} & \dots & a_{1n} \\ \vdots & & \vdots \\ a_{m1} & \dots & a_{mn} \end{bmatrix}^H := \begin{bmatrix} \overline{a_{11}} & \dots & \overline{a_{m1}} \\ \vdots & & \vdots \\ \overline{a_{1n}} & \dots & \overline{a_{nm}} \end{bmatrix} \in \mathbb{K}^{n,m}.$$

 Notation: $\overline{a_{ij}} = \Re(a_{ij}) - i\Im(a_{ij})$ denotes the complex conjugate of a_{ij} .

1.2.2 Classes of matrices

Most matrices occurring in mathematical modelling have a special structure. This section presents a few of these. More will come up throughout the remainder of this chapter; see also [1, Sect. 4.3].

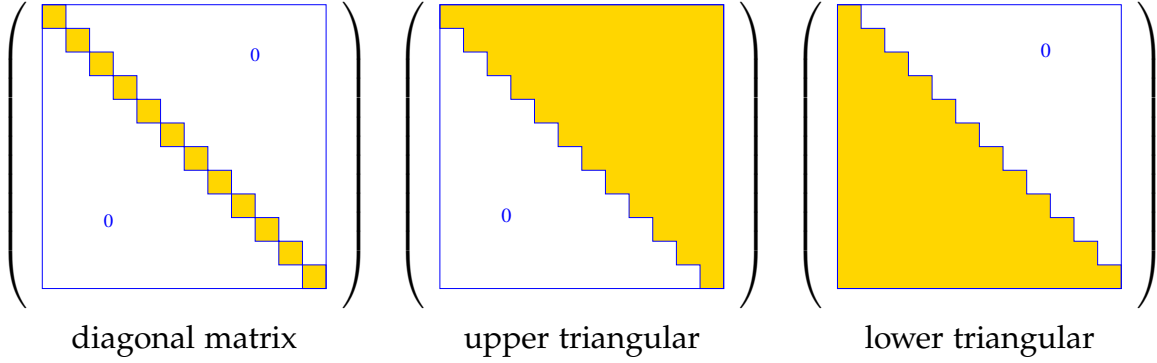
Diagonal and triangular matrices

A little terminology to quickly refer to matrices whose non-zero entries occupy special locations:

Definition 1.2.1 (Types of matrices). A matrix $\mathbf{A} = (a_{ij}) \in \mathbb{R}^{m,n}$ is

- diagonal matrix, if $a_{ij} = 0$ for $i \neq j$,
- upper triangular matrix if $a_{ij} = 0$ for $i > j$,
- lower triangular matrix if $a_{ij} = 0$ for $i < j$.

A triangular matrix is normalized, if $a_{ii} = 1, i = 1, \dots, \min\{m, n\}$.



Symmetric matrices

Definition 1.2.2 (Hermitian/symmetric matrices). A matrix $\mathbf{M} \in \mathbb{K}^{n,n}$, $n \in \mathbb{N}$, is *Hermitian*, if $\mathbf{M}^H = \mathbf{M}$. If $\mathbb{K} = \mathbb{R}$, the matrix is called *symmetric*.

Definition 1.2.3 (Symmetric positive definite matrices). A matrix $\mathbf{M} \in \mathbb{K}^{n,n}$, $n \in \mathbb{N}$, is symmetric (Hermitian) positive definite (s.p.d.), if

$$\mathbf{M} = \mathbf{M}^H \quad \text{and} \quad \forall \mathbf{x} \in \mathbb{K}^n: \quad \mathbf{x}^H \mathbf{M} \mathbf{x} > 0 \quad \Leftrightarrow \quad \mathbf{x} \neq \mathbf{0}.$$

If $\mathbf{x}^H \mathbf{M} \mathbf{x} \geq 0$ for all $\mathbf{x} \in \mathbb{K}^n$, we say that \mathbf{M} is positive semi-definite.

Lemma 1.2.1 (Necessary conditions for s.p.d.). For a symmetric/Hermitian positive definite matrix $\mathbf{M} = \mathbf{M}^H \in \mathbb{K}^{n,n}$ the following holds true:

1. $m_{ii} > 0, i = 1, \dots, n$,
2. $m_{ii}m_{jj} - |m_{ij}|^2 > 0 \quad \forall 1 \leq i < j \leq n$,
3. all eigenvalues of \mathbf{M} are positive. (\leftarrow also sufficient for symmetric/Hermitian \mathbf{M})

1.3 Software and Libraries

Whenever algorithms involve matrices and vectors (in the sense of linear algebra) it is advisable to rely on suitable code libraries or numerical programming environments.

1.3.1 Eigen

Currently, the most widely used programming language for the development of new simulation software in scientific and industrial high-performance computing is C++. In this course we are going to use and discuss EIGEN as an example of a C++ library for numerical linear algebra (“embedded” domain specific language: DSL). EIGEN is a header-only C++ template library designed to enable easy, natural and efficient numerical linear algebra: it provides data structures and a wide range of operations for matrices and vectors, see below. EIGEN also implements many more fundamental algorithms (see the documentation page or the discussion below).

EIGEN relies on expression templates to allow the efficient evaluation of complex expressions involving matrices and vectors. Refer to the example given in the EIGEN documentation for details.

Compilation of codes using Eigen

Compiling and linking on Mac OS X 10.10:

```
clang -D_HAS_CPP0X -std=c++11 -Wall -g \
    -Wno-deprecated-register -DEIGEN3_ACTIVATED \
    -I/opt/local/include -I/usr/local/include/eigen3 \
    -o main.cpp.o -c main.cpp
/usr/bin/c++ -std=c++11 -Wall -g -Wno-deprecated-register \
    -DEIGEN3_ACTIVATED -Wl,-search_paths_first \
    -Wl,-headerpad_max_install_names main.cpp.o \
    -o executable /opt/local/lib/libboost_program_options-mt.dylib
```

Of course, different compilers may be used on different platforms. In all cases, basic EIGEN functionality can be used without linking with a special library. Usually the generation of such elaborate calls of the compiler is left to a build system like CMAKE.

“EIGEN Cheat Sheet” (quick reference relating to MATLAB commands) can be found here: <http://eigen.tuxfamily.org/dox/AsciiQuickReference.txt>

Matrix and vector data types in Eigen

A generic matrix data type is given by the templated class

```
1  Matrix<typename Scalar ,
2      int RowsAtCompileTime, int ColsAtCompileTime>
```

Here **Scalar** is the underlying scalar type of the matrix entries, which must support the usual operations ‘+’, ‘-’, ‘*’, ‘/’, and ‘+=’, ‘*=’, ‘/=’, etc. Usually the scalar type will be either double, float, or complex<>. The cardinal template arguments RowsAtCompileTime and

ColsAtCompileTime can pass a fixed size of the matrix, if it is known at compile time. There is a specialization selected by the template argument `Eigen::Dynamic` supporting variable size “dynamic” matrices.

Code Snippet 1.8: Vector types and their use in EIGEN

```

1  #include <Eigen/Dense >
2
3  template<typename Scalar>
4  void eigenTypeDemo(unsigned int dim)
5  {
6      // General dynamic (variable size) matrices
7      using dynMat_t = Eigen::Matrix<Scalar, Eigen::Dynamic, Eigen::Dynamic>;
8      // Dynamic (variable size) column vectors
9      using dynColVec_t = Eigen::Matrix<Scalar, Eigen::Dynamic, 1>;
10     // Dynamic (variable size) row vectors
11     using dynRowVec_t = Eigen::Matrix<Scalar, 1, Eigen::Dynamic>;
12     using index_t = typename dynMat_t::Index;
13     using entry_t = typename dynMat_t::Scalar;
14
15     // Declare vectors of size 'dim'; not yet initialized
16     dynColVec_t colvec(dim);
17     dynRowVec_t rowvec(dim);
18     // Initialisation through component access
19     for(index_t i=0; i< colvec.size(); ++i) colvec(i) = (Scalar)i;
20     for(index_t i=0; i< rowvec.size(); ++i) rowvec(i) = (Scalar)1/(i+1);
21     colvec[0] = (Scalar)3.14; rowvec[dim-1] = (Scalar)2.718; //
22     // Form tensor product, a matrix
23     dynMat_t vecprod = colvec*rowvec; //
24     const int nrows = vecprod.rows();
25     const int ncols = vecprod.cols();
26 }

```

Note that in line 23 we could have relied on automatic type deduction via `auto vecprod = ...`. However, it is often safer to forgo this option and specify the type directly.

The following convenience data types are provided by EIGEN, see documentation:

- **MatrixXd**: generic variable size matrix with double precision entries,
- **VectorXd**, **RowVectorXd**: dynamic column and row vectors
(= dynamic matrices with one dimension equal to 1),
- **MatrixNd** with $N = 2, 3, 4$ for small fixed size square $N \times N$ -matrices (type double),
- **VectorNd** with $N = 2, 3, 4$ for small column vectors with fixed length N .

The `d` in the type name may be replaced with `i` (for `int`), `f` (for `float`), and `cd` (for complex<double>) to select another basic scalar type.

All matrix types feature the methods `cols()`, `rows()`, and `size()` outputting the number of columns, rows, and total number of entries.

Access to individual matrix entries and vector components, both as Rvalue and Lvalue, is possible through the `()`-operator taking two arguments of type `index_t`. If only one argument is supplied, the matrix is accessed as a linear array according to its memory layout. For vectors, that is, matrices where one dimension is fixed to 1, the `[]`-operator can replace `()` with one argument, see line 21 of Code Snippet 1.8.

Initialization of *dense* matrices in Eigen

The entry access operator (`int i, int j`) allows the most direct setting of matrix entries; there is hardly any runtime penalty. Of course, in EIGEN, dedicated functions take care of the initialization of special matrices:

```
1 Eigen::MatrixX<double> I = Eigen::MatrixX<double>::Identity(n,n);
2 Eigen::MatrixX<double> O = Eigen::MatrixX<double>::Zero(n,m);
3 Eigen::MatrixX<double> D = d_vector.asDiagonal();
```

Code Snippet 1.9: Initializing special matrices in EIGEN

```
1 #include <Eigen/Dense>
2 // Just allocate space for matrix, no initialisation
3 Eigen::MatrixX<double> A(rows,cols);
4 // Zero matrix. Similar to MATLAB command zeros(rows,cols);
5 Eigen::MatrixX<double> B = Eigen::MatrixX<double>::Zero(rows,cols);
6 // Ones matrix. Similar to MATLAB command ones(rows,cols);
7 Eigen::MatrixX<double> C = Eigen::MatrixX<double>::Ones(rows,cols);
8 // Matrix with all entries same as value.
9 Eigen::MatrixX<double> D = Eigen::MatrixX<double>::Constant(rows,cols,value);
10 // Random matrix, entries uniformly distributed in [0,1]
11 Eigen::MatrixX<double> E = Eigen::MatrixX<double>::Random(rows,cols);
12 // (Generalized) identity matrix, 1 on main diagonal
13 Eigen::MatrixX<double> I = Eigen::MatrixX<double>::Identity(rows,cols);
14 std::cout << "size of A = (" << A.rows() << ", " << A.cols() << ") " << std::endl;
```

A versatile way to initialize a matrix relies on a combination of the operators `<<` and `,`, which allows the construction of a matrix from blocks:

```
1 MatrixX<double> mat3(6,6);
2 mat3 <<
3     MatrixX<double>::Constant(4,2,1.5), // top row, first block
4     MatrixX<double>::Constant(4,3,3.5), // top row, second block
5     MatrixX<double>::Constant(4,1,7.5), // top row, third block
6     MatrixX<double>::Constant(2,4,2.5), // bottom row, left block
7     MatrixX<double>::Constant(2,2,4.5); // bottom row, right block
```

The matrix is filled top to bottom left to right, block dimensions have to match (like in MATLAB). Thus, the above code will generate the following matrix:

$$\text{mat3} = \begin{pmatrix} 1.5 & 1.5 & 3.5 & 3.5 & 3.5 & 7.5 \\ 1.5 & 1.5 & 3.5 & 3.5 & 3.5 & 7.5 \\ 1.5 & 1.5 & 3.5 & 3.5 & 3.5 & 7.5 \\ 1.5 & 1.5 & 3.5 & 3.5 & 3.5 & 7.5 \\ 2.5 & 2.5 & 2.5 & 2.5 & 4.5 & 4.5 \\ 2.5 & 2.5 & 2.5 & 2.5 & 4.5 & 4.5 \end{pmatrix}$$

Access to submatrices in Eigen (see [Documentation](#))

The method `block(int i, int j, int p, int q)` returns a reference to the submatrix with upper left corner at position (i, j) and size $p \times q$.

The methods `row(int i)` and `col(int j)` provide a reference to the corresponding row and column of the matrix. Even more specialised access methods are

`topLeftCorner(p, q), bottomLeftCorner(p, q),`
`topRightCorner(p, q), bottomRightCorner(p, q),`
`topRows(q), bottomRows(q),`
`leftCols(p), and rightCols(q),`

with obvious purposes.

Code Snippet 1.10: Demonstration code for access to matrix blocks in EIGEN → GITLAB

```

19 template<typename MatType>
20 void blockAccess(Eigen::MatrixBase<MatType> &M)
21 {
22     using index_t = typename Eigen::MatrixBase<MatType>::Index;
23     using entry_t = typename Eigen::MatrixBase<MatType>::Scalar;
24     const index_t nrows(M.rows()); // No. of rows
25     const index_t ncols(M.cols()); // No. of columns
26
27     cout << "Matrix M = " << endl << M << endl; // Print matrix
28     // Block size half the size of the matrix
29     index_t p = nrows/2, q = ncols/2;
30     // Output submatrix with left upper entry at position (i,i)
31     for(index_t i=0; i < min(p,q); i++)
32         cout << "Block (" << i << ', ' << i << ', ' << p << ', ' << q
33             << ") = " << M.block(i, i, p, q) << endl;
34     // l-value access: modify sub-matrix by adding a constant
35     M.block(1, 1, p, q) += Eigen::MatrixBase<MatType>::Constant(p, q, 1.0);
36     cout << "M = " << endl << M << endl;
37     // r-value access: extract sub-matrix
38     MatrixXd B = M.block(1, 1, p, q);

```

```

39  cout << "Isolated modified block = " << endl << B << endl;
40  // Special sub-matrices
41  cout << p << " top rows of m = " << M.topRows(p) << endl;
42  cout << p << " bottom rows of m = " << M.bottomRows(p) << endl;
43  cout << q << " left cols of m = " << M.leftCols(q) << endl;
44  cout << q << " right cols of m = " << M.rightCols(p) << endl;
45  // r-value access to upper triangular part
46  const MatrixXd T = M.template triangularView<Upper>(); //
47  cout << "Upper triangular part = " << endl << T << endl;
48  // l-value access to upper triangular part
49  M.template triangularView<Lower>() *= -1.5; //
50  cout << "Matrix M = " << endl << M << endl;
51  }
52  /*

```

EIGEN offers views for access to triangular parts of a matrix, see line 46 and line 49, according to

`M.triangularView<XX>()`

where XX can stand for one of the following: Upper, Lower, StrictlyUpper, StrictlyLower, UnitUpper, UnitLower, see documentation.

For column and row vectors, references to sub-vectors can be obtained by the methods `head(int length)`, `tail(int length)`, and `segment(int pos, int length)`.

Note: Unless the preprocessor switch NDEBUG is set, EIGEN performs range checks on all indices.

Component-wise operations in Eigen

EIGEN uses the Array concept to furnish entry-wise operations on matrices. An EIGEN-Array contains the same data as a matrix, supports the same methods for initialisation and access, but replaces the operators of matrix arithmetic with entry-wise actions. Matrices and arrays can be converted into each other by the `array()` and `matrix()` methods, see documentation for details.

*Code Snippet 1.11: Using **Array** in EIGEN → GITLAB*

```

19  void matArray(int nrows, int ncols) {
20      Eigen::MatrixXd m1(nrows, ncols), m2(nrows, ncols);
21      for(int i = 0; i < m1.rows(); i++)
22          for(int j = 0; j < m1.cols(); j++) {
23              m1(i, j) = (double)(i+1)/(j+1);
24              m2(i, j) = (double)(j+1)/(i+1);
25          }
26      // Entry-wise product, not a matrix product
27      Eigen::MatrixXd m3 = (m1.array() * m2.array()).matrix();
28      // Explicit entry-wise operations on matrices are possible

```

```

29 Eigen::MatrixXd m4(m1.cwiseProduct(m2));
30 // Entry-wise logarithm
31 cout << "Log(m1) = " << endl << log(m1.array()) << endl;
32 // Entry-wise boolean expression, true cases counted
33 cout << (m1.array() > 3).count() << " entries of m1 > 3" << endl;
34 }
35 /*

```

The application of a functor to all entries of a matrix can also be done via the `unaryExpr()` method of a matrix:

```

1 // Apply a lambda function to all entries of a matrix
2 auto fnct = [](double x) { return (x+1.0/x); };
3 cout << "f(m1) = " << endl << m1.unaryExpr(fnct) << endl;

```

1.3.2 (Dense) Matrix storage formats

All numerical libraries store the entries of a (generic = *dense*) matrix $\mathbf{A} \in \mathbb{K}^{m,n}$ in a linear array of length $m \cdot n$ (or longer). Accessing entries entails suitable index computations.

There are two natural options for *vectorisation* of a matrix: *row major* and *column major*.

$\mathbf{A} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$	Row major (C-arrays, bitmaps, Python):									
	A_arr	1	2	3	4	5	6	7	8	9
	Column major (Fortran, MATLAB, EIGEN):									
	A_arr	1	4	7	2	5	8	3	6	9

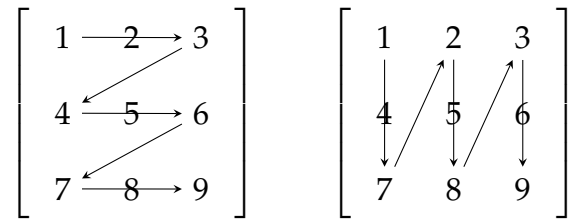
Access to entry $(\mathbf{A})_{ij}$ of $\mathbf{A} \in \mathbb{K}^{m,n}$,
 $i = 1, \dots, m, j = 1, \dots, n$:

row major:

$$(\mathbf{A})_{ij} \leftrightarrow \text{A_arr}(n*(i-1)+(j-1)),$$

column major:

$$(\mathbf{A})_{ij} \leftrightarrow \text{A_arr}(m*(j-1)+(i-1)).$$



row major

column major

Code Snippet 1.12: Single index access of matrix entries in EIGEN → GITLAB

```

19 void storageOrder(int nrows=6,int ncols=7)
20 {
21     cout << "Different matrix storage layouts in Eigen" << endl;
22     // Template parameter ColMajor selects column major data layout
23     Matrix<double,Dynamic,Dynamic,ColMajor> mcm(nrows,ncols);
24     // Template parameter RowMajor selects row major data layout
25     Matrix<double,Dynamic,Dynamic,RowMajor> mrm(nrows,ncols);
26     // Direct initialization; lazy option: use int as index type

```


1 Computing with Matrices and Vectors

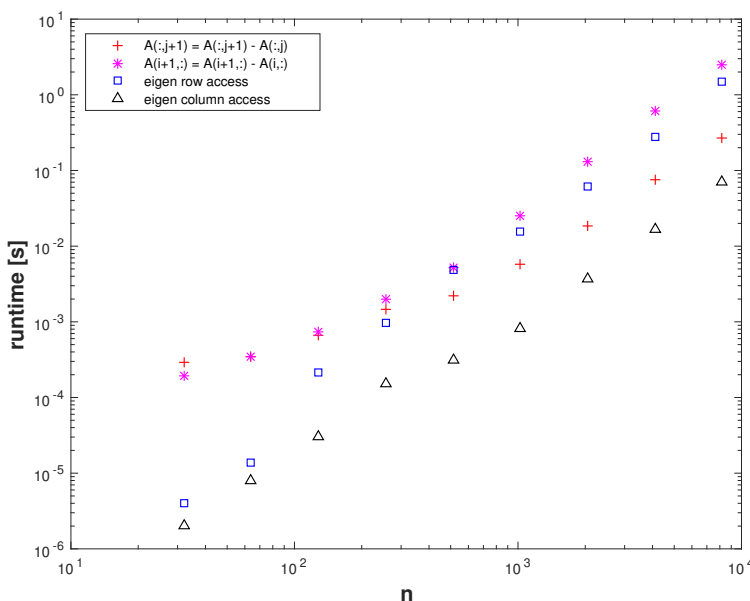
```
27 for (int l=1,i= 0; i< nrows; i++)
28     for (int j= 0; j< ncols; j++,l++)
29         mcm(i,j) = mmm(i,j) = 1;
30
31 cout << "Matrix mrm = " << endl << mmm << endl;
32 cout << "mcm linear = ";
33 for (int l=0;l < mcm.size(); l++) cout << mcm(l) << ',';
34 cout << endl;
35
36 cout << "mrm linear = ";
37 for (int l=0;l < mmm.size(); l++) cout << mmm(l) << ',';
38 cout << endl;
39 }
40 /*
```

The function call `storageOrder(3,3)`, cf. Code Snippet 1.12 yields the output

```
1 Different matrix storage layouts in Eigen
2 Matrix mmm =
3 1 2 3
4 4 5 6
5 7 8 9
6 mcm linear = 1,4,7,2,5,8,3,6,9,
7 mmm linear = 1,2,3,4,5,6,7,8,9,
```

Impact of matrix data access patterns on runtime

Modern CPU feature several levels of memories (registers, L1 cache, L2 cache, ..., main memory) of different latency, bandwidth, and size. Frequently accessing memory locations with widely different addresses results in many cache misses and will considerably slow down the CPU.



Platform:

- ubuntu 14.04 LTS
- i7-3517U CPU 1.90GHz x 4
- L1 32 KB, L2 256 KB, L3 4096 KB, Mem 8 GB
- gcc 4.8.4, -O3, -DNDEBUG

The compiler flags `-O3` and `-DNDEBUG` are essential. The C++ code would be significantly slower if the default compiler options were used!

For both MATLAB and EIGEN codes we observe a glaring discrepancy of CPU time required for accessing entries of a matrix in rowwise or columnwise fashion. This reflects the impact of features of the underlying hardware architecture, like cache size and memory bandwidth:

Interpretation of timings:

Since matrices in MATLAB are stored column major all the matrix elements in a column occupy contiguous memory locations, which will all reside in the cache together. Hence, column oriented access will mainly operate on data in the cache even for large matrices. Conversely, row oriented access addresses matrix entries that are stored in distant memory locations, which incurs frequent cash misses (*cache thrashing*).

The impact of hardware architecture on the performance of algorithms will **not** be taken into account in this course, because hardware features tend to be both intricate and ephemeral. However, for modern high performance computing it is essential to adapt implementations to the hardware on which the code is supposed to run.

1.4 Computational effort

The computational effort required by a numerical code amounts to the number of elementary operations (additions, subtractions, multiplications, divisions, square roots) executed in a run.

Note (Computational effort \approx runtime). The computational effort involved in a run of a numerical code is only loosely related to overall execution time on modern computers.

The runtime itself is influenced by the memory access patterns. Therefore, on today's computers a key bottleneck for fast execution is latency and bandwidth of memory. Thus, concepts like I/O-complexity might be more appropriate for gauging the efficiency of a code, because they take into account the pattern of memory access.

1.4.1 Asymptotic complexity

The concept of computational effort from before is still useful in a particular context:

Definition 1.4.1 (Asymptotic complexity). The asymptotic complexity of an algorithm characterises the worst-case dependence of its computational effort on one or more problem size parameter(s) when these tend to ∞ .

- *Problem size parameters* in numerical linear algebra usually are the lengths and dimensions of the vectors and matrices that an algorithm takes as inputs.
- *Worst case* indicates that the maximum effort over a set of admissible data is taken into account.

When dealing with asymptotic complexities, a mathematical formalism is useful:

Definition 1.4.2 (Landau symbol). We write $F(n) = \mathcal{O}(G(n))$ for two functions $F, G : \mathbb{N} \rightarrow \mathbb{R}$, if there exists a constant $C > 0$ and $n_* \in \mathbb{N}$ such that

$$F(n) \leq C G(n) \quad \forall n \geq n_* .$$

More generally, $F(n_1, \dots, n_k) = \mathcal{O}(G(n_1, \dots, n_k))$ for two functions $F, G : \mathbb{N}^k \rightarrow \mathbb{R}$ implies the existence of a constant $C > 0$ and a threshold value $n_* \in \mathbb{N}$ such that

$$F(n_1, \dots, n_k) \leq C G(n_1, \dots, n_k) \quad \forall n_1, \dots, n_k \in \mathbb{N}, \quad n_\ell \geq n_*, \ell = 1, \dots, k .$$

This is called the Landau- \mathcal{O} Notation and we can use it to describe the asymptotic complexity ("cost") of an algorithm depending on the problem size parameter n :

$$\text{Cost}(n) = \mathcal{O}(n^\alpha) \quad , \quad \alpha > 0 ,$$

or more strictly formulated:

$$\exists C > 0, n_0 \in \mathbb{N} : \quad \text{Cost}(n) \leq C n^\alpha \quad \forall n \geq n_0 ,$$

where an underlying implicit assumption of sharpness was made

$$\text{Cost}(n) \neq \mathcal{O}(n^\beta) \quad \forall \beta < \alpha .$$

Note (Sharpness of a complexity bound). Whenever the asymptotic complexity of an algorithm is stated as $\mathcal{O}(n^\alpha \log^\beta n \exp(\gamma n^\delta))$ with non-negative parameters $\alpha, \beta, \gamma, \delta \geq 0$ in terms of the problem size parameter n , we take for granted that choosing a smaller value for any of the parameters will no longer yield a valid (or provable) asymptotic bound.

Relevance of asymptotic complexity

It was already shown that computational effort and, thus, asymptotic complexity, of an algorithm for a concrete problem on a particular platform may not have much to do with the actual runtime (the blame goes to memory hierarchies, internal pipelining, vectorization, etc.).

Then, why do we pay so much attention to asymptotic complexity in this course?

To a certain extent, the asymptotic complexity allows to predict the *dependence of the runtime* of a particular implementation of an algorithm *on the problem size* (for large problems). For instance, an algorithm with asymptotic complexity $\mathcal{O}(n^2)$ is likely to take 4 times as much time when the problem size is doubled.

Example 1.4.1: Concluding polynomial complexity from runtime measurements

Available: “Measured runtimes” $t_i = t_i(n_i)$ for different values $n_1, n_2, \dots, n_N \in \mathbb{N}$, of the problem size parameter

Conjectured: “power law dependence”: $t_i \approx Cn_i^\alpha$ (also “algebraic dependence”), $\alpha \in \mathbb{R}$

This can be reformulated as:

$$t_i \simeq Cn_i^\alpha \implies \log(t_i) \simeq \log C + \alpha \log(n_i), \quad i = 1, \dots, N.$$

If the conjecture holds true, then the points (n_i, t_i) will approximately lie on a straight line with slope α in a doubly logarithmic plot.

1.4.2 Cost of basic operations

Performing basic linear algebra operations through simple (nested) loops, we arrive at the following obvious complexity bounds:

Operation	Description	Number of $\cdot, /$ operations	Number of $+, -$ operations	Asymp. Compl.
Dot product	$(\mathbf{x} \in \mathbb{R}^n, \mathbf{y} \in \mathbb{R}^n) \mapsto \mathbf{x}^H \mathbf{y}$	n	$n - 1$	$\mathcal{O}(n)$
Tensor product	$(\mathbf{x} \in \mathbb{R}^m, \mathbf{y} \in \mathbb{R}^n) \mapsto \mathbf{x} \mathbf{y}^H$	nm	0	$\mathcal{O}(mn)$
Matrix product [†]	$(\mathbf{A} \in \mathbb{R}^{m,n}, \mathbf{B} \in \mathbb{R}^{n,k}) \mapsto \mathbf{A} \mathbf{B}$	mnk	$mk(n - 1)$	$\mathcal{O}(mnk)$

1.5 Cancellation

In general, predicting the impact of roundoff errors on the result of a multi-stage computation is very difficult, if possible at all. However, there is a constellation that is particularly prone to dangerous amplification of roundoff and still can be detected easily.

[†]Due to three loop naive implementation

Computing the roots of a quadratic polynomial

The following simple EIGEN code computes the real roots of a quadratic polynomial $p(\xi) = \xi^2 + \alpha\xi + \beta$ by the discriminant formula

$$p(\xi_1) = p(\xi_2) = 0, \quad \xi_{1,2} = \frac{1}{2} \left(-\alpha \pm \sqrt{D} \right), \text{ if } D := \alpha^2 - 4\beta \geq 0.$$

Code Snippet 1.13: Discriminant formula for the real roots of $p(\xi) = \xi^2 + \alpha\xi + \beta \rightarrow \text{GITLAB}$

```

18  //! C++ function computing the zeros of a quadratic polynomial
19  //!  $\xi \rightarrow \xi^2 + \alpha\xi + \beta$  by means of the familiar discriminant
20  //! formula  $\xi_{1,2} = \frac{1}{2}(-\alpha \pm \sqrt{\alpha^2 - 4\beta})$ . However
21  //! this implementation is vulnerable to round-off! The zeros are
22  //! returned in a column vector
23  Vector2d zerosquadpol(double alpha, double beta){
24      Vector2d z;
25      double D = std::pow(alpha,2) - 4*beta; // discriminant
26      if(D < 0) throw "no real zeros";
27      else{
28          // The famous discriminant formula
29          double wD = std::sqrt(D);
30          z << (-alpha-wD)/2, (-alpha+wD)/2; //
31      }
32      return z;
33  }
34  /*

```

This formula is applied to the quadratic polynomial $p(\xi) = (\xi - \gamma)(\xi - \frac{1}{\gamma})$ after its coefficients α, β have been computed from γ , which will have introduced *small* relative roundoff errors (of size EPS).

Code Snippet 1.14: Testing the accuracy of computed roots of a quadratic polynomial $\rightarrow \text{GITLAB}$

```

25  //! Eigen Function for testing the computation of the zeros of a parabola
26  void compzeros() {
27      int n = 100;
28      MatrixXd res(n,4);
29      VectorXd gamma = VectorXd::LinSpaced(n, 2,992);
30      for(int i = 0; i < n; ++i){
31          double alpha = -(gamma(i) + 1./gamma(i));
32          double beta = 1.;
33          Vector2d z1 = zerosquadpol(alpha, beta);
34          Vector2d z2 = zerosquadpolstab(alpha, beta);
35          double ztrue = 1./gamma(i), z2true = gamma(i);
36          res(i,0) = gamma(i);
37          res(i,1) = std::abs((z1(0)-ztrue)/ztrue);
38          res(i,2) = std::abs((z2(0)-ztrue)/ztrue);
39          res(i,3) = std::abs((z1(1)-z2true)/z2true);
40      }

```

```

41 // Graphical output of relative error of roots computed by unstable implementation
42 mgl::Figure fig1;
43 fig1.setFontSize(3);
44 fig1.title("Roots of a parabola computed in an unstable manner");
45 fig1.plot(res.col(0),res.col(1), "+r").label("small root");
46 fig1.plot(res.col(0),res.col(3), "*b").label("large root");
47 fig1.xlabel("\\gamma");
48 fig1.ylabel("relative errors in \\xi_1, \\xi_2");
49 fig1.legend(0.05,0.95);
50 fig1.save("zqperrinstab");
51 // Graphical output of relative errors (comparison), small roots
52 mgl::Figure fig2;
53 fig2.title("Roundoff in the computation of zeros of a parabola");
54 fig2.plot(res.col(0), res.col(1), "+r").label("unstable");
55 fig2.plot(res.col(0), res.col(2), "*m").label("stable");
56 fig2.xlabel("\\gamma");
57 fig2.ylabel("relative errors in \\xi_1");
58 fig2.legend(0.05,0.95);
59 fig2.save("zqperr");
60 }
61 /*

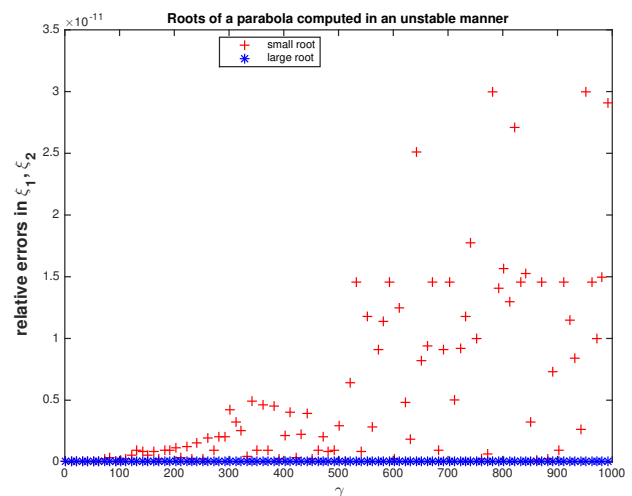
```

Observation:

Roundoff incurred during the computation of α and β leads to “inaccurate” roots.

For large γ the computed small root may be fairly inaccurate with regards to its *relative error*, which can be several orders of magnitude larger than machine precision EPS.

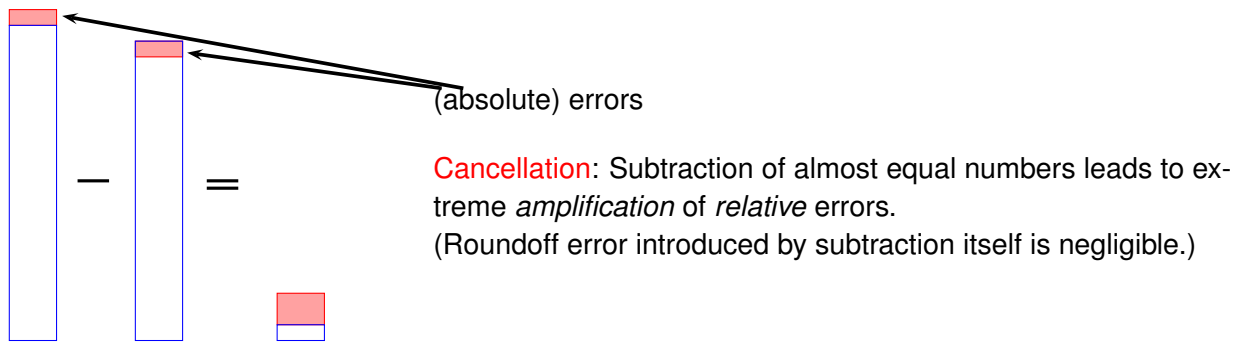
The large root always enjoys a small relative error about the size of EPS.



In order to understand why the small root is much more severely affected by roundoff, note that its computation involves the subtraction of two large numbers, if γ is large. This is the typical situation, in which *cancellation* occurs.

Visualisation of cancellation effect

We look at the *exact* subtraction of two almost equal positive numbers both of which have small relative errors (red boxes) with respect to some desired exact value (indicated by blue boxes). The result of the subtraction will be small, but the errors may add up during the subtraction, ultimately constituting a large fraction of the result.



Stable computation of the roots of a quadratic polynomial

If ξ_1 and ξ_2 are the two roots of the quadratic polynomial $p(\xi) = \xi^2 + \alpha\xi + \beta$, then $\xi_1 \cdot \xi_2 = \beta$ (Vieta's formula). Once, we have computed a root, we can obtain the other by simple division.

Approach:

- Depending on the sign of α compute “stable root” without cancellation.
- Compute other root from Vieta's formula (avoiding subtraction).

Code Snippet 1.15: Stable computation of real roots of a quadratic polynomial → GITLAB

```

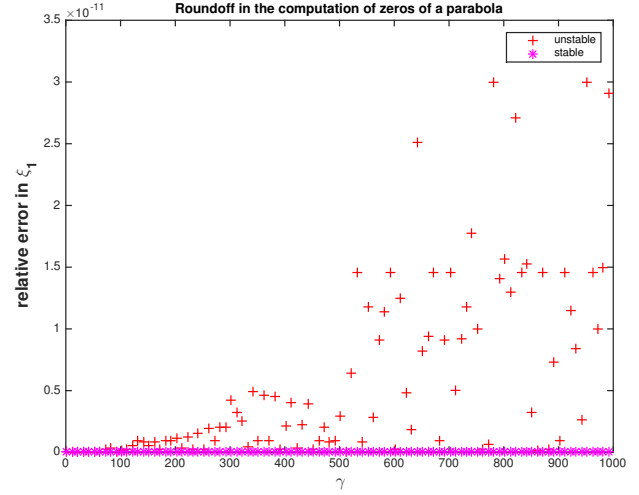
18 //! C++ function computing the zeros of a quadratic polynomial
19 //!  $\xi \rightarrow \xi^2 + \alpha\xi + \beta$  by means of the familiar discriminant
20 //! formula  $\xi_{1,2} = \frac{1}{2}(-\alpha \pm \sqrt{\alpha^2 - 4\beta})$ .
21 //! This is a stable implementation based on Vieta's theorem.
22 //! The zeros are returned in a column vector
23 VectorXd zerosquadpolstab(double alpha, double beta){
24     Vector2d z(2);
25     double D = std::pow(alpha,2) - 4*beta; // discriminant
26     if(D < 0) throw "no real zeros";
27     else{
28         double wD = std::sqrt(D);
29         // Use discriminant formula only for zero far away from 0
30         // in order to avoid cancellation. For the other zero
31         // use Vieta's formula.
32         if(alpha >= 0){
33             double t = 0.5*(-alpha-wD); //
34             z << t, beta/t;
35         }
36         else{
37             double t = 0.5*(-alpha+wD); //
38             z << beta/t, t;
39         }
40     }
41     return z;
42 }

```

Note. Invariably, we add numbers with the same sign in line 33 and line 37.

Numerical experiment based on the driver code (Code Snippet 1.14).

Observation: The new code can also compute the small root of the polynomial $p(\xi) = (\xi - \gamma)(\xi - \frac{1}{\gamma})$ (expanded in monomials) with a relative error $\approx \text{EPS}$.



Cancellation when evaluating difference quotients

From analysis we know that the derivative of a differentiable function $f : I \subset \mathbb{R} \rightarrow \mathbb{R}$ at a point $x \in I$ is the limit of a *difference quotient*

$$f'(x) = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h}.$$

This suggests the following approximation of the derivative by a difference quotient with small but finite $h > 0$:

$$f'(x) \approx \frac{f(x+h) - f(x)}{h} \quad \text{for } |h| \ll 1.$$

Results from analysis tell us that the *approximation error* should tend to zero for $h \rightarrow 0$. More precise quantitative information is provided by the Taylor formula for a twice continuously differentiable function

$$f(x+h) = f(x) + f'(x)h + \frac{1}{2}f''(\xi)h^2 \quad \text{for some } \xi = \xi(x,h) \in [\min\{x, x+h\}, \max\{x, x+h\}],$$

from which we infer

$$\frac{f(x+h) - f(x)}{h} - f'(x) = \frac{1}{2}hf''(\xi) \quad \text{for some } \xi = \xi(x,h) \in [\min\{x, x+h\}, \max\{x, x+h\}]. \quad (1.1)$$

We investigate the approximation of the derivative by difference quotients for $f = \exp$, $x = 0$, and different values of $h > 0$:

Code Snippet 1.16: Difference quotient approximation of the derivative of \exp → GITLAB

18 `#!/ Difference quotient approximation`

1 Computing with Matrices and Vectors

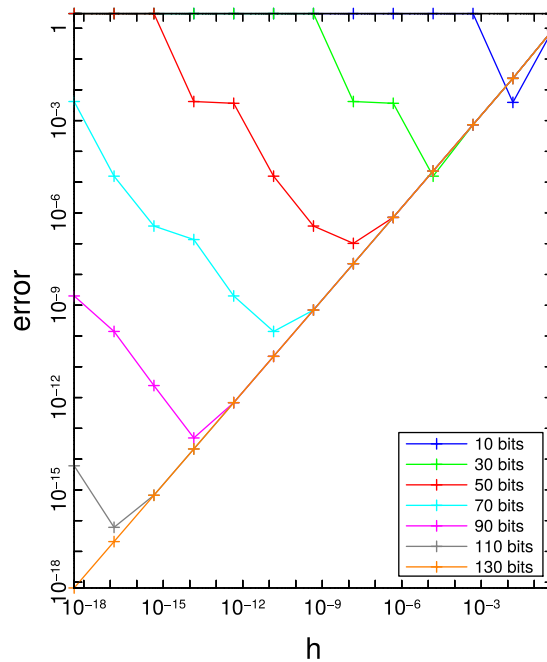
```

19 //! of the derivative of exp
20 void diffq() {
21     double h = 0.1, x = 0.0;
22     for(int i = 1; i <= 16; ++i){
23         double df = (exp(x+h)-exp(x))/h;
24         cout << setprecision(14) << fixed;
25         cout << setw(5) << -i
26         << setw(20) << df-1 << endl;
27         h /= 10;
28     }
29 }
30 /*

```

This gives the following measured relative errors:

$\log_{10}(h)$	relative error
-1	0.05170918075648
-2	0.00501670841679
-3	0.00050016670838
-4	0.00005000166714
-5	0.00000500000696
-6	0.00000049996218
-7	0.00000004943368
-8	-0.00000000607747
-9	0.00000008274037
-10	0.00000008274037
-11	0.00000008274037
-12	0.00008890058234
-13	-0.00079927783736
-14	-0.00079927783736
-15	0.11022302462516
-16	-1.00000000000000



We observe an initial decrease of the relative approximation error followed by a steep increase when h drops below 10^{-8} . The error plot on the right-hand side in the above figure confirms that the observed errors are really due to roundoff errors. For these numerical results a variable precision floating point module of EIGEN, the MPFR++ Support module was used.

Obvious culprit: *cancellation* when computing the numerator of the difference quotient for small $|h|$ leads to a strong amplification of inevitable errors introduced by the evaluation of

the transcendent exponential function.

We witness the competition of two opposite effects: Smaller h results in a better approximation of the derivative by the difference quotient, but the impact of cancellation is stronger for a smaller $|h|$.

$$\left. \begin{array}{l} \text{Approximation error } f'(x) - \frac{f(x+h) - f(x)}{h} \rightarrow 0 \\ \text{Impact of roundoff } \rightarrow \infty \end{array} \right\} \text{ as } h \rightarrow 0 .$$

In order to provide a rigorous underpinning for our conjecture, in this example we embark on our first roundoff error analysis merely based on Assumption 1.1.1: As in the computational example above we study the approximation of $f'(x) = e^x$ for $f = \exp$, $x \in \mathbb{R}$ by the difference quotient dq .

$$\begin{aligned} \text{dq} &= \frac{e^{x+h}(1+\delta_1) - e^x(1+\delta_2)}{h} \\ &= e^x \left(\frac{e^h - 1}{h} + \frac{\delta_1 e^h - \delta_2}{h} \right) \end{aligned} \quad \begin{array}{l} \text{Correction factors take into account roundoff:} \\ \text{(Assumption 1.1.1)} \\ |\delta_1|, |\delta_2| \leq \text{EPS} . \end{array}$$

Choose h such that the relative error $\left| \frac{\text{dq} - e^x}{e^x} \right|$ is minimized.

$$\begin{aligned} \frac{\text{dq} - e^x}{e^x} &= \left(\frac{e^h - 1}{h} - 1 \right) + \left(\frac{\delta_1 e^h - \delta_2}{h} \right) \\ \left| \frac{\text{dq} - e^x}{e^x} \right| &\leq \left| \left(\frac{e^h - 1}{h} - 1 \right) \right| + \left| \left(\frac{\delta_1 e^h - \delta_2}{h} \right) \right| \\ \left| \frac{\text{dq} - e^x}{e^x} \right| &\lesssim \frac{h}{2} + \frac{2\text{EPS}}{h} \end{aligned}$$

Use Equation (1.1) with $x = 0$ to get:

$$\frac{e^h - 1}{h} - 1 = \frac{he^\xi}{2} \text{ for some } \xi \in [0, h]$$

Minimising this expression in h yields

$$\begin{aligned} \frac{1}{2} - \frac{2\text{EPS}}{h^2} &\approx 0 \\ \implies h &\approx 2\sqrt{\text{EPS}} \end{aligned}$$

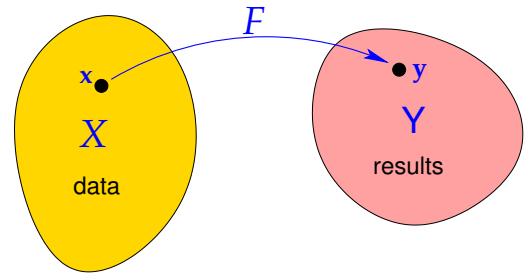
In double precision: $2\sqrt{\text{EPS}} = 1.483239697419 \cdot 10^{-8}$

1.6 Numerical stability

We have seen that a particular *problem* can be tackled by different algorithms, which produce different results due to roundoff errors. This section will clarify what distinguishes a “good” algorithm from a rather abstract point of view.

A mathematical notion of *problem*:

- Data space X , usually $X \subset \mathbb{R}^n$
- Result space Y , usually $Y \subset \mathbb{R}^m$
- Mapping (problem function) $F : X \mapsto Y$



A problem is a well defined *function* that assigns to each datum a result.

Note. In this course, both the data space X and the result space Y will always be subsets of finite dimensional vector spaces.

Example 1.6.1: Matrix-vector multiplication

We consider the problem of computing the product \mathbf{Ax} for a given matrix $\mathbf{A} \in \mathbb{K}^{m,n}$ and a given vector $\mathbf{x} \in \mathbb{K}^n$.

- Data space $X = \mathbb{K}^{m,n} \times \mathbb{K}^n$ (input is a matrix and a vector)
- Result space $Y = \mathbb{R}^m$ (space of column vectors)
- Problem function $F : X \rightarrow Y, F(\mathbf{A}, \mathbf{x}) := \mathbf{Ax}$

Numerical algorithm

The only way to describe an algorithm is through a concrete code function written in, for instance, MATLAB or C++. This function defines another mapping $\tilde{F} : X \rightarrow Y$ on the data space of the problem. Since the input data to this function can only be represented in the set \mathbb{M} of machine numbers, it is implicitly understood in the definition of \tilde{F} that the input data is subject to *rounding* before passing it to the code function.

Problem	Algorithm
$F : X \subset \mathbb{R}^n \rightarrow Y \subset \mathbb{R}^m$	$\tilde{F} : X \subset \mathbb{M}^n \rightarrow \tilde{Y} \subset \mathbb{M}^m$

Stable algorithm

- We study a problem $F : X \rightarrow Y$ on data space X into result space Y .
- We assume that both X and Y are equipped with norms $\|\cdot\|_X$ and $\|\cdot\|_Y$, respectively.
- We consider a concrete algorithm $\tilde{F} : X \rightarrow Y$ according to Definition 1.6.1.

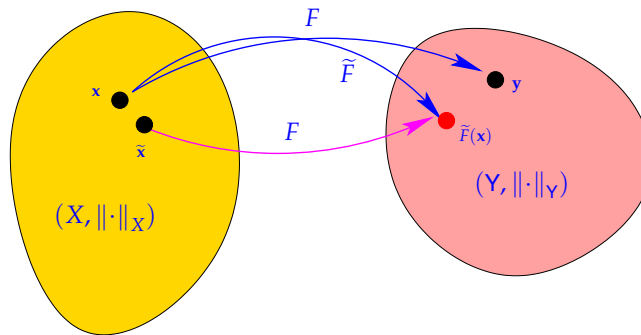
We write $w(\mathbf{x})$, $\mathbf{x} \in X$, for the *computational effort* required by the algorithm for input \mathbf{x} .

Definition 1.6.1 (Stable algorithm). An algorithm \tilde{F} for solving a problem $F : X \mapsto Y$ is *numerically stable* if for all $\mathbf{x} \in X$ its result $\tilde{F}(\mathbf{x})$ (possibly affected by roundoff) is the exact result for “slightly perturbed” data:

$$\exists C \approx 1: \quad \forall \mathbf{x} \in X: \quad \exists \tilde{\mathbf{x}} \in X: \quad \|\mathbf{x} - \tilde{\mathbf{x}}\|_X \leq Cw(\mathbf{x}) \text{ EPS } \|\mathbf{x}\|_X \quad \wedge \quad \tilde{F}(\mathbf{x}) = F(\tilde{\mathbf{x}}) .$$

Here EPS should be read as machine precision according to Assumption 1.1.1.

Illustration of Definition 1.6.1 (\mathbf{y} : exact result for exact data \mathbf{x}):



Terminology: Definition 1.6.1 introduces stability in the sense of *backward error analysis*

Note. Sloppily speaking, the impact of roundoff[‡] on a *stable algorithm* is of the same order of magnitude as the effect of the inevitable perturbations due to rounding the input data. In other words, backward stability guarantees stability with respect to roundoff errors. However, as we have seen in Example 1.6.1, this does not guarantee stability with respect to small input errors (for this we need forward stability).

Note on matrix norms

Norms provide tools for measuring errors. Recall from linear algebra and calculus [3, Sect. 4.3], [4, Sect. 6.1]:

[‡]In some cases the definition of \tilde{F} will also involve some approximations. Then the above statement also includes approximation errors.

1 Computing with Matrices and Vectors

Definition 1.6.2 (Matrix norm). Given vector norms $\|\cdot\|_1$ and $\|\cdot\|_2$ on \mathbb{K}^n and \mathbb{K}^m , respectively, the associated *matrix norm* is defined by

$$\mathbf{M} \in \mathbb{R}^{m,n}: \quad \|\mathbf{M}\| := \sup_{\mathbf{x} \in \mathbb{R}^n \setminus \{0\}} \frac{\|\mathbf{M}\mathbf{x}\|_2}{\|\mathbf{x}\|_1} .$$

By virtue of definition, the matrix norms enjoy an important property, they are sub-multiplicative:

$$\forall \mathbf{A} \in \mathbb{K}^{m,n}, \mathbf{B} \in \mathbb{K}^{n,k}: \quad \|\mathbf{AB}\| \leq \|\mathbf{A}\| \|\mathbf{B}\| .$$

We have briefly discussed the concept of condition number of a matrix in section 1.1.4. Note that the condition number can also be computed as:

$$\text{cond}(\mathbf{A}) = \|\mathbf{A}\| \|\mathbf{A}^{-1}\| .$$

Direct Methods for Linear Systems of Equations

In many applications, the problem solution involves linear systems of equations (LSEs). To name a few applications, LSEs arise in filtering, spline interpolation, gradient descent methods and the solution of partial differential equations. We will start our discussion with the case of square matrices. Linear systems with rectangular system matrices $\mathbf{A} \in \mathbb{K}^{m,n}$, called “overdetermined” for $m > n$, and “underdetermined” for $m < n$, will be treated in a latter chapter about least squares methods.

The problem: Solving a linear system

Given : square matrix $\mathbf{A} \in \mathbb{K}^{n,n}$, vector $\mathbf{b} \in \mathbb{K}^n$, $n \in \mathbb{N}$

Sought : solution vector $\mathbf{x} \in \mathbb{K}^n$ that solves the LSE $\mathbf{Ax} = \mathbf{b}$

(Formal problem mapping $(\mathbf{A}, \mathbf{b}) \mapsto \mathbf{A}^{-1}\mathbf{b}$)

(Terminology: \mathbf{A} : system matrix/coefficient matrix, \mathbf{b} : right-hand side vector)

2.1 Existence and uniqueness of solutions

Recall from linear algebra:

Definition 2.1.1 (Invertible matrix).

$\mathbf{A} \in \mathbb{K}^{n,n}$ *invertible / regular* : There exists a unique matrix $\mathbf{B} \in \mathbb{K}^{n,n}$ such that

$$\mathbf{AB} = \mathbf{BA} = \mathbf{I} .$$

We say that \mathbf{B} is the inverse of \mathbf{A} and write $\mathbf{B} = \mathbf{A}^{-1}$.

We need a few concepts from linear algebra to state criteria for the invertibility of a matrix.

Definition 2.1.2 (Image space and kernel of a matrix). Given $\mathbf{A} \in \mathbb{K}^{m,n}$, the *range/image* (space) of \mathbf{A} is the subspace of \mathbb{K}^m spanned by the columns of \mathbf{A}

$$\mathcal{R}(\mathbf{A}) := \{\mathbf{Ax}, \mathbf{x} \in \mathbb{K}^n\} \subset \mathbb{K}^m .$$

The *kernel/nullspace* of \mathbf{A} is

$$\mathcal{N}(\mathbf{A}) := \{\mathbf{z} \in \mathbb{K}^n : \mathbf{Az} = \mathbf{0}\} .$$

Definition 2.1.3 (Rank of a matrix). The *rank* of a matrix $\mathbf{A} \in \mathbb{K}^{m,n}$, denoted by $\text{rank}(\mathbf{A})$, is the maximal number of linearly independent rows/columns of \mathbf{A} . Equivalently,

$$\text{rank}(\mathbf{A}) = \dim \mathcal{R}(\mathbf{A}) .$$

Theorem 2.1.1 (Criteria for invertibility of matrix). A square matrix $\mathbf{A} \in \mathbb{K}^{n,n}$ is invertible/regular if one of the following equivalent conditions is satisfied:

1. $\exists \mathbf{B} \in \mathbb{K}^{n,n} : \mathbf{BA} = \mathbf{AB} = \mathbf{I}$,
2. $\mathbf{x} \mapsto \mathbf{Ax}$ defines an endomorphism of \mathbb{K}^n ,
3. the columns of \mathbf{A} are linearly independent (full column rank),
4. the rows of \mathbf{A} are linearly independent (full row rank),
5. $\det \mathbf{A} \neq 0$ (non-vanishing determinant),
6. $\text{rank}(\mathbf{A}) = n$ (full rank).


2.1.1 Solution of an LSE as a *problem*

Linear algebra gives us a *formal* way to denote a solution of an LSE:

If $\mathbf{A} \in \mathbb{K}^{n,n}$ regular & $\mathbf{Ax} = \mathbf{b}$, then the solution is given by $\mathbf{x} = \underbrace{\mathbf{A}^{-1}}_{\text{inverse matrix}} \mathbf{b} .$

Now recall our notion of *problem* from before as a function F mapping data in a data space X to a result in a result space Y . Concretely, for $n \times n$ linear systems of equations:

$$F : \begin{cases} X := \mathbb{K}_*^{n,n} \times \mathbb{K}^n & \rightarrow & Y := \mathbb{K}^n \\ (\mathbf{A}, \mathbf{b}) & \mapsto & \mathbf{A}^{-1}\mathbf{b} \end{cases}$$

 Notation: (open) set of regular matrices $\subset \mathbb{K}^{n,n}$:

$$\mathbb{K}_*^{n,n} := \{\mathbf{A} \in \mathbb{K}^{n,n} : \mathbf{A} \text{ regular/invertible, see Definition 2.1.1}\}.$$

The inverse matrix and the solution of an LSE

EIGEN: inverse of a matrix \mathbf{A} is available through `A.inverse()`

Always avoid computing the inverse of a matrix (which can almost always be avoided)!

In particular, never use `x = A.inverse()*b` to solve the linear system of equations $\mathbf{Ax} = \mathbf{b}$. The next sections present a better way to solve this system.

2.1.2 Sensitivity of linear systems

Note. The *sensitivity* of a problem (for given data) gauges the impact of small perturbations of the data on the result.

We have already discussed in Chapter 1, that the condition number of the system matrix determines the stability of an LSE. We want to make this more explicit and derive that the condition number $\text{cond}(\mathbf{A}) = \|\mathbf{A}\| \|\mathbf{A}^{-1}\|$ determines the effect of relative errors in the input on relative errors in the output. Before we examine sensitivity for linear systems of equations, we look at the simpler problem of matrix \times vector multiplication.

Example 2.1.1: Sensitivity of linear mappings

For a *fixed* given regular $\mathbf{A} \in \mathbb{K}^{n,n}$ we study the problem map

$$F : \mathbb{K}^n \rightarrow \mathbb{K}^n, \quad \mathbf{x} \mapsto \mathbf{Ax},$$

that is, now we consider only the vector \mathbf{x} as data.

Goal: Estimate relative perturbations in $F(\mathbf{x})$ due to relative perturbations in \mathbf{x} .

We assume that \mathbb{K}^n is equipped with *some* vector norm and we use the induced matrix norm (see Definition 1.6.2) on $\mathbb{K}^{n,n}$. Using linearity and the elementary estimate $\|\mathbf{Mx}\| \leq \|\mathbf{M}\| \|\mathbf{x}\|$, which is a direct consequence of the definition of an induced matrix norm, we obtain

$$\begin{aligned} \mathbf{Ax} = \mathbf{b} &\Rightarrow \|\mathbf{x}\| \leq \|\mathbf{A}^{-1}\| \|\mathbf{b}\|, \\ \mathbf{A}(\mathbf{x} + \Delta\mathbf{x}) = \mathbf{b} + \Delta\mathbf{b} &\Rightarrow \mathbf{A}\Delta\mathbf{x} = \Delta\mathbf{b} \Rightarrow \|\Delta\mathbf{b}\| \leq \|\mathbf{A}\| \|\Delta\mathbf{x}\| \end{aligned}$$

$$\Rightarrow \frac{\|\Delta \mathbf{b}\|}{\|\mathbf{b}\|} \leq \frac{\|\mathbf{A}\| \|\Delta \mathbf{x}\|}{\|\mathbf{A}^{-1}\|^{-1} \|\mathbf{x}\|} = \|\mathbf{A}\| \|\mathbf{A}^{-1}\| \left(\frac{\|\Delta \mathbf{x}\|}{\|\mathbf{x}\|} \right). \quad (2.1)$$

\uparrow \uparrow
 relative perturbation in result relative perturbation in data

We have found that the quantity $\text{cond}(\mathbf{A}) = \|\mathbf{A}\| \|\mathbf{A}^{-1}\|$ bounds amplification of relative errors in the argument vector in a matrix×vector-multiplication with the matrix \mathbf{A} .

Now we study the sensitivity of the problem of finding the solution of a linear system of equations $\mathbf{Ax} = \mathbf{b}$, $\mathbf{A} \in \mathbb{R}^{n,n}$ regular, $\mathbf{b} \in \mathbb{R}$. We write $\tilde{\mathbf{x}}$ for the solution of the perturbed linear system.

Question: Suppose that the system matrix \mathbf{A} and the right-hand side \mathbf{b} are perturbed. Can we give an upper bound on the

$$(\text{normwise}) \text{ relative error: } \epsilon_r := \frac{\|\mathbf{x} - \tilde{\mathbf{x}}\|}{\|\mathbf{x}\|} ?$$

($\|\cdot\|$: suitable vector norm, e.g., maximum norm $\|\cdot\|_\infty$)

Perturbed linear system:

$$\mathbf{Ax} = \mathbf{b} \leftrightarrow (\mathbf{A} + \Delta \mathbf{A})\tilde{\mathbf{x}} = \mathbf{b} + \Delta \mathbf{b} \implies (\mathbf{A} + \Delta \mathbf{A})(\tilde{\mathbf{x}} - \mathbf{x}) = \Delta \mathbf{b} - \Delta \mathbf{Ax}. \quad (2.2)$$

Theorem 2.1.2 (Conditioning of LSEs). *Let \mathbf{A} be regular, and suppose that for a perturbed system as in (2.2), we have that $\|\Delta \mathbf{A}\| < \|\mathbf{A}^{-1}\|^{-1}$, then*

(i) $\mathbf{A} + \Delta \mathbf{A}$ is also regular/invertible,

(ii)

$$\frac{\|\mathbf{x} - \tilde{\mathbf{x}}\|}{\|\mathbf{x}\|} \leq \frac{\|\mathbf{A}^{-1}\| \|\mathbf{A}\|}{1 - \|\mathbf{A}^{-1}\| \|\Delta \mathbf{A}\|} \left(\frac{\|\Delta \mathbf{b}\|}{\|\mathbf{b}\|} + \frac{\|\Delta \mathbf{A}\|}{\|\mathbf{A}\|} \right).$$

\uparrow $\nwarrow \nearrow$
 relative error of data relative perturbations

The proof is based on the following fundamental result:

Lemma 2.1.1 (Perturbation lemma). $\mathbf{B} \in \mathbb{R}^{n,n}$, $\|\mathbf{B}\| < 1 \implies \mathbf{I} + \mathbf{B}$ is regular and $\|(\mathbf{I} + \mathbf{B})^{-1}\| \leq \frac{1}{1 - \|\mathbf{B}\|}$.

Proof. (Lemma 2.1.1):

Using the (reverse) \triangle -inequality, we can obtain

$$\begin{aligned} \|(\mathbf{I} + \mathbf{B})\mathbf{x}\| &= \|\mathbf{x} - (-\mathbf{B}\mathbf{x})\|, \quad \forall \mathbf{x} \in \mathbb{R}^n \\ &\geq \|\mathbf{x}\| - \|(-\mathbf{B}\mathbf{x})\|, \quad \forall \mathbf{x} \in \mathbb{R}^n \\ &\geq \|\mathbf{x}\| - \|\mathbf{B}\| \|\mathbf{x}\| = (1 - \|\mathbf{B}\|) \|\mathbf{x}\|, \quad \forall \mathbf{x} \in \mathbb{R}^n. \end{aligned} \quad (2.3)$$

Since $\|\mathbf{B}\| < 1$, we can conclude that $(\mathbf{I} + \mathbf{B})$ is regular. The second part of the Lemma can be proved using the definition of the matrix norm (see Definition 1.6.2), as follows:

$$\|(\mathbf{I} + \mathbf{B})^{-1}\| = \sup_{\mathbf{x} \in \mathbb{R}^n \setminus \{0\}} \frac{\|(\mathbf{I} + \mathbf{B})^{-1}\mathbf{x}\|}{\|\mathbf{x}\|} \stackrel{=}{=} \sup_{\mathbf{y} := (\mathbf{I} + \mathbf{B})^{-1}\mathbf{x}} \frac{\|\mathbf{y}\|}{\|(\mathbf{I} + \mathbf{B})\mathbf{y}\|}$$

By (2.3), $\|(\mathbf{I} + \mathbf{B})\mathbf{y}\| \geq (1 - \|\mathbf{B}\|) \|\mathbf{y}\|$. This implies:

$$\|(\mathbf{I} + \mathbf{B})^{-1}\| \leq \frac{1}{1 - \|\mathbf{B}\|}.$$

□

Proof. (Theorem 2.1.2):

$$\|(\mathbf{A} + \Delta\mathbf{A})^{-1}\| = \|(\mathbf{I} + \mathbf{A}^{-1}\Delta\mathbf{A})^{-1}\mathbf{A}^{-1}\| \leq \|\mathbf{A}^{-1}\| \|(\mathbf{I} + \mathbf{A}^{-1}\Delta\mathbf{A})^{-1}\|. \quad (2.4)$$

Since $\|\Delta\mathbf{A}\| < \|\mathbf{A}^{-1}\|^{-1}$, this implies $\|\mathbf{A}^{-1}\| \|\Delta\mathbf{A}\| < 1$, that is, $\|\mathbf{A}^{-1}\Delta\mathbf{A}\| < 1$. Therefore, we can use (2.4) and Lemma 2.1.1 to obtain $\|(\mathbf{A} + \Delta\mathbf{A})^{-1}\| \leq \frac{\|\mathbf{A}^{-1}\|}{1 - \|\mathbf{A}^{-1}\Delta\mathbf{A}\|}$.

Using (2.2), $\Delta\mathbf{x} = (\mathbf{A} + \Delta\mathbf{A})^{-1}(\Delta\mathbf{b} - \Delta\mathbf{A}\mathbf{x})$, which implies $\|\Delta\mathbf{x}\| \leq \|(\mathbf{A} + \Delta\mathbf{A})^{-1}\| \|\Delta\mathbf{b} - \Delta\mathbf{A}\mathbf{x}\|$.

Combining these two results, we obtain

$$\|\Delta\mathbf{x}\| \leq \frac{\|\mathbf{A}^{-1}\|}{1 - \|\mathbf{A}^{-1}\Delta\mathbf{A}\|} (\|\Delta\mathbf{b}\| + \|\Delta\mathbf{A}\mathbf{x}\|) \leq \frac{\|\mathbf{A}^{-1}\| \|\mathbf{A}\|}{1 - \|\mathbf{A}^{-1}\| \|\Delta\mathbf{A}\|} \left(\frac{\|\Delta\mathbf{b}\|}{\|\mathbf{A}\| \|\mathbf{x}\|} + \frac{\|\Delta\mathbf{A}\|}{\|\mathbf{A}\|} \right) \|\mathbf{x}\|.$$

□

Note. The condition number $\text{cond}(\mathbf{A})$ depends on the chosen matrix norm $\|\cdot\|$.

Note. If there is no perturbation in \mathbf{A} and we consider errors in the right-hand side \mathbf{b} only, then the bound on the relative error in the solution that is given in Theorem 2.1.2 simplifies to:

$$\frac{\|\mathbf{x} - \tilde{\mathbf{x}}\|}{\|\mathbf{x}\|} \leq \frac{\text{cond}(\mathbf{A}) \|\Delta\mathbf{b}\|}{\|\mathbf{b}\|}.$$

On the other hand, if $\Delta\mathbf{b} = 0$, we obtain

$$\epsilon_r := \frac{\|\mathbf{x} - \tilde{\mathbf{x}}\|}{\|\mathbf{x}\|} \leq \frac{\text{cond}(\mathbf{A}) \delta_A}{1 - \text{cond}(\mathbf{A}) \delta_A} \quad \text{where} \quad \delta_A := \frac{\|\Delta\mathbf{A}\|}{\|\mathbf{A}\|}. \quad (2.5)$$

We conclude:

- If $\text{cond}(\mathbf{A}) \gg 1$, *small perturbations* in \mathbf{A} can lead to *large relative errors* in the solution of the LSE.
- If $\text{cond}(\mathbf{A}) \gg 1$, even a numerically stable algorithm (see Definition 1.6.1) can produce solutions with large relative error.

If $\text{cond}(\mathbf{A}) \gg 1$, then small relative changes of data \mathbf{A}, \mathbf{b} *may* lead to huge relative changes in the solution. Thus, $\text{cond}(\mathbf{A})$ indicates the sensitivity of an LSE problem

$$(\mathbf{A}, \mathbf{b}) \mapsto \mathbf{x} = \mathbf{A}^{-1}\mathbf{b} .$$

(as “amplification factor” of (worst-case) relative perturbations in the data \mathbf{A}, \mathbf{b}).

Terminology:

small changes in data \Rightarrow small perturbations of result : **well-conditioned** problem
 small changes in data \Rightarrow large perturbations of result : **ill-conditioned** problem

Note. Sensitivity gauge depends on the chosen norm.

2.2 Gaussian Elimination

Idea: Transformation to “simpler”, but equivalent LSE by means of successive (invertible) *row transformations*

row transformations \leftrightarrow left-multiplication with transformation matrix

Obviously, left multiplication with a regular matrix does not affect the solution of an LSE:
 For any *regular* $\mathbf{T} \in \mathbb{K}^{n,n}$

$$\mathbf{Ax} = \mathbf{b} \Rightarrow \mathbf{A}'\mathbf{x} = \mathbf{b}' \quad , \text{ if } \mathbf{A}' = \mathbf{TA}, \mathbf{b}' = \mathbf{Tb} .$$

So we may try to convert the linear system of equations to a form that can be solved more easily by multiplying with regular matrices from the left, which boils down to applying row transformations. A suitable target format is a diagonal linear system of equations, for which all equations are completely decoupled. This is the gist of Gaussian elimination.

Example 2.2.1: Gaussian elimination

① (Forward) elimination:

$$\begin{pmatrix} 1 & 1 & 0 \\ 2 & 1 & -1 \\ 3 & -1 & -1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 4 \\ 1 \\ -3 \end{pmatrix} \longleftrightarrow \begin{array}{rrc} x_1 + x_2 & & = 4 \\ 2x_1 + x_2 - x_3 & = & 1 \\ 3x_1 - x_2 - x_3 & = & -3 \end{array}.$$

$$\begin{bmatrix} 1 & 1 & 0 \\ 2 & 1 & -1 \\ 3 & -1 & -1 \end{bmatrix} \begin{bmatrix} 4 \\ 1 \\ -3 \end{bmatrix} \rightarrow \begin{bmatrix} \mathbf{1} & 1 & 0 \\ \mathbf{0} & -1 & -1 \\ 3 & -1 & -1 \end{bmatrix} \begin{bmatrix} 4 \\ -7 \\ -3 \end{bmatrix} \rightarrow \begin{bmatrix} \mathbf{1} & 1 & 0 \\ 0 & -1 & -1 \\ \mathbf{0} & -4 & -1 \end{bmatrix} \begin{bmatrix} 4 \\ -7 \\ -15 \end{bmatrix}$$

$$\rightarrow \underbrace{\begin{bmatrix} 1 & 1 & 0 \\ 0 & \mathbf{-1} & \mathbf{-1} \\ 0 & \mathbf{0} & 3 \end{bmatrix}}_{=U} \begin{bmatrix} 4 \\ -7 \\ 13 \end{bmatrix}$$

 = pivot row, pivot element in bold.

Goal. Transformation of an LSE to *upper triangular form*

② Solve by *back substitution*:

$$\begin{array}{rrc} x_1 + x_2 & & = 4 & x_3 = \frac{13}{3} \\ & - x_2 - x_3 & = -7 & \Rightarrow x_2 = 7 - \frac{13}{3} = \frac{8}{3} \\ & 3x_3 & = 13 & x_1 = 4 - \frac{8}{3} = \frac{4}{3}. \end{array}$$

More detailed examples: [4, Sect. 1.1], [3, Sect. 1.1].

More general:

$$\begin{array}{cccccccc} a_{11}x_1 & + & a_{12}x_2 & + & \cdots & + & a_{1n}x_n & = & b_1 \\ a_{21}x_1 & + & a_{22}x_2 & + & \cdots & + & a_{2n}x_n & = & b_2 \\ \vdots & & \vdots & & \vdots & & \vdots & & \vdots \\ \vdots & & \vdots & & \vdots & & \vdots & & \vdots \\ a_{n1}x_1 & + & a_{n2}x_2 & + & \cdots & + & a_{nn}x_n & = & b_n \end{array}$$

2 Direct Methods for Linear Systems of Equations

- i -th row - $l_{i1} \cdot$ 1st row (*pivot row*), $l_{i1} := \frac{a_{i1}}{a_{11}}, i = 2, \dots, n$

$$\begin{array}{ccccccccc}
 a_{11} x_1 & + & a_{12} x_2 & + & \cdots & + & a_{1n} x_n & = & b_1 \\
 & & a_{22}^{(1)} x_2 & + & \cdots & + & a_{2n}^{(1)} x_n & = & b_2^{(1)} \quad \text{with} \\
 & & \vdots & & \vdots & & \vdots & & \vdots \\
 & & \vdots & & \vdots & & \vdots & & \vdots \\
 & & a_{n2}^{(1)} x_2 & + & \cdots & + & a_{nn}^{(1)} x_n & = & b_n^{(1)}
 \end{array}
 \quad
 \begin{array}{l}
 a_{ij}^{(1)} = a_{ij} - a_{1j} l_{i1}, \quad i, j = 2, \dots, n, \\
 b_i^{(1)} = b_i - b_1 l_{i1}, \quad i = 2, \dots, n.
 \end{array}$$

- i -th row - $l_{i1} \cdot$ 2nd row (*pivot row*), $l_{i2} := \frac{a_{i2}^{(1)}}{a_{22}^{(1)}}, i = 3, \dots, n$.

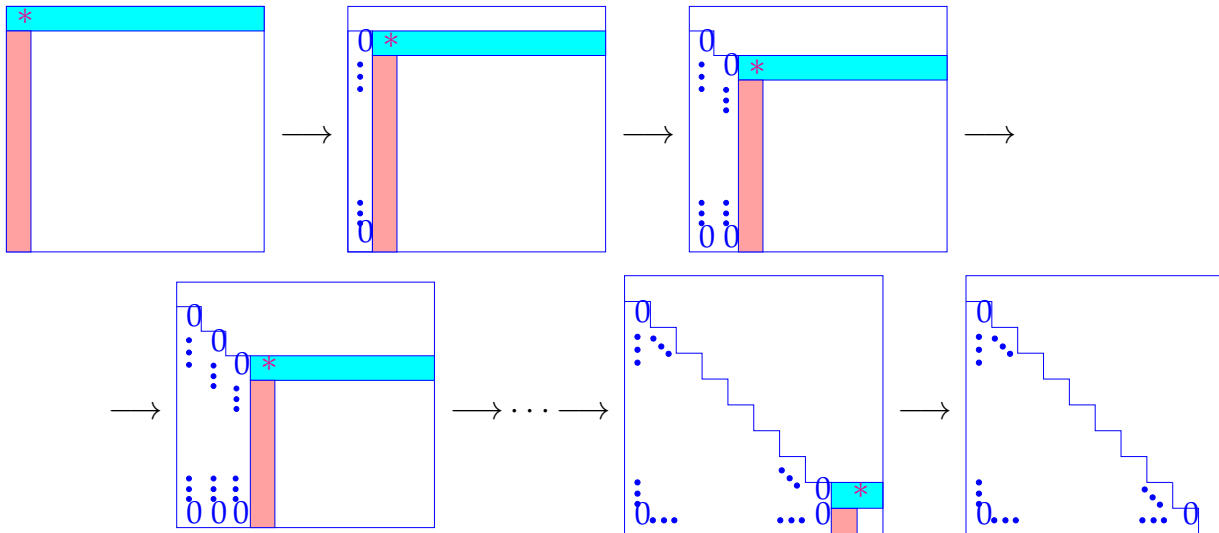
$$\begin{array}{ccccccccccc}
 a_{11} x_1 & + & a_{12} x_2 & + & a_{13} x_3 & + & \cdots & + & a_{1n} x_n & = & b_1 \\
 & & a_{22}^{(1)} x_2 & + & a_{23}^{(1)} x_3 & + & \cdots & + & a_{2n}^{(1)} x_n & = & b_2^{(1)} \\
 & & & & a_{33}^{(2)} x_3 & + & \cdots & + & a_{3n}^{(2)} x_n & = & b_3^{(2)} \\
 & & & & \vdots & & \vdots & & \vdots & & \vdots \\
 & & & & a_{n3}^{(2)} x_3 & + & \cdots & + & a_{nn}^{(2)} x_n & = & b_n^{(2)}
 \end{array}$$

\Rightarrow After $n - 1$ steps: linear systems of equations in *upper triangular form*

$$\begin{array}{ccccccccccc}
 a_{11} x_1 & + & a_{12} x_2 & + & a_{13} x_3 & + & \cdots & + & a_{1n} x_n & = & b_1 \\
 & & a_{22}^{(1)} x_2 & + & a_{23}^{(1)} x_3 & + & \cdots & + & a_{2n}^{(1)} x_n & = & b_2^{(1)} \\
 & & & & a_{33}^{(2)} x_3 & + & \cdots & + & a_{3n}^{(2)} x_n & = & b_3^{(2)} \\
 & & & & \ddots & \ddots & \ddots & & \vdots & & \vdots \\
 & & & & & \ddots & \ddots & & \vdots & & \vdots \\
 & & & & & & & & a_{nn}^{(n-1)} x_n & = & b_n^{(n-1)}
 \end{array}$$

Terminology: $a_{11}, a_{22}^{(1)}, a_{33}^{(2)}, \dots, a_{n-1, n-1}^{(n-2)}$ = *pivots/pivot elements*

Graphical depiction:



*: The pivot element is always assumed to be non-zero. In practice, this has to be enforced with a procedure called *pivoting*, which we briefly discuss below.

Here we give a direct EIGEN implementation of Gaussian elimination for an LSE $\mathbf{Ax} = \mathbf{b}$ (This is for demonstration purposes only and is grossly inefficient).

Code Snippet 2.1: Solving an LSE $\mathbf{Ax} = \mathbf{b}$ with Gaussian elimination → GITLAB

```

16 /// Gauss elimination without pivoting, x = A\b
17 /// A must be an n x n-matrix, b an n-vector
18 /// The result is returned in x
19 void gausselimsolve(const MatrixXd &A, const VectorXd& b,
20     VectorXd& x) {
21     int n = A.rows();
22     MatrixXd Ab(n,n+1); // Augmented matrix [A,b]
23     Ab << A, b; //
24     // Forward elimination (cf. step ① in Example 2.2.1)
25     for(int i = 0; i < n-1; ++i) {
26         double pivot = Ab(i,i);
27         for(int k = i+1; k < n; ++k) {
28             double fac = Ab(k,i)/pivot;
29             Ab.block(k,i+1,1,n-i)-= fac * Ab.block(i,i+1,1,n-i); //
30         }
31     }
32     // Back substitution (cf. step ② in Example 2.2.1)
33     Ab(n-1,n) = Ab(n-1,n) / Ab(n-1,n-1);
34     for(int i = n-2; i >= 0; --i) {
35         for(int l = i+1; l < n; ++l) Ab(i,n) -= Ab(l,n)*Ab(i,l);
36         Ab(i,n) /= Ab(i,i);
37     }
38     x = Ab.rightCols(1); //
39 }
40 /*

```

Line 23: Right-hand side vector set as last column of matrix, facilitates simultaneous row

transformations of matrix and r.h.s.

Variable fac: multiplier

Line 38: extract solution from last column of transformed matrix.

Computational effort of Gaussian elimination

Forward elimination consists of three nested loops.

(Note: compact vector operation in Line 29 involves another loop from $i + 1$ to m)

Back substitution consists of two nested loops.

We count the number of elementary operations of Gaussian elimination to obtain its computational cost:

$$\begin{aligned} \text{Elimination : } & \sum_{i=1}^{n-1} (n-i)(2(n-i)+3) = n(n-1)\left(\frac{2}{3}n + \frac{7}{6}\right) \text{ Ops ,} \\ \text{Back substitution : } & \sum_{i=1}^n 2(n-i) + 1 = n^2 \text{ Ops .} \end{aligned} \quad (2.6)$$

Note. Asymptotic complexity (see Section 1.4.1) of Gaussian elimination (without pivoting) for a generic LSE $\mathbf{Ax} = \mathbf{b}$, where $\mathbf{A} \in \mathbb{R}^{n,n}$ is equal to $\frac{2}{3}n^3 + \mathcal{O}(n^2) = \mathcal{O}(n^3)$

Numerical stability and pivoting

As with any algorithm, we need to ask whether Gaussian elimination is numerically stable. In general, the answer is "No". The reason for this is the possibility of dividing by small numbers and that cancellation can occur in the subtraction.

A good remedy in practice is *partial pivoting*. In this procedure, at every iteration step, one looks through the current column and permutes the rows of the matrix so that the one with the largest absolute values lies on the diagonal. This way, one avoids dividing by small numbers. Note that row permutations do not change the LSE.

Partial pivoting works well in practice, however it does not guarantee numerical stability. System matrices for which Gaussian elimination is numerically stable are, for example, s.p.d. matrices and diagonally dominant matrices. The latter means that $\mathbf{A} \in \mathbb{K}^{n,n}$ is diagonally dominant, if:

$$\forall k \in \{1, \dots, n\}: \quad \sum_{j \neq k} |a_{kj}| \leq |a_{kk}| .$$

2.2.1 Alternative way: LU-Decomposition

A *matrix factorization* (ger. Matrixzerlegung) expresses a general matrix \mathbf{A} as product of two *special* (factor) matrices. Requirements for these special matrices define the matrix

factorization.

Mathematical issue: existence & uniqueness

Numerical issue: algorithm for computing factor matrices

Matrix factorizations

- often capture the essence of algorithms in compact form (here: Gaussian elimination),
- are important building blocks for complex algorithms,
- are key theoretical tools for algorithm analysis.

In this section, the forward elimination step of Gaussian elimination will be related to a special matrix factorization, the so-called LU-decomposition or LU-factorization. The motivation of considering this factorization is the following: Suppose we want to solve the same system for N different right-hand sides $\mathbf{b}_1, \dots, \mathbf{b}_N$. Then, it would be more economical to store the transformations from Gaussian elimination that were undertaken to obtain the upper triangular form. This is what the LU factorization does.

Note. The (forward) Gaussian elimination (without pivoting), for $\mathbf{Ax} = \mathbf{b}$, $\mathbf{A} \in \mathbb{R}^{n,n}$, if possible, is *algebraically equivalent* to an LU-factorization/ LU-decomposition $\mathbf{A} = \mathbf{LU}$ of \mathbf{A} into a normalized lower triangular matrix \mathbf{L} and an upper triangular matrix \mathbf{U} .

Definition 2.2.1 (LU-decomposition/LU-factorization). Given a square matrix $\mathbf{A} \in \mathbb{K}^{n,n}$, an upper triangular matrix $\mathbf{U} \in \mathbb{K}^{n,n}$, a normalized lower triangular matrix $\mathbf{L} \in \mathbb{K}^{n,n}$ and a permutation matrix $\mathbf{P} \in \mathbb{K}^{n,n}$ form an *LU-decomposition/ LU-factorization* of \mathbf{A} , if $\mathbf{PA} = \mathbf{LU}$.

$$\begin{bmatrix} \text{Matrix A} \end{bmatrix} = \begin{bmatrix} \text{Matrix L} \end{bmatrix} \cdot \begin{bmatrix} \text{Matrix U} \end{bmatrix} = \begin{bmatrix} \text{Matrix LU} \end{bmatrix}$$

Lemma 2.2.1 (Existence of LU-decomposition). The LU-decomposition of $\mathbf{A} \in \mathbb{K}^{n,n}$ exists, if all submatrices $(\mathbf{A})_{1:k,1:k}$, $1 \leq k \leq n$, are regular.

2.2.2 Using LU-factorization to solve a linear system of equations

Solving an $n \times n$ linear system of equations by LU-factorization:

① LU-decomposition $\mathbf{L} \underbrace{\mathbf{U}\mathbf{x}}_{\mathbf{z}} = \mathbf{b},$

#elementary operations $\frac{1}{3}n(n-1)(n+1)$

$\mathbf{Ax} = \mathbf{b}$: ② forward substitution: solve $\mathbf{Lz} = \mathbf{b}$ for $\mathbf{z},$

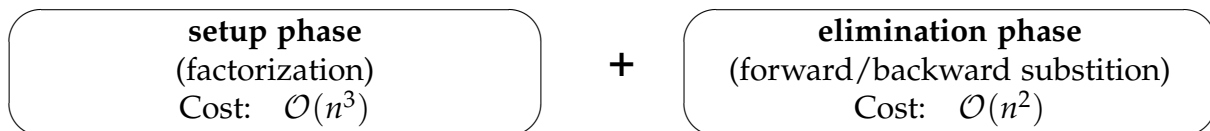
#elementary operations $\frac{1}{2}n(n-1)$

③ backward substitution: solve $\mathbf{Ux} = \mathbf{z}$ for $\mathbf{x},$

#elementary operations $\frac{1}{2}n(n+1)$

Asymptotic complexity: (in leading order) the same as for Gaussian elimination.

However, the perspective of LU-factorization reveals that the solution of linear systems of equations can be split into two separate phases with different asymptotic complexity in terms of the number n of unknowns:



Rationale for using LU-decomposition in algorithms

Gauss elimination and LU-factorization for the solution of a linear system of equations are equivalent and only differ in the ordering of the steps. Then, why is it important to know about LU-factorization?

Because in the case of LU-factorization, the expensive forward elimination and the less expensive (forward/backward) substitutions are separated, which can be exploited sometimes to reduce computational cost. Let's look at the following example where we want to solve the LSE for N different right-hand sides \mathbf{b} :

Code Snippet 2.2: Wasteful approach → GITLAB

```

28 // Setting:  $N \gg 1,$ 
29 // large matrix  $\mathbf{A} \in \mathbb{K}^{n,n}$ 
30 for(int j = 0; j < N; ++j){
31     x = A.lu().solve(b);
32     b = some_function(x);
33 }
34 /*

```

computational effort $\mathcal{O}(Nn^3)$

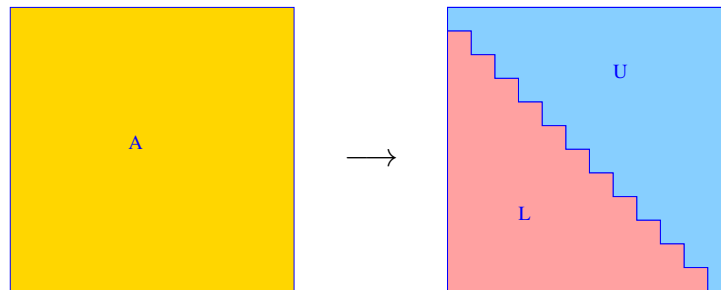
Code Snippet 2.3: Smart approach → GITLAB

```

27 // Setting:  $N \gg 1$ ,
28 // large matrix  $A \in \mathbb{K}^{n,n}$ 
29 auto A_lu_dec = A.lu();
30 for(int j = 0; j < N; ++j){
31     x = A_lu_dec.solve(b);
32     b = some_function(x);
33 }
34 /*

```

computational effort $\mathcal{O}(n^3 + Nn^2)$

In-situ LU-decomposition

Replace entries of **A** with entries of **L** (strict lower triangle) and **U** (upper triangle).

Remark. In general, do not implement a general solver for linear systems of equations yourself, but rather use algorithms from numerical libraries when possible.

2.3 Exploiting structure when solving Linear Systems

By “structure” of a linear system, we mean prior knowledge that

- either certain entries of the system matrix are zero,
- or the system matrix is generated by a particular formula.

A simple example is using the knowledge that the system matrix is triangular to reduce the complexity of solving the LSE from $\mathcal{O}(n^3)$ to $\mathcal{O}(n^2)$.

We will now consider two cases of system matrices with special structures. First, matrices for which a submatrix has a very specific structure and is easy to solve. With a block matrix representation, one can exploit this property. The other one deals with “small” updates of a matrix: Do we need to discard all our knowledge of the system matrix A , if, for example, one entry of A changes?

