



how to prepare for numcs in 20 mins



Alle

Videos

Bilder

News

Maps

Shopping

Bü

1 2
3 4

Numerical Methods for CS



This and many more summaries can be found on <https://n.ethz.ch/~dcamenisch>. Feel free to leave a comment in the document if you spot any mistakes! As always no guarantees for completeness or correctness are made.

▼ 0. Table of Contents

- 0. Table of Contents
- 1. Computing with Matrices and Vectors
 - 1.2.1 Eigen
 - 1.2.3 Dense Matrix Storage Formats
 - 1.4 Computational Effort
 - 1.4.2 Cost of Basic Linear-Algebra Operations
 - 1.4.3 The Kronecker product
 - 1.5 Machine Arithmetic and Consequences
 - 1.5.2 Machine Numbers
 - 1.5.3 Roundoff Errors
 - 1.5.4 Cancellation
- 2. Direct Methods for Square Linear Systems of Equations
 - 2.1 Introduction: Linear Systems of Equations (LSE)
 - 2.3 Gaussian Elimination (GE)
 - 2.3.1 Basic Algorithm
 - 2.3.2 LU-Decomposition
 - 2.3.3 Pivoting
 - 2.6 Exploiting Structure when Solving Linear Systems
 - 2.7 Sparse Linear Systems
 - 2.7.1 Sparse Matrix Storage Formats
- 3. Direct Methods for Linear Least Squares Problems
 - 3.1 Least Squares Solution Concepts
 - 3.1.1 Least Squares Solutions: Definitions
 - 3.1.2 Normal Equations
 - 3.1.3 Moore-Penrose Pseudoinverse
 - 3.2 Normal Equation Methods
 - 3.3 Orthogonal Transformation Methods
 - 3.3.1 Transformation Idea
 - 3.3.2 Orthogonal/Unitary Matrices
 - 3.3.3 QR-Decomposition
 - 3.3.4 QR-Based Solver for Linear Least Squares Problems

- 3.4 [Singular Value Decomposition \(SVD\)](#)
 - 3.4.1 [SVD: Definition and Theory](#)
 - 3.4.2 [SVD in Eigen](#)
 - 3.4.3 [Solving General Least-Squares Problems by SVD](#)
 - 3.4.4 [SVD-Based Optimization and Approximation](#)
- [Solving Linear Systems of Equations Overview](#)
- 4. [Filtering Algorithms](#)
 - 4.1 [Filters and Convolutions](#)
 - 4.1.1 [Discrete Finite Linear Time-Invariant Casual Channels/Filters](#)
 - 4.1.2 [LT-FIR Linear Mappings](#)
 - 4.1.3 [Discrete Convolutions](#)
 - 4.1.4 [Periodic Convolutions](#)
 - 4.2 [Discrete Fourier Transform \(DFT\)](#)
 - 4.2.1 [Diagonalizing Circulant Matrices](#)
 - 4.2.2 [Discrete Convolution via Discrete Fourier Transform](#)
 - 4.2.5 [Two-dimensional DFT](#)
 - 4.3 [Fast Fourier Transform \(FFT\)](#)
- 5. [Data Interpolation and Data Filtering in 1D](#)
 - 5.1 [Abstract Interpolation](#)
 - 5.2 [Global Polynomial Interpolation](#)
 - 5.2.1 [Uni-Variante Polynomials](#)
 - 5.2.2 [Polynomial Interpolation: Theory](#)
 - 5.2.3 [Polynomial Interpolation: Algorithms](#)
 - 5.2.4 [Polynomial Interpolation: Sensitivity](#)
 - 5.3 [Shape-Preserving Interpolation](#)
 - 5.3.1 [Shape Properties of Functions and Data](#)
 - 5.3.2 [Piecewise Linear Interpolation](#)
 - 5.3.3 [Cubic Hermite Interpolation](#)
 - 5.4 [Splines](#)
 - 5.4.2 [Cubic-Spline Interpolation](#)
 - 5.4.3 [Structural Properties of Cubic Spline Interpolation](#)
 - 5.6 [Trigonometric Interpolation](#)
 - 5.6.1 [Trigonometric Polynomials](#)
 - 5.6.2 [Reduction to Lagrange Interpolation](#)
 - 5.6.3 [Equidistant Trigonometric Interpolation](#)
 - 5.7 [Least Squares Data Fitting](#)
- 8. [Iterative Methods for Non-Linear Systems of Equations](#)
 - 8.1 [Introduction](#)
 - 8.2 [Iterative Methods](#)
 - 8.2.1 [Fundamental Concepts](#)
 - 8.2.2 [Speed of Convergence](#)
 - 8.2.3 [Termination Criteria / Stopping Rule](#)
 - 8.3 [Fixed-Point Iteration](#)
 - 8.4 [Finding Zeros of Scalar Functions](#)
 - 8.4.1 [Bisection](#)
 - 8.4.2 [Model Function Methods](#)
 - 8.4.3 [Asymptotic Efficiency of Iterative Methods for Zero Finding](#)
 - 8.5 [Newton's Method in \$\mathbb{R}^n\$](#)
 - 8.5.1 [The Newton Iteration](#)
 - 8.5.3 [Termination of Newton Iteration](#)
 - 8.5.4 [Damped Newton Method](#)
 - 8.6 [Quasi-Newton Method](#)
 - 8.7 [Non-Linear Least Squares](#)

1. Computing with Matrices and Vectors

1.2.1 Eigen

The method `.block(int i, int j, int m, int n)` returns a sub-matrix starting at the top left corner (i, j) with size m, n .

1.2.3 Dense Matrix Storage Formats

All numerical libraries store the entries of a dense matrix $A \in \mathbb{K}^{m,n}$ in a linear array of length mn .

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

Row major: [1, 2, 3, 4, 5, 6, 7, 8, 9]
Column major: [1, 4, 7, 2, 5, 8, 3, 6, 9]

1.4 Computational Effort

1.4.2 Cost of Basic Linear-Algebra Operations

Performing elementary operations through simple (nested) loops, we arrive at the following complexity bounds:

Computational Cost of Basic Operations

Operation	Description	asymptotic complexity
<u>dot product</u>	$(x \in \mathbb{R}^n, y \in \mathbb{R}^n) \rightarrow x^H y$	$O(n)$
<u>tensor product</u>	$(x \in \mathbb{R}^m, y \in \mathbb{R}^n) \rightarrow xy^H$	$O(mn)$
<u>matrix * vector</u>	$(x \in \mathbb{R}^n, A \in \mathbb{R}^{m,n}) \rightarrow Ax$	$O(mn)$
<u>matrix product</u>	$(A \in \mathbb{R}^{m,n}, B \in \mathbb{R}^{n,k}) \rightarrow AB$	$O(mnk)$

1.4.3 The Kronecker product

Definition: The *Kronecker product* $A \otimes B$ of two matrices $A \in \mathbb{K}^{m,n}$ and $B \in \mathbb{K}^{l,k}$ is the $(ml) \times (nk)$ -matrix

$$A \otimes B := \begin{bmatrix} (A)_{11}B & (A)_{12}B & \cdots & (A)_{1n}B \\ (A)_{21}B & (A)_{22}B & \cdots & (A)_{2n}B \\ \vdots & \vdots & \ddots & \vdots \\ (A)_{m1}B & (A)_{m2}B & \cdots & (A)_{mn}B \end{bmatrix} \in \mathbb{K}^{ml, nk}$$

1.5 Machine Arithmetic and Consequences

1.5.2 Machine Numbers

Computers are finite automatons, which therefore can only handle *finitely many number*, not \mathbb{R} .

The set of machine numbers \mathbb{M} cannot be closed under elementary arithmetic operations $+$, $-$, \cdot , $/$, that is, when adding, multiplying, etc. This leads to the fact, that **roundoff errors** are *inevitable*.

1.5.3 Roundoff Errors

Definition: Let $\tilde{x} \in \mathbb{K}$ be an approximation of $x \in \mathbb{K}$. Then the **absolute error** is given by

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

and its **relative error** is defined as

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

The *number of correct digits* of an approximation \tilde{x} of $x \in \mathbb{K}$ is defined through the relative error: If $\epsilon_{\text{rel}} \leq 10^{-l}$, then \tilde{x} has l correct digits, $l \in \mathbb{N}_0$.

1.5.4 Cancellation

We define the term *cancellation* as the **subtraction** of almost equal numbers (with both having some relative error), which leads to an extreme amplification of the relative errors. It is important to see that cancellation only happens if we have subtraction, therefore it is advisable to avoid it or have it at the innermost part of an equation.

A important formula to avoid cancellation is given by:

$$a - b = \frac{a^2 - b^2}{a + b}$$

Further the logarithmic formulas can be useful.

2. Direct Methods for Square Linear Systems of Equations

2.1 Introduction: Linear Systems of Equations (LSE)

The problem: solving a linear system

We are given the following input and are looking for the output as shown below:

- Input/data: square matrix $A \in \mathbb{K}^{n,n}$, vector $b \in \mathbb{K}^n$
- Output/result: solution vector $x \in \mathbb{K}^n$, such that $Ax = b$

We call A the *system matrix* or *coefficient matrix* and b the *right hand side vector*

Definition: The **rank** of a matrix $M \in \mathbb{K}^{m,n}$, denoted by $\text{rank}(M)$, is the maximal number of linearly independent rows/columns of M . Equivalently, $\text{rank}(M) = \dim \mathcal{R}(A)$.

Theorem: A square matrix $A \in \mathbb{K}^{n,n}$ is **invertible/regular** if one of the following *equivalent* conditions is satisfied:

1. $\exists B \in \mathbb{K}^{n,n} : BA = AB = I$
2. the columns or rows of A are linearly independent
3. $\det(A) \neq 0$
4. $\text{rank}(A) = n$

2.3 Gaussian Elimination (GE)

2.3.1 Basic Algorithm

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

The computational cost of Gaussian elimination is given by

- forward elimination: $n(n-1)(\frac{2}{3}n + \frac{7}{6}) \approx n^3$ Ops.
- backward elimination: n^2 Ops.

which yields a total **asymptotic complexity for GE** without pivoting for a generic LSE of $O(n^3)$.

2.3.2 LU-Decomposition

A *matrix factorization* expresses a general matrix A as the product of two special matrices.

We can perform **LU-Decomposition** by performing the known Gaussian elimination algorithm, but keeping track of the *negative multipliers* and let them take the places of matrix entries made to vanish.

After performing the above Gaussian elimination, we get the following decomposition

$$A = LU \Rightarrow \begin{bmatrix} 1 & 1 & 0 \\ 2 & 1 & -1 \\ 3 & -1 & -1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 3 & 4 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 & 0 \\ 0 & -1 & -1 \\ 0 & 0 & 3 \end{bmatrix}$$

Definition: Given a square matrix $A \in \mathbb{K}^{n,n}$, an *upper triangular matrix* $U \in \mathbb{K}^{n,n}$ and a *normalized lower triangular matrix* L form an **LU-decomposition** of A , if $A = LU$.

The **asymptotic complexity for LU-factorization** of $A \in \mathbb{R}^{n,n}$ is given by $O(n^3)$ if $n \rightarrow \infty$. But if we once solve the decomposition, we can reuse it with an asymptotic complexity of $O(n^2)$.

! If we give a matrix to the `.solve()` function, each column gets treated like a vector, meaning we solve n systems of linear equation. Therefore we get a runtime of $O(n^3)$ for the backwards substitution.

2.3.3 Pivoting

When doing *pivoting* in numerical methods we usually choose the **relatively largest pivot**.

Lemma: For any regular $A \in \mathbb{K}^{n,n}$ there is a permutation matrix $P \in \mathbb{K}^{n,n}$, a normalized lower triangular matrix $L \in \mathbb{K}^{n,n}$, and a regular upper triangular matrix $U \in \mathbb{K}^{n,n}$, such that $PA = LU$.

2.6 Exploiting Structure when Solving Linear Systems

By *structure* of a linear system we mean prior knowledge that

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

System matrix: coefficient matrix of an linear system of equations

Triangular linear systems

Triangular linear systems are linear systems of equations whose system matrix is a triangular matrix. They can be solved by backward/forward elimination within $O(n^2)$ compared to $O(n^3)$ for a generic dense matrix.

Linear Systems with arrow matrices

From $n \in \mathbb{N}$, a diagonal matrix $D \in \mathbb{K}^{n,n}$, $c \in \mathbb{K}^n$, $b \in \mathbb{K}^n$, and $\alpha \in \mathbb{K}$, we can build an $(n+1) \times (n+1)$ **arrow matrix**

$$A = \begin{bmatrix} D & c \\ b & \alpha \end{bmatrix} \Rightarrow \begin{bmatrix} \cdot & \cdot & \cdot & \vdots \\ & \cdot & \cdot & \vdots \\ \dots & \dots & \cdot & \vdots \\ & \dots & \dots & \cdot \end{bmatrix}$$

In this case we have that

$$Ax = \begin{bmatrix} D & c \\ b^T & \alpha \end{bmatrix} \begin{bmatrix} x_1 \\ \zeta \end{bmatrix} = y := \begin{bmatrix} y_1 \\ \eta \end{bmatrix}$$

$$\Rightarrow \zeta = \frac{\eta - b^T D^{-1} y_1}{\alpha - b^T D^{-1} c}, \quad x_1 = D^{-1}(y_1 - \zeta c).$$

This yields an **asymptotic complexity for solving arrow systems** of $O(n)$ for $n \rightarrow \infty$.

Solving LSE subject to low-rank modification of system matrix

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

- either A has a favorable structure, eg. triangular,
- or an LU-decomposition of A is already available

Now, a \tilde{A} is obtained by changing a *single entry* of A . This modification represent so-called **rank-1-modification** of A . A generic rank-1-modification reads

$$A \in \mathbb{K}^{n,n} \rightarrow \tilde{A} := A + uv^H, \quad u, v \in \mathbb{K}^n.$$

We consider the block partitioned linear system

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

The *Schur complement* system after elimination of ζ reads $(A + uv^H)\tilde{x} = b \Leftrightarrow \tilde{A}\tilde{x} = b$. We do block elimination again, now getting rid of \tilde{x} first, which yields the other Schur complement system

-- . -- .

$$(1 + v^H A^{-1} u) \zeta = v^H A^{-1} b$$

$$\Rightarrow A \tilde{x} = b - \frac{u v^H A^{-1}}{1 + v^H A^{-1} u} b.$$

2.7 Sparse Linear Systems

Notion: $A \in \mathbb{K}^{n,n}$, $m, n \in \mathbb{N}$ is said to be **sparse**, if

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

The matrix is said to be **dense** otherwise.

