# Numerical Methods for Computer Science

Robin Bacher, Janis Hutz
https://github.com/janishutz/eth-summaries

30. September 2025

TITLE PAGE COMING SOON

"*Denken vor Rechnen*"
- Vasile Grudinaru, 2025

HS2025, ETHZ
Summary of the Script and Lectures

# Inhaltsverzeichnis

# 0 Introduction

This summary is intended to give you a broad overview of the topics relevant for the exam and is not intended to serve as a full on replacement for the script. We have decided to write it in German, as is the new script and for some of the topics that are poorly explained in the script, we have added further explanations.

The numbering should match the script's numbering exactly, making it easier for you to look up the relevant definitions, theorems, etc in context in the script.

# 1 Einführung

## 1.1 Rundungsfehler

**Absoluter & Relativer Fehler** **Definition 1.1.1**

- ***Absoluter Fehler***: $||\widetilde{x} - x||$
- ***Relativer Fehler***: $\frac{||\widetilde{x} - x||}{||x||}$ für $||x|| \neq 0$

wobei $\widetilde{x}$ eine Approximation an $x \in \mathbb{R}$ ist

Rundungsfehler entstehen durch die (verhältnismässig) geringe Präzision die man mit der Darstellung von Zahlen auf Computern erreichen kann. Zusätzlich kommt hinzu, dass durch Unterläufe (in diesem Kurs ist dies eine Zahl die zwischen $0$ und der kleinsten darstellbaren, positiven Zahl liegt) Präzision verloren gehen kann.

Überläufe hingegen sind konventionell definiert, also eine Zahl, die zu gross ist und nicht mehr dargestellt werden kann.

**Auslöschung** **Bemerkung 1.1.9**

Bei der Subtraktion von zwei ähnlich grossen Zahlen kann es zu einer Addition der Fehler der beiden Zahlen kommen, was dann den relativen Fehler um einen sehr grossen Faktor vergrössert. Die Subtraktion selbst hat einen vernachlässigbaren Fehler

**Beispiel 1.1.18:** *(Ableitung mit imaginärem Schritt)* Als Referenz in Graphen wird hier oftmals die Implementation des Differenzialquotienten verwendet.

Der Trick hier ist, dass wir mit Komplexen Zahlen in der Taylor-Approximation einer glatten Funktion in $x_0$ einen rein imaginären Schritt durchführen können:

$$f(x_0 + ih) = f(x_0) + f'(x_0)ih - \frac{1}{2}f''(x_0)h^2 - iC \cdot h^3 \text{ für } h \in \mathbb{R} \text{ und } h \to 0$$

Da $f(x_0)$ und $f''(x_0)h^2$ reell sind, verschwinden die Terme, wenn wir nur den Imaginärteil des Ausdruckes weiterverwenden. Nach weiteren Vereinfachungen und Umwandlungen erhalten wir

$$f'(x_0) \approx \frac{\text{Im}(f(x_0 + ih))}{h}$$

Falls jedoch hier die Auswertung von $\text{Im}(f(x_0 + ih))$ nicht exakt ist, so kann der Fehler beträchtlich sein.

**Beispiel 1.1.20:** *(Konvergenzbeschleunigung nach Richardson)*

$$\begin{aligned} yf'(x) &= yd\left(\frac{h}{2}\right) + \frac{1}{6}f'''(x)h^2 + \frac{1}{480}f^{(s)}h^4 + \ldots - f'(x) \\ &= -d(h) - \frac{1}{6}f'''(x)h^2 + \frac{1}{120}f^{(s)}(x)h^n \Leftrightarrow 3f'(x) \\ &= 4d\left(\frac{h}{2}\right)d(h) + \mathcal{O}\left(h^4\right) \Leftrightarrow \end{aligned}$$

**Schema**

$$d(h) = \frac{f(x+h) - f(x-h)}{2h}$$

wobei im Schema dann

$$R_{l,0} = d\left(\frac{h}{2^l}\right)$$

und

$$R_{l,k} = \frac{4^k \cdot R_{l,k-1} - R_{l-1,k-1}}{4^k - 1}$$

und $f'(x) = R_{l,k} + C \cdot \left(\frac{h}{2^l}\right)^{2k+2}$

## 1.2 Rechenaufwand

In NumCS wird die Anzahl elementarer Operationen wie Addition, Multiplikation, etc benutzt, um den Rechenaufwand zu beschreiben. Wie in Algorithmen und * ist auch hier wieder $\mathcal{O}(\dots)$ der Worst Case. Teilweise werden auch andere Funktionen wie $\sin, \cos, \sqrt{\dots}, \dots$ dazu gezählt.