2.3.1 Block elimination

Given matrix dimensions $M, N, K \in \mathbb{N}$, block dimensions $n < N$ ($n' := N - n$), $m < M$ ($m' := M - m$), $k < K$ ($k' := K - k$), we start from the following matrices:

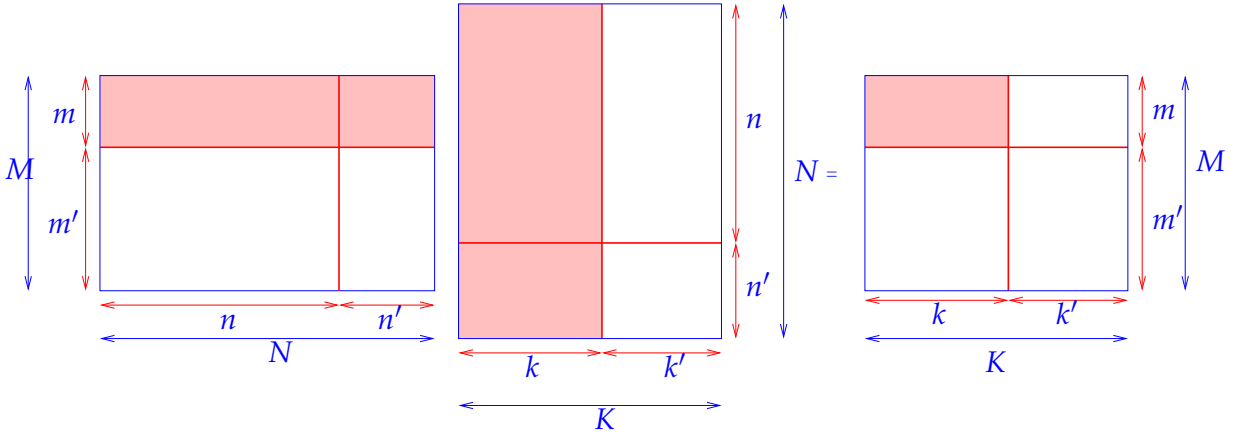
$$\begin{array}{llll} \mathbf{A}_{11} \in \mathbb{K}^{m,n} & \mathbf{A}_{12} \in \mathbb{K}^{m,n'} & \mathbf{B}_{11} \in \mathbb{K}^{n,k} & \mathbf{B}_{12} \in \mathbb{K}^{n,k'} \\ \mathbf{A}_{21} \in \mathbb{K}^{m',n} & \mathbf{A}_{22} \in \mathbb{K}^{m',n'} & \mathbf{B}_{21} \in \mathbb{K}^{n',k} & \mathbf{B}_{22} \in \mathbb{K}^{n',k'} \end{array}.$$

These matrices serve as sub-matrices or matrix blocks and are assembled into larger matrices

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{bmatrix} \in \mathbb{K}^{M,N}, \quad \mathbf{B} = \begin{bmatrix} \mathbf{B}_{11} & \mathbf{B}_{12} \\ \mathbf{B}_{21} & \mathbf{B}_{22} \end{bmatrix} \in \mathbb{K}^{N,K}.$$

It turns out that the matrix product \mathbf{AB} can be computed by the same formula as the product of simple 2×2 -matrices:

$$\begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{bmatrix} \begin{bmatrix} \mathbf{B}_{11} & \mathbf{B}_{12} \\ \mathbf{B}_{21} & \mathbf{B}_{22} \end{bmatrix} = \begin{bmatrix} \mathbf{A}_{11}\mathbf{B}_{11} + \mathbf{A}_{12}\mathbf{B}_{21} & \mathbf{A}_{11}\mathbf{B}_{12} + \mathbf{A}_{12}\mathbf{B}_{22} \\ \mathbf{A}_{21}\mathbf{B}_{11} + \mathbf{A}_{22}\mathbf{B}_{21} & \mathbf{A}_{21}\mathbf{B}_{12} + \mathbf{A}_{22}\mathbf{B}_{22} \end{bmatrix}.$$



Bottom line: One can compute with block-structured matrices in *almost** the same way as with matrices with real/complex entries, see [6, Sect. 1.3.3].

Expressing the matrix product in a block-like fashion allows to conduct Gaussian elimination on the level of matrix blocks.

For $k, \ell \in \mathbb{N}$, consider the block partitioned $n \times n$ linear system, where $n = k + \ell$:

$$\begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{bmatrix} \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{bmatrix} = \begin{bmatrix} \mathbf{b}_1 \\ \mathbf{b}_2 \end{bmatrix}, \quad \begin{array}{l} \mathbf{A}_{11} \in \mathbb{K}^{k,k}, \mathbf{A}_{12} \in \mathbb{K}^{k,\ell}, \mathbf{A}_{21} \in \mathbb{K}^{\ell,k}, \mathbf{A}_{22} \in \mathbb{K}^{\ell,\ell}, \\ \mathbf{x}_1 \in \mathbb{K}^k, \mathbf{x}_2 \in \mathbb{K}^\ell, \mathbf{b}_1 \in \mathbb{K}^k, \mathbf{b}_2 \in \mathbb{K}^\ell. \end{array} \quad (2.7)$$

*you cannot use the commutativity of multiplication because matrix multiplication is not commutative

Using block matrix multiplication (applied to the matrix×vector product in (2.7)) we find an equivalent way to write the block partitioned linear system of equations:

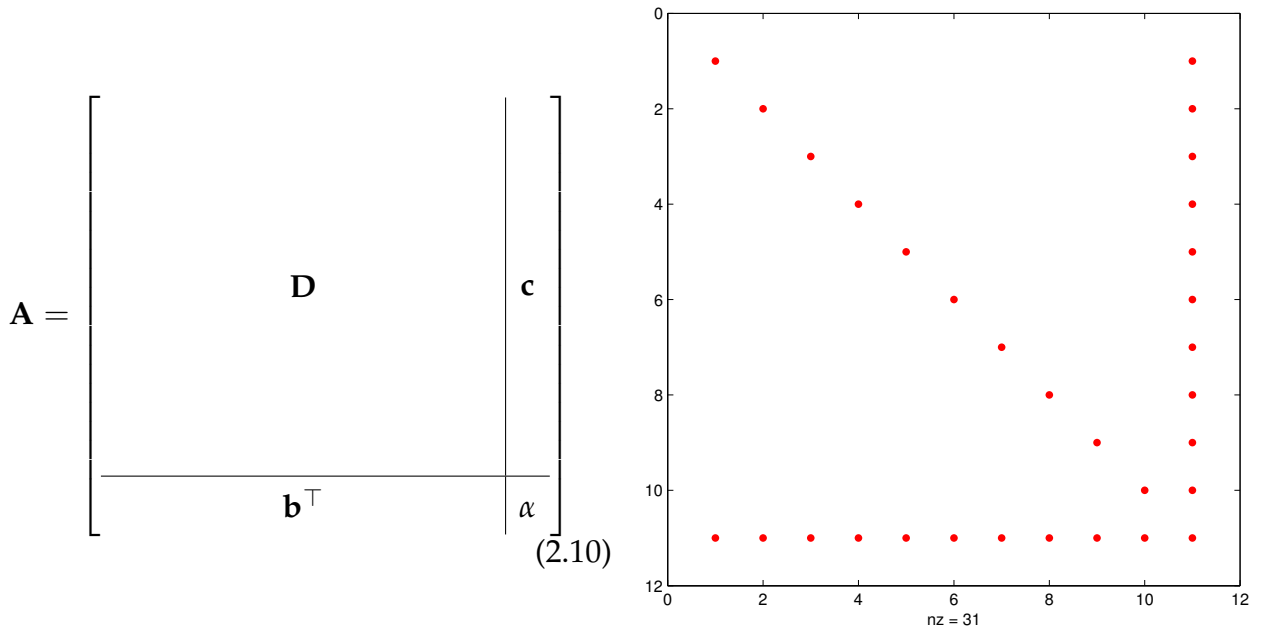
$$\begin{aligned} \mathbf{A}_{11}\mathbf{x}_1 + \mathbf{A}_{12}\mathbf{x}_2 &= \mathbf{b}_1, \\ \mathbf{A}_{21}\mathbf{x}_1 + \mathbf{A}_{22}\mathbf{x}_2 &= \mathbf{b}_2. \end{aligned} \quad (2.8)$$

We assume that \mathbf{A}_{11} is *regular* (invertible) so that we can solve for \mathbf{x}_1 from the first equation. By elementary algebraic manipulations (“block Gaussian elimination”) we find

$$\begin{aligned} \mathbf{x}_1 &= \mathbf{A}_{11}^{-1}(\mathbf{b}_1 - \mathbf{A}_{12}\mathbf{x}_2), \\ \underbrace{(\mathbf{A}_{22} - \mathbf{A}_{21}\mathbf{A}_{11}^{-1}\mathbf{A}_{12})}_{\text{Schur complement}} \mathbf{x}_2 &= \mathbf{b}_2 - \mathbf{A}_{21}\mathbf{A}_{11}^{-1}\mathbf{b}_1. \end{aligned} \quad (2.9)$$

The resulting $\ell \times \ell$ linear system of equations for the unknown vector \mathbf{x}_2 is called the *Schur complement system* for (2.7).

Unless \mathbf{A} has a special structure that allows the efficient solution of linear systems with system matrix \mathbf{A}_{11} , the Schur complement system is mainly of theoretical interest. It can be useful, however, if \mathbf{A} has a special structure so that the inverse of \mathbf{A}_{11} is easy to compute. For example, let’s consider an arrow matrix \mathbf{A} , meaning \mathbf{A}_{11} is diagonal and $\ell = 1$. Given $k \in \mathbb{N}$, a diagonal matrix $\mathbf{D} \in \mathbb{K}^{k,k}$, vectors $\mathbf{c} \in \mathbb{K}^k$, $\mathbf{b} \in \mathbb{K}^k$, and a number $\alpha \in \mathbb{K}$, we can build a $(k+1) \times (k+1)$ arrow matrix as shown below.



We can apply the block partitioning (2.7) with $k = n - 1$ and $\ell = 1$ to a linear system $\mathbf{Ax} = \mathbf{b}$ with system matrix \mathbf{A} and obtain $\mathbf{A}_{11} = \mathbf{D}$, which can be inverted easily, provided that all diagonal entries of \mathbf{D} are different from zero. In this case,

$$\mathbf{Ax} = \begin{bmatrix} \mathbf{D} & \mathbf{c} \\ \mathbf{b}^\top & \alpha \end{bmatrix} \begin{bmatrix} \mathbf{x}_1 \\ \xi \end{bmatrix} = \mathbf{b} := \begin{bmatrix} \mathbf{b}_1 \\ \beta \end{bmatrix}, \quad (2.11)$$

$$\xi = \frac{\beta - \mathbf{b}^\top \mathbf{D}^{-1} \mathbf{b}_1}{\alpha - \mathbf{b}^\top \mathbf{D}^{-1} \mathbf{c}}, \quad \mathbf{x}_1 = \mathbf{D}^{-1}(\mathbf{b}_1 - \xi \mathbf{c}). \quad (2.12)$$

These formulas make sense, if \mathbf{D} is regular and $\alpha - \mathbf{b}^\top \mathbf{D}^{-1} \mathbf{c} \neq 0$, which together guarantee the invertibility of this system. The complexity of computing $\mathbf{D}^{-1} \mathbf{c}$ and $\mathbf{D}^{-1} \mathbf{b}_1$ is $\mathcal{O}(n)$ since inverting a diagonal matrix is trivial: $\mathbf{D}^{-1} = \text{diag}(\frac{1}{d_{11}}, \dots, \frac{1}{d_{nn}})$. Therefore using formulas (2.11) and (2.12) for computing \mathbf{x} and ξ is also of linear complexity. Thus, the overall asymptotic complexity is $\mathcal{O}(n)$ instead of the $\mathcal{O}(n^3)$ for the conventional LU factorization.

Code Snippet 2.4: Solving an arrow system according to (2.12) → GITLAB

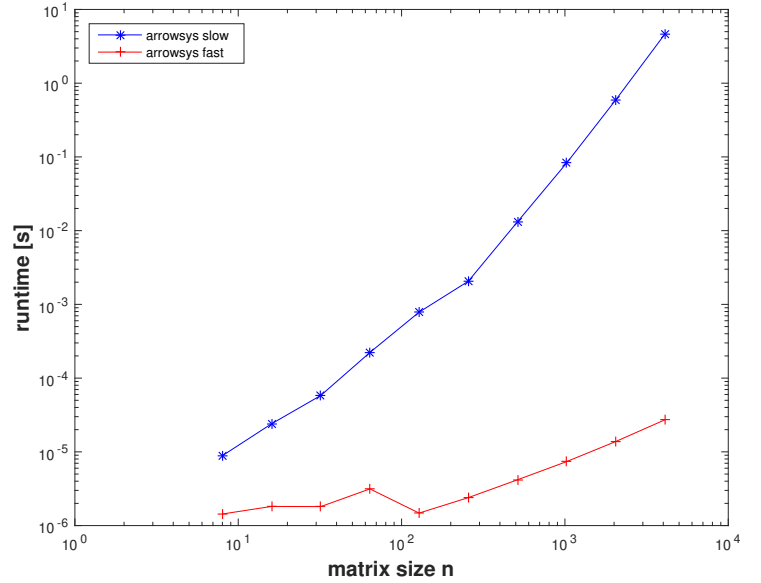
```

16 VectorXd arrowsys_fast(const VectorXd &d, const VectorXd &c, const
    ↪ VectorXd &b, const double alpha, const VectorXd &y){
17     int n = d.size();
18     VectorXd z = c.array() / d.array(); // z = D-1c
19     VectorXd w = y.head(n).array() / d.array(); // w = D-1y1
20     double xi = (y(n) - b.dot(w)) / (alpha - b.dot(z));
21     VectorXd x(n+1);
22     x << w - xi*z, xi;
23     return x;
24 }
25 /*

```

(Intel i7-3517U CPU @ 1.90GHz x 4,
64-bit, ubuntu 14.04 LTS, gcc 4.8.4,
-O3)

A runtime comparison of Code Snippet 2.4 and LU factorization for solving an LSE with arrow system matrix



Note. Block elimination can suffer from numerical instability (equivalent to Gauss elimination without pivoting).

2.3.2 Low-rank perturbation/modification of an LSE

Given a regular matrix $\mathbf{A} \in \mathbb{K}^{n,n}$, let us assume that at some point in a code, we are in a position to solve any linear system $\mathbf{A}\mathbf{x} = \mathbf{b}$ “fast”, because

- either \mathbf{A} has a favorable structure, e.g. triangular, see Section 2.3,
- or an LU-decomposition of \mathbf{A} is already available, see Section 2.2.2.

Now, suppose that the system matrix is updated to $\tilde{\mathbf{A}}$ by only a "small" change. The simplest instance we can think of, is that only one entry has been updated i.e.

$$\tilde{a}_{ij} = \begin{cases} a_{ij} & , \text{ if } (i, j) \neq (i^*, j^*) , \\ z + a_{ij} & , \text{ if } (i, j) = (i^*, j^*) , \end{cases} \quad i^*, j^* \in \{1, \dots, n\} .$$

or more compactly: $\boxed{\tilde{\mathbf{A}} = \mathbf{A} + z \cdot \mathbf{e}_{i^*} \mathbf{e}_{j^*}^\top}$. (2.13)

(Note: \mathbf{e}_i denotes the i -th unit vector)

We may also consider a matrix modification affecting a single row: Given $\mathbf{z} \in \mathbb{K}^n$, the update is defined as:

$$\tilde{a}_{ij} = \begin{cases} a_{ij} & , \text{ if } i \neq i^* , \\ (\mathbf{z})_j + a_{ij} & , \text{ if } i = i^* , \end{cases} \quad i^* \in \{1, \dots, n\} .$$

or more compactly: $\boxed{\tilde{\mathbf{A}} = \mathbf{A} + \mathbf{e}_{i^*} \mathbf{z}^\top}$. (2.14)

Both matrix modifications (2.13) and (2.14) represent *rank-1 modifications* of \mathbf{A} . In general, a rank-1 modification means adding a rank-1 matrix to \mathbf{A} , which can be written as

$$\mathbf{A} \in \mathbb{K}^{n,n} \mapsto \boxed{\tilde{\mathbf{A}} := \mathbf{A} + \mathbf{u} \mathbf{v}^H} , \quad \mathbf{u}, \mathbf{v} \in \mathbb{K}^n . \quad (2.15)$$

The question now is whether we can reuse some of the computations spent on solving $\mathbf{A} \mathbf{x} = \mathbf{b}$ in order to solve $\tilde{\mathbf{A}} \mathbf{x} = \tilde{\mathbf{b}}$ with less effort than entailed by a direct Gaussian elimination from scratch.

We can again use *Block elimination* as already seen in Section 2.3.1 on the following extended linear system:

$$\begin{bmatrix} \mathbf{A} & \mathbf{u} \\ \mathbf{v}^H & -1 \end{bmatrix} \begin{bmatrix} \tilde{\mathbf{x}} \\ \tilde{\zeta} \end{bmatrix} = \begin{bmatrix} \tilde{\mathbf{b}} \\ 0 \end{bmatrix} .$$

The *Schur complement* system after elimination of $\tilde{\zeta}$ reads

$$(\mathbf{A} + \mathbf{u} \mathbf{v}^H) \tilde{\mathbf{x}} = \tilde{\mathbf{b}} \Leftrightarrow \tilde{\mathbf{A}} \tilde{\mathbf{x}} = \tilde{\mathbf{b}} \quad !$$

Hence, we have solved the modified LSE, once we have found the component $\tilde{\mathbf{x}}$ of the solution of the extended linear system (2.3.2). We do block elimination again, now eliminating $\tilde{\mathbf{x}}$ first, which yields the other *Schur complement* system

$$(1 + \mathbf{v}^H \mathbf{A}^{-1} \mathbf{u}) \tilde{\boldsymbol{\zeta}} = \mathbf{v}^H \mathbf{A}^{-1} \tilde{\mathbf{b}}, \quad (2.16)$$

\Downarrow

$$\mathbf{A} \tilde{\mathbf{x}} = \tilde{\mathbf{b}} - \mathbf{u} \frac{\mathbf{v}^H \mathbf{A}^{-1} \tilde{\mathbf{b}}}{\mathbf{v}^H \mathbf{A}^{-1} \mathbf{u} + 1}. \quad (2.17)$$

The generalization of this formula to rank- k perturbations is given by the following lemma:

Lemma 2.3.1 (Sherman-Morrison-Woodbury formula). *For regular $\mathbf{A} \in \mathbb{K}^{n,n}$, and $\mathbf{U}, \mathbf{V} \in \mathbb{K}^{n,k}$, $k \leq n$, the following holds*

$$(\mathbf{A} + \mathbf{U} \mathbf{V}^H)^{-1} = \mathbf{A}^{-1} - \mathbf{A}^{-1} \mathbf{U} (\mathbf{I} + \mathbf{V}^H \mathbf{A}^{-1} \mathbf{U})^{-1} \mathbf{V}^H \mathbf{A}^{-1},$$

if $\mathbf{I} + \mathbf{V}^H \mathbf{A}^{-1} \mathbf{U}$ is regular.

We use this result to solve $\tilde{\mathbf{A}} \mathbf{x} = \tilde{\mathbf{b}}$ with $\tilde{\mathbf{A}}$ from (2.15) more efficiently than straightforward elimination, provided that the LU-factorisation $\mathbf{A} = \mathbf{L} \mathbf{U}$ (cost $\mathcal{O}(n^3)$) is already known. Applying Lemma 2.3.1 for $k = 1$, we get:

$$\tilde{\mathbf{x}} = (\mathbf{A}^{-1} \tilde{\mathbf{b}}) - (\mathbf{A}^{-1} \mathbf{u}) \frac{\mathbf{v}^H (\mathbf{A}^{-1} \tilde{\mathbf{b}})}{\mathbf{v}^H (\mathbf{A}^{-1} \mathbf{u}) + 1}. \quad (2.18)$$

If the LU factorization of \mathbf{A} is indeed available, computing $\tilde{\mathbf{x}}$ as in (2.18) has complexity $\mathcal{O}(n^2)$. Note that under some assumptions, the condition number of $\mathbf{I} + \mathbf{V}^H \mathbf{A}^{-1} \mathbf{U}$ can be bounded from above:

$$\text{cond}(\mathbf{I} + \mathbf{V}^H \mathbf{A}^{-1} \mathbf{U}) \leq \text{cond}(\mathbf{A}) \text{cond}(\mathbf{A} + \mathbf{U} \mathbf{V}^H).$$

This means that when the system matrices $\tilde{\mathbf{A}}$ and \mathbf{A} are well-conditioned, solving for $\mathbf{I} + \mathbf{V}^H \mathbf{A}^{-1} \mathbf{U}$ is also a well-conditioned problem.

Code Snippet 2.5: Solving a rank-1 modified LSE \rightarrow GITLAB

```

21 // Solving rank-1 updated LSE based on (2.18)
22 template <class LUDec>
23 VectorXd smw(const LUDec &lu, const MatrixXd &u, const VectorXd &v, const
    ↪ VectorXd &b) {
24     VectorXd z = lu.solve(b); //
25     VectorXd w = lu.solve(u); //
26     double alpha = 1.0 + v.dot(w);
27     if (std::abs(alpha) < std::numeric_limits<double>::epsilon())
28         throw std::runtime_error("A nearly singular");
29     else return (z - w * v.dot(z) / alpha);
30 }
31 /*

```

2.4 Sparse Linear Systems

In different applications, it happens that the system matrix has the special structure that many of its entries are zero. Examples of such applications include spline interpolation or solving a Poisson equation. Intuitively, one may question whether it makes sense to store the full matrix, i.e. with all its zero entries and whether it is necessary to carry out all operations in solving the system, that will involve many operations with zero elements.

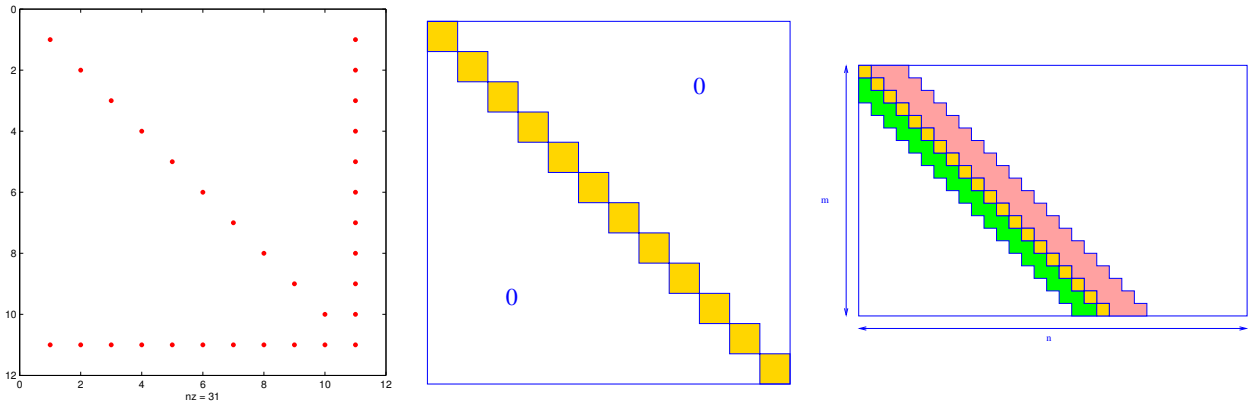
A matrix that has "many" zero entries is called *sparse*. We want to discuss how to store a sparse matrix in an efficient way (by only storing the non-zero entries and their locations). *Sparse solvers* can then be employed to operate effectively on sparse matrices.

Definition 2.4.1 (Sparse matrix). $\mathbf{A} \in \mathbb{K}^{m,n}$, $m, n \in \mathbb{N}$, is *sparse*, if

$$\text{nnz}(\mathbf{A}) := \#\{(i, j) \in \{1, \dots, m\} \times \{1, \dots, n\} : a_{ij} \neq 0\} \ll mn.$$

We write $\text{nnz}(\mathbf{A})$ for the number of non-zeros of \mathbf{A} .

Examples that we have already seen are "arrow matrices", diagonal matrices and banded matrices.



2.4.1 Sparse matrix storage formats

Sparse matrix storage formats for storing a sparse matrix $\mathbf{A} \in \mathbb{K}^{m,n}$ are designed to achieve two objectives:

- ① Amount of memory required is only slightly more than $\text{nnz}(\mathbf{A})$ scalars.
- ② Computational effort for matrix \times vector multiplication is proportional to $\text{nnz}(\mathbf{A})$.

In this section we see a few schemes used by numerical libraries.

Triplet/coordinate list (COO) format

This format simply stores a vector of *triplets* $(i, j, \alpha_{i,j})$, for the non-zero entries of a matrix \mathbf{A} at the (i,j) -th entry. Note that here, indexing starts at zero. A Triplet object in EIGEN can be initialized as follows:

Example 2.4.1:

```

1  unsigned int row_idx = 2;
2  unsigned int col_idx = 4;
3  double value = 2.5;
4  Eigen::Triplet<double> triplet(row_idx, col_idx, value);
5  std::cout << '(' << triplet.row() << ', ' << triplet.col()
6      << ', ' << triplet.value() << ')' << std::endl;

```

As shown, a Triplet object offers the access member functions `row()`, `col()`, and `value()` to fetch the row index, column index, and scalar value stored in a Triplet.

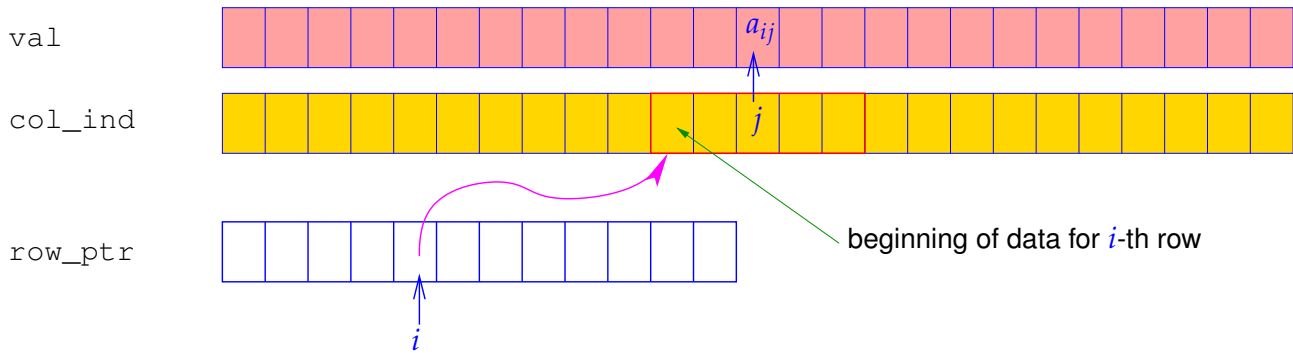
We note that in this format, *repetitions* of index pairs (i, j) are *allowed*. This needs some convention and the matrix entry $(\mathbf{A})_{i,j}$ is defined to be the sum of all values $\alpha_{i,j}$ associated with the index pair (i, j) .

Compressed row-storage/column-storage (CRS/CCS) format

Next, we introduce CRS/CCS format which is the default in EIGEN. The CRS format for a sparse matrix $\mathbf{A} = (a_{ij}) \in \mathbb{K}^{n,n}$ keeps the data in three *contiguous* arrays:

vector<scalar_t>	val	size $\text{nnz}(\mathbf{A}) := \#\{(i, j) \in \{1, \dots, n\}^2, a_{ij} \neq 0\}$
vector<size_t>	col_ind	size $\text{nnz}(\mathbf{A})$
vector<size_t>	row_ptr	size $n + 1$ & $\text{row_ptr}[n] = \text{nnz}(\mathbf{A})$ (sentinel value)

- ① val: Array of non-zero entries of the matrix (length = $\text{nnz}(\mathbf{A})$)
- ② col_ind: Column index vector: Contains the column index corresponding to each of the above entries (length = $\text{nnz}(\mathbf{A})$)
- ③ row_ptr: A vector that keeps track of the number of non-zeros in each row. It is defined as follows:
 $\text{row_ptr}[0] = 0,$
 $\text{row_ptr}[i] = \text{row_ptr}[i - 1] + \text{nnz}((i - 1)^{\text{th}} \text{ row})$
 (length = $m + 1$ for $\mathbf{A} \in \mathbb{K}^{m,n}$; $\text{row_ptr}[m] = \text{nnz}(\mathbf{A})$)



$$\mathbf{A} = \begin{bmatrix} 10 & 0 & 0 & 0 & -2 & 0 \\ 3 & 9 & 0 & 0 & 0 & 3 \\ 0 & 7 & 8 & 7 & 0 & 0 \\ 3 & 0 & 8 & 7 & 5 & 0 \\ 0 & 8 & 0 & 9 & 9 & 13 \\ 0 & 4 & 0 & 0 & 2 & -1 \end{bmatrix}$$

val-vector:

10	-2	3	9	3	7	8	7	3	...	9	13	4	2	-1
----	----	---	---	---	---	---	---	---	-----	---	----	---	---	----

 col_ind-array:

0	4	0	1	5	1	2	3	0	...	4	5	1	4	5
---	---	---	---	---	---	---	---	---	-----	---	---	---	---	---

 row_ptr-array:

0	2	5	8	12	16	19
---	---	---	---	----	----	----

The CCS format is equivalent to CRS format for the transposed matrix.

2.4.2 Sparse matrices in Eigen

Eigen can handle sparse matrices in the standard *Compressed Row Storage* (CRS) and *Compressed Column Storage* (CCS) format, see the documentation:

```

1 #include<Eigen/Sparse>
2 Eigen::SparseMatrix<int> Asp(rows,cols); // Default is CCS format
3 Eigen::SparseMatrix<double, Eigen::RowMajor> Bsp(rows,cols); // CRS format

```

Initialization of a sparse matrix:

```

1 Asp.insert(i,j) = v_ij; //insert non-existing new element at (i,j)
2 Asp.coeffRef(i,j) += w_ij; //update entry at (i,j)
3 Asp.makeCompressed(); //needed to construct CCS/CRS format after every
    insert/coefficient operation

```

Note. When inserting a new element there could possibly be multiple reallocations occurring in the background. The cost to insert a new element can be around the number of current non-zero entries at the time of insertion.

There are two ways to avoid this:

- ① The standard approach is to use triplet format for initialization and then change to CCS/CRS format:

2 Direct Methods for Linear Systems of Equations

```
1 std::vector <Eigen::Triplet <double > > triplets;
2 // .. fill the std::vector triplets ..
3 Eigen::SparseMatrix<double, Eigen::RowMajor> spMat(rows, cols);
4 spMat.setFromTriplets(triplets.begin(), triplets.end());
5 spMat.makeCompressed();
```

- ② Alternatively, one can “reserve” enough space in each row (if in row-major) for non-zero entries. However, this is only helpful when we have a good estimate of the number of non-zeros *for each row*:

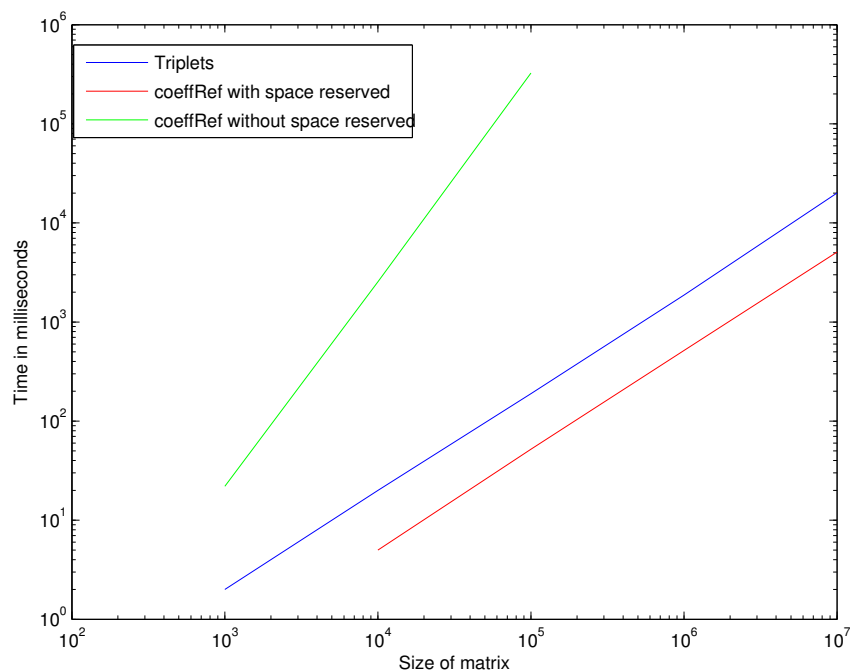
Code Snippet 2.6: Accessing entries of a sparse matrix: potentially inefficient!

```
1 unsigned int rows, cols, max_no_nnz_per_row;
2 .....
3 SparseMatrix<double, RowMajor> mat(rows, cols);
4 mat.reserve(RowVectorXi::Constant(cols, nr));
5 // do many (incremental) initializations
6 for ( ) {
7     mat.insert(i, j) = value_ij;
8     mat.coeffRef(i, j) += increment_ij;
9 }
10 mat.makeCompressed();
```

The usual matrix operations are supported for sparse matrices; addition and subtraction may involve only sparse matrices stored in the *same format*. These operations may incur large hidden costs and have to be used with care!

Example 2.4.2: Runtime of initialization of sparse matrices in Eigen

We study the runtime behavior of the initialization of a banded matrix with bandwidth 2 (i.e. 5 non-zero diagonals) where we use the methods described from before.



Runtimes (in ms) for the initialization of a banded matrix using different techniques in EIGEN.
 Green line: timing for entry-wise initialization with only 4 non-zero entries per row reserved in advance.
 (OS: Ubuntu Linux 14.04, CPU: Intel i5@1.80 Ghz, Compiler: g++-4.8.2, -O2)

Remark. Insufficient advance allocation of memory massively slows down the set-up of a sparse matrix in the case of direct entry-wise initialization.

Reason: Massive internal copying of data required to create space for “unexpected” entries.

Note. It is advisable to also reserve space for an estimated number of nonzeros in the triple format:

```

1 std::vector<Eigen::Triplet<double>> triplets;
2 triplets.reserve(nnz);
3 // .. fill the std::vector triplets
4 Eigen::SparseMatrix<double, Eigen::RowMajor> spMat(rows, cols);
5 spMat.setFromTriplets(triplets.begin(), triplets.end());

```

2.4.3 Direct solution of sparse LSEs

Sparse matrix solvers are very sophisticated algorithms and therefore one should always use available solvers. Most solvers from libraries accept both dense and sparse formats. Therefore only when the argument for the function call is of a sparse matrix format such as CCS/CRS will the algorithm exploit the structure to reduce the complexity by avoiding unnecessary computations. Their calling syntax remains unchanged, however:

```

1 Eigen::SolverType<Eigen::SparseMatrix<double>> solver(A);
2 Eigen::VectorXd x = solver.solve(b);

```

The standard sparse solver is SparseLU.

Code Snippet 2.7: Function for solving a sparse LSE with EIGEN → GITLAB

```

6 using SparseMatrix = Eigen::SparseMatrix<double>;
7 // Perform sparse elimination
8 void sparse_solve(const SparseMatrix& A, const VectorXd& b, VectorXd& x)
    ↪ {
9     Eigen::SparseLU<SparseMatrix> solver(A);
10    x = solver.solve(b);
11 }
12 /*

```

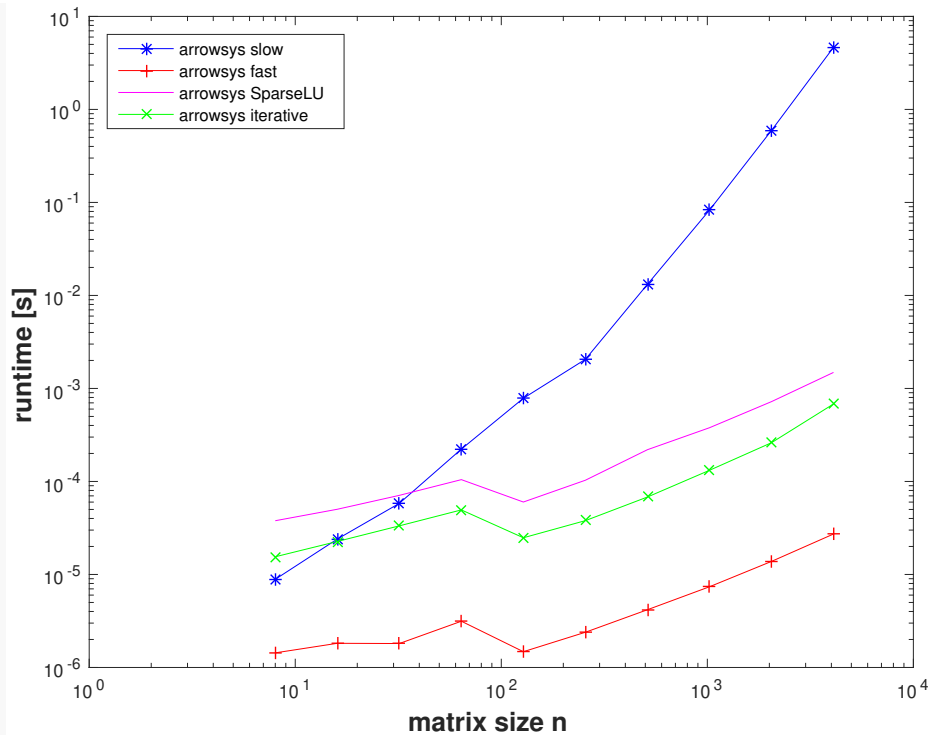
Example 2.4.3: LSE with arrow matrix

Code Snippet 2.8: Invoking sparse elimination solver for arrow matrix → GITLAB

```

17 template <class solver_t>
18 VectorXd arrowsys_sparse(const VectorXd &d, const VectorXd &c, const
    ↪ VectorXd &b, const double alpha, const VectorXd &y){
19     int n = d.size();
20     SparseMatrix<double> A(n+1, n+1); // default: column-major
21     VectorXi reserveVec = VectorXi::Constant(n+1, 2); // nnz per col
22     reserveVec(n) = n+1; // last full col
23     A.reserve(reserveVec);
24     for(int j = 0; j < n; ++j){ // initialize along cols for efficiency
25         A.insert(j, j) = d(j); // diagonal entries
26         A.insert(n, j) = b(j); // bottom row entries
27     }
28     for(int i = 0; i < n; ++i){
29         A.insert(i, n) = c(i); // last col
30     }
31     A.insert(n, n) = alpha; // bottomRight entry
32     A.makeCompressed();
33     return solver_t(A).solve(y);
34 }
35 /*

```



Remark. Observe that the sparse elimination solver is several orders of magnitude faster than `lu()` operating on a dense matrix. However, the sparse solver is still slower than `arrowsys_fast`. The reason is that it is a general algorithm that has to keep track of non-zero entries and has to be prepared to do pivoting.

Direct Methods for Linear Least Squares Problems

Suppose we want to solve a common problem in data science, which is the estimation of system parameters from a series of measurements. One example would be modelling causal relationships between parameters in biological systems. One approach could be to model the function f that takes inputs x_1, \dots, x_n and outputs a value y with a linear model:

$$f(\mathbf{x}) = a_1x_1 + \dots + a_nx_n, \quad f: \mathbb{R}^n \rightarrow \mathbb{R} \quad (3.1)$$

Given: Measured data points $(\mathbf{x}^k, y^k)_{k=1}^m$, $\mathbf{x}^k \in \mathbb{R}^n$, $y^k \in \mathbb{R}$, where (ideally) $\mathbf{x}^k \mapsto y^k = f(\mathbf{x}^k)$.

Goal: With this series of experiments estimate parameters a_1, \dots, a_n .

We can write this problem in matrix form:

$$\begin{bmatrix} x_1^1 & x_2^1 & \dots & x_n^1 \\ x_1^2 & x_2^2 & \dots & x_n^2 \\ \vdots & \vdots & \ddots & \vdots \\ x_1^m & x_2^m & \dots & x_n^m \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix} = \begin{bmatrix} y^1 \\ y^2 \\ \vdots \\ y^m \end{bmatrix}, \quad (3.2)$$

meaning that we conduct m trials stored rowwise in a matrix $\mathbf{X} \in \mathbb{R}^{m,n}$, outcomes stored in $\mathbf{y} \in \mathbb{R}^m$. Estimate parameters by solving

$$\mathbf{X}\mathbf{a} = \mathbf{y}, \quad \mathbf{a} \in \mathbb{R}^n.$$

3.1 Least Squares Solutions

Definition 3.1.1 (Least squares solution). For given $\mathbf{A} \in \mathbb{K}^{m,n}$ and $\mathbf{b} \in \mathbb{K}^m$, the vector $\mathbf{x} \in \mathbb{K}^n$ is a *least squares solution* of the linear system of equations $\mathbf{Ax} = \mathbf{b}$, if and only if

$$\mathbf{x} \in \operatorname{argmin}_{\mathbf{y} \in \mathbb{K}^n} \|\mathbf{Ay} - \mathbf{b}\|_2,$$

or equivalently

$$\|\mathbf{Ax} - \mathbf{b}\|_2 = \inf_{\mathbf{y} \in \mathbb{K}^n} \|\mathbf{Ay} - \mathbf{b}\|_2.$$

In other words, a least squares solution is any vector \mathbf{x} that minimizes the Euclidean norm of the residual $\mathbf{r} = \mathbf{b} - \mathbf{Ax}$.

Example 3.1.1:

In parameter estimation we look for parameters fulfilling $\mathbf{Xa} = \mathbf{y}$ which can be reformulated as the search for $\mathbf{a}^* = \operatorname{argmin}_{\mathbf{p} \in \mathbb{R}^n} \sum_{k=1}^m |(\mathbf{x}^k)^T \cdot \mathbf{p} - y^k|^2$.

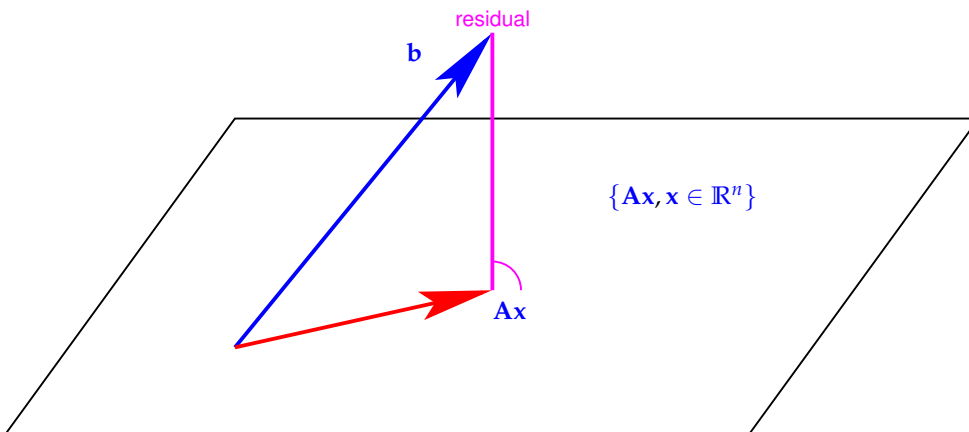
We write $\operatorname{lsq}(\mathbf{A}, \mathbf{b})$ for the set of least squares solutions of the linear system of equations $\mathbf{Ax} = \mathbf{b}$:

$$\operatorname{lsq}(\mathbf{A}, \mathbf{b}) := \{\mathbf{x} \in \mathbb{R}^n : \mathbf{x} \text{ is a least squares solution of } \mathbf{Ax} = \mathbf{b}\} \subset \mathbb{R}^n. \quad (3.5)$$

For a least squares solution $\mathbf{x} \in \mathbb{R}^n$, the vector $\mathbf{Ax} \in \mathbb{R}^m$ is the unique orthogonal projection of \mathbf{b} onto

$$\mathcal{R}(\mathbf{A}) = \operatorname{Span}\{(\mathbf{A})_{:,1}, \dots, (\mathbf{A})_{:,n}\},$$

because the orthogonal projection provides the nearest (w.r.t. the Euclidean distance) point to \mathbf{b} in the subspace (hyperplane) $\mathcal{R}(\mathbf{A})$. A geometric illustration for this argument is given in the following figure:



From this geometric consideration we conclude that $\text{lsq}(\mathbf{A}, \mathbf{b})$ is the space of solutions of $\mathbf{Ax} = \mathbf{b}^*$, where \mathbf{b}^* is the orthogonal projection of \mathbf{b} onto $\mathcal{R}(\mathbf{A})$. Since the set of solutions of a linear system of equations invariably is an affine space, this argument shows that $\text{lsq}(\mathbf{A}, \mathbf{b})$ is an affine subspace of \mathbb{R}^n .


Theorem 3.1.1 (Existence of least squares solutions). *For any $\mathbf{A} \in \mathbb{R}^{m,n}$, $\mathbf{b} \in \mathbb{R}^m$, a least squares solution of $\mathbf{Ax} = \mathbf{b}$ exists.*

3.1.1 Normal Equations

Recall from linear algebra:

Lemma 3.1.1 (Kernel and range of (Hermitian) transposed matrices). *For any matrix $\mathbf{A} \in \mathbb{K}^{m,n}$, the following holds:*

$$\mathcal{N}(\mathbf{A}) = \mathcal{R}(\mathbf{A}^H)^\perp, \quad \mathcal{N}(\mathbf{A})^\perp = \mathcal{R}(\mathbf{A}^H).$$

 Notation: *Orthogonal complement* of a subspace $V \subset \mathbb{K}^k$:

$$V^\perp := \{\mathbf{x} \in \mathbb{K}^k : \mathbf{x}^H \mathbf{y} = 0 \quad \forall \mathbf{y} \in V\}.$$

Appealing to the geometric intuition from before, we deduce that \mathbf{x} being a least squares solution is equivalent to $\mathbf{b} - \mathbf{Ax}$ being orthogonal to $\mathcal{R}(\mathbf{A})$. From Lemma 3.1.1, we can further conclude that this is equivalent to $\mathbf{b} - \mathbf{Ax} \in \mathcal{N}(\mathbf{A}^H)$. Thus, we can characterize least squares solutions as solutions to $\mathbf{A}^H \mathbf{Ax} = \mathbf{A}^H \mathbf{b}$.

Theorem 3.1.2 (Obtaining least squares solutions by solving normal equations). *The vector $\mathbf{x} \in \mathbb{R}^n$ is a least squares solution of the linear system of equations $\mathbf{Ax} = \mathbf{b}$, $\mathbf{A} \in \mathbb{R}^{m,n}$, $\mathbf{b} \in \mathbb{R}^m$, if and only if it solves the normal equations*

$$\mathbf{A}^\top \mathbf{Ax} = \mathbf{A}^\top \mathbf{b}. \tag{3.6}$$

Note that the normal equations (3.6) are an $n \times n$ square linear system of equations with a symmetric positive semi-definite coefficient matrix $\mathbf{A}^\top \mathbf{A}$:

$$\begin{bmatrix} \mathbf{A}^\top \end{bmatrix} \begin{bmatrix} \mathbf{A} \end{bmatrix} \begin{bmatrix} \mathbf{x} \end{bmatrix} = \begin{bmatrix} \mathbf{A}^\top \end{bmatrix} \begin{bmatrix} \mathbf{b} \end{bmatrix},$$

$$\iff \begin{bmatrix} \mathbf{A}^\top \mathbf{A} \end{bmatrix} \begin{bmatrix} \mathbf{x} \end{bmatrix} = \begin{bmatrix} \mathbf{A}^\top \end{bmatrix} \begin{bmatrix} \mathbf{b} \end{bmatrix}.$$

Theorem 3.1.2 together with Theorem 3.1.1 already confirms that the normal equations will always have a solution and that $\text{lsq}(\mathbf{A}, \mathbf{b})$ is a subspace of \mathbb{R}^n parallel to $\mathcal{N}(\mathbf{A}^\top \mathbf{A})$. The next theorem gives even more detailed information.

Theorem 3.1.3 (Kernel and range of $\mathbf{A}^\top \mathbf{A}$). *For $\mathbf{A} \in \mathbb{R}^{m,n}$, $m \geq n$, the following holds:*

$$\mathcal{N}(\mathbf{A}^\top \mathbf{A}) = \mathcal{N}(\mathbf{A}), \quad (3.7)$$

$$\mathcal{R}(\mathbf{A}^\top \mathbf{A}) = \mathcal{R}(\mathbf{A}^\top). \quad (3.8)$$

Note the following relations:

$$\begin{aligned} \text{least squares solution is unique} &\iff \mathcal{N}(\mathbf{A}^\top \mathbf{A}) = \{\mathbf{0}\} \\ &\iff \mathcal{N}(\mathbf{A}) = \{\mathbf{0}\} \\ &\iff \mathcal{R}(\mathbf{A}^\top) = \mathbb{R}^n \\ &\iff \text{rank}(\mathbf{A}) = n. \end{aligned}$$

The last property is also called *full-rank condition*.

Corollary 3.1.1 (Uniqueness of least squares solutions). *If $m \geq n$ and $\mathcal{N}(\mathbf{A}) = \{\mathbf{0}\}$, then the linear system of equations $\mathbf{Ax} = \mathbf{b}$, $\mathbf{A} \in \mathbb{R}^{m,n}$, $\mathbf{b} \in \mathbb{R}^m$, has a unique least squares solution that can be obtained by solving the normal equations (3.6), i.e.*

$$\mathbf{x} = (\mathbf{A}^\top \mathbf{A})^{-1} \mathbf{A}^\top \mathbf{b}.$$

Note that $\mathbf{A}^\top \mathbf{A}$ is symmetric positive definite if $\mathcal{N}(\mathbf{A}) = \{\mathbf{0}\}$.

3.1.2 Generalized Solutions & Moore-Penrose Pseudoinverse

As we have just seen, there can be many least squares solutions of $\mathbf{Ax} = \mathbf{b}$, in case $\mathcal{N}(\mathbf{A}) \neq \{\mathbf{0}\}$. We can impose another condition to single out a unique element of $\text{lsq}(\mathbf{A}, \mathbf{b})$:

Definition 3.1.2 (Generalized solution of a linear system of equations). The *generalized solution* $\mathbf{x}^\dagger \in \mathbb{R}^n$ of a linear system of equations $\mathbf{Ax} = \mathbf{b}$ is defined as

$$\mathbf{x}^\dagger := \operatorname{argmin}\{\|\mathbf{x}\|_2 : \mathbf{x} \in \operatorname{lsq}(\mathbf{A}, \mathbf{b})\} . \quad (3.9)$$

Thus, the generalized solution is the least squares solution with minimal norm.

Theorem 3.1.4. For any matrix $\mathbf{A} \in \mathbb{R}^{m,n}$ and right-hand side $\mathbf{b} \in \mathbb{R}^m$, the generalized solution \mathbf{x}^\dagger is unique.

Proof. Let \mathbf{x}^\dagger be a generalized solution. Clearly, since $\mathbb{R}^n = \mathcal{N}(\mathbf{A}) \oplus \mathcal{N}(\mathbf{A})^\perp$, we can write \mathbf{x}^\dagger as $\mathbf{x}^\dagger = \mathbf{x}_1 + \mathbf{x}_2$, where $\mathbf{x}_1 \in \mathcal{N}(\mathbf{A})^\perp$ and $\mathbf{x}_2 \in \mathcal{N}(\mathbf{A})$. Furthermore, $\|\mathbf{x}^\dagger\|_2^2 = \|\mathbf{x}_1\|_2^2 + \|\mathbf{x}_2\|_2^2$.

By definition,

$$\mathbf{A}^\top (\mathbf{Ax}^\dagger - \mathbf{b}) = \mathbf{0} .$$

Plugging in our decomposition of \mathbf{x}^\dagger we further obtain:

$$\begin{aligned} \mathbf{A}^\top (\mathbf{Ax}_1 + \mathbf{Ax}_2 - \mathbf{b}) &= \mathbf{0} \\ \implies \mathbf{A}^\top (\mathbf{Ax}_1 - \mathbf{b}) &= \mathbf{0}, \text{ since } \mathbf{x}_2 \in \mathcal{N}(\mathbf{A}) , \\ \implies \mathbf{x}_1 &\in \operatorname{lsq}(\mathbf{A}, \mathbf{b}) . \end{aligned}$$

Since it holds that $\|\mathbf{x}_1\|_2^2 \leq \|\mathbf{x}^\dagger\|_2^2 = \min \|\mathbf{Ax} - \mathbf{b}\|_2$, we have proven that $\mathbf{x}^\dagger \in \mathcal{N}(\mathbf{A})^\perp$.

Now, we show uniqueness. Suppose $\mathbf{x}_1^\dagger, \mathbf{x}_2^\dagger$ are generalized solutions, then using the previous result, we get:

$$\mathbf{x}_1^\dagger, \mathbf{x}_2^\dagger \in \mathcal{N}(\mathbf{A})^\perp \implies \mathbf{x}_1^\dagger - \mathbf{x}_2^\dagger \in \mathcal{N}(\mathbf{A})^\perp .$$

Also,

$$\mathbf{A}^\top \mathbf{A}(\mathbf{x}_1^\dagger - \mathbf{x}_2^\dagger) = \mathbf{0} \implies \mathbf{x}_1^\dagger - \mathbf{x}_2^\dagger \in \mathcal{N}(\mathbf{A}^\top \mathbf{A}) = \mathcal{N}(\mathbf{A}) .$$

Since $\mathbf{x}_1^\dagger - \mathbf{x}_2^\dagger$ is an element of both $\mathcal{N}(\mathbf{A})$ and $\mathcal{N}(\mathbf{A})^\perp$, we can conclude that $\mathbf{x}_1^\dagger = \mathbf{x}_2^\dagger$, thereby proving the uniqueness of the generalized solution. \square

As we have seen in the proof, the generalized solution \mathbf{x}^\dagger of $\mathbf{Ax} = \mathbf{b}$ is an element of $\mathcal{N}(\mathbf{A})^\perp$. Therefore, given a basis $\{\mathbf{v}_1, \dots, \mathbf{v}_k\} \subset \mathbb{R}^n$ of $\mathcal{N}(\mathbf{A})^\perp$, $k := \dim \mathcal{N}(\mathbf{A})^\perp$, we can find $\mathbf{y} \in \mathbb{R}^k$ such that $\mathbf{x}^\dagger = \mathbf{V}\mathbf{y}$, $\mathbf{V} := [\mathbf{v}_1, \dots, \mathbf{v}_k] \in \mathbb{R}^{n,k}$. Plugging this representation into the normal equations and multiplying with \mathbf{V}^\top yields the *reduced* normal equations:

$$\mathbf{V}^\top \mathbf{A}^\top \mathbf{A} \mathbf{V} \mathbf{y} = \mathbf{V}^\top \mathbf{A}^\top \mathbf{b} \quad (3.10)$$

$$\begin{array}{c}
 \begin{array}{c} \left[\begin{array}{c} \mathbf{V}^\top \end{array} \right] \left[\begin{array}{c} \mathbf{A}^\top \end{array} \right] \left[\begin{array}{c} \mathbf{A} \end{array} \right] \left[\begin{array}{c} \mathbf{V} \end{array} \right] \left[\begin{array}{c} \mathbf{y} \end{array} \right] = \\
 \updownarrow \\
 \left[\begin{array}{c} \mathbf{V}^\top \end{array} \right] \left[\begin{array}{c} \mathbf{A}^\top \end{array} \right] \left[\begin{array}{c} \mathbf{b} \end{array} \right] .
 \end{array}
 \end{array}$$

The very construction of \mathbf{V} ensures $\mathcal{N}(\mathbf{A}\mathbf{V}) = \{\mathbf{0}\}$, and hence $\mathcal{N}(\mathbf{V}^\top \mathbf{A}^\top \mathbf{A} \mathbf{V}) = \{\mathbf{0}\}$ so that the linear system of equations (3.10) has a unique solution. If \mathbf{y} is the unique solution of (3.10), then $\mathbf{x}^\dagger = \mathbf{V}\mathbf{y}$.

Theorem 3.1.5 (Formula for generalized solution). *Given $\mathbf{A} \in \mathbb{R}^{m,n}$, $\mathbf{b} \in \mathbb{R}^m$, the generalized solution \mathbf{x}^\dagger of the linear system of equations $\mathbf{A}\mathbf{x} = \mathbf{b}$ is given by*

$$\mathbf{x}^\dagger = \mathbf{V}(\mathbf{V}^\top \mathbf{A}^\top \mathbf{A} \mathbf{V})^{-1}(\mathbf{V}^\top \mathbf{A}^\top \mathbf{b}),$$

where \mathbf{V} is any matrix whose columns form a basis of $\mathcal{N}(\mathbf{A})^\perp$.

Note: The matrix $\mathbf{V}(\mathbf{V}^\top \mathbf{A}^\top \mathbf{A} \mathbf{V})^{-1} \mathbf{V}^\top$ is called the *Moore-Penrose pseudoinverse* \mathbf{A}^\dagger of \mathbf{A} and does not depend on the specific choice of \mathbf{V} .

3.2 Normal Equation Methods

Given $\mathbf{A} \in \mathbb{R}^{m,n}$, $m \geq n$, $\text{rank}(\mathbf{A}) = n$, $\mathbf{b} \in \mathbb{R}^m$, we introduce a first practical numerical method to determine the unique least squares solution of the overdetermined linear system of equations $\mathbf{A}\mathbf{x} = \mathbf{b}$.

In fact, Corollary 3.1.1 suggests a simple algorithm for solving linear least squares problems of the form (3.9) satisfying the full (maximal) rank condition $\text{rank}(\mathbf{A}) = n$: it boils down to solving the *normal equations* (3.6):

Algorithm: Normal equation method to solve full-rank least squares problem $\mathbf{A}\mathbf{x} = \mathbf{b}$

- ❶ Compute regular matrix $\mathbf{C} := \mathbf{A}^\top \mathbf{A} \in \mathbb{R}^{n,n}$
- ❷ Compute right hand side vector $\mathbf{c} := \mathbf{A}^\top \mathbf{b}$
- ❸ Solve symmetric positive definite linear system of equations: $\mathbf{C}\mathbf{x} = \mathbf{c}$

This can be done in EIGEN in a single line of code:

Code Snippet 3.1: Solving a linear least squares problem via normal equations

```

16 //! Solving the overdetermined linear system of equations
17 //! Ax = b by solving normal equations (3.6)
18 //! The least squares solution is returned by value
19 VectorXd normeqsolve(const MatrixXd &A, const VectorXd &b) {
20     if (b.size() != A.rows()) throw runtime_error("Dimension mismatch");
21     // Cholesky solver
22     VectorXd x = (A.transpose() * A).llt().solve(A.transpose() * b);
23     return x;
24 }
25 /*

```

Since $\mathbf{A}^\top \mathbf{A}$ is an s.p.d. matrix, Gaussian elimination remains stable even without pivoting. This is taken into account by requesting the Cholesky decomposition of $\mathbf{A}^\top \mathbf{A}$ by calling the method `llt()`. In Section 1.4.2 we discussed the asymptotic complexity of the operations involved in step ❶-❸ of the normal equation method:

$$\left. \begin{array}{ll} \text{step ❶:} & \text{cost } \mathcal{O}(mn^2) \\ \text{step ❷:} & \text{cost } \mathcal{O}(nm) \\ \text{step ❸:} & \text{cost } \mathcal{O}(n^3) \end{array} \right\} \Rightarrow \text{cost } \mathcal{O}(n^2m + n^3) \quad \text{for } m, n \rightarrow \infty.$$

Note that for small fixed n , $n \ll m$, $m \rightarrow \infty$ the computational effort scales linearly in m .

Note. By going from a system matrix \mathbf{A} to a system matrix $\mathbf{A}^\top \mathbf{A}$, the condition number squares: $\text{cond}_2(\mathbf{A}^\top \mathbf{A}) = \text{cond}_2(\mathbf{A})^2$.

Recall from Theorem 2.1.2: $\text{cond}_2(\mathbf{A}^\top \mathbf{A})$ governs amplification of (roundoff) errors in $\mathbf{A}^\top \mathbf{A}$ and $\mathbf{A}^\top \mathbf{b}$ when solving normal equations (3.6). If the size of $\text{cond}_2(\mathbf{A})$ is still acceptable, $\text{cond}_2(\mathbf{A}^\top \mathbf{A})$ might nevertheless be huge. Thus, using the normal equations (3.6) to numerically solve the linear least squares problem from Definition 3.1.1 may run the risk of huge amplification of roundoff errors incurred during the computation of the right hand side $\mathbf{A}^\top \mathbf{b}$: this implies potential instability of the normal equation approach.

Example 3.2.1: Roundoff effects in normal equations

Consider the following matrix \mathbf{A} and the corresponding matrix $\mathbf{A}^\top \mathbf{A}$

$$\mathbf{A} = \begin{bmatrix} 1 & 1 \\ \delta & 0 \\ 0 & \delta \end{bmatrix} \Rightarrow \mathbf{A}^\top \mathbf{A} = \begin{bmatrix} 1 + \delta^2 & 1 \\ 1 & 1 + \delta^2 \end{bmatrix}.$$

If δ is of the order of $\sqrt{\text{EPS}}$, e.g. $\delta = \sqrt{\text{EPS}}/2$, then $1 + \delta^2 = 1 + \text{EPS}/4$ is rounded to 1 in machine number arithmetic \mathbb{M} .

```

16 int main() {
17     MatrixXd A(3,2);
18     // Inquire about machine precision
19     double eps = std::numeric_limits<double>::epsilon();
20     // « initialization of matrix → section 1.3.1
21     A << 1, 1, sqrt(eps)/2, 0, 0, sqrt(eps)/2;
22     // Output rank of  $A^T A$ 
23     std::cout << "Rank of A: " << A.fullPivLu().rank() << std::endl
24               << "Rank of  $A^T A$ : "
25               << (A.transpose() * A).fullPivLu().rank() << std::endl;
26     return 0;
27 }
28 /*

```

Output:

```

1 Rank of A: 2
2 Rank of  $A^T A$ : 1

```

Hence the computed $A^T A$ as an element of $\mathbb{M}^{(2,2)}$ will fail to be regular. Note that $\text{rank}(A) = 2$, $\text{cond}_2(A) \approx \sqrt{\text{EPS}/2}$.

Further note:

A sparse $\not\Rightarrow A^T A$ sparse

Example 3.2.2:

Arrow matrices: Loss of sparsity

$$\begin{bmatrix} \text{diagonal} & \text{off-diagonal} \\ \text{off-diagonal} & \text{diagonal} \end{bmatrix} \begin{bmatrix} \text{diagonal} & \text{off-diagonal} \\ \text{off-diagonal} & \text{diagonal} \end{bmatrix} = \begin{bmatrix} \text{dense} \end{bmatrix}.$$

We summarize the challenges of the normal equations approach as follows:

- ① squaring of condition number

② possible loss of sparsity

A way to avoid the computation of $\mathbf{A}^\top \mathbf{A}$ is to extend normal equations (3.6) by introducing the *residual* $\mathbf{r} := \mathbf{Ax} - \mathbf{b}$ as a new unknown to maintain sparsity:

$$\mathbf{A}^H \mathbf{Ax} = \mathbf{A}^H \mathbf{b} \Leftrightarrow \mathbf{B} \begin{bmatrix} \mathbf{r} \\ \mathbf{x} \end{bmatrix} := \begin{bmatrix} -\mathbf{I} & \mathbf{A} \\ \mathbf{A}^H & 0 \end{bmatrix} \begin{bmatrix} \mathbf{r} \\ \mathbf{x} \end{bmatrix} = \begin{bmatrix} \mathbf{b} \\ 0 \end{bmatrix}. \quad (3.11)$$

The benefit of using the *augmented* system (3.11) instead of the standard normal equations (3.6) is that sparsity is preserved. However, the conditioning of the system matrix in (3.11) is not better than that of $\mathbf{A}^\top \mathbf{A}$. A more general substitution $\mathbf{r} := \alpha^{-1}(\mathbf{Ax} - \mathbf{b})$ with $\alpha > 0$ may improve the conditioning for a suitably chosen parameter α

$$\mathbf{A}^H \mathbf{Ax} = \mathbf{A}^H \mathbf{b} \Leftrightarrow \mathbf{B}_\alpha \begin{bmatrix} \mathbf{r} \\ \mathbf{x} \end{bmatrix} := \begin{bmatrix} -\alpha \mathbf{I} & \mathbf{A} \\ \mathbf{A}^H & 0 \end{bmatrix} \begin{bmatrix} \mathbf{r} \\ \mathbf{x} \end{bmatrix} = \begin{bmatrix} \mathbf{b} \\ 0 \end{bmatrix}. \quad (3.12)$$

For $m, n \gg 1$, \mathbf{A} sparse, both (3.11) and (3.12) lead to large sparse linear systems of equations, amenable to sparse direct elimination techniques.

3.3 Orthogonal Transformation Methods

We consider the full-rank linear least squares problem:

$$\text{Given } \mathbf{A} \in \mathbb{R}^{m,n}, \mathbf{b} \in \mathbb{R}^m, \text{ find } \mathbf{x} = \underset{\mathbf{y} \in \mathbb{R}^n}{\operatorname{argmin}} \|\mathbf{Ay} - \mathbf{b}\|_2. \quad (3.9)$$

Here, we assume that $m \geq n$ and \mathbf{A} has full rank, i.e. $\operatorname{rank}(\mathbf{A}) = n$.

Idea: Instead of solving $\mathbf{Ax} = \mathbf{b}$ find $\tilde{\mathbf{A}}, \tilde{\mathbf{b}}$ with $\operatorname{lsq}(\tilde{\mathbf{A}}, \tilde{\mathbf{b}}) = \operatorname{lsq}(\mathbf{A}, \mathbf{b})$ such that $\tilde{\mathbf{A}}\mathbf{x} = \tilde{\mathbf{b}}$ is easier to solve.

Two questions: ❶ Which linear least squares problems are “easy to solve” ?

❷ How can we obtain them by *equivalent transformations* of (3.9) ?

Here we call two overdetermined linear systems $\mathbf{Ax} = \mathbf{b}$ and $\tilde{\mathbf{A}}\mathbf{x} = \tilde{\mathbf{b}}$ equivalent in the sense of (3.9), if both have the same set of least squares solutions: $\operatorname{lsq}(\mathbf{A}, \mathbf{b}) = \operatorname{lsq}(\tilde{\mathbf{A}}, \tilde{\mathbf{b}})$.

Answer to question ❶:

As for LSE: Linear least squares problems with upper *triangular* system matrices are easy to solve.

$$\left\| \begin{bmatrix} \text{yellow staircase} \\ \mathbf{R} \\ \mathbf{0} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} - \begin{bmatrix} b_1 \\ \vdots \\ b_m \end{bmatrix} \right\|_2 \rightarrow \min \xRightarrow{(*)} \mathbf{x} = \begin{bmatrix} \text{yellow staircase} \\ \mathbf{R} \end{bmatrix}^{-1} \begin{bmatrix} b_1 \\ \vdots \\ \vdots \\ b_n \end{bmatrix}.$$

How can we draw the conclusion $(*)$? The components $n+1, \dots, m$ of the vector inside the norm are fixed and do not depend on \mathbf{x} . All we can do is to make the first components $1, \dots, n$ vanish, by choosing a suitable \mathbf{x} . Obviously, $\mathbf{x} = \mathbf{R}^{-1}(\mathbf{b})_{1:n}$ accomplishes this.

Note. Since \mathbf{A} has full rank n , the upper triangular part $\mathbf{R} \in \mathbb{R}^{n,n}$ of \mathbf{A} is regular!

Answer to question ②:

Idea: If we have a (transformation) matrix $\mathbf{T} \in \mathbb{R}^{m,m}$ satisfying

$$\|\mathbf{T}\mathbf{y}\|_2 = \|\mathbf{y}\|_2 \quad \forall \mathbf{y} \in \mathbb{R}^m, \quad (3.13)$$

$$\text{then} \quad \operatorname{argmin}_{\mathbf{y} \in \mathbb{R}^n} \|\mathbf{A}\mathbf{y} - \mathbf{b}\|_2 = \operatorname{argmin}_{\mathbf{y} \in \mathbb{R}^n} \|\tilde{\mathbf{A}}\mathbf{y} - \tilde{\mathbf{b}}\|_2,$$

where $\tilde{\mathbf{A}} = \mathbf{T}\mathbf{A}$ and $\tilde{\mathbf{b}} = \mathbf{T}\mathbf{b}$. Transformations fulfilling (3.13) are called orthogonal/unitary matrices.

Definition 3.3.1 (Unitary and orthogonal matrices).

- A matrix $\mathbf{Q} \in \mathbb{K}^{n,n}$, $n \in \mathbb{N}$, is *unitary*, if and only if $\mathbf{Q}^{-1} = \mathbf{Q}^H$.
- A matrix $\mathbf{Q} \in \mathbb{R}^{n,n}$, $n \in \mathbb{N}$, is *orthogonal*, if and only if $\mathbf{Q}^{-1} = \mathbf{Q}^\top$.

Theorem 3.3.1 (Preservation of Euclidean norm). *A matrix is unitary/orthogonal, if and only if the associated linear mapping preserves the 2-norm:*

$$\mathbf{Q} \in \mathbb{K}^{n,n} \quad \text{unitary} \quad \Leftrightarrow \quad \|\mathbf{Q}\mathbf{x}\|_2 = \|\mathbf{x}\|_2 \quad \forall \mathbf{x} \in \mathbb{K}^n.$$

From Theorem 3.3.1 we immediately conclude that, if a matrix $\mathbf{Q} \in \mathbb{K}^{n,n}$ is unitary/orthogonal, then

- all rows/columns (regarded as vectors $\in \mathbb{K}^n$) have Euclidean norm equal to 1,
- all rows/columns are pairwise orthogonal (w.r.t. Euclidean inner product),
- $|\det \mathbf{Q}| = 1$, $\|\mathbf{Q}\|_2 = 1$, and all eigenvalues are elements of $\{z \in \mathbb{C}: |z| = 1\}$.
- $\|\mathbf{QA}\|_2 = \|\mathbf{A}\|_2$ for any matrix $\mathbf{A} \in \mathbb{K}^{n,m}$.

3.3.1 QR-Decomposition

This section will answer the question whether and how it is possible to find orthogonal transformations that convert any given matrix $\mathbf{A} \in \mathbb{R}^{m,n}$, $m \geq n$, $\text{rank}(\mathbf{A}) = n$, to upper triangular form, as required for the application of the “equivalence transformation idea”.

First approach: Gram-Schmidt orthogonalization

Input: $\{\mathbf{a}^1, \dots, \mathbf{a}^n\} \subset \mathbb{R}^n$ linearly independent vectors

Output: orthonormal system $\{\mathbf{q}^1, \dots, \mathbf{q}^n\}$ (assuming no early termination)

```

1:   $\mathbf{q}^1 := \frac{\mathbf{a}^1}{\|\mathbf{a}^1\|_2}$   % 1st output vector
2:  for  $j = 2, \dots, n$  do
      { % Orthogonal projection
3:     $\mathbf{q}^j := \mathbf{a}^j$ 
4:    for  $\ell = 1, 2, \dots, j-1$  do
5:      {  $\mathbf{q}^j \leftarrow \mathbf{q}^j - \langle \mathbf{a}^j, \mathbf{q}^\ell \rangle \mathbf{q}^\ell$  }
6:      if ( $\mathbf{q}^j = \mathbf{0}$ ) then STOP
7:      else {  $\mathbf{q}^j \leftarrow \frac{\mathbf{q}^j}{\|\mathbf{q}^j\|_2}$  }
8:    }
```

(GS)

Theorem 3.3.2 (Span property of G.S. vectors). *If $\{\mathbf{a}^1, \dots, \mathbf{a}^n\}$ is linearly independent, then the Gram-Schmidt algorithm computes orthonormal vectors $\mathbf{q}^1, \dots, \mathbf{q}^n$ satisfying*

$$\text{Span}\{\mathbf{q}^1, \dots, \mathbf{q}^\ell\} = \text{Span}\{\mathbf{a}^1, \dots, \mathbf{a}^\ell\},$$

for all $\ell \in \{1, \dots, n\}$.

The transformations of the columns of \mathbf{A} through the G.S. algorithm can be written successively in matrix form. For a matrix $\mathbf{A} = [\mathbf{a}^1, \mathbf{a}^2, \dots, \mathbf{a}^n] \in \mathbb{R}^{m,n}$:

Step 1:

$$\left[\begin{array}{c|c|c|c|c} | & | & & & | \\ \mathbf{a}^1 & \mathbf{a}^2 & \dots & & \mathbf{a}^n \\ | & | & & & | \end{array} \right] \begin{bmatrix} t_{11} & 0 & \dots & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ \vdots & 0 & 1 & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & 0 \\ 0 & 0 & \dots & 0 & 1 \end{bmatrix} = \left[\begin{array}{c|c|c|c|c} | & | & & & | \\ t_{11}\mathbf{a}^1 & \mathbf{a}^2 & \dots & & \mathbf{a}^n \\ | & | & & & | \end{array} \right]$$

Step 2:

$$\left[\begin{array}{c|c|c|c|c} | & | & & & | \\ t_{11}\mathbf{a}^1 & \mathbf{a}^2 & \dots & & \mathbf{a}^n \\ | & | & & & | \end{array} \right] \begin{bmatrix} 1 & \tilde{t}_{12} & 0 & \dots & 0 \\ 0 & t_{22} & 0 & \dots & 0 \\ \vdots & 0 & 1 & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & \dots & 0 & 1 \end{bmatrix} \underbrace{=}_{t_{12}=t_{11}\tilde{t}_{12}} \left[\begin{array}{c|c|c|c|c} | & | & & & | \\ t_{11}\mathbf{a}^1 & t_{12}\mathbf{a}^1 + t_{22}\mathbf{a}^2 & \dots & & \mathbf{a}^n \\ | & | & & & | \end{array} \right]$$

Altogether: We apply a series of multiplications from the right by upper triangular matrices to obtain an orthogonal matrix:

$$\mathbf{Q} = \mathbf{A} \underbrace{\mathbf{T}_1 \mathbf{T}_2 \dots \mathbf{T}_n}_{:=\mathbf{T}}.$$

$\mathbf{Q} = [\mathbf{q}^1, \dots, \mathbf{q}^n] \in \mathbb{R}^{m,n}$ is a matrix with orthonormal columns such that

$$\begin{aligned} \mathbf{q}^1 &= t_{11}\mathbf{a}^1 \\ \mathbf{q}^2 &= t_{12}\mathbf{a}^1 + t_{22}\mathbf{a}^2 \\ \mathbf{q}^3 &= t_{13}\mathbf{a}^1 + t_{23}\mathbf{a}^2 + t_{33}\mathbf{a}^3 \\ &\vdots \\ \mathbf{q}^n &= t_{1n}\mathbf{a}^1 + t_{2n}\mathbf{a}^2 + \dots + t_{nn}\mathbf{a}^n. \end{aligned}$$

The matrix $\mathbf{T} \in \mathbb{R}^{n,n}$ is regular because $\{\mathbf{a}^1, \dots, \mathbf{a}^n\}$ and $\{\mathbf{q}^1, \dots, \mathbf{q}^n\}$ are linearly independent sets.

Recall from linear algebra that the inverse of a regular upper triangular matrix is again upper triangular.

$$\left[\begin{array}{c|c|c|c|c|c|c} \text{yellow} & & & & & & \\ & \text{yellow} & & & & & \\ & & \text{yellow} & & & & \\ & & & \text{yellow} & & & \\ & & & & \text{yellow} & & \\ & & & & & \text{yellow} & \\ & & & & & & \text{yellow} \end{array} \right]^{-1} = \left[\begin{array}{c|c|c|c|c|c|c} \text{yellow} & & & & & & \\ & \text{yellow} & & & & & \\ & & \text{yellow} & & & & \\ & & & \text{yellow} & & & \\ & & & & \text{yellow} & & \\ & & & & & \text{yellow} & \\ & & & & & & \text{yellow} \end{array} \right]$$

3 Direct Methods for Linear Least Squares Problems

Thus, we have found an *upper triangular* matrix $\mathbf{R} := \mathbf{T}^{-1} \in \mathbb{R}^{n,n}$ such that

$$\mathbf{A} = \mathbf{Q}\mathbf{R} \Leftrightarrow \left[\begin{array}{c} \mathbf{A} \end{array} \right] = \left[\begin{array}{c} \mathbf{Q} \end{array} \right] \left[\begin{array}{c} \mathbf{R} \end{array} \right].$$

Augmentation: Next we add $m - n$ zero rows at the bottom of \mathbf{R} and complement columns of \mathbf{Q} to an orthonormal basis of \mathbb{R}^m , which yields an orthogonal matrix $\tilde{\mathbf{Q}} \in \mathbb{R}^{m,m}$:

$$\mathbf{A} = \tilde{\mathbf{Q}} \begin{bmatrix} \mathbf{R} \\ 0 \end{bmatrix}$$

$$\Leftrightarrow \left[\begin{array}{c} \mathbf{A} \end{array} \right] = \left[\begin{array}{c} \tilde{\mathbf{Q}} \end{array} \right] \left[\begin{array}{c} \mathbf{R} \\ 0 \end{array} \right]$$

$$\Leftrightarrow \tilde{\mathbf{Q}}^\top \mathbf{A} = \begin{bmatrix} \mathbf{R} \\ 0 \end{bmatrix}.$$

Thus we have shown through Gram-Schmidt orthonormalization how to orthogonalize a matrix. This is summarized in the following theorem:

Theorem 3.3.3 (QR-decomposition). *For any matrix $\mathbf{A} \in \mathbb{K}^{n,k}$ with $\text{rank}(\mathbf{A}) = k$, there exists*

- (i) *a unique matrix $\mathbf{Q}_0 \in \mathbb{R}^{n,k}$ that satisfies $\mathbf{Q}_0^H \mathbf{Q}_0 = \mathbf{I}_k$, and a unique upper triangular matrix*

$\mathbf{R}_0 \in \mathbb{K}^{k,k}$ with $(\mathbf{R}_0)_{i,i} > 0, i \in \{1, \dots, k\}$, such that

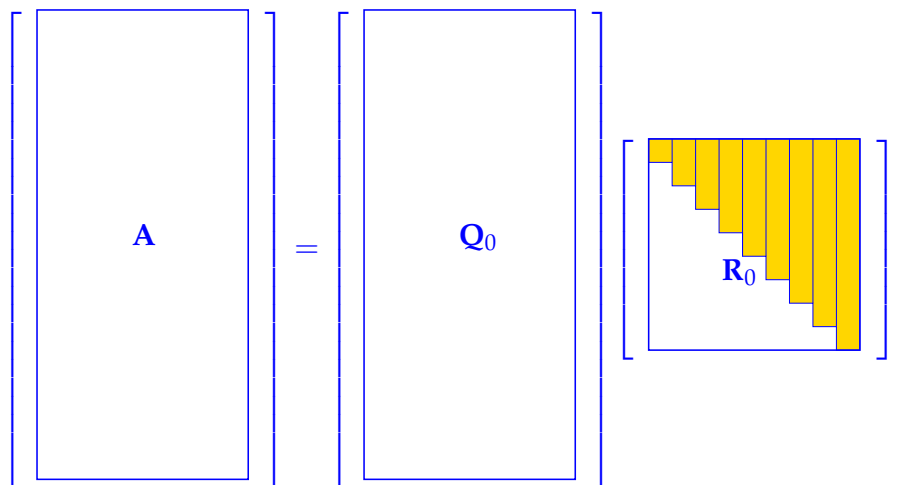
$$\mathbf{A} = \mathbf{Q}_0 \cdot \mathbf{R}_0 \quad (\text{"economical/thin/reduced" QR-decomposition}),$$

(ii) a unitary matrix $\mathbf{Q} \in \mathbb{K}^{n,n}$ and a unique upper triangular matrix $\mathbf{R} \in \mathbb{K}^{n,k}$ with $(\mathbf{R})_{i,i} > 0, i \in \{1, \dots, n\}$, such that

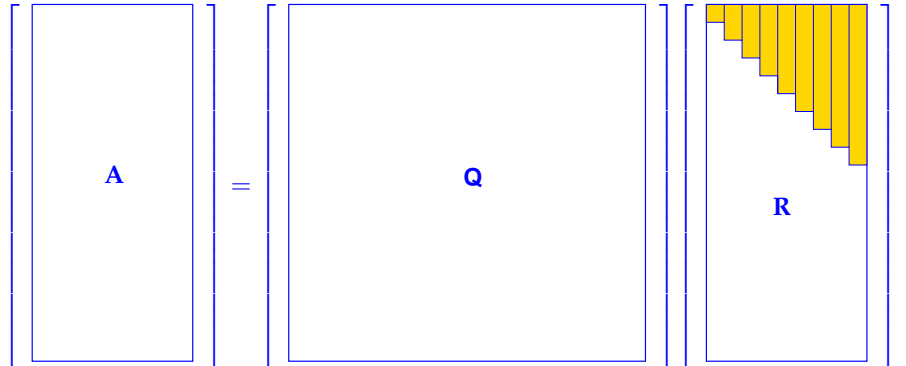
$$\mathbf{A} = \mathbf{Q} \cdot \mathbf{R} \quad (\text{full QR-decomposition}).$$

If $\mathbb{K} = \mathbb{R}$, all matrices will be real and \mathbf{Q} will be orthogonal.

Visualisation: Reduced QR-decomposition: $\mathbf{Q}_0^H \mathbf{Q}_0 = \mathbf{I}_k$ (orthonormal columns):

$$\mathbf{A} = \mathbf{Q}_0 \mathbf{R}_0, \quad \mathbf{Q}_0 \in \mathbb{K}^{n,k}, \quad \mathbf{R}_0 \in \mathbb{K}^{k,k} \text{ upper triangular,}$$


Visualisation: full QR-decomposition: $\mathbf{Q}^H \mathbf{Q} = \mathbf{Q} \mathbf{Q}^H = \mathbf{I}_n$ (orthogonal matrix):

$$\mathbf{A} = \mathbf{Q} \mathbf{R}, \quad \mathbf{Q} \in \mathbb{K}^{n,n}, \quad \mathbf{R} \in \mathbb{K}^{n,k},$$


Note. Gram-Schmidt orthogonalisation suffers from numerical instabilities.

Example 3.3.1:

Suppose we want to orthonormalize the following set of two vectors:

$$\mathbf{a}_1 = \begin{pmatrix} 1 \\ 1 \end{pmatrix} \quad \mathbf{a}_2 = \begin{pmatrix} 1 + \epsilon \\ 1 \end{pmatrix} \quad \epsilon \ll 1.$$

We know that theoretically,

$$\text{Span}\{\mathbf{a}_1, \mathbf{a}_2\} = \text{Span}\left\{\begin{pmatrix} 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \end{pmatrix}\right\}.$$

But Gram-Schmidt gives

$$\begin{aligned} \mathbf{q}_1 &= \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ 1 \end{pmatrix}, \quad \text{proj}_{\mathbf{q}_1} \mathbf{a}_2 = \langle \mathbf{a}_2, \mathbf{q}_1 \rangle \mathbf{q}_1 = \frac{2 + \epsilon}{2} \begin{pmatrix} 1 \\ 1 \end{pmatrix} \\ \Rightarrow \tilde{\mathbf{q}}_2 &= \begin{pmatrix} 1 + \epsilon \\ 1 \end{pmatrix} - \frac{2 + \epsilon}{2} \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \frac{\epsilon}{2} \begin{pmatrix} 1 \\ -1 \end{pmatrix} \end{aligned}$$

Note that the last step involves subtraction of close by numbers. Hence, $\|\tilde{\mathbf{q}}_2\|_2 = \frac{\epsilon}{2}$ so that computing $\mathbf{q}_2 = \frac{\tilde{\mathbf{q}}_2}{\|\tilde{\mathbf{q}}_2\|_2}$ involves division by a small number of the order of ϵ .

Computation of QR-Decomposition

Idea: So far: manipulation of columns of \mathbf{A} by multiplication from the right [with triangular matrices], i.e. $\mathbf{Q} = \mathbf{A} \mathbf{T}_1 \dots \mathbf{T}_n$.

Now: find a series of orthogonal transformations such that when applied from the left yield triangular matrix, i.e. $\mathbf{Q}_1 \dots \mathbf{Q}_n \mathbf{A} = \mathbf{R}$.

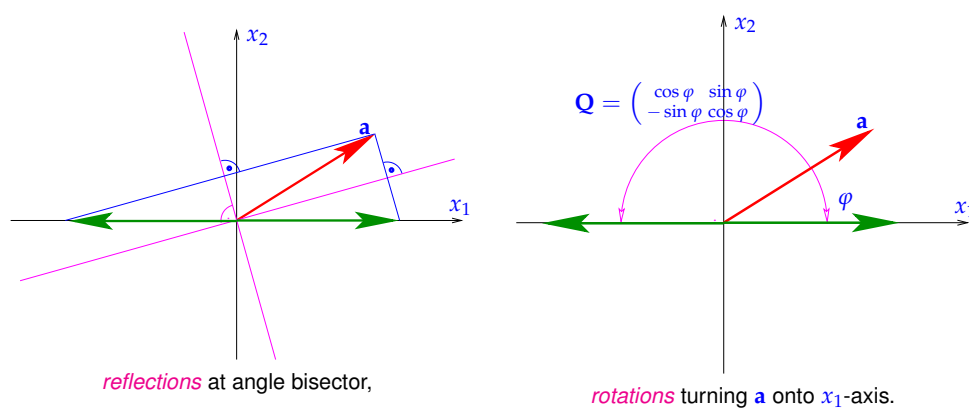
Then the product $\mathbf{Q} := \mathbf{Q}_1^\top \dots \mathbf{Q}_n^\top$ is also orthogonal and yields $\mathbf{A} = \mathbf{Q} \cdot \mathbf{R}$. This idea is similar to Gauss elimination but now with orthogonal transformations.

Idea: find simple unitary/orthogonal (row) transformations rendering certain matrix elements zero:

$$Q \begin{pmatrix} \text{yellow square} \end{pmatrix} = \begin{bmatrix} \text{yellow square} \\ 0 \end{bmatrix} \quad \text{with} \quad Q^H = Q^{-1}.$$

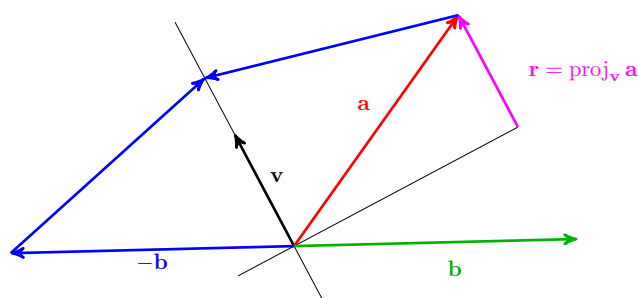
3.3.2 Householder Transformations

Intuitively the set of orthogonal transformations consists of maps that preserve the length of vectors and the angles between vectors, i.e. orthogonal transformations merely rotate and reflect vectors. In the following, we will consider reflections only and see that they may be represented by projections.



