

Numerical Methods for Computer Science

Robin Bacher, Janis Hutz

<https://github.com/janishutz/eth-summaries>

26. September 2025

TITLE PAGE COMING SOON

“Denken vor Rechnen”

- Vasile Grudinaru, 2025

HS2025, ETHZ

Summary of the Script and Lectures

Inhaltsverzeichnis

1	Einführung	3
1.1	Rundungsfehler	3
1.2	Rechenaufwand	5
1.3	Rechnen mit Matrizen	6
2	Interpolation	8

1 Einführung

1.1 Rundungsfehler

Absoluter & Relativer Fehler

Definition 1.1.1

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

wobei \tilde{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:

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

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*)

$$\begin{aligned} yf'(x) &= yd\left(\frac{h}{2}\right) + \frac{1}{6}f'''(x)h^2 + \frac{1}{480}f^{(5)}(x)h^4 + \dots - f'(x) \\ &= -d(h) - \frac{1}{6}f'''(x)h^2 + \frac{1}{120}f^{(5)}(x)h^4 \Leftrightarrow 3f'(x) \\ &= 4d\left(\frac{h}{2}\right) d(h) + \mathcal{O}(h^4) \Leftrightarrow \end{aligned}$$

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 , $\sqrt{\dots}$, ... 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', A, B)`) 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 $A = ab^T$. Dann gilt $y = Ax \Leftrightarrow y = a(b^T x)$, 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 $\text{Upper}(AB^T)x$ 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:

```
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:

```
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
    # 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 Interpolation