Die Basic Linear Algebra Subprograms (= BLAS), also grundlegende Operationen der Linearen Algebra, wurden bereits stark optimiert und sollten wann immer möglich verwendet werden und man sollte auf keinen Fall diese selbst implementieren.

Dieser Kurs verwendet `numpy`, `scipy`, `sympy` (collection of implementations for symbolic computations) und `matplotlib`. Dieses Ecosystem ist eine der Stärken von Python und ist interessanterweise zu einem Grossteil nicht in Python geschrieben, da dies sehr langsam wäre.

## 1.3 Rechnen mit Matrizen

Wie in Lineare Algebra besprochen, ist das Resultat der Multiplikation einer Matrix $A \in \mathbb{C}^{m\times n}$ und einer Matrix $B \in \mathbb{C}^{n\times p}$ ist eine Matrix $AB = \in \mathbb{C}^{m\times p}$

**In NumPy** haben wir folgende Funktionen:

- `b @ a` (oder `np.dot(b, a)` oder `np.einsum('i,i', b, a)` für das Skalarprodukt
- `A @ B` (oder `np.einsum('ik,kj->ij', )`) für das Matrixprodukt
- `A @ x` (oder `np.einsum('ij,j->i', A, x)`) für Matrix $\times$ Vektor
- `A.T` für die Transponierung
- `A.conj()` für die komplexe Konjugation (kombiniert mit `.T` = Hermitian Transpose)
- `np.kron(A, B)` für das Kroneker Produkt
- `b = np.array([4.j, 5.j])` um einen Array mit komplexen Zahlen zu erstellen (`j` ist die imaginäre Einheit, aber es muss eine Zahl direkt daran geschrieben werden)

**Bemerkung 1.3.4:** *(Rang der Matrixmultiplikation)* $\text{Rang}(AX) = \min(\text{Rang}(A), \text{Rang}(X))$

**Bemerkung 1.3.7:** *(Multiplikation mit Diagonalmatrix $D$)* $D \times A$ skaliert die Zeilen von $A$ während $A \times D$ die Spalten skaliert

**Beispiel 1.3.9:** `D @ A` braucht $\mathcal{O}(n^3)$ Operationen, wenn wir jedoch `D.diagonal()[:, np.newaxis] * A` verwenden, so haben wir nur noch $\mathcal{O}(n^2)$ Operationen, da wir die vorige Bemerkung Nutzen und also nur noch eine Skalierung vornehmen. So können wir also eine ganze Menge an Speicherzugriffen sparen, was das Ganze bedeutend effizienter macht

**Bemerkung 1.3.14:** Wir können bestimmte Zeilen oder Spalten einer Matrix skalieren, in dem wir einer Identitätsmatrix im unteren Dreieck ein Element hinzufügen. Wenn wir nun diese Matrix $E$ (wie die in der $LU$-Zerlegung) linksseitig mit der Matrix $A$ multiplizieren (bspw. $E^{(2,1)}A$), dann wird die zugehörige Zeile skaliert. Falls wir aber $AE^{(2,1)}$ berechnen, so skalieren wir die Spalte

**Bemerkung 1.3.15:** *(Blockweise Berechnung)* Man kann das Matrixprodukt auch Blockweise berechnen. Dazu benutzen wir eine Matrix, deren Elemente andere Matrizen sind, um grössere Matrizen zu generieren. Die Matrixmultiplikation funktioniert dann genau gleich, nur dass wir für die Elemente Matrizen und nicht Skalare haben.