Householder Reflections

For a given a vector $\mathbf{a} \in \mathbb{R}^m$, we consider its reflection at a hyperplane with a normal vector $\mathbf{v} \in \mathbb{R}^m$. The resulting reflection is denoted by $\mathbf{b} \in \mathbb{R}^m$.



3 Direct Methods for Linear Least Squares Problems

The vector \mathbf{b} can be expressed as

$$\mathbf{b} = \mathbf{a} - 2\mathbf{r} = \mathbf{a} - 2(\text{proj}_{\mathbf{v}} \mathbf{a}) = \mathbf{a} - 2 \frac{\mathbf{v}^\top \mathbf{a}}{\mathbf{v}^\top \mathbf{v}} \mathbf{v}.$$

Here we use the same formal rewriting as in the Gram-Schmidt orthogonalisation with $\|\mathbf{v}\|_2^2 = \mathbf{v}^\top \mathbf{v}$.

Note that since $\mathbf{v}^\top \mathbf{a}$ is a scalar, one can rewrite $\mathbf{v}^\top \mathbf{a} \mathbf{v} = \mathbf{v} \mathbf{v}^\top \mathbf{a}$, so that

$$\mathbf{b} = \mathbf{a} - 2 \frac{\mathbf{v} \mathbf{v}^\top}{\mathbf{v}^\top \mathbf{v}} \mathbf{a} = \left(\mathbf{I}_m - 2 \frac{\mathbf{v} \mathbf{v}^\top}{\mathbf{v}^\top \mathbf{v}} \right) \mathbf{a},$$

where \mathbf{I}_m stands for the identity matrix of size $m \times m$.

Therefore we can define the *Householder matrix* $\mathbf{H}_{\mathbf{v}}$ that performs reflection with respect to the hyperplane with normal vector \mathbf{v} as:

$$\mathbf{H}_{\mathbf{v}} := \mathbf{I}_m - 2 \frac{\mathbf{v} \mathbf{v}^\top}{\mathbf{v}^\top \mathbf{v}}. \quad (3.14)$$

In our previous notation, this implies $\mathbf{H}_{\mathbf{v}} \mathbf{a} = \mathbf{b}$. The idea of using Householder reflections to obtain a QR decomposition is that by successive application of Householder reflections we obtain an upper triangular matrix \mathbf{R} . In the first step, this means that

$$\mathbf{H}_{\mathbf{v}^1} \mathbf{a}^1 = c_1 \mathbf{e}^1 = \begin{pmatrix} c_1 \\ 0 \\ \vdots \\ 0 \end{pmatrix}, \text{ for } \mathbf{a}^1, \text{ the first column of } \mathbf{A}, \text{ and } \mathbf{e}^j, \text{ the } j\text{-th unit vector.}$$

This can be rewritten as:

$$\mathbf{H}_{\mathbf{v}^1} \mathbf{A} = \begin{bmatrix} c_1 & \boxed{\phantom{\mathbf{A}_{12} \dots \mathbf{A}_{1n}}} \\ 0 & \boxed{\phantom{\mathbf{A}_{22} \dots \mathbf{A}_{2n}}} \\ \vdots & \boxed{\phantom{\mathbf{A}_{32} \dots \mathbf{A}_{3n}}} \\ 0 & \boxed{\phantom{\mathbf{A}_{n2} \dots \mathbf{A}_{nn}}} \end{bmatrix}.$$

Thus, we need to find a constant c_1 such that $\mathbf{a}^1 - 2 \frac{\mathbf{v}^\top \mathbf{a}^1}{\mathbf{v}^\top \mathbf{v}} \mathbf{v} = c_1 \mathbf{e}^1$.

We deduce

$$\mathbf{v} = (\mathbf{a}^1 - c_1 \mathbf{e}^1) \frac{\mathbf{v}^\top \mathbf{v}}{2 \mathbf{v}^\top \mathbf{a}^1},$$

thus \mathbf{v} must be parallel to $\mathbf{a}^1 - c_1 \mathbf{e}^1$.

We note that the scaling of \mathbf{v} does not affect the formula and set:

$$\mathbf{v} = \mathbf{a}^1 - c_1 \mathbf{e}^1.$$

With this choice, one obtains

$$\frac{\mathbf{v}^\top \mathbf{v}}{2\mathbf{v}^\top \mathbf{a}^1} = 1$$

and hence

$$\begin{aligned} \|\mathbf{a}^1 - c_1 \mathbf{e}^1\|_2^2 &= 2(\mathbf{a}^1 - c_1 \mathbf{e}^1)^\top \mathbf{a}^1 \\ \iff \|\mathbf{a}^1\|_2^2 - 2c_1 \mathbf{a}^{1\top} \mathbf{e}^1 + c_1^2 &= 2\|\mathbf{a}^1\|_2^2 - 2c_1 \mathbf{a}^{1\top} \mathbf{e}^1 \\ \iff c_1^2 &= \|\mathbf{a}^1\|_2^2 \end{aligned}$$

Thus, $c_1 = \pm \|\mathbf{a}^1\|_2$.

The computation of the vector \mathbf{v} can be prone to cancellation, if the vector \mathbf{a}^1 encloses a very small angle with the first unit vector, because in this case \mathbf{v} can be very small and beset with a huge relative error. This is a concern, because in the formula for the Householder matrix, \mathbf{v} is normalized to unit length (division by $\|\mathbf{v}\|_2$). Fortunately, as we have just derived, two choices for \mathbf{v} are possible and at most one can be affected by cancellation. The right choice is

$$\mathbf{v} = \begin{cases} \frac{1}{2}(\mathbf{a}^1 - \|\mathbf{a}^1\|_2 \mathbf{e}^1) & \text{if } (\mathbf{a}^1)_1 < 0, \\ \frac{1}{2}(\mathbf{a}^1 + \|\mathbf{a}^1\|_2 \mathbf{e}^1) & \text{if } (\mathbf{a}^1)_1 > 0. \end{cases}$$

Suppose we want to find the j -th Householder reflection in our transformation. This amounts to finding a Householder reflection $\mathbf{H}_{\mathbf{v}^j}$ such that:

$$\mathbf{H}_{\mathbf{v}^j} \tilde{\mathbf{a}}^j = \begin{bmatrix} \mathbf{r}_1^j \\ 0 \\ \vdots \\ 0 \end{bmatrix} \text{ with } m-j \text{ zeros and } \mathbf{r}_1^j \in \mathbb{R}^j.$$

Here, $\tilde{\mathbf{a}}^j$ is the j -th column of the transformed matrix $\tilde{\mathbf{A}}$ that we obtained after $j-1$ Householder reflections on \mathbf{A} . Vector $\tilde{\mathbf{a}}^j$ is split into: $\begin{bmatrix} \tilde{\mathbf{a}}_1^j \\ \tilde{\mathbf{a}}_2^j \end{bmatrix}$, where $\tilde{\mathbf{a}}_1^j \in \mathbb{R}^{j-1}$ and $\tilde{\mathbf{a}}_2^j \in \mathbb{R}^{m-j+1}$.

Choose $\mathbf{v}^j = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ \tilde{\mathbf{a}}_2^j \end{bmatrix} + c_j \mathbf{e}^j$ with $c_j = \pm \|\tilde{\mathbf{a}}_2^j\|$. The sign of c_j is chosen depending on the sign

of $(\tilde{\mathbf{a}}_2^j)_1$ so as to prevent cancellation.

3 Direct Methods for Linear Least Squares Problems

We can compute $\mathbf{H}_{\mathbf{v}^j} \tilde{\mathbf{a}}^j$ as follows:

$$\mathbf{H}_{\mathbf{v}^j} \tilde{\mathbf{a}}^j = \tilde{\mathbf{a}}^j - 2 \frac{(\tilde{\mathbf{a}}^j)^\top \mathbf{v}^j}{\|\mathbf{v}^j\|_2^2} \mathbf{v}^j.$$

We can simplify this to:

$$\begin{aligned} (\tilde{\mathbf{a}}^j)^\top \mathbf{v}^j &= (\tilde{\mathbf{a}}_2^j)^\top (\tilde{\mathbf{a}}_2^j + c_j [1, 0, \dots, 0]^\top) \\ &= \|\tilde{\mathbf{a}}_2^j\|_2^2 + (\tilde{\mathbf{a}}_2^j)_1 c_j, \end{aligned}$$

and:

$$\begin{aligned} \|\mathbf{v}^j\|_2^2 &= (\mathbf{v}^j)^\top \mathbf{v}^j = (\tilde{\mathbf{a}}_2^j + [c_j, 0, \dots, 0]^\top)^\top (\tilde{\mathbf{a}}_2^j + c_j [1, 0, \dots, 0]^\top) \\ &= \|\tilde{\mathbf{a}}_2^j\|_2^2 + 2 (\tilde{\mathbf{a}}_2^j)_1 c_j + (c_j)^2 \\ &= 2 \left(\|\tilde{\mathbf{a}}_2^j\|_2^2 + (\tilde{\mathbf{a}}_2^j)_1 c_j \right). \end{aligned}$$

This implies:

$$\frac{(\tilde{\mathbf{a}}^j)^\top \mathbf{v}^j}{\|\mathbf{v}^j\|_2^2} = \frac{1}{2}.$$

Therefore,

$$\mathbf{H}_{\mathbf{v}^j} \tilde{\mathbf{a}}^j = \tilde{\mathbf{a}}^j - \mathbf{v}^j.$$

Note that the vectors $\tilde{\mathbf{a}}^j$ and \mathbf{v}^j coincide on indices $j+1$ to m so that

$$\mathbf{H}_{\mathbf{v}^j} \tilde{\mathbf{a}}^j = \begin{bmatrix} \tilde{\mathbf{a}}_1^j \\ \tilde{\mathbf{a}}_2^j \end{bmatrix} - \begin{bmatrix} 0 \\ \vdots \\ 0 \\ \tilde{\mathbf{a}}_2^j \end{bmatrix} - c_j \begin{bmatrix} 0 \\ \vdots \\ 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} = \begin{bmatrix} \mathbf{r}_1^j \\ 0 \\ \vdots \\ 0 \end{bmatrix},$$

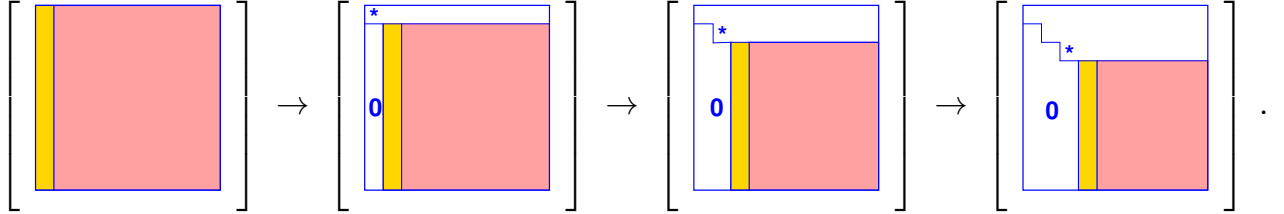
where \mathbf{r}_1^j is some vector of length j , so that $\mathbf{H}_{\mathbf{v}^j} \tilde{\mathbf{a}}^j$ has $m-j$ zeros.

What happens to the previous columns $\mathbf{r}^k = \mathbf{H}_{\mathbf{v}^k} \mathbf{a}^k$ for $k < j$ as we apply $\mathbf{H}_{\mathbf{v}^j}$?

$$\mathbf{H}_{\mathbf{v}^j} \mathbf{r}^k = \mathbf{r}^k - 2 \frac{(\mathbf{v}^j)^\top \mathbf{r}^k}{\|\mathbf{v}^j\|_2^2} \mathbf{v}^j = \mathbf{r}^k \text{ since the following holds:}$$

By definition: first $j - 1$ entries of \mathbf{v}^j are zero and only first k entries of \mathbf{r}^k are nonzero (i.e. $\mathbf{r}^k = [(\mathbf{r}_1^k)^\top, 0, \dots, 0]^\top$). Thus, $(\mathbf{v}^j)^\top \mathbf{r}^k = 0$.

Iteratively, we might decompose $\mathbf{H}_{\mathbf{v}^n} \dots \mathbf{H}_{\mathbf{v}^1} \mathbf{A} = \mathbf{R}$, where all $\mathbf{H}_{(\cdot)}$ are orthogonal matrices so that $\mathbf{Q} = (\mathbf{H}_{\mathbf{v}^n} \dots \mathbf{H}_{\mathbf{v}^1})^\top = \mathbf{H}_{\mathbf{v}^1} \dots \mathbf{H}_{\mathbf{v}^n}$ is also orthogonal.



One may see that the first $(j - 1)$ entries of \mathbf{v}^j are zero. Thus, \mathbf{Q} may be stored implicitly by storing the vectors $\mathbf{v}^1, \dots, \mathbf{v}^n$ as a lower triangular matrix (compressed format).

Householder reflections in EIGEN:

Code Snippet 3.2: QR-decompositions in EIGEN → GITLAB

```

14 # include <Eigen/QR>
15
16 // Computation of full QR-decomposition,
17 // dense matrices built for both QR-factors (expensive!)
18 std::pair<MatrixXd, MatrixXd> qr_decomp_full(const MatrixXd& A) {
19     Eigen::HouseholderQR<MatrixXd> qr(A);
20     MatrixXd Q = qr.householderQ(); //
21     MatrixXd R = qr.matrixQR().template triangularView<Eigen::Upper>();
22     return std::pair<MatrixXd, MatrixXd>(Q, R);
23 }
24
25 // Computation of economical QR-decomposition,
26 // dense matrix built for Q-factor (possibly expensive!)
27 std::pair<MatrixXd, MatrixXd> qr_decomp_eco(const MatrixXd& A) {
28     using index_t = MatrixXd::Index;
29     const index_t m = A.rows(), n = A.cols();
30     Eigen::HouseholderQR<MatrixXd> qr(A);
31     MatrixXd Q = (qr.householderQ()*MatrixXd::Identity(m, n)); //
32     MatrixXd R = qr.matrixQR().block(0, 0, n, n).template triangularView<Eigen
        ↳ ::Upper>(); //
33     return std::pair<MatrixXd, MatrixXd>(Q, R);
34 }
35 /*

```

In general: Only Householder reflections are applied instead of building \mathbf{Q} .

Asymptotic complexity of Householder QR-decomposition

The computational effort for **HouseholderQR()** of $\mathbf{A} \in \mathbb{R}^{m,n}$, $m > n$, is $\mathcal{O}(mn^2)$ for $m, n \rightarrow \infty$.

We derive the above complexity by considering the following complexities:

Each Householder reflection applied to one vector: $\mathcal{O}(m)$

Each Householder reflection applied to \mathbf{A} : $\mathcal{O}(mn)$

n such reflections overall lead to: $\mathcal{O}(n^2m)$

3.3.3 QR-Based Solver for Linear Least Squares Problems

Suppose we have computed the decomposition $\mathbf{A} = \mathbf{QR}$ and that we seek to find a least-squares solution to $\mathbf{Ax} = \mathbf{b}$. We can use the QR decomposition to obtain an equivalent problem to be solved in the least-squares sense:

$$\|\mathbf{Ax} - \mathbf{b}\|_2 = \|\mathbf{Q}(\mathbf{Rx} - \mathbf{Q}^H\mathbf{b})\|_2 = \|\mathbf{Rx} - \tilde{\mathbf{b}}\|_2, \quad \tilde{\mathbf{b}} := \mathbf{Q}^H\mathbf{b}.$$

In the above, we have used that $\mathbf{QQ}^H = \mathbf{I}$ and in a second step, that \mathbf{Q} preserves the 2-norm. With this, we have obtained an *equivalent* triangular linear least squares problem:

$$\|\mathbf{Ax} - \mathbf{b}\|_2 \rightarrow \min \Leftrightarrow \left\| \begin{bmatrix} \mathbf{R}_0 \\ \mathbf{0} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} - \begin{bmatrix} \tilde{b}_1 \\ \vdots \\ \tilde{b}_m \end{bmatrix} \right\|_2 \rightarrow \min.$$

The entries of \mathbf{b} with index $n + 1$ to m can never be fulfilled, so that the least-squares

solution boils down to solving the upper part of the above system:

$$\mathbf{x} = \left[\begin{array}{c|c} \text{yellow blocks} & \mathbf{R}_0 \end{array} \right]^{-1} \begin{bmatrix} \tilde{b}_1 \\ \vdots \\ \tilde{b}_n \end{bmatrix}, \quad \text{with residual } \mathbf{r} = \mathbf{Q} \begin{bmatrix} 0 \\ \vdots \\ 0 \\ \tilde{b}_{n+1} \\ \vdots \\ \tilde{b}_m \end{bmatrix}.$$

Note. By Theorem 3.3.1, the norm of the residual is readily available:

$$\|\mathbf{r}\|_2 = \sqrt{\tilde{b}_{n+1}^2 + \cdots + \tilde{b}_m^2}.$$

There is also an alternative method for QR factorization: *Givens rotations* [build \mathbf{Q} through rotations], which we will not discuss here.

3.4 Singular Value Decomposition (SVD)

Beside the QR-decomposition, there are other orthogonal factorizations of matrices. The most important among them is the singular value decomposition (SVD), which can be used to obtain the generalized solution (i.e. to calculate the Moore-Penrose Pseudoinverse), to compress signals or to extract common features in a data set (principal component analysis), just to name a few applications.

3.4.1 Theory

Theorem 3.4.1 (Singular value decomposition). *For any $\mathbf{A} \in \mathbb{K}^{m,n}$, there are unitary matrices $\mathbf{U} \in \mathbb{K}^{m,m}$, $\mathbf{V} \in \mathbb{K}^{n,n}$ and a (generalized) diagonal ^(*) matrix $\mathbf{\Sigma} = \text{diag}(\sigma_1, \dots, \sigma_p) \in \mathbb{K}^{m,n}$, $p := \min\{m, n\}$, $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_p \geq 0$, such that*

$$\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^H.$$

Remember the property of unitary matrices: $\mathbf{U}^H\mathbf{U} = \mathbf{I}_m$ and $\mathbf{V}^H\mathbf{V} = \mathbf{I}_n$.

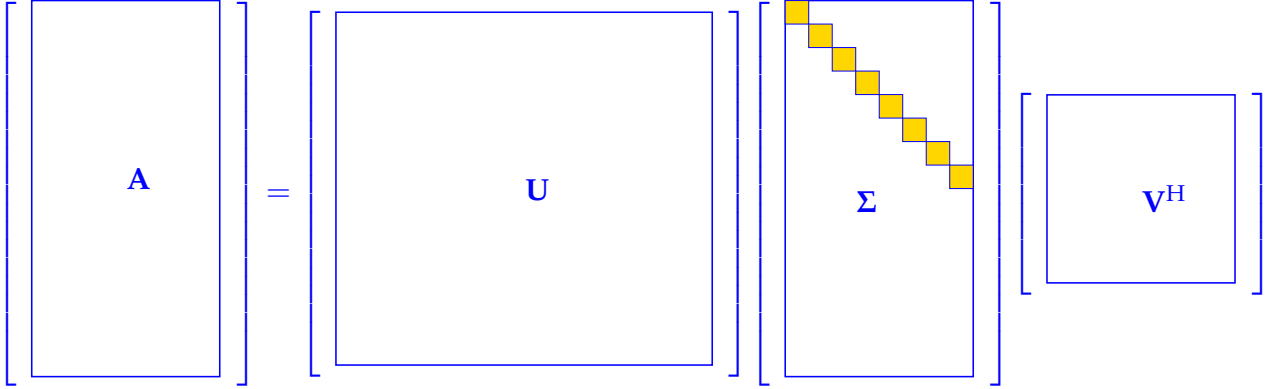
Terminology (*): A matrix $\mathbf{\Sigma}$ is called a generalized diagonal matrix, if $(\mathbf{\Sigma})_{i,j} = 0$, if $i \neq j$, $1 \leq i \leq m$, $1 \leq j \leq n$. We still use the diag operator to create it from a vector.

Definition 3.4.1 (Singular value decomposition (SVD)). The decomposition $\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^H$ from Theorem 3.4.1 is called *singular value decomposition (SVD)* of \mathbf{A} . The diagonal entries σ_i of $\mathbf{\Sigma}$ are the *singular values* of \mathbf{A} .

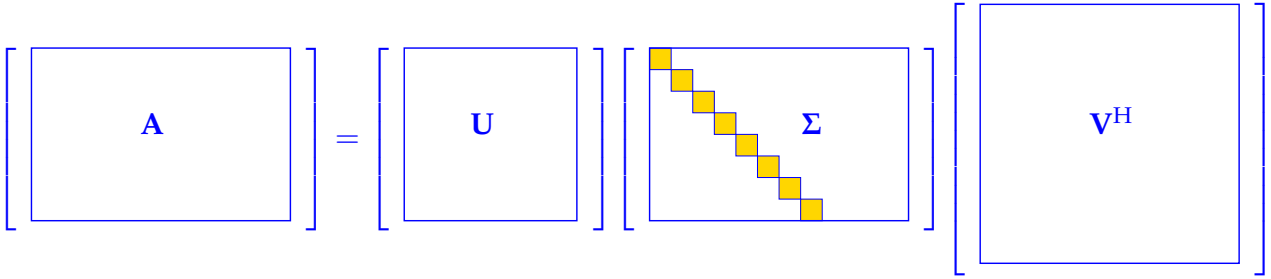
3 Direct Methods for Linear Least Squares Problems

Visualization of the structure of the singular value decomposition of a matrix $\mathbf{A} \in \mathbb{K}^{m,n}$:

The case of a "tall" matrix ($m > n$):



The case of a "fat" matrix ($m < n$):



As in the case of the QR-decomposition, we can also drop the bottom zero rows of $\mathbf{\Sigma}$ and the corresponding columns of \mathbf{U} in the case of $m > n$ (and similarly, we can drop the right part of $\mathbf{\Sigma}$ containing only zero columns and the bottom part of \mathbf{V}^H in the case $m < n$). Thus, we end up with a "reduced/thin" singular value decomposition of $\mathbf{A} \in \mathbb{K}^{m,n}$ with a true diagonal matrix $\mathbf{\Sigma}$, whose diagonal contains the singular values of \mathbf{A} .

Dimensions of the matrices for the reduced SVD in the "tall" vs. the "fat" matrix case:

$$\begin{aligned} m \geq n: \quad \mathbf{A} &= \mathbf{U}\mathbf{\Sigma}\mathbf{V}^H, \quad \mathbf{U} \in \mathbb{K}^{m,n}, \quad \mathbf{\Sigma} \in \mathbb{K}^{n,n}, \quad \mathbf{V} \in \mathbb{K}^{n,n}, \\ m < n: \quad \mathbf{A} &= \mathbf{U}\mathbf{\Sigma}\mathbf{V}^H, \quad \mathbf{U} \in \mathbb{K}^{m,m}, \quad \mathbf{\Sigma} \in \mathbb{K}^{m,m}, \quad \mathbf{V} \in \mathbb{K}^{n,m}. \end{aligned} \quad (3.15)$$

Visualization of reduced/thin SVD for $m > n$:

$$\begin{bmatrix} \mathbf{A} \end{bmatrix} = \begin{bmatrix} \mathbf{U} \end{bmatrix} \begin{bmatrix} \Sigma \end{bmatrix} \begin{bmatrix} \mathbf{V}^H \end{bmatrix}$$

Lemma 3.4.1. The squares σ_i^2 of the non-zero singular values of \mathbf{A} are the non-zero eigenvalues of $\mathbf{A}^H \mathbf{A}$ and $\mathbf{A} \mathbf{A}^H$ with associated eigenvectors $(\mathbf{V})_{:,1}, \dots, (\mathbf{V})_{:,p}$ and $(\mathbf{U})_{:,1}, \dots, (\mathbf{U})_{:,p}$, respectively.

Lemma 3.4.2 (SVD and rank of a matrix). If, for $p := \min\{m, n\}$ and $r \in \{1, \dots, p\}$, the singular values of $\mathbf{A} \in \mathbb{K}^{m,n}$ satisfy $\sigma_1 \geq \dots \geq \sigma_r > \sigma_{r+1} = \dots = \sigma_p = 0$, then

- $\text{rank}(\mathbf{A}) = r$ (no. of non-zero singular values) ,
- $\mathcal{N}(\mathbf{A}) = \text{Span}\{(\mathbf{V})_{:,r+1}, \dots, (\mathbf{V})_{:,n}\}$,
- $\mathcal{R}(\mathbf{A}) = \text{Span}\{(\mathbf{U})_{:,1}, \dots, (\mathbf{U})_{:,r}\}$.

The lemma implies that the SVD of \mathbf{A} automatically reveals the rank of \mathbf{A} and also that \mathbf{U} , \mathbf{V} encode an orthonormal representation of $\mathcal{N}(\mathbf{A})$, $\mathcal{R}(\mathbf{A})$ respectively.

Illustration for $m > n$:

columns = Orthonormal basis (ONB) of $\mathcal{R}(\mathbf{A})$

rows = ONB of $\mathcal{N}(\mathbf{A})$

$$\begin{bmatrix} \mathbf{A} \end{bmatrix} = \begin{bmatrix} \begin{array}{c|c} & \\ \hline & \end{array} \mathbf{U} \end{bmatrix} \begin{bmatrix} \begin{array}{c|c} \Sigma_r & \mathbf{0} \\ \hline \mathbf{0} & \mathbf{0} \end{array} \end{bmatrix} \begin{bmatrix} \mathbf{V}^H \end{bmatrix}$$

More precisely, we can write the decomposition in block format as follows:

$$\begin{array}{c}
\mathbf{A} = [\mathbf{U}_1 \quad \mathbf{U}_2] \begin{bmatrix} \Sigma_r & 0 \\ 0 & 0 \end{bmatrix} \underbrace{\begin{bmatrix} \mathbf{V}_1^H \\ \mathbf{V}_2^H \end{bmatrix}}_{\in \mathbb{K}^{n,n}} \\
\begin{array}{c} \boxed{\mathbf{A}} \\ \in \mathbb{K}^{m,n} \end{array} = \begin{array}{c} \boxed{\mathbf{U}_1 \quad \mathbf{U}_2} \\ \in \mathbb{K}^{m,m} \end{array} \begin{array}{c} \boxed{\begin{array}{cc} \Sigma_r & 0 \\ 0 & 0 \end{array}} \\ \in \mathbb{K}^{m,n} \end{array} \begin{array}{c} \boxed{\begin{array}{c} \mathbf{V}_1^H \\ \mathbf{V}_2^H \end{array}} \\ \in \mathbb{K}^{n,n} \end{array}
\end{array}$$

with $\mathbf{U}_1 \in \mathbb{K}^{m,r}$, $\mathbf{U}_2 \in \mathbb{K}^{m,m-r}$ with orthonormal columns,
 $\Sigma_r = \text{diag}(\sigma_1, \dots, \sigma_r) \in \mathbb{K}^{r,r}$ (singular values),
 $\mathbf{V}_1 \in \mathbb{K}^{n,r}$, $\mathbf{V}_2 \in \mathbb{K}^{n,n-r}$ with orthonormal columns.

3.4.2 SVD in Eigen

The EIGEN class **JacobiSVD** is constructed from a matrix data type, and computes the SVD of its argument during construction and offers access methods **MatrixU()**, **singularValues()**, and **MatrixV()** to request the SVD-factors and singular values.

Code Snippet 3.3: Computing SVDs in EIGEN → GITLAB

```

38 # include <Eigen/SVD>
39
40 // Computation of (full) SVD  $\mathbf{A} = \mathbf{U}\Sigma\mathbf{V}^H \rightarrow$  theorem 3.4.1
41 // SVD factors are returned as dense matrices in natural order
42 std::tuple<MatrixXd, MatrixXd, MatrixXd> svd_full(const MatrixXd& A) {
43     Eigen::JacobiSVD<MatrixXd> svd(A, Eigen::ComputeFullU | Eigen::
        ↳ ComputeFullV);
44     MatrixXd U = svd.matrixU(); // get unitary (square) matrix  $\mathbf{U}$ 
45     MatrixXd V = svd.matrixV(); // get unitary (square) matrix  $\mathbf{V}$ 
46     VectorXd sv = svd.singularValues(); // get singular values as vector
47     MatrixXd Sigma = MatrixXd::Zero(A.rows(), A.cols());
48     const unsigned p = sv.size(); // no. of singular values
49     Sigma.block(0,0,p,p) = sv.asDiagonal(); // set diagonal block of  $\Sigma$ 
50     return std::tuple<MatrixXd, MatrixXd, MatrixXd>(U, Sigma, V);
51 }
52
53 // Computation of economical (thin) SVD  $\mathbf{A} = \mathbf{U}\Sigma\mathbf{V}^H$ , see (3.15)
54 // SVD factors are returned as dense matrices in natural order
55 std::tuple<MatrixXd, MatrixXd, MatrixXd> svd_eco(const MatrixXd& A) {
56     Eigen::JacobiSVD<MatrixXd> svd(A, Eigen::ComputeThinU | Eigen::
        ↳ ComputeThinV);
57     MatrixXd U = svd.matrixU(); // get unitary (square) matrix  $\mathbf{U}$ 
58     MatrixXd V = svd.matrixV(); // get unitary (square) matrix  $\mathbf{V}$ 

```

```

59 VectorXd sv = svd.singularValues(); // get singular values as vector
60 MatrixXd Sigma = sv.asDiagonal(); // build diagonal matrix  $\Sigma$ 
61 return std::tuple<MatrixXd, MatrixXd, MatrixXd>(U, Sigma, V);
62 }
63 /*

```

The second argument in the constructor of **JacobiSVD** determines, whether the methods **matrixU()** and **matrixV()** return the factor for the full SVD or of the economical (thin) SVD (3.15).

`Eigen::ComputeFull*` will select the full versions, whereas `Eigen::ComputeThin*` picks the economical versions. Furthermore without these flags only the singular values are computed. Internally, the computation of the SVD is done by a sophisticated algorithm, for which key steps rely on orthogonal/unitary transformations. The routine **JacobiSVD** is numerically stable since it only relies on norm-preserving transformations. According to EIGEN's documentation, the SVD of a general dense matrix involves the following asymptotic complexity:

Asymptotic cost of thin ($m > n$) SVD of $\mathbf{A} \in \mathbb{K}^{m,n} : \mathcal{O}(\min\{m, n\}^2 \max\{m, n\}) = \mathcal{O}(mn^2)$, which is linear in the larger matrix dimension.

3.4.3 Generalized solutions by SVD

Assume: $\mathbf{A} \in \mathbb{R}^{m,n}$ $m \geq n$ $\text{rank}(\mathbf{A}) = r \leq n$. As before,

$$\mathbf{A} = \begin{bmatrix} \mathbf{U}_1 & \mathbf{U}_2 \end{bmatrix} \begin{bmatrix} \Sigma_r & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} \mathbf{V}_1^H \\ \mathbf{V}_2^H \end{bmatrix}$$

The least squares problem is to find \mathbf{x} such that:

$$\min_{\mathbf{x}} \|\mathbf{Ax} - \mathbf{b}\|_2.$$

We use the SVD in the above block form to write:

$$\begin{aligned} \|\mathbf{Ax} - \mathbf{b}\|_2^2 &= \left\| \mathbf{U} \Sigma \mathbf{V}^H \mathbf{x} - \mathbf{b} \right\|_2^2 \underbrace{=}_{\|\mathbf{U}\|_2=1} \left\| \mathbf{U}^H (\mathbf{U} \Sigma \mathbf{V}^H \mathbf{x} - \mathbf{b}) \right\|_2^2 \\ &\underbrace{=}_{\mathbf{U}^H \mathbf{U} = \mathbf{I}} \left\| \Sigma \mathbf{V}^H \mathbf{x} - \mathbf{U}^H \mathbf{b} \right\|_2^2 = \left\| \begin{bmatrix} \Sigma_r \mathbf{V}_1^H \mathbf{x} \\ 0 \end{bmatrix} - \begin{bmatrix} \mathbf{U}_1^H \mathbf{b} \\ \mathbf{U}_2^H \mathbf{b} \end{bmatrix} \right\|_2^2 \\ &= \underbrace{\left\| \begin{bmatrix} \Sigma_r \mathbf{V}_1^H \mathbf{x} - \mathbf{U}_1^H \mathbf{b} \\ -\mathbf{U}_2^H \mathbf{b} \end{bmatrix} \right\|_2^2}_{\|\cdot\|_{\mathbb{K}^m}} = \underbrace{\left\| \Sigma_r \mathbf{V}_1^H \mathbf{x} - \mathbf{U}_1^H \mathbf{b} \right\|_2^2}_{\|\cdot\|_{\mathbb{K}^r}} + \underbrace{\left\| \mathbf{U}_2^H \mathbf{b} \right\|_2^2}_{\text{fixed, } \|\cdot\|_{\mathbb{K}^{m-r}}}. \end{aligned}$$

Therefore the least-squares problem is equivalent to solving:

$$\min_{\mathbf{x}} \left\| \Sigma_r \mathbf{V}_1^H \mathbf{x} - \mathbf{U}_1^H \mathbf{b} \right\|_2^2.$$

Choose \mathbf{x} such that $\Sigma_r \mathbf{V}_1^H \mathbf{x} = \mathbf{U}_1^H \mathbf{b}$ which is an $r \times n$ LSE (possibly underdetermined).

If $r < n$, the generalized solution is not unique since $\mathcal{N}(\mathbf{V}_1^H) = \mathcal{R}(\mathbf{V}_1)^\perp = \mathcal{R}(\mathbf{V}_2)$.

Choose $\mathbf{x} \in \mathcal{R}(\mathbf{V}_1)$, therefore $\mathbf{x} = \mathbf{V}_1 \mathbf{z}$ for some $\mathbf{z} \in \mathbb{K}^r$. Thus, we can obtain:

$$\Sigma_r \underbrace{\mathbf{V}_1^H \mathbf{V}_1}_{\mathbf{I}} \mathbf{z} = \mathbf{U}_1^H \mathbf{b} \implies \mathbf{z} = \Sigma_r^{-1} \mathbf{U}_1^H \mathbf{b}$$

Hence, the generalized solution can be written as: $\mathbf{x} = \mathbf{V}_1 \mathbf{z} = \mathbf{V}_1 \Sigma_r^{-1} \mathbf{U}_1^H \mathbf{b}$.

Theorem 3.4.2 (Pseudoinverse and SVD). *If $\mathbf{A} \in \mathbb{K}^{m,n}$ has the SVD decomposition $\mathbf{A} = \mathbf{U} \Sigma \mathbf{V}^H$ partitioned as in (3.4.1), then its Moore-Penrose pseudoinverse (see Theorem 3.1.5) is given by $\mathbf{A}^\dagger = \mathbf{V}_1 \Sigma_r^{-1} \mathbf{U}_1^H$.*

Note. This requires $\text{rank}(\mathbf{A})$. Therefore we need to determine the numerical rank of \mathbf{A} , which is a computational substitute for the exact rank and is computed as $r := \#\{\sigma_i : |\sigma_i| \geq \text{tol} \cdot \max_j |\sigma_j|\}$. The default tolerance is $\text{tol} = \text{EPS}$ and can be set manually with the command `setThreshold()`.

3.4.4 SVD-Based Optimization and Approximation

For the general least squares problem (3.4) we have now seen the use of SVD for its numerical solution. Thus, one can say that the SVD has been a powerful tool for solving a minimization problem for a 2-norm. In many other contexts, the SVD is also a key component in numerical optimization.

Norm-Constrained Extrema of Quadratic Forms

We consider the following minimization problem of finding the extrema of quadratic forms on the Euclidean unit sphere $\{\mathbf{x} \in \mathbb{K}^n : \|\mathbf{x}\|_2 = 1\}$:

Given $\mathbf{A} \in \mathbb{K}^{m,n}$, $m \geq n$, find $\mathbf{x} \in \mathbb{K}^n$, $\|\mathbf{x}\|_2 = 1$, such that

$$\|\mathbf{A}\mathbf{x}\|_2 \rightarrow \min. \quad (3.16)$$

We use the property that multiplication with orthogonal/unitary matrices preserves the 2-norm and exploit the singular value decomposition $\mathbf{A} = \mathbf{U} \Sigma \mathbf{V}^H$:

$$\begin{aligned} \min_{\|\mathbf{x}\|_2=1} \|\mathbf{A}\mathbf{x}\|_2^2 &= \min_{\|\mathbf{x}\|_2=1} \|\mathbf{U} \Sigma \mathbf{V}^H \mathbf{x}\|_2^2 = \min_{\|\mathbf{V}^H \mathbf{x}\|_2=1} \|\mathbf{U} \Sigma (\mathbf{V}^H \mathbf{x})\|_2^2 \\ &= \min_{\|\mathbf{y}\|_2=1} \|\Sigma \mathbf{y}\|_2^2 = \min_{\|\mathbf{y}\|_2=1} (\sigma_1^2 y_1^2 + \cdots + \sigma_n^2 y_n^2) \geq \sigma_n^2. \end{aligned}$$

In the second line of the above we have used the substitution $\mathbf{y} = \mathbf{V}^H \mathbf{x}$.

Since the singular values are assumed to be sorted as $\sigma_1 \geq \sigma_2 \geq \dots \geq \sigma_n$, the minimum value σ_n^2 is attained for $\mathbf{V}^H \mathbf{x} = \mathbf{y} = \mathbf{e}_n$. Thus, the minimizer takes the value $\mathbf{x} = \mathbf{V} \mathbf{e}_n = (\mathbf{V})_{:,n}$.

Example (computational geometry): Fitting a hyperplane

Recall the *Hesse normal form* of a hyperplane \mathcal{H} (= affine subspace of dimension $d - 1$) in \mathbb{R}^d :

$$\mathcal{H} = \{\mathbf{x} \in \mathbb{R}^d: c + \mathbf{n}^\top \mathbf{x} = 0\}, \quad \|\mathbf{n}\|_2 = 1, \quad (3.17)$$

where \mathbf{n} is the unit normal to \mathcal{H} and $|c|$ gives the distance of \mathcal{H} from $\mathbf{0}$. The Hesse normal form is convenient for computing the distance of points from \mathcal{H} , because the Euclidean distance of $\mathbf{y} \in \mathbb{R}^d$ from the plane is

$$\text{dist}(\mathcal{H}, \mathbf{y}) = |c + \mathbf{n}^\top \mathbf{y}|. \quad (3.18)$$

Goal: Given the points $\mathbf{y}_1, \dots, \mathbf{y}_m$, $m > d$, find $c \in \mathbb{R}$ and $\mathbf{n} \in \mathbb{R}^d$, with $\|\mathbf{n}\|_2 = 1$, such that

$$\sum_{j=1}^m \text{dist}(\mathcal{H}, \mathbf{y}_j)^2 = \sum_{j=1}^m |c + \mathbf{n}^\top \mathbf{y}_j|^2 \rightarrow \min. \quad (3.19)$$

Note that (3.19) is **not** a linear least squares problem due to the *constraint* $\|\mathbf{n}\|_2 = 1$. However, it turns out to be a minimization problem with *almost* the structure of (3.16).

Best Low-Rank Approximation

Matrix compression addresses the problem of approximating a given “generic” matrix (of a certain class) by means of matrix, whose “information content”, that is, the number of reals needed to store it, is significantly lower than the information content of the original matrix. Sparse matrices are a prominent class of matrices with “low information content”. Unfortunately, they cannot approximate dense matrices very well. Another type of matrices that enjoy “low information content”, also called data sparse, are low-rank matrices.

Low-rank approximations allow, for example, image compression and they are also the foundation for principal component analysis (PCA).

For a given matrix \mathbf{A} , we want to find a low-rank matrix that is closest to \mathbf{A} :

Best Low-Rank Approximation

3 Direct Methods for Linear Least Squares Problems

Given a matrix $\mathbf{A} \in \mathbb{K}^{m,n}$, find a matrix $\tilde{\mathbf{A}} \in \mathbb{K}^{m,n}$, $\text{rank}(\tilde{\mathbf{A}}) \leq k$, such that

$$\|\mathbf{A} - \tilde{\mathbf{A}}\|_{2/F} \rightarrow \min \quad \text{over all rank-}k \text{ matrices .}$$

Here we explore low-rank best approximation of general matrices with respect to the Euclidean matrix norm $\|\cdot\|_2$ induced by the 2-norm for vectors, and the Frobenius norm $\|\cdot\|_F$. Typically: $k \ll \min\{m, n\}$.

Definition 3.4.2 (Frobenius norm). The *Frobenius norm* of $\mathbf{A} \in \mathbb{K}^{m,n}$ is defined as

$$\|\mathbf{A}\|_F^2 := \sum_{i=1}^m \sum_{j=1}^n |a_{ij}|^2 .$$

It is readily seen that $\|\mathbf{A}\|_F$ is invariant under orthogonal/unitary transformations. Thus the Frobenius norm of a matrix \mathbf{A} , can be expressed through its singular values σ_j :

$$\text{Frobenius norm and SVD:} \quad \|\mathbf{A}\|_F^2 = \sum_{j=1}^r \sigma_j^2, \text{ where } r = \text{rank}(\mathbf{A}) . \quad (3.20)$$

Recall: Rank-1 matrices are tensor products of vectors:

$$\mathbf{A} \in \mathbb{K}^{m,n} \quad \text{and} \quad \text{rank}(\mathbf{A}) = 1 \quad \Leftrightarrow \quad \exists \mathbf{u} \in \mathbb{K}^m, \mathbf{v} \in \mathbb{K}^n, \text{ such that } \mathbf{A} = \mathbf{u}\mathbf{v}^H .$$

Note. The singular value decomposition provides an additive decomposition into rank-1 matrices:

$$\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^H = \sum_{i=1}^r \sigma_i \underbrace{(\mathbf{U})_{:,i} (\mathbf{V})_{:,i}^H}_{\text{outer product}} . \quad (3.21)$$

Thus an intuitive way to approximate \mathbf{A} in this context is to truncate this sum. A k -rank approximation is therefore given by:

$$\tilde{\mathbf{A}} = \sum_{i=1}^k \sigma_i (\mathbf{U})_{:,i} (\mathbf{V})_{:,i}^H, \quad k \leq r . \quad (3.22)$$

Is this already a best k -rank approximation of \mathbf{A} ? Let us define the space of all matrices with rank less or equal to k by

$$\mathcal{R}_k(m, n) := \{\mathbf{F} \in \mathbb{K}^{m,n} : \text{rank}(\mathbf{F}) \leq k\}, \quad m, n, k \in \mathbb{N}.$$

The following profound theorem links best approximation in $\mathcal{R}_k(m, n)$ and the singular value decomposition.

Theorem 3.4.3 (Best low rank approximation). *Let $\mathbf{A} = \mathbf{U}\Sigma\mathbf{V}^H$ be the SVD of $\mathbf{A} \in \mathbb{K}^{m,n}$. For $1 \leq k \leq \text{rank}(\mathbf{A}) = r$, set $\mathbf{U}_k := [\mathbf{u}_{:,1}, \dots, \mathbf{u}_{:,k}] \in \mathbb{K}^{m,k}$, $\mathbf{V}_k := [\mathbf{v}_{:,1}, \dots, \mathbf{v}_{:,k}] \in \mathbb{K}^{n,k}$, $\Sigma_k := \text{diag}(\sigma_1, \dots, \sigma_k) \in \mathbb{K}^{k,k}$. Then, for $\|\cdot\| = \|\cdot\|_F$ and $\|\cdot\| = \|\cdot\|_2$, the following holds:*

$$\left\| \mathbf{A} - \mathbf{U}_k \Sigma_k \mathbf{V}_k^H \right\| \leq \|\mathbf{A} - \mathbf{F}\| \quad \forall \mathbf{F} \in \mathcal{R}_k(m, n).$$

Furthermore the error made in this approximation can easily be written out:

$$\mathbf{A} - \tilde{\mathbf{A}} = \sum_{i=k+1}^r \sigma_i(\mathbf{U})_{:,i} (\mathbf{V})_{:,i}^H.$$

Also note that the singular values of $\mathbf{A} - \tilde{\mathbf{A}}$ are $\sigma_{k+1} \geq \dots \geq \sigma_r$.

Therefore, we can conclude that the largest new singular value is σ_{k+1} . Thus:

$$\left\| \mathbf{A} - \tilde{\mathbf{A}} \right\|_2 = \sigma_{k+1}, \quad \left\| \mathbf{A} - \tilde{\mathbf{A}} \right\|_F^2 = \sum_{i=k+1}^r \sigma_i^2.$$

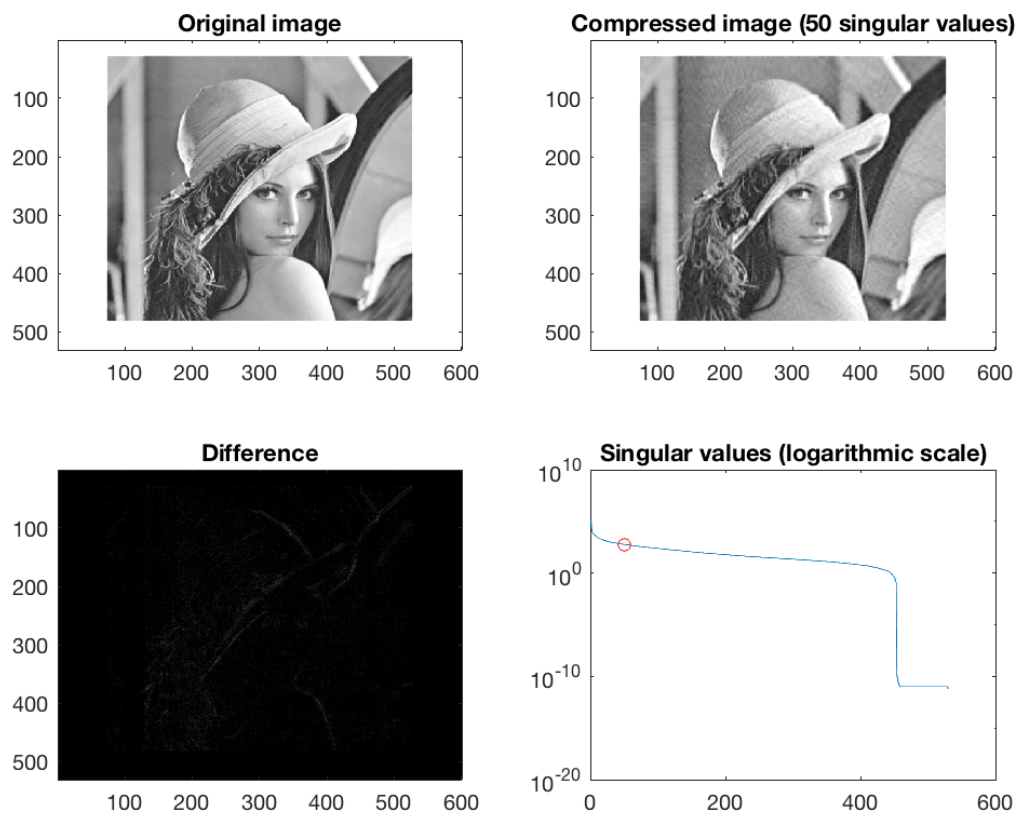
Theorem 3.4.3 states that the rank- k matrix that is *closest* to \mathbf{A} (rank- k best approximation) in both the Euclidean matrix norm and the Frobenius norm can be obtained by truncating the rank-1 sum expansion (3.21) obtained from the SVD of \mathbf{A} after k terms.

What we have gained is an easy way to compress matrices. Storing a generic dense matrix $\mathbf{A} \in \mathbb{R}^{m,n}$ by definition requires $m \cdot n$ elements, whereas storing the best rank- k approximation for a fixed $k \in \mathbb{N}$ only requires $k(m + n - 1)$ elements. (Note that we can get $k(m + n - 1)$ instead of $k(m + n + 1)$ because each column of \mathbf{U} and \mathbf{V} is normalized.)

Example 3.4.1: Image compression

A rectangular greyscale image composed of $m \times n$ pixels (greyscale, BMP format) can be regarded as a matrix $\mathbf{A} \in \mathbb{R}^{m,n}$, $a_{ij} \in \{0, \dots, 255\}$.

3 Direct Methods for Linear Least Squares Problems



Original image: $\underbrace{530}_m \times \underbrace{600}_n \approx 3 \cdot 10^5$

Compressed image: $50 \cdot \underbrace{1129}_{m+n-1} \approx 5 \cdot 10^4$

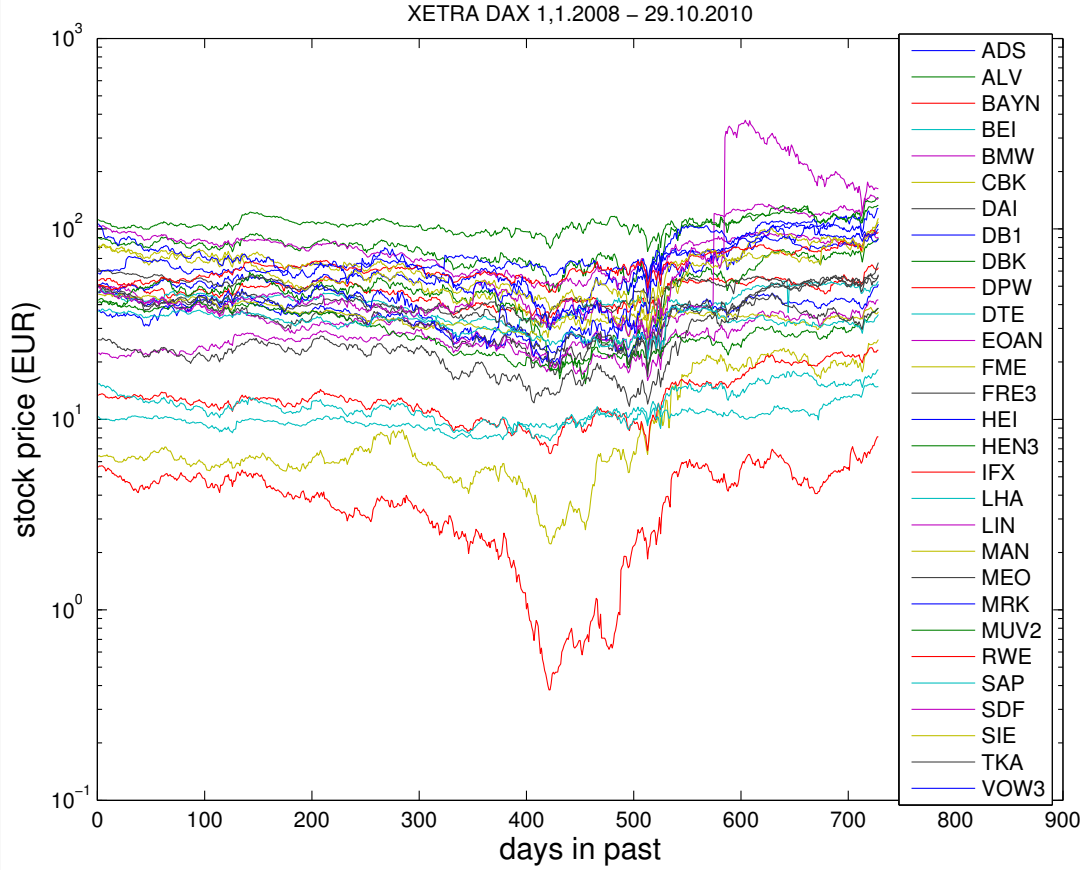
Note. In practice, there are better and faster ways to compress images than SVD (JPEG, Wavelets, etc.)

Principal Component Data Analysis (PCA)

Important tool for:

- dimensionality reduction
- trend analysis
- data classification

Example 3.4.2:



In the figure, we see stock prices – measured at the end of the day – as a vector $\mathbf{a}_j \in \mathbb{R}^m$ for n stocks.

Given: n data points $\mathbf{a}_j \in \mathbb{R}^m, j = 1, \dots, n$, in m -dimensional (feature) space.
(E.g., \mathbf{a}_j may represent a finite *time series* or a *measured relationship* of physical quantities)

Are there underlying governing trends?

Trend \triangleq Are there vectors $\widetilde{\mathbf{u}}_1, \dots, \widetilde{\mathbf{u}}_p, p < n$, such that

$$\mathbf{a}_j \in \text{Span}\{\widetilde{\mathbf{u}}_1, \dots, \widetilde{\mathbf{u}}_p\}, \quad \forall j \in \{1, \dots, n\} \quad (3.23)$$

Additionally we want them to be orthonormal since trends should be as “independent as possible” (minimally correlated).

Perspective of linear algebra:

$$(3.23) \quad \Leftrightarrow \quad \text{rank}(\mathbf{A}) = p \text{ for } \mathbf{A} := [\mathbf{a}_1, \dots, \mathbf{a}_n] \in \mathbb{R}^{m,n} \text{ and } \mathcal{R}(\mathbf{A}) \subseteq \text{Span}\{\widetilde{\mathbf{u}}_1, \dots, \widetilde{\mathbf{u}}_p\}. \quad (3.24)$$

The above is an unrealistic scenario because small *random fluctuations* will be present in each stock price.

3 Direct Methods for Linear Least Squares Problems

More realistic: Stock prices *approximately* follow a few trends p , i.e. $p \ll n$. Thus,

$$\mathbf{a}_j \in \text{Span}\{\widetilde{\mathbf{u}}_1, \dots, \widetilde{\mathbf{u}}_p\} + \text{"small perturbations"}, \quad \forall j = 1, \dots, m,$$

with orthonormal trend vectors $\widetilde{\mathbf{u}}_i$, where $i \in \{1, \dots, p\}$.

The task of PCA is to find the minimal p and the orthonormal trend vectors $\widetilde{\mathbf{u}}_1, \dots, \widetilde{\mathbf{u}}_p$. Now the *singular value decomposition* (SVD) comes into play, because Lemma 3.4.2 tells us that it can supply an orthonormal basis of the image space of a matrix. An issue is how to deal with the small random perturbations. For this we might exploit the structure of the SVD:

Recall (3.21): If $\mathbf{U}\Sigma\mathbf{V}^\top$ is the SVD of $\mathbf{A} \in \mathbb{R}^{m,n}$, then (with \mathbf{u}_j denoting the j -th column of \mathbf{U} , \mathbf{v}_j denoting the j -th column of \mathbf{V}), we can write the SVD alternatively as follows:

$$\left[\begin{array}{c} \mathbf{A} \end{array} \right] = \sigma_1 \left[\begin{array}{c} \mathbf{u}_1 \end{array} \right] \left[\begin{array}{c} \mathbf{v}_1^\top \end{array} \right] + \sigma_2 \left[\begin{array}{c} \mathbf{u}_2 \end{array} \right] \left[\begin{array}{c} \mathbf{v}_2^\top \end{array} \right] + \dots$$

Thus, each \mathbf{a}_j is a linear combination of $\mathbf{u}_1, \dots, \mathbf{u}_n$. The vectors \mathbf{v}_i carry the weights of the linear combinations, i.e.

$$\mathbf{a}_j = (\sigma_1(\mathbf{v}_1)_j)\mathbf{u}_1 + (\sigma_2(\mathbf{v}_2)_j)\mathbf{u}_2 + \dots + (\sigma_n(\mathbf{v}_n)_j)\mathbf{u}_n.$$

We distinguish between two cases: The (unrealistic) case of no perturbations (i.e. \mathbf{A} is exactly low-rank):

$$\begin{aligned} \text{SVD: } \mathbf{A} = \mathbf{U}\Sigma\mathbf{V}^\text{H} \quad &\text{satisfies } \sigma_1 \geq \sigma_2 \geq \dots \sigma_p > \sigma_{p+1} = \dots = \sigma_{\min\{m,n\}} = 0, \\ &\text{with orthonormal trend vectors } (\mathbf{U})_{:,1}, \dots, (\mathbf{U})_{:,p}. \end{aligned}$$

The more realistic case with perturbations in which \mathbf{A} is only approximately low-rank:

$$\begin{aligned} \text{SVD: } \mathbf{A} = \mathbf{U}\Sigma\mathbf{V}^\text{H} \quad &\text{satisfies } \sigma_1 \geq \sigma_2 \geq \dots \sigma_p \gg \sigma_{p+1} \approx \dots \approx \sigma_{\min\{m,n\}} \approx 0, \\ &\text{with orthonormal trend vectors } (\mathbf{U})_{:,1}, \dots, (\mathbf{U})_{:,p}. \end{aligned}$$

If there is a pronounced gap $\sigma_p \gg \sigma_{p+1}$ in the distribution of the singular values, which separates p large singular values from $\min\{m,n\} - p$ relatively small singular values, this

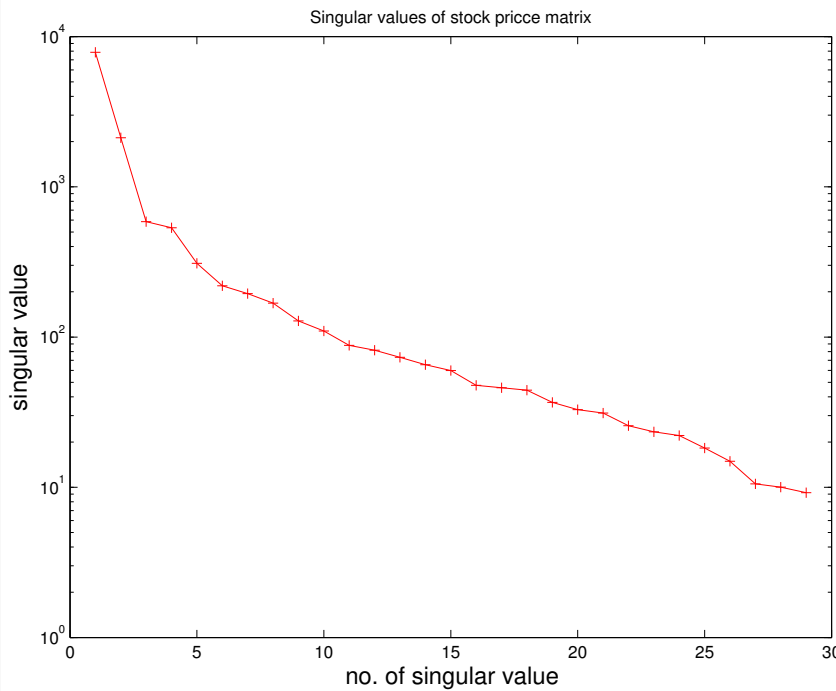
hints at $\mathcal{R}(\mathbf{A})$ having essentially dimension p . It depends on the application what one accepts as a “pronounced gap”.

A frequently used criterion is:

$$p = \min \left\{ q: \sum_{j=1}^q \sigma_j^2 \geq (1 - \tau) \sum_{j=1}^{\min\{m,n\}} \sigma_j^2 \right\} \quad \text{for } \tau \ll 1,$$

i.e. the sum over the first q σ_j^2 's is almost as large as the sum over all σ_j^2 's.

Example 3.4.3:



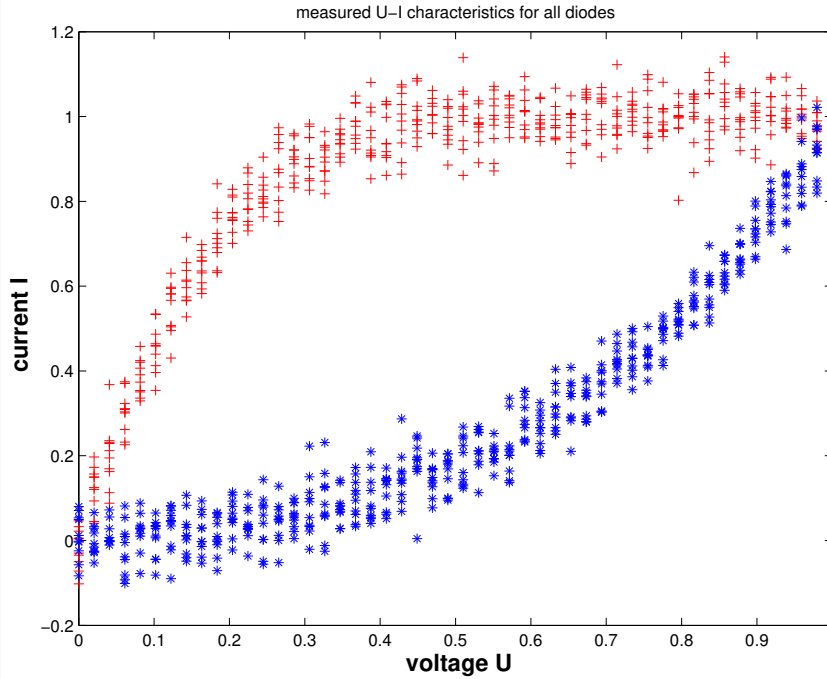
Example: Roughly exponential decay of the singular values of the stock price matrix $\mathbf{A} = [\mathbf{a}_1 \dots \mathbf{a}_n]$ in this logarithmic scale plot.

We observe a pronounced decay of the singular values (\approx exponential decay, logarithmic scale in the figure above). A few trends (corresponding to a few of the largest singular values) govern the time series. The data was obtained from Yahoo Finance.

Example 3.4.4: Classification of measured data

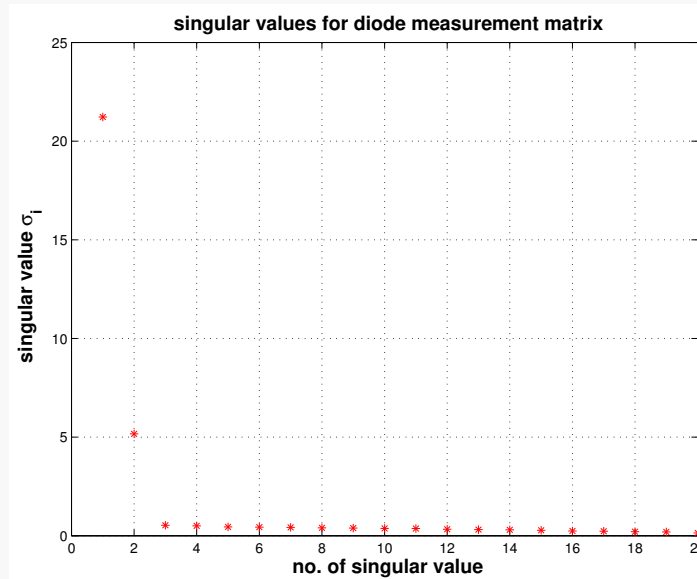
Given measured data of voltage vs current of n diodes $((U_j, I_j^{(k)}), j = 1, \dots, m, k = 1, \dots, n)$, we want to determine the number of different types of diodes in this data set.

3 Direct Methods for Linear Least Squares Problems



In this example, $m = 50$ and $n = 20$.

To find out the number of different types, we write the data vectors in matrix form and compute the SVD.



Distribution of singular values of matrix

As we can see from the distribution of the singular values, there are two dominant singular values. Thus, we can deduce that there are two principal components and hence two types of diodes in the batch.

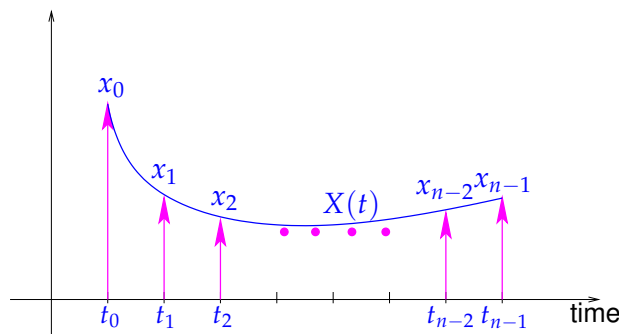
Filtering Algorithms

In the last two chapters, we have discussed linear systems of equations and solution techniques. In the following chapters, we will see this theory used in different numerical algorithms. In this chapter, we will start by considering *filtering*, which is the basis of signal and image processing. Most notably, the discrete Fourier transform (DFT) and its efficient implementation, the fast Fourier transform (FFT), will be introduced. The FFT is a fundamental tool not only for filtering, but also for e.g. fast polynomial multiplication and Chebyshev interpolation.

Motivation: Time-discrete signals and sampling

From the perspective of *signal processing*, we can represent a finite discrete (sampled) signal by a vector $\mathbf{x} \in \mathbb{R}^n$.

Sampling converts a time-continuous signal, represented by some real-valued physical quantity (pressure, voltage, power, etc.) into a time-discrete signal:



A *time-continuous* signal can be denoted by $X = X(t)$, where $0 \leq t \leq T$ and the corresponding "sampled" signal can be denoted by $x_j = X(j\Delta t)$, where $j = 0, \dots, n-1$, $n \in \mathbb{N}$ and

$n\Delta t \leq T$. The time between two samples, also referred to as the sampling rate, is given by Δt .

As already indicated by the indexing, the sampled values can be arranged in a vector $\mathbf{x} = [x_0, \dots, x_{n-1}]^\top \in \mathbb{R}^n$.

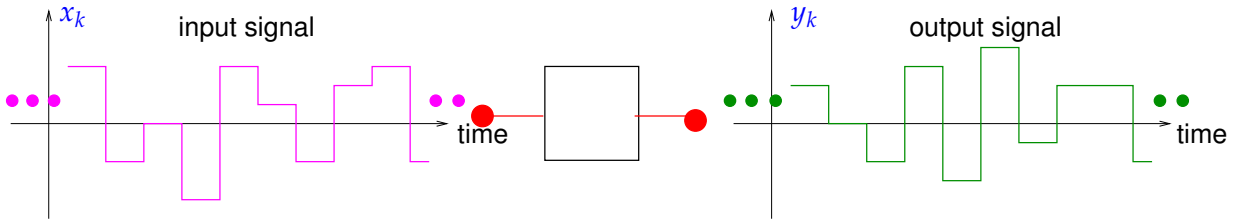
As an idealization, one sometimes considers a signal of infinite duration $X = X(t)$, $-\infty < t < \infty$. In this case, sampling yields a bi-infinite time-discrete signal, represented by a sequence $(x_k)_{k \in \mathbb{Z}}$. If this sequence has a finite number of non-zero terms only, then we write $(\dots, 0, x_\ell, x_{\ell+1}, \dots, x_{n-1}, x_n, 0, \dots)$.

4.1 Discrete Convolutions

4.1.1 Discrete finite linear time-invariant causal channels/filters

Now we study a finite linear time-invariant causal channel/filter, which is a widely used model for digital communication channels, e.g. in wireless communication theory. Mathematically speaking, a (discrete) channel/filter is a mapping $F : \ell^\infty(\mathbb{Z}) \rightarrow \ell^\infty(\mathbb{Z})$ from the vector space $\ell^\infty(\mathbb{Z})$ of bounded input sequences $\{x_j\}_{j \in \mathbb{Z}}$ to bounded output sequences $\{y_j\}_{j \in \mathbb{Z}}$.

$$\ell^\infty(\mathbb{Z}) = \left\{ (x_j)_{j \in \mathbb{Z}} : \sup_{j \in \mathbb{Z}} |x_j| < \infty \right\}$$



$$\text{Channel/filter: } F : \ell^\infty(\mathbb{Z}) \rightarrow \ell^\infty(\mathbb{Z}) \quad , \quad (y_j)_{j \in \mathbb{Z}} = F((x_j)_{j \in \mathbb{Z}}) .$$

For the description of filters, we rely on special input signals, analogous to the description of a linear mapping $\mathbb{R}^n \rightarrow \mathbb{R}^m$ through a matrix, that is, its action on unit vectors. In order to link digital filters to linear algebra, we have to assume certain properties for F that are indicated by the attributes “finite”, “time-invariant”, “linear” and “causal”:

Definition 4.1.1 (Finite channel/filter). A filter $F : \ell^\infty(\mathbb{Z}) \rightarrow \ell^\infty(\mathbb{Z})$ is called *finite*, if every input signal of finite duration produces an output signal of finite duration. More precisely, if there exists an $M \in \mathbb{N}$ such that

$$\forall j \text{ with } |j| > M : x_j = 0,$$

then, there exists $N \in \mathbb{N}$ such that

$$\forall k \text{ with } |k| > N : \left(F \left((x_j)_{j \in \mathbb{Z}} \right) \right)_k = 0 . \quad (4.1)$$

It should not matter when exactly a signal is fed into the channel. To express this intuition more rigorously we introduce the *time shift operator* for signals: For $k \in \mathbb{Z}$,

$$S_k : \ell^\infty(\mathbb{Z}) \rightarrow \ell^\infty(\mathbb{Z}) \quad , \quad S_k((x_j)_{j \in \mathbb{Z}}) = (x_{j-k})_{j \in \mathbb{Z}} . \quad (4.2)$$

Definition 4.1.2 (Time-invariant channel/filter). A filter $F : \ell^\infty(\mathbb{Z}) \rightarrow \ell^\infty(\mathbb{Z})$ is called *time-invariant*, if shifting the input in time leads to the same output shifted in time by the same amount; it *commutes with the time shift operator*:

$$\forall (x_j)_{j \in \mathbb{Z}} \in \ell^\infty(\mathbb{Z}), \forall k \in \mathbb{Z}: \quad F(S_k((x_j)_{j \in \mathbb{Z}})) = S_k(F((x_j)_{j \in \mathbb{Z}})) . \quad (4.3)$$

Definition 4.1.3 (Linear channel/filter). A filter $F : \ell^\infty(\mathbb{Z}) \rightarrow \ell^\infty(\mathbb{Z})$ is called *linear*, if F is a linear mapping, that is, for all bounded sequences $(x_j)_{j \in \mathbb{Z}} \in \ell^\infty(\mathbb{Z})$ and $(y_j)_{j \in \mathbb{Z}} \in \ell^\infty(\mathbb{Z})$ and all $\alpha, \beta \in \mathbb{R}$:

$$F(\alpha (x_j)_{j \in \mathbb{Z}} + \beta (y_j)_{j \in \mathbb{Z}}) = \alpha F((x_j)_{j \in \mathbb{Z}}) + \beta F((y_j)_{j \in \mathbb{Z}}) . \quad (4.4)$$

In other words: The linear combination of inputs has an output equal to the linear combination of the individual outputs.

For all scaling factors $\alpha, \beta \in \mathbb{R}$, this can be depicted as:

$$\text{output}(\alpha \cdot \text{signal 1} + \beta \cdot \text{signal 2}) = \alpha \cdot \text{output}(\text{signal 1}) + \beta \cdot \text{output}(\text{signal 2}) .$$

Of course, a signal should not trigger an output before it arrives at the filter. Therefore, the output may depend only on past and present inputs, but not on the future. This is the principle of causality.

Definition 4.1.4 (Causal channel/filter). A filter $F : \ell^\infty(\mathbb{Z}) \rightarrow \ell^\infty(\mathbb{Z})$ is called *causal* (or physical, or non-anticipative), if the output does not start before the input, meaning the output only depends on past and present inputs and not on the future:

If for some $M \in \mathbb{N}$,

$$x_j = 0 \text{ for all } j \leq M,$$

then

$$F((x_j)_{j \in \mathbb{Z}})_k = 0 \text{ for all } k \leq M .$$

Impulse response:

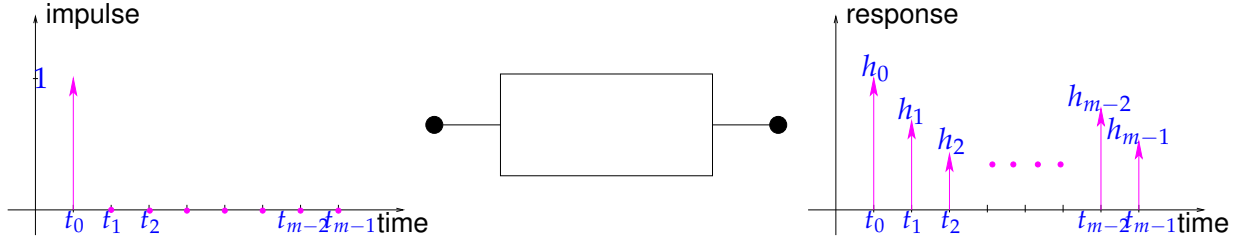
Analogue from LSEs: Matrix $\hat{=}$ describing the action of a linear mapping $\mathbb{R}^n \rightarrow \mathbb{R}^m$ through its action on unit vectors (since $\mathbf{A} \mathbf{e}_j = (\mathbf{A})_{:,j}$).

Now in the same spirit: Describe filters through their action on “impulses”.

Definition 4.1.5 (Impulse response). The *impulse response* of a channel/filter is the output for the single unit impulse

$$x_j = \delta_{j,0} := \begin{cases} 1 & \text{if } j = 0 \\ 0 & \text{else} \end{cases} .$$

Visualization of the (finite) impulse response of a (causal) channel/filter:



The impulse response of a finite and causal filter is a sequence of the form $(\dots, 0, h_0, h_1, \dots, h_{m-1}, 0, \dots)$, where $m \in \mathbb{N}$. The impulse response of a finite filter can be described by a vector \mathbf{h} of finite length m .

Acronyms: $FIR \triangleq$ finite impulse response filters (only finitely many nonzero h_k 's)
 $LT-FIR \triangleq$ finite, linear, time-invariant, and causal filter $F : \ell^\infty(\mathbb{Z}) \rightarrow \ell^\infty(\mathbb{Z})$

4.1.2 Transmission through LT-FIR filters

Let $(\dots, 0, h_0, h_1, \dots, h_{m-1}, 0, \dots)$, $m \in \mathbb{N}$, be the impulse response of an LT-FIR filter $F : \ell^\infty(\mathbb{Z}) \rightarrow \ell^\infty(\mathbb{Z})$:

$$F((\delta_{j,0})_{j \in \mathbb{Z}}) = (\dots, 0, h_0, h_1, \dots, h_{m-1}, 0, \dots).$$

Owing to time-invariance, we already know the response to a shifted unit pulse:

$$F((\delta_{j,k})_{j \in \mathbb{Z}}) = (h_{j-k})_{j \in \mathbb{Z}} = \left(\dots, 0, \underset{\substack{\uparrow \\ t = k\Delta t}}{h_0}, \underset{\substack{\uparrow \\ t = (k+m-1)\Delta t}}{h_1}, \dots, h_{m-1}, 0, \dots \right).$$

Every finite input signal $(\dots, 0, x_0, x_1, \dots, x_{n-1}, 0, \dots) \in \ell^\infty(\mathbb{Z})$ can be written as the superposition of scaled unit impulses, which, in turn, are time-shifted copies of a unit pulse at $t = 0$:

$$(x_j)_{j \in \mathbb{Z}} = \sum_{k=0}^{n-1} x_k (\delta_{j,k})_{j \in \mathbb{Z}} = \sum_{k=0}^{n-1} x_k S_k \left((\delta_{j,0})_{j \in \mathbb{Z}} \right),$$

where S_k is the time-shift operator from (4.2). Applying the filter on both sides of this equation and using linearity leads to the general formula for the output signal $(y_j)_{j \in \mathbb{Z}}$:

$$\begin{aligned} F((x_j)_{j \in \mathbb{Z}}) &= F \left(\sum_{k=0}^{n-1} x_k \cdot S_k \left((\delta_{j,0})_{j \in \mathbb{Z}} \right) \right) \\ &\stackrel{\text{linearity}}{=} \sum_{k=0}^{n-1} x_k \cdot F \left(S_k \left((\delta_{j,0})_{j \in \mathbb{Z}} \right) \right) \\ &\stackrel{\text{time-invariance}}{=} \sum_{k=0}^{n-1} x_k \cdot S_k \left(\underbrace{F \left((\delta_{j,0})_{j \in \mathbb{Z}} \right)}_{\text{impulse response}} \right) \end{aligned}$$

This can be represented in matrix form as:

$$\begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \\ \vdots \\ y_{m+n-3} \\ y_{m+n-2} \end{bmatrix} = x_0 \begin{bmatrix} h_0 \\ \vdots \\ h_{m-1} \\ 0 \\ 0 \\ \vdots \\ \vdots \\ 0 \end{bmatrix} + x_1 \begin{bmatrix} 0 \\ h_0 \\ \vdots \\ h_{m-1} \\ 0 \\ \vdots \\ \vdots \\ 0 \end{bmatrix} + x_2 \begin{bmatrix} 0 \\ 0 \\ h_0 \\ \vdots \\ h_{m-1} \\ 0 \\ \vdots \\ 0 \end{bmatrix} + \cdots + x_{n-1} \begin{bmatrix} 0 \\ \vdots \\ \vdots \\ \vdots \\ 0 \\ h_0 \\ \vdots \\ h_{m-1} \end{bmatrix} .$$

In compact notation, we can write the non-zero components of the output signal $(y_j)_{j \in \mathbb{Z}}$ as:

$$F((x_j)_{j \in \mathbb{Z}})_k = y_k = \sum_{j=0}^{n-1} h_{k-j} x_j, \quad k = 0, \dots, m+n-2 \quad \underbrace{(h_j := 0 \text{ for } j < 0 \text{ and } j \geq m)}_{\text{channel is causal and finite}} . \quad (4.5)$$

The maximal duration of the output is $(\underbrace{m}_{\text{length of filter}} + \underbrace{n}_{\text{length of signal}} - 1) \cdot \Delta t$.

Summary of the above considerations:

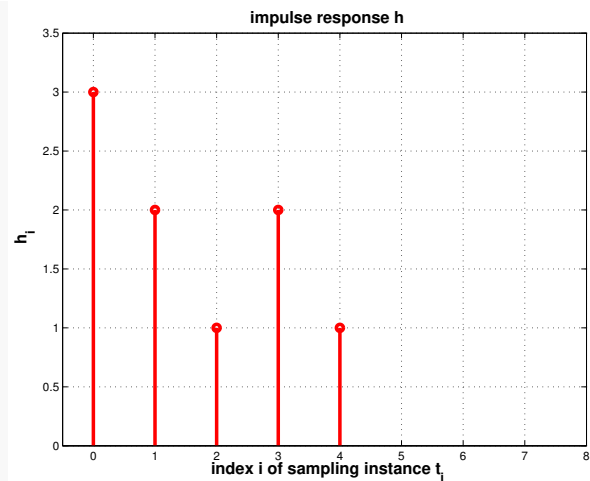
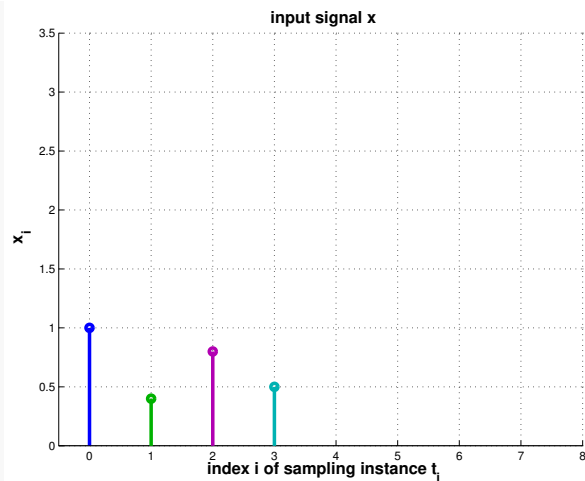
Superposition of impulse responses

The output $y = (\dots, 0, y_0, y_1, y_2, \dots)$ of a LT-FIR channel for finite length input $x = (\dots, 0, x_0, \dots, x_{n-1}, 0, \dots) \in \ell^\infty(\mathbb{Z})$ is a superposition of x_j -weighted and $j\Delta t$ time-shifted impulse responses.

Example 4.1.1:

The following diagrams give a visual display of the superposition of impulse responses for a particular LT-FIR filter, and an input signal of duration $3\Delta t$, where Δt denotes the time between samples.

4 Filtering Algorithms

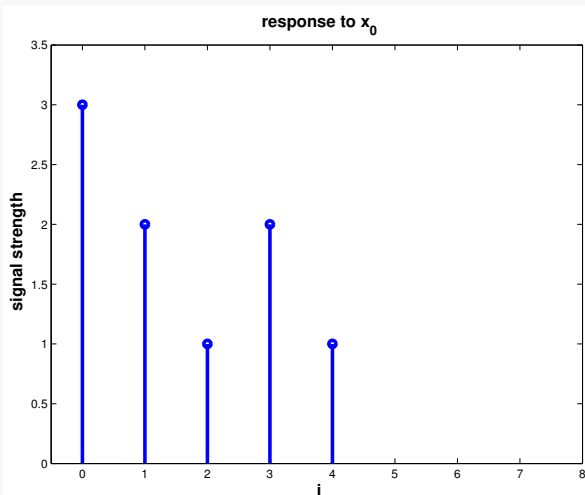


Signal: $(\dots, 0, x_0, \dots, x_3, 0, \dots)$ $n = 4$

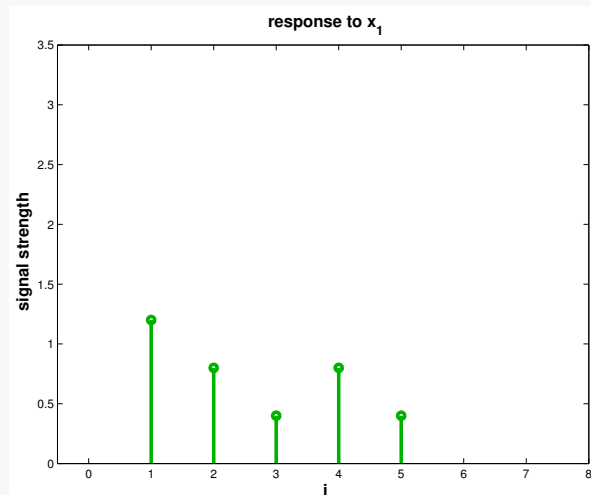
Filter: $(\dots, 0, h_0, \dots, h_4, 0, \dots)$ $m = 5$

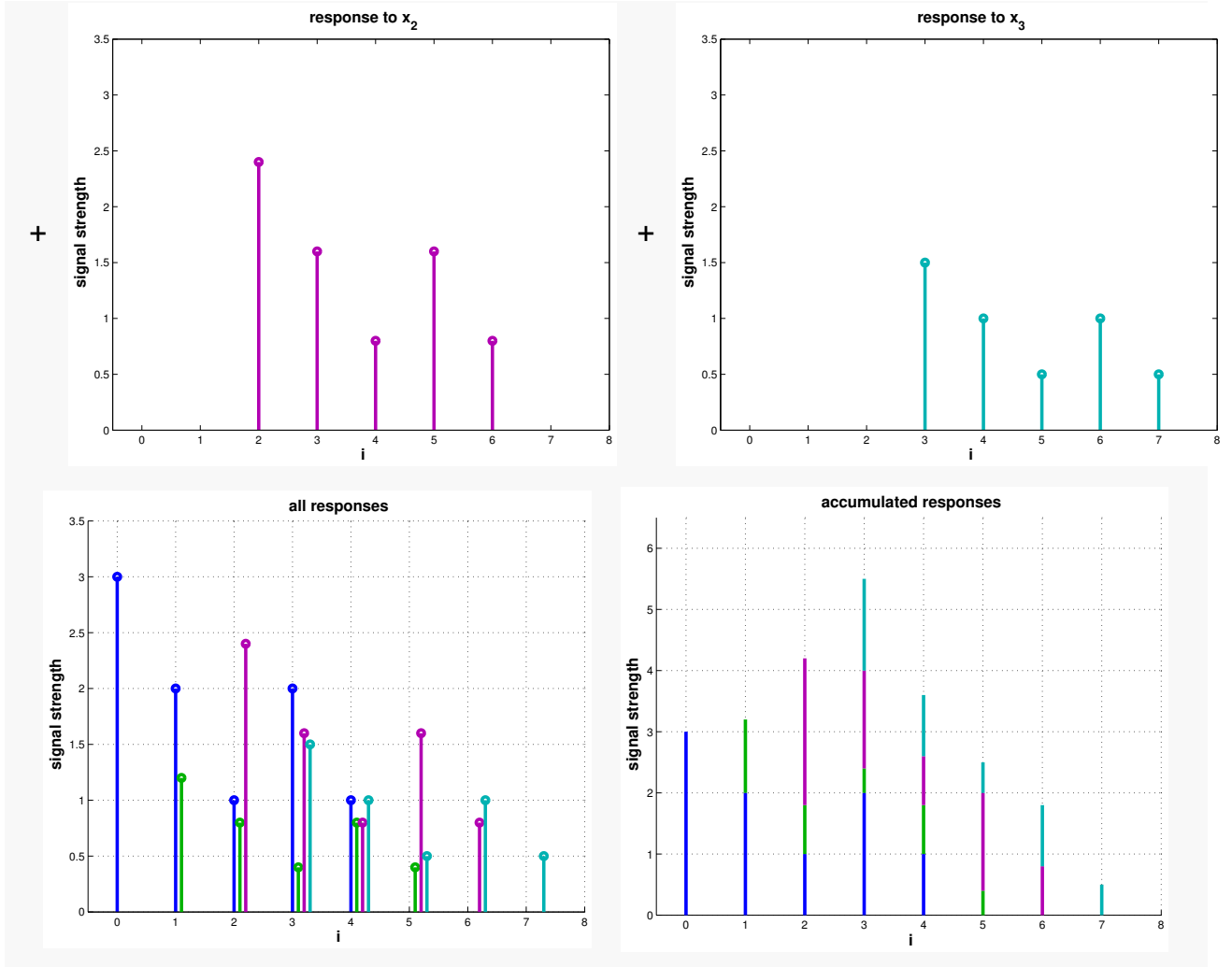
Output = linear superposition of impulse responses:

$$\begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ \vdots \\ y_7 \end{bmatrix} = x_0 \begin{bmatrix} h_0 \\ \vdots \\ h_4 \\ 0 \\ 0 \\ 0 \end{bmatrix} + x_1 \begin{bmatrix} 0 \\ h_0 \\ \vdots \\ h_4 \\ 0 \\ 0 \end{bmatrix} + x_2 \begin{bmatrix} 0 \\ 0 \\ h_0 \\ \vdots \\ h_4 \\ 0 \end{bmatrix} + x_3 \begin{bmatrix} 0 \\ 0 \\ 0 \\ h_0 \\ \vdots \\ h_4 \end{bmatrix}$$



+





Note that we have seen from (4.5) that the output has at most length $m + n - 1$. Therefore, if we know that all input signals are of the form $(\dots, 0, x_0, x_1, \dots, x_{n-1}, 0, \dots)$, we can model them as vectors $\mathbf{x} = [x_0, \dots, x_{n-1}]^\top \in \mathbb{R}^n$, and the filter can be viewed as a linear mapping $F : \mathbb{R}^n \rightarrow \mathbb{R}^{m+n-1}$.