2.7.1 Sparse Matrix Storage Formats

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

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

Triplet/coordinate list (COO) format

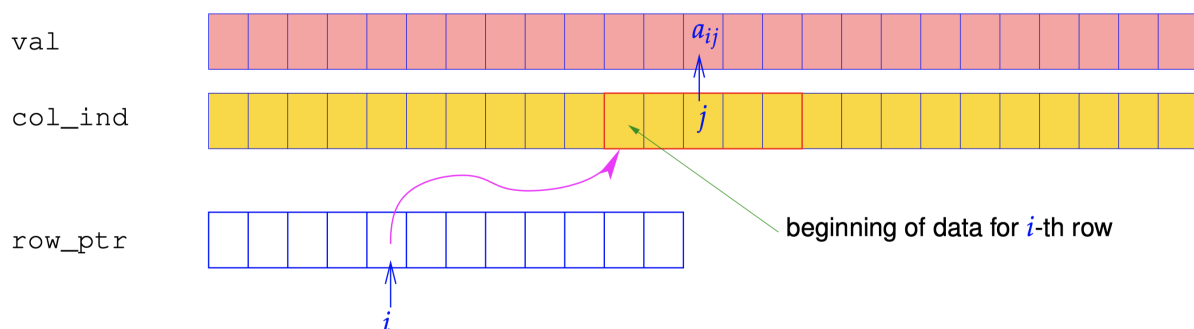
This format stores triplets (i, j, α) , $1 \leq i \leq m$, $1 \leq j \leq n$:

The vector of triplets in a `TripletMatrix` has size $\geq \text{nnz}(A)$. We write \geq because *repetitions* of index pairs (i, j) are allowed. The matrix entry $(A)_{ij}$ is defined to be the sum of all values α_{ij} associated with the index pair (i, j) .

Compressed row-storage (CRS) format

The CRS format for a sparse matrix $A \in \mathbb{K}^{m,n}$ keeps the data in three contiguous arrays:

- `std::vector<scalar_t> val` \rightarrow size $\text{nnz}(A)$
- `std::vector<size_t> col_ind` \rightarrow size $\text{nnz}(A)$
- `std::vector<size_t> row_ptr` \rightarrow size $m + 1$ and `row_ptr[m] = nnz(A) + 1`



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

val-vector:

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

col_ind-array:

1	5	1	2	6	2	3	4	1...5	6	2	5	6
---	---	---	---	---	---	---	---	-------	---	---	---	---

row_ptr-array:

1	3	6	9	13	17	20
---	---	---	---	----	----	----

3. Direct Methods for Linear Least Squares Problems

Overdetermined (OD) linear systems of equations, a linear system with a "tall" rectangular system matrix:

$$Ax = b : x \in \mathbb{R}^n, b \in \mathbb{R}^m, A \in \mathbb{R}^{m,n}, m \geq n$$

$$\begin{bmatrix} A \end{bmatrix} \begin{bmatrix} x \end{bmatrix} = \begin{bmatrix} b \end{bmatrix}$$

3.1 Least Squares Solution Concepts

Recall from linear algebra that $Ax = b$ has a solution, if and only if the right hand side vector b lies in the image of the matrix A :

$$\exists x \in \mathbb{R}^n : Ax = b \Leftrightarrow b \in \mathcal{R}(A).$$

Following the notation for important subspaces associated with a matrix $A \in \mathbb{K}^{m,n}$:

- *image/range*: $\mathcal{R}(A) := \{Ax, x \in \mathbb{K}^n\} \subset \mathbb{K}^m$,
- *kernel/nullspace*: $\mathcal{N}(A) := \{x \in \mathbb{K}^n : Ax = 0\}$.

3.1.1 Least Squares Solutions: Definitions

Definition: For a given $A \in \mathbb{R}^{m,n}$, $b \in \mathbb{R}^m$ the vector $x \in \mathbb{R}^n$ is a **least squares solution** of the linear system of $Ax = b$, if

$$x \in \operatorname{argmin}_{y \in \mathbb{R}^n} \|Ay - b\|_2^2$$

We write $\operatorname{lsq}(A, b)$ for the set of least squares solutions of the linear system of equations $Ax = b$, $A \in \mathbb{R}^{m,n}$, $b \in \mathbb{R}^m$:

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

Theorem: For any $A \in \mathbb{R}^{m,n}$, $b \in \mathbb{R}^m$ a least squares solution of $Ax = b$ exists.

3.1.2 Normal Equations

Theorem: The vector $x \in \mathbb{R}^n$ is a *least squares solution* of the linear system of equations $Ax = b$, if and only if it solves the **normal equations**

$$A^T Ax = A^T b.$$

Theorem: For $A \in \mathbb{R}^{m,n}$, $m \geq n$, holds

$$\begin{aligned} \mathcal{N}(A^T A) &= \mathcal{N}(A), \\ \mathcal{R}(A^T A) &= \mathcal{R}(A^T). \end{aligned}$$

Lemma: For any matrix $A \in \mathbb{K}^{m,n}$ holds

-- .

$$\begin{aligned}\mathcal{N}(A) &= \mathcal{R}(A^H)^\perp \\ \mathcal{N}(A)^\perp &= \mathcal{R}(A^H).\end{aligned}$$

Corollary: If $m \geq n$ and $\mathcal{N}(A) = \{0\}$, then the linear system of equations $Ax = b$, $A \in \mathbb{R}^{m,n}$, $b \in \mathbb{R}^m$, has a **unique least squares solution**

$$x = (A^T A)^{-1} A^T b,$$

that can be obtained by solving the *normal equations*.

3.1.3 Moore-Penrose Pseudoinverse

Theorem: Given $A \in \mathbb{R}^{m,n}$, $b \in \mathbb{R}^m$, the generalized solution x^\dagger of the linear system of equations $Ax = b$ is given by

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

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

The matrix $A^\dagger := V(V^T A^T A V)^{-1} V^T A^T \in \mathbb{R}^{n,m}$ is called the **Moore-Penrose pseudoinverse** of A . Note, that the Moore-Penrose pseudoinverse does *not depend* on the choice of V .

3.2 Normal Equation Methods

We can give a simple algorithm for the normal equation method for solving *full-rank* least squares problems $Ax = b$:

1. Compute *regular* matrix $C := A^T A \in \mathbb{R}^{n,n}$. $O(n^2 m)$
2. Compute right hand side vector $c := A^T b$. $O(nm)$
3. Solve symmetric positive definite (s.p.d.) linear system of equations $Cx = c$. $O(n^3)$

The asymptotic complexity of the normal equation method is given by $O(n^2 m + n^3)$ for $m, n \rightarrow \infty$.

3.3 Orthogonal Transformation Methods

3.3.1 Transformation Idea

In this chapter we consider the full-rank linear least squares problem $A \in \mathbb{R}^{m,n}$, $b \in \mathbb{R}^m$ given and we try to find $x = \operatorname{argmin}_{y \in \mathbb{R}^n} \|Ay - b\|_2$. We furthermore know that $m \geq n$ and A has full rank: $\operatorname{rank}(A) = n$.

The idea is that if we have a transformation matrix $T \in \mathbb{R}^{m,m}$ satisfying $\|Ty\|_2 = \|y\|_2 \forall y \in \mathbb{R}^m$, then

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

where $\tilde{A} = TA$ and $\tilde{b} = Tb$.

3.3.2 Orthogonal/Unitary Matrices

Definition: Unitary and orthogonal matrices

- $Q \in \mathbb{K}^{n,n}$, $n \in \mathbb{N}$, is *unitary*, if $Q^{-1} = Q^H$
- $Q \in \mathbb{K}^{n,n}$, $n \in \mathbb{N}$, is *orthogonal*, if $Q^{-1} = Q^T$

Theorem: A matrix is *unitary/orthogonal*, if and only if the associated linear mapping preserves the 2-norm:

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

From the above theorem we can directly state the following conclusions. If a matrix $Q \in \mathbb{K}^{n,n}$ is unitary/orthogonal, then

- all rows/columns have Euclidean norm = 1
- all rows/columns are pairwise orthogonal
- $|\det Q| = 1$, $\|Q\|_2 = 1$, and all eigenvalues $\in \{z \in \mathbb{C} : |z| = 1\}$
- $\|QA\|_2 = \|A\|_2$ for any matrix $A \in \mathbb{K}^{n,m}$

3.3.3 QR-Decomposition

3.3.3.1 QR-Decomposition: Theory

Theorem: If $\{a^1, \dots, a^n\} \subset \mathbb{R}^m$ is linearly independent, then the Gram-Schmidt algorithm computes **orthogonal vectors** $q^1, \dots, q^n \in \mathbb{R}^m$ satisfying

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

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

Theorem: For any matrix $A \in \mathbb{K}^{n,k}$ with $\text{rank}(A) = k$ there exists

1. a unique matrix $Q_0 \in \mathbb{R}^{n,k}$ that satisfies $Q_0^H Q_0 = I_k$, and a unique *upper triangular* Matrix $R_0 \in \mathbb{K}^{k,k}$ with $(R)_{ii} > 0$, $i \in \{1, \dots, k\}$, such that

$$A = Q_0 \cdot R_0 \text{ ("economical" QR-decomposition)}$$

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

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

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

3.3.3.2 Computation of QR-Decomposition

Corollary: The product of two orthogonal/unitary matrices of the same size is again orthogonal/unitary.

The following so called **Householder matrices (HHM)** effect the reflection of a vector into a multiple of the first unit vector with the same length:

$$Q = H(v) := I - 2 \frac{vv^T}{v^T v} \text{ with } v = a \pm \|a\|_2 e_1$$

where e_1 is the first Cartesian basis vector.

Suitable **successive Householder transformations** determined by the left most column of shrinking bottom right matrix blocks can be used to achieve upper triangular form R . Writing Q_l for the Householder matrix used in the l -th factorization yields for the **QR-decomposition** of $A \in \mathbb{C}^{n,n}$, $A = QR$:

$$Q_{n-1} \cdot Q_{n-2} \cdots Q_1 A = R \text{ and } Q := Q_1^T \cdots Q_{n-1}^T.$$

The following orthogonal transformation, a **Givens rotation**, annihilates the k -th component of a vector $a = [a_1, \dots, a_n]^T \in \mathbb{R}^n$. Here γ stands for $\cos \phi$ and σ for $\sin \phi$, ϕ the angle of rotation:

$$G_{1k}(a_1, a_k)a := \begin{bmatrix} \gamma & \cdots & \sigma & \cdots & 0 \\ \vdots & \ddots & \vdots & & \vdots \\ -\sigma & \cdots & \gamma & \cdots & 0 \\ \vdots & & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & 1 \end{bmatrix} \cdot \begin{bmatrix} a_1 \\ \vdots \\ a_k \\ \vdots \\ a_n \end{bmatrix} = \begin{bmatrix} a_1 \\ \vdots \\ 0 \\ \vdots \\ a_n \end{bmatrix},$$

$$\gamma = \frac{a_1}{\sqrt{|a_1|^2 + |a_k|^2}}, \sigma = \frac{a_k}{\sqrt{|a_1|^2 + |a_k|^2}}$$

The QR-decomposition by successive Householder transformations has asymptotic complexity $O(mn^2)$ for $m, n \rightarrow \infty$.

3.3.4 QR-Based Solver for Linear Least Squares Problems

We consider the full-rank linear least squares problem: Given $A \in \mathbb{R}^{m,n}$, $m \geq n$, $\text{rank}(A) = n$, seek $x \in \mathbb{R}^n$ such that $\|Ax - b\|_2 \rightarrow \min$. We assume that we are given a QR-decomposition: $A = QR$, $Q \in \mathbb{R}^{m,m}$ orthogonal, $R \in \mathbb{R}^{m,n}$ regular upper triangular matrix.

We then apply the orthogonal 2-norm preserving transformation encoded in Q to $Ax - b$:

$$\|Ax - b\|_2 = \|QRx - b\|_2 = \|Q(Rx - Q^T b)\|_2 = \|Rx - \tilde{b}\|_2, \tilde{b} := Q^T b.$$

Normal equations vs. orthogonal transformations methods

- Use *orthogonal transformation methods* for least squares problems, whenever $A \in \mathbb{R}^{m,n}$ is dense and n is small.
- Use *normal equations* in the expanded form, when $A \in \mathbb{R}^{m,n}$ is sparse and m, n are big.

3.4 Singular Value Decomposition (SVD)

3.4.1 SVD: Definition and Theory

For any $A \in \mathbb{K}^{m,n}$ there are unitary/orthogonal matrices $U \in \mathbb{K}^{m,m}$, $V \in \mathbb{K}^{n,n}$ and a generalized diagonal matrix $\Sigma = \text{diag}(\sigma_1, \dots, \sigma_p) \in \mathbb{R}^{m,n}$, $p := \min\{m, n\}$, $\sigma_1 \geq \sigma_2 \geq \dots \geq \sigma_p \geq 0$ such that

$$A = U\Sigma V^H.$$

Definition: The decomposition $A = U\Sigma V^H$ is called the **singular value decomposition (SVD)** of A . The diagonal entries of σ_i of Σ are the *singular values* of A . The columns of U/V are the left/right *singular vectors* of A .

Remark: As in the case of QR-decomposition we can also drop the bottom zero rows of Σ and the corresponding columns of U in the case of $m > n$. Thus we end up with an **economical singular value decomposition**, also called **thin SVD** in literature.