---

Untenstehend eine Tabelle zum Vergleich der Operationen auf Matrizen

| Name | Operation | Mult | Add | Komplexität |
|---|---|---|---|---|
| Skalarprodukt | $x^H y$ | $n$ | $n-1$ | $\mathcal{O}(n)$ |
| Tensorprodukt | $xy^H$ | $nm$ | $0$ | $\mathcal{O}(mn)$ |
| Matrix $\times$ Vektor | $Ax$ | $mn$ | $(n-1)m$ | $\mathcal{O}(mn)$ |
| Matrixprodukt | $AB$ | $mnp$ | $(n-1)mp$ | $\mathcal{O}(mnp)$ |

**Bemerkung 1.3.16:** Das Matrixprodukt kann mit Strassen's Algorithmus mithilfe der Block-Partitionierung in $\mathcal{O}(n^{\log_2(7)}) \approx \mathcal{O}(n^{2.81})$ berechnet werden.

**Bemerkung 1.3.17:** *(Rang 1 Matrizen)* Können als Tensorprodukt von zwei Vektoren geschrieben werden. Dies ist beispielsweise hierzu nützlich:

Sei $A = ab^\top$. Dann gilt $y = Ax \Leftrightarrow y = a(b^\top x)$, was dasselbe Resultat ergibt, aber nur $\mathcal{O}(m+n)$ Operationen und nicht $\mathcal{O}(mn)$ benötigt wie links.

**Beispiel 1.3.18:** Für zwei Matrizen $A, B \in \mathbb{R}^{n\times p}$ mit geringem Rang $p \ll n$, dann kann mithilfe eines Tricks die Rechenzeit von `np.triu(A @ B.T) @ x` von $\mathcal{O}(pn^2)$ auf $\mathcal{O}(pn)$ reduziert werden. Die hier beschriebene Operation berechnet $\text{Upper}(AB^\top)x$ wobei $\text{Upper}(X)$ das obere Dreieck der Matrix $X$ zurück gibt. Wir nennen diese Matrix hier $R$. Wir können in NumPy den folgenden Ansatz verwenden, um die Laufzeit zu verringern: Da die Matrix $R$ eine obere Dreiecksmatrix ist, ist das Ergebnis die Teilsummen von unserem Umgekehrten Vektor $x$, also können wir mit

`np.cumsum(x[::-1], axis=0)[::-1]` die Kummulative Summe berechnen. Das `[::-1]` dient hier lediglich dazu, den Vektor $x$ umzudrehen, sodass das richtige Resultat entsteht. Die vollständige Implementation sieht so aus:

```python
import numpy as np

def low_rank_matrix_vector_product(A: np.ndarray, B: np.ndarray, x: np.ndarray):
    n, _ = A.shape
    y = np.zeros(n)

    # Compute B * x with broadcasting (x needs to be reshaped to 2D)
    v = B * x[:, None]

    # s is defined as the reverse cummulative sum of our vector
    # (and we need it reversed again for the final calculation to be correct)
    s = np.cumsum(v[::-1], axis=0)[::-1]

    y = np.sum(A * s)
```

**Definition 1.3.21:** *(Kronecker-Produkt)* Das Kronecker-Produkt ist eine $(ml) \times (nk)$-Matrix, für $A \in \mathbb{R}^{m \times n}$ und $B \in \mathbb{R}^{l \times k}$, die wir in NumPy einfach mit `np.kron(A, B)` berechnen können (ist jedoch nicht immer ideal):

$$A \otimes B := \begin{bmatrix} (A)_{1,1}B & (A)_{1,2}B & \dots & \dots & (A)_{1,n}B \\ (A)_{2,1}B & (A)_{2,2}B & \dots & \dots & (A)_{2,n}B \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ (A)_{m,1}B & (A)_{m,2}B & \dots & \dots & (A)_{m,n}B \end{bmatrix}$$

**Beispiel 1.3.22:** *(Multiplikation des Kronecker-Produkts mit Vektor)* Wenn man $A \otimes B \cdot x$ berechnet, so ist die Laufzeit $\mathcal{O}(m \times n \times l \times k)$, aber wenn wir den Vektor $x$ in $n$ gleich grosse Blöcke aufteilen (was man je nach gewünschter nachfolgender Operation in NumPy in $\mathcal{O}(1)$ machen kann mit `x.reshape(n, x.shape[0] / n)`), dann ist es möglich das Ganze in $\mathcal{O}(m \cdot l \cdot k)$ zu berechnen.