Thus, for the filter, we have a matrix representation of (4.5). Writing $\mathbf{y} = [y_0, \dots, y_{2n-2}]^\top \in \mathbb{R}^{2n-1}$ for the vector of the output signal, we find in the case $m = n$:

$$\begin{bmatrix} y_0 \\ \vdots \\ \vdots \\ y_{m+n-2} \end{bmatrix} = \begin{bmatrix} h_0 & 0 & \cdots & 0 \\ h_1 & & & \\ & \ddots & & \\ h_{m-1} & & & h_1 & h_0 \\ 0 & & & & \\ & \ddots & & & \\ 0 & & & 0 & h_{m-1} \end{bmatrix} \begin{bmatrix} x_0 \\ \vdots \\ \vdots \\ x_{n-1} \end{bmatrix}. \quad (4.6)$$

4.1.3 Discrete convolution

In (4.5) we can recognise a pattern which can be used to simplify the notation further using the definition of discrete convolution.

Definition 4.1.6 (Discrete convolution). For two sequences $f, g \in \ell^\infty(\mathbb{Z})$, their *discrete convolution* $u := f * g (= g * f) \in \ell^\infty(\mathbb{Z})$ is defined by:

$$u_k = \sum_{j \in \mathbb{Z}} f_j g_{k-j} = \sum_{j \in \mathbb{Z}} f_{k-j} g_j. \quad (4.7)$$

Using Definition 4.1.6, we may rewrite (4.5) to:

$$y = F((x_i)_{i \in \mathbb{Z}}) = x * h.$$

One can see this by considering that the convolution y of two finite-length sequences $x \in \mathbb{R}^n$ and $h \in \mathbb{R}^m$ is in \mathbb{R}^{n+m-1} and satisfies the equation

$$y_k = \sum_{j=0}^{n-1} x_j h_{k-j}, \quad k = 0, \dots, n+m-1,$$

where $h_j = 0$ for $j < 0$ and $j \geq m$.

The above introduced *linear* convolution can be related to *periodic/circular* convolution that we will now introduce. This relation is interesting because periodic convolution can be related to the discrete Fourier transform (DFT), which can be implemented in a fast way (FFT). The intuition behind periodic convolution is that for a periodic signal, due to the

repetitions in its entries, simplifications are possible. Note that for now, $(x_j)_{j \in \mathbb{Z}}$ is an n -periodic signal (and hence no longer finite), i.e.

$$x_{j+n} = x_j \quad \forall j \in \mathbb{Z}$$

and h is an LT-FIR filter. In this case, the output signal is again n -periodic. The expression for the output can be rewritten as:

$$\begin{aligned} y_k &= \sum_{j \in \mathbb{Z}} x_j h_{k-j} = \sum_{j \in \mathbb{Z}} x_{k-j} h_j \quad (\text{i.e. } y \text{ is also } n\text{-periodic}) \\ &= \sum_{j=0}^{n-1} \underbrace{\left(\sum_{\ell \in \mathbb{Z}} h_{j+\ell n} \right)}_{\text{periodic summation}} x_{k-j}. \end{aligned}$$

We define $p_j := \sum_{\ell \in \mathbb{Z}} h_{j+\ell n}$, where $h = (\dots, 0, h_0, \dots, h_{n-1}, 0, \dots)$ is the impulse response.

By definition:

$$p_{j+n} = p_j \quad \forall j \in \mathbb{Z},$$

and thus p is n -periodic.

Thus,

$$y_k = \sum_{j=0}^{n-1} p_j x_{k-j} = \sum_{j=0}^{n-1} p_{k-j} x_j, \quad k \in \mathbb{Z}, \quad (4.8)$$

since convolution is commutative.

This convolution of two n -periodic signals is called *periodic convolution*:

Definition 4.1.7 (Discrete periodic convolution). The *discrete periodic convolution* of two n -periodic sequences $(p_k)_{k \in \mathbb{Z}}$, $(x_k)_{k \in \mathbb{Z}}$ yields the n -periodic sequence:

$$(y_k) := (p_k) *_{\text{n}} (x_k) \quad , \quad y_k := \sum_{j=0}^{n-1} p_{k-j} x_j = \sum_{j=0}^{n-1} x_{k-j} p_j, \quad k \in \mathbb{Z}.$$

In matrix notation, (4.8) can be written as:

$$\begin{bmatrix} y_0 \\ \vdots \\ \vdots \\ y_{n-1} \end{bmatrix} = \underbrace{\begin{bmatrix} p_0 & p_{n-1} & p_{n-2} & \cdots & \cdots & p_1 \\ p_1 & p_0 & p_{n-1} & & & \vdots \\ p_2 & p_1 & p_0 & \ddots & & \\ \vdots & & \ddots & \ddots & \ddots & \\ \vdots & & & \ddots & \ddots & \ddots \\ \vdots & & & & \ddots & \ddots & p_{n-1} \\ p_{n-1} & \cdots & & & p_1 & p_0 \end{bmatrix}}_{=: \mathbf{P}} \begin{bmatrix} x_0 \\ \vdots \\ \vdots \\ x_{n-1} \end{bmatrix}. \quad (4.9)$$

where $(\mathbf{P})_{ij} = p_{i-j}, 0 \leq i, j \leq n-1$.

The coefficients p_0, \dots, p_{n-1} can be regarded as a *periodic impulse response*.

We may write out the first two equations defined by (4.9) as

$$\begin{aligned} y_0 &= p_0 x_0 + p_{-1} x_1 + \dots + p_{-(n-1)} x_{n-1} \\ &= p_0 x_0 + p_{n-1} x_1 + \dots + p_1 x_{n-1}, \\ y_1 &= p_1 x_0 + p_0 x_1 + \dots + p_{-(n-2)} x_{n-1} \\ &= p_1 x_0 + p_0 x_1 + \dots + p_2 x_{n-1}. \end{aligned}$$

4.1.4 Circulant matrices

The matrix \mathbf{P} in (4.9) has a very special structure:

Definition 4.1.8 (Circulant matrix). A matrix $\mathbf{C} = [c_{ij}]_{i,j=0}^{n-1} \in \mathbb{K}^{n,n}$ is *circulant* if and only if there exists an n -periodic sequence $(p_k)_{k \in \mathbb{Z}}$ such that:

$$c_{ij} = p_{j-i}, 0 \leq i, j \leq n-1.$$

✎ Notation: We write $\text{circ}(\mathbf{p}) \in \mathbb{K}^{n,n}$ for the circulant matrix generated by the periodic sequence/vector $\mathbf{p} = [p_0, \dots, p_{n-1}]^\top \in \mathbb{K}^n$.

Structure of a generic circulant matrix (“constant diagonals”):

$$\text{circ}(\mathbf{p}) = \begin{bmatrix} p_0 & p_1 & p_2 & \cdots & \cdots & p_{n-1} \\ p_{n-1} & p_0 & & & & p_{n-2} \\ p_{n-2} & & & & & \vdots \\ \vdots & & & & & \\ \vdots & & & & & \\ p_2 & & & & & p_1 \\ p_1 & p_2 & \cdots & \cdots & p_{n-1} & p_0 \end{bmatrix} \in \mathbb{K}^{n,n}.$$

The circulant matrix $\mathbf{P} = \text{circ}(\mathbf{p}) \in \mathbb{K}^{n,n}$ can be uniquely constructed from a sequence/vector $\mathbf{p} = [p_0, \dots, p_{n-1}]^\top \in \mathbb{K}^n$. Thus, the information content of the circulant matrix \mathbf{P} is n numbers in the field \mathbb{K} . Circulant matrices have the following properties:

1. The main diagonal as well as the sub- and super-diagonals are constant.
2. The columns/rows arise by cyclic permutation from the first column/row.

One can readily see that discrete periodic convolution is the same as multiplying by a circulant matrix.

Reduction to periodic convolution

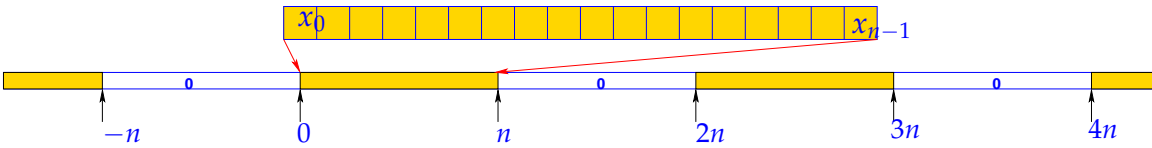
Next, we show how *linear* convolution can be expressed as *periodic* convolution. Recall that the discrete linear convolution of two vectors $\mathbf{x} = (x_0, \dots, x_{n-1})^\top \in \mathbb{K}^n$, $\mathbf{h} = (h_0, \dots, h_{n-1})^\top \in \mathbb{K}^n$ is defined as:

$$y_k := (\mathbf{x} * \mathbf{h})_k = \sum_{j=0}^{n-1} x_j h_{k-j}, \quad k = 0, \dots, 2n-2.$$

Now expand x_0, \dots, x_{n-1} and h_0, \dots, h_{n-1} to $(2n-1)$ -periodic sequences by *zero padding*.

$$\tilde{x}_k := \begin{cases} x_k, & \text{if } 0 \leq k < n, \\ 0, & \text{if } n \leq k < 2n-1, \end{cases}, \quad \tilde{h}_k := \begin{cases} h_k, & \text{if } 0 \leq k < n, \\ 0, & \text{if } n \leq k < 2n-1, \end{cases} \quad (4.10)$$

and periodic extension: $\tilde{x}_k = \tilde{x}_{2n-1+k}$, $\tilde{h}_k = \tilde{h}_{2n-1+k}$ for all $k \in \mathbb{Z}$. The zero components prevent interaction of different periods.



$$\triangleright \quad (\mathbf{x} * \mathbf{h})_k = (\tilde{\mathbf{x}} *_{2n-1} \tilde{\mathbf{h}})_k, \quad k = 0, \dots, 2n-2. \quad (4.11)$$

In the spirit of (4.6) we can switch to a matrix view of the reduction to periodic convolution:

$$\begin{bmatrix} y_0 \\ \vdots \\ \vdots \\ y_{2n-2} \end{bmatrix} = \underbrace{\begin{bmatrix} \tilde{h}_0 & 0 & \cdots & 0 & \tilde{h}_{n-1} & \cdots & \tilde{h}_1 & \tilde{h}_0 \\ \tilde{h}_1 & & & & 0 & & & \\ & & & & & & & \\ & & & & 0 & & & \\ \tilde{h}_{n-1} & \cdots & \tilde{h}_1 & \tilde{h}_0 & 0 & \cdots & 0 & \tilde{h}_{n-1} \\ 0 & & & \tilde{h}_1 & \tilde{h}_0 & & & \\ & & & & & & & \\ & & & & & & & \\ 0 & \cdots & 0 & \tilde{h}_{n-1} & \cdots & \tilde{h}_1 & \tilde{h}_0 & 0 \end{bmatrix}}_{\text{a } (2n-1) \times (2n-1) \text{ circulant matrix!}} \begin{bmatrix} \tilde{x}_0 \\ \vdots \\ \vdots \\ \tilde{x}_{n-1} \\ 0 \\ \vdots \\ \vdots \\ 0 \end{bmatrix}. \quad (4.12)$$

Note. We conclude that discrete convolution can be realized by multiplication with a circulant matrix.

4.2 Discrete Fourier Transform (DFT)

Algorithms dealing with circulant matrices make use of their very special spectral properties. Full understanding requires familiarity with the theory of eigenvalues and eigenvectors of matrices from linear algebra, see [3, Ch. 7], [4, Ch. 9].

4.2.1 Eigenvalues and eigenvectors of circulant matrices

Recall from linear algebra the eigenvalue/eigenvector equation: Let $\mathbf{C} \in \mathbb{K}^{n,n}$

$$\mathbf{C}\mathbf{v}_k = \lambda_k \mathbf{v}_k, \quad (4.13)$$

where λ_k is called eigenvalue of the matrix \mathbf{C} to the eigenvector $\mathbf{v}_k \in \mathbb{K}^n$. Consider the n -th root of unity:

$$\omega_n := \exp\left(-\frac{2\pi i}{n}\right) = \cos\left(\frac{2\pi}{n}\right) - i \sin\left(\frac{2\pi}{n}\right), \quad n \in \mathbb{N}, \quad (4.14)$$

so that

$$\omega_n^n = \exp(-2\pi i) = 1.$$

We know:

$$\omega_n^{-\ell} = \overline{\omega_n^\ell}, \text{ where } \ell \in \mathbb{Z},$$

and

$$\omega_n^{jk} = (\omega_n^j)^k = (\omega_n^k)^j.$$

where for any complex number $z \in \mathbb{C}$, \bar{z} denotes its complex conjugation. With this, we can now define the vectors:

$$\mathbf{v}_k := [\omega_n^{-jk}]_{j=0}^{n-1} \in \mathbb{C}^n, \quad k \in \{0, \dots, n-1\}, \quad (4.15)$$

and represent the matrix \mathbf{C} as:

$$\mathbf{C}_{i,j} = c_{ij} := p_{i-j} \quad \text{for } n\text{-periodic } \mathbf{p} \in \ell^\infty(\mathbb{Z}), p_i \in \mathbb{C}.$$

We can now verify that $\mathbf{v}_0, \dots, \mathbf{v}_{n-1}$ are the eigenvectors of \mathbf{C} :

$$\begin{aligned} (\mathbf{C}\mathbf{v}_k)_j &= \sum_{\ell=0}^{n-1} c_{j\ell} (\mathbf{v}_k)_\ell = \sum_{\ell=0}^{n-1} p_{j-\ell} \omega_n^{-\ell k}, \\ &= \sum_{\ell=0}^{n-1} p_\ell \omega_n^{-(j-\ell)k} = \omega_n^{-jk} \underbrace{\sum_{\ell=0}^{n-1} p_\ell \omega_n^{\ell k}}_{=: \lambda_k}, \\ &= (\mathbf{v}_k)_j \cdot \lambda_k. \end{aligned}$$

With this, we have also found the eigenvalues λ_k of \mathbf{C} to be

$$\lambda_k = \sum_{\ell=0}^{n-1} p_\ell \omega_n^{\ell k}. \quad (4.16)$$

Note. The eigenvalues $\lambda_0, \dots, \lambda_{n-1}$ depend on \mathbf{C} . But, the eigenvectors $\mathbf{v}_0, \dots, \mathbf{v}_{n-1}$ are independent of \mathbf{C} !

Thus all circulant matrices of the same dimensions have the same set of eigenvectors! The set $\{\mathbf{v}_0, \dots, \mathbf{v}_{n-1}\} \subset \mathbb{C}^n$ provides the so-called *orthogonal* trigonometric basis of \mathbb{C}^n :

$$\{\mathbf{v}_0, \dots, \mathbf{v}_{n-1}\} = \left\{ \begin{bmatrix} \omega_n^0 \\ \vdots \\ \omega_n^0 \end{bmatrix}, \begin{bmatrix} \omega_n^0 \\ \omega_n^{-1} \\ \vdots \\ \omega_n^{-(n-1)} \end{bmatrix}, \dots, \begin{bmatrix} \omega_n^0 \\ \omega_n^{-(n-2)} \\ \omega_n^{-2(n-2)} \\ \vdots \\ \omega_n^{-(n-1)(n-2)} \end{bmatrix}, \begin{bmatrix} \omega_n^0 \\ \omega_n^{-(n-1)} \\ \omega_n^{-2(n-1)} \\ \vdots \\ \omega_n^{-(n-1)^2} \end{bmatrix} \right\}. \quad (4.17)$$

From (4.15) we can conclude orthogonality of the basis vectors by straightforward computations:

$$\begin{aligned} \mathbf{v}_k^H \mathbf{v}_m &= \sum_{j=0}^{n-1} \omega_n^{jk} \omega_n^{-jm} = \sum_{j=0}^{n-1} \omega_n^{j(k-m)} \\ &= \begin{cases} \frac{1-\omega_n^{(k-m)n}}{1-\omega_n^{k-m}} = \frac{1-\exp^{-2\pi i \frac{(k-m)n}{n}}}{1-\exp^{-2\pi i \frac{(k-m)}{n}}} = 0 & k \neq m, \\ \sum_{j=0}^{n-1} \underbrace{\omega_n^{jm} \omega_n^{-jm}}_{=1} = n & k = m. \end{cases} \end{aligned}$$

The matrix effecting the change of basis (*trigonometric basis* \rightarrow *standard basis*) is called the *Fourier-matrix*:

$$\mathbf{F}_n = \begin{bmatrix} \omega_n^0 & \omega_n^0 & \cdots & \omega_n^0 \\ \omega_n^0 & \omega_n^1 & \cdots & \omega_n^{n-1} \\ \omega_n^0 & \omega_n^2 & \cdots & \omega_n^{2n-2} \\ \vdots & \vdots & & \vdots \\ \omega_n^0 & \omega_n^{n-1} & \cdots & \omega_n^{(n-1)^2} \end{bmatrix} = [\omega_n^{lj}]_{l,j=0}^{n-1} \in \mathbb{C}^{n,n}. \quad (4.18)$$

Lemma 4.2.1 (Properties of Fourier matrix). *The scaled Fourier-matrix $\frac{1}{\sqrt{n}}\mathbf{F}_n$ is unitary (see Definition 3.3.1) :*

$$\mathbf{F}_n^{-1} = \frac{1}{n}\mathbf{F}_n^H = \frac{1}{n}\bar{\mathbf{F}}_n.$$

Note that (4.13) can be rewritten in compact notation as:

$$\mathbf{C}\bar{\mathbf{F}}_n = \bar{\mathbf{F}}_n \text{diag}(\lambda_0, \dots, \lambda_{n-1}).$$

Therefore,

$$\begin{aligned} \mathbf{C} &= \bar{\mathbf{F}}_n \text{diag}(\lambda_0, \dots, \lambda_{n-1})(\bar{\mathbf{F}}_n)^{-1}, \\ \mathbf{C} &= \bar{\mathbf{F}}_n \text{diag}\left(\underbrace{\mathbf{F}_n \mathbf{p}}_{\lambda_k = \mathbf{v}_k^H \mathbf{p}}\right)(\bar{\mathbf{F}}_n)^{-1}. \end{aligned} \quad (4.19)$$

Equivalently, from Lemma 4.2.1,

$$\mathbf{C} = \mathbf{F}_n^{-1} \text{diag}(\mathbf{F}_n \mathbf{p}) \mathbf{F}_n.$$

Thus we have proven the following lemma:

Lemma 4.2.2 (Diagonalization of circulant matrices). *For any circulant matrix $\mathbf{C} \in \mathbb{K}^{n,n}$, $c_{ij} = p_{i-j}$, where $(p_k)_{k \in \mathbb{Z}}$ is a n -periodic sequence, the following holds true:*

$$\mathbf{C}\bar{\mathbf{F}}_n = \bar{\mathbf{F}}_n \text{diag}(d_1, \dots, d_n) \quad , \quad \mathbf{d} = \mathbf{F}_n [p_0, \dots, p_{n-1}]^\top .$$

The mapping $\mathcal{F}_n : y \mapsto \mathbf{F}_n y$ is called the discrete Fourier transform *DFT*:

Definition 4.2.1 (Discrete Fourier transform (DFT)). The linear map $\mathcal{F}_n : \mathbb{C}^n \mapsto \mathbb{C}^n$, $\mathcal{F}_n(\mathbf{y}) := \mathbf{F}_n \mathbf{y}$, $\mathbf{y} \in \mathbb{C}^n$, is called *discrete Fourier transform* (DFT), i.e. for $\mathbf{c} := \mathcal{F}_n(\mathbf{y})$:

$$c_k = \sum_{j=0}^{n-1} y_j \omega_n^{kj} \quad , \quad k = 0, \dots, n-1 . \quad (4.20)$$

We can find the inverse discrete Fourier transform by:

$$\begin{aligned} \mathbf{c} = \mathbf{F}_n \mathbf{y} &\Leftrightarrow \mathbf{y} = \mathbf{F}_n^{-1} \mathbf{c} \stackrel{\text{Lemma 4.2.1}}{=} \frac{1}{n} \bar{\mathbf{F}}_n \mathbf{c} \\ c_k = \underbrace{\sum_{j=0}^{n-1} y_j \omega_n^{kj}}_{\text{DFT of } \mathbf{y}} &\Leftrightarrow y_j = \underbrace{\frac{1}{n} \sum_{k=0}^{n-1} c_k \omega_n^{-kj}}_{\text{inverse DFT of } \mathbf{c}} \end{aligned} \quad (4.21)$$

4.2.2 Discrete Convolution via DFT

Computing the discrete periodic convolution of two vectors \mathbf{x} and \mathbf{u} is equivalent to computing the product $\text{DFT}(\mathbf{x}) \cdot \text{DFT}(\mathbf{u})$ and taking the inverse DFT.

Theorem 4.2.1 (Convolution theorem). *The discrete periodic convolution $*_n$ between n -dimensional vectors \mathbf{u} and \mathbf{x} is equal to the inverse DFT of the component-wise product between the DFTs of \mathbf{u} and \mathbf{x} ; i.e.:*

$$(\mathbf{u}) *_n (\mathbf{x}) := \sum_{j=0}^{n-1} u_{k-j} x_j = \mathbf{F}_n^{-1} [(\mathbf{F}_n \mathbf{u})_j (\mathbf{F}_n \mathbf{x})_j]_{j=1}^n .$$

Discrete periodic convolution in Eigen

The EIGEN-functions of discrete Fourier transform and its inverse can be computed via:

$$\text{DFT: } \mathbf{c} = \text{fft.fwd}(\mathbf{y}) \quad \text{and} \quad \text{inverse DFT: } \mathbf{y} = \text{fft.inv}(\mathbf{c});$$

Before using `fft`, remember to:

4 Filtering Algorithms

1. `# include <unsupported/Eigen/FFT>`
2. Instantiate helper class `Eigen::FFT<double> fft;`
(The template argument should always be `double`.)

Code Snippet 4.1: Discrete periodic convolution: DFT implementation → GITLAB

```
7 VectorXcd pconvfft(const VectorXcd& u, const VectorXcd& x) {  
8     Eigen::FFT<double> fft;  
9     VectorXcd tmp = ( fft.fwd(u) ).cwiseProduct( fft.fwd(x) );  
10    return fft.inv(tmp);  
11 }  
12 /*
```

In (4.10) we first saw that the discrete convolution of n -vectors can be accomplished by the *periodic* discrete convolution of $(2n - 1)$ -vectors (obtained by zero padding):

Code Snippet 4.2: Implementation of discrete convolution (see definition 4.1.6) based on periodic discrete convolution → GITLAB

```
4 VectorXcd myconv(const VectorXcd& h, const VectorXcd& x) {  
5     const long n = h.size();  
6     // Zero padding, cf. (4.10)  
7     VectorXcd hp(2*n - 1), xp(2*n - 1);  
8     hp << h, VectorXcd::Zero(n - 1);  
9     xp << x, VectorXcd::Zero(n - 1);  
10    // Periodic discrete convolution of length 2n - 1  
11    return pconvfft(hp, xp);  
12 }  
13 /*
```

4.2.3 Fast Fourier Transform (FFT)

So far, introducing the DFT has not brought an advantage, because computing it via $\mathbf{F}_n \mathbf{x}$ has asymptotic complexity $\mathcal{O}(n^2)$. We will now see that a fast implementation of DFT is possible. Such fast implementations are called fast Fourier transform (FFT).

An FFT is any algorithm which can perform DFT within asymptotic complexity of $\mathcal{O}(n \cdot \log_2(n))$.

Idea: Use the divide-and-conquer strategy to implement an algorithm.

Let us assume that n is a power of 2, i.e. $n = 2^L$. An elementary manipulation of (4.20) for

$n = 2m$, $m \in \mathbb{N}$ yields:

$$\begin{aligned}
 c_k &= \sum_{j=0}^{n-1} y_j e^{-\frac{2\pi i}{n} jk} \\
 &= \sum_{j=0}^{m-1} y_{2j} e^{-\frac{2\pi i}{n} 2jk} + \sum_{j=0}^{m-1} y_{2j+1} e^{-\frac{2\pi i}{n} (2j+1)k} \\
 &= \underbrace{\sum_{j=0}^{m-1} y_{2j} e^{-\frac{2\pi i}{m} jk}}_{=: \tilde{c}_k^{\text{even}}} + e^{-\frac{2\pi i}{n} k} \cdot \underbrace{\sum_{j=0}^{m-1} y_{2j+1} e^{-\frac{2\pi i}{m} jk}}_{=: \tilde{c}_k^{\text{odd}}} .
 \end{aligned} \tag{4.22}$$

Note the m -periodicity: $\tilde{c}_k^{\text{even}} = \tilde{c}_{k+m}^{\text{even}}$, $\tilde{c}_k^{\text{odd}} = \tilde{c}_{k+m}^{\text{odd}}$.

Note: $\tilde{c}_k^{\text{even}}, \tilde{c}_k^{\text{odd}}$ form DFTs of length $m = \frac{n}{2}$!

If we define $\mathbf{y}_{\text{even}} := (y_0, y_2, \dots, y_{n-2})^\top \in \mathbb{C}^m$, $\mathbf{y}_{\text{odd}} := (y_1, y_3, \dots, y_{n-1})^\top \in \mathbb{C}^m$, then: $(\tilde{c}_k^{\text{even}})_{k=0}^{m-1} = \mathbf{F}_m \mathbf{y}_{\text{even}}$, $(\tilde{c}_k^{\text{odd}})_{k=0}^{m-1} = \mathbf{F}_m \mathbf{y}_{\text{odd}}$.

(4.22): DFT of length $2m = 2 \times$ DFT of length $m + 2m$ additions & multiplications

Idea: Divide & conquer recursion to obtain an *FFT-algorithm*.

Recursive demonstration code for DFT of length $n = 2^L$:

Code Snippet 4.3: Recursive FFT → GITLAB

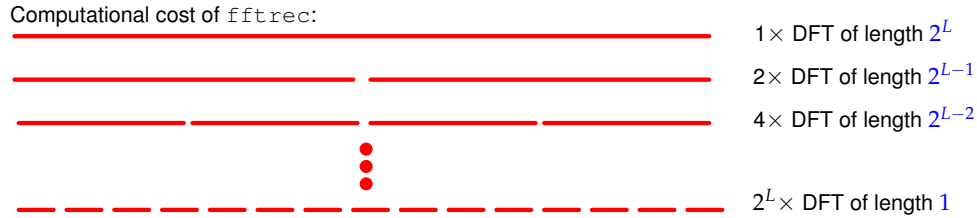
```

8 // Recursive DFT for vectors of length  $n = 2^L$ 
9 VectorXcd fftrec(const VectorXcd& y) {
10     using idx_t = VectorXcd::Index;
11     using comp = std::complex<double>;
12     // Nothing to do for DFT of length 1
13     const idx_t n = y.size();
14     if (n == 1) return y;
15     if (n % 2 != 0) throw std::runtime_error("size(y) must be even!");
16     const Eigen::Map<const Eigen::Matrix<comp, Eigen::Dynamic, Eigen::Dynamic
        ↪ , Eigen::RowMajor>>
17     Y(y.data(), n/2, 2); // for selecting even and off components
18     const VectorXcd c1 = fftrec(Y.col(0)), c2 = fftrec(Y.col(1));
19     const comp omega = std::exp(-2*M_PI/n*comp(0,1)); // root of unity  $\omega_n$ 
20     comp s(1.0, 0.0);
21     VectorXcd c(n);
22     // Scaling of DFT of odd components plus periodic copying
23     for (long k = 0; k < n; ++k) {
24         c(k) = c1(k%(n/2)) + c2(k%(n/2))*s;
25         s *= omega;
26     }
27     return c;

```

```
28 }
29 /*
```

Code Snippet 4.3: Each level of the recursion requires $\mathcal{O}(2^L)$ elementary operations.



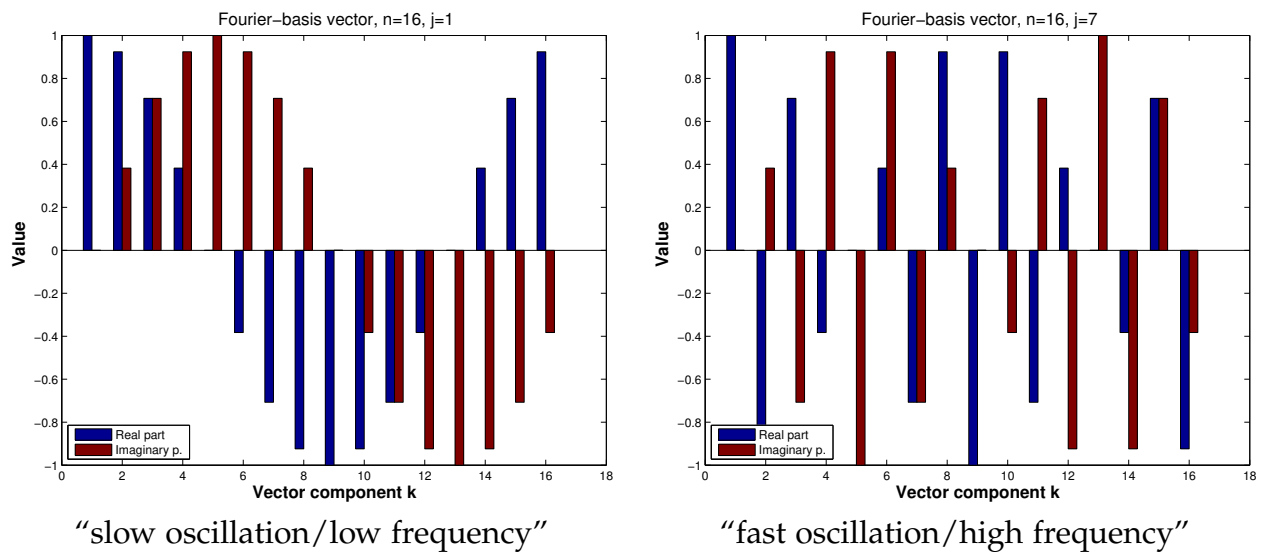
Asymptotic complexity

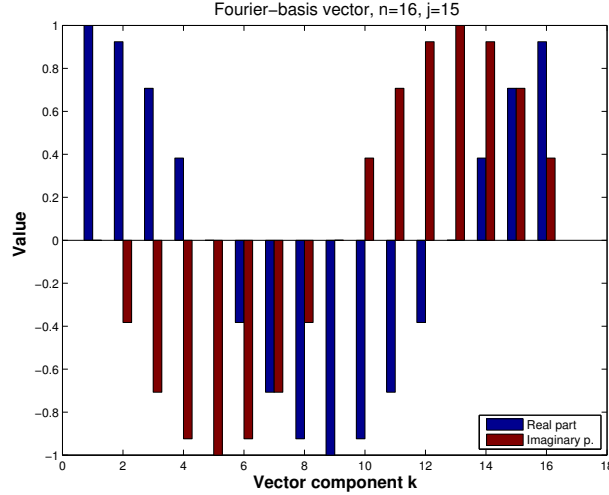
Asymptotic complexity of FFT algorithm, $n = 2^L$: $\mathcal{O}(L2^L) = \mathcal{O}(n \log_2 n)$

`fft.fwd()/fft.inv()`-function calls: computational cost is $\approx 5n \log_2 n$.

4.2.4 Frequency filtering via DFT

Given a signal $\mathbf{x} = [x_0, \dots, x_{n-1}]^\top$, what is the actual information about \mathbf{x} contained in $\mathbf{F}_n \mathbf{x}$? This question is similar to the one addressed in the section about data compression, but here we use the knowledge about the Fourier matrix. The trigonometric basis vectors, when interpreted as time-periodic signals, represent harmonic oscillations. This is illustrated when plotting some vectors of the trigonometric basis ($n = 16$):





“slow oscillation/low frequency”

Note. Dominant coefficients of a signal after transformation to trigonometric basis indicate dominant frequency components.

Terminology: The original signal is referred to as the signal in *time domain* while its coefficients w.r.t. trigonometric basis is referred to as the signal represented in *frequency domain*.

Recall the definition of the DFT (4.20) and inverse DFT (4.21):

$$c_k = \sum_{j=0}^{n-1} y_j \omega_n^{kj} \quad \Leftrightarrow \quad y_j = \frac{1}{n} \sum_{k=0}^{n-1} c_k \omega_n^{-kj} \quad (4.21)$$

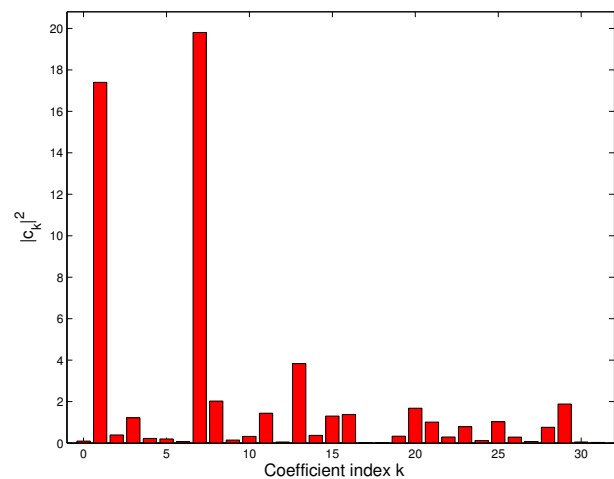
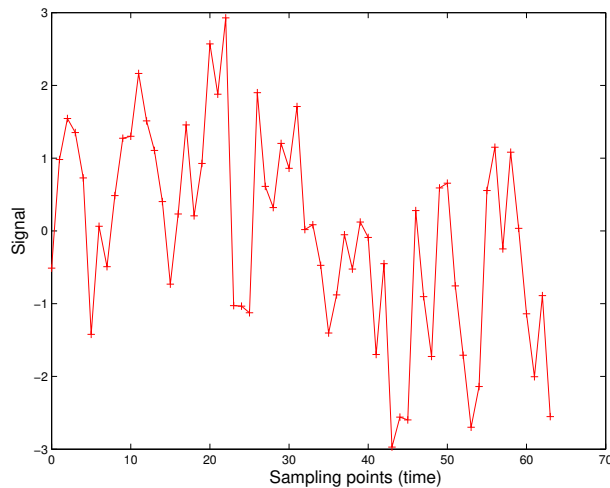
Consider a vector \mathbf{y} with only real components y_k . Then, $c_k = \bar{c}_{n-k}$, because $\omega_n^{kj} = \bar{\omega}_n^{(n-k)j}$, and $n = 2m + 1$. We can use this symmetry to deduce the following:

$$\begin{aligned} ny_j &= c_0 + \sum_{k=1}^m c_k \omega_n^{-kj} + \sum_{k=m+1}^{2m} c_k \omega_n^{-kj} = c_0 + \sum_{k=1}^m c_k \omega_n^{-kj} + c_{n-k} \omega_n^{(k-n)j} \\ &= c_0 + 2 \sum_{k=1}^m \Re(c_k) \cos(2\pi \frac{kj}{n}) + \Im(c_k) \sin(2\pi \frac{kj}{n}), \end{aligned}$$

since $\omega_n^\ell = \cos(2\pi \frac{\ell}{n}) + i \sin(2\pi \frac{\ell}{n})$.

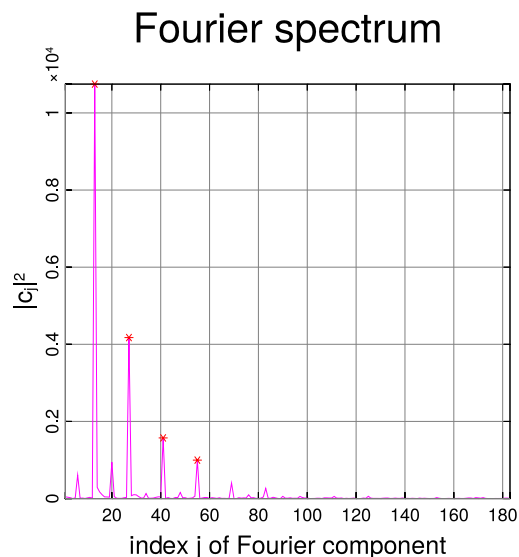
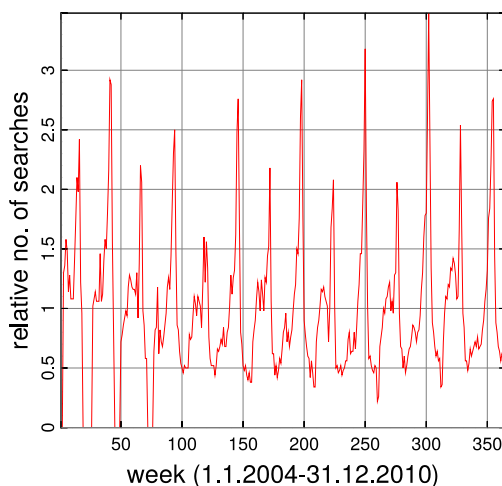
Therefore, $|c_k|, |c_{n-k}|$ measures the strength with which an oscillation with frequency k is represented in the signal, $0 \leq k \leq \lfloor \frac{n}{2} \rfloor$.

4 Filtering Algorithms



The magnitude squared of the signal's DFT locates which frequencies are present in the signal and how much they are present.

Google: 'Vorlesungsverzeichnis'



The pronounced peaks in the power spectrum point to a periodic structure of the data. The location of peaks contains information about the lengths of dominant periods.

“Low” and “high” frequencies

When a signal is captured, processed or transmitted it can become corrupted by noise. One approach to remove such unwanted noise is to model it as an additive, high-frequency component. The signal can then be denoised by applying a filter that cuts off high frequency components:

- ① Transform the noisy signal to frequency domain.
- ② Apply low-pass filter (i.e. cut-off high frequencies).
- ③ Transform back to time/space domain to obtain a denoised signal.

Frequency filtering of real discrete periodic signals by suppressing certain *Fourier coefficients* can be performed as follows:

Code Snippet 4.4: DFT-based frequency filtering → GITLAB

```

7 void freqfilter(const VectorXd& y, int k,
8   VectorXd& low, VectorXd& high) {
9   const VectorXd::Index n = y.size();
10  if (n%2 != 0)
11    throw std::runtime_error("Even vector length required!");
12  const VectorXd::Index m = y.size()/2;
13
14  Eigen::FFT<double> fft; // DFT helper object
15  VectorXcd c = fft.fwd(y); // Perform DFT of input vector
16
17  VectorXcd clow = c;
18  // Set high frequency coefficients to zero
19  for (int j = -k; j <= +k; ++j) clow(m+j) = 0;
20  // (Complementary) vector of high frequency coefficients
21  VectorXcd chigh = c - clow;
22
23  // Recover filtered time-domain signals
24  low = fft.inv(clow).real();
25  high = fft.inv(chigh).real();
26 }
27 /*

```

(It can be optimised by exploiting $y_j \in \mathbb{R}$ and $c_{\frac{n}{2}-k} = \bar{c}_{\frac{n}{2}+k}$)

Map $y \mapsto \text{low}$ (in Code Snippet 4.4) \triangleq *low pass filter*.

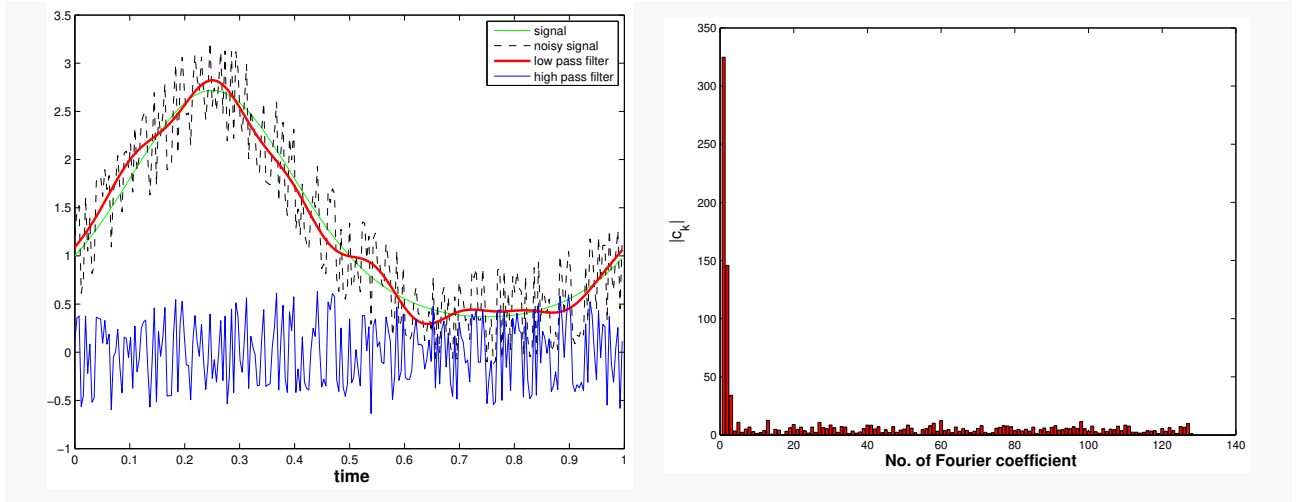
Map $y \mapsto \text{high}$ (in Code Snippet 4.4) \triangleq *high pass filter*.

Example 4.2.1:

Frequency filtering by Code Snippet 4.4 with $k = 120$.

Noisy signal:

```
n = 256; y = exp(sin(2*pi*((0:n-1)')/n)) + 0.5*sin(exp(1:n)');
```

4.2.5 Two-dimensional DFT

Given a matrix $\mathbf{Y} \in \mathbb{C}^{m,n}$, its 2D DFT is defined as two nested 1D DFTs:

$$(\mathbf{C})_{k_1, k_2} = \sum_{j_1=0}^{m-1} \sum_{j_2=0}^{n-1} y_{j_1, j_2} \omega_m^{j_1 k_1} \omega_n^{j_2 k_2} = \sum_{j_1=0}^{m-1} \omega_m^{j_1 k_1} \left(\sum_{j_2=0}^{n-1} \omega_n^{j_2 k_2} y_{j_1, j_2} \right), \quad 0 \leq k_1 < m, 0 \leq k_2 < n.$$

Rewriting the above expressions, for all $0 \leq k_1 < m, 0 \leq k_2 < n$:

$$(\mathbf{C})_{k_1, k_2} = \sum_{j_1=0}^{m-1} \left(\mathbf{F}_n(\mathbf{Y})_{j_1, :}^\top \right)_{k_2} \omega_m^{j_1 k_1} \implies \mathbf{C} = \mathbf{F}_m(\mathbf{F}_n \mathbf{Y}^\top)^\top = \mathbf{F}_m \mathbf{Y} \mathbf{F}_n. \quad (4.23)$$

[Recall that in 1D: $\mathbf{c} = \mathbf{F}_n \mathbf{y}$ $\mathbf{y} \in \mathbb{R}^n$]

Furthermore, consider the 2D inverse DFT:

$$\mathbf{C} = \sum_{j_1=0}^{m-1} \sum_{j_2=0}^{n-1} y_{j_1, j_2} (\mathbf{F}_m)_{:, j_1} (\mathbf{F}_n)_{:, j_2}^\top \implies \mathbf{Y} = \mathbf{F}_m^{-1} \mathbf{C} \mathbf{F}_n^{-1} = \frac{1}{mn} \bar{\mathbf{F}}_m \mathbf{C} \bar{\mathbf{F}}_n. \quad (4.24)$$

Code Snippet 4.5: Two-dimensional discrete Fourier transform → GITLAB

```

14 template <typename Scalar>
15 void fft2(Eigen::MatrixXcd &C, const Eigen::MatrixBase<Scalar> &Y) {
16     using idx_t = Eigen::MatrixXcd::Index;
17     const idx_t m = Y.rows(), n=Y.cols();
18     C.resize(m,n);
19     Eigen::MatrixXcd tmp(m,n);
20 
```

```

21 Eigen::FFT<double> fft; // Helper class for DFT
22 // Transform rows of matrix Y
23 for (idx_t k=0;k<m;k++) {
24     Eigen::VectorXcd tv(Y.row(k));
25     tmp.row(k) = fft.fwd(tv).transpose();
26 }
27
28 // Transform columns of temporary matrix
29 for (idx_t k=0;k<n;k++) {
30     Eigen::VectorXcd tv(tmp.col(k));
31     C.col(k) = fft.fwd(tv);
32 }
33 }
34 /*

```

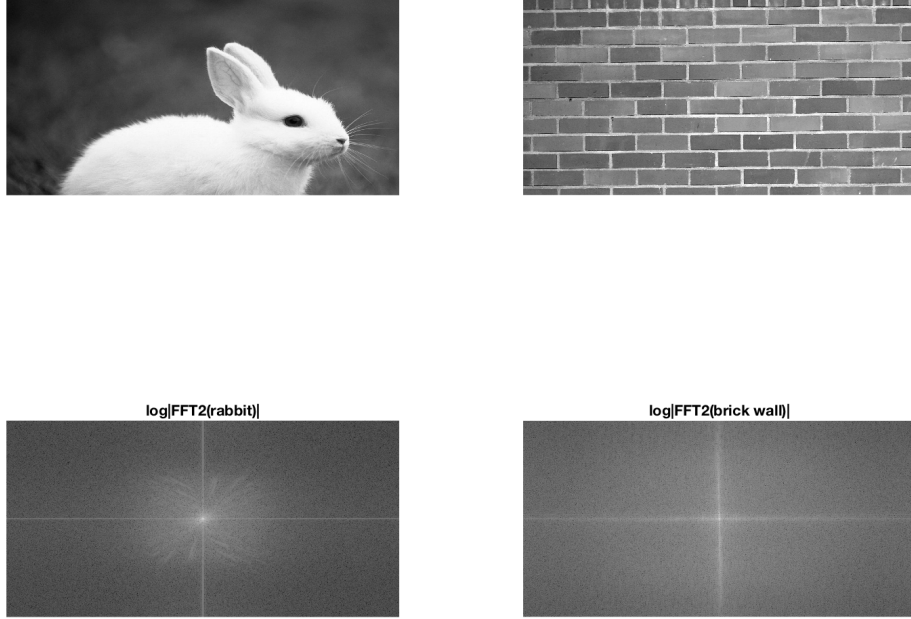
Code Snippet 4.6: Inverse two-dimensional discrete Fourier transform → GITLAB

```

37 template <typename Scalar>
38 void ifft2(Eigen::MatrixXcd &C, const Eigen::MatrixBase<Scalar> &Y) {
39     using idx_t = Eigen::MatrixXcd::Index;
40     const idx_t m = Y.rows(), n=Y.cols();
41     fft2(C,Y.conjugate()); C = C.conjugate()/(m*n);
42 }
43 /*

```

Example 4.2.2: Concentration of frequencies in the 2D Fourier space



The FFT of the brick wall is more concentrated because it has a periodic structure which lends it a sharper frequency response. On the other hand, the FFT of the rabbit is more spread out since the image can only be described using a wide spectrum of frequencies.

Filtering with 2D DFT

As in the 1D case, discrete periodic convolution can be linked to the DFT via the convolution theorem. Thus, to perform discrete linear convolution, it suffices to zero-pad the 2D signals accordingly and apply the following theorem:

Theorem 4.2.2 (2D convolution theorem). *Let $\mathbf{U}, \mathbf{X} \in \mathbb{C}^{m,n}$ and let the 2D discrete periodic convolution $\mathbf{U} *_{m,n} \mathbf{X}$ be defined by:*

$$(\mathbf{U} *_{m,n} \mathbf{X})_{k,l} := \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} (\mathbf{U})_{i,j} (\mathbf{X})_{(k-i) \bmod m, (l-j) \bmod n}$$

Then,

$$\begin{aligned} \mathbf{U} *_{m,n} \mathbf{X} &= \frac{1}{nm} \bar{\mathbf{F}}_m \left[(\mathbf{F}_m \mathbf{U} \mathbf{F}_n)_{i,j} \cdot^{component\ wise} (\mathbf{F}_m \mathbf{X} \mathbf{F}_n)_{i,j} \right]_{i=0, \dots, m-1, j=0, \dots, n-1} \bar{\mathbf{F}}_n, \\ \mathbf{U} *_{m,n} \mathbf{X} &= \text{IDFT2} \{ [\text{DFT2}(\mathbf{U})]_{i,j} \cdot [\text{DFT2}(\mathbf{X})]_{i,j} \}_{i=0, \dots, m-1, j=0, \dots, n-1}. \end{aligned}$$

Code Snippet 4.7: DFT-based 2D discrete periodic convolution → GITLAB

```

68 // DFT based implementation of 2D periodic convolution
69 template <typename Scalar1,typename Scalar2,class EigenMatrix>
70 void pmconv(const Eigen::MatrixBase<Scalar1> &X,const Eigen::MatrixBase<
    ↪ Scalar2> &Y,
71             EigenMatrix &Z) {
72     using Comp = std::complex<double>;
73     using idx_t = typename EigenMatrix::Index;
74     using val_t = typename EigenMatrix::Scalar;
75     const idx_t n=X.cols(),m=X.rows();
76     if ((m!=Y.rows()) || (n!=Y.cols())) throw std::runtime_error("pmconv:
    ↪ size mismatch");
77     Z.resize(m,n); Eigen::MatrixXcd Xh(m,n),Yh(m,n);
78     // Step ❶: 2D DFT of Y
79     fft2(Yh,(Y.template cast<Comp>()));
80     // Step ❷: 2D DFT of X
81     fft2(Xh,(X.template cast<Comp>()));
82     // Steps ❸, ❹: inverse DFT of component-wise product
83     ifft2(Z,Xh.cwiseProduct(Yh));
84 }
85 /*

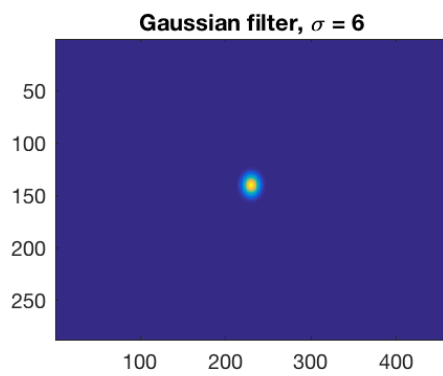
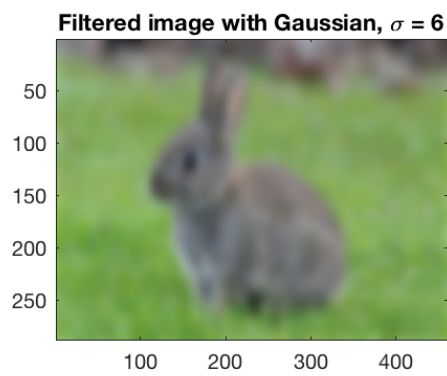
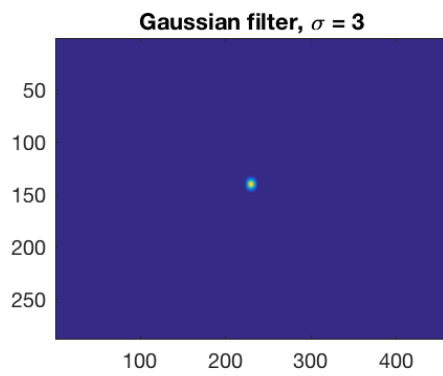
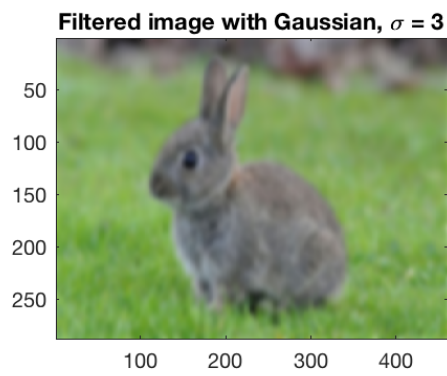
```

Example 4.2.3: Smoothing with Gaussian filter

This example demonstrates how convolving an image with a Gaussian filter can blur the image and the degree of blurring depends on the width of the Gaussian filter.

Original Image

4 Filtering Algorithms



Data Interpolation in 1D

5.1 Introduction

In this chapter, we focus on the problem of data interpolation. Given data values for a discrete set of points, can we define a suitable continuous model for it? That is, can we find a function, that interpolates between the data points? Such an interpolating function allows, for example, to predict the outcome at some intermediate values that are not part of the data set. Also, such an interpolating function is helpful in applications that require the differentiation of the model, which cannot be done on the discrete data set itself. The task of (one-dimensional, scalar) **data interpolation** (point interpolation) can be described as follows:

Given: Data points (t_i, y_i) , where $i = 0, \dots, n, n \in \mathbb{N}, t_i \in I \subset \mathbb{R}, y_i \in \mathbb{R}$

Objective: Reconstruction of a (continuous) function $f : I \rightarrow \mathbb{R}$ satisfying the $n + 1$ interpolation conditions which are given by:

$$f(t_i) = y_i, \quad i = 0, \dots, n. \quad (5.1)$$

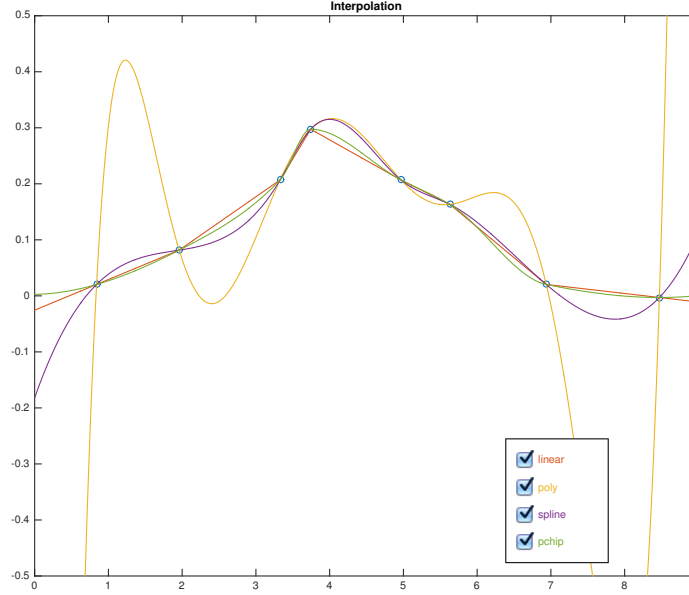
The function f is called an *interpolant* of the given data set $\{(t_i, y_i)\}_{i=0}^n$.

Parlance: The numbers $t_i \in \mathbb{R}$ are called *nodes* and $y_i \in \mathbb{R}$ are called the *(data) values*.

We will assume the minimal requirement on the data that the points t_i are pairwise distinct, i.e. $t_i \neq t_j$, if $i \neq j$ for all $i, j \in \{0, \dots, n\}$.

5 Data Interpolation in 1D

For ease of presentation, we will usually assume that the nodes are ordered: $t_0 < t_1 < \dots < t_n$ and $[t_0, t_n] \subset I$. However, algorithms often must not take sorted nodes for granted. Note that there are many different options to interpolate. Therefore, we need additional assumptions on f such as smoothness properties.

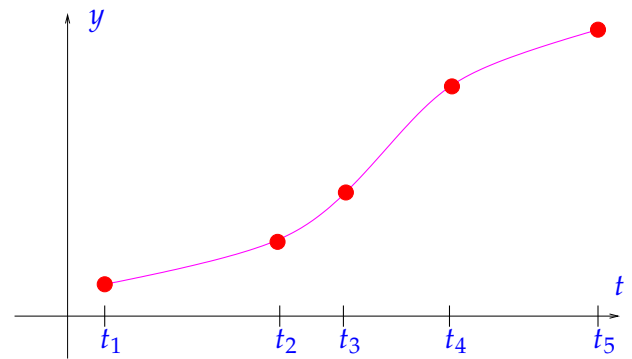


There are infinitely many ways to fix an interpolant for given data points.

Typically, we search for a function $f \in S \subset C^0(I)$, where S is an $(m+1)$ -dimensional subspace, i.e. $S = \text{Span}\{b_0, \dots, b_m\}$, where the $b_j \in C^0(I)$ form a basis of S . In this context, t and y are interpreted as two state variables of a physical system, where t determines y . In other words, a functional dependence $y = y(t)$ is assumed. A direct application of interpolation is the reconstruction of constitutive relationships from measurements.

Examples: t and y could be

t	y
voltage U	current I
pressure p	density ρ
magnetic field H	magnetic flux B
\vdots	\vdots



Known: Several *accurate** measurements (t_i, y_i) , $i = 1, \dots, m$.

Imagine that t and y correspond to the voltage U and electrical current I , respectively, measured for a 2-port non-linear circuit element (like a diode). This element will be part of

*Meaning of attribute “accurate”: justification for interpolation. If measured values y_i were affected by considerable errors, one would not impose the interpolation conditions (5.1), but instead opt for *data fitting* (see Section 3.1).

a circuit, which we want to simulate based on nodal analysis. In order to solve the resulting non-linear system of equations $F(\mathbf{u}) = 0$ for the nodal potentials (collected in the vector \mathbf{u}) by means of Newton's method (see Section 5.2.3), we need the voltage-current relationship for the circuit element as a continuously differentiable function $I = f(U)$.

Note that our task in this section is to algorithmically find a function $f : I \rightarrow \mathbb{R}$, where $I \subset \mathbb{R}$ is an interval. It is not immediately clear how we can do this under the serious restriction that computers can only hold finite data – after all, our hard drives and RAM's are limited. What we will do is to find subroutines that allow us to compute $f(t)$ for all $t \in I$. Let $\{b_0, \dots, b_m\}$ be a basis for S , i.e.,

$$S = \text{Span}\{b_0, \dots, b_m\}.$$

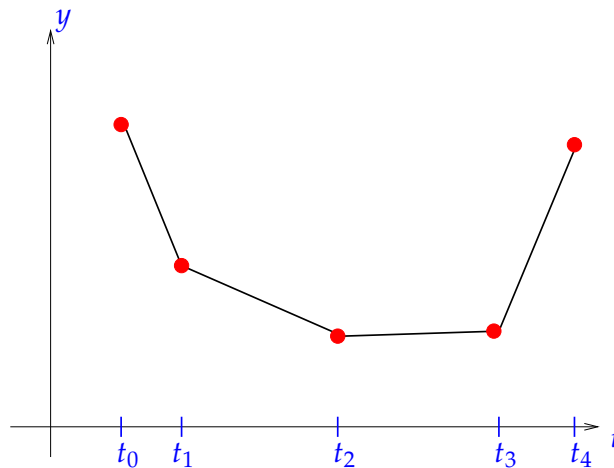
Then, we can find coefficients c_j so that $f \in S$ can be represented as

$$f(t) = \sum_{j=0}^m c_j b_j(t)$$

and thus the finite number of coefficients $\{c_0, \dots, c_m\}$ completely characterize f . The simplest example of this is a piecewise linear interpolation, where we make use of the mapping $x \mapsto ax + b$ which is fully determined by $a, b \in \mathbb{R}$.

5.1.1 Piecewise linear interpolation

We start with the simplest continuous interpolant: Data points (t_i, y_i) , $i = 0, \dots, n$, $t_{i-1} < t_i$, are connected by line segments, that is, we construct an interpolating polygon.



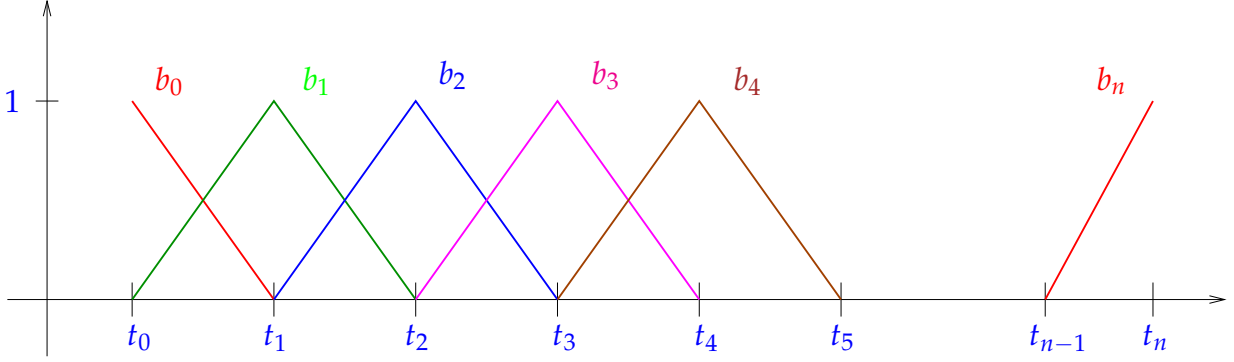
Piecewise linear interpolant of data

For piecewise linear interpolation, the subspace S is:

$$S = \{f \in C^0(I) \mid f(t) = \beta_i t + \gamma_i \text{ on } [t_{i-1}, t_i], \text{ for } i = 0, \dots, n \quad \beta_i, \gamma_i \in \mathbb{R}\}$$

and the points t_i are fixed. One may show that $\dim S = n + 1$ as the choice of y_0, \dots, y_n determines f uniquely, thus yielding $n + 1$ degrees of freedom.

A convenient set of basis functions $\{b_0, \dots, b_n\}$ for S is given by the tent/hat basis functions:



Note. The basis functions have to be extended by zero outside the non-zero range on which they are drawn.

Explicit formulas for these basis functions can be given as:

$$\begin{aligned} b_0(t) &= \begin{cases} 1 - \frac{t-t_0}{t_1-t_0} & \text{for } t_0 \leq t < t_1, \\ 0 & \text{for } t \geq t_1. \end{cases} \\ b_j(t) &= \begin{cases} 1 - \frac{t_j-t}{t_j-t_{j-1}} & \text{for } t_{j-1} \leq t < t_j, \\ 1 - \frac{t-t_j}{t_{j+1}-t_j} & \text{for } t_j \leq t < t_{j+1}, \\ 0 & \text{elsewhere in } [t_0, t_n]. \end{cases}, \quad j = 1, \dots, n-1, \\ b_n(t) &= \begin{cases} 1 - \frac{t_n-t}{t_n-t_{n-1}} & \text{for } t_{n-1} \leq t < t_n, \\ 0 & \text{for } t < t_{n-1}. \end{cases} \end{aligned} \quad (5.2)$$

Moreover, these basis functions are *uniquely* determined by the following conditions:

- b_j is continuous on $[t_0, t_n]$,
- b_j is linear on each subinterval $[t_{i-1}, t_i]$, $i = 1, \dots, n$,
- $b_j(t_i) = \delta_{ij} := \begin{cases} 1 & \text{if } i = j, \\ 0 & \text{else.} \end{cases}$

The last condition implies a simple basis representation of a piecewise linear interpolant of the data points (t_i, y_i) , $i = 0, \dots, n$:

$$\begin{aligned} f(t) &= \sum_{j=0}^n y_j b_j(t), \quad t_0 \leq t \leq t_n, \\ f(t_i) &= \sum_{j=0}^n y_j b_j(t_i) = y_i \underbrace{b_i(t_i)}_{=1} = y_i, \end{aligned} \quad (5.3)$$

where the b_j are given by (5.2). A basis $\{b_0, \dots, b_n\}$ such that $b_j(t_i) = \delta_{ij}$ is called a *cardinal basis*. Note that:

- Both S and the basis $\{b_j\}_{j=0}^n$ depend on the points t_i .
- There are infinitely many choices for a basis of S .
- The cardinal basis for S is unique because the b_j are continuous, piecewise linear and thus uniquely defined on any subinterval.

5.1.2 The general interpolation problem

We recall the setting of the interpolation problem for a general subspace S :

- Interpolating conditions:

$$f(t_i) = y_i, \quad i = 0, \dots, n.$$

- Basis representation:

$$f(t) = \sum_{j=0}^m c_j b_j(t). \quad (5.4)$$

Together, these conditions imply the following:

$$f(t_i) = \sum_{j=0}^m c_j b_j(t_i) = y_i, \quad i = 0, \dots, n. \quad (5.5)$$

This might be rewritten as a matrix equation. Namely:

$$\mathbf{A}\mathbf{c} := \begin{bmatrix} b_0(t_0) & \dots & b_m(t_0) \\ \vdots & & \vdots \\ b_0(t_n) & \dots & b_m(t_n) \end{bmatrix} \begin{bmatrix} c_0 \\ \vdots \\ c_m \end{bmatrix} = \begin{bmatrix} y_0 \\ \vdots \\ y_n \end{bmatrix} =: \mathbf{y}. \quad (5.6)$$

This is an $(m+1) \times (n+1)$ linear system of equations. Thus, solving for $\mathbf{c} = [c_0, \dots, c_m]^\top$ would determine f . Note the following two statements about uniqueness and solvability.

Note. \mathbf{A} is regular \iff For all $\{y_0, \dots, y_n\}$, there exists a unique interpolant.

Note. A necessary condition for unique solvability of the interpolation problem stated in (5.5) is: $m = n$.

If $m = n$ and \mathbf{A} from (5.6) is regular, then for *any* values y_j , $j = 0, \dots, n$ we can find coefficients c_j , $j = 0, \dots, n$. From these, we can build the interpolant:

$$f = \sum_{j=0}^n (\mathbf{A}^{-1}\mathbf{y})_j b_j. \quad (5.7)$$

\Downarrow

For *fixed* nodes t_i , the interpolation problem (5.5) defines the linear mapping

$$\mathcal{I} : \begin{cases} \mathbb{R}^{n+1} & \rightarrow S \subset C^0(I) \\ \text{data space} & \text{function space} \\ \mathbf{y} & \mapsto f = \sum_{j=0}^n \underbrace{(\mathbf{A}^{-1}\mathbf{y})_j}_{c_j} b_j \end{cases}$$

Beware, “linear” in the statement above has nothing to do with a linear function or piecewise linear interpolation. The linearity here is a property of the interpolation operator:

Definition 5.1.1 (Linear interpolation operator). An *interpolation operator* $\mathcal{I} : \mathbb{R}^{n+1} \rightarrow C^0([t_0, t_m])$ for the given nodes $t_0 < t_1 < \dots < t_n$ is called *linear*, if

$$\mathcal{I}(\alpha\mathbf{y} + \beta\mathbf{z}) = \alpha\mathcal{I}(\mathbf{y}) + \beta\mathcal{I}(\mathbf{z}) \quad \forall \mathbf{y}, \mathbf{z} \in \mathbb{R}^{n+1}, \alpha, \beta \in \mathbb{R}. \quad (5.8)$$

 Notation: $C^0([t_0, t_m])$ corresponds to the vector space of continuous functions on $[t_0, t_m]$.

A remaining question is: When is \mathbf{A} invertible? This depends strongly on the nodes t_i and the space S but is independent of the choice of basis $\{b_j\}_{j=0}^n$. Suppose $\{b_0, \dots, b_n\}$ and $\{b'_0, \dots, b'_n\}$ are bases of S and that we want to solve for

$$\sum_{j=0}^n d_j b'_j(t_i) = y_i, \quad i = 0, \dots, n. \quad (5.9)$$

We know that $b'_j \in \text{Span}\{b_0, \dots, b_n\}$. This implies that there exist coefficients $\{\gamma_{k,j}\}_{k=0, \dots, n}$ such that:

$$b'_j(t_i) = \sum_{k=0}^n \gamma_{k,j} b_k(t_i).$$

Therefore,

$$\begin{aligned} (5.9) & \iff \sum_{j=0}^n d_j \left(\sum_{k=0}^n \gamma_{k,j} b_k(t_i) \right) = y_i \\ & \iff \sum_{k=0}^n \left(\sum_{j=0}^n d_j \gamma_{k,j} \right) b_k(t_i) = y_i \end{aligned}$$

Thus, by defining $c_k := \sum_{j=0}^n d_j \gamma_{k,j}$, we see that (5.9) is uniquely solvable if and only if

$$\sum_{k=0}^n c_k b_k(t_i) = y_i, \quad i = 0, \dots, n \quad (5.10)$$

is uniquely solvable. Hence, whether or not the interpolation problem is uniquely solvable is independent of the particular choice of basis. Finally, note that for the choice of the cardinal basis, \mathbf{A} is equal to the identity matrix: $\mathbf{A} = \mathbf{I}$.

5.2 Global Polynomial Interpolation

(Global) polynomial interpolation, that is, interpolation by mapping into spaces of functions spanned by polynomials up to a certain degree, is the simplest interpolation scheme. Despite its simplicity, polynomial interpolation is of great importance as a building block for more complex algorithms.

We will first look at the polynomials of degree less than or equal to n , $n \in \mathbb{N}$:

$$\mathcal{P}_n := \{t \mapsto \sum_{i=0}^n \alpha_i t^i, \alpha_i \in \mathbb{R}\}, \quad (5.11)$$

where α_n is called the *leading coefficient* of the polynomial.

Terminology: The functions $t \mapsto t^k$, $k \in \mathbb{N}_0$, are called *monomials* and $t \mapsto \alpha_n t^n + \alpha_{n-1} t^{n-1} + \dots + \alpha_0$ is called *monomial representation* of a polynomial.

We know that \mathcal{P}_n is a vector space (see [3, Sect. 4.2, Bsp. 4]) of dimension $\dim \mathcal{P}_n = n + 1$. Polynomials are infinitely many times differentiable, i.e., $\mathcal{P}_n \subset C^\infty(\mathbb{R})$.

Why are polynomials important in computational mathematics?

- Integration and differentiation are simple to compute.
- Vector space and algebra.
- Analysis: Taylor polynomials and power series.
- *Efficient* evaluation of a polynomial through the *Horner scheme*:

$$p(t) = t(\dots t(t(\alpha_n t + \alpha_{n-1}) + \alpha_{n-2}) + \dots + \alpha_1) + \alpha_0. \quad (5.12)$$

The following code gives an implementation of the Horner scheme based on vector data types of EIGEN. The function is vectorized in the sense that many evaluation points are processed in parallel.

Code Snippet 5.1: Horner scheme (vectorized version) → GITLAB

```

13 // Efficient evaluation of a polynomial in monomial representation
14 // using the Horner scheme (5.12)
15 // IN: p = vector of monomial coefficients, length = degree + 1
16 // (leading coefficient in p(0), C++ convention)
17 // t = vector of evaluation points t_i
18 // OUT: y = polynomial evaluated at t_i
19 void horner(const VectorXd& p, const VectorXd& t, VectorXd& y) {
20     const VectorXd::Index n = t.size();
21     y.resize(n); y = p(0)*VectorXd::Ones(n);
22     for (unsigned i = 1; i < p.size(); ++i)
23         y = t.cwiseProduct(y) + p(i)*VectorXd::Ones(n);
24 }
25 /*

```

Here: $p := [\alpha_n, \dots, \alpha_0]$. Note that the asymptotic complexity of the Horner scheme is $\mathcal{O}(n)$.

5.2.1 Lagrange Interpolation

We now consider the global polynomial interpolation problem: the sought interpolant belongs to the polynomial space \mathcal{P}_n , i.e., $S = \mathcal{P}_n$:

Definition 5.2.1 (Lagrange polynomial interpolation problem). Given the *simple nodes* t_0, \dots, t_n , for $n \in \mathbb{N}$, $-\infty < t_0 < t_1 < \dots < t_n < \infty$ and the values $y_0, \dots, y_n \in \mathbb{R}$, find $p \in \mathcal{P}_n$ such that

$$p(t_j) = y_j \quad \text{for } j = 0, \dots, n. \quad (5.13)$$

This is a well-defined problem because \mathcal{P}_n is a finite-dimensional space of functions, for which we already know a basis, the monomials. Thus, in principle, we could examine the matrix \mathbf{A} from (5.6) to decide, whether the polynomial interpolant exists and is unique. However, there is a shorter way through building a cardinal basis for $\{t_j\}_{j=0}^n$ and $S = \mathcal{P}_n$.

Lagrange polynomials

For nodes $t_0 < t_1 < \dots < t_n$, the *Lagrange polynomials* are defined as:

$$L_i(t) := \prod_{\substack{j=0 \\ j \neq i}}^n \frac{t - t_j}{t_i - t_j}, \quad i = 0, \dots, n. \quad (5.14)$$

The Lagrange polynomials have the following properties:

- $L_i(t) \in \mathcal{P}_n$.
- $L_i(t_l) = \delta_{il}$. This can be seen by a straightforward calculation:

$$L_i(t_l) = \prod_{\substack{j=0 \\ j \neq i}}^n \frac{t_l - t_j}{t_i - t_j} = \begin{cases} \prod_{j \neq i} \frac{t_i - t_j}{t_i - t_j} = 1 & \text{if } l = i, \\ 0 & \text{if } l \neq i. \end{cases} \quad (5.15)$$

- The set $\{L_0(t), \dots, L_n(t)\}$ is linearly independent. To see this, we consider any arbitrary linear combination that is equal to zero:

$$\gamma_0 L_0(t) + \gamma_1 L_1(t) + \dots + \gamma_n L_n(t) = 0.$$

Then, choosing $t = t_i$ results in

$$\gamma_i \underbrace{L_i(t_i)}_{=1} = 0$$

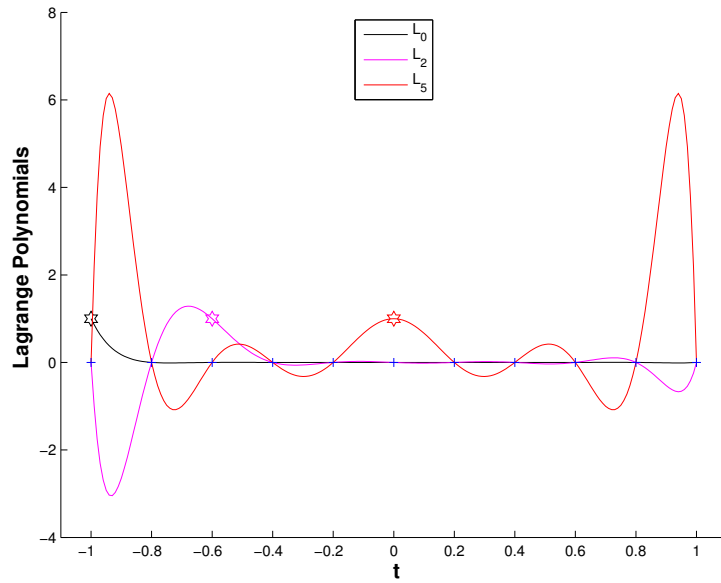
and consequently $\gamma_i = 0$. Hence, we obtain $\gamma_0 = \gamma_1 = \dots = \gamma_n = 0$, which proves the linear independence of the Lagrange polynomials.

Since there are $n + 1$ different Lagrange polynomials which are linearly independent and $\dim \mathcal{P}_n = n + 1$, we conclude that they form a basis of \mathcal{P}_n . We have also seen that $L_i(t_j) = \delta_{ij}$, hence, we conclude that the Lagrange polynomials form a cardinal basis with node set $\{t_j\}_{j=0}^n$. Thus, in the formulation (5.6) we obtain $\mathbf{A} = \mathbf{I}$ so that existence and uniqueness of the solution follow:

Theorem 5.2.1 (Existence & uniqueness of Lagrange interpolation polynomial). *The general Lagrange polynomial interpolation problem admits a unique solution $p \in \mathcal{P}_n$.*

Consider the equidistant nodes in $[-1, 1]$:

$$\mathcal{T} := \{t_j = -1 + \frac{2}{n} j\}, j = 0, \dots, n.$$



The plot shows the Lagrange polynomials for this set of nodes that do not vanish in the nodes t_0 , t_2 , and t_5 , respectively.

The Lagrange polynomial interpolant p for data points $(t_i, y_i)_{i=0}^n$ allows a straightforward representation with respect to the basis of Lagrange polynomials for the node set $\{t_i\}_{i=0}^n$:

$$p(t) = \sum_{i=0}^n y_i L_i(t) \quad \Leftrightarrow \quad p \in \mathcal{P}_n \quad \text{and} \quad p(t_i) = y_i. \quad (5.16)$$

5.2.2 Polynomial interpolation algorithms

Now we consider the algorithmic realization of Lagrange interpolation. The setting is as follows:

Given data points (t_i, y_i) , $i \in \{0, \dots, n\}$, $t_i \in I$, find the interpolant $p(t) \in \mathcal{P}_n$ and evaluate $p(x)$ for some $x \in I$.

5 Data Interpolation in 1D

To evaluate p at an arbitrary point $x \in I$, we need to know the Lagrange polynomials L_i , $i = 0, \dots, n$. Evaluating a single Lagrange polynomial may be done using the Horner scheme and does therefore incur a computational cost of $\mathcal{O}(n)$. Hence, the computational cost of evaluating

$$\sum_{j=0}^n y_j L_j(x) = p(x)$$

is $\mathcal{O}(n^2)$.

Next we want to look at the case of a series of interpolation problems:

- Fixed nodes $t_0, \dots, t_n \in I$.
- N different data value sets $\{y_0^k, \dots, y_n^k\}$, $k \in \{1, \dots, N\}$.
- For every k : find interpolant $p_k \in \mathcal{P}_n$ and evaluate p_k at a series of points $x_k \in I$, $k \in \{1, \dots, N\}$.

In this case, the overall complexity becomes $\mathcal{O}(n^2 N)$.

Next, we consider a more economical approach, in which the Lagrange interpolation is done in a way that we store all the information involving the fixed nodes t_i once and save computational complexity when solving a series of N interpolation problems with the same node set.

Barycentric interpolation formula

By means of pre-calculations, the asymptotic effort can be reduced substantially: Simple manipulations starting from (5.16) give an alternative representation of p :

$$p(t) = \sum_{i=0}^n y_i L_i(t) = \sum_{i=0}^n y_i \prod_{\substack{j=0 \\ j \neq i}}^n \frac{t - t_j}{t_i - t_j} = \sum_{i=0}^n \lambda_i y_i \prod_{\substack{j=0 \\ j \neq i}}^n (t - t_j),$$

with

$$\lambda_i := \frac{1}{(t_i - t_0)(t_i - t_1) \cdots (t_i - t_{i-1})(t_i - t_{i+1}) \cdots (t_i - t_n)}, i = 0, \dots, n.$$

This implies

$$p(t) = \sum_{i=0}^n \frac{\lambda_i}{t - t_i} y_i \cdot \prod_{j=0}^n (t - t_j). \quad (5.17)$$

This representation holds for any interpolating polynomial through the points (t_i, y_i) , so it also holds for $y_0 = \dots = y_n = 1$ which results in the constant polynomial $p(t) \equiv 1$. Thus,

$$1 = \sum_{i=0}^n \frac{\lambda_i}{t - t_i} \prod_{j=0}^n (t - t_j),$$

and hence

$$\prod_{j=0}^n (t - t_j) = \frac{1}{\sum_{i=0}^n \frac{\lambda_i}{t - t_i}}.$$

Plugging this back into (5.17) results in the *Barycentric interpolation formula*:

$$p(t) = \frac{\sum_{i=0}^n \frac{\lambda_i}{t - t_i} y_i}{\sum_{i=0}^n \frac{\lambda_i}{t - t_i}}. \quad (5.18)$$

The total computational effort of this approach consists of

- computation of weights $\lambda_0, \dots, \lambda_n$ with cost $\mathcal{O}(n^2)$ (only once since weights are constant for fixed nodes) and

- evaluating $p(x_k) = \frac{\sum_{i=0}^n y_i^k \frac{\lambda_i}{x_k - t_i}}{\sum_{i=0}^n \frac{\lambda_i}{x_k - t_i}}$ for each k with effort $\mathcal{O}(n)$, $k \in \{1, \dots, N\}$.

Hence, the total asymptotic complexity is $\mathcal{O}(n^2 + Nn)$. For large N , this approach is more efficient than the straightforward evaluation we saw before.

Note. If the nodes t_i are close to each other we get numerical instability when computing the weights λ_i , $i = 0, \dots, n$. This is not a property specific to the barycentric approach but rather of Lagrange interpolation in general (recall the definition of the Lagrange polynomials involving the division by the expressions $(t_i - t_j)$).

Alternatively, we consider a different approach, the *Newton basis*, for polynomial interpolation that does not exhibit these numerical instabilities. This comes at the cost that the Newton basis is no longer a cardinal basis (while the Lagrange polynomials are).

5.2.3 Newton basis

Define the *Newton basis* for the polynomial space \mathcal{P}_n via

$$N_0(t) := 1, \quad N_i(t) := \prod_{j=0}^{i-1} (t - t_j), \quad i = 1, \dots, n. \quad (5.19)$$

Since each N_i has degree i , the set $\{N_0, \dots, N_n\}$ is linearly independent and thus forms a basis of \mathcal{P}_n . By definition, $N_i(t_l) = 0$, for every $l < i$.

We want to find the interpolant $p(t) = \sum_{i=0}^n a_i N_i(t)$:

$$\begin{aligned} p(t_0) &= a_0 + 0, \\ p(t_1) &= a_0 + a_1 N_1(t_1) + 0, \\ p(t_2) &= a_0 + a_1 N_1(t_2) + a_2 N_2(t_2) + 0, \\ &\vdots \\ p(t_n) &= a_0 + a_1 N_1(t_n) + a_2 N_2(t_n) + \dots + a_n N_n(t_n). \end{aligned}$$

5 Data Interpolation in 1D

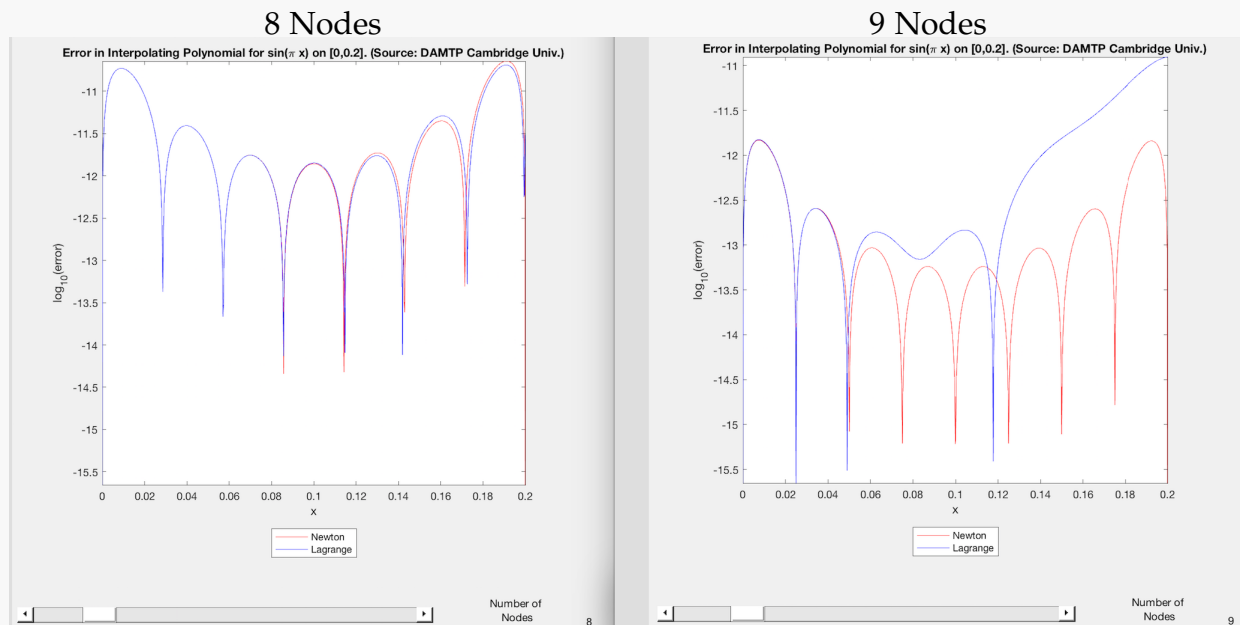
This is a lower triangular system for $[a_0, \dots, a_n]^\top$, which we can write in matrix form as follows:

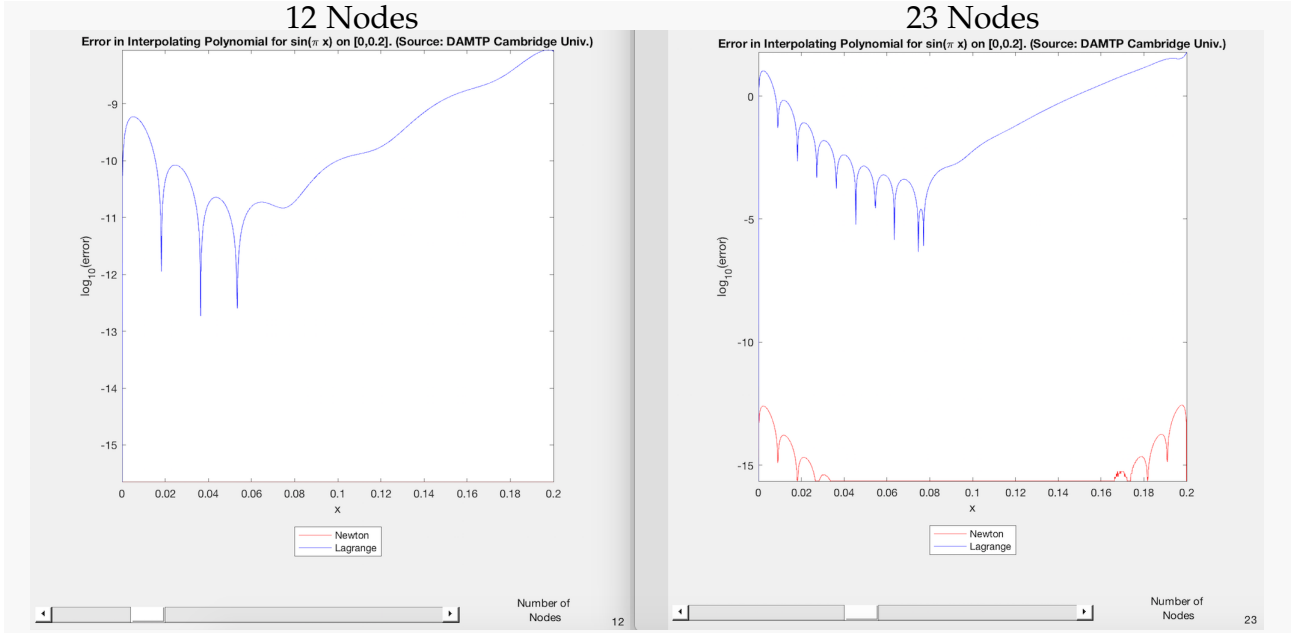
$$\begin{bmatrix} 1 & 0 & \cdots & \cdots & \cdots & 0 \\ 1 & N_1(t_1) & 0 & \cdots & \cdots & 0 \\ \vdots & N_1(t_2) & N_2(t_2) & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \ddots & \vdots \\ 1 & \vdots & \vdots & & \ddots & 0 \\ 1 & N_1(t_n) & N_2(t_n) & & & N_n(t_n) \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ \vdots \\ \vdots \\ a_n \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ \vdots \\ \vdots \\ y_n \end{bmatrix}.$$

Computing the i -th row of the above matrix can be performed in $\mathcal{O}(i)$. This is because of the lower triangular form and since $N_j(t_i) = N_{j-1}(t_i)(t_i - t_j)$ allows to compute the entries of each row recursively. Hence, assembling the matrix can be performed in $\mathcal{O}(\sum_{i=1}^n i) = \mathcal{O}(n^2)$. However, note that this matrix has to be computed only once. Solving the above system for $[a_0, \dots, a_n]^\top$ incurs a cost of $\mathcal{O}(n^2)$ as the involved matrix is lower triangular and we may therefore use forward substitution. The evaluation of $p(x)$ at some point $x \in I$ comes with an additional cost of $\mathcal{O}(n^2)$, since evaluating each $N_i(x)$ with the Horner scheme takes the effort $\mathcal{O}(i)$. This computational effort can be improved through a more sophisticated scheme (divided difference scheme), which we do not discuss in this lecture.