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

Lemma: If, for some $1 \leq r \leq p := \min\{m, n\}$, the singular values of $A \in \mathbb{K}^{m,n}$ satisfy $\sigma_1 \geq \dots \geq \sigma_r > \sigma_{r+1} = \dots = \sigma_p = 0$, then

- $\text{rank}(A) = r$ (the number of non-zero singular values)
- $\mathcal{N}(A) = \text{Span}\{(V)_{:,r+1}, \dots, (V)_{:,n}\}$
- $\mathcal{R}(A) = \text{Span}\{(U)_{:,1}, \dots, (U)_{:,r}\}$

3.4.2 SVD in Eigen

The asymptotic complexity for the economical SVD is $O(\min\{m, n\}^2 \cdot \max\{m, n\})$

3.4.3 Solving General Least-Squares Problems by SVD

In this chapter we consider the most general setting

$$Ax = b \in \mathbb{R}^m \text{ with } A \in \mathbb{R}^{m,n}, \text{ rank}(A) = r \leq \min\{m, n\}.$$

We can use the invariance of the 2-norm of a vector with respect to multiplication with $U := [U_1 \ U_2]$ together with the fact that U is unitary:

$$\|Ax - b\|_2 = \left\| [U_1 \ U_2] \begin{bmatrix} \Sigma_r & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} V_1^T \\ V_2^T \end{bmatrix} x - b \right\|_2 = \left\| \begin{bmatrix} \Sigma_r V_1^T x \\ 0 \end{bmatrix} - \begin{bmatrix} U_1^T b \\ U_2^T b \end{bmatrix} \right\|_2.$$

With this equation we arrive at the generalized solution

$$x^\dagger = V_1 \Sigma_r^{-1} U_1^T b, \quad \|r\|_2 = \|U_2^T b\|_2.$$

Theorem: If $A \in \mathbb{K}^{m,n}$ has the SVD decomposition $A = U \Sigma V^H$ then its *Moore-Penrose pseudoinverse* is given by $A^\dagger = V_1 \Sigma_r^{-1} U_1^H$.

3.4.4 SVD-Based Optimization and Approximation

3.4.4.1 Norm-Constrained Extrema of Quadratic Forms

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

$$\text{given } A \in \mathbb{K}^{m,n}, \ m \geq n, \text{ find } x \in \mathbb{K}^n, \ \|x\|_2 = 1, \ \|Ax\|_2 \rightarrow \min.$$

This problem can be solved with SVD with the minimizer $x^* = V e_n = (V)_{:,n}$ from which we can obtain the minimal value $\|Ax^*\|_2 = \sigma_n$.

Lemma: If $A \in \mathbb{K}^{m,n}$ has singular values $\sigma_1 \geq \sigma_2 \geq \dots \geq \sigma_p \geq 0$, $p := \min\{m, n\}$, then its Euclidean matrix norm is given by $\|A\|_2 = \sigma_1(A)$. If $m = n$ and A is regular/invertible, then its 2-norm condition number is $\text{cond}_2(A) = \sigma_1/\sigma_n$.

3.4.4.2 Best Low-Rank Approximation

TLDR: for the best k -rank approximation you turn the sigma into $k \times k$ matrix (cut everything else away) then you take away the columns in U and V accordingly.

Solving Linear Systems of Equations Overview

Given $Ax = b$, $A \in \mathbb{R}^{m,n}$, there are 6 different cases we have to differentiate:

Case 1.1 : $m = n$ & $\text{rank}(A) = n$

Algorithm: We use LU-Decomposition with Gauss elimination to solve this LSE.

Alternatively we could use QR-Decomposition.

Runtime: $O(n^3)$

Case 1.2 : $m = n$ & $\text{rank}(A) < n$

Algorithm: We use SVD to compute a LSQ solution.

Runtime: $O(n^3)$

Case 2.1 : $m > n$ & $\text{rank}(A) = n$

Algorithm: We use QR-Decomposition to get the LSQ.

Alternatively we could use the normal equation with LU-Decomposition.

Runtime: $O(mn^2 + n^2) = O(mn^2) / O(mn^2 + n^3) = O(mn^2)$

Case 2.2 : $m > n$ & $\text{rank}(A) < n$

Algorithm: We use SVD to compute a LSQ solution.

Runtime: $O(mn^2)$

Case 3.1 : $m < n$ & $\text{rank}(A) = m$

Algorithm: We use SVD to compute a LSQ solution.

Runtime: $O(m^2n)$

Case 3.2 : $m < n$ & $\text{rank}(A) < m$

Algorithm: We use SVD to compute a LSQ solution.

Runtime: $O(m^2n)$

4. Filtering Algorithms

4.1 Filters and Convolutions

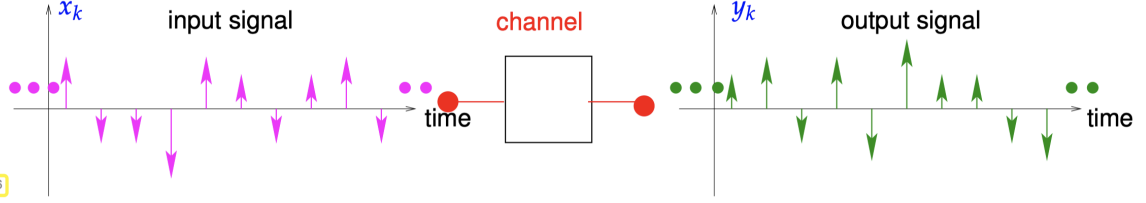
4.1.1 Discrete Finite Linear Time-Invariant Casual Channels/Filters

Mathematically speaking, a **discrete channel / filter** is a function or mapping $F : l^\infty(\mathbb{Z}) \rightarrow l^\infty(\mathbb{Z})$ from the vector space $l^\infty(\mathbb{Z})$ of bounded input sequences $\{x_j\}_{j \in \mathbb{Z}}$,

$$l^\infty(\mathbb{Z}) := \{(x_j)_{j \in \mathbb{Z}} : \sup |x_j| < \infty\},$$

to the vector space $l^\infty(\mathbb{Z})$ of bounded output sequences $(y_j)_{j \in \mathbb{Z}}$.

Fig. 106



$$\text{Channel/filter: } F : \ell^\infty(\mathbb{Z}) \rightarrow \ell^\infty(\mathbb{Z}) \quad , \quad (y_j)_{j \in \mathbb{Z}} = F((x_j)_{j \in \mathbb{Z}}) . \quad (4.1.1.1)$$

Definition: A channel/filter $F : \ell^\infty \rightarrow \ell^\infty(\mathbb{Z})$ is called **finite**, if every input signal of finite duration produces an output signal of finite duration,

$$\{\exists M \in \mathbb{N} : |j| > M \Rightarrow x_j = 0\} \Rightarrow \exists N \in \mathbb{N} : |k| > N \Rightarrow (F((x_j)_{j \in \mathbb{Z}}))_k = 0$$

Since it should not matter when exactly signals are fed into a channel, we introduce the **time shift operator** for signals. For $m \in \mathbb{Z}$:

$$S_m : \ell^\infty(\mathbb{Z}) \rightarrow \ell^\infty(\mathbb{Z}), \quad S_m((x_j)_{j \in \mathbb{Z}}) = (x_{j-m})_{j \in \mathbb{Z}}.$$

Hence, by applying S_m we delay a signal by $|m| \cdot \Delta t$.

Definition: A filter is called **time-invariant (TI)**, if shifting the input time leads to the same output shifted in time by the same amount:

$$\forall (x_j)_{j \in \mathbb{Z}} \in \ell^\infty(\mathbb{Z}), \forall m \in \mathbb{Z} : F(S_m((x_j)_{j \in \mathbb{Z}})) = S_m(F((x_j)_{j \in \mathbb{Z}})).$$

Definition: A filter $F : \ell^\infty(\mathbb{Z}) \rightarrow \ell^\infty(\mathbb{Z})$ is called **causal**, if the output does not start before the input:

$$\forall M \in \mathbb{N} : (x_j)_{j \in \mathbb{Z}} \in \ell^\infty(\mathbb{Z}), x_j = 0 \quad \forall j \leq -M \Rightarrow F((x_j)_{j \in \mathbb{Z}})_k = 0 \quad \forall k \leq -M.$$

With the above definitions we can state the following acronym:

- **LT-FIR:** finite, linear, time-invariant, and causal filter

4.1.2 LT-FIR Linear Mappings

We aim for a precise mathematical description of the impact of a finite, time-invariant, linear, causal filter on an input signal: Let $(\dots, 0, h_0, h_1, \dots, h_{n-1}, 0, \dots)$, $n \in \mathbb{N}$, be the impulse responses of that LT-FIR $F : \ell^\infty(\mathbb{Z}) \rightarrow \ell^\infty(\mathbb{Z})$:

$$F((\delta_{j,0})_{j \in \mathbb{Z}}) = (\dots, 0, h_0, h_1, \dots, h_{n-1}, 0, \dots).$$

In compact notation we can write the non-zero components of the output signal $(y_j)_{j \in \mathbb{Z}}$ as

$$y_k = F((x_j)_{j \in \mathbb{Z}})_k = \sum_{j=0}^{m-1} h_{k-j} x_j, \quad k = 0, \dots, m+n-2$$

$$(h_j := 0 \text{ for } j < 0 \text{ and } j \geq n).$$

Superposition of impulse responses: The output $(\dots, y_0, y_1, y_2, \dots)$ of a LT-FIR 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 $j\Delta t$ time-shifted impulse

responses.

Definition: Given two sequences $(h_k)_{k \in \mathbb{Z}}$, $(x_k)_{k \in \mathbb{Z}}$, at least one of which is finite or decays sufficiently fast, their **convolution** is another sequence $(y_k)_{k \in \mathbb{Z}}$, defined as

$$y_k = \sum_{j \in \mathbb{Z}} h_{k-j} x_j, \quad k \in \mathbb{Z}.$$

If well-defined, the convolution of sequences commutes

$$y_k = (x_k) * (h_k) = (h_k) * (x_k).$$

4.1.3 Discrete Convolutions

Given $x = [x_0, \dots, x_{m-1}]^T \in \mathbb{K}^m$, $h = [h_0, \dots, h_{n-1}]^T \in \mathbb{K}^n$, their **discrete convolution** is the vector $y \in \mathbb{K}^{m+n-1}$ with components

$$y_k = \sum_{j=0}^{m-1} h_{k-j} x_j, \quad k = 0, \dots, n+m-1,$$

where we have adopted the convention $h_j := 0$ for $j < 0$ or $j \geq n$. We denote a discrete convolution by $y = h * x$.

4.1.4 Periodic Convolutions

An **n -periodic signal**, $n \in \mathbb{N}$, is a sequence $(x_j)_{j \in \mathbb{Z}} \in l^\infty(\mathbb{Z})$ satisfying

$$x_{j+n} = x_j \quad \forall j \in \mathbb{Z}.$$

Though infinite, an n -periodic signal $(x_j)_{j \in \mathbb{Z}}$ is *uniquely determined by the finitely many values* x_0, \dots, x_{n-1} and can be associated with a vector $x = [x_0, \dots, x_{n-1}]^T \in \mathbb{R}^n$.

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

$$(y_k) := (p_k) * (x_k), \quad y_k := \sum_{j=0}^{n-1} p_{k-j} x_j = \sum_{j=0}^{n-1} x_{k-j} p_j, \quad k \in \mathbb{Z}.$$

We denote a discrete periodic convolution by $(p_k) *_n (x_k)$.

Definition: A matrix $\mathbf{C} = [c_{ij}]_{i,j=1}^n \in \mathbb{K}^{n,n}$ is **circulant** if

$$\exists (p_k)_{k \in \mathbb{Z}} \text{ } n\text{-periodic sequence: } c_{ij} = p_{j-i}, \quad 1 \leq i, j \leq n$$

4.2 Discrete Fourier Transform (DFT)

4.2.1 Diagonalizing Circulant Matrices

We introduce the following notation: The n -th root of unity is defined as $\omega_n := \exp(\frac{-2\pi i}{n}) = \cos(\frac{2\pi}{n}) - i \sin(\frac{2\pi}{n})$, $n \in \mathbb{N}$. The n -th root of unity satisfies the following properties:

- $\bar{\omega}_n = \omega_n^{-1}$

- $\omega_n^n = 1$
- $\omega_n^{n/2} = -1$
- $\omega_n^k = \omega_n^{k+n}, \forall k \in \mathbb{Z}$

We consider a general circulant matrix $C \in \mathbb{C}^{n,n}$, with $c_{ij} := (C)_{i,j} = u_{i-j}$, for an n -periodic sequence $(u_k)_{k \in \mathbb{Z}}, u_k \in \mathbb{C}$. We define the following vector:

$$v_k \in \mathbb{C}^n : v_k := \left[\omega_n^{-jk} \right]_{j=0}^{n-1}, k \in \{0, \dots, n-1\}.$$

Then it holds, that v_k is an **eigenvector** of C for eigenvalue $\lambda_k = \sum_{l=0}^{n-1} u_l \omega_n^{lk}$. The set $\{v_0, \dots, v_{n-1}\} \subset \mathbb{C}^n$ provides the so-called **orthogonal trigonometric basis** of \mathbb{C}^n .

Definition: The matrix effecting the change of basis from the *trigonometric basis* to the *standard basis* is called the **Fourier-matrix**:

$$\mathbf{F}_n = \begin{bmatrix} \omega_n^0 & \omega_n^0 & \dots & \omega_n^0 \\ \omega_n^0 & \omega_n^1 & \dots & \omega_n^{n-1} \\ \omega_n^0 & \omega_n^2 & \dots & \omega_n^{n-2} \\ \vdots & \vdots & & \vdots \\ \omega_n^0 & \omega_n^{n-1} & \dots & \omega_n^{(n-1)^2} \end{bmatrix} = \left[\omega_n^{lj} \right]_{l,j=0}^{n-1} \in \mathbb{C}^{n,n}.$$

Lemma: The scaled Fourier-matrix $\frac{1}{\sqrt{n}} F_n$ is unitary: $F_n^{-1} = \frac{1}{n} F_n^H = \frac{1}{n} \overline{F}_n$.