Die vollständige Implementation ist auch hier nicht schwer und sieht folgendermassen aus:

```python
import numpy as np

def fast_kron_vector_product(A: np.ndarray, B: np.ndarray, x: np.ndarray):
    # First multiply Bx_i, (and define x_i as a reshaped numpy array to save cost (as that
      ↪ will create a valid array))
    # This will actually crash if x.shape[0] is not divisible by A.shape[0]
    bx = B * x.reshape(A.shape[0], round(x.shape[0] / A.shape[0]))
    # Then multiply a with the resulting vector
    y = A * bx
```

Um die oben erwähnte Laufzeit zu erreichen muss erst ein neuer Vektor berechnet werden, oben im Code `bx` genannt, der eine Multiplikation von `Bx_i` als Einträge hat.

# 2 Polynomiale Interpolation

Bei der Interpolation versuchen wir eine Funktion $\widetilde{f}$ durch eine Menge an Datenpunkten einer Funktion $f$ zu finden. Die $x_i$ heissen Stützstellen/Knoten, für welche $\widetilde{f}(x_i) = y_i$ gelten soll. (Interpolationsbedingung)

$$\begin{bmatrix} x_0 & x_1 & \dots & x_n \\ y_0 & y_1 & \dots & y_n \end{bmatrix}, \quad x_i, y_i \in \mathbb{R}$$

Normalerweise stellt $f$ eine echte Messung dar, d.h. macht es Sinn anzunehmen dass $f$ glatt ist.

Die informelle Problemstellung oben lässt sich durch Vektorräume formalisieren:

$f \in \mathcal{V}$, wobei $\mathcal{V}$ ein Vektorraum mit $\dim(\mathcal{V}) = \infty$ ist.
Wir suchen d.h. $\widetilde{f}$ in einem Unterraum $\mathcal{V}_n$ mit endlicher $\dim(\mathcal{V}_n) = n$. Sei $B_n = \{b_1, \dots, b_n\}$ eine Basis für $\mathcal{V}_n$. Dann lässt sich der Bezug zwischen $f$ und $\widetilde{f} = f_n(x)$ so ausdrücken:

$$f(x) \approx f_n(x) = \sum_{j=1}^{n} \alpha_j b_j(x)$$

**Bemerkung 2.0.2:** Unterräume $\mathcal{V}_n$ existieren nicht nur für Polynome, wir beschränken uns aber auf $b_j(x) = x^{i-1}$. Andere Möglichkeiten: $b_j = \cos((j-1)\cos^- 1(x))$ *(Chebyshev)* oder $b_j = e^{i2\pi jx}$ *(Trigonometrisch)*

**Satz 2.0.5:** *(Peano)* $f$ stetig $\implies \exists p(x)$ welches $f$ in $||\cdot||_\infty$ beliebig gut approximiert.

**Definition 2.0.7:** *(Raum der Polynome)* $\mathcal{P}_k := \{x \mapsto \sum_{j=0}^{k} \alpha_j x^j\}$ **Definition 2.0.8:** *(Monom)* $f : x \mapsto x^k$

**Satz 2.0.9:** *(Eigenschaft von $\mathcal{P}_k$)* $\mathcal{P}_k$ ist ein Vektorraum mit $\dim(\mathcal{P}_k) = k + 1$.