Example 5.2.1: Polynomial interpolation for $\sin(\pi x)$ on $[0, 0.2]$.

We compare the error of interpolating with Lagrange (blue) and Newton (pink) whereby the nodes are equidistant.





Experimentally, the Newton interpolation is more robust. In addition, we find that higher degree polynomials do not necessarily lead to better approximations. This is called *Runge's phenomenon*: Interpolation with high degree polynomials may lead to oscillatory behavior of the interpolant close to the endpoints of the interval (when equidistant nodes are used).

We can avoid such a behaviour with the following techniques:

- Distribute the interpolation points more densely at the endpoints (*cf.* Chebyshev nodes, see Section 5.2.5).
- Avoid high degree polynomials by constructing a piecewise polynomial interpolation (see Section 5.3).

5.2.4 Approximation of functions by interpolating polynomials

Next, we turn to the question of how good polynomial interpolation can be in approximating a continuous function f of which we only know discrete values $f(t_0), \dots, f(t_n)$. For a node set $\mathcal{T} = \{t_j\}_{j=0}^n$, the Lagrange interpolation operator may be defined as

$$\mathcal{I}_{\mathcal{T}}(y_0, \dots, y_n) := \sum_{i=0}^n y_i L_i(t),$$

where L_i denotes the i -th Lagrange polynomial. Suppose now that there is a true underlying continuous function f that we try to approximate by Lagrange interpolation through the data points $f(t_0), \dots, f(t_n)$. Then, we can think of the Lagrangian interpolation operator inducing an approximation scheme on $C^0(I)$, where $I \subset \mathbb{R}$ is an interval containing the node set \mathcal{T} .

Definition 5.2.2 (Lagrangian (interpolation polynomial) approximation scheme). Given an interval $I \subset \mathbb{R}$, $n \in \mathbb{N}$, a node set $\mathcal{T} = \{t_0, \dots, t_n\} \subset I$, the *Lagrangian (interpolation polynomial) approximation scheme* $\mathcal{L}_{\mathcal{T}} : C^0(I) \rightarrow \mathcal{P}_n$ is defined by

$$\mathcal{L}_{\mathcal{T}}(f) := \mathcal{I}_{\mathcal{T}}(\mathbf{y}) \in \mathcal{P}_n \quad \text{with} \quad \mathbf{y} := (f(t_0), \dots, f(t_n))^{\top} \in \mathbb{K}^{n+1}.$$

We are interested in the behaviour of the Lagrangian approximation scheme as the number of nodes is increased and different families $\mathcal{T}_n = \{t_0^{(n)}, \dots, t_n^{(n)}\}$ are considered. Mostly, we are interested in the *interpolation error* $\|f - \mathcal{L}_{\mathcal{T}}(f)\|$ (for relevant norms on $C^0(I)$). The best approximation in \mathcal{P}_n will be a function of polynomial degree n . Similarly, we may study a *family* of Lagrange interpolation schemes $\{\mathcal{L}_{\mathcal{T}_n}\}_{n \in \mathbb{N}_0}$ on $I \subset \mathbb{R}$ induced by a *family of node sets* $\{\mathcal{T}_n\}_{n \in \mathbb{N}_0}$, $\mathcal{T}_n \subset I$.

An example for such a family of node sets on $I := [a, b]$ are the *equidistant* or *equispaced* nodes

$$\mathcal{T}_n := \left\{ t_j^{(n)} := a + (b - a) \frac{j}{n} : j = 0, \dots, n \right\} \subset I.$$

For a family of approximation schemes $\{\mathcal{L}_{\mathcal{T}_n}\}_{n \in \mathbb{N}}$, can we find a bound for the interpolation error?

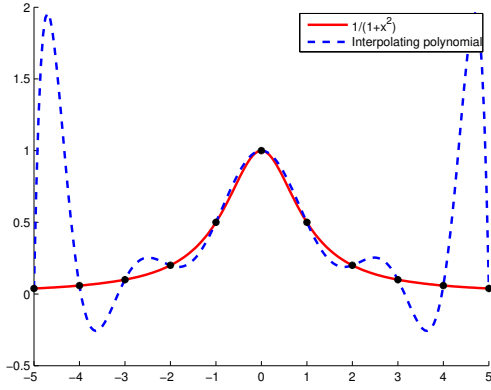
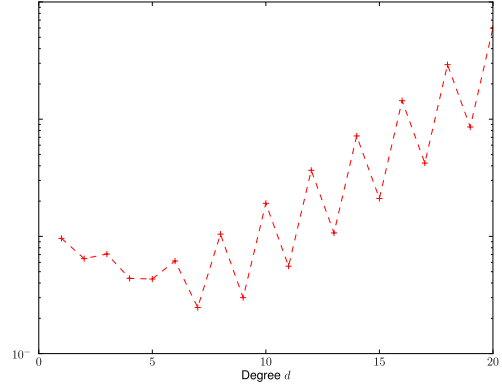
Let us consider an example:

Example 5.2.2: Polynomial interpolation of $\frac{1}{1+t^2}$ on equidistant nodes

We examine the polynomial interpolant of $f(t) = \frac{1}{1+t^2}$ for equidistant nodes:

$$\mathcal{T}_n := \left\{ t_j := -5 + \frac{10}{n} j \right\}_{j=0}^n, \quad j = 0, \dots, n \quad \implies \quad y_j = \frac{1}{1 + t_j^2}.$$

We rely on an approximate computation of the supremum norm of the interpolation error by means of sampling. Note that we approximate $\|f - \mathcal{L}_{\mathcal{T}_n} f\|_{\infty}$ by *sampling* at 1000 equidistant points.


 Interpolating polynomial, $n = 10$

 Approximate $\|f - L_{T_n}f\|_{\infty}$ on $[-5, 5]$

Recall Runge's phenomenon: Strong oscillations of $L_{\mathcal{T}}(f)$ near the endpoints of the interval. Additionally:

$$\|f - L_{\mathcal{T}_n}f\|_{L^{\infty}([-5,5])} \xrightarrow{n \rightarrow \infty} \infty .$$

Theorem 5.2.2 (Divergent polynomial interpolants). *Given a sequence of meshes of increasing size $\{\mathcal{T}_n\}_{n=1}^{\infty}$, $\mathcal{T}_j = \{t_0^{(j)}, \dots, t_n^{(j)}\} \subset [a, b]$, $a \leq t_0^{(j)} < t_2^{(j)} < \dots < t_n^{(j)} \leq b$, there exists a continuous function f such that the sequence of interpolating polynomials $(L_{\mathcal{T}_n}f)_{n=1}^{\infty}$ does not converge to f uniformly as $n \rightarrow \infty$.*

Now, we aim to establish bounds for the supremum norm of the approximation error of Lagrangian interpolation.

Theorem 5.2.3 (Representation of interpolation error). *We consider $f \in C^{n+1}(I)$ and the Lagrangian interpolation approximation scheme (see Definition 5.2.2) for a node set $\mathcal{T} := \{t_0, \dots, t_n\} \subset I$. Then, for every $t \in I$ there exists a $\tau_t \in (\min\{t, t_0, \dots, t_n\}, \max\{t, t_0, \dots, t_n\})$ such that*

$$f(t) - L_{\mathcal{T}}(f)(t) = \frac{f^{(n+1)}(\tau_t)}{(n+1)!} \cdot \prod_{j=0}^n (t - t_j) . \quad (5.20)$$

Proof. Define $w_{\mathcal{T}}(t) := \prod_{j=0}^n (t - t_j) \in \mathcal{P}_{n+1}$ and fix $t \in I \setminus \mathcal{T}$, i.e., t is not a node. This implies that $w_{\mathcal{T}}(t) \neq 0$. Therefore, we can choose $c \in \mathbb{R}$ such that

$$f(t) - L_{\mathcal{T}}(f)(t) = cw_{\mathcal{T}}(t) .$$

Consider the auxiliary function

$$\varphi(x) := f(x) - \underbrace{L_{\mathcal{T}}(f)(x)}_{\in \mathcal{P}_n} - \underbrace{cw_{\mathcal{T}}(x)}_{\in \mathcal{P}_{n+1}} \in C^{n+1}(I) .$$

5 Data Interpolation in 1D

The function φ has at least $n + 2$ distinct zeros $\{t_0, \dots, t_n, t\}$ since

$$\begin{aligned} \varphi(t_j) &= 0, \quad \text{for } j = 0, \dots, n \quad \text{from the interpolating conditions and} \\ \varphi(t) &= 0, \quad \text{by definition.} \end{aligned}$$

By iterated application of the *mean value theorem* [2, Thm 5.2.1]

$$f \in C^1([a, b]), \quad f(a) = f(b) = 0 \quad \Rightarrow \quad \exists \xi \in]a, b[: \quad f'(\xi) = 0,$$

to higher and higher derivatives, we conclude that

$$\begin{aligned} \varphi' &\text{ has at least } n + 1 \text{ distinct zeros} \\ \varphi'' &\text{ has at least } n \text{ distinct zeros} \\ &\vdots \\ \varphi^{(n+1)}(x) &\text{ has at least 1 distinct zero.} \end{aligned}$$

We denote this zero of $\varphi^{(n+1)}(x)$ by τ_t . Note that

$$\varphi^{(n+1)}(x) = f^{(n+1)}(x) - 0 - c(n+1)!$$

Thus,

$$\begin{aligned} \varphi^{(n+1)}(\tau_t) &= 0 = f^{(n+1)}(\tau_t) - c(n+1)! \\ \Rightarrow c &= \frac{f^{(n+1)}(\tau_t)}{(n+1)!}. \end{aligned}$$

□

This yields the following interpolation error estimate for degree- n Lagrange interpolation on the node set $\{t_0, \dots, t_n\}$:

$$\text{Theorem 5.2.3} \quad \Rightarrow \quad \|f - \mathcal{L}_{\mathcal{T}} f\|_{L^\infty(I)} \leq \frac{\|f^{(n+1)}\|_{L^\infty(I)}}{(n+1)!} \max_{t \in I} |(t - t_0) \cdots (t - t_n)|. \quad (5.21)$$

Example 5.2.3: Polynomial interpolation of $\sin(t)$ on equidistant nodes

We consider polynomial interpolation of $f(t) = \sin(t)$ on equidistant nodes. From the error estimate (5.21) and

$$\|f^{(k)}\|_{L^\infty(I)} \leq 1, \quad \forall k \in \mathbb{N}_0, \quad (5.22)$$

it follows that

$$\begin{aligned}
 \|f - \mathcal{L}_{\mathcal{T}_n} f\|_{L^\infty(I)} &\leq \frac{1}{(1+n)!} \underbrace{\max_{t \in I} |(t-0)(t-\frac{\pi}{n})(t-\frac{2\pi}{n}) \cdots (t-\pi)|}_{\text{extrema at } \approx \frac{\pi}{2n}} \quad (5.23) \\
 &\leq \frac{1}{(n+1)!} \left| \frac{\pi}{2n} \left(\frac{\pi}{2n} - \frac{\pi}{n} \right) \cdots \left(\frac{\pi}{2n} - \pi \right) \right| \\
 &\leq \frac{1}{(n+1)!} \left(\frac{\pi}{n} \right)^{n+1} \underbrace{\left| \frac{1}{2} \left(\frac{1}{2} - 1 \right) \left(\frac{1}{2} - 2 \right) \cdots \left(\frac{1}{2} - n \right) \right|}_{\leq n!} \\
 &\leq \frac{1}{n+1} \left(\frac{\pi}{n} \right)^{n+1}.
 \end{aligned}$$

Thus, we have proven uniform asymptotic (more than) exponential convergence of the interpolation polynomials for equidistant node sets \mathcal{T}_n . Next, we analyze Example 5.2.2 according to Theorem 5.2.3.

Example 5.2.4: Polynomial interpolation of $\frac{1}{1+t^2}$ on equidistant nodes

We examine the polynomial interpolation of $f(t) = \frac{1}{1+t^2}$, on $I = [-5, 5]$ using equidistant nodes. We know

$$\|f^{(n+1)}\|_{L^\infty([-5,5])} \sim 2^{n+1}(n+1)! \quad \text{for } n \rightarrow \infty.$$

The right hand side of (5.21) is roughly

$$\sim 2^{n+1}(n+1)! \frac{1}{(n+1)!} n! \left(\frac{5}{n} \right)^{n+1} 2^n = n! \left(\frac{20}{n} \right)^n \frac{10}{n}.$$

By Stirling's formula this grows exponentially:

$$n! \left(\frac{20}{n} \right)^n \frac{10}{n} \geq \underbrace{n^{n+\frac{1}{2}} \sqrt{2\pi} e^{-n}}_{=10\left(\frac{20}{e}\right)^n \sqrt{\frac{2\pi}{n}}} \left(\frac{20}{n} \right)^n \frac{10}{n}.$$

So estimate (5.21) no longer guarantees convergence (i.e. blow-up is possible).

Note that there is also an L^2 -estimate for $f \in C^{n+1}(I)$ and $\mathcal{T} = \{t_0, \dots, t_n\} \in I$:

$$\|f - \mathcal{L}_{\mathcal{T}} f\|_{L^2(I)} \leq \frac{2^{\frac{n-1}{4}} |I|^{n+1}}{\sqrt{n!(n+1)!}} \|f^{(n+1)}\|_{L^2(I)}.$$

5.2.5 Chebyshev Interpolation

When we build approximation schemes from interpolation schemes we have the extra freedom to choose the sampling points (= interpolation nodes). Now, based on the insight in the structure of the interpolation error gained from Theorem 5.2.3, we seek to choose “optimal” sampling points.

We have the setting:

- $I = [-1, 1]$ (which we can assume without loss of generality),
- the interpolant $f : I \rightarrow \mathbb{R}$, $f \in C^0(I)$,
- and the set of interpolation nodes $\mathcal{T} := \{-1 \leq t_0 < t_1 < \dots < t_{n-1} < t_n \leq 1\}$, $n \in \mathbb{N}$.

We recall Theorem 5.2.3:

$$\|f - \mathcal{L}_{\mathcal{T}} f\|_{L^\infty(I)} \leq \frac{1}{(n+1)!} \|f^{(n+1)}\|_{L^\infty(I)} \|w\|_{L^\infty(I)}$$

with *nodal polynomial* $w(t) := \prod_{j=0}^n (t - t_j)$.

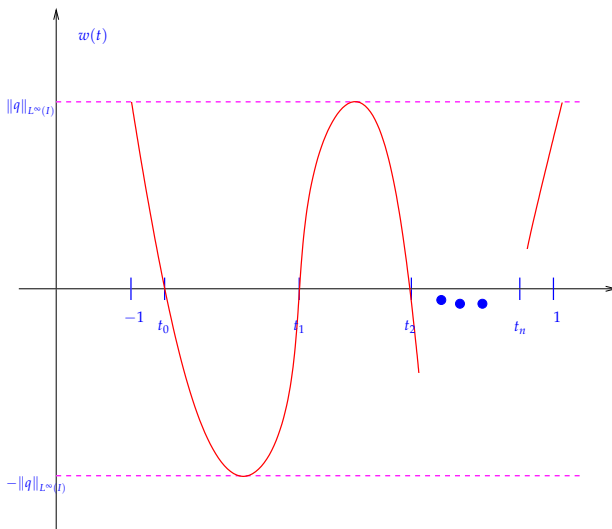
Optimal choice of interpolation nodes independent of interpolant

Idea: Choose nodes t_0, \dots, t_n such that $\|w\|_{L^\infty(I)}$ is minimal.

This is equivalent to finding a polynomial $q \in \mathcal{P}_{n+1}$

- with leading coefficient = 1,
- and which minimizes the norm $\|q\|_{L^\infty(I)}$.

\Rightarrow Then, choose nodes t_0, \dots, t_n as zeros of q (caution: t_j must belong to I).



Requirements on q (by heuristic reasoning):
We stress that we aim for an “optimal” *a priori* choice of interpolation nodes, a choice that is made **before** any information about the interpolant becomes available.

Of course, an *a posteriori* choice based on information gleaned from evaluations of the interpolant f may yield much better interpolants (in the sense of smaller norm of the interpolation error). Many modern algorithms employ this *a posteriori adaptive approximation policy*, but this chapter will not cover them.

Based on the requirement that q minimizes $\|\cdot\|_{L^\infty(I)}$, one can already deduce two properties of q :

- q has $n + 1$ zeros given by t_0, \dots, t_n since $q := \prod_{j=0}^n (t - t_j)$.
- All these zeros lie in $[-1, 1]$.

Proof. Suppose $t_0 < -1$. Then, we can define

$$p(t) := (t + 1)(t - t_1)(t - t_2) \cdots (t - t_n).$$

This implies

$$\begin{aligned} |p(t)| &= |q(t)| \cdot \underbrace{\frac{|t + 1|}{|t - t_0|}}_{< 1} \\ \implies |p(t)| &< |q(t)| \quad \forall t \in I \\ \implies \|p(t)\|_{L^\infty(I)} &< \|q(t)\|_{L^\infty(I)}. \end{aligned}$$

This contradicts q being the optimal choice. Thus, $t_0 \geq -1$.

Similarly, suppose $t_n > 1$. Then, we can define

$$p(t) := (t - t_0)(t - t_1)(t - t_2) \cdots (t - t_{n-1})(t - 1).$$

This implies

$$\begin{aligned} |p(t)| &= |q(t)| \cdot \underbrace{\frac{|t - 1|}{|t - t_n|}}_{< 1} \\ \implies |p(t)| &< |q(t)| \quad \forall t \in I \\ \implies \|p(t)\|_{L^\infty(I)} &< \|q(t)\|_{L^\infty(I)}. \end{aligned}$$

Again, this contradicts q being the optimal choice. Thus, $t_n \leq 1$.

□

Are there polynomials satisfying these requirements? If so, do they allow a simple characterization? The answer is: Yes. These polynomials are (up to a normalization factor) equal to the Chebyshev polynomials:

Definition 5.2.3 (Chebyshev polynomials). The n^{th} Chebyshev polynomial is $T_n(t) := \cos(n \arccos t)$, where $-1 \leq t \leq 1$, $n \in \mathbb{N}$.

The next result confirms that the T_n are indeed polynomials.

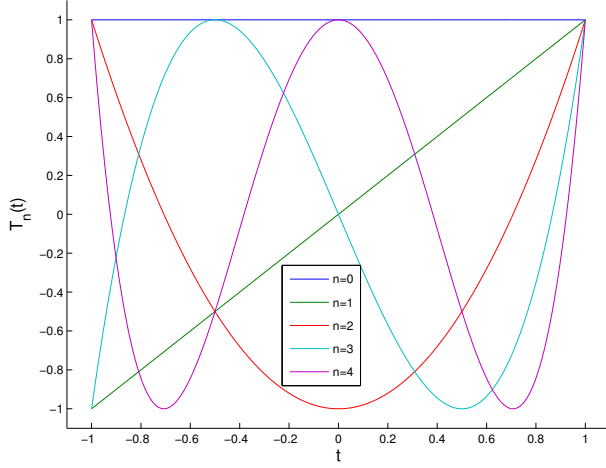
Theorem 5.2.4 (3-term recursion for Chebyshev polynomials). The functions T_n defined in Definition 5.2.3 satisfy the 3-term recursion

$$T_{n+1}(t) = 2t T_n(t) - T_{n-1}(t), \quad T_0 \equiv 1, \quad T_1(t) = t, \quad n \in \mathbb{N}. \quad (5.24)$$

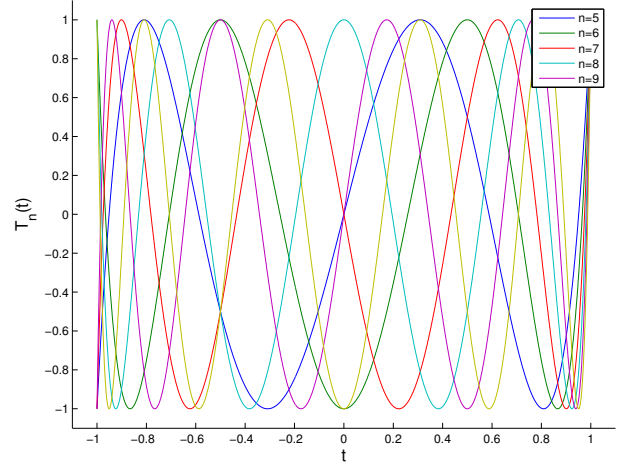
5 Data Interpolation in 1D

The theorem implies that

- $T_n \in \mathcal{P}_n$,
- the leading coefficient of the n -th Chebyshev polynomial is equal to 2^{n-1} ,
- the T_n are linearly independent,
- $\{T_j\}_{j=0}^n$ is a basis of $\mathcal{P}_n = \text{Span}\{T_0, \dots, T_n\}$, $n \in \mathbb{N}_0$.



Chebyshev polynomials T_0, \dots, T_4



Chebyshev polynomials T_5, \dots, T_9

Definition 5.2.3 implies that the zeros of the T_n are

$$t_j = \cos\left(\frac{2j+1}{2n}\pi\right), \quad j = 0, \dots, n-1, \quad (5.25)$$

which are called the *Chebyshev nodes*.

To see this, notice that

$$\begin{aligned} T_n(t) = 0 & \stackrel{\text{zeros of } \cos}{\Leftrightarrow} n \arccos t \in (2\mathbb{Z} + 1)\frac{\pi}{2} \\ & \stackrel{\arccos \in [0, \pi]}{\Leftrightarrow} t \in \left\{ \cos\left(\frac{2j+1}{n}\frac{\pi}{2}\right), j = 0, \dots, n-1 \right\}. \end{aligned}$$

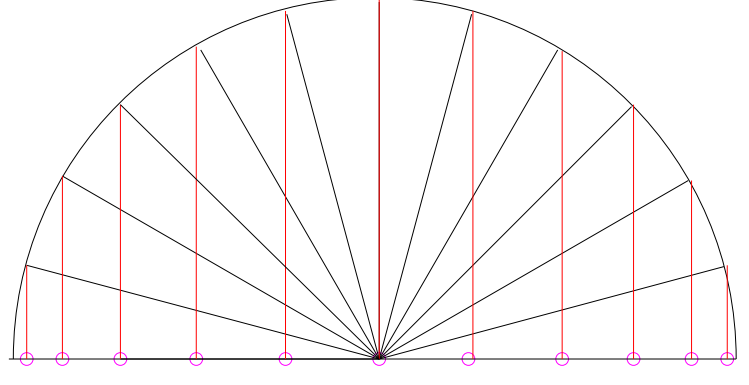
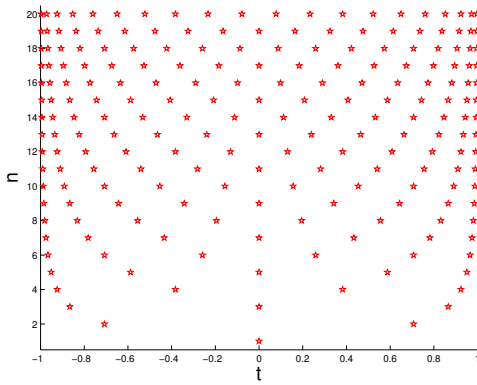
The next theorem will demonstrate that, after scaling, the T_n indeed constitute polynomials on $[-1, 1]$ with fixed leading coefficient and minimal supremum norm.

Theorem 5.2.5 (Minimax property of the Chebyshev polynomials). *The polynomials T_n from Definition 5.2.3 minimize the supremum norm in the following sense:*

$$\|T_n\|_{L^\infty([-1,1])} = \inf\{\|p\|_{L^\infty([-1,1])} : p \in \mathcal{P}_n, p(t) = 2^{n-1}t^n + \dots\}, \quad \forall n \in \mathbb{N}.$$

So the nodal polynomial w minimizing the supremum norm may be found by setting $w(t) = 2^{-n}T_{n+1}(t)$. The interpolation nodes corresponding to this nodal polynomial are given by the zeros of T_{n+1} .

Thus, we have identified the points t_0, \dots, t_n from (5.25) as optimal interpolation nodes for a Lagrangian approximation scheme. The t_k are known as *Chebyshev nodes*. Their distribution in $[-1, 1]$ and a geometric construction are plotted below.



Observe that the Chebyshev nodes cluster at the boundary of I for large n . The geometric construction depicts that the Chebyshev nodes result from an equidistributed set of points on the circle. When we use Chebyshev nodes for polynomial interpolation, we call the resulting Lagrangian approximation scheme *Chebyshev interpolation*. On the interval $[-1, 1]$, it is characterized by:

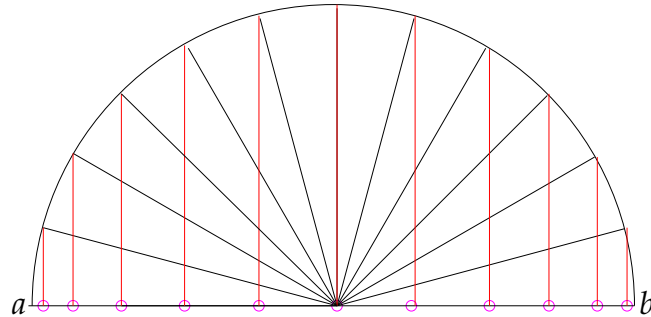
- “Optimal” interpolation nodes: $\mathcal{T} = \left\{ \cos \left(\frac{2k+1}{2(n+1)} \pi \right), k = 0, \dots, n \right\}$.
- $w(t) = (t - t_0) \cdots (t - t_n) = 2^{-n} T_{n+1}(t)$, $\|w\|_{L^\infty(I)} = 2^{-n}$, with leading coefficient 1.

By Theorem 5.2.3, we immediately get an interpolation error estimate for Chebyshev interpolation of $f \in C^{n+1}([-1, 1])$:

$$\|f - L_{\mathcal{T}_n} f\|_{L^\infty([-1, 1])} \leq \frac{2^{-n}}{(n+1)!} \|f^{(n+1)}\|_{L^\infty([-1, 1])}. \quad (5.26)$$

Note. We can define the Chebyshev interpolation on an arbitrary interval $[a, b]$. The same Lagrangian approximation scheme is obtained by transforming the Chebyshev nodes (5.25) from $[-1, 1]$ to $[a, b]$ using the unique affine transformation:

$$\Phi : [-1, 1] \rightarrow [a, b], \quad \hat{t} \mapsto t := a + \frac{1}{2}(\hat{t} + 1)(b - a) \in [a, b].$$



The Chebyshev nodes in the interval $I = [a, b]$ are thus given by

$$\hat{t}_k := a + \frac{1}{2}(b - a) \left(\cos \left(\frac{2k+1}{2(n+1)} \pi \right) + 1 \right), \quad k = 0, \dots, n. \quad (5.27)$$

Then, if $p \in \mathcal{P}_n$ is the interpolating polynomial with $p(t_j) = f(t_j)$, where $f \in C^0[a, b]$, and $\hat{p} := p \circ \Phi \in \mathcal{P}_n$, the equivalent interpolation problem on $[-1, 1]$ can be formulated as

$$\hat{p}(\hat{t}_j) = \hat{f}(\hat{t}_j),$$

where $\hat{f}(\hat{t}) := f(\Phi(\hat{t}))$.

Now we can use $\frac{d^n \hat{f}}{d\hat{t}^n}(\hat{t}) = (\frac{1}{2}|I|)^n \frac{d^n f}{dt^n}(t)$ along with (5.26) to obtain

$$\|f - \mathcal{L}_{\mathcal{T}}(f)\|_{L^\infty(I)} = \|\hat{f} - \mathcal{L}_{\hat{\mathcal{T}}}(\hat{f})\|_{L^\infty([-1,1])}, \quad (5.28)$$

$$\begin{aligned} &\leq \frac{2^{-n}}{(n+1)!} \left\| \frac{d^{n+1} \hat{f}}{d\hat{t}^{n+1}} \right\|_{L^\infty([-1,1])}, \\ &\leq \frac{2^{-2n-1}}{(n+1)!} |I|^{n+1} \|f^{(n+1)}\|_{L^\infty(I)}. \end{aligned} \quad (5.29)$$

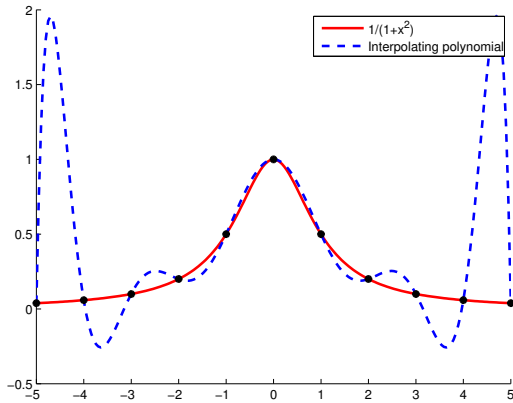
In spite of (5.29) being a better estimate than its counterpart for equidistant points, divergence is still possible when $\|f^{(n+1)}\|_{L^\infty(I)}$ grows very fast. For demonstration, we go back to *Runge's example*, cf. Example 5.2.2:

$$\begin{aligned} f(t) &= \frac{1}{1+t^2}, \quad t \in [-5, 5], \\ \|f^{(n+1)}\|_{L^\infty([-5,5])} &\sim 2^{n+1}(n+1)! \quad . \end{aligned}$$

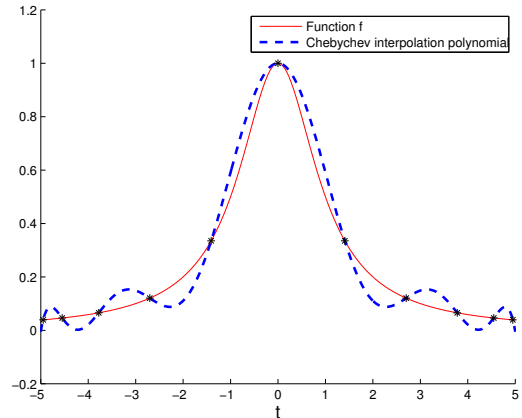
In this case, the right-hand side of (5.29) still blows up with exponential growth:

$$\text{RHS} \sim \frac{2^{-2n-1}}{(n+1)!} \cdot 10^{n+1} \cdot 2^{n+1} \cdot (n+1)! = 10 \cdot 5^n .$$

Note. The RHS is a bound that is not necessarily sharp. Hence, the blow-up of the right-hand side of (5.29) does not necessarily imply divergence of the approximation. Comparing equidistant nodes vs. Chebyshev nodes for Runge's example we see that there is no Runge phenomenon for the choice of Chebyshev nodes.



Equidistant nodes



Chebyshev nodes $n = 10$

We observe that the Chebyshev nodes cluster at the endpoints of the interval, which successfully suppresses the huge oscillations haunting equidistant interpolation.

Now, we empirically investigate the behaviour of norms of the interpolation error for Chebyshev interpolation and functions with different (smoothness) properties as we increase the number of interpolation nodes.

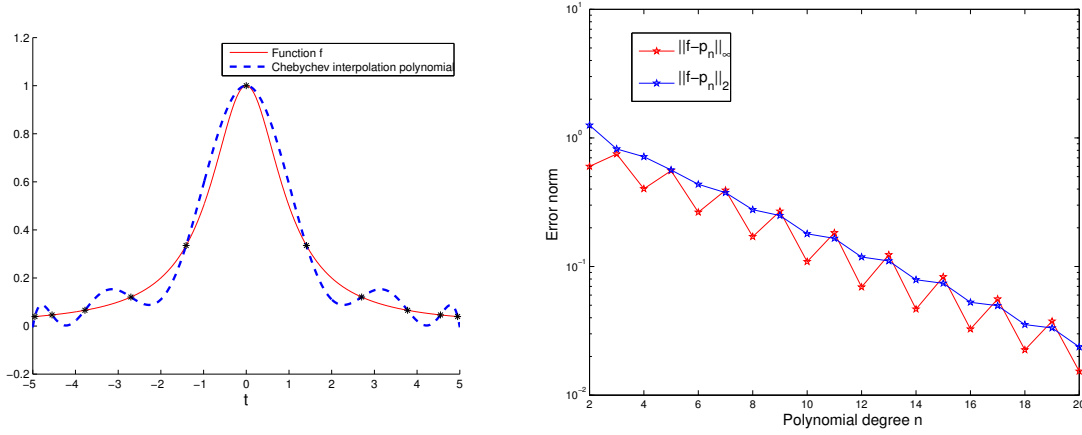
For $I = [a, b]$ we set $x_l := a + \frac{b-a}{N}l$, $l = 0, \dots, N$, $N = 1000$, and we approximate the norms of the interpolation error as follows ($p \hat{=}$ interpolating polynomial):

$$\|f - p\|_{\infty} \approx \max_{0 \leq l \leq N} |f(x_l) - p(x_l)|, \quad (5.30)$$

$$\|f - p\|_2^2 \approx \frac{b-a}{2N} \sum_{0 \leq l < N} (|f(x_l) - p(x_l)|^2 + |f(x_{l+1}) - p(x_{l+1})|^2). \quad (5.31)$$

① $f(t) = \frac{1}{1+t^2}$, $I = [-5, 5]$ (see Example 5.2.2):

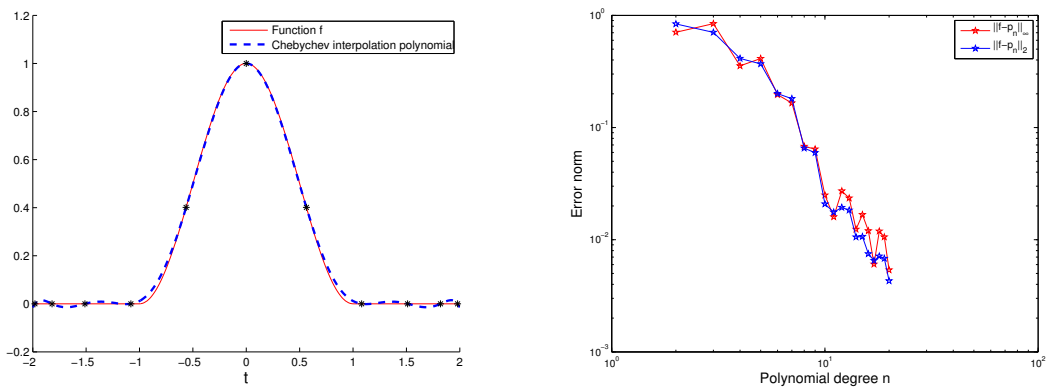
Interpolation with $n = 10$ Chebyshev nodes. In this case $f \in C^{\infty}$.



Notice the exponential convergence of the Chebyshev interpolation:

$$p_n \rightarrow f, \quad \|f - \mathcal{L}_{\mathcal{T}_n}(f)\|_{L^{\infty}([-5,5])} \approx 0.8^n.$$

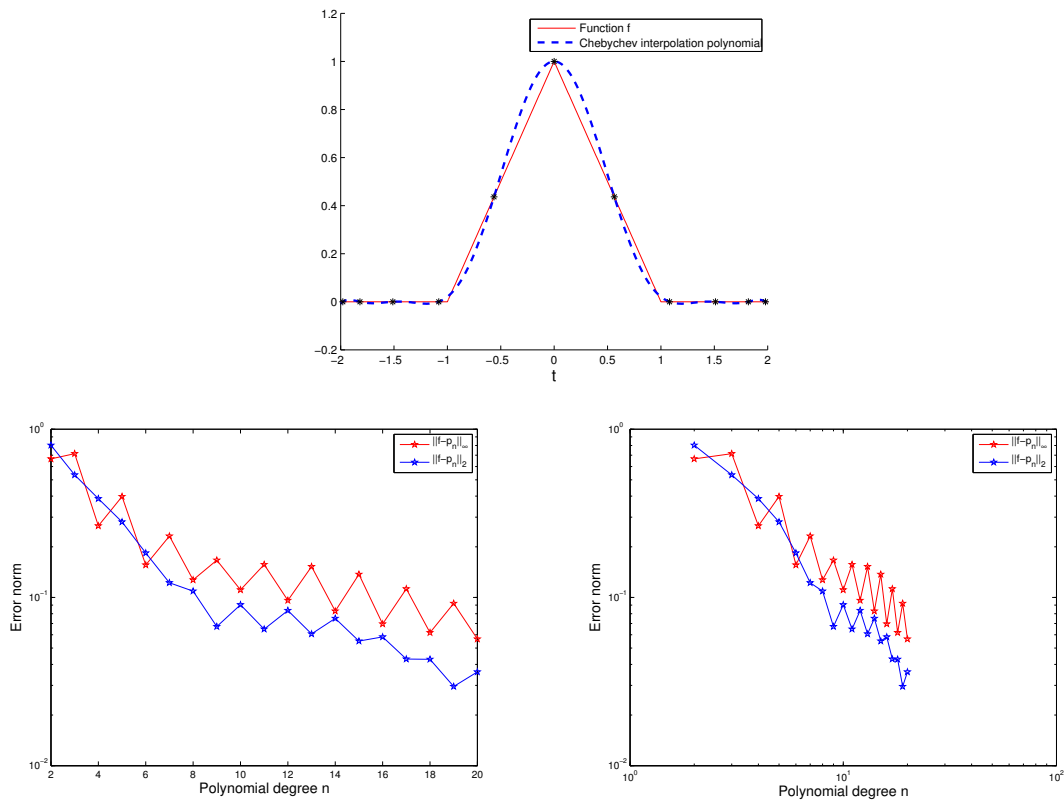
② $f(t) = \begin{cases} \frac{1}{2}(1 + \cos \pi t) & |t| < 1 \\ 0 & 1 \leq |t| \leq 2 \end{cases}$ $I = [-2, 2]$ and $n = 10$. Here, $f \in C^1(I)$.



We obtain algebraic convergence for this example.

5 Data Interpolation in 1D

③ $f(t) = \max\{1 - |t|, 0\}$, $I = [-2, 2]$ and $n = 10$. Now $f \in C^0(I)$ but $f \notin C^1(I)$.



From the doubly logarithmic plot we conclude:

- No exponential convergence.
- But: Algebraic convergence!

Again, one can observe algebraic convergence but one that is a bit slower than in the previous (smooth) case.

Summary of observations, *cf.* Section 5.2.4:

- Essential role of smoothness of f : Slow convergence of approximation error of the Chebyshev interpolant if f enjoys little smoothness, *cf.* also (5.21).
- For smooth $f \in C^\infty$, the approximation error of the Chebyshev interpolant decays to zero exponentially in the polynomial degree n .

Implementation of Chebyshev interpolation

We close our discussion on Chebyshev interpolation by finding an efficient scheme for its implementation. We will see that this interpolation problem can be formulated via the DFT for which we know a fast implementation (FFT).

We start with the following idea: Write the interpolants $p \in \mathcal{P}_n$ as a linear combination of Chebyshev polynomials, a so-called *Chebyshev expansion*:

$$p(x) = \sum_{j=0}^n \alpha_j T_j(x), \quad \alpha_j \in \mathbb{R}. \quad (5.32)$$

$$\deg T_j = j \implies \{T_0, \dots, T_n\} \text{ basis of } \mathcal{P}_n.$$

The implementation of the Chebyshev interpolation is divided into the following two steps. First, we assume that the α_j are already known and we study the evaluation of $p(x)$. Next, we consider the question of finding the coefficients α_j .

① Efficient evaluation of p given coefficients α_j :

Idea: Use the 3-term recursion (5.24) of Chebyshev polynomials:

$$T_0 \equiv 1, \quad T_1(x) = x, \quad T_j(x) = 2xT_{j-1}(x) - T_{j-2}(x), \quad j \geq 2. \quad (5.33)$$

By means of (5.33) rewrite (5.32) as

$$\begin{aligned} p(x) &= \sum_{j=0}^{n-1} \alpha_j T_j(x) + \alpha_n T_n(x) \\ &\stackrel{(5.33)}{=} \left(\sum_{j=0}^{n-1} \alpha_j T_j(x) \right) + \alpha_n (2xT_{n-1}(x) - T_{n-2}(x)) \\ &= \left(\sum_{j=0}^{n-3} \alpha_j T_j(x) \right) + \alpha_{n-2} T_{n-2}(x) + \alpha_{n-1} T_{n-1}(x) + \alpha_n (2xT_{n-1}(x) - T_{n-2}(x)) \\ &= \left(\sum_{j=0}^{n-3} \alpha_j T_j(x) \right) + (\alpha_{n-2} - \alpha_n) T_{n-2}(x) + (\alpha_{n-1} + 2x\alpha_n) T_{n-1}(x). \end{aligned}$$

We recover the point value $p(x)$ as the point value of another polynomial of degree $n-1$ with known Chebyshev expansion:

$$p(x) = \sum_{j=0}^{n-1} \tilde{\alpha}_j T_j(x) \quad \text{with} \quad \tilde{\alpha}_j = \begin{cases} \alpha_j + 2x\alpha_{j+1} & , \text{ if } j = n-1, \\ \alpha_j - \alpha_{j+2} & , \text{ if } j = n-2, \\ \alpha_j & \text{ else.} \end{cases} \quad (5.34)$$

These calculations inspire the following recursive algorithm, which is also known as the *Clenshaw algorithm*.

Define

$$\beta_{n+2} = \beta_{n+1} = 0.$$

For $k = n, \dots, 1$:

$$\beta_k = \alpha_k + 2x\beta_{k+1} - \beta_{k+2} \quad (5.35)$$

and

$$\beta_0 = 2\alpha_0 + 2x\beta_1 - \beta_2.$$

The evaluation of p at x can now be written as:

$$p(x) = \frac{1}{2}(\beta_0 - \beta_2). \quad (5.36)$$

Code Snippet 5.2: Clenshaw algorithm for evaluation of Chebyshev expansion (5.32)

```

13 // Clenshaw algorithm for evaluating  $p = \sum_{j=1}^{n+1} a_j T_{j-1}$ 
14 // at points passed in vector  $x$ 
15 // IN :  $a = [\alpha_j]$ , coefficients for  $p = \sum_{j=1}^{n+1} \alpha_j T_{j-1}$ 
16 //  $x$  = evaluation point
17 // OUT: value  $p(x)$ 
18 VectorXd clenshaw(const VectorXd& a, const double x) {
19     const int n = a.size() - 1; // degree of polynomial
20     double beta_kp2 = 0.0; // storage for  $\beta_{(k+2)}$  value initialised by  $\beta_{(n+2)}$ 
21     double beta_kp1 = 0.0; // storage for  $\beta_{(k+1)}$  value initialised by  $\beta_{(n+1)}$ 
22     double beta_k;
23     for (int k = n; k > 0; --k) {
24         beta_k = a(k) + 2*x*beta_kp1 - beta_kp2; // see (5.35)
25         beta_kp2 = beta_kp1;
26         beta_kp1 = beta_k;
27     }
28
29     //  $p(x) = \alpha_0 + \beta_1 * x - \beta_2$ 
30     return a(0) + beta_kp1*x - beta_kp2;
31 }
32 /*

```

This algorithm has complexity $\mathcal{O}(n)$.

② Computation of coefficients α_j in (5.32):

Chebyshev interpolation is a linear interpolation scheme. Thus, the expansion α_j in (5.32) can be computed by solving a linear system of equations of the form (5.6). However, for Chebyshev interpolation, this linear system can be cast into a very special form, which paves the way for its fast direct solution:

We will use the interpolation conditions for this task:

$$p(t_k) = f(t_k), \quad k = 0, \dots, n, \quad (5.37)$$

$$t_k = \cos \left(\frac{2k+1}{2(n+1)} \pi \right).$$

Define

$$s_k := \frac{2k+1}{4(n+1)}.$$

Then,

$$t_k = \cos(2\pi s_k) .$$

Trick: Transform p into a 1-periodic function, which turns out to be a *Fourier sum* (= finite Fourier series):

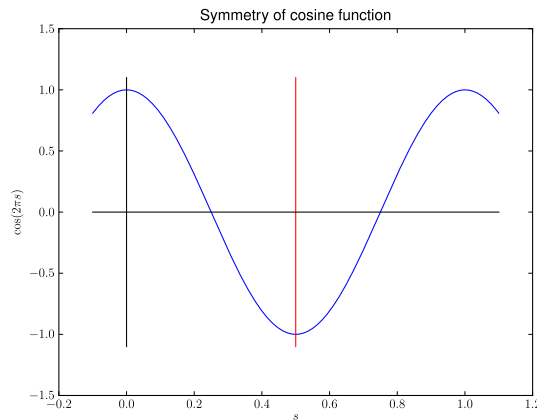
$$\begin{aligned} q(s) &:= p(\cos 2\pi s) = \sum_{j=0}^n \alpha_j T_j(\cos 2\pi s) \stackrel{\text{Def. 5.2.3}}{=} \sum_{j=0}^n \alpha_j \cos(2\pi j s) \\ &= \sum_{j=0}^n \frac{1}{2} \alpha_j (\exp(2\pi i j s) + \exp(-2\pi i j s)) \quad [\text{since } \cos z = \frac{1}{2}(e^z + e^{-z})] \\ &= \sum_{j=-n}^{n+1} \beta_j \exp(-2\pi i j s) , \quad \text{with } \beta_j := \begin{cases} 0, & \text{for } j = n+1 , \\ \frac{1}{2} \alpha_j, & \text{for } j = 1, \dots, n , \\ \alpha_0, & \text{for } j = 0 , \\ \frac{1}{2} \alpha_{-j}, & \text{for } j = -n, \dots, -1 . \end{cases} \end{aligned} \quad (5.38)$$

Having written $q(s)$ in such a nice Fourier sum, we next plug in the interpolation conditions (5.37) with the goal of constructing a linear system of equations with the Fourier matrix:

$$\begin{aligned} T_j(t) &= \cos(j \cdot \arccos(t)) \\ T_j(\cos(2\pi s)) &= \cos(j \cdot 2\pi s) \quad s \in \left[0, \frac{1}{2}\right] \\ \stackrel{(5.37)}{\implies} q\left(\frac{2k+1}{4(n+1)}\right) &= y_k := f(t_k) , \quad k = 0, \dots, n . \end{aligned} \quad (5.39)$$

This is an interpolation problem for *equidistant points on the unit circle*. Note that so far we can, given the $n+1$ interpolation conditions, build a system of $n+1$ linear equations for the vector of unknowns β_j , with length $2n+2$. Our goal is to extend this to a $(2n+2) \times (2n+2)$ system. This is possible due to the following symmetry:

$$q(s) = q(1-s).$$



Thus, we can extend the interpolation conditions (5.39) by

$$q\left(1 - \frac{2k+1}{4(n+1)}\right) = y_k, \quad k = 0, \dots, n.$$

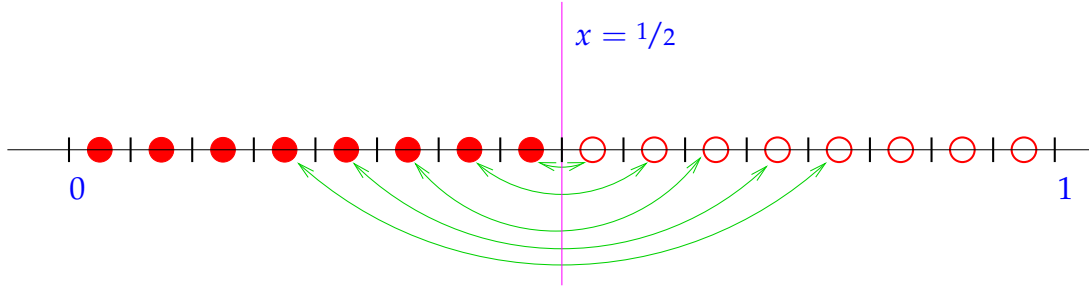
Note also that this ensures that the coefficients β_j actually satisfy the constraints implied by their relationship with α_j .

Altogether, we obtain the following $2n+2$ equations:

$$q\left(\frac{k}{2(n+1)} + \frac{1}{4(n+1)}\right) = z_k := \begin{cases} y_k, & \text{for } k = 0, \dots, n, \\ y_{2n+1-k}, & \text{for } k = n+1, \dots, 2n+1, \end{cases} \quad (5.40)$$

because for $k = n+1, \dots, 2n+1$: $q(s_k) = q(1 - s_k) = q(s_{2n+1-k}) = y_{2n+1-k}$.

In a sense, we can mirror the interpolation conditions at $x = \frac{1}{2}$:



We are now in a position to form the $(2n+2) \times (2n+2)$ linear system of equations:

$$q\left(\frac{k}{2(n+1)} + \frac{1}{4(n+1)}\right) = z_k, \quad k = 0, \dots, 2n+1.$$

Using these interpolation conditions in

$$q(s) = \sum_{j=-n}^{n+1} \beta_j \exp(-2\pi i j s)$$

yields

$$\begin{aligned}
\sum_{j=-n}^{n+1} \beta_j \exp \left[-2\pi i j \left(\frac{k}{2(n+1)} + \frac{1}{4(n+1)} \right) \right] &= z_k, \\
\sum_{j=-n}^{n+1} \beta_j \exp \left(-\frac{\pi i j k}{n+1} \right) \exp \left(-\frac{\pi i j}{2(n+1)} \right) &= z_k, \\
\sum_{j=0}^{2n+1} \beta_{j-n} \exp \left(-\frac{\pi i (j-n) k}{n+1} \right) \exp \left(-\frac{\pi i (j-n)}{2(n+1)} \right) &= z_k, \\
\left[\sum_{j=0}^{2n+1} \beta_{j-n} \underbrace{\exp \left(-\frac{\pi i j k}{n+1} \right)}_{\exp(-2\pi i \frac{jk}{2(n+1)})} \exp \left(-\frac{\pi i (j-n)}{2(n+1)} \right) \right] \exp \left(\frac{\pi i n k}{n+1} \right) &= z_k, \\
\sum_{j=0}^{2n+1} \beta_{j-n} \exp \left(-\frac{\pi i (j-n)}{2(n+1)} \right) \omega_{2(n+1)}^{jk} &= \exp \left(-\frac{\pi i n k}{n+1} \right) z_k.
\end{aligned}$$

Thus, defining the vectors

$$\mathbf{c} := \left[\beta_j \exp \left(-\frac{2\pi i j}{4(n+1)} \right) \right]_{j=0}^{2n+1}, \quad \mathbf{b} := \left[\exp \left(-\pi i \frac{nk}{n+1} \right) z_k \right]_{k=0}^{2n+1}, \quad (5.41)$$

we finally obtain

$$\boxed{\mathbf{F}_{2(n+1)} \mathbf{c} = \mathbf{b}.} \quad (5.42)$$

\mathbf{F} is a $(2n+2) \times (2n+2)$ Fourier matrix, see Chapter 4. With the inverse DFT we can recover \mathbf{c} from (5.42), which allows us to recover all β_j 's, which then leads to recovering all α_j 's. Furthermore, this can all be done in $\mathcal{O}(n \log(n))$ complexity.

Code Snippet 5.3: Efficient computation of Chebyshev expansion coefficient of Chebyshev interpolant \rightarrow GITLAB

```

18 // efficiently compute coefficients  $\alpha_j$  in the Chebyshev expansion
19 //  $p = \sum_{j=0}^n \alpha_j T_j$  of  $p \in \mathcal{P}_n$  based on values  $y_k$ ,
20 //  $k = 0, \dots, n$ , in Chebyshev nodes  $t_k$ ,  $k = 0, \dots, n$ 
21 // IN: values  $y_k$  passed in y
22 // OUT: coefficients  $\alpha_j$ 
23 VectorXd chebexp(const VectorXd& y) {

```

```

24  const int n = y.size() - 1;           // degree of polynomial
25  const std::complex<double> M_I(0, 1); // imaginary unit
26  // create vector z, see (5.40)
27  VectorXcd b(2*(n + 1));
28  const std::complex<double> om = -M_I*(M_PI*n)/((double)(n+1));
29  for (int j = 0; j <= n; ++j) {
30      b(j) = std::exp(om*double(j))*y(j); // this cast to double is necessary!!
31      b(2*n+1-j) = std::exp(om*double(2*n+1-j))*y(j);
32  }
33
34  // Solve linear system (5.42) with effort O(n log n)
35  Eigen::FFT<double> fft; // EIGEN's helper class for DFT
36  VectorXcd c = fft.inv(b); // -> c = ifft(z), inverse fourier transform
37  // recover  $\beta_j$ , see (5.41)
38  VectorXd beta(c.size());
39  const std::complex<double> sc = M_PI_2/(n + 1)*M_I;
40  for (unsigned j = 0; j < c.size(); ++j)
41      beta(j) = ( std::exp(sc*double(-n+j))*c[j] ).real();
42  // recover  $\alpha_j$ , see (5.38)
43  VectorXd alpha = 2*beta.segment(n,n); alpha(0) = beta(n);
44  return alpha;
45 }
46 /*

```

Computers use approximation by sums of Chebyshev polynomials in the computation of functions like log, exp, sin, cos, etc. The evaluation by means of the Clenshaw algorithm is more efficient and stable than the approximation by Taylor polynomials.

5.3 Piecewise polynomial interpolation

As mentioned previously, in order to prevent Runge's phenomenon from occurring in interpolation, an alternative to Chebyshev interpolation is to form an interpolant that is piecewise polynomial. In what follows, we discuss two ways to do this. The first is standard piecewise (Lagrange) polynomial interpolation. The second one is *spline* interpolation.

5.3.1 Piecewise polynomial Lagrange interpolation

Recall our estimate for the Chebyshev interpolation:

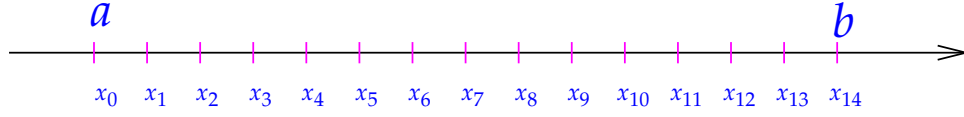
$$\|f - \mathcal{L}_T(f)\|_{L^\infty(I)} \leq \frac{2^{-2n-1}}{(n+1)!} |I|^{n+1} \|f^{(n+1)}\|_{L^\infty(I)}. \quad (5.43)$$

Note that the RHS of (5.43) grows with the length of the interval I . This is the motivation for creating a mesh of the interval I and performing only local Lagrange interpolation on each cell of the mesh.

Given an interval $[a, b] \subset \mathbb{R}$, create a *mesh* \mathcal{M} of I :

$$\mathcal{M} := \{a = x_0 < x_1 < \dots < x_{m-1} < x_m = b\}.$$

We will work with a *local* Lagrange interpolation (see (5.13)) of $f \in C(I)$ on the mesh \mathcal{M} .



Terminology:

- $x_j \triangleq$ nodes of the mesh \mathcal{M} .
- $[x_{j-1}, x_j[\triangleq$ intervals/cells of the mesh.
- $h_{\mathcal{M}} := \max_j |x_j - x_{j-1}| \triangleq$ mesh width.
- If $x_j = a + jh$, we call the mesh *equidistant* or uniform with meshwidth $h > 0$.

General local Lagrange interpolation on a mesh

- 1 Choose a *local degree* $n_j \in \mathbb{N}_0$ for each cell of the mesh, $j = 1, \dots, m$.
- 2 Choose a set of *local* interpolation nodes

$$\mathcal{T}^j := \{t_0^j, \dots, t_{n_j}^j\} \subset I_j := [x_{j-1}, x_j], \quad j = 1, \dots, m,$$

for each mesh cell/grid interval I_j .

- 3 Define a *piecewise polynomial* interpolant $s : [x_0, x_m] \rightarrow \mathbb{K}$:

$$s_j := s|_{I_j} \in \mathcal{P}_{n_j} \quad \text{and} \quad s_j(t_i^j) = f(t_i^j) \quad i = 0, \dots, n_j, \quad j = 1, \dots, m. \quad (5.44)$$

Owing to Theorem 5.2.1, s_j is well defined.

Thus, in each cell the size of the node set is $n_j + 1$, $j = 1, \dots, m$.

Corollary 5.3.1 (Continuous local Lagrange interpolants). *If the local degrees n_j are at least 1 and the local interpolation nodes t_k^j , $j = 1, \dots, m$, $k = 0, \dots, n_j$, for local Lagrange interpolation satisfy*

$$t_{n_j}^j = t_0^{j+1} \quad \forall j = 1, \dots, m-1, \quad (5.45)$$

then the piecewise polynomial Lagrange interpolant according to (5.44) is continuous on $[a, b]$: $s \in C^0([a, b])$.

Example 5.3.1:

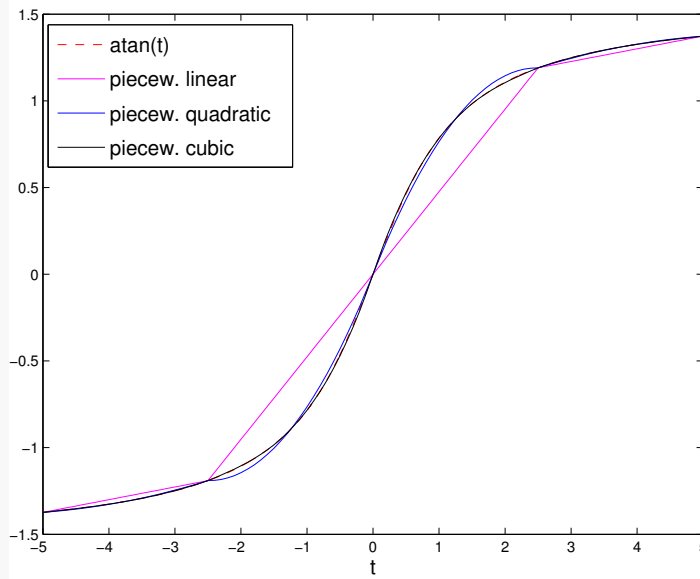
We have the function $f(t) = \arctan t$, $I = [-5, 5]$ and the mesh

$$\mathcal{M} := \{-5 = x_0 < -\frac{5}{2} = x_1 < 0 = x_2 < \frac{5}{2} = x_3 < 5 = x_4\}.$$

Local interpolation nodes are equidistant on I_j .

piecewise linear: $n_j = 1 \quad \mathcal{T}^j = \{t_0^j = x_{j-1}, t_1^j = x_j\}$

piecewise quadratic: $n_j = 2 \quad \mathcal{T}^j = \{x_{j-1}, \frac{x_{j-1}+x_j}{2}, x_j\}$



Plots of the piecewise linear, quadratic and cubic polynomial interpolants

Note. We see overall that the interpolant is C^0 (but not C^1) since x_1, \dots, x_{m-1} are interpolation nodes. A special case is when $n_j = n$ (fixed). The subsequent question is how can we improve our error estimate by decreasing the mesh width $h_{\mathcal{M}} := \max_j |x_{j-1} - x_j|$, i.e. the asymptotic behaviour as $h_{\mathcal{M}} \rightarrow 0$ ("h-convergence").

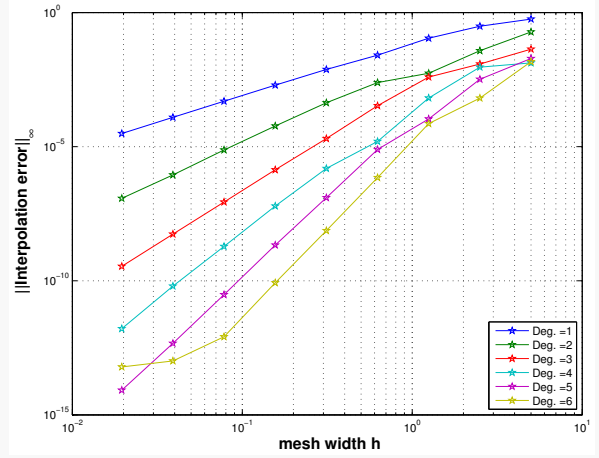
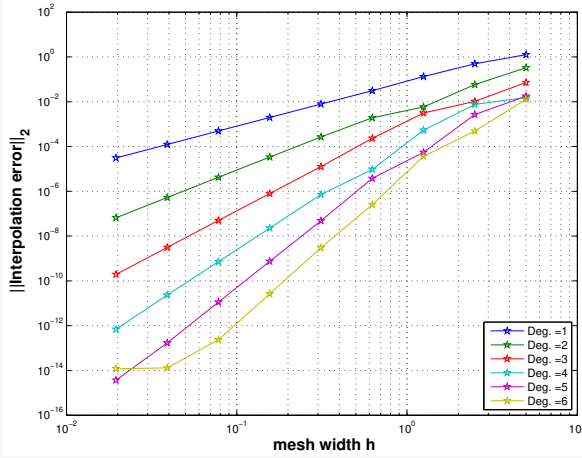
Note that as $h_{\mathcal{M}}$ decreases, the total number of cells increases and therefore also the number of nodes.

$$\# \text{ number of cells} \geq \frac{|b-a|}{h_{\mathcal{M}}}$$

$$\# \text{ number of nodes} \geq \frac{|b-a|(n-1)}{h_{\mathcal{M}}}$$

Example 5.3.2:

We again look at the example $f(t) = \arctan(t)$ on $[-5, 5]$ with equidistant mesh width and $\frac{10}{h_{\mathcal{M}}}$ cells.



We observe in this log-log plot algebraic convergence in $h_{\mathcal{M}}$. Also the rate of algebraic convergence increases with polynomial degree n . The rates α of algebraic convergence $O(h_{\mathcal{M}}^\alpha)$ of the norms of the interpolation error are enlisted below.

n	1	2	3	4	5	6
w.r.t. L^2 -norm	1.9957	2.9747	4.0256	4.8070	6.0013	5.2012
w.r.t. L^∞ -norm	1.9529	2.8989	3.9712	4.7057	5.9801	4.9228

Note. A higher polynomial degree provides faster algebraic decrease of interpolation error norms. The above example shows empiric evidence for rates $\alpha = n + 1$.

Let us derive an error estimate by applying the estimate (5.21) on each subinterval individually:

$$\|f - L_{\mathcal{T}}f\|_{L^\infty(I_j)} \leq \frac{\|f^{(n+1)}\|_{L^\infty(I_j)}}{(n+1)!} \underbrace{\max_{t \in I_j} |(t - t_0^j) \dots (t - t_n^j)|}_{\leq h_{\mathcal{M}}^{n+1}}.$$

Note that

$$\|f - s\|_{L^\infty([x_0, x_m])} = \max_{j \in \{1, \dots, m\}} \|f - L_{\mathcal{T}}f\|_{L^\infty(I_j)}.$$

Hence,

$$\|f - s\|_{L^\infty([x_0, x_m])} \leq \frac{h_{\mathcal{M}}^{n+1}}{(n+1)!} \|f^{(n+1)}\|_{L^\infty([x_0, x_m])}.$$

Thus, we have shown algebraic convergence in $h_{\mathcal{M}}$ with rate $n + 1$.

Note. We can make the following conclusions:

- n is now fixed (and small), e.g. for piecewise linear interpolation, we have $n = 1$ and the estimate holds if $f|_{I_j} \in C^2$.
- Piecewise smoothness of f is sufficient.
- Since n can be small, we do also obtain convergence result for f with low regularity.
- But: only slow convergence (algebraic as opposed to exponential for Chebyshev interpolation).

5.3.2 Spline interpolation

We have seen that polynomial interpolation can straightforwardly be extended to *piecewise* polynomial interpolation by subdividing the global interval with a mesh and performing local polynomial interpolation on each individual cell. This leads to continuous interpolants $s \in C^0([a, b])$. However, in many applications, it is desirable to have smooth interpolants. This is the idea of *spline interpolation*, which can be described as follows:

- Using a polynomial of degree d on each subinterval $[t_{i-1}, t_i]$.
- Matching the first $d - 1$ derivatives at the nodes t_i .

Definition 5.3.1 (Spline space). Given an interval $I := [a, b] \subset \mathbb{R}$ and a *knot set/mesh* $\mathcal{M} := \{a = t_0 < t_1 < \dots < t_{n-1} < t_n = b\}$, the vector space $\mathcal{S}_{d,\mathcal{M}}$ of the *spline functions* of degree d (or order $d + 1$) is defined by

$$\mathcal{S}_{d,\mathcal{M}} := \{s \in C^{d-1}(I) : s_j := s|_{[t_{j-1}, t_j]} \in \mathcal{P}_d \forall j = 1, \dots, n\}.$$

Spline space are mapped onto each other by differentiation and integration:

$$s \in \mathcal{S}_{d,\mathcal{M}} \Rightarrow s' \in \mathcal{S}_{d-1,\mathcal{M}} \quad \text{and} \quad \int_a^t s(\tau) d\tau \in \mathcal{S}_{d+1,\mathcal{M}}.$$

Spline spaces of the lowest degrees:

- $d = 0$: \mathcal{M} -piecewise constant *discontinuous* functions.
- $d = 1$: \mathcal{M} -piecewise linear *continuous* functions.
- $d = 2$: *continuously differentiable* \mathcal{M} -piecewise quadratic functions.

The dimension of a spline space can be found by a counting argument. We count the number of “degrees of freedom” of an \mathcal{M} -piecewise polynomial of degree d , and subtract the number of linear constraints implicitly contained in Definition 5.3.1:

$$\dim \mathcal{S}_{d,\mathcal{M}} = n \cdot \dim \mathcal{P}_d - \#\{C^{d-1} \text{ continuity constraints}\} = n \cdot (d + 1) - (n - 1) \cdot d = n + d.$$

Theorem 5.3.1 (Dimension of spline space). *The space $\mathcal{S}_{d,\mathcal{M}}$ from Definition 5.3.1 has dimension*

$$\dim \mathcal{S}_{d,\mathcal{M}} = n + d.$$

We already know the special case of interpolation in $\mathcal{S}_{1,\mathcal{M}}$, when the interpolation nodes are the knots of \mathcal{M} , because this boils down to simple piecewise linear interpolation, see Section 5.2.1.

Cubic spline interpolation

The human eye perceives a C^2 -functions as “smooth”. Thus, cubic spline interpolation is a very appealing concept. Consider the space:

$$\mathcal{S}_{3,\mathcal{M}} = \{s \in C^2(I) : s_j := s|_{[t_{j-1}, t_j]} \in \mathcal{P}_3 \ \forall j = 1, \dots, n\},$$

where $s \in C^2(I)$ implies the conditions: $s'_j(t_j) = s'_{j+1}(t_j)$, $s''_j(t_j) = s''_{j+1}(t_j)$. Since each $s_j \in \mathcal{P}_3$, $\forall j = 1, \dots, n$ we may denote them as:

$$s_j(t) = a_j + b_j t + c_j t^2 + d_j t^3.$$

In this setting, we have to determine $4n$ coefficients. Next, let us see what $s \in \mathcal{S}_{3,\mathcal{M}}$ implies for the coefficients:

① Interpolating conditions:

$$s_j(t_{j-1}) = y_{j-1}, \quad s_j(t_j) = y_j,$$

yield

$$\begin{aligned} a_j + b_j t_{j-1} + c_j t_{j-1}^2 + d_j t_{j-1}^3 &= y_{j-1}, \\ a_j + b_j t_j + c_j t_j^2 + d_j t_j^3 &= y_j. \end{aligned}$$

Thus, we get $2n$ conditions.

② Smoothness conditions (i):

$$s'_j(t_j) = s'_{j+1}(t_j), \quad j = 1, \dots, n-1,$$

implies

$$b_j + 2c_j t_j + 3d_j t_j^2 = b_{j+1} + 2c_{j+1} t_j + 3d_{j+1} t_j^2.$$

This results in $n-1$ additional conditions.

③ Smoothness conditions (ii):

$$s''_j(t_j) = s''_{j+1}(t_j), \quad j = 1, \dots, n-1,$$

so that

$$2c_j + 6d_j t_j = 2c_{j+1} + 6d_{j+1} t_j,$$

yields $n - 1$ additional conditions.

From ①, ② and ③ we have $2n + 2(n - 1) = 4n - 2$ conditions to determine $4n$ coefficients resulting in 2 degrees of freedom. To fully determine the system we have to add two additional constraints.

④ Additional constraints, e.g. natural/simple boundary conditions:

$$s_1''(t_0) = 0, \quad s_n''(t_n) = 0.$$

Altogether we have constructed a fully determined LSE for the coefficients $\{a_j, b_j, c_j, d_j\}_{j=1, \dots, n}$.

Economical implementation of cubic spline interpolation

Next, we derive an efficient implementation of cubic spline interpolation. For this, we first introduce a slightly altered ansatz for the polynomials s_j :

$$\forall j = 1, \dots, n: \quad s_j(t) = \tilde{a}_j + \tilde{b}_j(t - t_{j-1}) + \tilde{c}_j(t - t_{j-1})^2 + \tilde{d}_j(t - t_{j-1})^3.$$

Defining

$$\sigma_j := s_j''(t_j) = s_{j+1}''(t_j) \text{ and } h_j := t_j - t_{j-1}, \quad j = 1, \dots, n - 1,$$

where σ_j are unknowns, one can obtain the following equations for the coefficients of s_j :

$$\tilde{a}_j = y_{j-1}, \tag{5.46}$$

$$\tilde{b}_j = \frac{y_j - y_{j-1}}{h_j} - \frac{h_j(2\sigma_{j-1} + \sigma_j)}{6}, \tag{5.47}$$

$$\tilde{c}_j = \frac{\sigma_{j-1}}{2}, \tag{5.48}$$

$$\tilde{d}_j = \frac{\sigma_j - \sigma_{j-1}}{6h_j}. \tag{5.49}$$

Equations (5.46)-(5.49) can be verified as follows:

First, the interpolation condition at t_{j-1} and the definition of s_j imply (5.46):

$$s_j(t_{j-1}) = \tilde{a}_j = y_{j-1}.$$

Next, we can express \tilde{c}_j, \tilde{d}_j through the second derivatives σ_{j-1}, σ_j :

$$\begin{aligned} s_j''(t_{j-1}) = 2\tilde{c}_j = \sigma_{j-1} &\implies \tilde{c}_j = \frac{\sigma_{j-1}}{2} \quad \text{and} \\ s_j''(t_j) = \underbrace{2\tilde{c}_j}_{\sigma_{j-1}} + 6\tilde{d}_j(t_j - t_{j-1}) = \sigma_j &\implies \tilde{d}_j = \frac{\sigma_j - \sigma_{j-1}}{6h_j}. \end{aligned}$$

Next, we can derive the equation for \tilde{b}_j by employing the interpolating condition at t_j :

$$s_j(t_j) = \tilde{a}_j + \tilde{b}_j(t_j - t_{j-1}) + \tilde{c}_j(t_j - t_{j-1})^2 + \tilde{d}_j(t_j - t_{j-1})^3 = y_j.$$

Plugging in the expressions derived for \tilde{a}_j, \tilde{c}_j and \tilde{d}_j then yields:

$$\begin{aligned} y_{j-1} + \tilde{b}_j h_j + \frac{\sigma_{j-1}}{2} h_j^2 + \frac{\sigma_j - \sigma_{j-1}}{6 h_j} h_j^3 &= y_j \\ \implies \tilde{b}_j h_j &= y_j - y_{j-1} - h_j^2 \left(\frac{2\sigma_{j-1}}{6} + \frac{\sigma_j}{6} \right) \\ \implies \tilde{b}_j &= \frac{y_j - y_{j-1}}{h_j} - \frac{h_j(2\sigma_{j-1} + \sigma_j)}{6}. \end{aligned}$$

Next, we use the matching of the first derivatives

$$\tilde{b}_j + 2\tilde{c}_j h_j + 3\tilde{d}_j h_j^2 = \tilde{b}_{j+1}$$

and Equations (5.46)-(5.49) to formulate an LSE for the σ_j 's. We obtain

$$\begin{aligned} \frac{y_j - y_{j-1}}{h_j} - \frac{h_j(2\sigma_{j-1} + \sigma_j)}{6} + \sigma_{j-1} h_j + \frac{\sigma_j - \sigma_{j-1}}{2} h_j &= \frac{y_{j+1} - y_j}{h_{j+1}} - \frac{h_{j+1}(2\sigma_j + \sigma_{j+1})}{6} \\ \implies \sigma_{j-1} \left(-\frac{h_j}{3} - \frac{h_j}{2} + h_j \right) + \sigma_j \left(-\frac{h_j}{6} + \frac{h_j}{2} - \frac{h_{j+1}}{3} \right) + \sigma_{j+1} \left(\frac{h_{j+1}}{6} \right) &= \underbrace{\frac{y_{j+1} - y_j}{h_{j+1}} - \frac{y_j - y_{j-1}}{h_j}}_{=: r_j} \\ \implies \sigma_{j-1} \frac{h_j}{6} + \sigma_j \left(\frac{h_j + h_{j+1}}{3} \right) + \sigma_{j+1} \frac{h_{j+1}}{6} &= r_j, \quad j = 1, \dots, n-1, \end{aligned}$$

which is a tridiagonal system for unknowns $[\sigma_1, \dots, \sigma_{n-1}]^\top$:

$$\begin{bmatrix} \frac{h_1+h_2}{3} & \frac{h_2}{6} & 0 & \dots & 0 \\ \frac{h_2}{6} & \frac{h_2+h_3}{3} & \frac{h_3}{6} & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & \frac{h_{n-1}}{6} \\ 0 & \dots & 0 & \frac{h_{n-1}}{6} & \frac{h_{n-1}+h_n}{3} \end{bmatrix} \begin{bmatrix} \sigma_1 \\ \vdots \\ \vdots \\ \sigma_{n-1} \end{bmatrix} = \begin{bmatrix} r_1 \\ \vdots \\ \vdots \\ r_{n-1} \end{bmatrix}.$$

This $(n-1) \times (n-1)$ LSE is sparse and hence efficiently solvable. If we have $[\sigma_1, \dots, \sigma_{n-1}]^\top$, then the coefficients $\{\tilde{a}_j, \tilde{b}_j, \tilde{c}_j, \tilde{d}_j\}$, $j = 1, \dots, n$ are determined by (5.46)-(5.49).

Note. Finally, we remark that a similar estimate as seen for piecewise Lagrange interpolation is possible for cubic spline interpolation on an equidistant mesh with mesh width h :

$$f \in C^4([t_0, t_n]) : \quad \|f - s\|_{L^\infty([t_0, t_n])} \leq \frac{5}{384} h^4 \|f^{(4)}\|_{L^\infty([t_0, t_n])}.$$

Numerical Quadrature

Numerical quadrature deals with the *approximate* numerical evaluation of integrals $\left(\int_{\Omega} f(\mathbf{t}) \, d\mathbf{t}\right)$ for a given (closed) integration domain $\Omega \subset \mathbb{R}^d$ by using point evaluations of the function f . Thus, the underlying integration problem can be reformulated as a mapping:

$$I : \begin{cases} C^0(\Omega) & \rightarrow \mathbb{R} \\ f & \mapsto \int_{\Omega} f(\mathbf{t}) \, d\mathbf{t} \end{cases} ,$$

with the data space $X := C^0(\Omega)$ and the result space $Y := \mathbb{R}$.

Why is numerical integration important? There are a number of reasons:

- The function $f(\mathbf{t})$ itself could only be given by some sampling points.
- $f(\mathbf{t})$ could be given as a formula, however the integral $\int_{\Omega} f(\mathbf{t}) \, d\mathbf{t}$ could be too difficult to compute analytically.
- Even if we are given a formula for the integral $\int_{\Omega} f(\mathbf{t}) \, d\mathbf{t}$, numerical integration could be faster.

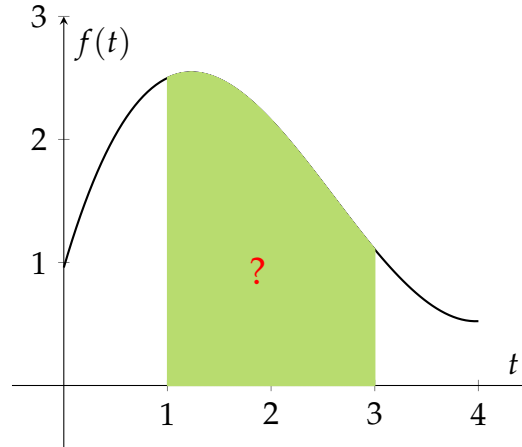
Throughout this chapter, for simplicity, we use the special case where $d = 1$, $\Omega = [a, b]$, $a, b \in \mathbb{R}$ is an interval and $\mathbf{t} \in \mathbb{R}$.

Remark. Multidimensional numerical quadrature is substantially more difficult, unless Ω is a tensor-product domain. Multidimensional numerical quadrature will be treated in the course “*Numerical Methods for Partial Differential Equations*”.

For $d = 1$, from a geometric point of view, methods for numerical quadrature aimed at computing

$$\int_a^b f(t) dt$$

seek to *approximate* an area under the graph of the function f .



In green: The area corresponding to the value of the integral on $\Omega = [1, 3]$.

6.1 Quadrature Formulas

Quadrature formulas realize the approximation of an integral through finitely many point evaluations of the integrand $f(t)$.

Definition 6.1.1 (Quadrature formula/quadrature rule). An n -point *quadrature formula/quadrature rule* on $[a, b]$ provides an approximation of the value of an integral through a *weighted sum* of point values of the integrand:

$$\int_a^b f(t) dt \approx Q_n(f) := \sum_{j=1}^n w_j^n f(c_j^n), \quad (6.1)$$

where w_j^n are called *quadrature weights* $\in \mathbb{R}$ and c_j^n are the *quadrature nodes* $\in [a, b]$.

A single invocation of $Q_n(f)$ costs n point evaluations of the integrand plus n additions and multiplications.

In the setting of function approximation by polynomials, it can be shown that an approximation scheme for any interval could be obtained from an approximation scheme on a single reference interval $[-1, 1]$ (cf. Section 5.2.5). A similar affine transformation technique makes it possible to derive quadrature formulas for arbitrary intervals from a single quadrature formula given on a reference interval.

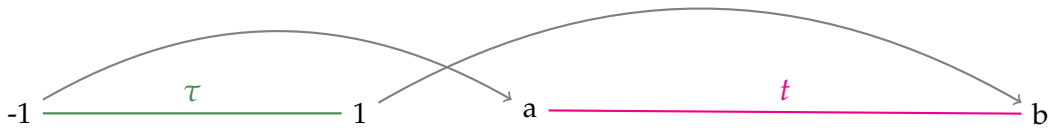
More precisely, suppose we are given the quadrature formula $(\hat{c}_j, \hat{w}_j)_{j=1}^n$ on the *reference interval* $[-1, 1]$. Then $\int_a^b f(t) dt$ can be transformed to $[-1, 1]$:

$$\int_a^b f(t) dt = \int_{-1}^1 f(\Phi(\tau)) \Phi'(\tau) d\tau = \frac{(b-a)}{2} \int_{-1}^1 \hat{f}(\tau) d\tau, \quad (6.2)$$

with

$$\Phi(\tau) = \frac{1}{2}(1-\tau)a + \frac{1}{2}(1+\tau)b,$$

and $\hat{f} = f \circ \Phi$.



$$\tau \mapsto t : \Phi(\tau) := \frac{1}{2}(1-\tau)a + \frac{1}{2}(\tau+1)b$$

We find the quadrature formula for a general interval $[a, b]$:

$$\int_a^b f(t) dt \approx \frac{1}{2}(b-a) \sum_{j=1}^n \hat{w}_j \hat{f}(\hat{c}_j) = \sum_{j=1}^n w_j f(c_j),$$

with

$$c_j = \frac{1}{2}(1-\hat{c}_j)a + \frac{1}{2}(1+\hat{c}_j)b, \quad w_j = \frac{1}{2}(b-a)\hat{w}_j.$$

In words, the nodes are just mapped through the affine transformation $c_j = \Phi(\hat{c}_j)$, while the weights are scaled by the ratio of lengths of $[a, b]$ and $[-1, 1]$: $w_j = \frac{|[a, b]|}{|[-1, 1]|} \hat{w}_j$.

6.1.1 Quadrature by approximation schemes

Given an approximation scheme $A : C^0([a, b]) \rightarrow V$, where V is a space of “simple functions” on $[a, b]$, we can find a numerical integration method:

$$\int_a^b f(t) dt \approx \int_a^b (Af)(t) dt =: Q_A(f). \quad (6.3)$$

Recall that every interpolation scheme induces an approximation scheme. An interpolation scheme $\mathcal{I}_{\mathcal{T}}$ with node set $\mathcal{T} = \{t_1, \dots, t_n\} \subset [a, b]$ induces:

$$\int_a^b f(t) dt \approx \int_a^b \mathcal{I}_{\mathcal{T}}[f(t_1), \dots, f(t_n)]^\top(t) dt. \quad (6.4)$$

We can therefore deduce the following lemma:

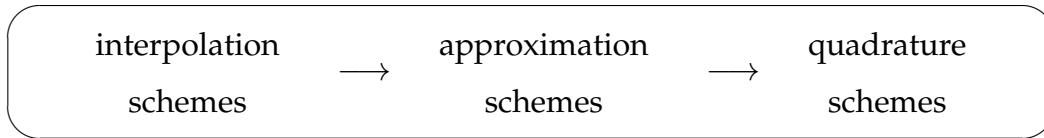
Lemma 6.1.1 (Quadrature formulas from linear interpolation schemes). *Every linear interpolation operator $\mathcal{I}_{\mathcal{T}}$ according to Definition 5.1.1 spawns a quadrature formula by (6.4).*

Proof. We can rewrite (6.4) in the following way with \mathbf{e}_j being the j -th unit vector of \mathbb{R}^n :

$$\begin{aligned} \int_a^b \mathcal{I}_{\mathcal{T}} [f(t_1), \dots, f(t_n)]^{\top} (t) dt &= \int_a^b \mathcal{I}_{\mathcal{T}} \left[\sum_{j=1}^n f(t_j) \cdot \mathbf{e}_j \right] (t) dt \\ &\stackrel{\substack{= \\ \uparrow \\ \text{linearity of } \mathcal{I}_{\mathcal{T}}}}{=} \sum_{j=1}^n f(t_j) \underbrace{\int_a^b (\mathcal{I}_{\mathcal{T}}(\mathbf{e}_j))(t) dt}_{\text{weight } w_j^n :=} \\ &= \sum_{j=1}^n w_j^n f(t_j) . \end{aligned} \tag{6.5}$$

Hence, we have arrived at an n -point quadrature formula with nodes t_j , whose weights are the integrals of the *cardinal interpolants* for the interpolation scheme $\mathcal{I}_{\mathcal{T}}$. \square

Summing up, we have found:



Furthermore, we introduce the concept of the *quadrature error* since, in general, the quadrature formula (6.1) will only provide an approximate value for the integral.

Definition 6.1.2 (Quadrature error). The quadrature error is defined as

$$E_n(f) := \left| \int_a^b f(t) dt - Q_n(f) \right| .$$

As in the case of function approximation by interpolation (Section 5.2.4), our focus will be on the *asymptotic* behavior of the quadrature error as a function of the number n of point evaluations of the integrand. Therefore consider *families* of quadrature rules $\{Q_n\}_n$ (see Definition 6.1.1) described by

- quadrature weights $\{w_j^n, j = 1, \dots, n\}_{n \in \mathbb{N}}$ and
- quadrature nodes $\{c_j^n, j = 1, \dots, n\}_{n \in \mathbb{N}}$.

We study the asymptotic behavior of the quadrature error $E_n(f)$ for $n \rightarrow \infty$. As in the case of interpolation errors, we make the qualitative distinction:

- algebraic convergence: $E_n(f) = O(n^{-p})$, rate $p > 0$
- exponential convergence: $E_n(f) = O(q^n)$, $0 \leq q < 1$

Note that the number n of nodes agrees with the number of f -evaluations required for evaluation of the quadrature formula. This is usually used as a *measure for the cost* of computing $Q_n(f)$. If interpolation is used as the approximation scheme, we can straightforwardly bound the quadrature error:

$$\begin{aligned} E_n(f) &= \left| \int_a^b \left(f(t) - \mathcal{I}_{\mathcal{T}}[f(t_1), \dots, f(t_n)]^\top(t) \right) dt \right| \\ &\leq |b-a| \cdot \underbrace{\left\| f(t) - \mathcal{I}_{\mathcal{T}}[f(t_1), \dots, f(t_n)]^\top(t) \right\|_{L^\infty([a,b])}}_{\text{interpolation error}}. \end{aligned} \quad (6.6)$$

Hence, the various estimates derived in Section 5.2.4 and Section 5.2.5 give us quadrature error estimates “for free”.

6.2 Polynomial Quadrature Formulas

Now we look at the quadrature formulas induced by Lagrange interpolation schemes $\mathcal{I}_{\mathcal{T}}(f)$ as introduced in Definition 5.2.2.