Lemma: For any circulant matrix $C \in \mathbb{K}^{n,n}$, $c_{ij} = u_{i-j}$, $(u_k)_{k \in \mathbb{Z}}$ n -periodic sequence, holds true

$$C \overline{F}_n = \overline{F}_n \text{diag}(d_1, \dots, d_n), \quad [d_0, \dots, d_{n-1}]^T = F_n [u_0, \dots, u_{n-1}]^T.$$

Definition: The linear map $\mathbf{DFT}_n : \mathbb{C}^n \rightarrow \mathbb{C}^n$, $\mathbf{DFT}(y) := F_n y$, $y \in \mathbb{C}^n$, is called **discrete Fourier transform (DFT)**, i.e. for $[c_0, \dots, c_{n-1}] := \mathbf{DFT}_n(y)$

$$c_k = \sum_{j=0}^{n-1} y_j \omega_n^{kj}, \quad k \in 0, \dots, n-1.$$

Discrete Fourier transform in Eigen

The Eigen-functions for discrete Fourier transform and its inverse are given by

- DFT: `c = fft.fwd(y)`
- inverse DFT: `y = fft.inv(c)`

Before using `fft`, remember to `#include <unsupported/Eigen/FFT>`.

```
int main() {
    using Comp = complex<double>;
    const VectorXd::Index n = 5;
    VectorXcd y(n), c(n), x(n);
    y << Comp(1, 0), Comp(2, 1), Comp(3, 2), Comp(4, 3), Comp(5, 4);
    FFT<double> fft; // DFT transform object
    c = fft.fwd(y); // DFT of y
    x = fft.inv(c); // inverse DFT of c

    cout << "y = " << y.transpose() << endl
}
```



```

    << "c = " << c.transpose() << endl
    << "x = " << x.transpose() << endl;
    return 0;
}

```

4.2.2 Discrete Convolution via Discrete Fourier Transform

Discrete periodic convolution: straightforward implementation

```

Eigen::VectorXcd pconv(const Eigen::VectorXcd &u, const Eigen::VectorXcd &x) {
    const int n = x.size();
    Eigen::VectorXcd z = VectorXcd::Zero(n);
    // native two loop implementation of discrete periodic convolution
    for(int k = 0; k < n; ++k) {
        for(int i = 0, l = k; j <= k; ++j, --l) {
            z[k] += u[l] * x[j];
        }

        for(int j = k+1, l = n; j < n; ++j, --l) {
            z[k] += u[l] * x[j];
        }
    }
    return z;
}

```

Convolution Theorem: The discrete periodic convolution $*_n$ between n -dimensional vector u and x is equal to the inverse DFT of the component-wise product between the DFTs of u and x , i.e.:

$$(u) *_n (x) := \left[\sum_{j=0}^{n-1} u_{(k-j) \bmod n} x_j \right]_{k=0}^{n-1} = F_n^{-1} [(F_n u)_j (F_n x)_j]_{j=1}^n.$$

Discrete periodic convolution: DFT implementation

```

Eigen::VectorXcd pconvfft(const Eigen::VectorXcd &u, const Eigen::VectorXcd &x){
    Eigen::FFT<double> fft;
    return fft.inv(((fft.fwd(u)).cwiseProduct(fft.fwd(x))).eval());
}

```

4.2.5 Two-dimensional DFT

In this section we study the frequency decomposition of matrices. Due to the natural analogy

- one-dimensional data ("audio signal") \rightarrow vector $y \in \mathbb{C}^n$,
- two-dimensional data ("image") \rightarrow matrix $Y \in \mathbb{C}^{m,n}$,

these techniques are of fundamental importance for *image processing*.

Definition: We can state the **two-dimensional discrete Fourier transform** of the matrix $Y \in \mathbb{C}^{m,n}$ as follows:

$$C = F_m (F_n Y^T)^T = F_m Y F_n.$$

We abbreviate it by **DFT** $_{m,n} : \mathbb{C}^{m,n} \rightarrow \mathbb{C}^{m,n}$. We state the **inversion formula** as follows:

$$Y = F_m^{-1} C F_n^{-1} = \frac{1}{mn} \bar{F}_m C \bar{F}_n$$

Two-dimensional discrete Fourier transform

```
template <typename Scalar>
void fft2(Eigen::MatrixXcd &C, const Eigen::MatrixBase<Scalar> &Y) {
    using idx_t = Eigen::MatrixXcd::Index;
    const idx_t m = Y.rows(), n = Y.cols();
    C.resize(m, n);
    Eigen::MatrixXcd tmp(m, n);

    Eigen::FFT<double> fft; //Helper class for DFT
    // Transform rows of matrix Y
    for(idx_t k = 0; k < m; k++) {
        Eigen::VectorXcd tv(Y.row(k));
        tmp.row(k) = fft.fwd(tv).transpose();
    }

    // Transform columns of temporary matrix
    for(idx_t k = 0; k < n; k++) {
        Eigen::VectorXcd tv(tmp.col(k));
        C.col(k) = fft.fwd(tv);
    }
}
```

Theorem: For any $X, Y \in \mathbb{C}^{m,n}$, we have

$$X *_m Y = \text{DFT}_{m,n}^{-1}(\text{DFT}_{m,n}(X) \odot \text{DFT}_{m,n}(Y)),$$

where \odot stands for the entrywise multiplication of matrices of equal size.

This suggests the following DFT-based algorithm for evaluating the periodic convolution of matrices:

1. Compute \hat{Y} by inverse 2D DFT of \overline{Y}
2. Compute $\widehat{\hat{Y}}$ by 2D DFT of \hat{Y}
3. Component-wise multiplication of \hat{X} and \hat{Y} : $\hat{Z} = \hat{X} * \hat{Y}$.
4. Compute Z through inverse 2D DFT of \hat{Z} .

4.3 Fast Fourier Transform (FFT)

To understand how the discrete Fourier transform of n -vectors can be implemented with an asymptotic computational effort smaller than $O(n^2)$ we start with an elementary manipulation for $n = 2m$, $m \in \mathbb{N}$:

$$\begin{aligned} c_k &= \sum_{j=0}^{n-1} y_j e^{-\frac{2\pi i}{n} jk} = \sum_{j=0}^{m-1} y_{2j} e^{-\frac{2\pi i}{n} 2jk} + \sum_{j=0}^{m-1} y_{2j+1} e^{-\frac{2\pi i}{n} (2j+1)k} \\ &= \sum_{j=0}^{m-1} y_{2j} e^{-\frac{2\pi i}{m} jk} + e^{-\frac{2\pi i}{n} k} \cdot \sum_{j=0}^{m-1} y_{2j+1} e^{-\frac{2\pi i}{m} jk}, \quad k \in \mathbb{Z}. \end{aligned}$$

This means that for even n we can compute $\text{DFT}_n(y)$ from DFTs of half the length plus $\sim n$ additions and multiplications.

The asymptotic complexity of the FFT algorithm for $n = 2^L$ is $O(L \cdot 2^L) = O(n \log_2 n)$.

5. Data Interpolation and Data Filtering in 1D

5.1 Abstract Interpolation

Definition: One-dimensional data interpolation

- Given: data points (t_i, y_i) , $i = 0, \dots, n$, $n \in \mathbb{N}$, $t_i \in I \subset \mathbb{R}$, $y_i \in \mathbb{R}$
- Objective: Reconstruction of a function $f : I \rightarrow \mathbb{R}$
 1. satisfying the $n + 1$ **interpolation conditions (IC)** $f(t_i) = y_i$, $i = 0, \dots, n$
 2. and belonging to a set V of eligible functions

The function f we find is called the **interpolant** of the given data set $\{(t_i, y_i)\}_{i=0}^n$.

When we talk about **interpolation schemes** in 1D, we mean a mapping

$$I : \mathbb{R}^{n+1} \times \mathbb{R}^{n+1} \rightarrow \{f : I \rightarrow \mathbb{R}\}, ([t_i]_{i=0}^n, [y_i]_{i=0}^n) \rightarrow \text{interpolant}$$

In the context of numerical methods, "function" should be read as "subroutine", a piece of code that can, for any $x \in I$, compute $f(x)$ in finite time.

C++ data type representing a real-valued function

```
class Function{
private;
    // various internal data describing f
public;
    // Constructor: expects information for specifying the function
    Function(/*...*/);
    // Evaluating operator
    double operator() (double t) const;
};
```

C++ class representing an interpolant in 1D

```
class Interpolant {
private;
    // various internal data describing f
    // can be the coefficients of a basis representation
public;
    // constructor: computation of coefficients c_j
    Interpolant(const vector<double> &t, const vector<double> &y);
    // evaluation operator for interpolant f
    double operator() (double t) const;
};
```

Definition: A basis $\{b_0, \dots, b_n\}$ of an $n + 1$ -dimensional vector space of functions $f : I \subset \mathbb{R} \rightarrow \mathbb{R}$ is a **cardinal basis** with respect to the set $\{t_0, \dots, t_n\} \subset I$ of nodes,

$$b_j(t_i) = \delta_{ij} := 1, \text{ if } i = j, 0 \text{ else.}$$

We consider the setting for interpolation that the interpolant belongs to a finite-dimension space V_m of functions spanned by basis functions b_0, \dots, b_m . Then the interpolation conditions imply that the basis expansion coefficients satisfy a linear system of equations:

$$f(t_i) = \sum_{j=0}^m c_j b_j(t_i) = y_i, \quad i = 0, \dots, n$$

$$\Longleftrightarrow Ac := \begin{bmatrix} b_0(t_0) & \cdots & b_m(t_0) \\ \vdots & & \vdots \\ b_0(t_n) & \cdots & b_m(t_n) \end{bmatrix} \begin{bmatrix} c_0 \\ \vdots \\ c_m \end{bmatrix} = \begin{bmatrix} y_0 \\ \vdots \\ y_n \end{bmatrix} =: y$$

5.2 Global Polynomial Interpolation

Global polynomial interpolation, that is, interpolation into spaces of functions spanned by polynomials, is the simplest interpolation scheme and of great importance as building block for complex algorithms.

Compared to the piecewise linear basis functions from the previous section, our functions isn't 0 outside of the given points.

5.2.1 Uni-Variante Polynomials

Polynomials in a single variable are familiar and simple objects:

- Notation: Vector space of the **uni-variate polynomials** of degree $\leq k$, $k \in \mathbb{N}$:

$$\mathcal{P}_k := \{t \rightarrow a_k t^k + a_{k-1} t^{k-1} + \cdots + a_1 t + a_0, a_i \in \mathbb{R}\}.$$

- Terminology: The functions $t \rightarrow t^k$, $k \in \mathbb{N}_0$, are called **monomials** and the formula $t \rightarrow a_k t^k + a_{k-1} t^{k-1} + \cdots + a_1 t + a_0$ is the **monomial representation** of a polynomial.

Dimension of space of polynomials

$$\dim \mathcal{P}_k = k + 1 \text{ and } \mathcal{P}_k \subset C^\infty(\mathbb{R}).$$

Efficient evaluation of a polynomial in monomial representation is achieved through the **Horner scheme** as indicated by the following representation:

$$p(t) = t(\cdots t(t(a_n t + a_{n-1}) + a_{n-2}) + \cdots + a_1) + a_0.$$

This allows us to evaluate a polynomial in $O(n)$.

Horner scheme (vectorized version)

```
Eigen::VectorXd horner(const Eigen::VectorXd &p, const Eigen::VectorXd &t) {
    const VectorXd::Index n = t.size();
    Eigen::VectorXd y{p[0] * VectorXd::Ones(n)};
    for(unsigned i = 1; i < p.size(); ++i) {
        y = t.cwiseProduct(y) + p[i] * VectorXd::Ones(n);
    }
    return y;
}
```

5.2.2 Polynomial Interpolation: Theory

In the previous section (5.2.1) we extended the local interpolation (5.1) to a global interpolation. This caused us to loose the cardinal basis property, making the computation of the coefficients $\alpha_i (\equiv c_j)$ a lot harder. Now we introduce the Lagrange polynomial which brings back this cardinal basis property.

Lagrange Polynomial Interpolation problem (LIP)

Given the set of **interpolation nodes** $\{t_0, \dots, t_n\} \subseteq \mathbb{R}$, $n \in \mathbb{N}$, and the value $y_0, \dots, y_n \in \mathbb{R}$ compute $p \in \mathcal{P}_n$ such that it satisfies the **interpolant conditions (IC)**

$$p(t_j) = y_j \text{ for } j = 0, \dots, n.$$

For a given set $\{t_0, t_1, \dots, t_n\} \subset \mathbb{R}$ of nodes consider the

$$\textbf{Lagrange polynomials } L_i(t) := \prod_{j=0, j \neq i}^n \frac{t - t_j}{t_i - t_j}, i = 0, \dots, n.$$

It is obvious that $L_i \in \mathcal{P}_n$ and that $L_i(t_j) = \delta_{ij}$. Further Lagrange polynomials are linearly independent.

The Lagrange polynomial interpolation 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 $\{t_i\}_{i=0}^n$:

$$p(t) = \sum_{i=0}^n y_i L_i(t) \iff p \in \mathcal{P}_n \text{ and } p(t_i) = y_i.$$

The general LPI problem admits a **unique** solution $p \in \mathcal{P}_n$ for any set of data points $\{(t_i, y_i)\}_{i=0}^n$ and $n \in \mathbb{N}$.

The polynomial interpolation in the nodes $\mathcal{T} := \{t_j\}_{j=0}^n$ defines a linear operator

$$I_{\mathcal{T}} : \mathbb{R}^{n+1} \rightarrow \mathcal{P}_n, (y_0, \dots, y_n)^T \rightarrow \text{interpolating polynomial } p.$$

5.2.3 Polynomial Interpolation: Algorithms

We are given the following setting:

- Given: nodes $\mathcal{T} := \{-\infty < t_0 < t_1 < \dots < t_n < \infty\}$, values $y := \{y_0, y_1, \dots, y_n\}$
- Notation: we write $p := I_{\mathcal{T}}(y)$ for the unique Lagrange polynomial interpolant.

When used in a numerical code, different demands can be made for a class that implements Lagrange interpolants. These demands determine, which algorithm is most suitable for constructors and the evaluation operators.

In the following part we will look at two algorithms.