## 2.1 Monombasis

**Satz 2.1.10:** *(Eindeutigkeit)* $p(x) \in (P)_k$ ist durch $k+1$ Punkte $y_i = p(x_i)$ eindeutig bestimmt.

Dieser Satz kann direkt angewendet werden zur Interpolation, in dem man $p(x)$ als Gleichungssystem schreibt.

$$p_n(x) = \alpha_n x^n + \dots + \alpha_0 x^0 \quad \iff \quad \underbrace{\begin{bmatrix} 1 & x_0 & \cdots & x_0^n \\ 1 & x_1 & \cdots & x_1^n \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & \cdots & x_n^n \end{bmatrix}}_{\text{Vandermonde Matrix}} \begin{bmatrix} \alpha_0 \\ \alpha_1 \\ \vdots \\ \alpha_n \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{bmatrix}$$

Um $\alpha_i$ zu finden ist die Vandermonde Matrix unbrauchbar, da die Matrix schlecht konditioniert ist.

Zur Auswertung von $p(x)$ kann man direkt die Matrix-darstellung nutzen, oder effizienter:

**Definition 2.1.11:** *(Horner Schema)* $p(x) = (x \dots x(x(\alpha_n x + \alpha_{n-1}) + \dots + \alpha_1) + \alpha_0)$

**In NumPy** `polyfit` liefert die direkte Auswertung, `polyval` wertet Polynome via Horner-Schema aus. (Gemäss Script, in der Praxis sind diese Funktionen `deprecated`)

## 2.2 Newton Basis

Session: Herleitung unwichtig, konzentrieren auf Funktion/Eigenschaften von Newton/Lagrange.

## 2.3 Lagrange- und Baryzentrische Interpolationsformeln

Session: Gemäss TA sehr gut beschrieben im alten Script

**Lagrange Polynome** — **Definition 2.3.1**

Für Knoten (auch gennannt Stützstellen) $x_0, x_1, \dots, x_n \in \mathbb{R}$ definieren wir die Lagrange-Polynome:

$$l_i(x) = \prod_{j=0 \neq i}^{n} \frac{x - x_j}{x_i - x_j}$$

Falls $j = i$ im Produkt, so überspringt $j$ diese Zahl.

**Beispiel 2.3.2:** Seien $x_0, x_1, x_2$ die Stützstellen für die Lagrange-Polynome (mit $n = 2$):

$$l_0(x) = \frac{x - x_1}{x_0 - x_1} \cdot \frac{x - x_2}{x_0 - x_2} \qquad l_1(x) = \frac{x - x_0}{x_1 - x_0} \cdot \frac{x - x_2}{x_1 - x_2} \qquad l_2(x) = \frac{x - x_0}{x_2 - x_0} \cdot \frac{x - x_1}{x_2 - x_1}$$

**Lagrange-Interpolationsformel** — **Satz 2.3.3**

Die Lagrange-Polynome $l_i$ zu den Stützstellen $(x_0, y_0), \dots, (x_n, y_n)$ bilden eine Basis der Polynome $\mathcal{P}_n$ und es gilt:

$$p(x) = \sum_{i=0}^{n} y_i l_i(x) \text{ mit } l_i(x) = \prod_{j \neq i} \frac{x - x_j}{x_i - x_j}$$

**Bemerkung 2.3.4:** *(Eigenschaften der Lagrange-Polynome)*

1. $l_i(x_j) = 0 \ \ \forall j \neq i$
2. $l_i(x_i) = 1 \ \ \forall i$
3. $\deg(l_i) = n \ \ \forall i$
4. $\sum_{k=0}^{n} l_k(x) = 1$ und $\sum_{k=0}^{n} l_k^{(m)}(x) = 0$ für $m > 0$

Da eine Implementation, welche direkt auf den Lagrange-Polynomen basiert, eine Laufzeit von $\mathcal{O}(n^3)$ hätte, suchte man nach einer besseren Methode. Mit der ***baryzentrischen Interpolationsformel*** wird zuerst ein Pre-Computing auf Teilen der Lagrange-Polynome durchgeführt, was dann dazu führt, dass die Laufzeit auf $\mathcal{O}(n^2)$ sinkt ($\mathcal{O}(n)$ für die Auswertung der Formel und $\mathcal{O}(n^2)$ für die Berechnung der $\lambda_k$):

**Baryzentrische Interpolationsformel** — **Formel**

$$p(x) = \frac{\sum_{k=0}^{n} \frac{\lambda_k}{x - x_k} y_k}{\sum_{k=0}^{n} \frac{\lambda_k}{x - x_k}}$$

Falls wir die Stützstellen als $(n+1)$ Chebyshev-Abszissen $x_k = \cos\left(\frac{k\pi}{n}\right)$ wählen, so sind alle $\lambda_k$ gegeben durch $\lambda_k = (-1)^k \delta_k$ mit $\delta_0 = \delta_n = 0.5$ und $\delta_i = 1$.