Idea: Replace integrand f with $p_{n-1} := \mathcal{I}_{\mathcal{T}}(f) \in \mathcal{P}_{n-1}$, which is the polynomial Lagrange interpolant of f for a given set of nodes $\mathcal{T} := \{t_0, \dots, t_{n-1}\} \subset [a, b]$. Then, integrate over this approximation of f to obtain an approximation of the integral of f :

$$\int_a^b f(t) dt \approx Q_n(f) := \int_a^b p_{n-1}(t) dt. \quad (6.7)$$

The cardinal interpolants for Lagrange interpolation are the Lagrange polynomials,

$$L_j(t) := \prod_{\substack{i=0 \\ i \neq j}}^{n-1} \frac{t - t_i}{t_j - t_i}, \quad j = 0, \dots, n-1 \quad \xrightarrow{(5.16)} \quad p_{n-1}(t) = \sum_{j=0}^{n-1} f(t_j) L_j(t).$$

Then (6.5) amounts to the n -point quadrature formula

$$\int_a^b p_{n-1}(t) dt = \sum_{j=0}^{n-1} f(t_j) \int_a^b L_j(t) dt = \sum_{j=1}^n f(t_{j-1}) \int_a^b L_{j-1}(t) dt, \quad (6.8)$$

with the nodes $c_j := t_{j-1}$ and the weights $w_j := \int_a^b L_{j-1}(t) dt$.

6.2.1 Newton-Cotes formulas

The n -point Newton-Cotes formulas arise from Lagrange interpolation on equidistant nodes (5.2.4) in the integration interval $[a, b]$.

The equidistant quadrature nodes are given by

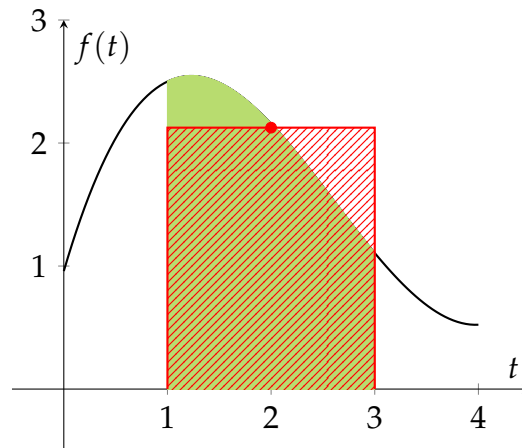
$$t_j := a + hj, \quad h := \frac{b-a}{n-1}, \quad j = 0, \dots, n-1.$$

The weights for the interval $[0, 1]$ can be found by e.g. symbolic computation using Mathematica. Weights on general intervals $[a, b]$ can then be deduced by the affine transformation rule as outlined in Section 6.1.

$n = 1$: Midpoint rule

The *midpoint rule* corresponds to (6.8) for $n = 1$ and $t_0 = \frac{1}{2}(a + b)$. It leads to the 1-point quadrature formula

$$\int_a^b f(t) dt \approx \hat{Q}_{\text{mp}}(f) = (b-a)f\left(\frac{1}{2}(a+b)\right).$$

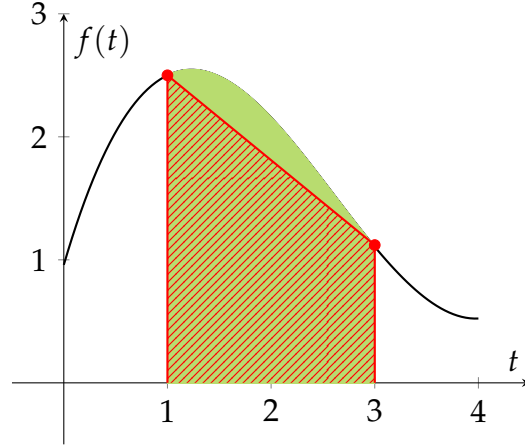


The area under the graph of f is approximated by the area of a rectangle.

$n = 2$: Trapezoidal rule

Here, an approximation of f is constructed from the linear interpolant through the endpoints. Integrating this approximation yields the trapezoidal quadrature rule:

$$\int_a^b f(t) dt \approx \hat{Q}_{\text{trp}}(f) := \frac{b-a}{2} (f(a) + f(b)). \quad (6.9)$$



Note that according to (6.8),

$$w_i = \int_a^b L_{i-1}(t) dt ,$$

so that for $n = 2$ we can calculate

$$\begin{aligned} w_1 &= \int_a^b L_0(t) dt = \int_a^b \frac{t-b}{a-b} dt = \frac{b-a}{2} , \\ w_2 &= \int_a^b L_1(t) dt = \int_a^b \frac{t-a}{b-a} dt = \frac{b-a}{2} . \end{aligned}$$

$n = 3$: **Simpson rule**

For $n = 3$, we can derive the Simpson rule:

$$\hat{Q}_{\text{simp}}(f) := \frac{h}{6} \left(f(0) + 4f\left(\frac{1}{2}\right) + f(1) \right) .$$

For a general interval $[a, b]$, this can be given by

$$\int_a^b f(t) dt \approx \frac{b-a}{6} \left(f(a) + 4f\left(\frac{a+b}{2}\right) + f(b) \right) . \quad (6.10)$$

Recall that the Lagrange interpolation with equidistant nodes is unstable for large n . The considerations of Section 5.2.5 confirmed the superior stability properties of the “optimal” Chebyshev nodes (5.25) for global polynomial Lagrange interpolation. This suggests that we also use these nodes for numerical quadrature with weights given by (6.8). This yields the so-called *Clenshaw-Curtis rules*. The weights of any n -point Clenshaw-Curtis rule can be computed with a computational effort of $\mathcal{O}(n \log n)$ using the FFT.

One common concept for the quality of a quadrature rule is that of its *order*. In what follows, we will introduce this concept and see that it gives rise to a different family of quadrature rules.

6.3 Gauss Quadrature

How do we gauge the “quality” of an n -point quadrature formula Q_n without testing it for specific integrands? The next definition gives an answer.

Definition 6.3.1 (Order of a quadrature rule). The *order* of quadrature rule $Q_n : C^0([a, b]) \rightarrow \mathbb{R}$ is defined as

$$\text{order}(Q_n) := \max\{m \in \mathbb{N}_0 : Q_n(p) = \int_a^b p(t) dt \quad \forall p \in \mathcal{P}_m\} + 1, \quad (6.11)$$

that is, as the maximal degree $+1$ of polynomials for which the quadrature rule is guaranteed to be exact.

Note that the order of a quadrature formula is invariant under affine transformations. Since we know that a polynomial quadrature formula with n points is exact for $p \in \mathcal{P}_{n-1}$, we can deduce that it is of order $\geq n$. This observation leads to the following question: When does an n -point quadrature formula have order $\geq n$? It is answered in the following theorem.

Theorem 6.3.1 (Sufficient order conditions for quadrature rules). *An n -point quadrature rule on $[a, b]$*

$$Q_n(f) := \sum_{j=1}^n w_j f(c_j), \quad f \in C^0([a, b]),$$

with nodes $c_j \in [a, b]$ and weights $w_j \in \mathbb{R}$, $j = 1, \dots, n$, has order $\geq n$, if and only if

$$w_j = \int_a^b L_{j-1}(t) dt, \quad j = 1, \dots, n,$$

where L_k , $k = 0, \dots, n-1$, is the k -th Lagrange polynomial (5.14) associated with the ordered node set $\{t_0, t_1, \dots, t_{n-1}\}$, where $t_{j-1} := c_j$.

Thus, for a quadrature formula Q_n with order $\geq n$, the weights w_j only depend on the node set $\mathcal{T} = \{c_1, c_2, \dots, c_n\}$.

Proof. The property that Q_n has order $\geq n$ is equivalent to

$$Q_n(p) = \int_a^b p(t) dt \quad \forall p \in \mathcal{P}_{n-1}.$$

Also note that $\mathcal{P}_{n-1} = \text{Span}\{L_0, \dots, L_{n-1}\}$. Thus,

$$\begin{aligned} Q_n(p) &= \int_a^b p(t) dt \quad \forall p \in \mathcal{P}_{n-1} \\ \iff Q_n(L_{i-1}) &= \int_a^b L_{i-1}(t) dt \quad \forall i \in \{1, \dots, n\} \\ \iff \sum_{j=1}^n w_j \underbrace{L_{i-1}(t_{j-1})}_{=\delta_{i,j}} &= \int_a^b L_{i-1}(t) dt \quad \forall i \in \{1, \dots, n\} \\ \iff w_i &= \int_a^b L_{i-1}(t) dt \quad \forall i \in \{1, \dots, n\}. \end{aligned}$$

□

Now that we have seen a necessary and sufficient condition for an n -point quadrature formula to have order at least n , another natural question arises: Can an n -point quadrature formula achieve order $> n$? The following result limits the maximal order that can be achieved:

Theorem 6.3.2 (Maximal order of n -point quadrature rule). *The maximal order of an n -point quadrature rule is $2n$.*

Proof. Consider a generic n -point quadrature rule according to Definition 6.1.1:

$$Q_n(f) := \sum_{j=1}^n w_j^n f(c_j^n) .$$

We build a polynomial of degree $2n$ that cannot be integrated exactly by the quadrature formula Q_n .

For this, we define: $q(t) := (t - c_1^n)^2 \cdots (t - c_n^n)^2 \in \mathcal{P}_{2n}$.

Note that $q(t) > 0$ almost everywhere implies $\int_a^b q(t) dt > 0$. On the other hand, by the definition of $q(t)$, the quadrature formula Q_n applied to q evaluates to:

$$Q_n(q) = \sum_{j=1}^n w_j^n \underbrace{q(c_j^n)}_{=0} = 0 .$$

Hence,

$$0 = Q_n(q) \neq \int_a^b q(t) dt > 0 .$$

This implies that Q_n has order $\leq 2n$.

□

Example 6.3.1: 2-point quadrature formula Q_2 with order 4 (on $[-1, 1]$)

We want to find a 2-point quadrature formula of order 4, i.e.

$$Q_2(p) = \int_a^b p(t) dt \quad \forall p \in \mathcal{P}_3 .$$

It suffices to verify the exactness of Q_2 for the monomials, since they form a basis of the space of polynomials:

$$Q_2(\{t \mapsto t^q\}) = \frac{1}{q+1} (b^{q+1} - a^{q+1}) , \quad \text{for } q = 0, 1, 2, 3 .$$

We obtain 4 equations for the weights w_j and nodes c_j , $j = 1, 2$: (recall: $a = -1, b = 1$)

$$\begin{aligned} \int_{-1}^1 1 \, dt &= 2 = 1w_1 + 1w_2, & \int_{-1}^1 t \, dt &= 0 = c_1w_1 + c_2w_2, \\ \int_{-1}^1 t^2 \, dt &= \frac{2}{3} = c_1^2w_1 + c_2^2w_2, & \int_{-1}^1 t^3 \, dt &= 0 = c_1^3w_1 + c_2^3w_2. \end{aligned} \quad (6.12)$$

These are 4 (non-linear) equations in 4 unknowns from which we can calculate the weights w_j and nodes c_j :

$$w_2 = 1, w_1 = 1, c_1 = \frac{1}{\sqrt{3}}, c_2 = -\frac{1}{\sqrt{3}}.$$

These weights and nodes yield the following quadrature formula of order 4:

$$\int_{-1}^1 f(x) \, dx \approx f\left(\frac{1}{\sqrt{3}}\right) + f\left(-\frac{1}{\sqrt{3}}\right). \quad (6.13)$$

The above example shows that there exists indeed a (unique) 2-point quadrature formula of order 4. Does this result generalize? More precisely, is there a family Q_n of quadrature formulas such that Q_n is n -point and of order $2n$? The following theorem answers this question affirmatively.

Theorem 6.3.3 (Existence of n -point quadrature formulas of order $2n$). *Let $\{\bar{P}_n\}_{n \in \mathbb{N}_0}$ be a family of non-zero polynomials that satisfies*

- $\bar{P}_n \in \mathcal{P}_n$,
- $\int_{-1}^1 q(t) \bar{P}_n(t) \, dt = 0$ for all $q \in \mathcal{P}_{n-1}$ ($L^2([-1, 1])$ -orthogonality),
- the set $\{c_j^n\}_{j=1}^m$, $m \leq n$, of real zeros of \bar{P}_n is contained in $[-1, 1]$.

Then the quadrature rule

$$Q_n(f) := \sum_{j=1}^m w_j^n f(c_j^n)$$

with weights chosen according to Theorem 6.3.1 provides a quadrature formula of order $2n$ on the interval $[-1, 1]$.

One can show that the nodes of an n -point quadrature formula with order $2n$ have to be the zeros of \bar{P}_n .

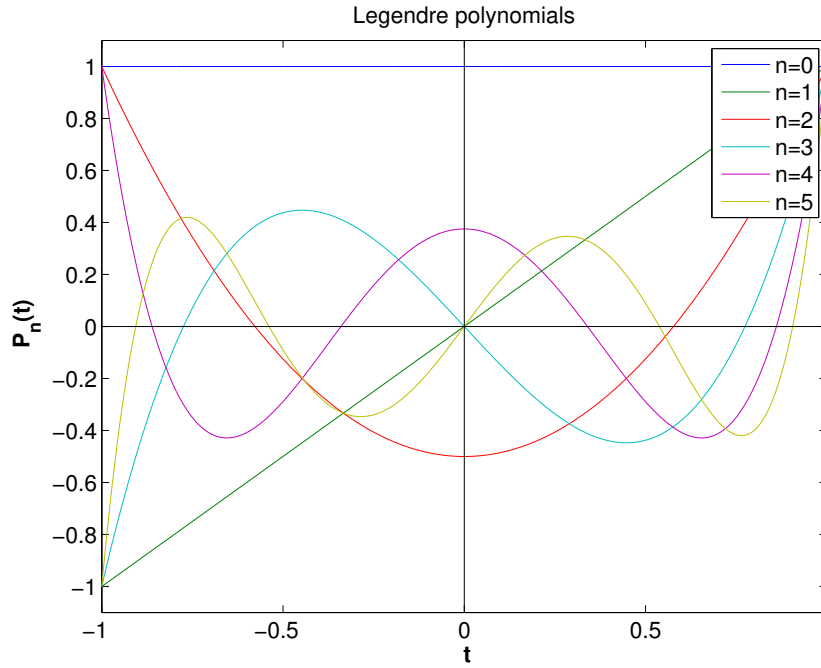
Note. n -point quadrature formulas of order $2n$ are unique.

The polynomials \bar{P}_n are equivalent to the *Legendre polynomials* up to a scaling factor:

Definition 6.3.2 (Legendre polynomials). The n -th Legendre polynomial P_n is defined by

- $P_n \in \mathcal{P}_n$,
- $\int_{-1}^1 P_n(t)q(t) dt = 0 \quad \forall q \in \mathcal{P}_{n-1}$,
- $P_n(1) = 1$.

The first six Legendre polynomials P_0, \dots, P_5 are depicted in the following figure:



In addition, we can prove the following lemma:

Lemma 6.3.1 (Zeros of Legendre polynomials). P_n has n distinct zeros in $(-1, 1)$.

Proof. Assume that on $(-1, 1)$, P_n has only $m < n$ distinct zeros given by $\zeta_1, \dots, \zeta_m \in (-1, 1)$. This implies that P_n changes sign at ζ_1, \dots, ζ_m . Now define

$$q(t) := \prod_{j=1}^m (t - \zeta_j) \in \mathcal{P}_m \subset \mathcal{P}_{n-1}.$$

This implies that q changes sign at ζ_1, \dots, ζ_m . So we can conclude that $P_n(t)q(t)$ cannot change sign on $(-1, 1)$. Thus

$$P_n \cdot q \geq 0 \text{ on } (-1, 1) \text{ or } P_n \cdot q \leq 0 \text{ on } (-1, 1).$$

This implies

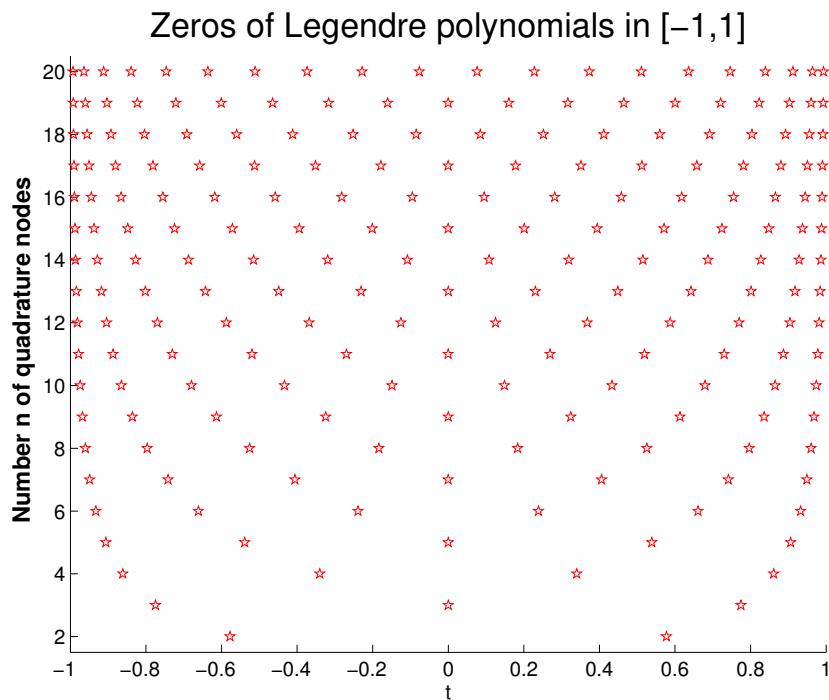
$$\int_{-1}^1 P_n(t)q(t) dt \neq 0.$$

However, by definition of P_n , we know that

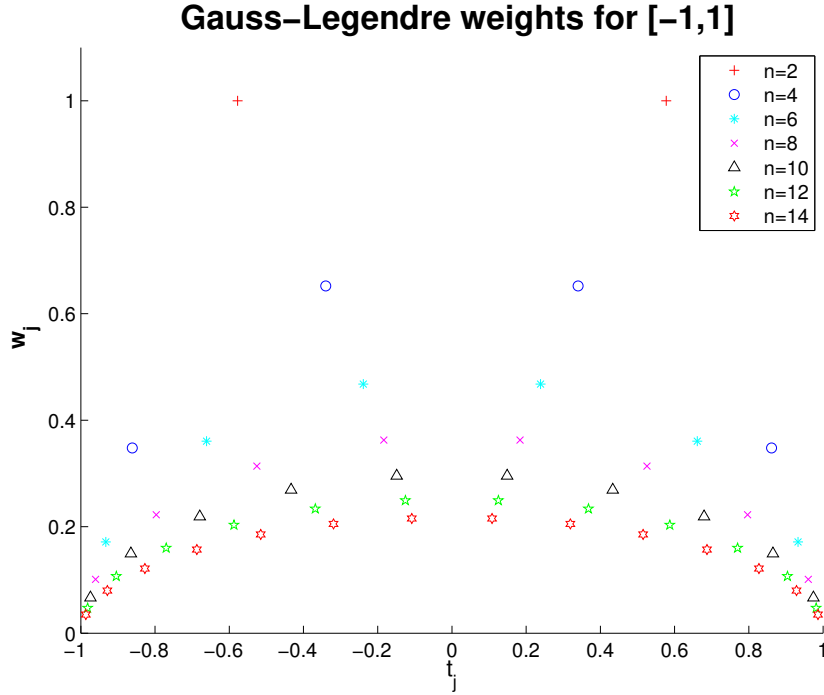
$$\int_{-1}^1 P_n(t)q(t) dt = 0 \quad \forall q \in \mathcal{P}_{n-1}.$$

This yields a contradiction and thus, $m = n$. □

The zeros of Legendre polynomials are referred to as the *Gauss points*, which are plotted below.



Definition 6.3.3 (Gauss-Legendre quadrature formulas). The n -point quadrature formulas whose nodes are given by the zeros of the n -th Legendre polynomial (see Definition 6.3.2), and whose weights are chosen according to Theorem 6.3.1, are called *Gauss-Legendre quadrature formulas*.



The Gauss-Legendre weights are all positive (cf. Lemma 6.3.2).

Lemma 6.3.2 (Positivity of Gauss-Legendre quadrature weights). *The weights of the Gauss-Legendre quadrature formulas are positive.*

Proof. Let ξ_j^n , $j = 1, \dots, n$ denote the Gauss points of the n -point Gauss-Legendre quadrature formula. Define the polynomial

$$q_k(t) := \prod_{\substack{j=1 \\ j \neq k}}^n (t - \xi_j^n)^2.$$

By definition, we have $q_k \in \mathcal{P}_{2n-2}$. This implies that the n -point Gauss-Legendre quadrature formula integrates q_k exactly:

$$\int_{-1}^1 q_k(t) dt = \sum_{j=1}^n w_j^n \cdot q_k(\xi_j^n).$$

Note that the left-hand side of the above is strictly positive and that the right-hand side reduces to

$$\sum_{j=1}^n w_j^n \cdot q_k(\xi_j^n) = w_k^n \cdot q_k(\xi_k^n)$$

since by definition, $q_k(\xi_j^n) = 0$ for $j \neq k$. Thus we obtain

$$0 < w_k^n \cdot \underbrace{q_k(\xi_k^n)}_{>0}$$

and hence

$$w_k^n > 0 \quad \text{for } k = 1, \dots, n.$$

□

Recursive formula for Legendre polynomials

Legendre polynomials satisfy a 3-term recursion (similar to Chebyshev polynomials):

$$P_{n+1}(t) := \frac{2n+1}{n+1} t P_n(t) - \frac{n}{n+1} P_{n-1}(t) \quad , \quad P_0 \equiv 1 \quad , \quad P_1(t) := t . \quad (6.14)$$

6.3.1 Quadrature error and best approximation error

The positivity of the weights w_j^n for all n -point Gauss-Legendre (and in fact also for all Clenshaw-Curtis) quadrature rules has important consequences.

Theorem 6.3.4 (Quadrature error estimate for quadrature rules with positive weights). *For every n -point quadrature rule Q_n as in (6.1) of order $q \in \mathbb{N}$ with weights $w_j \geq 0$, $j = 1, \dots, n$, the quadrature error satisfies*

$$E_n(f) := \left| \int_a^b f(t) dt - Q_n(f) \right| \leq 2|b-a| \underbrace{\inf_{p \in \mathcal{P}_{q-1}} \|f - p\|_{L^\infty([a,b])}}_{\text{best approximation error}} \quad \forall f \in C^0([a,b]) . \quad (6.15)$$

Next, we remark that the following bound holds for the best approximation error:

Theorem 6.3.5 (L^∞ polynomial best approximation estimate). *If $f \in C^r([a,b])$ (r times continuously differentiable), $r \in \mathbb{N}$, then, for any polynomial degree $n \geq r$,*

$$\inf_{p \in \mathcal{P}_n} \|f - p\|_{L^\infty([a,b])} \leq \left(1 + \frac{\pi^2}{2}\right)^r \frac{(n-r)!}{n!} \left(\frac{b-a}{2}\right)^r \|f^{(r)}\|_{L^\infty([a,b])} .$$

With this, we immediately get results about the asymptotic decay of the quadrature error for n -point Gauss-Legendre and Clenshaw-Curtis quadrature as $n \rightarrow \infty$. Appealing to inequality (5.21), the dependence of the constants on the length of the integration interval can be quantified for integrands with limited smoothness.

Lemma 6.3.3 (Quadrature error estimates for C^r -integrands). *For every n -point quadrature rule Q_n as in (6.1) of order $q \in \mathbb{N}$ with weights $w_j \geq 0$, $j = 1, \dots, n$ we find that the quadrature error $E_n(f)$ for an integrand $f \in C^r([a, b])$, $r \in \mathbb{N}_0$, satisfies*

$$\text{in the case } q \geq r: \quad E_n(f) \leq C q^{-r} |b-a|^{r+1} \|f^{(r)}\|_{L^\infty([a,b])}, \quad (6.16)$$

$$\text{in the case } q < r: \quad E_n(f) \leq \frac{|b-a|^{q+1}}{q!} \|f^{(q)}\|_{L^\infty([a,b])}, \quad (6.17)$$

with a constant $C > 0$ independent of n , f , and $[a, b]$.

If $f \in C^r([a, b])$: $E_n(f) = \mathcal{O}(n^{-r})$, i.e. algebraic convergence with rate r ,

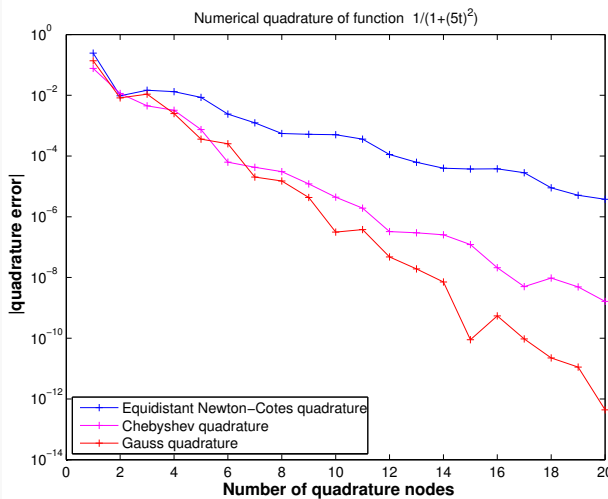
If $f \in C^\infty([a, b])$: $E_n(f) = \mathcal{O}(\lambda^n)$, where $\lambda \in (0, 1)$, i.e. exponential convergence.

Please note the different estimates depending on whether the smoothness of f (as described by r) or the order of the quadrature rule (q) is the “limiting factor”.

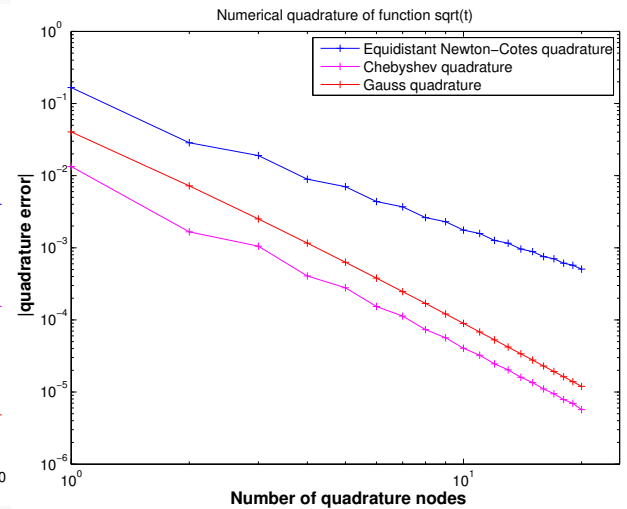
Example 6.3.2: Convergence of global quadrature formulas.

We examine three families of global polynomial quadrature rules: Newton-Cotes formulas, Gauss-Legendre rules, and Clenshaw-Curtis rules. We record the convergence of the quadrature errors for the interval $[0, 1]$ and two different functions:

1. $f_1(t) = \frac{1}{1+(5t)^2}$, a smooth function,
2. $f_2(t) = \sqrt{t}$, a merely continuous function with singular derivative in $t = 0$.



Quadrature error $E_n(f_1)$ on $[0, 1]$



Quadrature error $E_n(f_2)$ on $[0, 1]$

1. The linear-logarithmic plot for $E_n(f_1)$ indicates exponential convergence $E_n \approx \mathcal{O}(\lambda^n)$, $0 < \lambda < 1$.

- Newton-Cotes quadrature : $\lambda \approx 0.61$,

- Clenshaw-Curtis quadrature : $\lambda \approx 0.40$,
 - Gauss-Legendre quadrature : $\lambda \approx 0.27$.
2. The double-logarithmic plot for $E_n(f_2)$ indicates algebraic convergence $\epsilon_n \approx \mathcal{O}(n^{-r})$, $r > 0$.
- Newton-Cotes quadrature : $r \approx 1.8$,
 - Clenshaw-Curtis quadrature : $r \approx 2.5$,
 - Gauss-Legendre quadrature : $r \approx 2.7$.

Note. We saw in the second example that the lack of smoothness limits the convergence of the integrand severely. Next we look at possible manipulations to find a better integrand for the second function, $f_2(t) = \sqrt{t}$, $t \in [0, 1]$: Approximate

$$\int_0^1 \sqrt{t} dt$$

and use the substitution

$$s = \sqrt{t}, \quad dt = 2s ds,$$

so that

$$\int_0^1 \sqrt{t} dt = \int_0^1 \underbrace{2s^2}_{\in C^\infty} ds.$$

We can now apply the quadrature formula on the new smooth function $2s^2$ instead of \sqrt{t} .

For a more general case, the same argument applies: Approximate

$$\int_0^b \sqrt{t} g(t) dt, \quad g \in C^\infty([0, b])$$

and use the substitution

$$s = \sqrt{t}, \quad dt = 2s ds$$

to obtain

$$\int_0^b \sqrt{t} g(t) dt = \int_0^{\sqrt{b}} \underbrace{2s^2 g(s^2)}_{\in C^\infty([0, b])} ds.$$

Next, one can apply the Gauss-Legendre quadrature rule to the smooth integrand.

There is one drawback of most n -asymptotic estimates obtained from Theorem 6.3.4: the bounds usually involve quantities like norms of higher derivatives of the interpolant that are elusive in general. Such unknown quantities are often hidden in “generic constants C ”. Therefore, we have to ask if we can extract useful information out of such estimates. To answer this question, we assume *sharp* algebraic or exponential convergence and ask what the additional cost is for reducing the error by a certain factor ρ . This is what asymptotic rates can tell us despite the unknown hidden constants in those rates. We consider error reduction for both cases of algebraic and exponential convergence:

Sharp algebraic convergence

Fix an integrand f and assume *sharp* algebraic convergence (in n) with rate $r \in \mathbb{N}$ of the quadrature error $E_n(f)$ for a family of n -point quadrature rules:

$$E_n(f) = \mathcal{O}(n^{-r}) \xrightarrow{\text{sharp}} E_n(f) \approx Cn^{-r}, \quad \text{where } C > 0 \text{ is independent of } n. \quad (6.18)$$

Task: Change the quadrature formula to reduce the quadrature error by a factor of $\rho > 1$ by a minimal increase in the number n of quadrature points:

$$\frac{Cn_{\text{old}}^{-r}}{Cn_{\text{new}}^{-r}} \stackrel{!}{=} \rho \implies \boxed{n_{\text{new}} : n_{\text{old}} = \rho^{\frac{1}{r}}}. \quad (6.19)$$

We conclude: In the case of *algebraic convergence* with rate $r \in \mathbb{R}$, a reduction of the quadrature error by a factor of ρ is obtained by an increase of the number of quadrature points by a factor of $\rho^{\frac{1}{r}}$.

Note that (6.19) implies that improving accuracy is “cheaper” for larger r i.e. smoother integrand.

Sharp exponential convergence

Assume *sharp* exponential convergence (in n) for a family of n -point quadrature formulas (for a fixed integrand f):

$$E_n(f) = \mathcal{O}(\lambda^n) \xrightarrow{\text{sharp}} E_n(f) \approx C\lambda^n, \quad (6.20)$$

where we want to reduce the error by a factor of $\rho > 1$ and $C > 0$ denotes a “generic” constant independent of n .

$$\begin{aligned} \frac{C \cdot \lambda^{n_{\text{old}}}}{C \cdot \lambda^{n_{\text{new}}}} &\stackrel{!}{=} \rho \iff \lambda^{n_{\text{old}} - n_{\text{new}}} = \rho \\ &\iff (n_{\text{old}} - n_{\text{new}}) \cdot \log \lambda = \log \rho \\ &\iff n_{\text{new}} - n_{\text{old}} = \underbrace{-\frac{\log \rho}{\log \lambda}}_{>0} \\ &\implies \boxed{n_{\text{new}} = n_{\text{old}} + \left\lceil \left| \frac{\log \rho}{\log \lambda} \right| \right\rceil}. \end{aligned}$$

We conclude: In the case of *exponential convergence* given by (6.20), a fixed increase of the number of quadrature points by $\left\lceil \left| \frac{\log \rho}{\log \lambda} \right| \right\rceil$ results in a reduction of the quadrature error by a factor greater or equal to $\rho > 1$.

6.4 Composite Quadrature

As done in Chapter 5 for interpolation, we can divide any interval into smaller pieces (called cells) by introducing a mesh and then applying quadrature formulas on the individual cells. We denote our mesh by

$$\mathcal{M} := \{a = x_0 < x_1 < \dots < x_{m-1} < x_m = b\},$$

and appeal to the trivial identity

$$\int_a^b f(t) dt = \sum_{j=1}^m \int_{x_{j-1}}^{x_j} f(t) dt. \quad (6.21)$$

On each mesh interval $[x_{j-1}, x_j]$ we then use a *local quadrature rule*, which may be one of the polynomial quadrature formulas from Section 6.2.

General construction of composite quadrature rules

Idea: • Partition integration domain $[a, b]$ by a *mesh*/grid

$$\mathcal{M} := \{a = x_0 < x_1 < \dots < x_m = b\}.$$

- On each local subinterval $I_j := [x_{j-1}, x_j]$, $j = 1, \dots, m$, apply an n_j -point quadrature formula from Section 6.2.

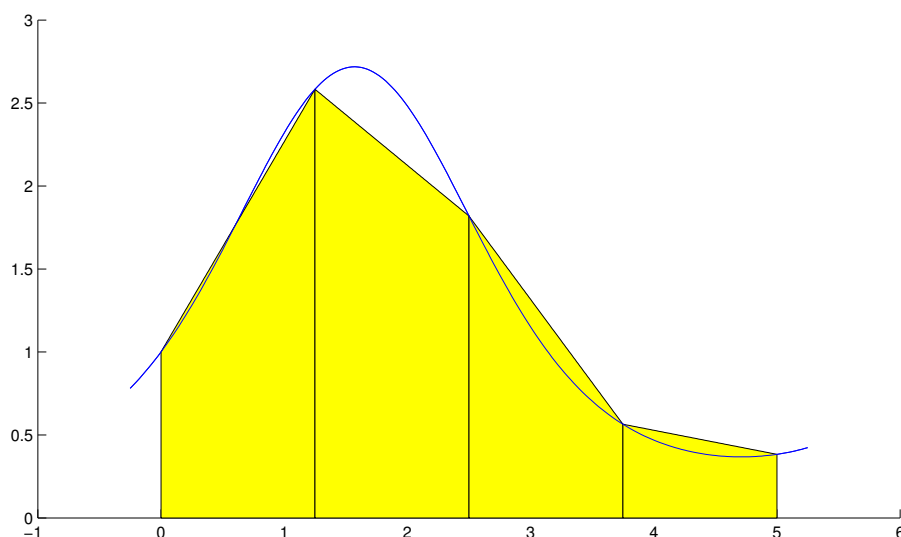
This yields a *composite quadrature rule* with the number of evaluations of f being $\sum_{j=1}^m n_j$.

6.4.1 The Composite trapezoidal and Composite Simpson rule

The Composite trapezoidal rule

The Composite trapezoidal rule, *cf.* (6.9), is given by

$$\int_a^b f(t) dt = \frac{1}{2}(x_1 - x_0)f(a) + \sum_{j=1}^{m-1} \frac{1}{2}(x_{j+1} - x_{j-1})f(x_j) + \frac{1}{2}(x_m - x_{m-1})f(b). \quad (6.22)$$

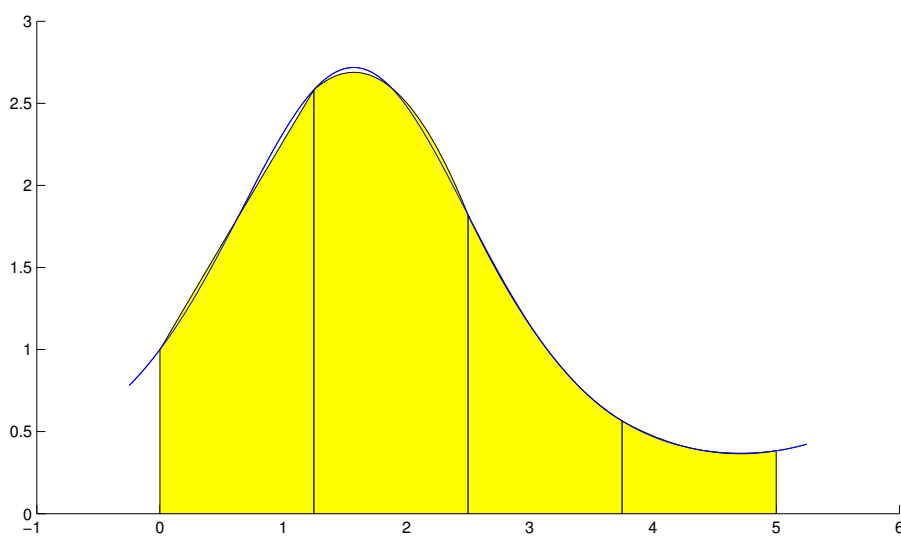


The composite trapezoidal rule. This approximation arises from the piecewise linear interpolation of f .

The composite Simpson rule

The composite Simpson rule, cf. (6.10), is given by

$$\begin{aligned} \int_a^b f(t) dt &= \frac{1}{6}(x_1 - x_0)f(a) + \sum_{j=1}^{m-1} \frac{1}{6}(x_{j+1} - x_{j-1})f(x_j) \\ &\quad + \sum_{j=1}^m \frac{2}{3}(x_j - x_{j-1})f\left(\frac{1}{2}(x_j + x_{j-1})\right) + \frac{1}{6}(x_m - x_{m-1})f(b). \end{aligned} \quad (6.23)$$



The composite Simpson rule. This approximation is obtained through piecewise quadratic Lagrange interpolation.

6.4.2 Errors and orders

To see the main rationale behind the use of composite quadrature rules, recall Lemma 6.3.3: For a polynomial quadrature rule (6.7) of order q with positive weights and $f \in C^r([a, b])$, the quadrature error shrinks with the $(\min\{r, q\} + 1)$ -st power of the length $|b - a|$ of the integration domain. Hence, applying polynomial quadrature rules to small mesh intervals should lead to a small overall quadrature error.

We next derive the overall quadrature error by adding the errors on each I_j :

Suppose on each I_j , we use a quadrature formula $Q_{n_j}^j$ of order q_j and with positive weights. For $f \in C^r([x_{j-1}, x_j])$ we have:

$$\left| \int_{x_{j-1}}^{x_j} f(t) dt - Q_{n_j}^j(f) \right| \underbrace{\leq}_{\text{Lemma 6.3.3}} C |x_j - x_{j-1}|^{\min\{r, q_j\}+1} \|f^{(\min\{r, q_j\})}\|_{L^\infty([x_{j-1}, x_j])} . \quad (6.24)$$

Here $C > 0$ is independent of f and j . Let us denote $h_j := |x_j - x_{j-1}|$. Therefore, the overall quadrature error can be computed as:

$$\begin{aligned} \left| \sum_{j=1}^m \left(\int_{x_{j-1}}^{x_j} f(t) dt - Q_{n_j}^j(f) \right) \right| &\leq \sum_{j=1}^m \left| \int_{x_{j-1}}^{x_j} f(t) dt - Q_{n_j}^j(f) \right| \\ &\leq C \cdot \sum_{j=1}^m h_j^{\min\{r, q_j\}+1} \cdot \|f^{(\min\{r, q_j\})}\|_{L^\infty(I_j)} . \end{aligned}$$

If $q_j = q$, $q \in \mathbb{N}$, for all $j = 1, \dots, m$, then, since $\sum_j h_j = b - a$, one obtains

$$\begin{aligned} \left| \int_{x_0}^{x_m} f(t) dt - Q(f) \right| &\leq C h_{\mathcal{M}}^{\min\{q, r\}} |b - a| \max_{j=1, \dots, m} \|f^{(\min\{q, r\})}\|_{L^\infty(I_j)} , \\ &\leq C h_{\mathcal{M}}^{\min\{q, r\}} |b - a| \|f^{(\min\{q, r\})}\|_{L^\infty([a, b])} , \end{aligned} \quad (6.25)$$

with (global) meshwidth $h_{\mathcal{M}} := \max_j h_j$.

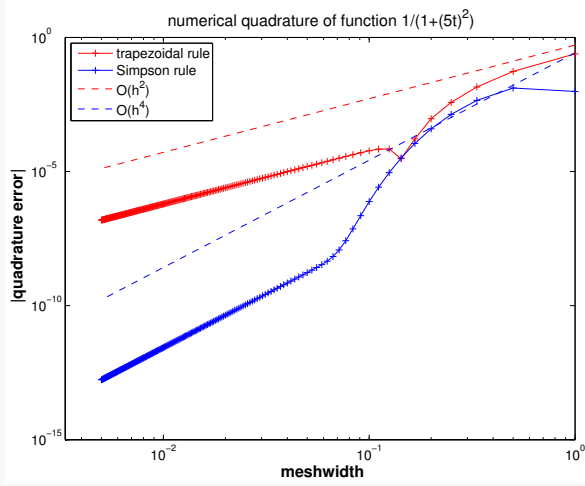
So we have shown that the composite quadrature rule previously described converges algebraically in the mesh width $h_{\mathcal{M}}$ (h -convergence). Note that when f is smooth – i.e. r is large – we obtain algebraic convergence in $h_{\mathcal{M}}$ of rate q .

Example 6.4.1: Composite quadrature formula for a smooth and a non-smooth function

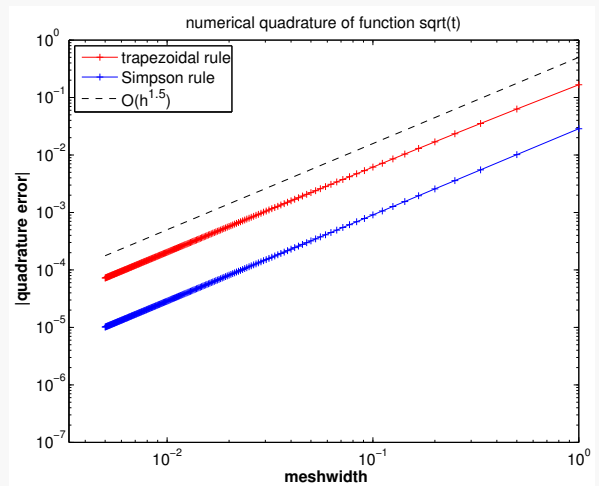
We consider the quadrature error for the composite trapezoidal and Simpson rule for the two functions from Example 6.3.1. Recall the order of the two methods:

- Composite trapezoidal rule (6.22): local order $q = 2$.

- Composite Simpson rule (6.23): local order $q = 4$.



Quadrature error, $f_1(t) := \frac{1}{1+(5t)^2}$ on $[0,1]$



Quadrature error, $f_2(t) := \sqrt{t}$ on $[0,1]$

For the smooth function, f_1 , the quadrature error decays indeed as $\mathcal{O}(h^2)$ for the composite trapezoidal rule and as $\mathcal{O}(h^4)$ for the composite Simpson rule.

For the non-smooth function, f_2 , the quadrature error decays like $\mathcal{O}(h^{3/2})$ for both the composite trapezoidal and the composite Simpson rule.

Remark: A comparison to Gauss quadrature

The asymptotic rates of composite quadrature formulas and the global Gauss quadrature formula compare as follows:

- For $f \in C^r([a, b])$:
 - Composite quadrature formula (with local order q): $\mathcal{O}(n^{-\min\{r, q\}})$
 - Gauss quadrature formula : $\mathcal{O}(n^{-r})$

Thus, Gauss is at least as good as the composite quadrature formula and achieves the best possible rate.

- For $f \in C^\infty([a, b])$, we obtain:
 - Composite quadrature formula (with local order q): $\mathcal{O}(n^{-q})$
 - Gauss quadrature formula : $\mathcal{O}(\lambda^n)$, $\lambda \in (0, 1)$

In case of C^∞ functions, the global Gauss quadrature formula converges exponentially whereas composite quadrature formulas converge algebraically. Hence, the Gauss quadrature formula outperforms the composite quadrature formulas. Note, however, that the Gauss quadrature formula is based on a specific choice of nodes. Depending on the application, we might or might not be able to choose the nodes for the quadrature formula. Thus, the composite quadrature formulas become specifically important if the node set cannot be chosen.

Numerical Methods for ODEs

7.1 Initial value problems for ordinary differential equations

Many problems in engineering and science can be formulated in terms of differential equations. Equations involving derivatives of only one independent variable are called ordinary differential equations (ODEs) and may be classified as either initial value problems (IVP) or boundary value problems (BVP). Loosely speaking, an initial value problem has all of the conditions specified at the same value (*initial*) of the independent variable in the equation while a boundary value problem has conditions specified at the extremes of the independent variable. This chapter will only cover methods for solving initial value problems of ordinary differential equations.

7.1.1 Terminology and notations related to ODEs

In our parlance, a (*first-order*) *ordinary differential equation* is an equation of the form


$$\dot{\mathbf{y}} = \mathbf{f}(t, \mathbf{y}) , \quad (7.1)$$

with a (continuous) *right hand side function* $f : I \times D \rightarrow \mathbb{R}^d$ of time $t \in \mathbb{R}$ and *state* $\mathbf{y} \in \mathbb{R}^d$, defined in the (finite) time interval $I \subset \mathbb{R}$ and on the *state space* $D \subset \mathbb{R}^d$.

In the context of mathematical modelling, the state vector $\mathbf{y} \in \mathbb{R}^d$ is supposed to provide a complete (in the sense of the model) description of a system. In that case, (7.1) models a *finite-dimensional dynamical system*.

For $d > 1$, $\dot{\mathbf{y}} = \mathbf{f}(t, \mathbf{y})$ can be viewed as a *system* of ordinary differential equations:

$$\dot{\mathbf{y}} = \mathbf{f}(t, \mathbf{y}) \iff \begin{bmatrix} \dot{y}_1 \\ \vdots \\ \dot{y}_d \end{bmatrix} = \begin{bmatrix} f_1(t, y_1, \dots, y_d) \\ \vdots \\ f_d(t, y_1, \dots, y_d) \end{bmatrix}.$$

 Notation: $\dot{y} \triangleq$ (total) derivative of y with respect to time t .

Definition 7.1.1 (Solution of an ordinary differential equation). A *solution* of the ODE $\dot{\mathbf{y}} = \mathbf{f}(t, \mathbf{y})$ with the continuous right hand side function \mathbf{f} , is a continuously differentiable function $\mathbf{y} : J \subset I \rightarrow D$, defined on an open interval J , for which $\dot{\mathbf{y}}(t) = \mathbf{f}(t, \mathbf{y}(t))$ holds for all $t \in J$.

A solution describes a continuous *trajectory* in state space or a one-parameter family of states, parameterized by time. It goes without saying that smoothness of the right hand side function \mathbf{f} is inherited by solutions of the ODE:

Lemma 7.1.1 (Smoothness of solutions of ODEs). Let $\mathbf{y} : I \subset \mathbb{R} \rightarrow D$ be a solution of the ODE $\dot{\mathbf{y}} = \mathbf{f}(t, \mathbf{y})$ on the time interval I . If $\mathbf{f} : I \times D \rightarrow \mathbb{R}^d$ is r -times continuously differentiable with respect to both arguments, $r \in \mathbb{N}_0$, then, the trajectory $t \mapsto \mathbf{y}(t)$ is $r + 1$ -times continuously differentiable in the interior of the time interval I .

Definition 7.1.2 (Autonomous ODE). An ODE of the form $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$, with a right hand side function that does not depend on time, but only on the state, is called *autonomous*.

For an autonomous ODE the right hand side function defines a vector field (“velocity field”) $\mathbf{y} \mapsto \mathbf{f}(\mathbf{y})$ on state space.

7.1.2 Modeling with ordinary differential equations: Examples

Example 7.1.1: Growth with limited resources

This is an example from population dynamics with a one-dimensional state space $D = \mathbb{R}_0^+$, $d = 1$, and $y : [0, T] \mapsto \mathbb{R}$ is the bacterial population density as a function of time.

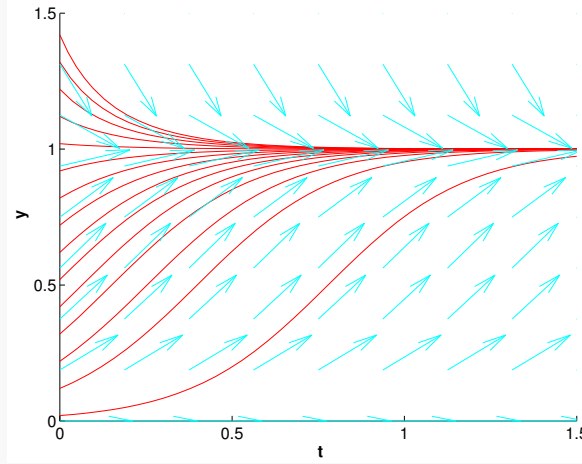
The ODE-based model is an autonomous, logistic differential equation:

$$\dot{y} = f(y) := (\alpha - \beta y) y, \tag{7.2}$$

where

- y is the population density, $[y] = \frac{1}{\text{m}^2}$. Therefore, \dot{y} is the instantaneous change (growth/decay) of population density.

- $\alpha - \beta y$ is the growth rate with coefficients $\alpha, \beta > 0$, $[\alpha] = \frac{1}{s}$, $[\beta] = \frac{m^2}{s}$. It decreases due to more fierce competition as the population density increases.


 Solution for different $y(0)$ ($\alpha, \beta = 5$)

By separation of variables, we can compute a *family of solutions* of (7.2), parameterized by the *initial value* $y(0) = y_0 > 0$:

$$y(t) = \frac{\alpha y_0}{\beta y_0 + (\alpha - \beta y_0) \exp(-\alpha t)}, \quad (7.3)$$

for all $t \in \mathbb{R}$.

Note. $f(y^*) = 0$ for $y^* \in \{0, \frac{\alpha}{\beta}\}$, which are the *stationary points* for the ODE given by (7.2). If $y(0) = y^*$ the solution will be constant in time. Also note that by fixing the initial value $y(0)$ we can single out a *unique* representative from the family of solutions. This will turn out to be a general principle, see Section 7.1.3.

Example 7.1.2: Predator-prey model

Predators and prey coexist in an ecosystem. Without predators, the population of prey would be governed by a simple exponential growth law. However, the growth rate of prey will decrease with increasing numbers of predators and, eventually, become negative. Similar considerations apply to the predator population and lead to an ODE model, referred to as the autonomous *Lotka-Volterra ODE*:

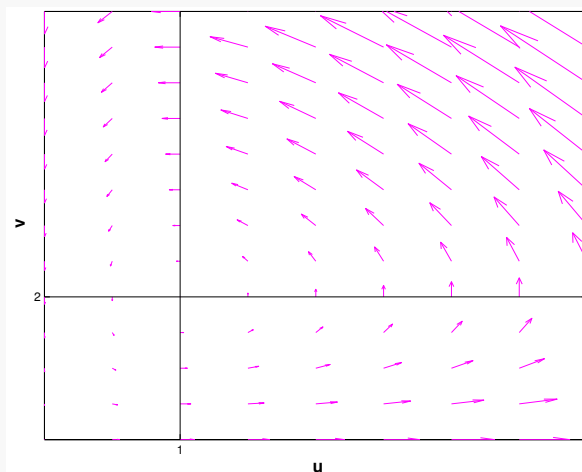
$$\begin{aligned} \dot{u} &= (\alpha - \beta v)u \\ \dot{v} &= (\delta u - \gamma)v \end{aligned}$$

This is equivalent to:

$$\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y}) \quad \text{with} \quad \mathbf{y} = \begin{bmatrix} u \\ v \end{bmatrix}, \quad \mathbf{f}(\mathbf{y}) = \begin{bmatrix} (\alpha - \beta v)u \\ (\delta u - \gamma)v \end{bmatrix}, \quad (7.4)$$

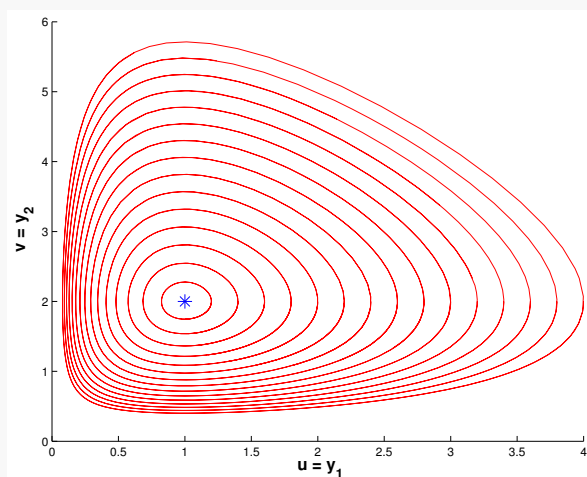
with positive model parameters $\alpha, \beta, \gamma, \delta > 0$.

The population densities are $u(t)$ and $v(t)$, which correspond to the the density of the prey and the predator, respectively, at time t

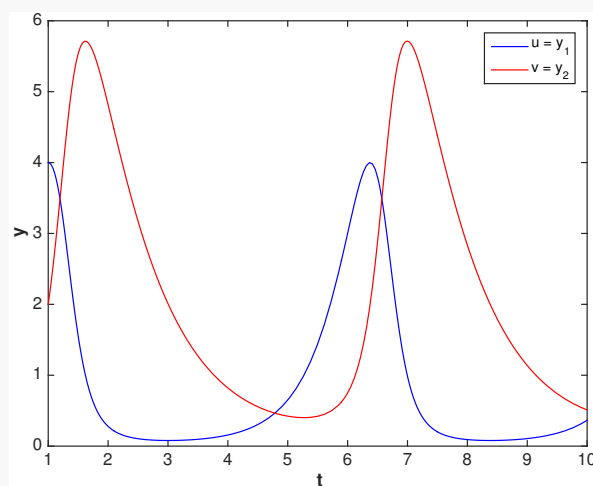


The right hand side vector field \mathbf{f} of (7.4) with the parameter values $\alpha = 2, \beta = 1, \delta = 1, \gamma = 1$

Solution curves are trajectories of particles carried along the velocity field \mathbf{f} .



Solution curves for (7.4), with the stationary point at $*$.



$$\text{Solution} \begin{bmatrix} u(t) \\ v(t) \end{bmatrix} \text{ for } \mathbf{y}_0 := \begin{bmatrix} u(0) \\ v(0) \end{bmatrix} = \begin{bmatrix} 4 \\ 2 \end{bmatrix}$$

The solution depends on the initial size of the population $u(0), v(0)$.

As seen in the previous example, we need additional conditions to solve an ODE uniquely. An initial value problem (IVP) consists of an ODE and some conditions at the initial time t_0 (in the above example, $t_0 = 0$).

7.1.3 Initial value problems

A generic *initial value problem* for a *first-order system of ordinary differential equations* can be stated as:

Find a function $\mathbf{y} : I \rightarrow D$ that satisfies, cf. Definition 7.1.1,

$$\dot{\mathbf{y}} = \mathbf{f}(t, \mathbf{y}) \quad , \quad \mathbf{y}(t_0) = \mathbf{y}_0 \quad , \quad (7.5)$$

where

- $\mathbf{f} : I \times D \mapsto \mathbb{R}^d$ is the *right hand side* (r.h.s.) ($d \in \mathbb{N}$),
- $I \subset \mathbb{R}$ is the (time) interval with the “time variable” t ,
- $D \subset \mathbb{R}^d$ is the *state space/phase space* with the “state variable” \mathbf{y} ,
- $\Omega := I \times D$ is the *extended state space* (of tuples (t, \mathbf{y})),
- and the initial conditions are given by $t_0 \in I$, the initial time, and the initial state, $\mathbf{y}_0 \in D$.

IVPs for autonomous ODEs

Recall Definition 7.1.2: For an autonomous ODE $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$, the right hand side \mathbf{f} does not depend on time t .

Hence, for autonomous ODEs, we have $I = \mathbb{R}$ and the right hand side function $\mathbf{y} \mapsto \mathbf{f}(\mathbf{y})$ can be regarded as a stationary vector field (velocity field).

Note. If $t \mapsto \mathbf{y}(t)$ is a solution of an autonomous ODE, then, for any $\tau \in \mathbb{R}$, the shifted function $t \mapsto \mathbf{y}(t - \tau)$ is also a solution. For initial value problems for autonomous ODEs, the initial time is irrelevant and therefore we can always make the canonical choice $t_0 = 0$.

Autonomous ODEs naturally arise when modeling time-invariant systems or phenomena. The examples from Section 7.1.2 belong to this class.

Autonomization: Conversion into autonomous ODE

In fact, autonomous ODEs already represent the general case because every ODE can be converted into an autonomous one:


Idea: Include time as an extra $d + 1$ -th component of an extended state vector, i.e. introduce a component $y_0(t) = t$. This solution component has to grow linearly, that is, the temporal derivative $\dot{y}_0(t) = 1$.

$$\underbrace{\begin{bmatrix} \dot{y}_1 \\ \vdots \\ \dot{y}_d \end{bmatrix} = \begin{bmatrix} f_1(t, y_1, \dots, y_d) \\ \vdots \\ f_d(t, y_1, \dots, y_d) \end{bmatrix}}_{\text{Non-autonomous system}} \iff \underbrace{\begin{bmatrix} \dot{y}_0 \\ \dot{y}_1 \\ \vdots \\ \dot{y}_d \end{bmatrix} = \begin{bmatrix} 1 \\ f_1(y_0, y_1, \dots, y_d) \\ \vdots \\ f_d(y_0, y_1, \dots, y_d) \end{bmatrix}}_{\text{Autonomous system}}$$

From higher order ODEs to first order systems

An ordinary differential equation of order $n \in \mathbb{N}$ has the form

$$\mathbf{y}^{(n)} = \mathbf{f}(t, \mathbf{y}, \dot{\mathbf{y}}, \dots, \mathbf{y}^{(n-1)}) . \quad (7.6)$$

 Notation: $y^{(n)} \triangleq n$ -th temporal derivative: $\frac{d^n}{dt^n}$

No special treatment of higher order ODEs is necessary, because (7.6) can be turned into a 1st-order ODE (a system of size nd) by adding all derivatives up to order $n - 1$ as additional components to the state vector. This extended state vector $\mathbf{z}(t) \in \mathbb{R}^{nd}$ is defined as

$$\mathbf{z}(t) := \begin{bmatrix} \mathbf{y}(t) \\ \mathbf{y}^{(1)}(t) \\ \vdots \\ \mathbf{y}^{(n-1)}(t) \end{bmatrix} = \begin{bmatrix} \mathbf{z}_1 \\ \mathbf{z}_2 \\ \vdots \\ \mathbf{z}_n \end{bmatrix} \in \mathbb{R}^{nd} . \quad (7.7)$$

Equation (7.6) can be rewritten as:

$$\dot{\mathbf{z}} = \mathbf{g}(\mathbf{z}) , \quad \mathbf{g}(\mathbf{z}) := \begin{bmatrix} \mathbf{z}_2 \\ \mathbf{z}_3 \\ \vdots \\ \mathbf{z}_n \\ \mathbf{f}(t, \mathbf{z}_1, \dots, \mathbf{z}_n) \end{bmatrix} . \quad (7.8)$$

Note that the extended system requires initial values $\mathbf{y}(t_0), \dot{\mathbf{y}}(t_0), \dots, \mathbf{y}^{(n-1)}(t_0)$. For ODEs of order $n \in \mathbb{N}$, well-posed initial value problems need to specify initial values for \mathbf{y} and its first $n - 1$ derivatives. This yields a total number of $n \cdot d$ initial values.

Remark. Altogether, it suffices to consider autonomous first order IVPs.

Existence and uniqueness of solutions of IVP

Now we review results about existence and uniqueness of solutions of initial value problems for first-order ODEs. These are surprisingly general and do not impose severe constraints on right hand side functions.

Theorem 7.1.1. *If the right hand side function $\mathbf{f} : \Omega \mapsto \mathbb{R}^d$ is a differentiable function, then for all initial conditions $(t_0, \mathbf{y}_0) \in \Omega$, the IVP*


$$\dot{\mathbf{y}} = \mathbf{f}(t, \mathbf{y}), \quad \mathbf{y}(t_0) = \mathbf{y}_0, \quad (7.9)$$

admits a unique solution $\mathbf{y} \in C^1(J(t_0, \mathbf{y}_0), \mathbb{R}^d)$ with maximal (temporal) domain of definition $J(t_0, \mathbf{y}_0) \subset \mathbb{R}$.

7.1.4 Domain of definition of solutions of IVPs

Note. Solutions of an IVP have an *intrinsic* maximal domain of definition. This domain of definition $J(t_0, \mathbf{y}_0)$ usually depends on (t_0, \mathbf{y}_0) .

Terminology: If $J(t_0, \mathbf{y}_0) = I$, then the solution $\mathbf{y} : I \mapsto \mathbb{R}^d$ is *global*.

 **Notation:** For an autonomous ODE, we always have $t_0 = 0$, and therefore we write $J(\mathbf{y}_0) := J(0, \mathbf{y}_0)$.

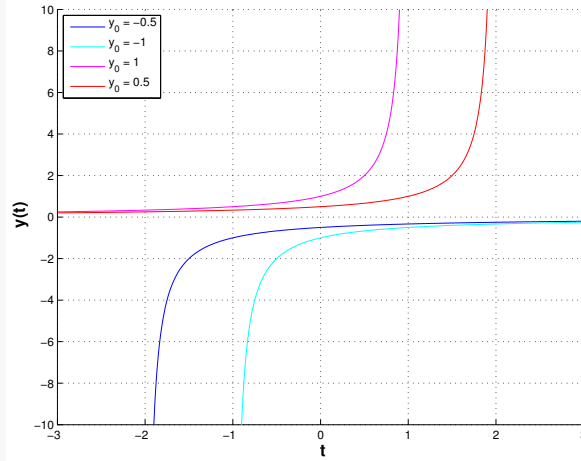
Example 7.1.3: Finite-time blow-up

Let us explain the “maximal domain of definition” in the statement of Theorem 7.1.1. It is related to the fact that every solution of an initial value problem (7.9) has its own largest possible time interval $J(\mathbf{y}_0) \subset \mathbb{R}$ on which it is defined naturally.

As an example, we consider the autonomous scalar ($d = 1$) initial value problem, modeling “explosive growth” with a growth rate increasing linearly with the density:

$$\dot{y} = y^2, \quad y(0) = y_0 \in \mathbb{R}. \quad (7.10)$$

We choose $I = D = \mathbb{R}$.



We find the solutions

$$y(t) = \begin{cases} \frac{1}{y_0^{-1}-t}, & \text{if } y_0 \neq 0, \\ 0, & \text{if } y_0 = 0, \end{cases} \quad (7.11)$$

with domains of definition

$$J(y_0) = \begin{cases} (-\infty, y_0^{-1}), & \text{if } y_0 > 0, \\ \mathbb{R}, & \text{if } y_0 = 0, \\ (y_0^{-1}, \infty), & \text{if } y_0 < 0. \end{cases}$$

In this example, for $y_0 > 0$, the solution experiences a *blow-up* in finite time and ceases to exist afterwards.

7.1.5 Evolution operators

Next, we will introduce the concept of evolution operators. In some sense they allow for a unified treatment of IVPs with the same system of ODEs. Instead of considering different given initial values with the same ODE system as different problems, we incorporate the initial value in the mapping and hence consider all solutions for any choice of initial value simultaneously. For the sake of simplicity, we restrict the discussion to autonomous IVPs (7.9) with a differentiable right-hand side and make the following assumption.

Assumption 7.1.1 (Global solutions). *All solutions of (7.9) are global: $J(y_0) = \mathbb{R}$ for all $y_0 \in D$.*

Now we study a generic ODE (7.1) instead of an IVP (7.5). We do this by temporarily changing the perspective: we fix a “time of interest” $t \in \mathbb{R} \setminus \{0\}$ and follow all trajectories for the duration t . This induces a mapping of points in state space:

$$\text{Mapping } \Phi^t : \begin{cases} D \mapsto D \\ \mathbf{y}_0 \mapsto \mathbf{y}(t) \end{cases}, \quad t \mapsto \mathbf{y}(t) \text{ solution of IVP (7.9).}$$

This is a well-defined mapping of the state space into itself, by Theorem 7.1.1 and Assumption 7.1.1.

Now, we may also let t vary, which spawns a *family* of mappings $\{\Phi^t\}$ of the state space into itself. However, it can also be viewed as a mapping with two arguments, a duration t and an initial state value \mathbf{y}_0 .

Definition 7.1.3 (Evolution operator/mapping). Under Assumption 7.1.1, the mapping

$$\Phi : \begin{cases} \mathbb{R} \times D \mapsto D \\ (t, \mathbf{y}_0) \mapsto \Phi^t \mathbf{y}_0 := \mathbf{y}(t) \end{cases},$$

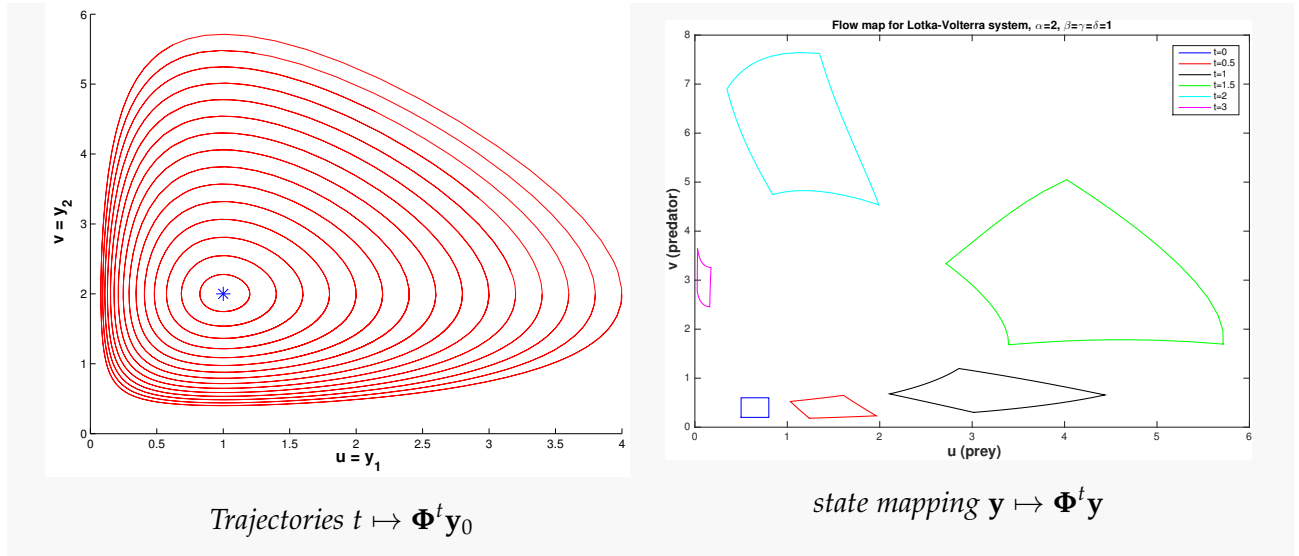
where $t \mapsto \mathbf{y}(t) \in C^1(\mathbb{R}, \mathbb{R}^d)$ is the unique (global) solution of the IVP $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$, $\mathbf{y}(0) = \mathbf{y}_0$, is the *evolution operator* or mapping for the autonomous ODE $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$.

Note. $t \mapsto \Phi^t \mathbf{y}_0$ describes the solution of $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$ for $\mathbf{y}(0) = \mathbf{y}_0$ (a trajectory). Therefore, by virtue of definition, we have

$$\frac{\partial \Phi}{\partial t}(t, \mathbf{y}) = \mathbf{f}(\Phi^t \mathbf{y}). \quad (7.12)$$

Example 7.1.4: Evolution operator for Lotka-Volterra ODE (7.4)

For $d = 2$, the action of an evolution operator can be visualized by tracking the movement of point sets in the state space. Here, this is done for the Lotka-Volterra ODE (7.4):



7.2 Polygonal Approximation Methods

For an initial value problem (7.5) for a first-order ordinary differential equation

$$\dot{\mathbf{y}} = \mathbf{f}(t, \mathbf{y}) \quad , \quad \mathbf{y}(t_0) = \mathbf{y}_0,$$

we want to obtain an approximate model for the evolution operator Φ on a temporal mesh

$$\mathcal{M} := \{t_0 < t_1 < t_2 < \cdots < t_{N-1} < t_N := T\} \subset [t_0, T] , \quad (7.13)$$

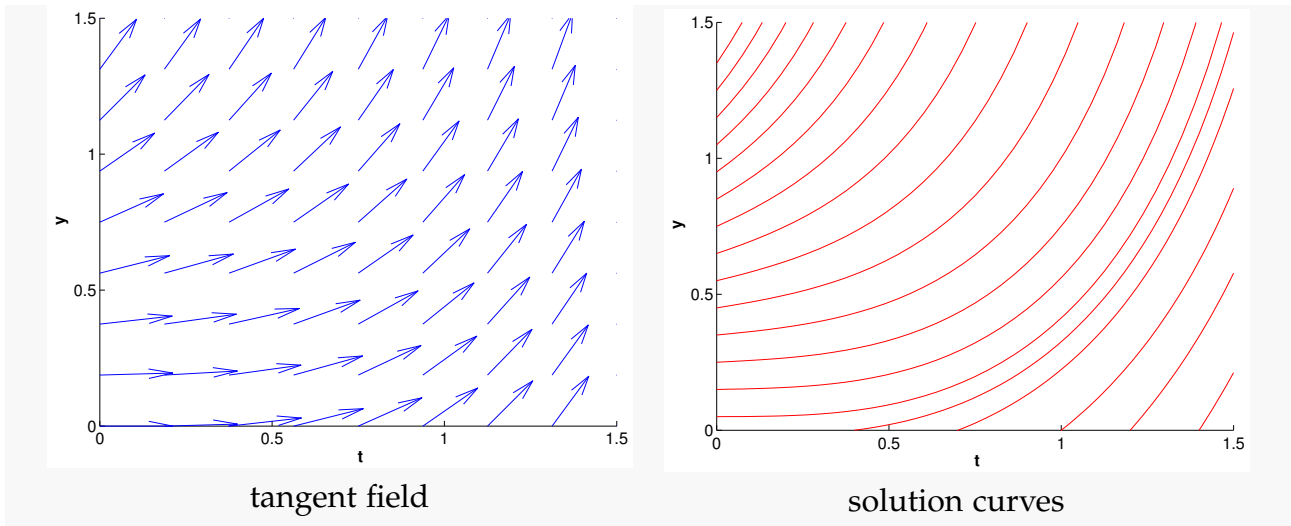
covering the time interval of interest between initial time t_0 and final time $T > t_0$. We assume that the interval of interest is contained in the domain of definition of the solution of the IVP: $[t_0, T] \subset J(t_0, \mathbf{y}_0)$. We start our considerations with the most intuitive methods (through polygonal approximation). In the next section, general methods for approximating the evolution operator will be introduced.

Explicit Euler method

Example 7.2.1: Tangent field and solution curves

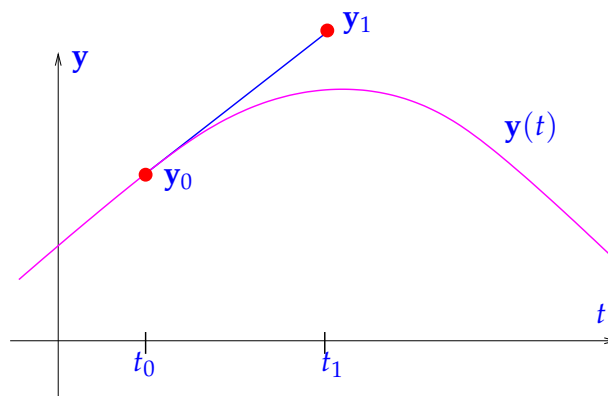
For $d = 1$, polygonal methods can be constructed by geometric considerations in the $t - y$ plane, a model for the extended state space. We explain this for the *Riccati differential equation*, a scalar ODE:

$$\dot{y} = y^2 + t^2 \quad I, D = \mathbb{R}^+ . \quad (7.14)$$



Idea: “Follow the tangents over short periods of time”

- ❶ Timestepping: Perform successive approximation of evolution on *mesh intervals* $[t_{k-1}, t_k]$, $k = 1, \dots, N$, $t_N := T$.
- ❷ Approximate the solution on $[t_{k-1}, t_k]$ by considering a tangent line to the solution trajectory through $(t_{k-1}, \mathbf{y}_{k-1})$. We know that the slope of the tangent at $(t_{k-1}, \mathbf{y}_{k-1})$ is given by $f(t_{k-1}, \mathbf{y}_{k-1})$.



First step of explicit Euler method ($d = 1$)

Example 7.2.2: Visualization of explicit Euler method

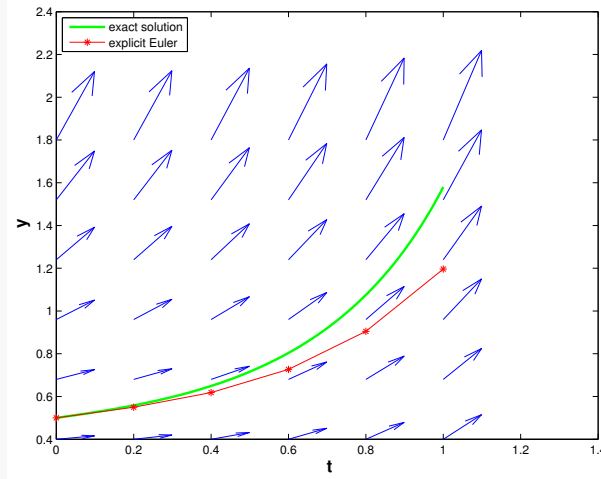
Temporal mesh:

$$\mathcal{M} := \{t_j := \frac{j}{5} : j = 0, \dots, 5\}.$$

IVP for Riccati differential equation (see Example 7.2.1) is given by:

$$\dot{y} = y^2 + t^2. \quad (7.14)$$

Here: $y_0 = \frac{1}{2}, t_0 = 0, T = 1$



— \triangleq “Euler polygon” for uniform timestep $h = 0.2$;

$\rightarrow \triangleq$ Tangent field of Riccati ODE

When applied to a general IVP of the form (7.5), the explicit Euler method generates a sequence $(\mathbf{y}_k)_{k=0}^N$ by the recursion:

$$\mathbf{y}_{k+1} = \mathbf{y}_k + h_k \mathbf{f}(t_k, \mathbf{y}_k), \quad k = 0, \dots, N-1, \quad (7.15)$$

with local (size of) timestep $h_k := t_{k+1} - t_k$.

Note. The explicit Euler method can be viewed as a difference scheme. One can obtain (7.15) by approximating the derivative $\frac{d}{dt}$ by a *forward difference quotient* on the (temporal) mesh $\mathcal{M} := \{t_0, t_1, \dots, t_N\}$:

$$\dot{\mathbf{y}} = \mathbf{f}(t, \mathbf{y}) \quad \longleftrightarrow \quad \underbrace{\frac{\mathbf{y}_{k+1} - \mathbf{y}_k}{h_k}}_{\text{Forward difference quotient}} \approx \mathbf{f}(t_k, \mathbf{y}(t_k)), \quad k = 0, \dots, N-1. \quad (7.16)$$

Difference schemes follow a simple policy for the *discretization* of differential equations: replace all derivatives by difference quotients connecting solution values on a set of discrete points (the mesh).

Implicit Euler method

Now we try to approximate the derivative $\frac{d}{dt}$ by a *backward difference quotient*.

On a (temporal) *mesh* $\mathcal{M} := \{t_0, t_1, \dots, t_N\}$, we obtain

$$\dot{\mathbf{y}} = \mathbf{f}(t, \mathbf{y}) \quad \longleftrightarrow \quad \frac{\mathbf{y}_{k+1} - \mathbf{y}_k}{h_k} \approx \mathbf{f}(t_{k+1}, \mathbf{y}(t_{k+1})), \quad k = 0, \dots, N-1, \quad (7.17)$$

by applying the *backward difference quotient*.

This leads to another simple timestepping scheme analogous to (7.15):

$$\mathbf{y}_{k+1} = \mathbf{y}_k + h_k \mathbf{f}(t_{k+1}, \mathbf{y}_{k+1}), \quad k = 0, \dots, N-1, \quad (7.18)$$

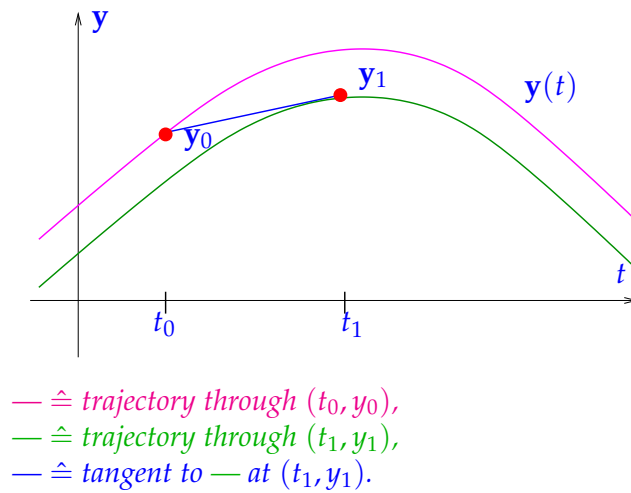
with local (size of) timestep $h_k := t_{k+1} - t_k$. Equation (7.18) is the *implicit Euler method*.

Note. The RHS of (7.18) depends on \mathbf{y}_{k+1} which is not known at step k . This requires solving a (possibly non-linear) system of equations to obtain \mathbf{y}_{k+1} . This is the reason that it is called an “implicit” method.

Geometric interpretation of the *implicit Euler method*:

Approximate solution through (t_0, \mathbf{y}_0) on $[t_0, t_1]$ by

- straight line through (t_0, \mathbf{y}_0)
- with slope $\mathbf{f}(t_1, \mathbf{y}_1)$



Implicit midpoint method

Besides using forward or backward difference quotients, the derivative $\dot{\mathbf{y}}$ can also be approximated by the *symmetric difference quotient*,

$$\dot{\mathbf{y}}(t) \approx \frac{\mathbf{y}(t+h) - \mathbf{y}(t-h)}{2h}. \quad (7.19)$$

The idea is to apply this formula in $t = \frac{1}{2}(t_k + t_{k+1})$, which transforms the ODE into

$$\dot{\mathbf{y}} = f(t, \mathbf{y}) \longleftrightarrow \frac{\mathbf{y}_{k+1} - \mathbf{y}_k}{h_k} \approx \mathbf{f}\left(\frac{1}{2}(t_k + t_{k+1}), \mathbf{y}\left(\frac{1}{2}(t_k + t_{k+1})\right)\right), \quad k = 0, \dots, N-1. \quad (7.20)$$

The issue is that the value $\mathbf{y}(\frac{1}{2}(t_k + t_{k+1}))$ is not available. Therefore we approximate it by $\mathbf{y}(\frac{1}{2}(t_k + t_{k+1})) \approx \frac{1}{2}(\mathbf{y}(t_k) + \mathbf{y}(t_{k+1}))$. This gives the recursion formula for the *implicit midpoint method* in analogy to (7.15) and (7.18):

$$\mathbf{y}_{k+1} = \mathbf{y}_k + h_k \mathbf{f}\left(\frac{1}{2}(t_k + t_{k+1}), \frac{1}{2}(\mathbf{y}_k + \mathbf{y}_{k+1})\right), \quad k = 0, \dots, N-1, \quad (7.21)$$

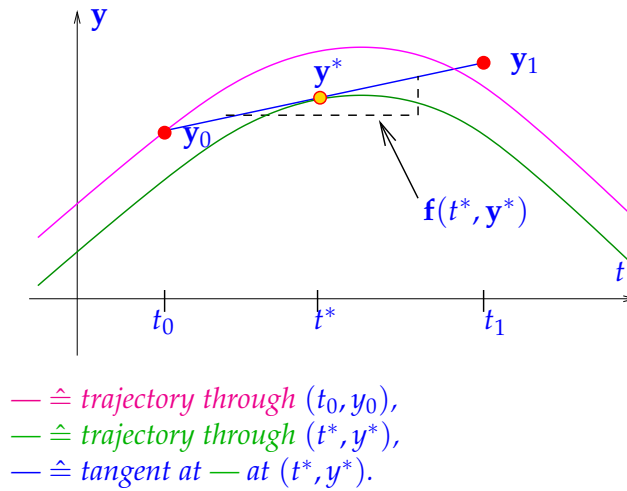
with local (size of) timestep $h_k := t_{k+1} - t_k$.

Geometric interpretation of the *implicit midpoint method*:

Approximate trajectory through (t_0, \mathbf{y}_0) on $[t_0, t_1]$ by

- straight line through (t_0, \mathbf{y}_0)
- with slope $\mathbf{f}(t^*, \mathbf{y}^*)$, where

$$t^* := \frac{1}{2}(t_0 + t_1), \quad \mathbf{y}^* = \frac{1}{2}(\mathbf{y}_0 + \mathbf{y}_1).$$



As in the case of (7.18), (7.21) also entails solving a (non-linear) system of equations in order to obtain \mathbf{y}_{k+1} .

Note. The solutions of the involved non-linear systems in implicit Euler and implicit midpoint method exist, if h is sufficiently small.

7.3 General single step methods

Now we fit the numerical schemes introduced in the previous section into a more general class of methods for the solution of (autonomous) initial value problems (7.9) for ODEs.

Throughout we assume that all times considered belong to the domain of definition of the unique solution $t \mapsto \mathbf{y}(t)$ of (7.9), that is, for $T > 0$ we take for granted that $[0, T] \subset J(\mathbf{y}_0)$.

7.3.1 Discrete evolution operators

Recall the methods we have seen so far for solving the autonomous ODE $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$:

$$\begin{aligned} \text{Explicit Euler: } & \mathbf{y}_{k+1} = \mathbf{y}_k + h_k \mathbf{f}(\mathbf{y}_k) , \\ \text{Implicit Euler: } & \mathbf{y}_{k+1} = \mathbf{y}_k + h_k \mathbf{f}(\mathbf{y}_{k+1}) , \\ \text{Implicit midpoint: } & \mathbf{y}_{k+1} = \mathbf{y}_k + h_k \mathbf{f}\left(\frac{1}{2}(t_k + t_{k+1}), \frac{1}{2}(\mathbf{y}_k + \mathbf{y}_{k+1})\right) . \end{aligned}$$

These formulas, for sufficiently small h , provide a mapping

$$(\mathbf{y}_k, h_k) \mapsto \Psi(h_k, \mathbf{y}_k) := \mathbf{y}_{k+1} . \quad (7.22)$$

If \mathbf{y}_0 is the initial value, then $\mathbf{y}_1 := \Psi(h, \mathbf{y}_0)$ can be regarded as an approximation of $\mathbf{y}(h)$, the value returned by the evolution operator (see Definition 7.1.3) for $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$ applied to \mathbf{y}_0 over the period h .

$$\begin{aligned} \mathbf{y}_1 = \Psi(h_0, \mathbf{y}_0) & \longleftrightarrow \mathbf{y}(t_1) = \Phi^{h_0} \mathbf{y}_0 , \\ \mathbf{y}_{k+1} = \Psi(h_k, \mathbf{y}_k) & \longleftrightarrow \mathbf{y}(t_{k+1}) = \Phi^{h_k} \mathbf{y}_k . \end{aligned}$$

More generally,

$$\Psi(h, \mathbf{y}) \approx \Phi^h \mathbf{y} . \quad (7.23)$$

In a sense, the polygonal approximation methods are based on approximations for the evolution operator associated with the ODE. Every single step method tries to approximate the evolution operator Φ for an ODE by a mapping of the type (7.22).

Note. The mapping Ψ from (7.22) is called the *discrete evolution operator*.

 **Notation:** For discrete evolutions, we often write $\Psi^h \mathbf{y} := \Psi(h, \mathbf{y})$.

Note. The adjective “*discrete*” used above designates (components of) methods that attempt to approximate the solution of an IVP by a sequence of finitely many states. “*Discretization*” is the process of converting an ODE into a discrete model. This parlance is adopted for all procedures that reduce a “continuous model” involving ordinary (or partial) differential equations to a form with a finite number of unknowns.

In the above, we identified the discrete evolutions underlying the polygonal approximation methods. More generally, a mapping Ψ as given in (7.22) defines a single step method.

Definition 7.3.1 (Single step method (for autonomous ODEs)). Given a discrete evolution $\Psi : \Omega \subset \mathbb{R} \times D \rightarrow \mathbb{R}^d$, an initial state \mathbf{y}_0 , and a temporal mesh $\mathcal{M} := \{0 =: t_0 < t_1 < \dots < t_N := T\}$, the recursion

$$\mathbf{y}_{k+1} := \Psi(t_{k+1} - t_k, \mathbf{y}_k), \quad k = 0, \dots, N-1, \quad (7.24)$$

defines a *single step method* (SSM) for the autonomous IVP $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$, $\mathbf{y}(0) = \mathbf{y}_0$ on the interval $[0, T]$.

Note. In a sense, a single step method defined through its associated discrete evolution does not approximate a concrete initial value problem, but tries to approximate an ODE in the form of its evolution operator.

The concept of single step method, according to Definition 7.3.1, can be generalized to non-autonomous ODEs, which leads to recursions of the form:

$$\mathbf{y}_{k+1} := \Psi(t_k, t_{k+1}, \mathbf{y}_k), \quad k = 0, \dots, N-1,$$

for a discrete evolution operator Ψ defined on $I \times I \times D$.

7.3.2 Consistent single step methods

So far, the definition of discrete evolution is not meaningful yet in the sense that it does not necessarily approximate the evolution operator Φ . We introduce the following basic requirement of consistency of the discrete evolution. In a sense it asks that in the limit $h \rightarrow 0$, the update direction of the discrete evolution should be in the direction of the derivative of \mathbf{y} , which is equal to $\mathbf{f}(\mathbf{y})$. In view of Equation (7.12), this is precisely the property that we would want in order to assure that the discrete evolution Ψ is a good approximation of Φ for h sufficiently small.

Consistent discrete evolution

The discrete evolution Ψ defining a single step method according to Definition 7.3.1 and (7.24) for the autonomous ODE $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$ invariably is of the form

$$\Psi^h \mathbf{y} = \mathbf{y} + h\psi(h, \mathbf{y}) \quad \text{with} \quad \begin{aligned} &\psi : I \times D \rightarrow \mathbb{R}^d \text{ continuous,} \\ &\psi(0, \mathbf{y}) = \mathbf{f}(\mathbf{y}). \end{aligned} \quad (7.25)$$

Definition 7.3.2 (Consistent single step methods). A single step method, according to Definition 7.3.1, based on a discrete evolution of the form (7.25) is called *consistent* with the ODE $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$.

Example 7.3.1: Consistency of implicit midpoint method

The discrete evolution Ψ and, hence, the function $\psi = \psi(h, \mathbf{y})$ for the implicit midpoint method are defined only implicitly, of course. Thus, consistency cannot immediately be seen from a formula for ψ .