5.2.3.1 Multiple evaluations

Task: For ❶ a **fixed** set $\{t_0, \dots, t_n\}$ of nodes,
and ❷ **many** different given data values $y_i, i = 0, \dots, n$
and ❸ **many** arguments $x_k, k = 1, \dots, N, N \gg 1$,
efficiently compute all $p(x_k)$ for $p \in \mathcal{P}_n$ interpolating in $(t_i, y_i), i = 0, \dots, n$.

The definition of a possible interpolator data type could be as follows:

```
class PolyInterp {
private: // various internal data describing p
    Eigen::VectorXd t;
public: // constructors taking node vector (t_0, ..., t_n) as argument
    PolyInterp(const Eigen::VectorXd &t);
```

```
template <typename SeqContainer> PolyInterp(const SeqContainer &v);
// Evaluation operator for data (y_0, ..., y_n);
// computes p(x_k) for x_k's passed in x
Eigen::VectorXd eval(const Eigen::VectorXd &y,
    const Eigen::VectorXd &x) const;
};
```

Barycentric interpolation formula

We want to precompute part of the Lagrange polynomial to reduce the asymptotic effort of `eval`. By some simple manipulations we end up with this:

► **Barycentric interpolation formula**

$$p(t) = \frac{\sum_{i=0}^n \frac{\lambda_i}{t - t_i} y_i}{\sum_{i=0}^n \frac{\lambda_i}{t - t_i}}.$$

Where $\lambda_i = \frac{1}{(t_i - t_0) \cdots (t_i - t_{i-1})(t_i - t_{i+1}) \cdots (t_i - t_n)}$ is independent of t and y_i . Therefore we can precompute these values!

We end up with the following complexity:

- ◆ computation of weights $\lambda_i, i = 0, \dots, n$: cost $O(n^2)$ (only once!),
- ◆ cost $O(n)$ for every subsequent evaluation of p .

⇒ total asymptotic complexity $O(Nn) + O(n^2)$

5.2.3.2 Single evaluation

Instead of evaluating at multiple point, we might only be interested in the evaluation at a single point $x \in \mathbb{R}$.

Aitken-Neville Scheme

We are still given a list of points (t_j, y_j) in a plane and want to fit a polynomial through these points. The starting point is a recursion formula for partial Lagrange interpolants, we define:

$p_{k,\ell} :=$ unique interpolating polynomial of degree $\ell - k$ through $(t_k, y_k), \dots, (t_\ell, y_\ell)$,

We easily find that:

$$\begin{aligned}
p_{k,k}(x) &\equiv y_k \quad (\text{"constant polynomial"}), \quad k = 0, \dots, n, \\
p_{k,\ell}(x) &= \frac{(x - t_k)p_{k+1,\ell}(x) - (x - t_\ell)p_{k,\ell-1}(x)}{t_\ell - t_k} \\
&= p_{k+1,\ell}(x) + \frac{x - t_\ell}{t_\ell - t_k} (p_{k+1,\ell}(x) - p_{k,\ell-1}(x)), \quad 0 \leq k \leq \ell \leq n,
\end{aligned}$$

Now the values of the partial Lagrange interpolants can be computed sequentially (based on their dependencies), expressed by the following so-called **Aitken-Neville Scheme**:

$n =$	0	1	2	3
t_0	$y_0 =: p_{0,0}(x)$	$\rightarrow p_{0,1}(x)$	$\rightarrow p_{0,2}(x)$	$\rightarrow p_{0,3}(x)$
t_1	$y_1 =: p_{1,1}(x)$	$\rightarrow p_{1,2}(x)$	$\rightarrow p_{1,3}(x)$	
t_2	$y_2 =: p_{2,2}(x)$	$\rightarrow p_{2,3}(x)$		
t_3	$y_3 =: p_{3,3}(x)$			

For the Aitken-Neville Scheme we get a runtime of $O(n^2)$.

5.2.3.3 Extrapolation to Zero

Extrapolation is interpolation with the evaluation point t outside the interval $[t_0, t_n]$. We assume $t = 0$ and that $t_i > 0$, $i = 0, \dots, n$. Of course, Lagrangian polynomial interpolation can also be used for extrapolation.

This is especially usefull, since often around 0 we encounter cancellation. To avoid this, we can compute, for a given function g , the values $g(h_i)$ with acceptable accuracy. Then we approximate $g(h)$ with a polynomial and evaluate it at position 0. I.e. we fit a polynomial through the points $(h_i, g(h_i))$ and evaluate this polynomial at the position 0. This is the exact problem we solve with Aitken-Neville.

5.2.3.4 Newton Basis and Divided Differences

This chapter we want to have an update friendly basis.

We define the Newtonbasis for \mathcal{P}_n as follows:

$$N_0(t) := 1, \quad N_1(t) := (t - t_0), \quad \dots, \quad N_n(t) := \prod_{i=0}^{n-1} (t - t_i).$$

We find the coefficients for the interpolating polynomial by a method similar to the Aitken-Neville Scheme (see the Lecture Document for more details). We end up with the following interpolating polynomial:

$$p(t) = a_0 + a_1(t - t_0) + a_2(t - t_0)(t - t_1) + \cdots + a_n \prod_{j=0}^{n-1} (t - t_j)$$

This polynomial representation already implies that we can use “backward evaluation” in the spirit of Horner Scheme.

5.2.4 Polynomial Interpolation: Sensitivity

This section addresses a *major shortcoming of polynomial interpolation*, that small perturbations of measurements gives huge errors in the function we construct.

For measuring the size of perturbations we need norms on data and result spaces. For the value vectors $y := [y_0, \dots, y_n]^T \in \mathbb{R}^{n+1}$ we can use any vector norm, for instance the maximum norm $\|y\|_\infty$. However the result space is a vector space of continuous functions $I \subset \mathbb{R} \rightarrow \mathbb{R}$ and so we also need norms on the vector space of continuous functions $C^0(I)$, $I \subset \mathbb{R}$. The following norms are the most relevant:

- supremum norm $\|f\|_{L^\infty(I)} := \sup\{|f(t)| : t \in I\}$
- L^2 -norm $\|f\|_{L^2(I)}^2 := \int_I |f(t)|^2 dt$
- L^1 -norm $\|f\|_{L^1(I)} := \int_I |f(t)| dt$

Now let $L : X \rightarrow Y$ be a linear problem map between two normed spaces, the data space X (with norm $\|\cdot\|_X$) and the result space Y (with norm $\|\cdot\|_Y$). Thanks to linearity, perturbations of the result $y := L(x)$ for the input $x \in X$ can be expressed as follows:

$$L(x + \delta x) = L(x) + L(\delta x) = y + L(\delta x).$$

Hence, the sensitivity can be measured by the **operator norm**

$$\|L\|_{X \rightarrow Y} := \sup_{\delta x \in X \setminus \{0\}} \frac{\|L(\delta x)\|_Y}{\|\delta x\|_X}.$$

Given a mesh $\mathcal{T} \subset \mathbb{R}$ with generalized Lagrange polynomials L_i , $i = 0, \dots, n$, and fixed $I \subset \mathbb{R}$, the norm of the interpolation operator satisfies

$$\begin{aligned} \|I_{\mathcal{T}}\|_{\infty \rightarrow \infty} &:= \sup_{y \in \mathbb{R}^{n+1} \setminus \{0\}} \frac{\|I_{\mathcal{T}}(y)\|_{L^\infty(I)}}{\|y\|_\infty} = \left\| \sum_{i=0}^n |L_i| \right\|_{L^\infty(I)}, \\ \|I_{\mathcal{T}}\|_{2 \rightarrow 2} &:= \sup_{y \in \mathbb{R}^{n+1} \setminus \{0\}} \frac{\|I_{\mathcal{T}}(y)\|_{L^2(I)}}{\|y\|_2} \leq \left(\sum_{i=0}^n \|L_i\|_{L^2(I)}^2 \right)^{\frac{1}{2}}. \end{aligned}$$

Definition: We define the **Lebesgue constant** of \mathcal{T} as follows:

$$\lambda_{\mathcal{T}} := \left\| \sum_{i=0}^n |L_i| \right\|_{L^\infty(I)} = \|I_{\mathcal{T}}\|_{\infty \rightarrow \infty}.$$

5.3 Shape-Preserving Interpolation

When reconstructing a quantitative dependence of quantities from measurements, first principles from physics often stipulate qualitative constraints, which translate into *shape properties* of the function f , e.g.

when modelling the material law for a gas:

→ t_i pressure values, y_i densities $\Rightarrow f$ positive and monotonic

Notation: given data is $(t_i, y_i) \in \mathbb{R}^2$, $i = 0, \dots, n$, $n \in \mathbb{N}$, $t_0 < t_1 < \dots < t_n$.

5.3.1 Shape Properties of Functions and Data

Definition: The data (t_i, y_i) are called **monotonic** when $y_i \geq y_{i-1}$ or $y_i \leq y_{i-1}$ for $i = 0, \dots, n$

Definition: The data $\{(t_i, y_i)\}_{i=0}^n$ are called **convex (concave)** if

$$\Delta_j \leq (\geq) \Delta_{j+1}, j = 1, \dots, n-1, \quad \Delta_j = \frac{y_j - y_{j-1}}{t_j - t_{j-1}}, j = 1, \dots, n.$$

5.3.2 Piecewise Linear Interpolation

There is a very simple method of achieving perfect shape preservation by means of a linear interpolation operator into the space of continuous functions. Given the following data: $(t_i, y_i) \in \mathbb{R}^2$, $i = 0, \dots, n$, $n \in \mathbb{N}$, $t_0 < t_1 < \dots < t_n$.

Then the **piecewise linear interpolant** $s : [t_0, t_n] \rightarrow \mathbb{R}$ is defined as

$$s(t) = \frac{(t_{i+1} - t)y_i + (t - t_i)y_{i+1}}{t_{i+1} - t_i} \text{ for } t \in [t_i, t_{i+1}].$$

The piecewise linear interpolant is also called a *polygonal curve*. It is continuous and consists of n line segments.

Theorem: Let $s \in C([t_0, t_n])$ be the piecewise linear interpolant of $(t_i, y_i) \in \mathbb{R}^2$, $i = 0, \dots, n$, for every subinterval $I = [t_j, t_k] \subset [t_0, t_n]$:

- if $(t_i, y_i)|_I$ are positive / negative $\Rightarrow s|_I$ is positive / negative
- if $(t_i, y_i)|_I$ are monotonic $\Rightarrow s|_I$ is monotonic
- if $(t_i, y_i)|_I$ are convex/concave $\Rightarrow s|_I$ is convex/concave

5.3.3 Cubic Hermite Interpolation

Aim: local shape-preserving (linear) interpolation operator that fixes short-coming of piecewise linear interpolation by ensuring C^1 -smoothness of the interpolant.

5.3.3.1 Definition and Algorithms

Given: Mesh points $(t_i, y_i) \in \mathbb{R}^2$, $i = 0, \dots, n$, $t_0 < t_1 < \dots < t_n$.

Goal: build function $f \in C^1([t_0, t_n])$ satisfying the interpolant conditions $f(t_i) = y_i$, $i = 0, \dots, n$.

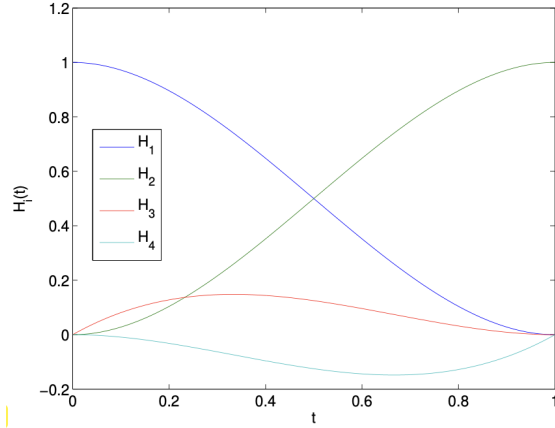
Definition: Given data points $(t_i, y_i) \in \mathbb{R} \times \mathbb{R}$, $j = 0, \dots, n$ with pairwise distinct ordered nodes t_j , and **slopes** $c_j \in \mathbb{R}$, the piecewise **cubic Hermite interpolants** $s : [t_0, t_n] \rightarrow \mathbb{R}$ is defined by the requirements

$$s|_{[t_{i-1}, t_i]} \in \mathcal{P}_3, i = 1, \dots, n, s(t_i) = y_i, s'(t_i) = c_i, i = 0, \dots, n.$$

Locally, we can write a piecewise cubic Hermit interpolant as a linear combination of generalized cardinal basis functions with coefficients based on y_i and c_i (representing the slope at point i).

$$s(t) = y_{i-1}H_1(t) + y_iH_2(t) + c_{i-1}H_3(t) + c_iH_4(t) \quad , \quad t \in [t_{i-1}, t_i] \quad ,$$

The basis functions $H_k, k = 1, 2, 3, 4$ are defined as follows:



Local basis polynomials on $[0, 1]$

$$H_1(t) := \phi\left(\frac{t_i - t}{h_i}\right) \quad , \quad (5.3.3.5a)$$

$$H_2(t) := \phi\left(\frac{t - t_{i-1}}{h_i}\right) \quad , \quad (5.3.3.5b)$$

$$H_3(t) := -h_i\psi\left(\frac{t_i - t}{h_i}\right) \quad , \quad (5.3.3.5c)$$

$$H_4(t) := h_i\psi\left(\frac{t - t_{i-1}}{h_i}\right) \quad , \quad (5.3.3.5d)$$

$$h_i := t_i - t_{i-1} \quad , \quad (5.3.3.5e)$$

$$\phi(\tau) := 3\tau^2 - 2\tau^3 \quad , \quad (5.3.3.5f)$$

$$\psi(\tau) := \tau^3 - \tau^2 \quad . \quad (5.3.3.5g)$$

By some computation we find that the following holds for H_k :

	$H(t_{i-1})$	$H(t_i)$	$H'(t_{i-1})$	$H'(t_i)$
H_1	1	0	0	0
H_2	0	1	0	0
H_3	0	0	1	0
H_4	0	0	0	1

However, the data only specifies y_i and not c_i . Thus to compute an interpolant, we need to supply a way of computing these c_i . One way of doing so would be to use the following linear mapping:

$$c_i = \begin{cases} \Delta_1 & , \text{ for } i = 0 \quad , \\ \Delta_n & , \text{ for } i = n \quad , \\ \frac{t_{i+1} - t_i}{t_{i+1} - t_{i-1}} \Delta_i + \frac{t_i - t_{i-1}}{t_{i+1} - t_{i-1}} \Delta_{i+1} & , \text{ if } 1 \leq i < n \quad . \end{cases} \quad , \quad \Delta_j := \frac{y_j - y_{j-1}}{t_j - t_{j-1}} \quad , j = 1, \dots, n \quad .$$

5.3.3.2 Local Monotonicity-Preserving Hermite Interpolation

When choosing the slopes like above, the Hermite interpolation does not preserve monotonicity. Therefore we introduce an alternative way of choosing the slopes, that preserves monotonicity.

$$\text{sgn}(\Delta_1) = \text{sgn}(\Delta_2) \rightarrow c_i = \begin{cases} \Delta_1 & , \text{ if } i = 0 \quad , \\ \frac{3(h_{i+1} + h_i)}{2h_{i+1} + h_i + \frac{2h_i + h_{i+1}}{\Delta_i + \Delta_{i+1}}} \Delta_i & , \text{ for } i \in \{1, \dots, n-1\} \quad , \quad h_i := t_i - t_{i-1} \quad . \\ \Delta_n & , \text{ if } i = n \quad , \end{cases}$$

Theorem: If, for fixed node set $\{t_j\}_{j=0}^n$, $n \geq 2$, an interpolation scheme $I : \mathbb{R}^{n+1} \rightarrow C^1(I)$ is *linear* as a mapping from data values to continuous functions on the interval covered by the nodes, and *monotonicity preserving*, then $I(y)'(t_j) = 0$ for all $y \in \mathbb{R}^{n+1}$ and $j = 1, \dots, n-1$.