Mit anderen $\lambda_k$ eröffnet die baryzentrische Formel einen Weg zur Verallgemeinerung der Interpolation mittels rationaler Funktionen und ist entsprechend kein Polynom mehr.

Eine weitere Anwendung der Formel ist als Ausganspunkt für die Spektralmethode für Differenzialgleichungen.

### 2.3.1 Fehler

Falls an den Stützstellen $x_i$ durch beispielsweise ungenaue Messungen unpräzise Werte $\widetilde{y_i}$ haben, so entsteht logischerweise auch ein unpräzises Polynom $\widetilde{p}(x)$. Verglichen in der Lagrange-Basis zum korrekten Interpolationspolynom $p(x)$ ergibt sich folgender Fehler:

$$|p(x) - \widetilde{p}(x)| = \left| \sum_{i=0}^{n} (y_i - \widetilde{y_i}) l_i(x) \right| \leq \max_{i=0,\dots,n} |y_i - \widetilde{y_i}| \cdot \sum_{i=0}^{n} |l_i(x)|$$

**Definition 2.3.5:** *(Lebesgue-Konstante)* Zu den Stützstellen $x_0, \dots, x_n$ im Intervall $[a, b]$ ist sie definiert durch

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

**Satz 2.3.7:** *(Auswirkung von Messfehlern)* Es gilt (wenn $\Lambda_n$ die beste Lebesgue-Konstante für die Ungleichung ist):

$$\max_{x \in [a,b]} |p(x) - \widetilde{p}(x)| \leq \Lambda_n \max_{i=0,\dots,n} |y_i - \widetilde{y_i}|$$

**Fehler** **Satz 2.3.8**

Sei $f : [a, b] \to \mathbb{R}$ und $p$ das Interpolationspolynom zu $f$. Seien $x_0, \dots, x_n$ die Stützstellen, dann gilt:

$$||f(x) - p(x)||_\infty = \max_{x \in [a,b]} |f(x) - p(x)| \leq (1 + \Lambda_n) \min_{q \in \mathcal{P}_n} \max_{x \in [a,b]} |f(x) - q(x)|$$

**Bemerkung 2.3.10:** Für gleichmässig auf $I$ verteilte Stützstellen gilt $\Lambda_n \approx \dfrac{2^{n+1}}{en \log(n)}$

**Wichtig:** ***Niemals gleichmässig verteilte Stützstellen verwenden für die Interpolation von Polynomen hohen Grades***

Präzisere Interpolationen lassen sich beispielsweise durch Unterteilen des Intervalls in kleinere Intervalle finden, indem man für jedes Intervall ein separates Polynom berechnet, oder indem eine ideale Verteilung der Stützstellen wählt (was wiederum nicht einfach zu erzielen ist, siehe nächstes Kapitel).

## 2.4 Chebychev Interpolation

Session: Chebyshev Pol. : Abzisse = Extrema, Knoten = Nullstellen

Lecture: Orthogonalität ist eine wichtige Eigenschaft: Siehe Lecture notes (handgeschr.) für Veranschaulichung.
$\rightarrow$ Orth. liefert die Koeff. ohne Rechenaufwand.

Lecture: Clenshaw-Alg. relativ zentral (Taschenrechner nutzen diesen intern)

# 3 Trigonometrische Interpolation

Lecture: Wir besitzen nicht das komplette Vorwissen in der Analysis für dieses Kapitel, d.h. wird totales Verständnis nicht

Lecture: Intuitiv wird Fourier-Trans. zur Kompression genutzt, z.b. jpg format.