We examine consistency of the implicit midpoint method, as defined by (7.21):

$$\mathbf{y}_{k+1} = \mathbf{y}_k + hf \left(\frac{1}{2}(t_k + t_{k+1}), \frac{1}{2}(\mathbf{y}_k + \mathbf{y}_{k+1}) \right), \quad k = 0, \dots, N-1.$$

Assume that

- the right hand side function \mathbf{f} is smooth, at least $\mathbf{f} \in C^1(D)$,
- and $|h|$ is sufficiently small to guarantee the existence of a solution \mathbf{y}_{k+1} of (7.21).

First, note that

$$\Psi^0 \mathbf{y} = \mathbf{y}.$$

Next, expressing the implicit midpoint method as

$$\Psi^h \mathbf{y} = \mathbf{y} + hf \left(\frac{1}{2}(\mathbf{y} + \Psi^h \mathbf{y}) \right),$$

one obtains

$$\psi(h, \mathbf{y}) = \mathbf{f} \left(\frac{1}{2}(\mathbf{y} + \Psi^h \mathbf{y}) \right).$$

Thus,

$$\psi(0, \mathbf{y}) = \mathbf{f}(\mathbf{y}),$$

i.e. consistency holds.

Note. In the literature, a single step method is often specified by writing down the first step for a general stepsize h , i.e.

$$\mathbf{y}_1 = \text{expression in } \mathbf{y}_0, h \text{ and } \mathbf{f}.$$

Actually, this fixes the underlying discrete evolution. This course will also adopt this practice sometimes.

7.3.3 Convergence of single step methods

Discretization error of single step methods

Errors in numerical integration are called *discretization errors*, cf. Section 7.3.1. Depending on the objective of numerical integration, different notions of discretization error are appropriate.

- (I) If only the solution at final time is sought, the discretization error is

$$\epsilon_N := \|\mathbf{y}(T) - \mathbf{y}_N\| ,$$

where $\|\cdot\|$ is some vector norm on \mathbb{R}^d .

- (II) If we want to approximate the solution trajectory for (7.9), the discretization error is the function

$$t \mapsto \mathbf{e}(t) \quad , \quad \mathbf{e}(t) := \mathbf{y}(t) - \mathbf{y}_h(t) ,$$

where $t \mapsto \mathbf{y}_h(t)$ is the approximate trajectory obtained by post-processing. In this case accuracy of the method is gauged by looking at norms of the function \mathbf{e} .

- (III) A compromise between Item (I) and Item (II) would be the pointwise discretization error, which is the sequence (grid function)

$$\mathbf{e} : \mathcal{M} \rightarrow D \quad , \quad \mathbf{e}_k := \mathbf{y}(t_k) - \mathbf{y}_k \quad , \quad k = 0, \dots, N. \quad (7.26)$$

In this case, we may consider the maximum error in the mesh points

$$\|(\mathbf{e})\|_\infty := \max_{k \in \{1, \dots, N\}} \|\mathbf{e}_k\| ,$$

where $\|\cdot\|$ is a suitable vector norm on \mathbb{R}^d , usually the Euclidean vector norm.

Asymptotic convergence of single step methods

Once the discrete evolution Ψ associated with the ODE $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$ is specified, the single step method according to Definition 7.3.1 is fixed. The only way to control the accuracy of the solution \mathbf{y}_N or $t \mapsto \mathbf{y}_h(t)$ is through the selection of the mesh $\mathcal{M} = \{0 = t_0 < t_1 < \dots < t_N = T\}$.

Hence, we study convergence of single step methods for families of meshes $\{\mathcal{M}_\ell\}_{\ell \in \mathbb{N}}$ and track the decay of (a norm) of the discretization error (see Section 7.3.3) as a function of the number of mesh points $N := \sharp \mathcal{M}$. In other words, we examine *h-convergence*. We already did this in the case of piecewise polynomial interpolation in Section 5.3 and composite numerical quadrature in Section 6.4.

When investigating asymptotic convergence of single step methods, we often resort to families of *equidistant* meshes of $[0, T]$:

$$\mathcal{M}_N := \{t_k := \frac{k}{N}T : k = 0, \dots, N\} . \quad (7.27)$$

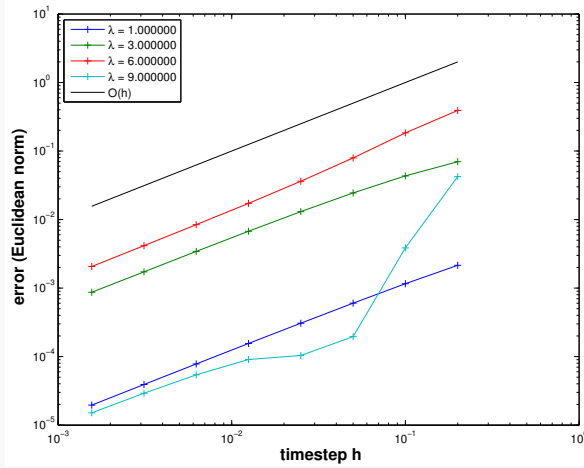
We refer to this as the use of *uniform* timesteps of size $h := \frac{T}{N}$.

Example 7.3.2: Speed of convergence of Euler and implicit midpoint methods

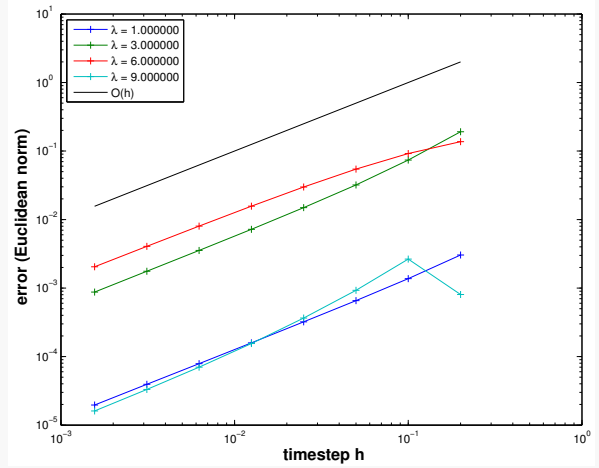
We consider the following IVP for the logistic ODE (see Example 7.1.1):

$$\dot{y} = \lambda y(1 - y), \quad y(0) = 0.01.$$

We apply the explicit Euler (7.15) and implicit Euler (7.18) methods with uniform timestep $h = 1/N$, $N \in \{5, 10, 20, 40, 80, 160, 320, 640\}$ and monitor the error at final time: $|e_N(h)| = |y(1) - y_N|$.



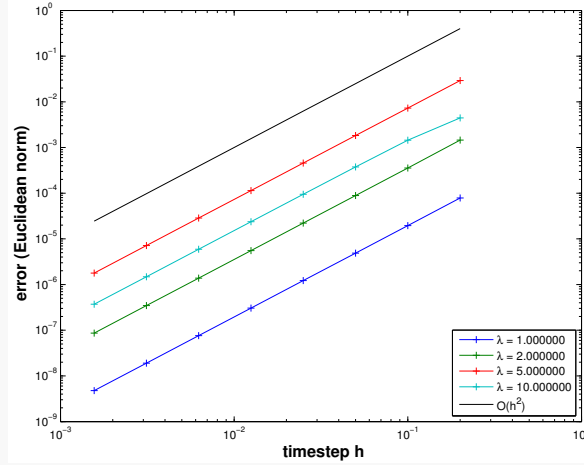
Explicit Euler method



Implicit Euler method

Note. Algebraic convergence of $\mathcal{O}(N^{-1}) = \mathcal{O}(h)$ is observed in both cases as $h \rightarrow 0$.

However, polygonal approximation methods can do better:



Implicit midpoint method

Note. For the implicit midpoint method, we observe algebraic convergence $\mathcal{O}(h^2)$ as $h \rightarrow 0$.

Parlance: Based on the observed rate of algebraic convergence, the two Euler methods are said to “converge with first order”, whereas the implicit midpoint method is called “second-order convergent”.

The observations made for polygonal timestepping methods reflect a general pattern:

Algebraic convergence of single step methods

Consider numerical integration of an initial value problem

$$\dot{\mathbf{y}} = \mathbf{f}(t, \mathbf{y}) \quad , \quad \mathbf{y}(t_0) = \mathbf{y}_0 \quad ,$$

with sufficiently smooth right hand side function $\mathbf{f} : I \times D \rightarrow \mathbb{R}^d$.

Then, customary single step methods (see Definition 7.3.1) will enjoy *algebraic convergence in the meshwidth*. More precisely (see [7, Thm. 11.25]), there is a $p \in \mathbb{N}$ such that the sequence $(\mathbf{y}_k)_k$ generated by the single step method for $\dot{\mathbf{y}} = \mathbf{f}(t, \mathbf{y})$ on a mesh $\mathcal{M} := \{t_0 < t_1 < \dots < t_N = T\}$ satisfies

$$\max_k \|\mathbf{y}_k - \mathbf{y}(t_k)\| \leq Ch^p \quad \text{for} \quad h := \max_{k=1, \dots, N} |t_k - t_{k-1}| \rightarrow 0 \quad , \quad (7.28)$$

with $C > 0$ independent of \mathcal{M} .

Definition 7.3.3 (Order of a single step method). The maximal integer $p \in \mathbb{N}$ for which (7.28) holds for a single step method when applied to an ODE with (sufficiently) smooth right hand side, is called the *order* of the method.

As in the case of quadrature rules (see Definition 6.3.1), their order is the principal intrinsic indicator for the “quality” of a single step method.

Convergence analysis for the explicit Euler method

In Example 7.3.2, we empirically derived the order of the Euler methods and the implicit midpoint method. In what follows, we derive this algebraic convergence for the explicit Euler method. Recall the pointwise discretization error defined in (7.26) and rewrite it using the evolution and the discrete evolution operator:

$$\begin{aligned} \mathbf{e}_{k+1} &:= \mathbf{y}(t_{k+1}) - \mathbf{y}_{k+1}, \quad k = 0, \dots, N-1. \\ &= \Psi^h \mathbf{y}_k - \Phi^h(\mathbf{y}(t_k)). \end{aligned}$$

We would now like to perform a convergence analysis for this error.

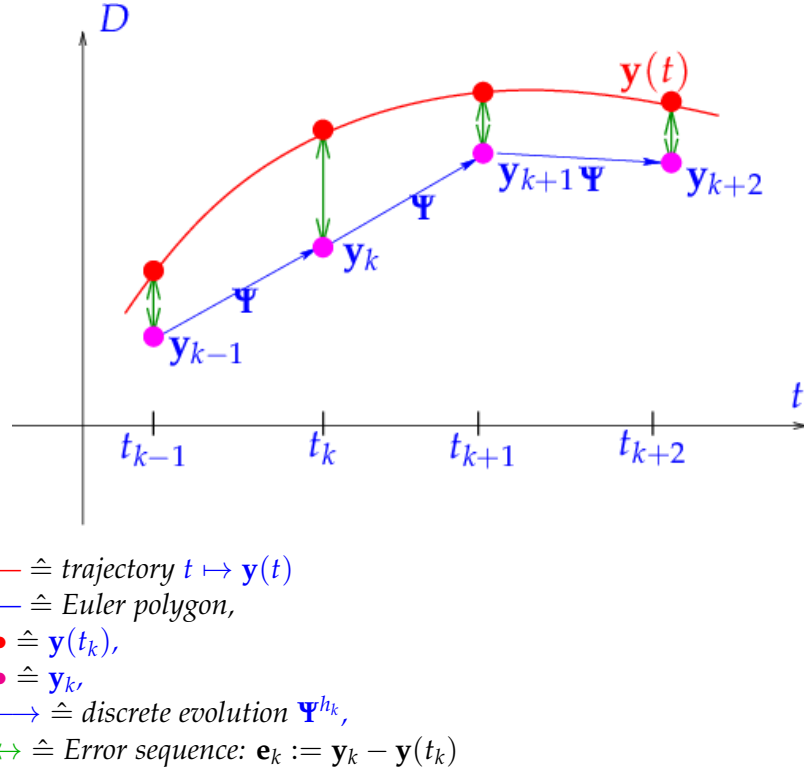
We consider the explicit Euler method (7.15) on a mesh $\mathcal{M} := \{0 = t_0 < t_1 < \dots < t_N = T\}$ for a generic autonomous IVP (7.5) with sufficiently smooth and (*globally*) Lipschitz continuous \mathbf{f} , that is,

$$\exists L > 0: \quad \|\mathbf{f}(\mathbf{y}) - \mathbf{f}(\mathbf{z})\| \leq L \|\mathbf{y} - \mathbf{z}\| \quad \forall \mathbf{y}, \mathbf{z} \in D, \quad (7.29)$$

and the exact solution $t \mapsto \mathbf{y}(t)$. We assume that solutions of $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$ are defined on $[0, T]$ for all initial states $\mathbf{y}_0 \in D$.

Recall the recursion for the explicit Euler method:

$$\mathbf{y}_{k+1} = \mathbf{y}_k + h_k \mathbf{f}(\mathbf{y}_k), \quad k = 1, \dots, N-1.$$



Abstract splitting of error:

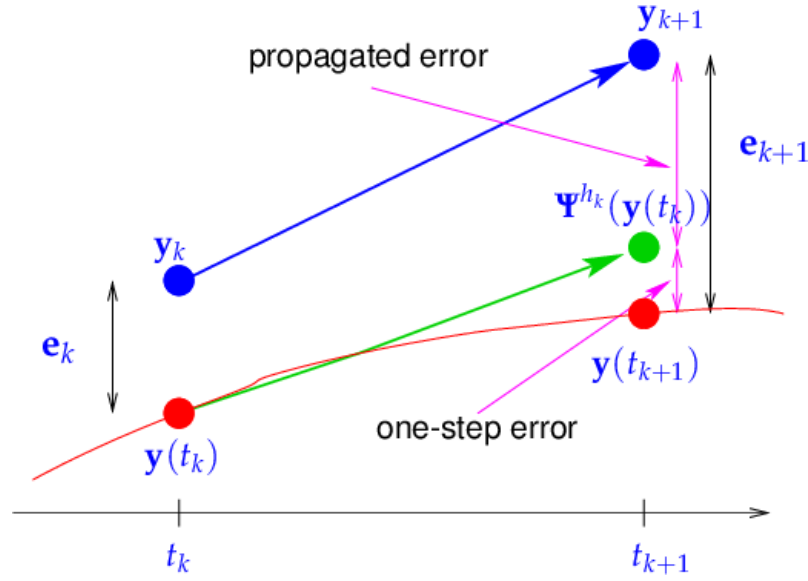
Here and in what follows we rely on the abstract concepts of the evolution operator Φ associated with the ODE $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$ (see Definition 7.1.3) and discrete evolution operator Ψ (see Definition 7.3.1) defining the explicit Euler method (see Equation (7.15)). The discrete evolution operator for the explicit Euler method may be written as:

$$\Psi^h \mathbf{y} = \mathbf{y} + h\mathbf{f}(\mathbf{y}) . \quad (7.30)$$

We argue that, in this context, the abstraction pays off because it helps elucidate a general technique for the convergence analysis of single step methods.

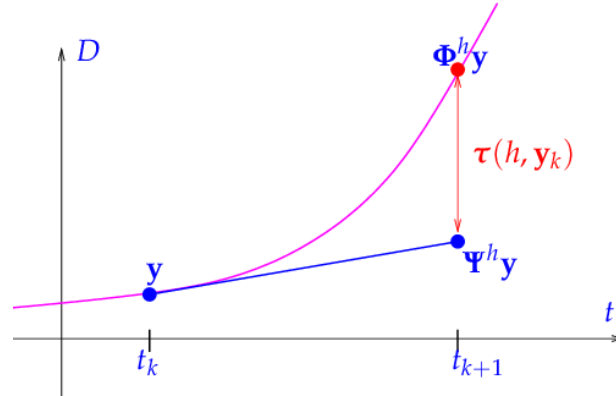
Fundamental error splitting:

$$\begin{aligned}
 \mathbf{e}_{k+1} &= \Psi^{h_k} \mathbf{y}_k - \Phi^{h_k} \mathbf{y}(t_k) \\
 &= \underbrace{\Psi^{h_k} \mathbf{y}_k - \Psi^{h_k} \mathbf{y}(t_k)}_{\text{propagated error (PE)}} + \underbrace{\Psi^{h_k} \mathbf{y}(t_k) - \Phi^{h_k} \mathbf{y}(t_k)}_{\text{one-step error (OE)}} .
 \end{aligned} \quad (7.31)$$



To give an upper bound on $\|\mathbf{e}_{k+1}\|$, it suffices to bound the propagated error and the one-step error separately. A generic *one-step error* can be expressed through continuous and discrete evolutions:

$$\tau(h, \mathbf{y}) := \mathbf{\Psi}^h \mathbf{y} - \mathbf{\Phi}^h \mathbf{y} . \quad (7.32)$$

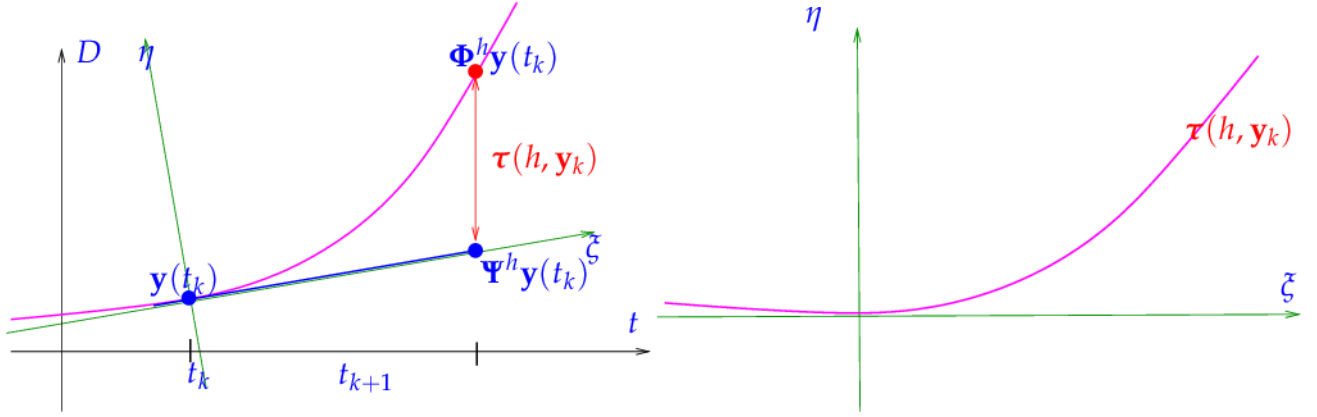


Geometric visualisation of the one-step error for the explicit Euler method

— : solution trajectory through (t_k, \mathbf{y})

① **Estimate for one-step error $\tau(h_k, \mathbf{y}(t_k))$:**

From geometric considerations, we can show that the distance of a smooth curve to its tangent shrinks as the square of the distance to the intersection point (i.e., the curve locally looks like a parabola in the $\xi - \eta$ coordinate system).



If $\mathbf{y} \in C^2([0, T])$, which is ensured for smooth \mathbf{f} (see Lemma 7.1.1), then the geometric considerations can be made rigorous by using Taylor's formula:

$$\mathbf{y}(t_{k+1}) - \mathbf{y}(t_k) = \dot{\mathbf{y}}(t_k)h_k + \frac{1}{2}\ddot{\mathbf{y}}(\xi_k)h_k^2 = \mathbf{f}(\mathbf{y}(t_k))h_k + \frac{1}{2}\ddot{\mathbf{y}}(\xi_k)h_k^2,$$

for some $t_k \leq \xi_k \leq t_{k+1}$. This leads to an expression for the one-step error from (7.32):

$$\begin{aligned} \tau(h_k, \mathbf{y}(t_k)) &= \Psi^{h_k} \mathbf{y}(t_k) - \mathbf{y}(t_{k+1}) \\ &\stackrel{(7.30)}{=} \mathbf{y}(t_k) + h_k \mathbf{f}(\mathbf{y}(t_k)) - \mathbf{y}(t_k) - \mathbf{f}(\mathbf{y}(t_k))h_k + \frac{1}{2}\ddot{\mathbf{y}}(\xi_k)h_k^2 \\ &= \frac{1}{2}\ddot{\mathbf{y}}(\xi_k)h_k^2. \end{aligned} \quad (7.33)$$

Sloppily speaking, we observe that $\tau(h_k, \mathbf{y}(t_k))$ converges uniformly with order $\mathcal{O}(h_k^2)$ for $h_k \rightarrow 0$.

② **Estimate for the propagated error in (7.31):**

$$\begin{aligned} \|\Psi^{h_k} \mathbf{y}_k - \Psi^{h_k} \mathbf{y}(t_k)\| &= \|\mathbf{y}_k + h_k \mathbf{f}(\mathbf{y}_k) - \mathbf{y}(t_k) - h_k \mathbf{f}(\mathbf{y}(t_k))\| \\ &= \|\mathbf{e}_k + h_k (\mathbf{f}(\mathbf{y}_k) - \mathbf{f}(\mathbf{y}(t_k)))\| \\ &\leq \|\mathbf{e}_k\| + h_k \|\mathbf{f}(\mathbf{y}_k) - \mathbf{f}(\mathbf{y}(t_k))\| \\ &\stackrel{(7.29)}{\leq} \|\mathbf{e}_k\| + h_k L \|\mathbf{e}_k\| = (1 + h_k L) \|\mathbf{e}_k\|. \end{aligned} \quad (7.34)$$

Total error:

We can now combine the two estimates to obtain one for the total error. For this, define

$$\epsilon_k := \|\mathbf{e}_k\| \text{ and } \rho_k := \frac{1}{2}h_k^2 \max_{t_k \leq \tau \leq t_{k+1}} \|\ddot{\mathbf{y}}(\tau)\|.$$

By the \triangle -inequality, (7.31) yields:

$$\epsilon_{k+1} \leq (1 + h_k L) \epsilon_k + \rho_k. \quad (7.35)$$

Taking into account $\epsilon_0 = 0$, this leads to

$$\epsilon_k \leq \sum_{l=1}^k \prod_{j=1}^{l-1} (1 + h_j L) \rho_l, \quad k = 1, \dots, N. \quad (7.36)$$

Using the estimate $1 \leq (1 + Lh_j) \leq e^{h_j L}$, we further obtain:

$$\epsilon_k \leq \sum_{l=1}^k \prod_{j=1}^{l-1} e^{h_j L} \cdot \rho_l = \sum_{l=1}^k e^{L \sum_{j=1}^{l-1} h_j} \rho_l.$$

Since $\sum_{j=1}^{l-1} h_j \leq T$, we can deduce

$$\epsilon_k \leq e^{LT} \sum_{l=1}^k \rho_l \leq e^{LT} \max_k \frac{\rho_k}{h_k} \sum_{l=1}^k h_l$$

and conclude that:

$$\epsilon_k = \|\mathbf{y}_k - \mathbf{y}(t_k)\| \leq Te^{LT} \left(\max_{l=1, \dots, k} h_l \right) \cdot \left(\max_{t_0 \leq \tau \leq t_k} \|\dot{\mathbf{y}}(\tau)\| \right). \quad (7.37)$$

Note. The total error arises from accumulation of the propagation of one-step errors.

We can summarize the insights gleaned through this theoretical analysis as follows: The error bound

- is $\mathcal{O}(h_{\mathcal{M}})$, where $h_{\mathcal{M}} := \max_l h_l$. In other words, we observe a first order algebraic convergence.
- grows *exponentially* with the length T of the integration interval.
- grows *exponentially* with the Lipschitz constant of \mathbf{f} .
- is *linear* in $\max_{t_0 \leq \tau \leq t_k} \|\ddot{\mathbf{y}}(\tau)\|$.

This implies that rapidly varying functions require much smaller time steps to maintain a certain error.

Note. In practice, an *adaptive* step size control is implemented. This involves the use of larger step sizes in slowly varying regions and use the of smaller step sizes in rapidly varying regions.

One-step error and order of a single step method

In the analysis of the global discretization error of the explicit Euler method in Section 7.3.3, a one-step error of size $\mathcal{O}(h_k^2)$ led to a total error of $\mathcal{O}(h_{\mathcal{M}})$ through the effect of error

accumulation over $N \approx h_{\mathcal{M}}^{-1}$ steps. This relationship remains valid for almost all single step methods:

Consider an IVP (7.5) with solution $t \mapsto \mathbf{y}(t)$, the evolution map Φ associated with the ODE, and a single step method defined by the discrete evolution Ψ . If the *one-step error along the solution trajectory* satisfies

$$\left\| \Psi^h \mathbf{y}(t) - \Phi^h \mathbf{y}(t) \right\| \leq Ch^{p+1} \quad \forall \quad h \text{ sufficiently small, } t \in [0, T],$$

for some $p \in \mathbb{N}$ and $C > 0$, then, usually,

$$\max_k \|\mathbf{y}_k - \mathbf{y}(t_k)\| \leq \bar{C} h_{\mathcal{M}}^p,$$

with $\bar{C} > 0$ independent of the temporal mesh \mathcal{M} .

A rigorous statement as a theorem would involve some particular assumptions on Ψ , which we do not want to give here.

7.4 Higher order Single-step methods (Runge-Kutta Methods)

So far we only know first and second order methods from Section 7.2: In Example 7.3.2, we observed that the explicit Euler (7.15) and implicit Euler (7.18) method are of first order, while the implicit midpoint method (7.21) is of second order.

Thus, barring the impact of roundoff, the low-order polygonal approximation methods are guaranteed to achieve any prescribed accuracy provided that the mesh is fine enough. Why should we need any other timestepping schemes? We argue that the use of higher-order timestepping methods is highly advisable for the sake of efficiency. In what follows, we introduce general higher order schemes. The attribute “single-step” (or “one-step”) indicates that for this family of methods, at each time step, only information from the last previous time step is used (as opposed to “multi-step” methods that we will not cover). We start by introducing two well-known schemes for solving ODEs: RK-2 and RK-4.

The Runge-Kutta-2 method (RK-2)

We consider the IVP (7.9) in the time interval $[0, T]$, where $T \in (0, \infty)$ is some fixed time and we assume the time-discretization described in (7.13). Then for all time levels t_k , $k \in \{0, 1, \dots, N-1\}$, by the Fundamental Theorem of calculus, it holds that

$$\begin{aligned} \mathbf{y}(t_{k+1}) - \mathbf{y}(t_k) &= \int_{t_k}^{t_{k+1}} \dot{\mathbf{y}}(t) dt, \\ &= \int_{t_k}^{t_{k+1}} \mathbf{f}(t, \mathbf{y}(t)) dt. \end{aligned} \tag{7.38}$$

We can then use numerical quadrature to approximate (7.38). In particular, using the mid-point rule yields

$$\int_{t_k}^{t_{k+1}} \mathbf{f}(t, \mathbf{y}(t)) dt \approx h \mathbf{f} \left(t_k + \frac{h}{2}, \mathbf{y} \left(t_k + \frac{h}{2} \right) \right). \quad (7.39)$$

Of course the mid-point value of \mathbf{y} is still unknown so we can perform a further approximation of this value using the Forward Euler method, i.e.,

$$\mathbf{y} \left(t_k + \frac{h}{2} \right) \approx \mathbf{y}(t_k) + \frac{h}{2} \mathbf{f}(t_k, \mathbf{y}(t_k)). \quad (7.40)$$

Finally, combining (7.9) and (7.40), we obtain a *two-stage* numerical scheme for approximating solutions to the IVP (7.9):

$$\begin{aligned} \mathbf{k}_1 &= \mathbf{f}(t_k, \mathbf{y}_k), \\ \mathbf{k}_2 &= \mathbf{f} \left(t_k + \frac{h}{2}, \mathbf{y}_k + \frac{h}{2} \mathbf{k}_1 \right), \\ \mathbf{y}_{k+1} &= \mathbf{y}_k + h \mathbf{k}_2. \end{aligned} \quad (7.41)$$

The numerical scheme given in (7.41) is termed the standard 2-stage Runge-Kutta (RK-2) method and is of second order. The 2-stages refer to the calculation of the two terms $\mathbf{k}_1, \mathbf{k}_2$ in Equation (7.41) in order to compute the solution at the next time level.

We remark that the RK-2 method (7.41) can be re-written in the update form as

$$\begin{aligned} \mathbf{y}_{k+1} &= \mathbf{y}_k + h \mathbf{f} \left(t_k + \frac{h}{2}, \mathbf{y}_k + \frac{h}{2} \mathbf{f}(t_k, \mathbf{y}_k) \right), \\ \mathbf{y}_0 &= \mathbf{y}(0), \end{aligned} \quad (7.42)$$

and therefore represents an explicit method.

The Runge-Kutta-4 method (RK-4)

An even higher-order accurate numerical method for approximating solutions to the IVP (7.9) can be obtained by considering the following 4-stage numerical method:

$$\begin{aligned} \mathbf{k}_1 &= \mathbf{f}(t_k, \mathbf{y}_k), \\ \mathbf{k}_2 &= \mathbf{f} \left(t_k + \frac{h}{2}, \mathbf{y}_k + \frac{h}{2} \mathbf{k}_1 \right), \\ \mathbf{k}_3 &= \mathbf{f} \left(t_k + \frac{h}{2}, \mathbf{y}_k + \frac{h}{2} \mathbf{k}_2 \right), \\ \mathbf{k}_4 &= \mathbf{f}(t_k + h, \mathbf{y}_k + h \mathbf{k}_3), \\ \mathbf{y}_{k+1} &= \mathbf{y}_k + \frac{h}{6} (\mathbf{k}_1 + 2\mathbf{k}_2 + 2\mathbf{k}_3 + \mathbf{k}_4). \end{aligned} \quad (7.43)$$

The numerical scheme given in (7.43) is termed the classical 4-stage Runge-Kutta (RK-4) method and is of fourth order.

We remark that this scheme is also an explicit method similar to the RK-2 method.

7.4.1 General form of Runge-Kutta methods

The RK-2 method and the RK-4 method suggest a general form of an s -stage Runge-Kutta method for approximating solutions to the IVP (7.9):

Let $s \in \mathbb{N}$ be an integer, let $\{a_{ij}\}_{i,j=1}^s$, $\{b_i\}_{i=1}^s$ and $\{c_i\}_{i=1}^s$ be real numbers and for all time levels t_k , $k \in \{0, 1, \dots, N-1\}$, define:

$$\begin{aligned} \mathbf{k}_1 &= \mathbf{f} \left(t_k + c_1 h, \mathbf{y}_k + h \sum_{j=1}^s a_{1j} \mathbf{k}_j \right), \\ &\vdots \\ \mathbf{k}_s &= \mathbf{f} \left(t_k + c_s h, \mathbf{y}_k + h \sum_{j=1}^s a_{sj} \mathbf{k}_j \right), \\ \mathbf{y}_{k+1} &= \mathbf{y}_k + h \sum_{i=1}^s b_i \mathbf{k}_i, \end{aligned} \tag{7.44}$$

where

$$c_i := \sum_{j=1}^s a_{ij}. \tag{7.45}$$

Then the coefficients $\{a_{ij}\}_{i,j=1}^s$, $\{b_i\}_{i=1}^s$ and $\{c_i\}_{i=1}^s$ uniquely specify an s -stage Runge-Kutta method given by Equation (7.44).

For increased clarity and simplicity, the coefficients $\{a_{ij}\}_{i,j=1}^s$, $\{b_i\}_{i=1}^s$ and $\{c_i\}_{i=1}^s$ associated with an s -stage RK method are usually presented in tabular format as a *Butcher Tableau*. The Butcher Tableau for an s -stage RK method is given by

$$\begin{array}{c|c} \mathbf{c} & \mathbf{A} \\ \hline & \mathbf{b}^\top \end{array} := \begin{array}{c|cccc} c_1 & a_{11} & a_{12} & \dots & a_{1s} \\ c_2 & a_{21} & a_{22} & \dots & a_{2s} \\ \vdots & \vdots & \vdots & & \vdots \\ c_s & a_{s1} & a_{s2} & \dots & a_{ss} \\ \hline & b_1 & b_2 & \dots & b_s \end{array} \tag{7.46}$$

Examples of Runge-Kutta methods

We consider two main classes of Runge-Kutta methods in this section.

- Explicit Runge-Kutta Methods

Consider an s -stage Runge-Kutta method of the form (7.44) for approximating solutions to the IVP (7.9) with the property that for all $i, j \in \{1, \dots, s\}$

$$a_{ij} = 0 \text{ if } j \geq i.$$

Then this numerical method is termed an *explicit* RK method.

We observe that by definition of explicit RK schemes, each stage \mathbf{k}_i , $i \in \{2, \dots, s\}$ can be computed using only the previous stages \mathbf{k}_j , $j < i$, and therefore an explicit RK method can be implemented as a time marching scheme:

$$\mathbf{y}_k \mapsto \mathbf{k}_1 \mapsto \mathbf{k}_2 \mapsto \mathbf{k}_3 \mapsto \dots \mapsto \mathbf{k}_{s-1} \mapsto \mathbf{k}_s \mapsto \mathbf{y}_{k+1}.$$

We also remark that the matrix $\mathbf{A} = \{a_{ij}\}_{i,j=1}^s$ in the Butcher tableau (7.46) associated with an explicit RK scheme has a strictly lower triangular structure with zero diagonal entries. Examples of explicit RK schemes include the RK-2 and the RK-4 methods.

Example 7.4.1: Butcher schemes for some explicit RK-SSM

The following explicit Runge-Kutta single step methods are often mentioned in literature.

– Explicit Euler method (7.15):

$$\begin{array}{c|c} 0 & 0 \\ \hline & 1 \end{array} \quad \rightarrow \quad \text{order} = 1$$

– Explicit trapezoidal rule:

$$\begin{array}{c|cc} 0 & 0 & 0 \\ \hline 1 & 1 & 0 \\ \hline & \frac{1}{2} & \frac{1}{2} \end{array} \quad \rightarrow \quad \text{order} = 2$$

– Classical 2nd-order RK-SSM (7.41):

$$\begin{array}{c|cc} 0 & 0 & 0 \\ \hline \frac{1}{2} & \frac{1}{2} & 0 \\ \hline & 0 & 1 \end{array} \quad \rightarrow \quad \text{order} = 2$$

– Classical 4th-order RK-SSM (7.43):

$$\begin{array}{c|cccc} 0 & 0 & 0 & 0 & 0 \\ \hline \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 \\ \frac{1}{2} & 0 & \frac{1}{2} & 0 & 0 \\ \hline 1 & 0 & 0 & 1 & 0 \\ \hline & \frac{1}{6} & \frac{2}{6} & \frac{2}{6} & \frac{1}{6} \end{array} \quad \rightarrow \quad \text{order} = 4$$

– Kutta's 3/8-rule:

$$\begin{array}{c|cccc}
 0 & 0 & 0 & 0 & 0 \\
 \frac{1}{3} & \frac{1}{3} & 0 & 0 & 0 \\
 \frac{2}{3} & -\frac{1}{3} & 1 & 0 & 0 \\
 1 & 1 & -1 & 1 & 0 \\
 \hline
 & \frac{1}{8} & \frac{3}{8} & \frac{3}{8} & \frac{1}{8}
 \end{array} \rightarrow \text{order} = 4$$

Further examples of Runge-Kutta methods can be found in the literature, see for example the Wikipedia page. They are stated in the form of Butcher schemes (7.46) most of the time.

- **Diagonally Implicit Runge-Kutta (DIRK) Methods**
Consider an s -stage Runge-Kutta method of the form (7.44) for approximating solutions to the IVP (7.9) with the property that for all $i, j \in \{1, \dots, s\}$

$$a_{ij} = 0 \text{ if } j > i,$$

and with the property that there exists at least one non-zero diagonal entry, i.e., there exists some $i \in \{1, \dots, s\}$ such that

$$a_{ii} \neq 0.$$

Then this numerical method is termed a *diagonally implicit* RK method.

We observe that by definition of diagonally implicit RK schemes, each stage \mathbf{k}_i , $i \in \{1, \dots, s\}$ can be computed using the stages \mathbf{k}_j , $j \leq i$ and therefore computing each stage \mathbf{k}_i requires the solution of a non-linear system of equations. This implies that we must use some numerical method such as, e.g., Newton's method (cf. Chapter 8) to find the approximate solution to one or more non-linear equations at each time step.

Example 7.4.2:

As a concrete example, we consider the Butcher Tableau for the 3-stage second-order DIRK method given by

$$\begin{array}{c|ccc}
 0 & 0 & 0 & 0 \\
 \frac{1}{2} & \frac{1}{4} & \frac{1}{4} & 0 \\
 1 & \frac{1}{3} & \frac{1}{3} & \frac{1}{3} \\
 \hline
 & \frac{1}{3} & \frac{1}{3} & \frac{1}{3}
 \end{array}$$

Notice that, in contrast to the explicit RK methods, this method has some non-zero diagonal entries. We remark that the DIRK-2 method is also known as the

Trapezoidal Rule with the second order Backward Difference Formula (TR-BDF2).

7.4.2 Consistency conditions for Runge-Kutta methods

In general, an arbitrary combination of the coefficients $\{a_{ij}\}_{i,j=1}^s$, $\{b_i\}_{i=1}^s$ and $\{c_i\}_{i=1}^s$ will not specify a consistent s -stage Runge-Kutta method (7.44). Instead we need to impose certain conditions on these coefficients in order to ensure consistency.

Corollary 7.4.1 (Consistent Runge-Kutta single step methods). *A Runge-Kutta single step method according to (7.44) is consistent (see Definition 7.3.2) with the ODE $\dot{\mathbf{y}} = \mathbf{f}(t, \mathbf{y})$, if and only if*

$$\sum_{i=1}^s b_i = 1. \quad (7.47)$$

Proof. The simplest way to see that the condition (7.47) must be satisfied in order to ensure consistency is to ensure that the consistency criterion from (7.25) is satisfied, i.e.,

$$\psi(0, \mathbf{y}) = \mathbf{f}(\mathbf{y}).$$

The discrete evolution of Runge-Kutta methods can be expressed as:

$$\Psi^h \mathbf{y} = \mathbf{y} + h \underbrace{\sum_{i=1}^s b_i \mathbf{k}_i(\mathbf{y})}_{=\psi(h, \mathbf{y})}.$$

We note that for $h = 0$, the expressions of $\mathbf{k}_i(\mathbf{y})$ in (7.44) simplify to

$$\mathbf{k}_i(\mathbf{y}) = \mathbf{f}(\mathbf{y}).$$

Thus, for RK methods one obtains:

$$\psi(0, \mathbf{y}) = \left(\sum_{i=1}^s b_i \right) \mathbf{f}(\mathbf{y}).$$

Hence, consistency holds if and only if

$$\sum_{i=1}^s b_i = 1.$$

□

Note. Condition (7.47) must hold together with (7.45).

Note. It is desirable to need as few stages as possible to obtain a method of order p . There is a limit to this, also referred to as the *Butcher barriers*. The following table gives lower bounds for the number of stages needed to achieve order p for an explicit Runge-Kutta method.

order p	1	2	3	4	5	6	7	8	≥ 9
minimal no. s of stages	1	2	3	4	6	7	9	11	$\geq p + 3$

No general formula has been discovered. However, it is known that for explicit Runge-Kutta single step methods according to Equation (7.44), the following can be said:

$$\text{order } p \leq \text{number } s \text{ of stages of an RK-SSM.}$$

7.5 Stability of Numerical Methods for ODEs

Convergence in the sense of (7.28) is merely a necessary condition for a 'good' numerical method but is by no means a sufficient condition. The following example helps illustrate this.

Example 7.5.1:

Consider the scalar IVP

$$\begin{aligned} \dot{y}(t) &= \lambda(y(t) - \sin(t)) + \cos(t), \\ y(0) &= 0, \end{aligned} \tag{7.48}$$

where $\lambda \in \mathbb{R}$ is a constant.

It can be easily shown that $y(t) = \sin(t)$ is the unique solution of the IVP (7.48) for any value of the constant λ . We compute the numerical solution of the IVP (7.48) using the explicit Euler method (7.15) for $\lambda = -100$ and different values of the time step h up to the final time $T = 10$. Our results are displayed in Table 7.1.

N	Error	$ 1 + \lambda h $
100	6.70×10^{66}	5.283
200	1.04×10^{59}	2.141
300	1.79×10^5	1.094
320	2.29×10^{-6}	0.964
400	3.74×10^{-7}	0.571

Table 7.1: Error table for the explicit Euler method for the IVP (7.48).

Clearly, the global error associated with the explicit Euler method for different values of h is very large. Indeed, Figure 7.1 displays the result for $N = \frac{T}{h} = 300$ points and indicates that the approximate solution contains very large oscillations and seems to blow up.

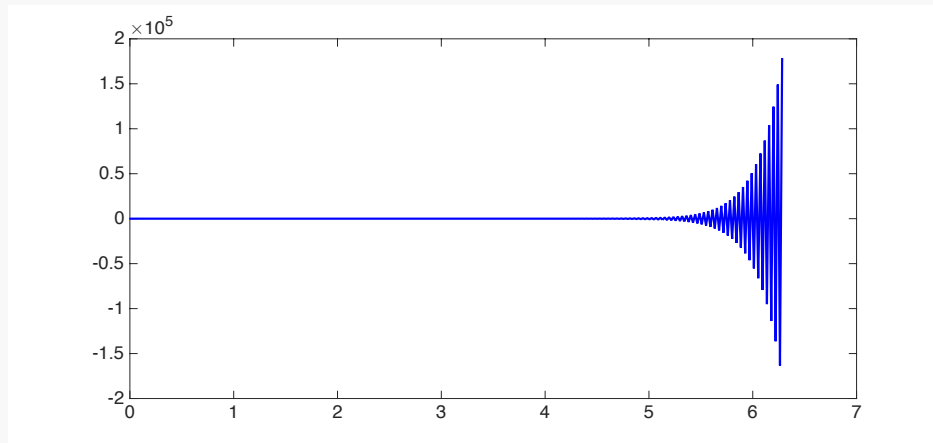


Figure 7.1: Approximate solution produced by the explicit Euler method for the IVP (7.48) using $N=300$ points.

The behavior of the explicit Euler method is very surprising considering that we have shown that this method produces approximate solutions that converge to the exact solution as $h \rightarrow 0$. Interestingly, this result is still true for very small values of the time step h . Indeed Figure 7.2 displays the results for $N = \frac{T}{h} = 400$ points and indicates that the approximate solution in this case is very close to the exact solution as evidenced by a global error of 3.74×10^{-7} .

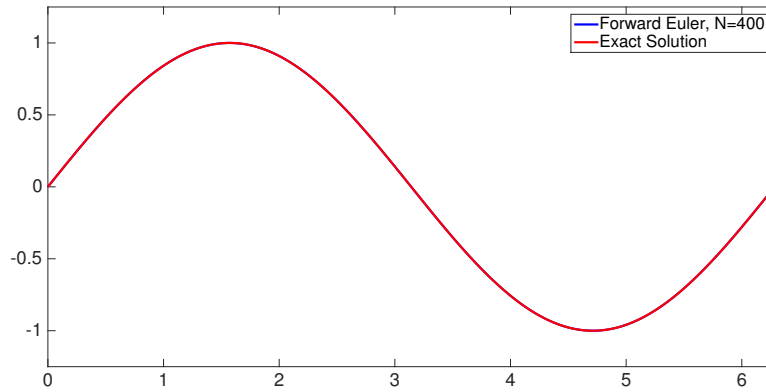


Figure 7.2: Approximate solution produced by the explicit Euler method for the IVP (7.48) using $N=400$ points.

Table 7.1 contains a clue pertaining to this behaviour. Based on our numerical experiments it seems that the value $|1 + \lambda h|$ plays a significant role in the value of the global error associated with the explicit Euler method as h is varied.

Finally, we remark that the implicit Euler method is able to produce accurate solutions to the IVP (7.48) even for very small values of the number of points N , as shown in Figure 7.3.

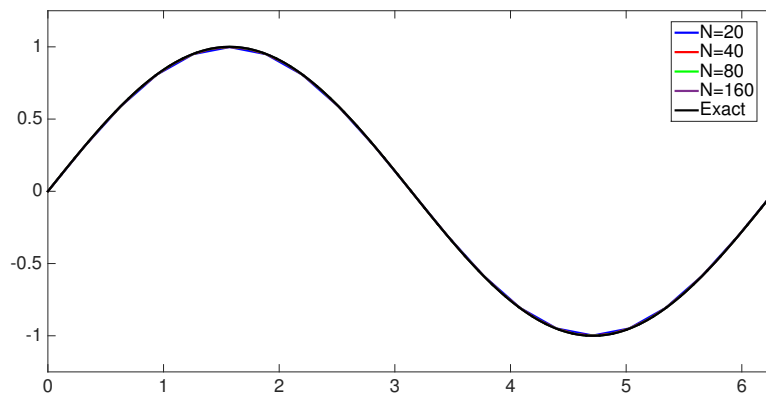


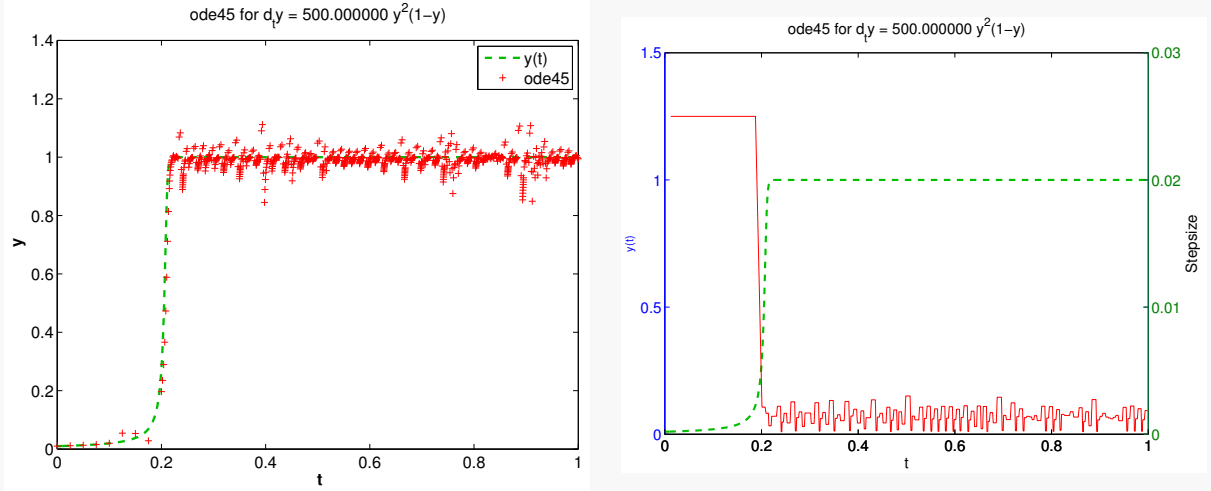
Figure 7.3: Approximate solutions produced by the implicit Euler method for the IVP (7.48).

Example 7.5.2:

In this example we will witness the near failure of a high-order explicit Runge-Kutta method with adaptive stepsize control for the logistic ODE (see Example 7.1.1):

$$\dot{y} = \lambda y^2(1 - y), \quad \lambda := 500, \quad y(0) = \frac{1}{100}. \quad (7.49)$$

We solve it with MATLAB using its built-in adaptive explicit Runge-Kutta method ode45:



The solution is virtually constant for $t > 0.2$. Nevertheless, the integrator uses unnecessarily tiny timesteps until the end of the integration interval.

7.5.1 Absolute Stability

The examples we have seen above provide motivation for a stronger notion of stability for numerical methods. Consider the following model IVP:

$$\begin{aligned} \dot{y}(t) &= \lambda y(t), \\ y(0) &= y_0 \end{aligned} \quad (7.50)$$

where $\lambda \in \mathbb{R}^-$ is a negative constant. Note that the exact solution of the IVP (7.50) is given by $y(t) = y_0 e^{\lambda t}$ and therefore the solution will decay rapidly to zero for any initial condition as long as $\lambda < 0$. In terms of an approximation (y_k) , this decay condition implies

$$|y_{k+1}| < |y_k|. \quad (7.51)$$

We will start by analyzing this condition for the Euler methods and then consider this concept of stability for general Runge-Kutta methods.

Absolute Stability of Explicit Euler Method

Applying the explicit Euler method (7.15) to approximate solutions to the IVP (7.50) leads to

$$y_{k+1} = (1 + \lambda h) y_k. \quad (7.52)$$

In view of Equation (7.51), we say that the explicit Euler method (7.15) is *absolutely stable* if it holds that

$$|1 + \lambda h| < 1. \quad (7.53)$$

Note. We observe that this notion of stability only makes sense when the constant λ is negative, so that the exact solution is strictly monotonically decreasing.

Equation (7.53) implies that the time step h must be chosen such that (see Figure 7.4)

$$-2 < \lambda h < 0.$$

Hence, the explicit Euler method (7.15) is absolutely stable only if the time step h , relative to the constant λ , is sufficiently small. This provides justification for the results of our numerical experiments in Example 7.5.1.

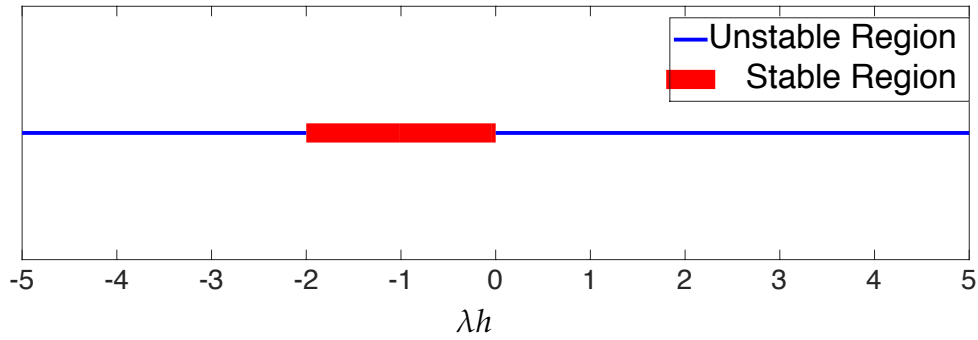


Figure 7.4: Stable and unstable regions of λh for the explicit Euler scheme.

Absolute Stability of the Implicit Euler Method

Applying the implicit Euler method (7.18) to approximate solutions to the IVP (7.50) leads to

$$\begin{aligned} \frac{y_{k+1} - y_k}{h} &= \lambda y_{k+1} \\ \implies y_k &= (1 - \lambda h) y_{k+1} \\ \implies y_{k+1} &= \frac{1}{1 - \lambda h} y_k. \end{aligned}$$

The implicit Euler method (7.18) is absolutely stable if (7.51) holds, i.e., if it holds that

$$\frac{1}{|1 - \lambda h|} < 1, \quad (7.54)$$

or equivalently if (see Figure 7.5)

$$\lambda h \in (-\infty, 0) \cup (2, \infty). \quad (7.55)$$

The large stability region for the implicit Euler scheme given by (7.55) indicates that the implicit Euler scheme remains stable even for large values of the time step h . This explains why the implicit Euler method performs well in the numerical experiments considered in Example 7.5.1.

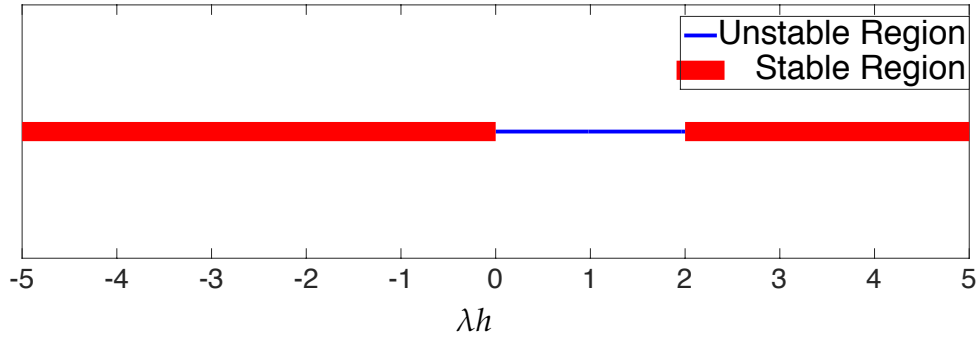


Figure 7.5: Stable and unstable regions of λh for the implicit Euler scheme.

Absolute Stability of Runge-Kutta methods

Prior to generalizing for all Runge-Kutta methods, we consider a specific example of the explicit trapezoidal method.

Example 7.5.3: Explicit trapezoidal method for decay equation

Recall the butcher tableau for the explicit trapezoidal method (see Example 7.4.1):

$$\begin{array}{c|cc} 0 & 0 & 0 \\ 1 & 1 & 0 \\ \hline & \frac{1}{2} & \frac{1}{2} \end{array}$$

Applying it to the IVP (7.50) leads to

$$\begin{aligned} k_1 &= f(t_k, y_k) = \lambda y_k, \\ k_2 &= f(t_k + h, y_k + hk_1) = \lambda(y_k + hk_1), \\ y_{k+1} &= y_k + \frac{h}{2}(k_1 + k_2) \\ &= y_k + \frac{h}{2}(\lambda y_k + \lambda y_k + \lambda^2 h y_k) \\ &= \left(1 + \lambda h + \frac{1}{2}(\lambda h)^2\right) y_k. \end{aligned}$$

We can now define the stability function

$$S(\lambda h) := 1 + \lambda h + \frac{1}{2}(\lambda h)^2. \quad (7.56)$$

The sequence of approximations generated by the explicit trapezoidal rule can be ex-

pressed as

$$\begin{aligned} y_{k+1} &= S(\lambda h)y_k, \quad k = 0, \dots, N-1, \\ y_k &= S(\lambda h)^k y_0, \quad k = 1, \dots, N. \end{aligned} \quad (7.57)$$

Clearly, the decay of the sequence $(y_k)_k$ can only be guaranteed if $|S(\lambda h)| < 1$:

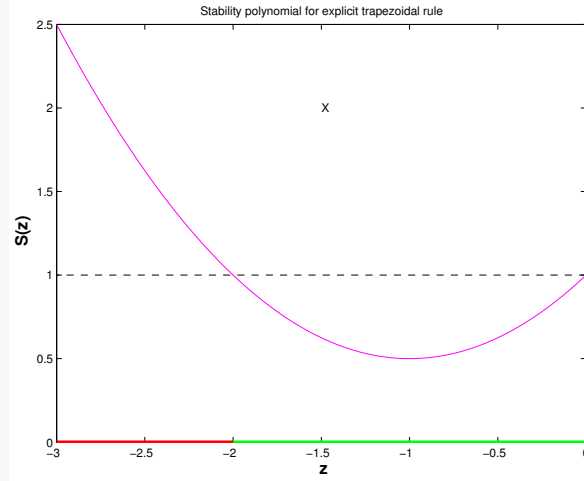


Figure 7.6: The stability function for the explicit trapezoidal method.

Therefore, the stability region can be given by:

$$-2 < \lambda h < 0. \quad (7.58)$$

Now we may generalize the stability analysis for a general Runge-Kutta single step method encoded by the Butcher scheme $\begin{array}{c|c} \mathbf{c} & \mathbf{A} \\ \hline & \mathbf{b}^\top \end{array}$, applied to the autonomous scalar linear ODE (7.50). We write down the equations for the increments and y_{k+1} from (7.44):

$$\begin{aligned} k_i &= \lambda(y_k + h \sum_{j=1}^s a_{ij}k_j), \\ y_{k+1} &= y_k + h \sum_{i=1}^s b_i k_i. \end{aligned}$$

Defining $\mathbf{k} := [k_1, \dots, k_s]^\top / \lambda \in \mathbb{R}^s$ as the vector of increments, and $z := \lambda h$, we obtain

$$\begin{aligned} \mathbf{k} &= y_k[1, \dots, 1]^\top + z\mathbf{A}\mathbf{k}, \\ y_{k+1} &= y_k + z\mathbf{b}^\top \mathbf{k}. \end{aligned}$$

This system of equations can then be converted into matrix form:

$$\begin{bmatrix} \mathbf{I} - z\mathbf{A} & 0 \\ -z\mathbf{b}^\top & 1 \end{bmatrix} \begin{bmatrix} \mathbf{k} \\ y_{k+1} \end{bmatrix} = y_k \begin{bmatrix} \mathbf{1} \\ 1 \end{bmatrix}. \quad (7.59)$$

Applying block Gaussian elimination to solve for y_{k+1} , we obtain

$$\begin{aligned} y_{k+1} &= y_k + z\mathbf{b}^\top (\mathbf{I} - z\mathbf{A})^{-1} y_k \mathbf{1} \\ &= y_k \left(1 + z\mathbf{b}^\top (\mathbf{I} - z\mathbf{A})^{-1} \mathbf{1} \right). \end{aligned}$$

Therefore we can define the stability function for a *general RK-SSM* as

$$S(z) := 1 + z\mathbf{b}^\top (\mathbf{I} - z\mathbf{A})^{-1} \mathbf{1}, \quad (7.60)$$

which implies

$$y_{k+1} = S(\lambda h)y_k.$$

Alternatively, we can express y_{k+1} through determinants by appealing to Cramer's rule,

$$y_{k+1} = y_k \frac{\det \begin{bmatrix} \mathbf{I} - z\mathbf{A} & \mathbf{1} \\ -z\mathbf{b}^\top & 1 \end{bmatrix}}{\det \begin{bmatrix} \mathbf{I} - z\mathbf{A} & \mathbf{0} \\ -z\mathbf{b}^\top & 1 \end{bmatrix}}. \quad (7.61)$$

This can be simplified using Schur's determinant identity:

$$\det \begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{bmatrix} = \det \mathbf{D} \cdot \det(\mathbf{A} - \mathbf{B}\mathbf{D}^{-1}\mathbf{C}).$$

Therefore

$$y_{k+1} = y_k \frac{\det(\mathbf{I} - z\mathbf{A} + z\mathbf{1}\mathbf{b}^\top)}{\det(\mathbf{I} - z\mathbf{A})},$$

which shows that the stability function for a *general RK-SSM* can also be written as

$$S(z) := \frac{\det(\mathbf{I} - z\mathbf{A} + z\mathbf{1}\mathbf{b}^\top)}{\det(\mathbf{I} - z\mathbf{A})},$$

and the general solution y_k can be related to the initial value y_0 as

$$y_k = S(\lambda h)^k y_0.$$

For an *explicit* RK method, note that \mathbf{A} is a strictly lower triangular matrix, which means that $\det(\mathbf{I} - z\mathbf{A}) = 1$. Therefore,

$$y_{k+1} = y_k \det(\mathbf{I} - z\mathbf{A} + z\mathbf{1}\mathbf{b}^\top).$$

Thus, the stability function for an *explicit* RK method can be written as

$$S(z) = \det(\mathbf{I} - z\mathbf{A} + z\mathbf{1}\mathbf{b}^\top).$$

Therefore, we have proven the following theorem.

Theorem 7.5.1 (Stability function of general Runge-Kutta methods). *The discrete evolution Ψ^h of an s -stage Runge-Kutta single step method (see (7.44)) with the Butcher scheme, $\begin{array}{c|c} \mathbf{c} & \mathbf{A} \\ \hline & \mathbf{b}^\top \end{array}$, for the ODE $\dot{y} = \lambda y$ is given by*

$$\Psi^h = S(\lambda h) \Leftrightarrow y_{k+1} = S(\lambda h)y_k,$$

where S is the stability function

$$S(z) := 1 + z\mathbf{b}^\top (\mathbf{I} - z\mathbf{A})^{-1} \mathbf{1} = \frac{\det(\mathbf{I} - z\mathbf{A} + z\mathbf{1}\mathbf{b}^\top)}{\det(\mathbf{I} - z\mathbf{A})}, \quad z := \lambda h, \quad \mathbf{1} := [1, \dots, 1]^\top \in \mathbb{R}^s. \quad (7.62)$$

Note. For an *explicit* s -stage Runge-Kutta single step method, the stability function in (7.62) simplifies to

$$S(z) := \det(\mathbf{I} - z\mathbf{A} + z\mathbf{1}\mathbf{b}^\top), \quad z := \lambda h, \quad \mathbf{1} := [1, \dots, 1]^\top \in \mathbb{R}^s. \quad (7.63)$$

Equation (7.63) confirms an immediate consequence of the determinant formula for the stability function $S(z)$.

Corollary 7.5.1 (Polynomial stability function of explicit RK-SSM). *For a consistent s -stage explicit Runge-Kutta single step method according to (7.44), the stability function S defined by (7.63) is a non-constant polynomial of degree $\leq s$: $S \in \mathcal{P}_s$.*

From the determinant formula (7.62) for the stability function $S(z)$, we can conclude a generalization of Corollary 7.5.1.

Corollary 7.5.2 (Rational stability function of explicit RK-SSM). *For a consistent (see Definition 7.3.2) s -stage general Runge-Kutta single step method according to (7.44), the stability function S is a non-constant rational function of the form $S(z) = \frac{P(z)}{Q(z)}$ with polynomials $P \in \mathcal{P}_s$, $Q \in \mathcal{P}_s$.*

Region of (absolute) stability of Runge-Kutta methods

We consider a general Runge-Kutta single step method with stability function S for the model linear scalar IVP $\dot{y} = \lambda y$, $y(0) = y_0$, $\lambda \in \mathbb{C}$, i.e. λ can be complex valued. From Theorem 7.5.1, we learn that for uniform stepsize $h > 0$, we have $y_k = S(\lambda h)^k y_0$ and conclude that

$$y_k \rightarrow 0 \quad \text{for } k \rightarrow \infty \Leftrightarrow |S(\lambda h)| < 1. \quad (7.64)$$

Hence, the modulus $|S(\lambda h)|$ indicates the combinations of λ and stepsize h for which we achieve exponential decay $y_k \rightarrow 0$ as $k \rightarrow \infty$, which is the desirable behavior of the approximations for $\Re(\lambda) < 0$.

Definition 7.5.1 (Region of (absolute) stability). Let the discrete evolution Ψ for a single step method applied to the scalar linear ODE $\dot{y} = \lambda y$, $\lambda \in \mathbb{C}$, be of the form

$$\Psi^h y = S(z)y, \quad y \in \mathbb{C}, h > 0 \quad \text{with} \quad z := \lambda h \quad (7.65)$$

and a function $S : \mathbb{C} \rightarrow \mathbb{C}$. Then the *region of (absolute) stability* of the single step method is given by

$$\mathcal{S}_\Psi := \{z \in \mathbb{C} : |S(z)| < 1\} \subset \mathbb{C}.$$

Of course, by Theorem 7.5.1, in the case of a general RK-SSM, the function S will coincide with their *stability function* from (7.62).

Note. We can conclude that

- the regions of (absolute) stability of explicit RK-SSM are bounded, since the stability function $S(z)$ is a non-constant polynomial (see Corollary 7.5.1).
- on the other hand, the stability function $S(z)$ of an implicit RK-SSM is a rational function (see Corollary 7.5.2) that can satisfy $\lim_{|z| \rightarrow \infty} |S(z)| < 1$. As a consequence, the region of stability for an implicit RK-SSM does not have to be bounded.

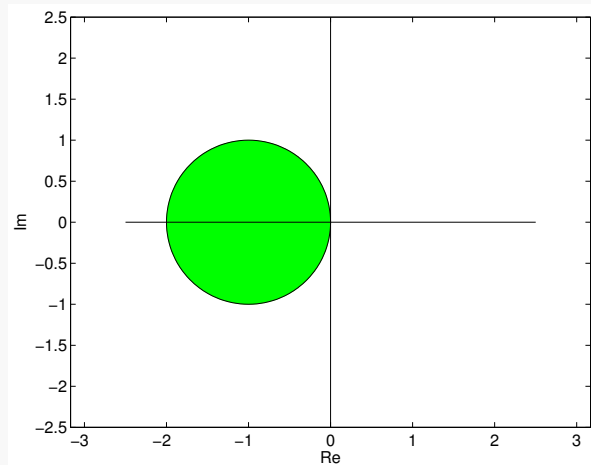
Example 7.5.4: Region of absolute stability of explicit RK-SSM

From Theorem 7.5.1 and the Butcher schemes, we can instantly compute the stability functions of explicit RK-SSM and use them to obtain the region of absolute stability. The domains in \mathbb{C} highlighted in green depict the bounded regions of stability for some RK-SSM from Example 7.4.1.

Explicit Euler method (7.15):

$$\begin{array}{c|c} 0 & 0 \\ \hline & 1 \end{array}$$

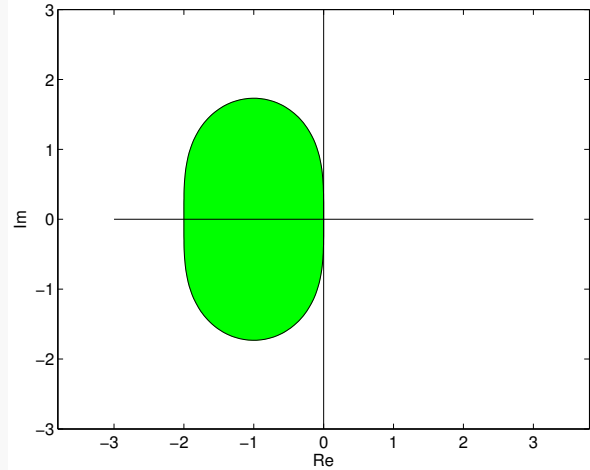
$$S(z) = 1 + z$$



Explicit trapezoidal method:

$$\begin{array}{c|cc} 0 & 0 & 0 \\ 1 & 1 & 0 \\ \hline & \frac{1}{2} & \frac{1}{2} \end{array}$$

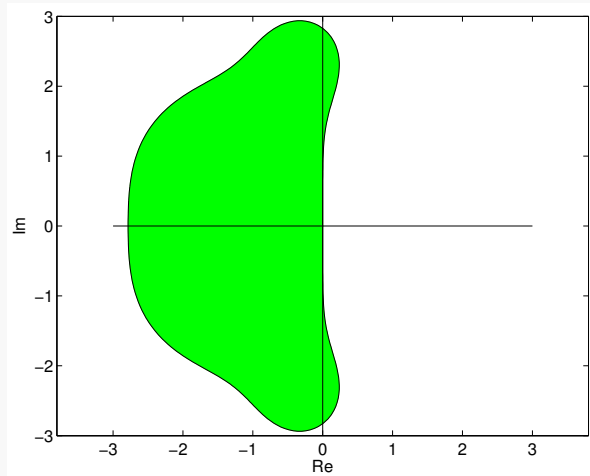
$$S(z) = 1 + z + \frac{1}{2}z^2$$



Classical RK4 method:

$$\begin{array}{c|cccc} 0 & 0 & 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 \\ \frac{1}{2} & 0 & \frac{1}{2} & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ \hline & \frac{1}{6} & \frac{2}{6} & \frac{2}{6} & \frac{1}{6} \end{array}$$

$$S(z) = 1 + z + \frac{1}{2}z^2 + \frac{1}{6}z^3 + \frac{1}{24}z^4$$



Note that the region of stability is bounded in each case.

In general, for a consistent explicit RK-SSM (see Definition 7.3.2), its stability function satisfies $S(z) = 1 + z + \mathcal{O}(z^2)$ for $z \rightarrow 0$. Therefore, $\mathcal{S}_{\Psi} \neq \emptyset$ and the imaginary axis will be tangent to \mathcal{S}_{Ψ} at $z = 0$.

Example 7.5.5: Regions of stability for simple implicit RK-SSM

We compute the stability functions for some simple implicit RK-SSM using Theorem 7.5.1. Their regions of stability \mathcal{S}_{Ψ} , as defined in Definition 7.5.1, can be easily found from the respective stability functions.

- Implicit Euler method:

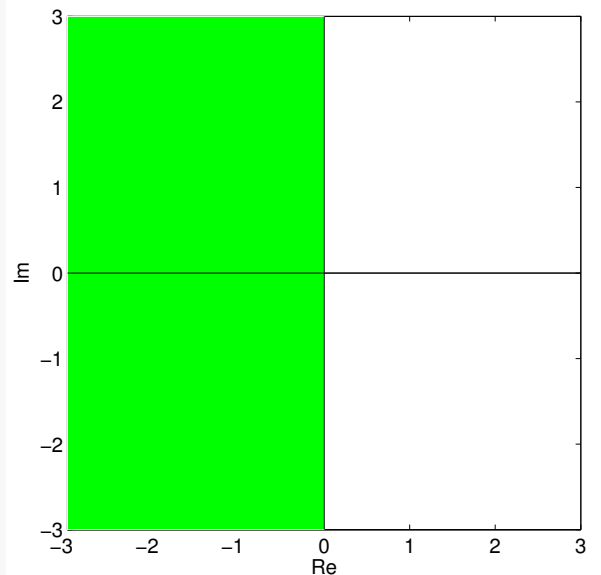
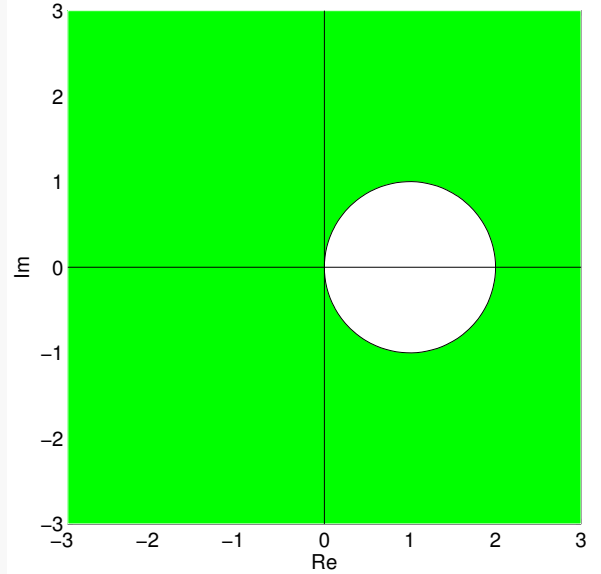
$$\begin{array}{c|c} 1 & 1 \\ \hline & 1 \end{array}$$

$$S(z) = \frac{1}{1-z}$$

- Implicit midpoint method:

$$\begin{array}{c|c} \frac{1}{2} & \frac{1}{2} \\ \hline & 1 \end{array}$$

$$S(z) = \frac{1 + \frac{1}{2}z}{1 - \frac{1}{2}z}$$



We see that in both cases $|S(z)| < 1$, if $\Re(z) < 0$. Also note that the region of stability is not bounded in either case.

7.5.2 A-stability

A general RK-SSM with stability function S applied to the scalar linear IVP $\dot{y} = \lambda y$, $y(0) = y_0 \in \mathbb{C}$, $\lambda \in \mathbb{C}$, with uniform timestep $h > 0$ will yield the sequence $(y_k)_{k=0}^{\infty}$ defined by

$$y_k = S(z)^k y_0 \quad , \quad z = \lambda h . \quad (7.66)$$

Hence, the next property of a RK-SSM guarantees that the sequence of approximations decays exponentially whenever the exact solution of the model problem IVP (7.50) does so.

Definition 7.5.2 (A-stability of a Runge-Kutta single step method). A Runge-Kutta single step method with stability function S is *A-stable*, if

$$\mathbb{C}^- := \{z \in \mathbb{C} : \Re(z) < 0\} \subset \mathcal{S}_\Psi . \quad (\mathcal{S}_\Psi \text{ is the region of stability, see Definition 7.5.1})$$

From Example 7.5.5, we conclude that both the implicit Euler method and the implicit midpoint method are A-stable.

Note. An A-stable method is necessarily implicit.

The family of A-stable Runge Kutta methods include the s -stage Gauss-Legendre methods (or Gauss collocation methods) which have order $2s$.

Example 7.5.6: Gauss-Legendre method of order 2 and 4

The Gauss-Legendre method of order 2 is also known as the midpoint rule. Its Butcher scheme is given by

$$\begin{array}{c|c} \frac{1}{2} & \frac{1}{2} \\ \hline & 1 \end{array} .$$

And the Butcher scheme of the Gauss-Legendre method of order 4 is given by:

$$\begin{array}{c|cc} \frac{1}{2} - \frac{\sqrt{3}}{6} & \frac{1}{4} & \frac{1}{4} - \frac{\sqrt{3}}{6} \\ \frac{1}{2} + \frac{\sqrt{3}}{6} & \frac{1}{4} + \frac{\sqrt{3}}{6} & \frac{1}{4} \\ \hline & \frac{1}{2} & \frac{1}{2} \end{array} .$$

The stability region of Gauss-Legendre methods is $\mathcal{S}_\Psi = \mathbb{C}^-$. Higher order Gauss-Legendre methods (order $p > 6$) are rarely used, due to high computational costs.

7.5.3 Systems of linear ordinary differential equations

So far, we have studied stability of numerical methods for ODEs for the scalar model problem (7.50). Next, we derive that the above results extend to general linear systems of ODEs. A generic system of linear ordinary differential equations on state space \mathbb{R}^d has the form

$$\dot{\mathbf{y}} = \mathbf{M}\mathbf{y} , \tag{7.67}$$

where matrix $\mathbf{M} \in \mathbb{R}^{d,d}$ is assumed to be diagonalizable, i.e., we can find a *regular* matrix $\mathbf{V} \in \mathbb{C}^{d,d}$ such that

$$\mathbf{M} = \mathbf{V}\mathbf{D}\mathbf{V}^{-1}, \quad (7.68)$$

where

$$\mathbf{D} = \begin{bmatrix} \lambda_1 & & 0 \\ & \ddots & \\ 0 & & \lambda_d \end{bmatrix} \in \mathbb{C}^{d,d}$$

is a diagonal matrix and the columns of

$$\mathbf{V} = \left[\begin{array}{c|c|c|c} \mathbf{v}_1 & \mathbf{v}_2 & \dots & \mathbf{v}_d \end{array} \right]$$

are a basis of *eigenvectors* of \mathbf{M} . Then, $\lambda_j \in \mathbb{C}$, $j = 1, \dots, d$ are the associated *eigenvalues* of \mathbf{M} . In other words,

$$\mathbf{M}\mathbf{v}_i = \lambda_i \mathbf{v}_i.$$

The idea behind diagonalization is the transformation of (7.67) into d *decoupled* scalar linear ODEs.

We apply an s -stage RK-SSM described by the Butcher scheme $\begin{array}{c|c} \mathbf{c} & \mathbf{A} \\ \hline & \mathbf{b}^\top \end{array}$ to the autonomous linear ODE (7.67), and obtain (for the first step with timestep size $h > 0$)

$$\mathbf{k}_\ell = \mathbf{M}(\mathbf{y}_0 + h \sum_{j=1}^{s-1} a_{\ell j} \mathbf{k}_j), \quad \ell = 1, \dots, s, \quad \mathbf{y}_1 = \mathbf{y}_0 + h \sum_{\ell=1}^s b_\ell \mathbf{k}_\ell. \quad (7.69)$$

Then, we introduce the substitutions

$$\hat{\mathbf{k}}_\ell := \mathbf{V}^{-1} \mathbf{k}_\ell, \quad \ell = 1, \dots, s, \quad \hat{\mathbf{y}}_k := \mathbf{V}^{-1} \mathbf{y}_k, \quad k = 0, 1$$

to (7.69). Owing to (7.68), this yields

$$\begin{aligned} \hat{\mathbf{k}}_\ell &= \mathbf{D}(\mathbf{V}^{-1} \mathbf{y}_0 + h \sum_{j=1}^{s-1} a_{\ell j} \mathbf{V}^{-1} \mathbf{k}_j), \quad \ell = 1, \dots, s, \quad \hat{\mathbf{y}}_1 = \mathbf{V}^{-1} \mathbf{y}_0 + h \sum_{\ell=1}^s b_\ell \mathbf{V}^{-1} \mathbf{k}_\ell. \\ \implies \hat{\mathbf{k}}_\ell &= \mathbf{D}(\hat{\mathbf{y}}_0 + h \sum_{j=1}^{s-1} a_{\ell j} \hat{\mathbf{k}}_j), \quad \ell = 1, \dots, s, \quad \hat{\mathbf{y}}_1 = \hat{\mathbf{y}}_0 + h \sum_{\ell=1}^s b_\ell \hat{\mathbf{k}}_\ell. \end{aligned} \quad (7.70)$$

Rewriting this in a componentwise format yields:

$$(\hat{\mathbf{k}}_\ell)_i = \lambda_i \left((\hat{\mathbf{y}}_0)_i + h \sum_{j=1}^{s-1} a_{\ell j} (\hat{\mathbf{k}}_j)_i \right), \quad (\hat{\mathbf{y}}_1)_i = (\hat{\mathbf{y}}_0)_i + h \sum_{\ell=1}^s b_\ell (\hat{\mathbf{k}}_\ell)_i, \quad i = 1, \dots, d. \quad (7.71)$$

We infer that, if $(\mathbf{y}_k)_k$ is the sequence produced by an RK-SSM applied to $\dot{\mathbf{y}} = \mathbf{M}\mathbf{y}$, then

$$\mathbf{y}_k = \mathbf{V}\hat{\mathbf{y}}_k ,$$

where $(\hat{\mathbf{y}}_k)_k$ is the sequence generated by the same RK-SSM with the same sequence of timesteps for the IVP $\hat{\mathbf{y}} = \mathbf{D}\hat{\mathbf{y}}$, $\hat{\mathbf{y}}(0) = \mathbf{V}^{-1}\mathbf{y}_0$.

Note. The RK-SSM generates uniformly bounded solution sequences $(\mathbf{y}_k)_{k=0}^\infty$ for $\dot{\mathbf{y}} = \mathbf{M}\mathbf{y}$ with diagonalizable matrix $\mathbf{M} \in \mathbb{R}^{d,d}$ with eigenvalues $\lambda_1, \dots, \lambda_d$, if and only if it generates uniformly bounded sequences for all the scalar ODEs $\dot{z} = \lambda_i z$, $i = 1, \dots, d$.

Hence, understanding the behavior of RK-SSMs for autonomous scalar linear ODEs $\dot{y} = \lambda y$ with $\lambda \in \mathbb{C}$ is enough to predict their behavior for general autonomous linear systems of ODEs.

Theorem 7.5.2 ((Absolute) stability of RK-SSMs for linear systems of ODEs). *The sequence $(\mathbf{y}_k)_k$ of approximations generated by an RK-SSM with stability function S (defined in (7.62)) applied to the linear autonomous ODE $\dot{\mathbf{y}} = \mathbf{M}\mathbf{y}$, $\mathbf{M} \in \mathbb{C}^{d,d}$, with uniform timestep $h > 0$ decays exponentially for every initial state $\mathbf{y}_0 \in \mathbb{C}^d$, if and only if $|S(\lambda_i h)| < 1$ for all eigenvalues λ_i of \mathbf{M} .*

Note. If

$$\Re(\lambda_i) < 0 \quad \forall i \in \{1, \dots, d\},$$

then

$$\|\mathbf{y}(t)\| \rightarrow 0 \quad \text{as } t \rightarrow \infty ,$$

for any solution of $\dot{\mathbf{y}} = \mathbf{M}\mathbf{y}$.

7.6 Stiff Initial Value Problems

Stiffness is a property of differential equations with strong implications for their practical solution using numerical methods. Stiff ODEs arise in various applications; e.g., when modeling chemical reactions, in control theory, or electrical circuits, such as the Van der Pol equation in relaxation oscillation. Unfortunately, a precise mathematical definition of stiffness covering all occurrences of this phenomenon has not been found. Some attempts at describing a stiff problem are:

- An initial value problem is called *stiff*, if stability imposes much tighter timestep constraints on *explicit single step methods* than the accuracy requirements.
- A problem is stiff if it contains widely varying time scales, i.e., some components of the solution decay much more rapidly than others.

- A problem is stiff if explicit methods with adaptive stepsize control work only extremely slowly.

How to distinguish stiff initial value problems

An initial value problem for an autonomous ODE $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$ will probably be stiff, if, for substantial periods of time,

$$\min\{\Re(\lambda) : \lambda \in \sigma(D\mathbf{f}(\mathbf{y}(t)))\} \ll 0, \quad (7.72)$$

where $t \mapsto \mathbf{y}(t)$ is the solution trajectory, $D\mathbf{f}(\mathbf{y})$ is the Jacobian matrix and $\sigma(D\mathbf{f}(\mathbf{y}))$ is the set of eigenvalues of the matrix $D\mathbf{f}(\mathbf{y})$.

Equation (7.72) means that we have at least one eigenvalue with a large negative real part.

Example 7.6.1: Predicting stiffness of non-linear IVPs

- ❶ We consider the IVP from Example 7.5.2:

$$\dot{y} = f(y) := \lambda y^2(1 - y), \quad \lambda := 500, \quad y(0) = \frac{1}{100}.$$

We are interested in the behavior of this ODE close to the stationary point $y = 1$, since this is where the solver ode45 in Example 7.5.2 had problems choosing a reasonable timestep. We find

$$f'(y) = \lambda(2y - 3y^2) \Rightarrow f'(1) = -\lambda.$$

Hence, in case $\lambda \gg 0$, we encounter stiffness close to the stationary state $y = 1$. The observations made in Figure 7.7 exactly match this prediction.

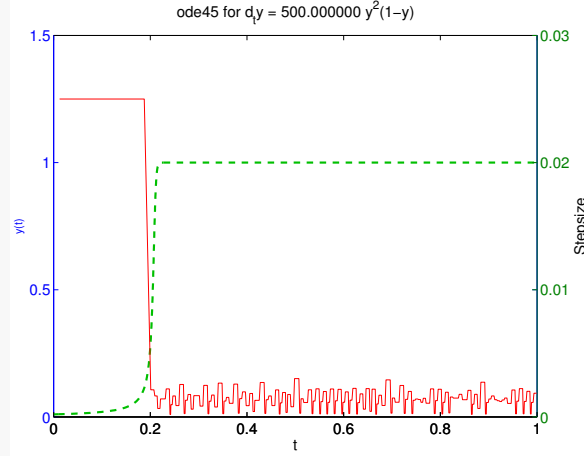


Figure 7.7: Stepsize control of ode45 solver running crazy!

The solution is virtually constant from $t > 0.2$ and, nevertheless, the integrator uses tiny timesteps until the end of the integration interval because of stability constraints.

② The solution of the IVP

$$\dot{\mathbf{y}} = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \mathbf{y} + \lambda(1 - \|\mathbf{y}\|^2) \mathbf{y}, \quad \|\mathbf{y}_0\|_2 = 1.$$

satisfies $\|\mathbf{y}(t)\|_2 = 1$ for all times t . Using the product rule of multi-dimensional differential calculus, we find

$$D\mathbf{f}(\mathbf{y}) = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} + \lambda \left(-2\mathbf{y}\mathbf{y}^\top + (1 - \|\mathbf{y}\|_2^2) \mathbf{I} \right).$$

And the eigenvalues,

$$\sigma(D\mathbf{f}(\mathbf{y})) = \left\{ -\lambda - \sqrt{\lambda^2 - 1}, -\lambda + \sqrt{\lambda^2 - 1} \right\}, \quad \text{if } \|\mathbf{y}\|_2 = 1.$$

Thus, for $\lambda \gg 1$, $D\mathbf{f}(\mathbf{y}(t))$ will always have an eigenvalue with large negative real part, whereas the other eigenvalue is close to zero. This implies that the IVP is stiff.

Note. There is no unified definition or characterization of stiffness. In the literature, one also often refers to stiffness if additionally to one eigenvalue having large negative real part, there is also one eigenvalue whose real part is close to zero (as in the above example). In this case, one speaks of a large *stiffness ratio*.

Note (Characteristics of stiff IVPs). One can often tell from the expected behavior of the solution of an IVP, which is usually clear from the modeling context, that one has to brace for stiffness.

Typical features of stiff IVPs:

- Presence of fast transients in the solution.
- Occurrence of strongly attractive fixed points/limit cycles.

However, recall from Example 7.5.1 that one has to be cautious: there, the solution itself was very nice ($y(t) = \sin(t)$), and stiffness was a property of the ODE itself rather than of the solution. One way to deal with a stiff problem is to have an unconditionally stable solver. This is where the notion of A-stability becomes useful and stiff problems should be solved with A-stable methods.

Iterative Methods for Non-Linear Systems of Equations

So far we have considered direct methods for solving *linear* systems of equations. Many real-world applications do, however, involve *non-linear* systems of equations. In general, such systems cannot be solved directly – or even exactly. In this chapter, we will present iterative methods for finding *approximations* to their solutions instead.

Example 8.0.1: Spherical Water Tank

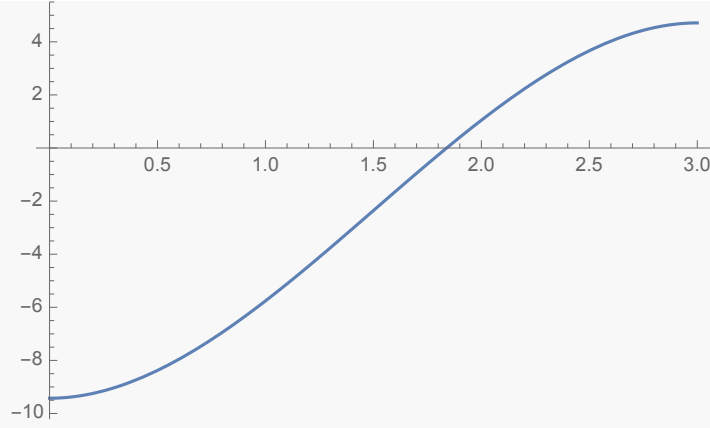
We look at a spherical water tank with radius r and constant outflow ρ . At the beginning $t = 0$, we have a full tank ($h_0 = 2r$) and our task is to determine the height h at any given time t . The height h can be shown to satisfy

$$-\frac{1}{3}\pi h^3 + \pi r h^2 + \left(\rho t - \frac{4}{3}\pi r^3\right) = 0.$$

We define

$$f_{t^*}(h) := -\frac{1}{3}\pi h^3 + \pi r h^2 + \left(\rho t^* - \frac{4}{3}\pi r^3\right)$$

and aim at finding a root h^* of $f_{t^*}(h)$, i.e. we solve for $f_{t^*}(h^*) = 0$.



Plot of $f_{t^*}(h)$ for $t^* = \frac{1}{3}t_{\text{end}}$.

If we want to obtain the height at $t^* = \frac{1}{3} \cdot \underbrace{\frac{\frac{4}{3}\pi r^3}{\rho}}_{t_{\text{end}}}$, where $r = 1.5m$, we can observe that the root lies at $h^* \approx 1.8391m$.