5.4 Splines

Definition: Given an interval $I := [a, b] \subset \mathbb{R}$ and a **knot sequence** $\mathcal{M} := \{a = t_0 < t_1 < \dots < t_n = b\}$, $n \in \mathbb{N}$, the vector space $\mathcal{S}_{d, \mathcal{M}}$ of the **spline functions** of degree d (or order $d+1$) is defined by

$$\mathcal{S}_{d, \mathcal{M}} := \{s \in C^{d-1}(I) : s_j := s|_{[t_{j-1}, t_j]} \in \mathcal{P}_d \forall j = 1, \dots, n\}.$$

The **dimension of a spline space** can be found by a *counting argument*: We count the number of "degrees of freedom" possessed by a \mathcal{M} -piecewise polynomial of degree d , and subtract the number of *linear constraints*:

$$\dim \mathcal{S}_{d, \mathcal{M}} = n \cdot \dim \mathcal{P}_d - \#\{C^{d-1} \text{ continuity constraints}\} = n \cdot (d+1) - (n-1) \cdot d = n + d.$$

5.4.2 Cubic-Spline Interpolation

Task: Given a mesh $\mathcal{M} := \{t_0 < t_1 < \dots < t_n\}$, $n \in \mathbb{N}$, "find" a cubic spline $s \in \mathcal{S}_{3, \mathcal{M}}$ that compiles with the interpolation conditions

$$s(t_j) = y_j, \quad j = 0, \dots, n.$$

When comparing the cubic spline interpolation to the cubic Hermit interpolation, we find that the cubic Hermit interpolation allows for $n+1$ degrees of freedom, while the cubic spline interpolation only allows 2 degrees of freedom. This is due to the fact that the interpolant has to be in C^2 and not only in C^1 .

To saturate the remainign two degree of freedom the following three approaches are popular:

1. Complete cubic spline interpolation: $s'(t_0) = c_0$, $s'(t_n) = c_n$ prescribed
2. Natural cubic spline interpolation: $s''(t_0) = s''(t_n) = 0$
3. Periodic cubic spline interpolation: $s'(t_0) = s'(t_n)$, $s''(t_0) = s''(t_n)$

<https://www.youtube.com/watch?v=pLfifROQ-MM>

Great video explaining cubic splines and some of their properties, including how to fix the remaining two degrees of freedom.

5.4.3 Structural Properties of Cubic Spline Interpolation

For a function $f : [a, b] \rightarrow \mathbb{R}$, $f \in C^2([a, b])$, the term

$$E_{\text{bd}}(f) := \frac{1}{2} \int_a^b |f''(t)|^2 dt,$$

models the elastic bending energy of a rod, whose shape is described by the graph of f . We will show the cubic spline interpolants have minimal bending energy among all C^2 -smooth interpolating functions.

Given: mesh $\mathcal{M} := \{a = t_0 < t_1 < \dots < t_n = b\}$ of $[a, b]$ with knots t_j

Set $s \in \mathcal{S}_{3, \mathcal{M}} :=$ natural cubic spline interpolant of data points $(t_i, y_i) \in \mathbb{R}^2, i = 0, \dots, n$.

Theorem: The natural cubic spline interpolant s minimizes the elastic curve energy among all interpolating functions in $C^2([a, b])$, that is

$$E_{\text{bd}}(S) \leq E_{\text{bd}}(f) \forall f \in C^2([a, b]), f(t_i) = y_i, i = 0, \dots, n.$$

5.6 Trigonometric Interpolation

We consider time series data $(t_i, y_i), i = 0, \dots, n, t_i, y_i \in \mathbb{R}$, obtained by sampling a time-dependent scalar physical quantity $t \rightarrow \phi(t)$. We know that ϕ is a **T-periodic** function with period $T > 0$, that is $\phi(t) = \phi(t + T)$ for all $t \in \mathbb{R}$. In the spirit of shape preservation an interpolant f of the time series should also be T -periodic: $f(t + T) = f(t)$ for all $t \in \mathbb{R}$.

Assumption: We assume the period $T > 0$ to be known and $t_i \in [0, T[$ for all interpolation nodes $t_i, i = 0, \dots, n$.

In the sequel, for the case of simplicity, we consider only $T = 1$.

Task: Given $T > 0$ and data points $(t_i, y_i), y_i \in \mathbb{K}, t_i \in [0, T[$, find a **T-periodic** function $f : \mathbb{R} \rightarrow \mathbb{K}$ (the interpolant), $f(t + T) = f(t) \forall t \in \mathbb{R}$, that satisfies the **interpolation conditions**

$$f(t_i) = y_i, i = 0, \dots, n.$$

5.6.1 Trigonometric Polynomials

The most fundamental periodic functions are derived from the trigonometric functions \sin and \cos and dilations of them. A **dilation** of a function $t \rightarrow \phi(t)$ is a function of the form $t \rightarrow \phi(ct)$ with some $c > 0$.

Definition: The vector space of 1-periodic **trigonometric polynomials** of degree $2n, n \in \mathbb{N}$, is given by

$$\mathcal{P}_{2n}^T := \text{Span}\{t \rightarrow \cos(2\pi jt), t \rightarrow \sin(2\pi jt)\}_{j=0}^{2n} \subset C^\infty(\mathbb{R}).$$

We can rewrite $q \in \mathcal{P}_{2n}^T$ given in the form

$$q(t) = \alpha_0 + \sum_{j=1}^{2n} \alpha_j \cos(2\pi jt) + \beta_j \sin(2\pi jt), \alpha_j, \beta_j \in \mathbb{R}.$$

Further manipulations give us:

$$q \in \mathcal{P}_{2n}^T \Rightarrow q(t) = e^{-2\pi i n t} \cdot p(e^{2\pi i t}) \text{ with } p(z) = \sum_{j=0}^{2n} \gamma_j z^j \in \mathcal{P}_{2n},$$

5.6.2 Reduction to Lagrange Interpolation

Trigonometric interpolation through data points $(t_k, y_k), k = 0, \dots, 2n$ \iff Lagrange polynomial interpolation through data points $(e^{2\pi i t_k}, e^{2\pi i n t_k} y_k), k = 0, \dots, 2n$

We can reuse the already known algorithms for Lagrange polynomial interpolation.

5.6.3 Equidistant Trigonometric Interpolation

Often timeseries data for a time-periodic quantity are measured with a constant rhythm over the entire period of duration $T > 0$, that is, $t_j = j\Delta t, \Delta t = \frac{T}{n+1}, j = 0, \dots, n$. In this case, the formulas for

computing coefficients of the interpolating trigonometric polynomial become special versions of the **discrete Fourier transform (DFT)**. Efficient implementation can thus harness the speed of *FFT*.

Now we consider trigonometric interpolation in the 1-periodic setting with $2n + 1$ *uniformly distributed* interpolation nodes $t_k = \frac{k}{2n+1}$, $k = 0, \dots, 2n$, and associated data values y_k . Existence and uniqueness of an interpolating trigonometric polynomial $q \in \mathcal{P}_{2n}^T$, $q(t_k) = y_k$, was established earlier. We rely on the following relation ship:

$$q \in \mathcal{P}_{2n}^T \Rightarrow q(t) = e^{-2\pi i n t} \cdot p(e^{2\pi i t}) \text{ with } p(z) = \sum_{j=0}^{2n} \gamma_j z^j \in \mathcal{P}_{2n},$$

to arrive at the following $(2n + 1) \times (2n + 1)$ linear system of equations for computing the unknown coefficients γ_j :

$$\bar{F}_{2n+1} c = b, c = [\gamma_0, \dots, \gamma_{2n}]^T \Rightarrow c = \frac{1}{2n+1} F_{2n+1} b.$$

5.7 Least Squares Data Fitting

When looking at interpolations, we find that having a interpolating polynomial of a high degree is often not recommendable, since it causes large fluctuations. Therefore we might want a polynomial of degree n , that approximates m data points, where $n < m$. This is equivalent to a overdetermined system of equations.

The most general task of multidimensional, vector-valued **least squares data fitting** can be described as follows:

Least square data fitting

Given: data points (t_i, y_i) , $i \in \{1, \dots, m\}$, $m \in \mathbb{N}$, $t_i \in D \subset \mathbb{R}^k$, $y_i \in \mathbb{R}^d$, $d \in \mathbb{N}$.

Objective: Find a continuous function $f : D \rightarrow \mathbb{R}^d$ in some set $S \subset C^0(D)$ of *admissible functions* satisfying

$$f \in \operatorname{argmin}_{g \in S} \sum_{i=1}^m \|g(t_i) - y_i\|_2^2.$$

Such a function f is called a **best least squares fit** for the data in S .

Consider a special variant of the general least squares data fitting problem: The set S of admissible continuous functions is now chosen as a *finite-dimensional vector space* $V_n \subset C^0(D)$, $\dim V_n = n \in \mathbb{N}$.

Choose a basis of V_n , $V_n = \operatorname{Span}\{b_1, \dots, b_n\}$, $b_j : D \rightarrow \mathbb{R}^d$ continuous.

→ The best least squares fit $f \in V_n$ can be represented by a finite linear combination of the basis functions b_j :

$$f(t) = \sum_{j=1}^n x_j b_j(t), x_j \in \mathbb{R}.$$

It can be furthermore be recast to the following problem:

General linear least squares fitting problem

Given: data points $(t_i, y_i) \in \mathbb{R}^k \times \mathbb{R}^d$, $i = 1, \dots, m$ and basis functions $b_j : D \subset \mathbb{R}^k \rightarrow \mathbb{R}$, $j = 1, \dots, n$, $n < m$.

Sought: coefficients $x_j \in \mathbb{R}$, $j = 1, \dots, n$, such that

$$x := [x_1, \dots, x_n]^T := \operatorname{argmin}_{z_j \in \mathbb{R}} \sum_{i=1}^m \left\| \sum_{j=1}^n z_j b_j(t_i) - y_i \right\|_2^2.$$

Theorem: The solution $[x_1, \dots, x_n]^T \in \mathbb{R}^n$ of the linear least squares fitting problem is the least squares solution of the overdetermined linear system of equations

$$\begin{bmatrix} A_1 \\ \vdots \\ A_d \end{bmatrix} x = \begin{bmatrix} b_1 \\ \vdots \\ b_d \end{bmatrix},$$

with

$$A_r := \begin{bmatrix} (b_1(t_1))_r & \cdots & (b_n(t_n))_r \\ \vdots & & \vdots \\ (b_1(t_m))_r & \cdots & (b_n(t_m))_r \end{bmatrix} \in \mathbb{R}^{m,n}, \quad b_r := \begin{bmatrix} (y_1)_r \\ \vdots \\ (y_m)_r \end{bmatrix} \in \mathbb{R}^m, \quad r = 1, \dots, d.$$

Lemma: The scalar one-dimensional linear least squares fitting problem with $\dim V_n = n$, V_n the vector space of admissible functions, has a unique solution, if and only if there are t_{i_1}, \dots, t_{i_n} such that

$$\begin{bmatrix} b_1(t_{i_1}) & \cdots & b_n(t_{i_1}) \\ \vdots & & \vdots \\ b_1(t_{i_n}) & \cdots & b_n(t_{i_n}) \end{bmatrix} \in \mathbb{R}^{n,n} \text{ is invertible,}$$

which is independent of the choice of basis of V_n .

8. Iterative Methods for Non-Linear Systems of Equations

8.1 Introduction

Non-linear systems naturally arise in mathematical models of electrical circuits, once non-linear circuit elements are introduced. A non-linear system of equations is a concept almost too abstract to be useful, because it covers an extremely wide variety of problems.

For a function $F : D \subset \mathbb{R}^m \rightarrow \mathbb{R}$, $n \in \mathbb{N}$, there are no general results existence and uniqueness of solutions of $F(x) = 0$.

8.2 Iterative Methods

8.2.1 Fundamental Concepts

We try to find a solution to the system $F(x) = 0$, by creating a **sequence of smart guesses** x^i . A **m -point iterative method** means that the next value in our sequence depends on the last m values. This also means that we need m **initial guesses**.

Such a sequence has the following properties,

$$x^{(k+1)} = \phi_F(x^{(k)}, \dots, x^{(k-m+1)})$$

and

$$\phi_F(x^*, \dots, x^*) = x^* \Leftrightarrow F(x^*) = 0$$

An iterative method **converges** (for fixed initial guess(es)) iff $x^{(k)} \xrightarrow{(k \rightarrow \infty)} x^*$ and $F(x^*) = 0$.

8.2.2 Speed of Convergence

We define the convergence as the speed that the sequence converges to x^* . We measure this “speed” by the following definition:

A convergent sequence $x^{(k)}$ with limit x^* converges with **order** $p, p \geq 1$, if

$$\exists C > 0 : \|x^{(k+1)} - x^*\| \leq C \cdot \|x^{(k)} - x^*\|^p \quad \forall k \in \mathbb{N}_0,$$

if $C < 1$ and $p = 1$, we call it **linear convergence**. We can approximate the order p by the following equation, where $\epsilon_k := \|x^{(k)} - x^*\|$ is the norm of the iteration error:

$$\frac{\log \epsilon_{k+1} - \log \epsilon_k}{\log \epsilon_k - \log \epsilon_{k-1}} \approx p$$

8.2.3 Termination Criteria / Stopping Rule

The termination criteria is used to determine when to stop calculating new elements of our sequence x^i . There are three general methods for termination:

1. A priori (number of steps)
2. Residual based ($F(x) = 0 \Rightarrow$ we stop when $|F(x^i)|$ is small)
3. **Correction based** (we stop when the difference between x^i and x^{i+1} is small)

Often the 3rd criteria is used, since it stops as soon as we don't make any progress anymore.

For $p = 1$ we have one more criteria: $\|x^{(k+1)} - x^*\| \leq \frac{L}{1-L} \cdot \|x^{(k)} - x^k\|, 0 < L < 1$

8.3 Fixed-Point Iteration

In this part we look at **1-point stationary iterations**, also called **fixed point iterations**. We can observe that generally a small derivative of $\phi(x)$ is good for the convergence. A first lemma gives us a condition for local convergence, that is at least linear.

Lemma 8.3.2.7. Sufficient condition for local linear convergence of fixed point iteration →
[Han02, Thm. 17.2], [DR08, Cor. 5.12]

If $\Phi : U \subset \mathbb{R}^n \mapsto \mathbb{R}^n$, $\Phi(x^*) = x^*$, Φ differentiable in x^* , and $\|D\Phi(x^*)\| < 1$, then the fixed point iteration

$$x^{(k+1)} := \Phi(x^{(k)}), \quad (8.3.1.2)$$

converges locally and at least linearly that is

matrix norm, Def. 1.5.5.10 !

$$\exists 0 \leq L < 1: \quad \|x^{(k+1)} - x^*\| \leq L \|x^{(k)} - x^*\| \quad \forall k \in \mathbb{N}_0,$$

provided that the initial guess $x^{(0)}$ belongs to some neighborhood of x^* .

notation: $D\Phi(x) \triangleq$ **Jacobian** (ger.: Jacobi-Matrix) of Φ at $x \in D \rightarrow$ [Str09, Sect. 7.6]

A second Lemma gives us a lower bound for the order of convergence.

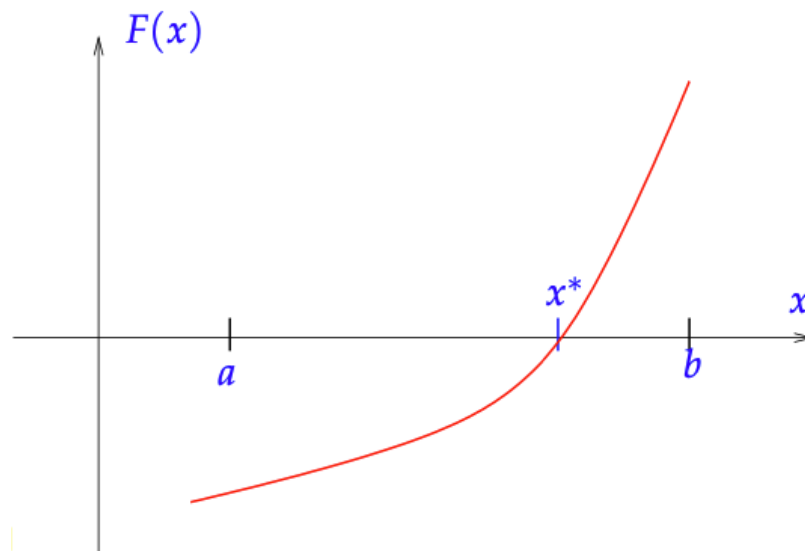
Lemma 8.3.2.15. Higher order local convergence of fixed point iterations

If $\Phi : U \subset \mathbb{R} \mapsto \mathbb{R}$ is $m+1$ times continuously differentiable, $\Phi(x^*) = x^*$ for some x^* in the interior of U , and $\Phi^{(l)}(x^*) = 0$ for $l = 1, \dots, m$, $m \geq 1$, then the fixed point iteration (8.3.1.2) converges locally to x^* with order $\geq m+1$ (\rightarrow Def. 8.2.2.10).

8.4 Finding Zeros of Scalar Functions

8.4.1 Bisection

This method is based on the idea of finding x^* by **shrinking the interval in each iteration in half**.



We start of with two points $a, b \in \mathbb{R}$ that have different signs. From there we can use the intermediate value theorem, to conclude that in between a and b there has to be a zero value. By testing the sign at the midpoint and shrinking the interval accordingly, we can find a $x^{(k)}$ that is close to x^* . This method is foolproof and works without any derivatives, but the drawback is, that it is only of “linear-type”.

8.4.2 Model Function Methods

This is a class of methods for finding zeroes of F , based on the following idea:



Given recent iterates $x^{(k)}, \dots, x^{(k-m+1)}$, $m \in \mathbb{N}$, replace F with a k -dependent **model function** \tilde{F}_k . Now $x^{(k+1)} := \text{zero of } \tilde{F}_k$.

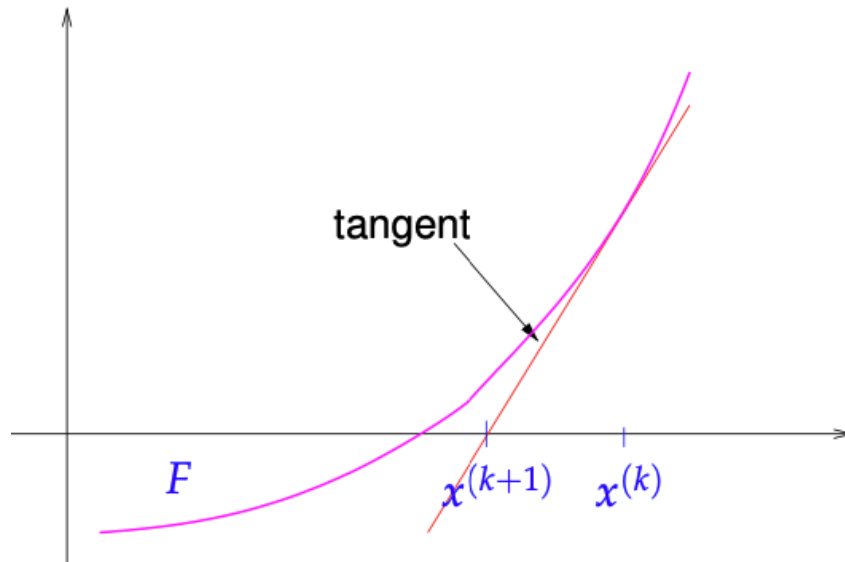
8.4.2.1 Newton Method in Scalar Case

The Newton Method is one of the most important Methods in Numerical Methods. Its formula is:

$$x^{k+1} := x^{(k)} - \frac{F(x^{(k)})}{F'(x^{(k)})}$$

We define our model function as $\tilde{F}_k(x) := F(x^{(k)}) + F'(x^{(k)})(x - x^{(k)})$. This is equal to the tangent at F in $x^{(k)}$. $x^{(k+1)}$ is now equal to the zero of the tangent and we get the Newton iteration:

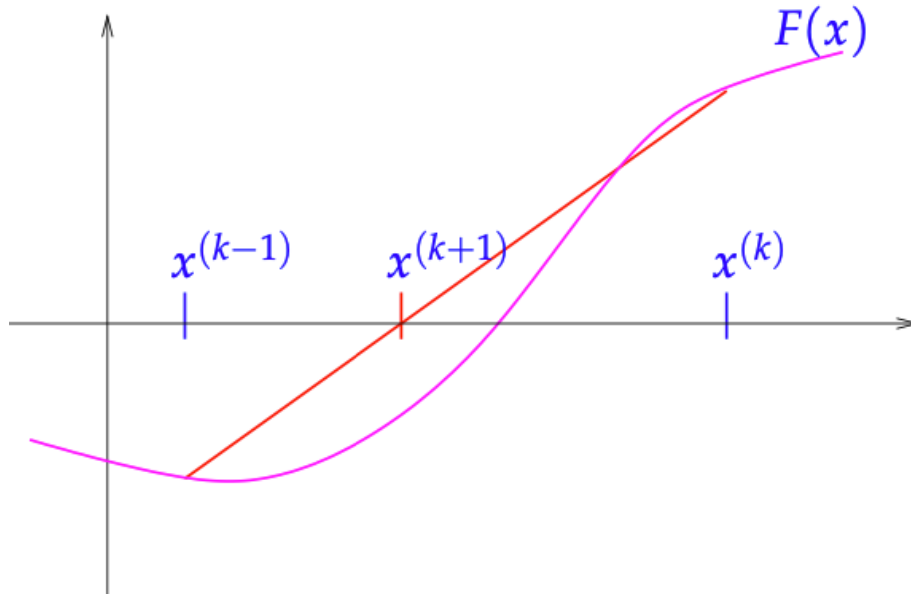
$$x^{(k+1)} := x^{(k)} - \frac{F(x^{(k)})}{F'(x^{(k)})}$$



When investigating the convergence, we find that Newton's method locally converges quadratically to a zero x^* of F , if $F'(x^*) \neq 0$.

8.4.2.3 Multi-Point Methods

The **secant method** is the simplest representative of model function multi-point methods. We just approximate the function linearly by drawing a line through the last two points we have in our approximation sequence x^i . Then we use a similar formula to the Newton method.



$$x^{(k+1)} := x^{(k)} - \frac{F(x^{(k)})(x^{(k)} - x^{(k-1)})}{F(x^{(k)}) - F(x^{(k-1)})}$$

Analysing the convergence, we find that the order of convergence is fractional, with $p \approx 1.62$.

Another class of multi-point methods are **inverse interpolation**. The approach here is, to interpolate the inverse function. We see that we can construct an interpolant of the inverse function by interpolating the points (y_i, t_i) (inverted tuple!).

$$p(F(x^{(k-j)})) = x^{(k-j)}, \quad j = 0, \dots, m-1$$

Where p is a polynomial of degree $m-1$. Then we can evaluate the interpolate for the inverse function at the point 0, $x^{(k+1)} := p(0)$. Which then is an approximation of the solution x^* we are looking for.

$$F(x^*) = 0 \Leftrightarrow F^{-1}(0) = x^*$$

As an example we have seen the case for $m = 3$:

We interpolate the points $(F(x^{(k)}), x^{(k)}), (F(x^{(k-1)}), x^{(k-1)}), (F(x^{(k-2)}), x^{(k-2)})$ with a parabola (polynomial of degree 2). Note the importance of monotonicity of F , which ensures that $F(x^{(k)}), F(x^{(k-1)}), F(x^{(k-2)})$ are mutually different.

MAPLE code:

```
p := x-> a*x^2+b*x+c;
solve({p(f0)=x0, p(f1)=x1, p(f2)=x2}, {a,b,c});
assign(%); p(0);
```

$$\blacktriangleright x^{(k+1)} = \frac{F_0^2(F_1x_2 - F_2x_1) + F_1^2(F_2x_0 - F_0x_2) + F_2^2(F_0x_1 - F_1x_0)}{F_0^2(F_1 - F_2) + F_1^2(F_2 - F_0) + F_2^2(F_0 - F_1)}.$$

$$(F_0 := F(x^{(k-2)}), F_1 := F(x^{(k-1)}), F_2 := F(x^{(k)}), x_0 := x^{(k-2)}, x_1 := x^{(k-1)}, x_2 := x^{(k)})$$

Here the interpolation and evaluation is done in one explicit formula. For such quadratic inverse interpolation, we find the fractional order of convergence to be $p \approx 1.8$.

8.4.3 Asymptotic Efficiency of Iterative Methods for Zero Finding

We can compare different methods in terms of efficiency. We define efficiency as the number of digits gained $\log(\rho)$, divided by the effort W to achieve the result.

Computational **effort** W per step

$$\text{e.g. } W \approx \frac{\#\{\text{evaluations of } DF\}}{\text{step}} + n \cdot \frac{\#\{\text{evaluations of } F'\}}{\text{step}} + \dots$$

Let $k(\rho)$ be the number of steps to achieve a relative reduction of the error by a factor of ρ (gain). Then we have

$$\text{Efficiency} := \frac{\text{\#digits gained}}{\text{total work required}} = \frac{|\log \rho|}{k(\rho) \cdot W}$$

Now we adopt an asymptotic perspective and require a large reduction of the error, that is $\rho \ll 1$.

► **asymptotic efficiency for $\rho \ll 1$** ($\rightarrow |\log \rho| \rightarrow \infty$):

$$\text{Efficiency}_{|\rho \ll 1} = \begin{cases} -\frac{\log C}{W} & , \text{ if } p = 1, \\ \frac{\log p}{W} \cdot \frac{|\log \rho|}{\log(|\log \rho|)} & , \text{ if } p > 1. \end{cases}$$

When using this to compare the secant method with Newton's method, we find that the secant method is more efficient, despite having a lower order of convergence.