Typical examples of non-linear equations are

- thermodynamic models (these involve equations of state for real gases),
- the Colebrook equation for the Darcy friction factor (which quantifies the pressure drop in oil or gas pipelines).
- solving for timestepping in implicit RK methods for solving ODEs.

Since, in general, nonlinear equations $f(x) = 0$ cannot be solved directly (nor exactly), the question of how to solve them becomes fundamental.

Before answering that question, let us first analyze when $f(x) = 0$ is solvable. For example, the function $f(x) = e^{-\pi x^2}$ does not have roots. A first simple criterion to decide whether a function f has roots is the *intermediate value theorem*.

Theorem 8.0.1 (Intermediate value theorem). *If $f : [a, b] \rightarrow \mathbb{R}$ is continuous and for some $t_\ell, t_r \in [a, b]$*

$$f(t_\ell) < u < f(t_r)$$

holds, then there exists a point $z \in (t_\ell, t_r)$ such that

$$f(z) = u.$$

In particular, for $f \in C^0([a, b], \mathbb{R})$ it suffices to find $t_\ell, t_r \in [a, b]$ with $f(t_\ell) < 0$ and $f(t_r) > 0$ in order to guarantee that f has a root in (t_ℓ, t_r) . We may refine this idea to devise the following root finding algorithm known as the *bisection algorithm*.

Code Snippet 8.1: Bisection Algorithm

14 // Searching zero of f in $[a, b]$ by bisection

```

15 template <typename Func, typename Scalar>
16 Scalar bisection(Func&& f, Scalar a, Scalar b, Scalar tol)
17 {
18     if (a > b) std::swap(a,b); // sort interval bounds
19     if (f(a)*f(b) > 0) throw "f(a) and f(b) have same sign";
20     static_assert(std::is_floating_point<Scalar>::value,
21                 "Scalar must be a floating point type");
22     int v=f(a) < 0 ? 1 : -1;
23     Scalar x = (a+b)/2; // determine midpoint
24     // termination, relies on machine arithmetic if tol = 0
25     while (b-a > tol) { //
26         assert(a<x && x<b); // assert invariant
27         // sgn(f(x)) = sgn(f(b)), then use x as next right boundary
28         if (v*f(x) > 0) b=x;
29         // sgn(f(x)) = sgn(f(a)), then use x as next left boundary
30         else a=x;
31         x = (a+b)/2; // determine next midpoint
32     }
33     return x;
34 }
35 /*

```

The C++ code above implements the bisection method for finding the zeros of a function passed through the *function handle* f in the interval $[a, b]$ with *absolute tolerance* tol . When f is continuous, we always search in a region where a root exists. The region/interval size is halved in each step which guarantees convergence.

Example 8.0.2: Spherical Water Tank

In Example 8.0.1, we wanted to find the height of a fluid in a spherical water tank at time

$$t^* = \frac{1}{3}t_{\text{end}}.$$

In order to do this, we needed to perform root finding on the function

$$f(h) = -\frac{1}{3}\pi h^3 + \pi r h^2 - \frac{8}{9}\pi r^3.$$

It is readily seen that $f(0) < 0$ as well as $f(2r) > 0$ and thus we may apply the bisection method with $t_\ell = 0$ as well as $t_r = 2r$.

For an iterative method to the root finding problem (such as the bisection method) we will denote the iterates as $(x^{(k)})_{k \in \mathbb{N}}$ and define the iteration error as

$$e^{(k)} := x^{(k)} - \underbrace{x^*}_{\text{exact solution}}.$$

For the bisection method, we can easily give the rate of convergence of $x^{(k)} \rightarrow x^*$: We have that

$$\begin{aligned} |e^{(1)}| &\leq \frac{1}{2} |a - b|, \\ |e^{(2)}| &\leq \frac{1}{2^2} |a - b|, \\ &\vdots \\ |x^{(k)} - x^*| &\leq \frac{1}{2^k} |a - b|, \quad |e^{(k)}| \rightarrow 0 \text{ as } k \rightarrow \infty. \end{aligned}$$

As we can see this is a "linear-type" of convergence since the error is reduced by a fixed factor (here: $\frac{1}{2}$) in each step. The bisection method has the advantage that it is rather robust and converges globally. On the other hand, bisection exhibits comparably slow convergence and cannot easily be extended to higher dimensions (n different quantities have to be zero *simultaneously*). In the following, we try to find methods which can be extended to higher dimension and guarantee faster convergence (under *additional* assumptions).

8.1 Fixed Point Iterations in 1D

As before, we want to find a root of f , i.e. $f(x) = 0$. We can reformulate this equivalently as finding a *fixed point* of $\Phi(x) := f(x) + x$, where Φ is called *iteration function*.

Definition 8.1.1 (Fixed point). The point x^* is a *fixed point (FP)* of an iteration function Φ if and only if $\Phi(x^*) = x^*$.

Note that a fixed point x^* of the iteration function $\Phi(x) := f(x) + x$ satisfies $f(x^*) = 0$, by definition. Remember that in the case of the bisection method, the continuity of f was sufficient for convergence. For more general iteration methods, we assume that Φ is Lipschitz continuous, however. Recall the following definition:

Definition 8.1.2 (Lipschitz Continuity). A function $\Phi : [a, b] \rightarrow \mathbb{R}$ is *Lipschitz continuous* if

$$\exists L > 0 : \quad \|\Phi(x) - \Phi(y)\| \leq L \|x - y\| \quad \forall x, y \in [a, b].$$

In general, Lipschitz continuity of Φ will not be sufficient to guarantee convergence of an iterative method. We will also need to assume that $L < 1$. In this case, we say that Φ is a *contractive mapping*.

Definition 8.1.3 (Contractive Mapping). An iteration function $\Phi : [a, b] \rightarrow \mathbb{R}$ is *contractive* (w.r.t. the norm $\|\cdot\|$ on \mathbb{R}) if

$$\exists L \in (0, 1) : \quad \|\Phi(x) - \Phi(y)\| \leq L \|x - y\| \quad \forall x, y \in [a, b].$$

The idea behind the fixed point iteration (FPI) is to start with an initial guess $x^{(0)}$ and to iterate according to the rule

$$x^{(k)} = \Phi \left(x^{(k-1)} \right).$$

The condition $L < 1$ guarantees that if Φ has a fixed point x^* , i.e. there exists a point x^* such that $\Phi(x^*) = x^*$, then the FPI will converge to x^* .

Lemma 8.1.1 (Convergence of FPI). *Let $\Phi : [a, b] \rightarrow [a, b]$ be a contractive mapping and let $x^* \in [a, b]$ be a fixed point of Φ , i.e. $\Phi(x^*) = x^*$. Then, the FPI*

$$x^{(k)} = \Phi \left(x^{(k-1)} \right)$$

with initial guess $x^{(0)} \in [a, b]$ converges to x^ .*

Proof. We need to show that $|e^{(k)}| \rightarrow 0$ as $k \rightarrow \infty$. Consider

$$|e^{(k)}| = |x^{(k)} - x^*| = |\Phi(x^{(k-1)}) - \Phi(x^*)|,$$

where we used $\Phi(x^*) = x^*$ and the definition of the FPI. As Φ is a contractive mapping and maps $[a, b]$ into itself, it follows that

$$|e^{(k)}| \leq L \cdot |x^{(k-1)} - x^*| = L \cdot |e^{(k-1)}|.$$

Iteratively, we find that

$$|e^{(k)}| \leq L^k \cdot |e^{(0)}|.$$

Thus $|e^{(k)}|$ tends to zero as $k \rightarrow \infty$ since $L < 1$. □

Note. We can make the following remarks about Lemma 8.1.1:

- It would suffice to have a Lipschitz continuous Φ with $L < 1$ on $[x^* - \delta, x^* + \delta]$ and initial guess $x^{(0)} \in [x^* - \delta, x^* + \delta]$ to deduce convergence.
- If $\Phi \in C^1([a, b])$ and $|\Phi'(x^*)| < 1$, we have convergence of the fixed point iteration in a neighborhood of x^* . This is because there are $\delta > 0, \epsilon > 0$ such that $|\Phi'(x)| < 1 - \epsilon$, for all $x \in [x^* - \delta, x^* + \delta] =: I^*$. Hence the mean value theorem implies that for all $x, y \in I^*$, there is a $\theta \in [x, y]$ such that

$$|\Phi(x) - \Phi(y)| = |\Phi'(\theta)| \cdot |x - y| < (1 - \epsilon) |x - y|.$$

Thus, Φ is Lipschitz continuous with $L = 1 - \epsilon < 1$ on I^* .

- The convergence rate of an FPI is at least *linear*.

Definition 8.1.4 (Linear Convergence). A sequence $(\mathbf{x}^{(k)})_{k \in \mathbb{N}}$ in \mathbb{R}^n converges linearly to $\mathbf{x}^* \in \mathbb{R}^n$ if

$$\exists L \in (0, 1) : \quad \|\mathbf{x}^{(k+1)} - \mathbf{x}^*\| \leq L \cdot \|\mathbf{x}^{(k)} - \mathbf{x}^*\| \quad \forall k \in \mathbb{N}_0.$$

Note. The bisection method was not linear, only "linear-type". Recall that we had shown $|e^{(k)}| \leq L^k |a - b|$. Note, however, that $|e^{(k)}| \geq L |e^{(k-1)}|$ is possible.

- If $\Phi'(x^*) = 0$ and $\Phi \in C^2$ in a neighborhood of x^* , then, the FPI converges quadratically to x^* . To see this, we consider the Taylor expansion of Φ at $x^{(k)}$ around x^* :

$$\Phi(x^{(k)}) = \Phi(x^*) + \underbrace{e^{(k)} \cdot \Phi'(x^*)}_{=0} + \frac{1}{2}(e^{(k)})^2 \Phi''(x^*) + \mathcal{O}((e^{(k)})^3).$$

Thereby, we obtain

$$|e^{(k)}| = |x^{(k)} - x^*| = |\Phi(x^{(k-1)}) - \Phi(x^*)| = \frac{1}{2}(e^{(k-1)})^2 \Phi''(x^*) + \mathcal{O}((e^{(k-1)})^3).$$

Now, let $k \in \mathbb{N}$ be sufficiently large such that $|x^{(k-1)} - x^*| \leq \delta < 1$ (this is guaranteed by the convergence of the FPI). Then, we get

$$|e^{(k)}| \leq \frac{1}{2} |e^{(k-1)}|^2 (|\Phi''(x^*)| + \mathcal{O}(\delta))$$

and the FPI converges quadratically.

Definition 8.1.5 (Order of Convergence). A **convergent** sequence $(\mathbf{x}^{(k)})_{k \in \mathbb{N}}$ in \mathbb{R}^n with limit $\mathbf{x}^* \in \mathbb{R}^n$ converges with *order* p if

$$\exists C > 0 : \quad \|\mathbf{x}^{(k+1)} - \mathbf{x}^*\| \leq C \cdot \|\mathbf{x}^{(k)} - \mathbf{x}^*\|^p \quad \forall k \in \mathbb{N},$$

and, in case of $p = 1$, an additional constraint of $C < 1$ is necessary (see Definition 8.1.4).

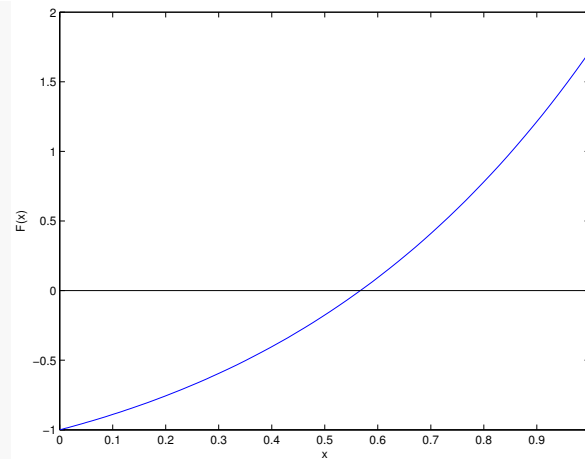
Thus, a higher order p implies faster convergence (and fewer iterations are needed to reach the same accuracy).

Note. If Φ is not globally Lipschitz continuous (on $[a, b]$), but only locally (e.g. $\Phi \in C^1, |\Phi'(x^*)| < 1$), then the fixed point iteration is also only locally convergent (i.e. $x^{(0)}$ has to be sufficiently close to x^*).

Example 8.1.1: Fixed Point Iteration

Let

$$F(x) = xe^x - 1, \quad x \in [0, 1].$$

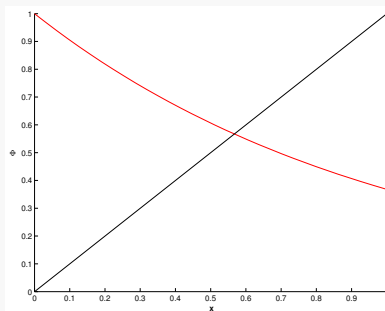


Consider three different fixed point forms

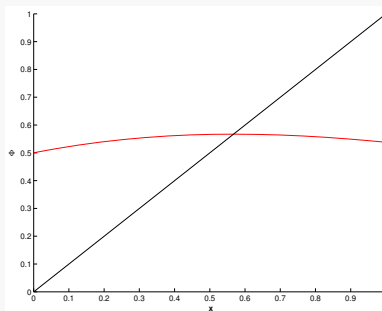
$$\Phi_1(x) = e^{-x},$$

$$\Phi_2(x) = \frac{1+x}{1+e^x},$$

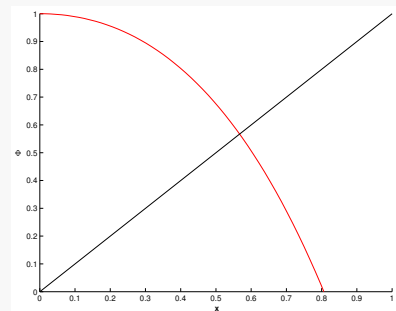
$$\Phi_3(x) = x + 1 - xe^x.$$



Function Φ_1



Function Φ_2



Function Φ_3

We tabulate the iteration error of the FPI's designed above with initial guess $x^{(0)} = 0.5$ and mark correct digits with red:

k	$ x_1^{(k+1)} - x^* $	$ x_2^{(k+1)} - x^* $	$ x_3^{(k+1)} - x^* $
0	0.067143290409784	0.067143290409784	0.067143290409784
1	0.039387369302849	0.000832287212566	0.108496074240152
2	0.021904078517179	0.000000125374922	0.219330611898582
3	0.012559804468284	0.0000000000000003	0.288178118764323
4	0.007078662470882	0.0000000000000000	0.723649245792953
5	0.004028858567431	0.0000000000000000	0.410183132337935
6	0.002280343429460	0.0000000000000000	1.186907542305364
7	0.001294757160282	0.0000000000000000	0.146569797006362
8	0.000733837662863	0.0000000000000000	0.310516641279937
9	0.000416343852458	0.0000000000000000	0.357777386500765
10	0.000236077474313	0.0000000000000000	0.974565695952037

We observe that Φ_1 converges roughly linearly, Φ_2 converges roughly quadratically and Φ_3 does not converge at all. We will try to explain how this comes about in the following.

Let us start with Φ_1 . Clearly, $\Phi_1'(x) = -e^{-x}$ and therefore $|\Phi_1'(x)| < 1$, for $x \in [\delta, 1] = I_\delta^*$, where $\delta > 0$. Hence, the FPI with iteration function Φ_1 and initial guess $x^{(0)} = 0.5$ converges (at least) linearly.

Similarly, we have

$$\Phi_2'(x) = \frac{1 - xe^x}{(1 + e^x)^2}$$

and therefore

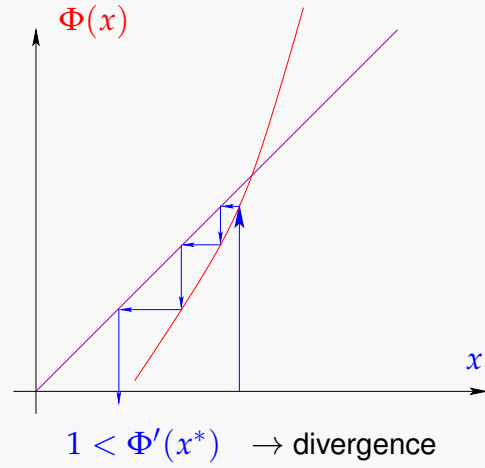
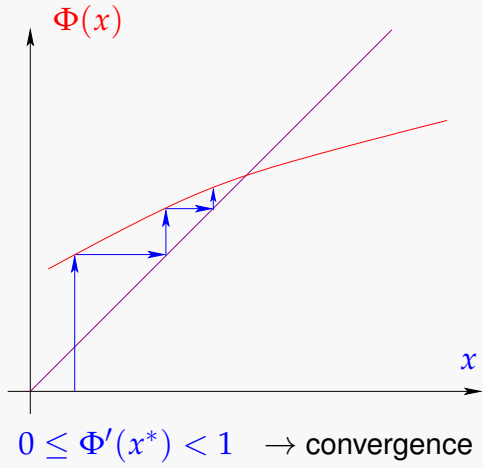
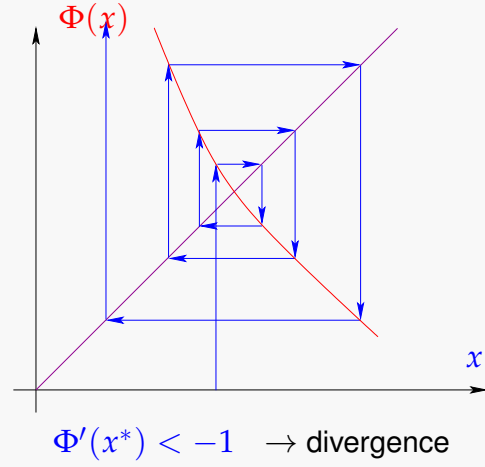
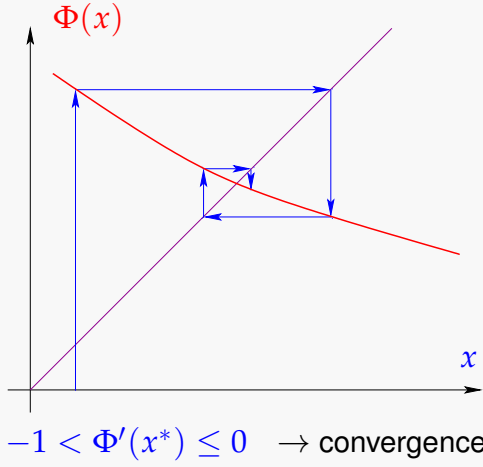
$$|\Phi_2'(x)| \leq \left| \max_{y \in [0,1]} \Phi_2'(y) \right| = \frac{|1 - e|}{4} < \frac{1}{2}, \quad x \in [0, 1] = I^*.$$

Hence, the FPI with iteration function Φ_2 and initial guess $x^{(0)} = 0.5$ converges (at least) linearly. In addition, one may see that $|\Phi_2''(x^*)| = 0$ and thus the FPI converges quadratically, in fact.

Finally, $\Phi_3'(x) = 1 - e^x - xe^x$ and thus $\Phi_3'(x^*) = -e^{x^*}$. In addition, we know that the fixed point satisfies $-e^{x^*} = -\frac{1}{x^*}$ whereby it follows that

$$|\Phi_3'(x^*)| = \left| -\frac{1}{x^*} \right| > 1, \quad \text{since } x^* \in (0, 1).$$

So, the iteration function Φ_3 is not contractive around x^* and thus the FPI cannot converge.



8.1.1 Algorithm for Root-Finding with Quadratic Convergence

In this subsection, we will assume $f \in C^1$. Therefore, we can use the first-order Taylor approximation around $x^{(k)}$:

$$f(x) \approx f(x^{(k)}) + (x - x^{(k)})f'(x^{(k)}).$$

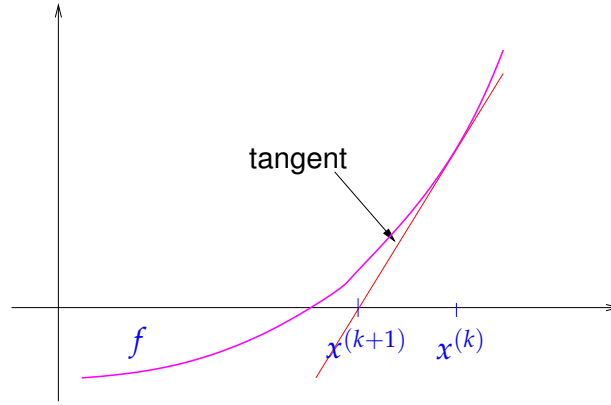
To find x^* such that $f(x^*) = 0$, we define the next iterate $x^{(k+1)}$ in such a way that

$$f(x^{(k)}) + (x^{(k+1)} - x^{(k)})f'(x^{(k)}) = 0.$$

In this way, we obtain *Newton's iteration*

$$x^{(k+1)} := x^{(k)} - \frac{f(x^{(k)})}{f'(x^{(k)})}, \quad (8.1)$$

which requires $f'(x^{(k)}) \neq 0$.



Do we have quadratic convergence when $f \in C^2$? To answer this question, we reformulate Newton's iteration as a fixed point iteration

$$\Phi(x) := x - \frac{f(x)}{f'(x)} \quad \text{and} \quad x^{(k+1)} = \Phi(x^{(k)}) = x^{(k)} - \frac{f(x^{(k)})}{f'(x^{(k)})}.$$

Thus, Newton's method for f is equivalent to a fixed point iteration with iteration function $\Phi(x)$. Let us look at the derivative of $\Phi(x)$. We have

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

It follows that if $f'(x^*) \neq 0$, then $\Phi'(x^*) = 0$ since $f(x^*) = 0$. Thus, as remarked after Definition 8.1.4, this yields local quadratic convergence in a neighborhood I^* of x^* .

Note.

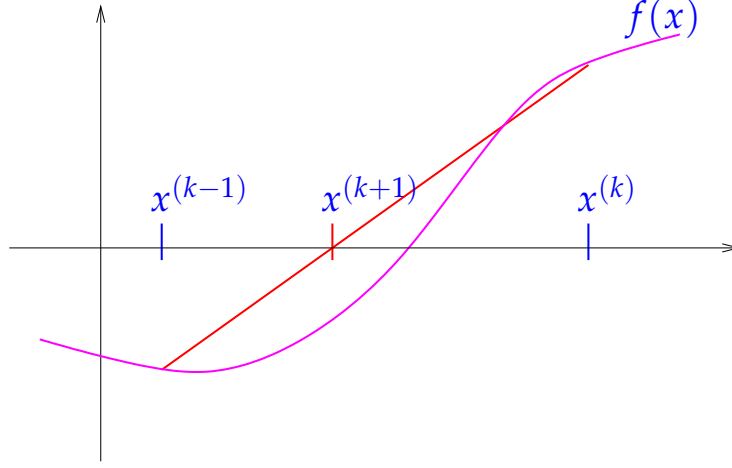
- If the iteration is started with $x^{(0)} \in I^*$, contractivity of Φ on I^* guarantees $f'(x^{(k)}) \neq 0$ for all the iterates. The FPI is ill-defined should this not be guaranteed.
- For quadratic convergence of Newton's method, it is sufficient to assume $f \in C^2(I^*)$. In particular, we do not need to assume $\Phi \in C^2(I^*)$. In summary, we need a neighborhood I^* of x^* such that
 - I^* is sufficiently small,
 - $f'(x) \neq 0$ on I^* ,
 - $f \in C^2(I^*)$.

8.1.2 Secant Method

Depending on the underlying problem, the computation of $f'(x^{(k)})$ in each Newton iteration could be costly. Thus, an alternative method might be to approximate $f'(x^{(k)})$ instead. The following realization of this idea is called the *secant method*:

$$f'(x^{(k)}) \approx \frac{f(x^{(k)}) - f(x^{(k-1)})}{x^{(k)} - x^{(k-1)}},$$

$$x^{(k+1)} := x^{(k)} - \frac{f(x^{(k)})(x^{(k)} - x^{(k-1)})}{f(x^{(k)}) - f(x^{(k-1)})}.$$



For the secant method, the iterate $x^{(k+1)}$ is obtained by approximating the derivative at $x^{(k)}$ through the secant between the points $x^{(k-1)}$ and $x^{(k)}$.

Note.

- The secant method requires no evaluation of derivatives.
- The secant method converges super-linearly but not quadratically. To be precise, its order is $p = \frac{1+\sqrt{5}}{2} \approx 1.618$.
- We call the secant method a *2-point method* as computing $x^{(k+1)}$ requires $x^{(k)}$ and $x^{(k-1)}$.

More generally, we have the following definition:

Definition 8.1.6 (*m*-Point Iteration). A stationary *m*-point ($m \in \mathbb{N}$) iterative method for $x^{(k)}$ depends on the *m* most recent iterates $x^{(k-1)}, \dots, x^{(k-m)}$ and is of the following form:

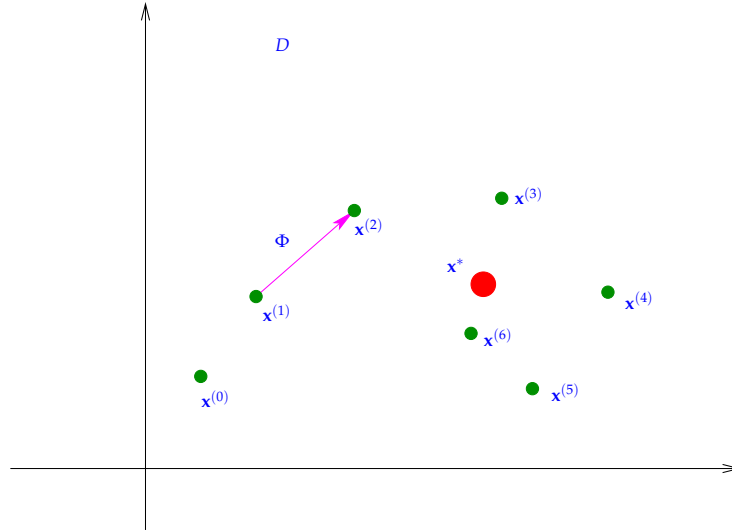
$$x^{(k)} = \Phi_F(x^{(k-1)}, \dots, x^{(k-m)}), \quad (8.2)$$

with iteration function Φ_F designed for solving $F(x) = 0$.

Note. An *m*-point method also requires *m* initial guesses $x^{(0)}, \dots, x^{(m-1)}$ in order to be described completely.

8.2 Nonlinear Systems of Equations

We will consider non-linear functions $F : D \subset \mathbb{R}^n \rightarrow \mathbb{R}^n$ describing non-linear systems of *n* equations for *n* unknowns. Our goal is to find a point $\mathbf{x}^* \in D$ such that $F(\mathbf{x}^*) = 0$.



We use the following general m -point iteration rule:

$$\mathbf{x}^{(k)} = \Phi_F(\mathbf{x}^{(k-1)}, \dots, \mathbf{x}^{(k-m)}),$$

with initial guess $\mathbf{x}^{(0)}$.

The following three aspects of such iteration rules are essential:

- *Convergence*: $(\mathbf{x}^{(k)})_{k \in \mathbb{N}}$ is said to be *convergent* if $\lim_{k \rightarrow \infty} \mathbf{x}^{(k)} = \mathbf{x}^*$.
- *Consistency*: Φ_F is said to be *consistent* if $\Phi_F(\mathbf{x}^*, \dots, \mathbf{x}^*) = \mathbf{x}^* \iff F(\mathbf{x}^*) = 0$.
- *Convergence rate*: rate at which $\|\mathbf{x}^{(k)} - \mathbf{x}^*\|$ decreases (see Definition 8.1.5).

For the convergence, we may work with any norm $\|\cdot\|$ on \mathbb{R}^n since all norms on \mathbb{R}^n (which is a finite dimensional vector space) are equivalent:

Definition 8.2.1 (Equivalence of Norms). Two norms $\|\cdot\|_a$ and $\|\cdot\|_b$ on a vector space V are equivalent if

$$\exists c_1, c_2 > 0 \text{ such that } \forall v \in V : \quad c_1 \cdot \|v\|_a \leq \|v\|_b \leq c_2 \cdot \|v\|_a.$$

This implies that the convergence in \mathbb{R}^n is independent of the choice of norm. But, in general, the convergence rate depends on the chosen norm. For the concept of convergence, we need to distinguish *local* vs. *global* convergence:

Definition 8.2.2 (Local and Global Convergence). A stationary m -point iterative method *converges locally* to $\mathbf{x}^* \in \mathbb{R}^n$ if there is a neighbourhood $U \subset D$ of \mathbf{x}^* such that $\mathbf{x}^{(0)}, \dots, \mathbf{x}^{(m-1)} \in U$ implies that $\mathbf{x}^{(k)}$ is well-defined and $\lim_{k \rightarrow \infty} \mathbf{x}^{(k)} = \mathbf{x}^*$, where $(\mathbf{x}^{(k)})_{k \in \mathbb{N}}$ denotes the (infinite) sequence of iterates.

If $U = D$, the iterative method is said to be *globally convergent*.

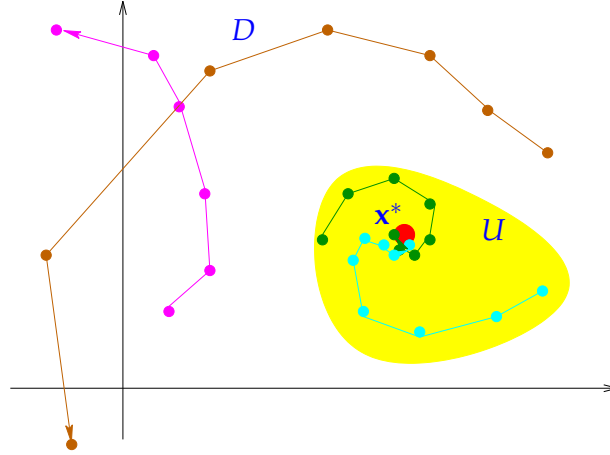


Illustration of local convergence in 2D. Only initial guesses that are “sufficiently close” to \mathbf{x}^* guarantee convergence. Unfortunately, the neighbourhood U is rarely known a-priori. It may also be very small.

8.2.1 Fixed Point Iterations in \mathbb{R}^n

Definition 8.2.3 (Contractive Mapping in \mathbb{R}^n). An iteration function $\Phi : U \subset \mathbb{R}^n \rightarrow \mathbb{R}^n$ is *contractive* (w.r.t. the norm $\|\cdot\|$ on \mathbb{R}^n) if

$$\exists L \in (0, 1) : \quad \|\Phi(x) - \Phi(y)\| \leq L \|x - y\| \quad \forall x, y \in U.$$

Note.

- $\Phi : U \rightarrow U$ being contractive implies that if $\Phi(\mathbf{x}^*) = \mathbf{x}^*$, then the fixed point iteration converges to \mathbf{x}^*

$$\underbrace{\|\mathbf{x}^{(k+1)} - \mathbf{x}^*\|}_{\|e^{(k+1)}\|} = \|\Phi(\mathbf{x}^{(k)}) - \Phi(\mathbf{x}^*)\| \leq \underbrace{L}_{<1} \cdot \underbrace{\|\mathbf{x}^{(k)} - \mathbf{x}^*\|}_{\|e^{(k)}\|}.$$

This implies

$$\|e^{(k+1)}\| \leq \underbrace{L^k}_{\rightarrow 0} \|e^{(0)}\|.$$

- The convergence is at least *linear*.
- When Φ is a contractive mapping, then Φ has at most one fixed point.

The last remark can be verified as follows: Suppose there are two fixed points x_1^*, x_2^* . Then, by the definition of a fixed point,

$$\|x_1^* - x_2^*\| = \|\Phi(x_1^*) - \Phi(x_2^*)\| \leq L \cdot \|x_1^* - x_2^*\|,$$

where $L < 1$ as Φ is a contractive mapping. Therefore,

$$(1 - L) \cdot \|x_1^* - x_2^*\| \leq 0,$$

which implies that $x_1^* = x_2^*$.

The following theorem (cf. course on Analysis) guarantees the existence of a fixed point under certain conditions on Φ :

Theorem 8.2.1 (Banach's Fixed Point Theorem). *If $D \subset \mathbb{K}^n$ ($\mathbb{K} = \mathbb{R}, \mathbb{C}$) is closed and bounded and $\Phi : D \rightarrow D$ satisfies*

$$\exists L < 1 : \quad \|\Phi(x) - \Phi(y)\| \leq L \cdot \|x - y\| \quad \forall x, y \in D,$$

then there is a unique fixed point $\mathbf{x}^ \in D$, $\Phi(\mathbf{x}^*) = \mathbf{x}^*$, which is the limit of the sequence of iterates $\mathbf{x}^{(k+1)} := \Phi(\mathbf{x}^{(k)})$, for any $\mathbf{x}^{(0)} \in D$.*

Next, we consider local convergence criteria for differentiable iteration functions Φ .

Lemma 8.2.1 (Sufficient condition for Local Linear Convergence of FPI). *If $\Phi : U \subset \mathbb{R}^n \rightarrow \mathbb{R}^n$, $\Phi(\mathbf{x}^*) = \mathbf{x}^*$, Φ differentiable in \mathbf{x}^* , and $\|D\Phi(\mathbf{x}^*)\| < 1$, then the fixed point iteration*

$$\mathbf{x}^{(k+1)} := \Phi(\mathbf{x}^{(k)})$$

converges locally and at least linearly.

Here $D\Phi(\mathbf{x})$ is the Jacobi matrix of Φ at $\mathbf{x} \in D$.

$$D\Phi(\mathbf{x}) := \left[\frac{\partial \Phi_i}{\partial x_j}(\mathbf{x}) \right]_{i,j=1}^n = \begin{bmatrix} \frac{\partial \Phi_1}{\partial x_1}(\mathbf{x}) & \frac{\partial \Phi_1}{\partial x_2}(\mathbf{x}) & \cdots & \cdots & \frac{\partial \Phi_1}{\partial x_n}(\mathbf{x}) \\ \frac{\partial \Phi_2}{\partial x_1}(\mathbf{x}) & \frac{\partial \Phi_2}{\partial x_2}(\mathbf{x}) & & & \frac{\partial \Phi_2}{\partial x_n}(\mathbf{x}) \\ \vdots & \vdots & & & \vdots \\ \frac{\partial \Phi_n}{\partial x_1}(\mathbf{x}) & \frac{\partial \Phi_n}{\partial x_2}(\mathbf{x}) & \cdots & \cdots & \frac{\partial \Phi_n}{\partial x_n}(\mathbf{x}) \end{bmatrix}. \quad (8.3)$$

We can furthermore use the information contained in the Jacobian for the following Lemma:

Lemma 8.2.2 (Sufficient condition for Linear Convergence of FPI). *Let U be convex and $\Phi : U \subset \mathbb{R}^n \rightarrow \mathbb{R}^n$ be continuously differentiable with*

$$L := \sup_{\mathbf{x} \in U} \|D\Phi(\mathbf{x})\| < 1.$$

If $\Phi(\mathbf{x}^) = \mathbf{x}^*$ for some interior point $\mathbf{x}^* \in U$, then the fixed point iteration $\mathbf{x}^{(k+1)} = \Phi(\mathbf{x}^{(k)})$ converges to \mathbf{x}^* at least linearly with rate L .*

Stopping Criterion for Contractive FPIs

A *stopping criterion*, as the name implies, determines when we stop the fixed point iteration Φ for solving $F(\mathbf{x}^*) = 0$.

A stopping criterion for a convergent iteration is deemed *reliable* if it lets the iteration *continue* until the iteration error $\mathbf{e}^{(k)} := \mathbf{x}^{(k)} - \mathbf{x}^*$ satisfies certain conditions (usually imposed before the start of the iteration). We will mostly consider the following two conditions:

$$\left\| \mathbf{x}^{(k)} - \mathbf{x}^* \right\| \leq \tau_{\text{abs}}, \quad \tau_{\text{abs}} \triangleq \text{prescribed (absolute) tolerance.}$$

or

$$\left\| \mathbf{x}^{(k)} - \mathbf{x}^* \right\| \leq \tau_{\text{rel}} \left\| \mathbf{x}^* \right\|, \quad \tau_{\text{rel}} \triangleq \text{prescribed (relative) tolerance.}$$

Note that the second condition can be problematic for some iterations. For example, when $\left\| \mathbf{x}^* \right\| = 0$, it is possible that the second condition is never satisfied and the algorithm never terminates. Thus, the “ideal” stopping rule is obtained by combining both conditions, that is, we terminate when

$$\left\| \mathbf{x}^{(k)} - \mathbf{x}^* \right\| \leq \begin{cases} \tau_{\text{abs}}, & \text{or} \\ \tau_{\text{rel}} \left\| \mathbf{x}^* \right\| \end{cases} . \quad (8.4)$$

Obviously, the optimal termination (8.4) cannot be used since \mathbf{x}^* is unknown. Hence, we construct a stopping criterion by choosing an approximation for \mathbf{x}^* . We introduce two choices, namely, ① residual based and ② correction based termination.

① *Residual based* termination: Stop convergent iteration $\{\mathbf{x}^{(k)}\}_{k \in \mathbb{N}_0}$ when

$$\left\| F(\mathbf{x}^{(k)}) \right\| \leq \tau, \quad \tau \triangleq \text{prescribed tolerance} > 0.$$

② *Correction based* termination: Stop convergent iteration $\{\mathbf{x}^{(k)}\}_{k \in \mathbb{N}_0}$ when

$$\left\| \mathbf{x}^{(k+1)} - \mathbf{x}^{(k)} \right\| \leq \begin{cases} \tau_{\text{abs}}, & \text{or} \\ \tau_{\text{rel}} \left\| \mathbf{x}^{(k+1)} \right\| \end{cases}$$

where $\tau_{\text{abs}} > 0$ and $\tau_{\text{rel}} > 0$ are the absolute and relative tolerances respectively.

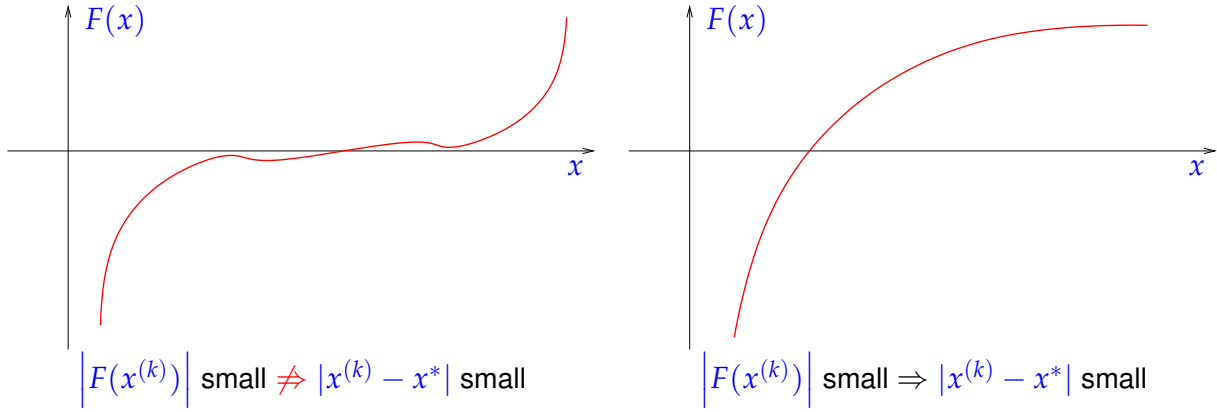
Recall from the discussion about the condition number that

$$\left\| F(\mathbf{x}^{(k)}) - \underbrace{F(\mathbf{x}^*)}_{=0} \right\|,$$

which is computable, being small does not necessarily imply that

$$\left\| \mathbf{x}^{(k)} - \mathbf{x}^* \right\|,$$

which is not computable, is small as well. Consider the following examples:



Our ultimate goal is a guarantee of the form: $\|\mathbf{x}^{(k)} - \mathbf{x}^*\| \leq \tau$. Suppose the iteration is linearly convergent. Then, we may compute

$$\|\mathbf{x}^{(k)} - \mathbf{x}^*\| \stackrel{\Delta\text{-ineq.}}{\leq} \|\mathbf{x}^{(k)} - \mathbf{x}^{(k+1)}\| + \|\mathbf{x}^{(k+1)} - \mathbf{x}^*\| \leq \|\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}\| + L \|\mathbf{x}^{(k)} - \mathbf{x}^*\|,$$

which implies

$$(1 - L) \cdot \|\mathbf{x}^{(k)} - \mathbf{x}^*\| \leq \|\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}\|. \quad (8.5)$$

Hence, we can conclude that linearly converging iterates satisfy:

$$\|\mathbf{x}^{(k+1)} - \mathbf{x}^*\| \underset{\substack{\leq \\ \uparrow \\ \text{contractivity}}}{\leq} L \cdot \|\mathbf{x}^{(k)} - \mathbf{x}^*\| \underset{\substack{\leq \\ \uparrow \\ (8.5)}}{\leq} \frac{L}{1 - L} \|\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}\|. \quad (8.6)$$

Thereby, we have found an upper bound on something not computable $(\|\mathbf{x}^{(k+1)} - \mathbf{x}^*\|)$ in terms of something we can compute $(\|\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}\|)$.

Based on this estimate, we may design the *a-posteriori* stopping criterion

$$\frac{L}{1 - L} \|\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}\| \leq \tau$$

which guarantees $\|\mathbf{x}^{(k)} - \mathbf{x}^*\| \leq \tau$. Note that estimating L directly can be difficult. However, if a more pessimistic estimate $\tilde{L} > L$ is easier to find, it can also serve as a suitable bound.

In a similar fashion, one can also obtain a more general estimate as follows:

$$\begin{aligned}
\left\| \mathbf{x}^{(k+m)} - \mathbf{x}^{(k)} \right\| &\stackrel{\Delta\text{-ineq.}}{\leq} \sum_{j=k}^{k+m-1} \left\| \mathbf{x}^{(j+1)} - \mathbf{x}^{(j)} \right\| \\
&\leq \sum_{j=k}^{k+m-1} L^{j-k} \left\| \mathbf{x}^{(k+1)} - \mathbf{x}^{(k)} \right\| \\
&= \frac{1-L^m}{1-L} \left\| \mathbf{x}^{(k+1)} - \mathbf{x}^{(k)} \right\| \\
&\leq \frac{1-L^m}{1-L} L^{k-l} \left\| \mathbf{x}^{(l+1)} - \mathbf{x}^{(l)} \right\| \quad \text{for some } l < k.
\end{aligned}$$

Note that for $m \rightarrow \infty$, $L^m \rightarrow 0$. With $\mathbf{x}^* := \lim_{k \rightarrow \infty} \mathbf{x}^{(k)}$, we find the estimate

$$\left\| \mathbf{x}^* - \mathbf{x}^{(k)} \right\| \leq \frac{L^{k-l}}{1-L} \left\| \mathbf{x}^{(l+1)} - \mathbf{x}^{(l)} \right\|. \quad (8.7)$$

Hence, we can obtain an *a-priori* stopping criterion by setting $l = 0$ in (8.7):

$$\left\| \mathbf{x}^* - \mathbf{x}^{(k)} \right\| \leq \frac{L^k}{1-L} \left\| \mathbf{x}^{(1)} - \mathbf{x}^{(0)} \right\| \quad (8.8)$$

Note that using a pessimistic value for L in (8.8) will result in a bound that is not even near as optimal as the original bound (if $k \gg 1$). Then the stopping criterion (8.8) will terminate the iteration long after the accuracy requirements were first met. This will thwart the efficiency of the method.

8.2.2 Newton's Method in Higher Dimensions

We extend the idea of Section 8.1.1 from the one dimensional case to higher dimensions. The first order approximation of \mathbf{F} around $\mathbf{x}^{(k)}$ is

$$\mathbf{F}(\mathbf{x}) \approx \mathbf{F}(\mathbf{x}^{(k)}) + \underbrace{\mathbf{D}\mathbf{F}(\mathbf{x}^{(k)})}_{\substack{\in \mathbb{R}^{n,n} \\ \text{Jacobian of } \mathbf{F} \text{ at } \mathbf{x}^{(k)}}} (\mathbf{x} - \mathbf{x}^{(k)}) =: \mathbf{F}_k(\mathbf{x}).$$

Definition 8.2.4 (Newton's Method in \mathbb{R}^n). *Newton's iteration* in \mathbb{R}^n may be defined by the recursive rule

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \mathbf{D}\mathbf{F}(\mathbf{x}^{(k)})^{-1} \mathbf{F}(\mathbf{x}^{(k)}), \quad (8.9)$$

if $\mathbf{D}\mathbf{F}(\mathbf{x}^{(k)})$ is regular. We call $-\mathbf{D}\mathbf{F}(\mathbf{x}^{(k)})^{-1} \mathbf{F}(\mathbf{x}^{(k)})$ the *Newton correction term*.

Recall that for the one-dimensional Newton's method, we required that $f'(x^{(k)}) \neq 0$. In higher dimensions, $\mathbf{D}\mathbf{F}(\mathbf{x}^{(k)})$ has to be invertible so that one can solve the LSE

$$\mathbf{D}\mathbf{F}(\mathbf{x}^{(k)}) \mathbf{y} = -\mathbf{F}(\mathbf{x}^{(k)})$$

to obtain the Newton correction term \mathbf{y} . Thereafter, we may update $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \mathbf{y}$.

The analysis of convergence of Newton's method in higher dimensions is much more complicated than one might think. Consider the following theorem.

Theorem 8.2.2 (Local Quadratic Convergence of Newton's Method). *Suppose that the following conditions hold:*

- (A) $D \subset \mathbb{R}^n$ is open and convex.
- (B) $\mathbf{F} : D \rightarrow \mathbb{R}^n$ is continuously differentiable.
- (C) $D\mathbf{F}(\mathbf{x})$ is regular for all $\mathbf{x} \in D$.
- (D) $\exists L \geq 0$ for all $\mathbf{v} \in \mathbb{R}^n$, $\mathbf{x} \in D$, $\mathbf{x} + \mathbf{v} \in D$ such that $\|D\mathbf{F}(\mathbf{x})^{-1}(D\mathbf{F}(\mathbf{x} + \mathbf{v}) - D\mathbf{F}(\mathbf{x}))\|_2 \leq L \cdot \|\mathbf{v}\|_2$.
- (E) $\exists \mathbf{x}^*$ such that $\mathbf{F}(\mathbf{x}^*) = 0$ (existence of a solution in D).
- (F) The initial guess $\mathbf{x}^{(0)} \in D$ satisfies $\rho := \|\mathbf{x}^* - \mathbf{x}^{(0)}\| < \frac{2}{L}$ and $B_\rho(\mathbf{x}^*) \subset D$.

Then, Newton's iteration (8.9) satisfies:

- (i) $\mathbf{x}^{(k)} \in B_\rho(\mathbf{x}^*)$ for all $k \in \mathbb{N}$.
- (ii) $\lim_{k \rightarrow \infty} \mathbf{x}^{(k)} = \mathbf{x}^*$.
- (iii) $\|\mathbf{x}^{(k+1)} - \mathbf{x}^*\|_2 \leq \frac{L}{2} \|\mathbf{x}^{(k)} - \mathbf{x}^*\|_2^2$ (local quadratic convergence).

Here we have used the notation: $B_\rho(\mathbf{x}^*) := \{\mathbf{y} \in \mathbb{R}^n, \|\mathbf{y} - \mathbf{x}^*\| < \rho\}$, which refers to a ball of radius ρ centered at \mathbf{x}^* .

Usually, it is hardly possible to verify the assumptions of the theorem for a concrete non-linear system of equations, because neither L nor \mathbf{x}^* are known.

However, we may condense this theorem down to the following result.

Note (Convergence of Newton's Method). If $\mathbf{F}(\mathbf{x}^*) = 0$ and $D\mathbf{F}(\mathbf{x}^*)$ is regular, then Newton's method is locally quadratic convergent.

Consider the following remaining questions:

1. What is a good stopping criterion for Newton's method?
2. Can we obtain convergence on a larger region? Maybe at the cost of losing quadratic convergence.
3. Computing the Newton correction term is costly as it involves solving an LSE in every iteration step whose system matrix also changes at every step. Is there a remedy to this problem?

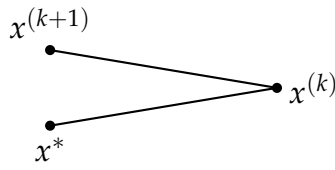
A Stopping Criterion for Newton's Method in \mathbb{R}^n

In Section 8.2.1, termination criteria for general FPIs have been discussed. We will now consider such criteria specifically for Newton's method. The intuition behind the stopping criterion we are about to derive is that one expects

$$\|\mathbf{x}^{(k+1)} - \mathbf{x}^*\| \ll \|\mathbf{x}^{(k)} - \mathbf{x}^*\|$$

due to the quadratic convergence of Newton's method. Therefore, we roughly have

$$\|\mathbf{x}^{(k)} - \mathbf{x}^*\| \approx \|\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}\|.$$



Thus, the computable stopping criterion

$$\|\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}\| = \|\mathbf{D}\mathbf{F}(\mathbf{x}^{(k)})^{-1}\mathbf{F}(\mathbf{x}^{(k)})\| \leq \tau \|\mathbf{x}^{(k)}\|$$

guarantees that

$$\|\mathbf{x}^{(k)} - \mathbf{x}^*\| \underset{\substack{\uparrow \\ (8.5)}}{\lesssim} \tau \|\mathbf{x}^{(k)}\|.$$

Note that this stopping criterion requires the computation of the Newton correction term $-\mathbf{D}\mathbf{F}(\mathbf{x}^{(k)})^{-1}\mathbf{F}(\mathbf{x}^{(k)})$ of the new iterate $\mathbf{x}^{(k+1)}$. Therefore, if $\mathbf{x}^{(k)}$ was a good approximation, we would have solved one LSE too many for verifying the stopping criterion. A cheaper stopping criterion would be

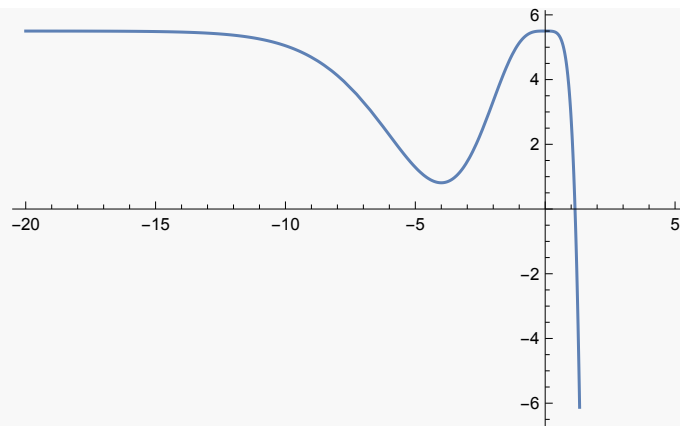
$$\|\mathbf{D}\mathbf{F}(\mathbf{x}^{(k-1)})^{-1}\mathbf{F}(\mathbf{x}^{(k)})\| \leq \tau \|\mathbf{x}^{(k)}\|,$$

where $-\mathbf{D}\mathbf{F}(\mathbf{x}^{(k-1)})^{-1}\mathbf{F}(\mathbf{x}^{(k)})$ is called *simplified Newton correction* term. The motivation behind this simplification is that due to the fast convergence of Newton's iteration, $\mathbf{D}\mathbf{F}(\mathbf{x}^{(k)}) \approx \mathbf{D}\mathbf{F}(\mathbf{x}^{(k-1)})$ during the last steps of the iteration. Note that we may then reuse the LU factorization of $\mathbf{D}\mathbf{F}(\mathbf{x}^{(k-1)})$ for the stopping criterion.

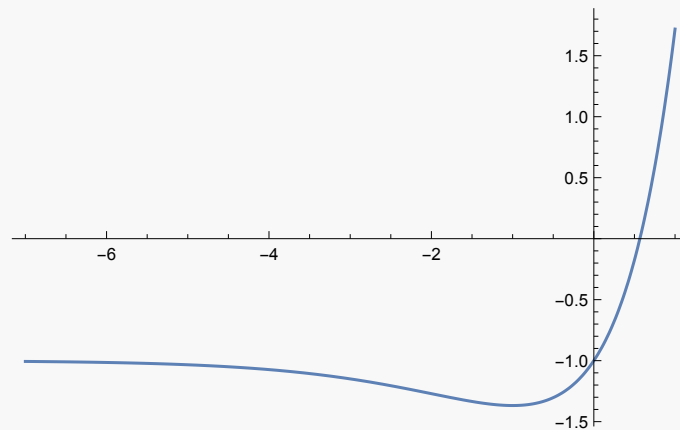
Example 8.2.1: Failures of Newton's Method

The following are cases in which Newton's method will fail unless one chooses the initial guess carefully.

1. Functions with local minima or maxima:

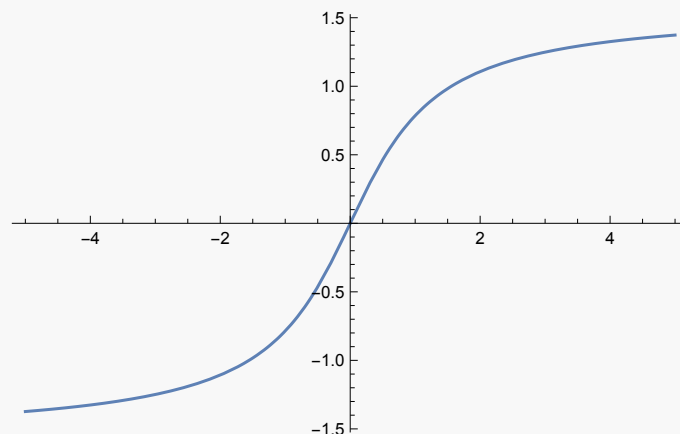


2. Functions with asymptotes such as $F(x) = xe^x - 1$:



For the aforementioned function, one may compute that $F'(-1) = 0$. In this way, one may show that, for $x^{(0)} < -1$, Newton's method diverges, that is, $\lim_{k \rightarrow \infty} x^{(k)} = -\infty$. For $x^{(0)} > -1$, Newton's method converges, that is, $\lim_{k \rightarrow \infty} x^{(k)} = x^*$.

3. Functions for which Newton's method is prone to overshoot its target such as $F(x) = \arctan(x)$:



The remedy for over/under-shooting is dampening, as we will see in the following subsection.

8.2.3 Damped Newton Method

Newton's method converges quadratically, but only locally, which may render it useless, if convergence is guaranteed only for initial guesses very close to the exact solution. Here, we study a method to enlarge the region of convergence, at the expense of quadratic convergence, of course. The idea behind the damped Newton method is to check in each iteration step whether the distance between $\mathbf{x}^{(k)}$ and $\mathbf{x}^{(k+1)}$ decreased by a factor of 2. Roughly speaking, we check whether

$$\left\| \mathbf{x}^{(k+2)} - \mathbf{x}^{(k+1)} \right\| \leq \frac{1}{2} \left\| \mathbf{x}^{(k+1)} - \mathbf{x}^{(k)} \right\|.$$

If this is not the case, we do not take the full Newton step and instead damp the Newton correction:

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \lambda^{(k)} \mathbf{D} \mathbf{F}(\mathbf{x}^{(k)})^{-1} \mathbf{F}(\mathbf{x}^{(k)}), \quad (8.10)$$

where $\lambda^{(k)} \in (0, 1]$ is called *damping factor*.

Our strategy for choosing the damping factor will be to take the largest $\lambda^{(k)}$ possible such that the distance between iterates decreases. A simple realization of this strategy is the *natural monotonicity test*.

Affine Invariant Damping Strategy

We present the affine invariant *natural monotonicity test* (NMT):

Choose the maximal $0 < \lambda^{(k)} \leq 1$ such that

$$\left\| \Delta \bar{\mathbf{x}}(\lambda^{(k)}) \right\| \leq \left(1 - \frac{\lambda^{(k)}}{2} \right) \cdot \left\| \Delta \mathbf{x}^{(k)} \right\|, \quad (8.11)$$

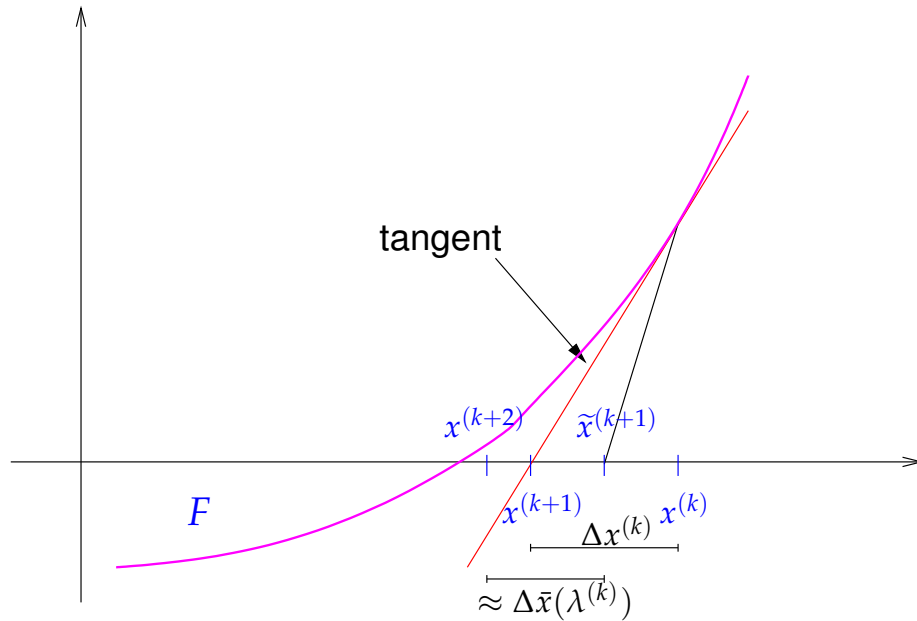
where $\Delta \mathbf{x}^{(k)} := -\mathbf{D} \mathbf{F}(\mathbf{x}^{(k)})^{-1} \mathbf{F}(\mathbf{x}^{(k)})$ denotes the current Newton correction and

$$\Delta \bar{\mathbf{x}}(\lambda^{(k)}) := \mathbf{D} \mathbf{F}(\mathbf{x}^{(k)})^{-1} \mathbf{F} \left(\mathbf{x}^{(k)} + \lambda^{(k)} \Delta \mathbf{x}^{(k)} \right)$$

is a tentative simplified Newton correction.

Note. First, note that $\Delta \mathbf{x}^{(k)}$ corresponds to $\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}$ without damping, $\mathbf{x}^{(k)} + \lambda^{(k)} \Delta \mathbf{x}^{(k)}$ is a tentative new iterate $\tilde{\mathbf{x}}^{(k+1)}$ with damping and thus $\Delta \bar{\mathbf{x}}(\lambda^{(k)})$ is an approximation for $\mathbf{x}^{(k+2)} - \tilde{\mathbf{x}}^{(k+1)}$. So in particular, the NMT checks whether $\left\| \Delta \bar{\mathbf{x}}(\lambda^{(k)}) \right\|$ is strictly smaller than $\left\| \mathbf{x}^{(k+1)} - \mathbf{x}^{(k)} \right\|$, where $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \Delta \mathbf{x}^{(k)}$.

In practice, we take $\lambda^{(k)} = 1$ and check the NMT (Equation (8.11)). We then repeatedly assign $\lambda^{(k)} \leftarrow \frac{\lambda^{(k)}}{2}$ until (8.11) is fulfilled for the first time. A lower bound λ_{\min} has to be specified for the termination of the NMT: if the test fails to find a $\lambda \geq \lambda_{\min}$ fulfilling the NMT, then the iteration is terminated and failure of the damped Newton method is reported.



A damped Newton step with $\lambda^{(k)} = \frac{1}{2}$.

Example 8.2.2: Damped Newton method

We test the damped Newton method for the third problem of Example 8.2.1, where $F(x) = \arctan(x)$ and we choose $x^{(0)} = 20$ and $\lambda_{\min} = 0.001$. The full Newton corrections had made Newton's method fail in this example.

k	$\lambda^{(k)}$	$x^{(k)}$	$F(x^{(k)})$
1	0.03125	0.94199967624205	0.75554074974604
2	0.06250	0.85287592931991	0.70616132170387
3	0.12500	0.70039827977515	0.61099321623952
4	0.25000	0.47271811131169	0.44158487422833
5	0.50000	0.20258686348037	0.19988168667351
6	1.00000	-0.00549825489514	-0.00549819949059
7	1.00000	0.00000011081045	0.00000011081045
8	1.00000	-0.00000000000001	-0.00000000000001

We observe that damping is effective and asymptotic quadratic convergence is recovered.

Example 8.2.3: Failure of damped Newton method

We examine the effect of damping in the case of the second problem of Example 8.2.1, where $F(x) = xe^x - 1$ and we choose $x^{(0)} = -1.5$ and $\lambda_{\min} = 0.001$.

k	$\lambda^{(k)}$	$x^{(k)}$	$F(x^{(k)})$
1	0.25000	-4.4908445351690	-1.0503476286303
2	0.06250	-6.1682249558799	-1.0129221310944
3	0.01562	-7.6300006580712	-1.0037055902301
4	0.00390	-8.8476436930246	-1.0012715832278
5	0.00195	-10.5815494437311	-1.0002685596314
Bailed out because of $\lambda < \lambda_{\min}$!			

We observe that the Newton correction is pointing in the “wrong direction”. Therefore, we obtain no convergence despite damping.

8.2.4 Quasi-Newton Method

At the beginning of Section 8.2.2, we formulated three questions to be addressed for Newton’s method. The last one of these was the question of computational cost: At each step, Newton’s method requires the solution of a LSE, with different system matrix at each step. We will now address how to design *approximate* Newton corrections, that can be obtained more efficiently.

Recall the secant method in one dimension (see Section 8.1.2) which made use of the approximation

$$f'(x^{(k)}) \approx \frac{f(x^{(k)}) - f(x^{(k-1)})}{x^{(k)} - x^{(k-1)}}.$$

Can we devise a similar (cheaper) approximate Newton method in the multidimensional case? Let us start with a higher dimensional analog to the above equation: an approximation $\mathbf{J}_k \in \mathbb{R}^{n \times n}$ of $D\mathbf{F}(\mathbf{x}^{(k)})$ such that

$$\mathbf{J}_k (\mathbf{x}^{(k)} - \mathbf{x}^{(k-1)}) = \mathbf{F}(\mathbf{x}^{(k)}) - \mathbf{F}(\mathbf{x}^{(k-1)}). \quad (8.12)$$

Performing an approximate Newton iteration step with \mathbf{J}_k yields

$$\mathbf{x}^{(k)} = \mathbf{x}^{(k-1)} - \mathbf{J}_{k-1}^{-1} \mathbf{F}(\mathbf{x}^{(k-1)}),$$

which is equivalent to

$$\mathbf{J}_{k-1} (\mathbf{x}^{(k)} - \mathbf{x}^{(k-1)}) = -\mathbf{F}(\mathbf{x}^{(k-1)}). \quad (8.13)$$

The equations (8.12) and (8.13) together yield the underdetermined linear system

$$(\mathbf{J}_k - \mathbf{J}_{k-1}) (\mathbf{x}^{(k)} - \mathbf{x}^{(k-1)}) = \mathbf{F}(\mathbf{x}^{(k)}).$$

A possible cheap choice for $\mathbf{J}_k - \mathbf{J}_{k-1}$ is

$$\mathbf{J}_k - \mathbf{J}_{k-1} = \frac{\mathbf{F}(\mathbf{x}^{(k)}) (\mathbf{x}^{(k)} - \mathbf{x}^{(k-1)})^\top}{\|\mathbf{x}^{(k)} - \mathbf{x}^{(k-1)}\|_2^2}.$$

This choice results in a rank-1 matrix for $\mathbf{J}_k - \mathbf{J}_{k-1}$. So given an initial \mathbf{J}_0 (take $\mathbf{J}_0 = \mathbf{D}\mathbf{F}(\mathbf{x}^{(0)})$), we can obtain \mathbf{J}_k , an approximation of $\mathbf{D}\mathbf{F}(\mathbf{x}^{(k)})$, by k subsequent rank-1 updates

$$\mathbf{J}_k = \mathbf{J}_{k-1} + \frac{\mathbf{F}(\mathbf{x}^{(k)}) (\mathbf{x}^{(k)} - \mathbf{x}^{(k-1)})^\top}{\|\mathbf{x}^{(k)} - \mathbf{x}^{(k-1)}\|_2^2}.$$

This choice results in *Broyden's quasi-Newton method* for solving $\mathbf{F}(\mathbf{x}) = 0$:

$$\begin{aligned} \mathbf{x}^{(k+1)} &:= \mathbf{x}^{(k)} + \Delta\mathbf{x}^{(k)}, \quad \Delta\mathbf{x}^{(k)} := -\mathbf{J}_k^{-1}\mathbf{F}(\mathbf{x}^{(k)}), \\ \mathbf{J}_{k+1} &:= \mathbf{J}_k + \frac{\mathbf{F}(\mathbf{x}^{(k+1)}) (\Delta\mathbf{x}^{(k)})^\top}{\|\Delta\mathbf{x}^{(k)}\|_2^2}. \end{aligned}$$

Note that we can use the Sherman-Morrison-Woodbury formula to calculate \mathbf{J}_{k+1}^{-1} from \mathbf{J}_k^{-1} :

$$\mathbf{J}_{k+1}^{-1} = \left(\mathbf{I} - \frac{\mathbf{J}_k^{-1}\mathbf{F}(\mathbf{x}^{(k+1)}) (\Delta\mathbf{x}^{(k)})^\top}{\|\Delta\mathbf{x}^{(k)}\|_2^2 + (\Delta\mathbf{x}^{(k)})^\top \mathbf{J}_k^{-1}\mathbf{F}(\mathbf{x}^{(k+1)})} \right) \mathbf{J}_k^{-1}$$

Note. We conclude the discussion on iterative methods for nonlinear systems with the remark that any such method should have a convergence monitor, i.e. a simple way to check in each iteration whether convergence is to be expected or not. An example of such a test is the natural monotonicity test for the damped Newton method. If the NMT fails repeatedly so that λ_{\min} is reached, we stop the iteration and report an error.

8.3 Unconstrained Optimization

Many problems in practical applications involve finding the optimum of some objective function, that is, finding a maximum or a minimum. This question is closely linked to the solution of nonlinear systems of equations, as we will see, although in its full generality, optimization is a much more general field and deserves its own treatment. However, with what we have seen in the previous discussions, we can derive iterative methods for solving simple (that is, convex unconstrained) optimization problems. We have already seen a few optimization problems in Chapters 2 and 3:

- Least-squares solution: Find an $\mathbf{x} \in \mathbb{K}^n$ such that $\|\mathbf{A}\mathbf{x} - \mathbf{b}\|_2 \rightarrow \min$.

- Generalized solution: Find a least-squares solution \mathbf{x} to $\mathbf{Ax} = \mathbf{b}$ such that $\|\mathbf{x}\|_2 \rightarrow \min$.
- Best low-rank approximation: Given $\mathbf{A} \in \mathbb{K}^{m,n}$, find $\tilde{\mathbf{A}} \in \mathbb{K}^{m,n}$, $\text{rank}(\tilde{\mathbf{A}}) \leq k$, such that $\|\mathbf{A} - \tilde{\mathbf{A}}\|_{2/F} \rightarrow \min$ over the set of rank- k matrices.

General Problem Formulation

Given a function $F : \mathbb{R}^n \rightarrow \mathbb{R}$, how do we find a minima/maxima of F ?

Example 8.3.1:

We first look at an application from machine learning which is the maximum likelihood estimation. Suppose some quantity can be modelled with a probability distribution. For example, the weight of 5-year olds in Switzerland (which is modelled by a normal distribution). Can we estimate the mean μ and variance σ through randomized samples?

Consider samples $\{w_1, \dots, w_n\}$. The normal distribution has the density function

$$f(w; \mu, \sigma) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(w-\mu)^2}{2\sigma^2}},$$

i.e. $f(w_i; \mu, \sigma)$ describes the likelihood to observe weight w_i . Let us assume that the weight of the children are independent and so the probability to observe our samples is

$$\mathbb{P}[\{w_1, \dots, w_n\}; \mu, \sigma] = \prod_{i=1}^n f(w_i; \mu, \sigma).$$

Note that we view this probability as a function of μ and σ , while the samples $\{w_1, \dots, w_n\}$ are fixed. To estimate the model parameters μ and σ , we may maximize the likelihood of observing $\{w_1, \dots, w_n\}$ by maximizing \mathbb{P} . A common trick to use is that this is, in fact, equivalent to maximizing $\log \mathbb{P}$ (to be precise, the logarithm of \mathbb{P} has the same location of the maximum but better numerical properties).

Note that maximizing an objective F is equivalent to minimizing $-F$. Hence, it suffices to only consider minimization problems. Finally, we need to distinguish between *local* and *global minima*:

- We call \mathbf{x}^* a *global minimum* of $F : \mathbb{R}^n \rightarrow \mathbb{R}$ if $F(\mathbf{x}^*) \leq F(\mathbf{x})$, for all $\mathbf{x} \in \mathbb{R}^n$.
- We call \mathbf{x}^* a *local minimum* of $F : \mathbb{R}^n \rightarrow \mathbb{R}$ if there is an $\epsilon > 0$ such that for all $\mathbf{x} \in \mathbb{R}^n$ with $\|\mathbf{x} - \mathbf{x}^*\| \leq \epsilon$ it holds that $F(\mathbf{x}^*) \leq F(\mathbf{x})$.

We call

$$B_\epsilon(\mathbf{x}^*) := \{\mathbf{x} \in \mathbb{R}^n : \|\mathbf{x} - \mathbf{x}^*\| \leq \epsilon\}$$

an ϵ -ball around \mathbf{x}^* .

8.3.1 Optimization with a Differentiable Objective Function

When the function $F : \mathbb{R}^n \rightarrow \mathbb{R}$ is differentiable, ∇F indicates the direction of greatest increase and $-\nabla F$ the direction of steepest descent. One can see this when considering the first-order Taylor approximation of F around $\bar{\mathbf{x}}$ given by

$$F(\mathbf{x}) \approx F(\bar{\mathbf{x}}) + \nabla F(\bar{\mathbf{x}})^\top (\mathbf{x} - \bar{\mathbf{x}}).$$

Taking \mathbf{x} as a (sufficiently close) update of $\bar{\mathbf{x}}$ in the direction of the gradient ∇F , we obtain

$$F(\bar{\mathbf{x}} + \tau \nabla F(\bar{\mathbf{x}})) \approx F(\bar{\mathbf{x}}) + \tau \|\nabla F(\bar{\mathbf{x}})\|^2,$$

and witness an increase of F whenever $\tau > 0$ and a decrease of F whenever $\tau < 0$. We call \mathbf{x} a *stationary point* of F if $\nabla F(\mathbf{x}) = 0$. Stationary points can be local/global maxima/minima or saddle points of F . If F is twice differentiable, we can use its *Hessian matrix* at the stationary point to identify which one is the case:

$$\mathbf{H}_F(\mathbf{x}) := \left(\frac{\partial^2 F(\mathbf{x})}{\partial x_i \partial x_j} \right)_{i,j=1}^n.$$

This can be seen by considering the second-order Taylor expansion around $\bar{\mathbf{x}}$ given by

$$F(\mathbf{x}) \approx F(\bar{\mathbf{x}}) + \nabla F(\bar{\mathbf{x}})^\top (\mathbf{x} - \bar{\mathbf{x}}) + \frac{1}{2}(\mathbf{x} - \bar{\mathbf{x}})^\top \mathbf{H}_F(\bar{\mathbf{x}})(\mathbf{x} - \bar{\mathbf{x}}).$$

For a stationary point $\bar{\mathbf{x}}$, we have $\nabla F(\bar{\mathbf{x}}) = 0$ and therefore

$$F(\mathbf{x}) \approx F(\bar{\mathbf{x}}) + \frac{1}{2}(\mathbf{x} - \bar{\mathbf{x}})^\top \mathbf{H}_F(\bar{\mathbf{x}})(\mathbf{x} - \bar{\mathbf{x}}).$$

Suppose that $\mathbf{H}_F(\bar{\mathbf{x}})$ is positive definite. Then, it follows that

$$(\mathbf{x} - \bar{\mathbf{x}})^\top \mathbf{H}_F(\bar{\mathbf{x}})(\mathbf{x} - \bar{\mathbf{x}}) > 0$$

and therefore there exists an $\epsilon > 0$ such that for all $\mathbf{x} \in B_\epsilon(\bar{\mathbf{x}})$, we have that $F(\mathbf{x}) \geq F(\bar{\mathbf{x}})$. Therefore, $\bar{\mathbf{x}}$ is a local minimum of F .

Similarly, one may show that if $\mathbf{H}_F(\bar{\mathbf{x}})$ is negative definite, then $\bar{\mathbf{x}}$ is a local maximum of F , and that if $\mathbf{H}_F(\bar{\mathbf{x}})$ is indefinite, then $\bar{\mathbf{x}}$ is a saddle point of F . Additionally, if $\mathbf{H}_F(\bar{\mathbf{x}})$ is not invertible, then there is a whole region of saddle points. One can check the positive definiteness of the Hessian, for instance, by checking whether a Cholesky factorization exists.

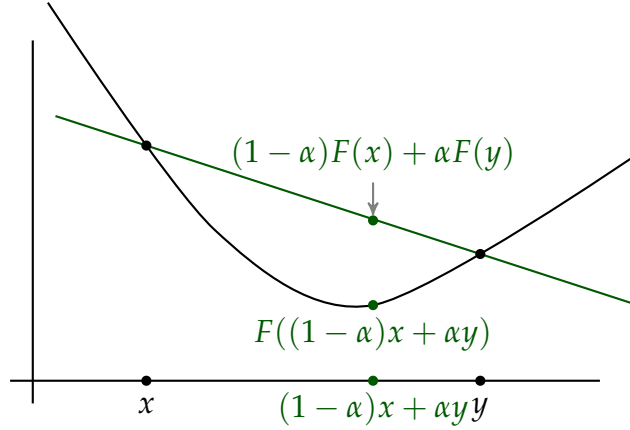
8.3.2 Optimization with a Convex Objective Function

As already hinted at in the beginning of this chapter, general optimization problems can be difficult to solve. However, for minimization problems with *convex* objective functions, the theory is rather straightforward.

Definition 8.3.1 (Convex Function). A function $F : \mathbb{R}^n \rightarrow \mathbb{R}$ is called *convex* if for all $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$, and for all $\alpha \in (0, 1)$, the following holds:

$$F((1 - \alpha)\mathbf{x} + \alpha\mathbf{y}) \leq (1 - \alpha)F(\mathbf{x}) + \alpha F(\mathbf{y}). \quad (8.14)$$

If the inequality holds in a strict sense, then the function is strictly convex.



Lemma 8.3.1 (Minimum of Convex Function). If $\bar{\mathbf{x}} \in \mathbb{R}^n$ is a local minimum of $F : \mathbb{R}^n \rightarrow \mathbb{R}$, then it is a global minimum.

Proof. We prove the claim by contradiction: Let $\bar{\mathbf{x}}$ be a local minimum of F , but not its global minimum. This implies that there exists an $\mathbf{x}_0 \in \mathbb{R}^n$ such that $F(\mathbf{x}_0) < F(\bar{\mathbf{x}})$. For $\alpha \in (0, 1)$, convexity implies

$$F(\underbrace{\bar{\mathbf{x}} + \alpha(\mathbf{x}_0 - \bar{\mathbf{x}})}_{=\alpha\mathbf{x}_0 + (1-\alpha)\bar{\mathbf{x}}}) \leq (1 - \alpha)F(\bar{\mathbf{x}}) + \underbrace{\alpha F(\mathbf{x}_0)}_{< F(\bar{\mathbf{x}})} < F(\bar{\mathbf{x}}).$$

We construct a sequence $\alpha_k \rightarrow 0$ and consider $(1 - \alpha_k)F(\bar{\mathbf{x}}) + \alpha_k F(\mathbf{x}_0) < F(\bar{\mathbf{x}})$. By construction, for every $\epsilon > 0$, we can find a $k \in \mathbb{N}$ such that $\|\mathbf{x}_k - \bar{\mathbf{x}}\| < \epsilon$, where $\mathbf{x}_k := \alpha_k \mathbf{x}_0 + (1 - \alpha_k)\bar{\mathbf{x}}$. However, for this sequence \mathbf{x}_k approaching $\bar{\mathbf{x}}$, one has $F(\mathbf{x}_k) < F(\bar{\mathbf{x}})$. Hence, $\bar{\mathbf{x}}$ cannot be a local minimum, which contradicts our assumption. \square

8.3.3 Optimization Algorithms

Gradient Descent

We had seen that for a differentiable function $F : \mathbb{R}^n \rightarrow \mathbb{R}$, its negative gradient $\Delta \mathbf{x} = -\nabla F(\mathbf{x})$ points in the direction of steepest descent. This provides us the guarantee that if $\nabla F(\mathbf{x}) \neq 0$ and $\alpha > 0$ is sufficiently small, then $F(\mathbf{x} - \alpha \nabla F(\mathbf{x})) \leq F(\mathbf{x})$. Based on this insight, we define the *gradient descent iteration* by

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - t^{(k)} \nabla F(\mathbf{x}^{(k)}),$$

where $t^{(k)} > 0$ is a step-size whose determination will be a 1D problem.

In each iteration $F(\mathbf{x}^{(k)})$ decreases and hence we terminate the search when $\nabla F(\mathbf{x}^{(k)}) \approx 0$. There are different methods to determine the step sizes $t^{(k)}$:

- *Exact line search*: $t^* = \operatorname{argmin}_{t \geq 0} F(\mathbf{x}^{(k)} - t \nabla F(\mathbf{x}^{(k)}))$.
- *Backtracking line search*: Note that for t which is small enough and $\alpha \in (0, 1)$, we have

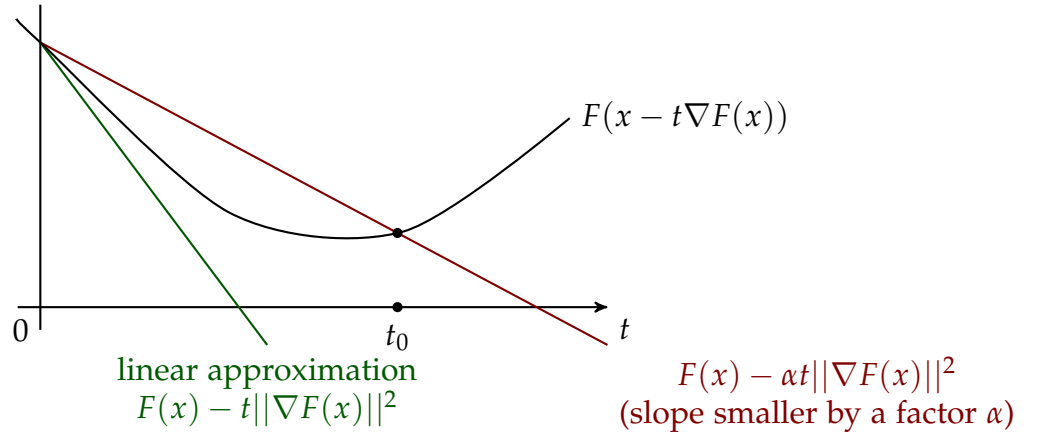
$$F(\mathbf{x} - t \nabla F(\mathbf{x})) \approx F(\mathbf{x}) - t \|\nabla F(\mathbf{x})\|^2 < F(\mathbf{x}) - \alpha t \|\nabla F(\mathbf{x})\|^2.$$

So, we might start with $t = 1$ and fix a parameter $\alpha \in (0, \frac{1}{2})$. Then, we decrease t through $t \leftarrow \frac{t}{2}$ until

$$F(\mathbf{x} - t \nabla F(\mathbf{x})) < F(\mathbf{x}) - \alpha t \|\nabla F(\mathbf{x})\|^2 \quad (8.15)$$

is satisfied. Equation (8.15) guarantees that we iterate until a "good decrease" is reached.

This guarantees that $F(\mathbf{x}^{(k)}) - F(\mathbf{x}^{(k+1)}) > \alpha t \|\nabla F(\mathbf{x}^{(k)})\|^2$, i.e. that there is a decrease in the value of F .



This method is called backtracking line-search since we start at $t = 1$ and stop once $t \leq t_0$ for the first time.

Newton's Method

As an alternative to gradient descent, one can formulate an iteration based on Newton's method. The idea is that now, instead of solving for $F(\mathbf{x}) = 0$ (i.e. root finding), the goal is to find the minimum of F , and hence, we solve for $\nabla F(\mathbf{x}) = 0$, which can be interpreted as a root finding problem for the gradient of F . We require that F is twice differentiable. This allows us to compute the Taylor approximation of second-order around $\mathbf{x}^{(k)}$ by

$$F(\mathbf{x}) \approx F(\mathbf{x}^{(k)}) + \nabla F(\mathbf{x}^{(k)})^\top (\mathbf{x} - \mathbf{x}^{(k)}) + \frac{1}{2} (\mathbf{x} - \mathbf{x}^{(k)})^\top \mathbf{H}_F(\mathbf{x}^{(k)}) (\mathbf{x} - \mathbf{x}^{(k)}).$$

We might differentiate the right-hand side and set it to zero (this yields the minimum of the quadratic approximation) and find the iteration rule

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - [\mathbf{H}_F(\mathbf{x}^{(k)})]^{-1} \nabla F(\mathbf{x}^{(k)}).$$

One can show that near the minima of F , the above iteration converges quadratically. Additionally note that

- Newton's method typically needs fewer iterations than gradient descent,
- Gradient descent typically converges on a larger region than Newton's method.

Finally, we observe that both Newton's method and Gradient Descent may get stuck at local minima.

Broyden–Fletcher–Goldfarb–Shanno (BFGS) Method

This method is a *quasi-Newton method*. Instead of working with the exact Hessian $\mathbf{H}_F(\mathbf{x}^{(k)})$, we will approximate it by a matrix $\mathbf{B}_k \in \mathbb{R}^{n \times n}$ with the property that \mathbf{B}_{k+1} is obtained from a simple update of \mathbf{B}_k . Similar to the procedure of Broyden's method (see Section 8.2.4), we will make use of a secant-like condition

$$\mathbf{B}_{k+1} \underbrace{(\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)})}_{=: \mathbf{s}^{(k)}} = \underbrace{\nabla F(\mathbf{x}^{(k+1)}) - \nabla F(\mathbf{x}^{(k)})}_{=: \mathbf{y}^{(k)}},$$

or

$$\mathbf{B}_{k+1} \mathbf{s}^{(k)} = \mathbf{y}^{(k)}. \quad (8.16)$$

Now, we will force \mathbf{B}_{k+1} to be symmetric positive definite (recall that this is the property of the Hessian at a local minimum) by making the ansatz

$$\mathbf{B}_{k+1} = \mathbf{B}_k + \alpha \mathbf{u} \mathbf{u}^\top + \beta \mathbf{v} \mathbf{v}^\top$$

and requiring (8.16) to hold. The choice

$$\begin{aligned} \mathbf{u} &= \mathbf{y}^{(k)}, & \mathbf{v} &= \mathbf{B}_k \mathbf{s}^{(k)}, \\ \alpha &= \frac{1}{(\mathbf{y}^{(k)})^\top \mathbf{s}^{(k)}}, & \beta &= -\frac{1}{(\mathbf{s}^{(k)})^\top \mathbf{B}_k \mathbf{s}^{(k)}}, \end{aligned}$$

results in (8.16) being satisfied. The update becomes

$$\mathbf{B}_{k+1} = \mathbf{B}_k + \frac{\mathbf{y}^{(k)} (\mathbf{y}^{(k)})^\top}{(\mathbf{y}^{(k)})^\top \mathbf{s}^{(k)}} - \frac{\mathbf{B}_k \mathbf{s}^{(k)} (\mathbf{B}_k \mathbf{s}^{(k)})^\top}{(\mathbf{s}^{(k)})^\top \mathbf{B}_k \mathbf{s}^{(k)}}.$$

The BFGS iteration is then given by

$$\boxed{\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \mathbf{B}_k^{-1} \nabla F(\mathbf{x}^{(k)})},$$

which suggests using the Sherman-Morrison-Woodbury formula

$$\mathbf{B}_{k+1}^{-1} = \left(\mathbf{I} - \frac{\mathbf{s}^{(k)} (\mathbf{y}^{(k)})^\top}{(\mathbf{y}^{(k)})^\top \mathbf{s}^{(k)}} \right) \mathbf{B}_k^{-1} \left(\mathbf{I} - \frac{\mathbf{s}^{(k)} (\mathbf{y}^{(k)})^\top}{(\mathbf{y}^{(k)})^\top \mathbf{s}^{(k)}} \right) + \frac{\mathbf{y}^{(k)} (\mathbf{y}^{(k)})^\top}{(\mathbf{s}^{(k)})^\top \mathbf{s}^{(k)}}$$

for the update procedure.

Note. A variant of the BFGS method is the *L-BFGS* method, where the L stands for "limited memory". The L-BFGS method does not store the dense matrices \mathbf{B}_k .

Bibliography

- [1] Uri M Ascher and Chen Greif. *A first course on numerical methods*. SIAM, 2011.
- [2] M. Struwe. Analysis für informatik. Website, 2009. Lecture notes, ETH Zürich, <https://people.math.ethz.ch/~struwe/Skripten/InfAnalysis-I-II-31-7-09.pdf>.
- [3] Kaspar Nipp and Daniel Stoffer. Lineare algebra. vdf. *Hochschulverlag an der ETH Zrich*, 5, 2002.
- [4] M.H. Gutknecht. Lineare algebra. Website, 2009. Lecture notes, SAM, ETH Zürich, <http://www.sam.math.ethz.ch/~mhg/unt/LA/HS07/LAS07.pdf>.
- [5] Martin Hanke-Bourgeois. *Grundlagen der numerischen Mathematik und des wissenschaftlichen Rechnens*, volume 178. Springer, 2002.
- [6] Alfio Quarteroni, Riccardo Sacco, and Fausto Saleri. *Numerical mathematics*, volume 37. Springer Science & Business Media, 2010.
- [7] Wolfgang Dahmen and Arnold Reusken. *Numerik für Ingenieure und Naturwissenschaftler*. Springer-Verlag, 2006.