8.5 Newton's Method in \mathbb{R}^n

8.5.1 The Newton Iteration

The Newton iteration seen in 8.4.2.1 can be generalized for \mathbb{R}^n by using the Jacobian of the function (We assume that F is continuously differentiable). This gives us the formula

► **Newton iteration:** (generalizes (8.4.2.1) to $n > 1$)

$$\mathbf{x}^{(k+1)} := \mathbf{x}^{(k)} - DF(\mathbf{x}^{(k)})^{-1}F(\mathbf{x}^{(k)}) \quad , \quad [\text{if } DF(\mathbf{x}^{(k)}) \text{ regular}]$$

Terminology: $-DF(\mathbf{x}^{(k)})^{-1}F(\mathbf{x}^{(k)})$ is called the **Newton correction**

Since we want to avoid calculating the inverse, we solve the LSE

$$DF(\mathbf{x}^{(k)}) \cdot \mathbf{s} = F(\mathbf{x}^{(k)})$$

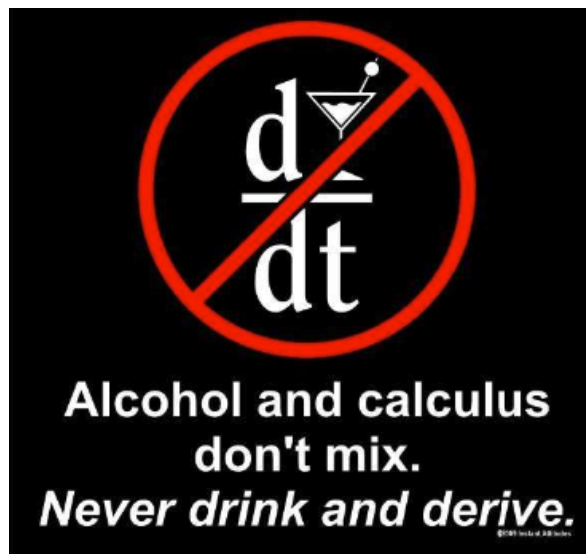
and then compute $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \mathbf{s}$. The generalized Newton iteration has the same quadratic order of convergence as the 1-D Newton iteration.

We also looked at a special variant of the Newton iteration, where we replace the derivative of the k -th iteration, by the derivative of the 0-th iteration. This is convenient, since we don't have to calculate the derivative for every step.

$$x^{(k+1)} = x^{(k)} - DF(x^{(0)})^{-1} \cdot F(x^{(k)})$$

When using the same Jacobian for all steps, we can reuse the LU-decomposition. As a drawback we end up with only linear convergence.

Remark: This image serves as a reminder, that reading the lecture document is a worthwhile endeavour. Especially, since it contains funny images like this.



8.5.3 Termination of Newton Iteration

In 8.5.2 we saw that Newton's method enjoys (asymptotic) quadratic convergence, which means rapid decrease of the relative error of the iterates, once we are close to the solution, which is exactly the point, when we want to stop. We use the **correction based termination criterion** (8.2.3), to determine when to stop.

➤ quit iterating as soon as $\|x^{(k+1)} - x^{(k)}\| = \|DF(x^{(k)})^{-1}F(x^{(k)})\| < \tau \|x^{(k)}\|$,
with τ = tolerance

This is uneconomical, as we have one needless update, because $x^{(k)}$ would already be accurate enough. Since $DF(x^{(k-1)}) \approx DF(x^{(k)})$ during the final steps, we can use the more economical termination criterion

Terminology: $\Delta \bar{x}^{(k)} := DF(x^{(k-1)})^{-1}F(x^{(k)}) \triangleq$ **simplified Newton correction**

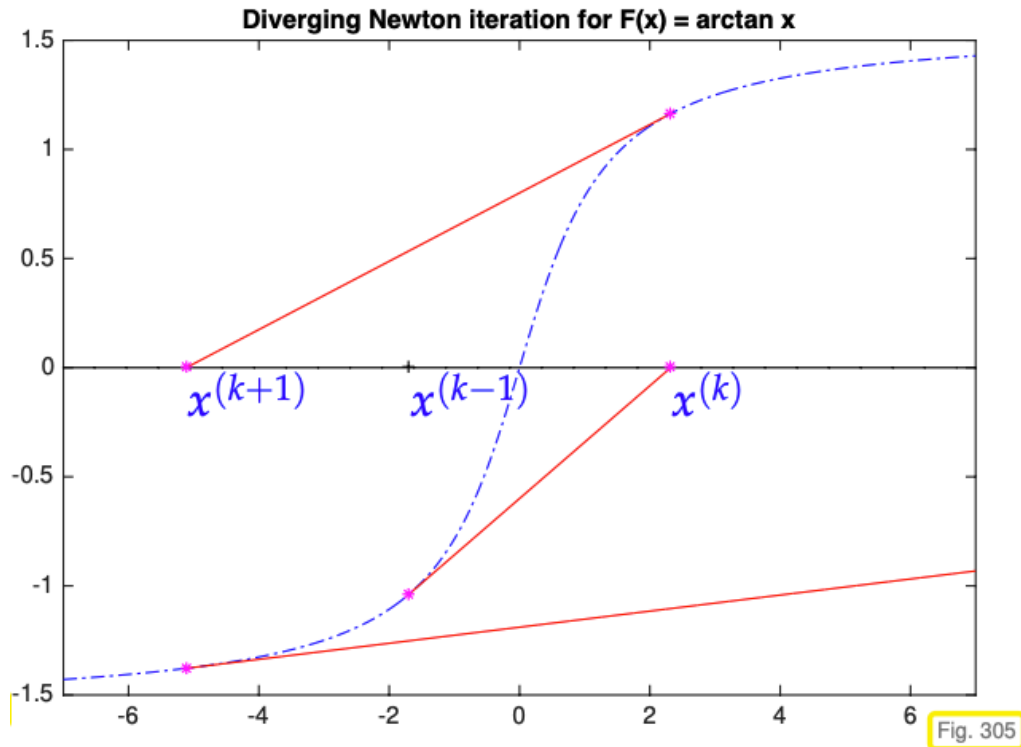
► Economical correction based termination criterion for Newton's method:

STOP, as soon as $\|\Delta \bar{x}^{(k)}\| \leq \tau_{\text{rel}} \|x^{(k)}\|$ or $\|\Delta \bar{x}^{(k)}\| \leq \tau_{\text{abs}}$,

with **simplified Newton correction** $\Delta \bar{x}^{(k)} := DF(x^{(k-1)})^{-1}F(x^{(k)})$.

8.5.4 Damped Newton Method

One big drawback of the Newton method is that it only converges locally! This can make it useless if the guess is not already close to the exact solution. In this section we explore a method to enlarge the region of convergence, at the expense of the quadratic convergence.



We observe a kind of “overshooting” of the Newton correction. To try and fix this, we introduce a dampening factor $\lambda^{(k)} \in]0, 1]$.

$$x^{(k+1)} = x^{(k)} - \lambda^{(k)} \cdot DF(x^{(k)})^{-1} \cdot F(x^{(k)})$$

To find this dampening factor, we use the following formula:

Affine invariant damping strategy

Choice of damping factor: affine invariant **natural monotonicity test** (NMT) [Deu11, Ch. 3]:

$$\text{choose “maximal” } 0 < \lambda^{(k)} \leq 1: \quad \left\| \Delta \bar{x}(\lambda^{(k)}) \right\|_2 \leq \left(1 - \frac{\lambda^{(k)}}{2} \right) \left\| \Delta x^{(k)} \right\|_2 \quad (8.5.4.4)$$

where $\Delta x^{(k)} := DF(x^{(k)})^{-1} F(x^{(k)}) \rightarrow$ current Newton correction ,
 $\Delta \bar{x}(\lambda^{(k)}) := DF(x^{(k)})^{-1} F(x^{(k)} - \lambda^{(k)} \Delta x^{(k)}) \rightarrow$ tentative simplified Newton correction .

8.6 Quasi-Newton Method

Computing the Jaccobian is expensive, therefore we look at a method to replace the derivative by some approximation of it. In the 1-D case, we can simply choose some similar method, like the secant method. In the general case we want to do something similar.

Idea: Rewrite (8.6.0.1) as a **secant condition** for an approximation $\mathbf{J}_k \approx D F(\mathbf{x}^{(k)})$, $\mathbf{x}^{(k)} \triangleq$ the current iterate:

$$\mathbf{J}_k(\mathbf{x}^{(k)} - \mathbf{x}^{(k-1)}) = F(\mathbf{x}^{(k)}) - F(\mathbf{x}^{(k-1)}) .$$

► Iteration: $\mathbf{x}^{(k+1)} := \mathbf{x}^{(k)} - \mathbf{J}_k^{-1} F(\mathbf{x}^{(k)}) .$

Instead of calculating \mathbf{J}_k for every step, we want to reuse \mathbf{J}_{k-1} . In the end we get the Broyden quasi-Newton method.

Final form of **Broyden's quasi-Newton method** for solving $F(\mathbf{x}) = 0$:

$$\begin{aligned} \mathbf{x}^{(k+1)} &:= \mathbf{x}^{(k)} + \Delta \mathbf{x}^{(k)} , \quad \Delta \mathbf{x}^{(k)} := -\mathbf{J}_k^{-1} F(\mathbf{x}^{(k)}) , \\ \mathbf{J}_{k+1} &:= \mathbf{J}_k + \frac{F(\mathbf{x}^{(k+1)}) (\Delta \mathbf{x}^{(k)})^\top}{\|\Delta \mathbf{x}^{(k)}\|_2^2} , \end{aligned} \quad k \in \mathbb{N}_0 .$$

To initialize \mathbf{J}_0 we can for example use the exact Jacobi matrix $DF(\mathbf{x}^{(0)})$.

To improve the range of local convergence, we can use the same ideas as we used for the damped Newton iteration.

Instead of calculating the inverse \mathbf{J}_k^{-1} in every step, we can use a faster but less stable approach. This is further explained in the lecture document, but since it is mostly complicated formulas, I left it out of this summary.

8.7 Non-Linear Least Squares

We want to increase our scope to include overdetermined non-linear systems of equations. For this we reuse many concepts from linear least squares.

Given $F : D \subset \mathbb{R}^n \mapsto \mathbb{R}^m$, $n, m \in \mathbb{N}$, $n < m$, we call x^* a **non-linear least squares solution** of $F(x) = 0$, if

$$x^* = \operatorname{argmin}_{x \in D} \|F(x)\|_2^2$$

We often write

$$\mathbf{x}^* \in D : \quad \mathbf{x}^* = \operatorname{argmin}_{\mathbf{x} \in D} \Phi(\mathbf{x}) , \quad \Phi(\mathbf{x}) := \frac{1}{2} \|F(\mathbf{x})\|_2^2 .$$

It has to be noted that the factor $\frac{1}{2}$ is simply a convention.

8.7.1 (Damped) Newton Method

We assume that F is twice continuously differentiable. Then the non-linear least squares solution x^* has to be a zero of the derivative of $x \mapsto \Phi(x)$.

$$\Phi(x^*) = \min \Rightarrow \text{grad } \Phi(x) = 0, \quad \text{where } \text{grad } \Phi(x) := \left[\frac{\partial \Phi}{\partial x_1}(x), \dots, \frac{\partial \Phi}{\partial x_n}(x) \right]^\top \in \mathbb{R}^n.$$

We use the Newton iteration to solve the $n \times n$ system of equations $\text{grad } \Phi(x) = 0$. We get the following iteration:

$$x^{(k+1)} = x^{(k)} - H \Phi(x^{(k)})^{-1} \text{grad } \Phi(x^{(k)}), \quad (8.7.1.2)$$

where $H \Phi(x) \in \mathbb{R}^{n,n}$ is the **Hessian matrix** of Φ , see Def. 8.5.1.18:

$$H \Phi(x) = \left[\frac{\partial^2 \Phi}{\partial x_i \partial x_j} \right]_{i,j=1}^n \in \mathbb{R}^{n,n}.$$

The computation of the gradient and the Hessian matrix of the gradient can be seen in the video or lecture document.


8.7.2 Gauss-Newton Method

The Gauss-Newton method is an alternative, that is not dependent on the second derivative. It is based on the idea of linearization.

$$\underset{x \in \mathbb{R}^n}{\text{argmin}} \|F(x)\|_2 \quad \text{is approximated by} \quad \underset{x \in \mathbb{R}^n}{\text{argmin}} \|F(x_0) + D F(x_0)(x - x_0)\|_2,$$

We end up with the following iterative method:

$$\begin{aligned} &\text{Initial guess } x^{(0)} \in D \\ &x^{(k+1)} := x^{(k)} + s, \quad s := \underset{s' \in \mathbb{R}^n}{\text{argmin}} \|F(x^{(k)}) + D F(x^{(k)})s'\|_2. \end{aligned}$$

a linear least squares problem! 

This can be solved with the techniques for linear least squares problems from chapter 3.

10. Additional Content

Ending this summary with one of the greatest copy pastas.

The Hiptmair knows where he is at all times. He knows this, because he knows where he isn't.

By iterating where he isn't from where he is, or where he is from where he isn't -- whichever is the supremum -- he obtains a difference or deviation. The guidance

subsystem uses discrete deviations to generate Piecewise Polynomial corrective commands to drive the Hiptmair from a node set where he is to a mesh interval where he isn't and arriving at a position where he wasn't, currently he is. Consequently, the Chebychev Nodes that he has are now the nodes that he hadn't and it follows that the Cardinal Basis that he computed is now the basis that he didn't. In the event that the Fixed-Point Approximation that he derived is not consistent with the n -th root of unity, the system has acquired a variation. The variation being the Least Squares between where the Hiptmair is and where his Orthogonal Complement wasn't. If the Eigenvalue of the variation is considered to be a significant roundoff error, it too may be corrected by the Basic Linear Algebra Subprograms(BLAS); however, the Hiptmair must also know where he was. The Hiptmair guidance computer scenario works as follows; because a periodic quadrature formula has modified some of the information the Hiptmair has obtained, he is not sure just where he is; however, he is sure where he isn't (within reason) and he knows where he was, in case the QR-Decomposition of the Cosine Transform is regular. He now subtracts where he should be from where he wasn't, or vice versa, and by differentiating the inverse of the Newton Correction from the algebraic sum of where he shouldn't be and where he was, he is able to obtain the exponential convergence and it's hermetic variation, which is called Matrix Multiplication.