

Numerical Methods for Computer Science

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

7. Oktober 2025

TITLE PAGE COMING SOON

“Denken vor Rechnen”

- Vasile Grudinaru, 2025

HS2025, ETHZ
Summary of the Script and Lectures

Inhaltsverzeichnis

0	Introduction	3
1	Einführung	4
1.1	Rundungsfehler	4
1.2	Rechenaufwand	6
1.3	Rechnen mit Matrizen	7
2	Polynominterpolation	9
2.1	Interpolation und Polynome	9
2.1.1	Monombasis	9
2.2	Newton Basis	10
2.2.1	Koeffizienten	10
2.2.2	Auswertung	11
2.2.3	Fehler	11
2.3	Lagrange- und Baryzentrische Interpolationsformeln	12
2.3.1	Fehler	14
2.4	Chebyshev Interpolation	15
2.4.1	Fehler	16
2.5	DFT und Chebyshev-Interpolation	16
3	Trigonometrische Interpolation	17
3.1	Fourier-Reihen	17
3.2	Diskrete Fourier Transformation	18
3.3	Schnelle Fourier Transformation	19

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 (apart from the cases where two definitions were combined due to being closely related and short), 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

▪ **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

Definition 1.1.1

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

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

Bemerkung 1.1.9

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:

```

1 def low_rank_matrix_vector_product(A: np.ndarray, B: np.ndarray, x: np.ndarray):
2     n, _ = A.shape
3     y = np.zeros(n)
4
5     # Compute B * x with broadcasting (x needs to be reshaped to 2D)
6     v = B * x[:, None]
7
8     # s is defined as the reverse cummulative sum of our vector
9     # (and we need it reversed again for the final calculation to be correct)
10    s = np.cumsum(v[::-1], axis=0)[::-1]
11
12    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:

```

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

2.1 Interpolation und Polynome

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

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

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. \tilde{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 $\tilde{f} = f_n(x)$ so ausdrücken:

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

Bemerkung 2.1.2: Unterräume \mathcal{V}_n existieren nicht nur für Polynome, wir beschränken uns aber auf $b_j(x) = x^{j-1}$. Andere Möglichkeiten: $b_j = \cos((j-1)\cos^{-1}(x))$ (Chebyshev) oder $b_j = e^{i2\pi jx}$ (Trigonometrisch)

Satz 2.1.5: (Peano) f stetig $\implies \exists p(x)$ welches f in $\|\cdot\|_\infty$ beliebig gut approximiert.

Definition 2.1.7: (Raum der Polynome) $\mathcal{P}_k := \{x \mapsto \sum_{j=0}^k \alpha_j x^j\}$ **Definition 2.1.8:** (Monom) $f : x \mapsto x^k$

Satz 2.1.9: (Eigenschaft von \mathcal{P}_k) \mathcal{P}_k ist ein Vektorraum mit $\dim(\mathcal{P}_k) = k+1$.

2.1.1 Monombasis

Satz 2.1.10: (Eindeutigkeit) $p(x) \in (\mathcal{P})_k$ 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.

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

Um α_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) $p(x) = (x \dots x(\alpha_n x + \alpha_{n-1}) + \dots + \alpha_1) + \alpha_0$

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

Die Newton-Basis hat den Vorteil, dass sie leichter erweiterbar als die Monombasis ist.

Die Konstruktion verläuft iterativ, und vorherige Datenpunkte müssen nicht Neuberechnet werden.

$$\begin{aligned}
 p_0(x) &= y_0 && (\text{Anfang: triviales Polynom}) \\
 p_1(x) &= p_0(x) + (x - x_0) \frac{(y_1 - y_0)}{(x_1 - x_0)} && (\text{Addition des zweiten Datenpunktes}) \\
 p_2(x) &= p_1(x) + \frac{\frac{(y_2 - y_1)}{(x_2 - x_1)} - \frac{(y_1 - y_0)}{(x_1 - x_0)}}{x_2 - x_0} (x - x_0)(x - x_1) && (\text{Schema lässt sich beliebig weiterführen}) \\
 p_3(x) &= p_2(x) + \dots
 \end{aligned}$$

Satz 2.2.3: (Newton-Basis) $\{N_0, \dots, N_n\}$ ist eine Basis von \mathcal{P}_n

$$\begin{aligned}
 N_0(x) &:= 1 \quad N_1(x) := x - x_0 \quad N_2(x) := (x - x_0)(x - x_1) \quad \dots \\
 N_n(x) &:= \prod_{i=0}^{n-1} (x - x_i)
 \end{aligned}$$

2.2.1 Koeffizienten

Wegen Satz 2.2.3 lässt sich jedes $p_n \in \mathcal{P}_n$ als $p_n(x) = \sum_{i=0}^n \beta_i N_i(x)$ darstellen. Ein Gleichungssystem liefert alle β_i :

$$\begin{bmatrix} 1 & 0 & \dots & 0 \\ 1 & N_0 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & N_0 & \dots & N_n \end{bmatrix} \begin{bmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_n \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{bmatrix}$$

Die Matrixmultiplikation in $\mathcal{O}(n^3)$ ist aber nicht nötig: Es gibt ein effizienteres System.

Definition 2.2.5: (Dividierte Differenzen)

$$\begin{aligned}
 y[x_i] &:= y_i \\
 y[x_i, \dots, x_{i+k}] &\stackrel{\text{Rec.}}{:=} \frac{y[x_{i+1}, \dots, x_{i+k}] - y[x_i, \dots, x_{i+k-1}]}{x_{i+k} - x_i}
 \end{aligned}
 \quad
 \begin{array}{l|l}
 x_0 & y[x_0] \\
 x_1 & y[x_1] \\
 x_2 & y[x_2] \\
 x_3 & y[x_3]
 \end{array}
 \begin{array}{l}
 > y[x_0, x_1] \\
 > y[x_0, x_1, x_2] \\
 > y[x_1, x_2] \\
 > y[x_1, x_2, x_3] \\
 > y[x_2, x_3]
 \end{array}$$

Bemerkung 2.2.6: (Äquidistante Stellen)

Falls $x_j = x_0 + \underbrace{j \cdot h}_{:= \Delta^j}$ gilt vereinfacht sich einiges:

$$\begin{aligned}
 y[x_0, x_1] &= \frac{1}{h} \Delta y_0 \\
 y[x_0, x_1, x_2] &= \frac{1}{2!h} \Delta^2 y_0 \\
 y[x_0, \dots, x_n] &= \frac{1}{n!h^n} \Delta^n y_0
 \end{aligned}$$

Satz 2.2.8: (Newton) Falls $\beta_j = y[x_0, \dots, x_j]$ geht das resultierende Polynom durch alle (x_i, y_i) .
(D.h. die dividierten Differenzen sind korrekt.)

Matrixmultiplikation in $\mathcal{O}(n^3)$, Speicher $\mathcal{O}(n^2)$

```

1 # Slow matrix approach
2 def divdiff_slow(x,y):
3     n = y.size
4     T = np.zeros((n,n))
5     T[:,0] = y
6
7     for l in range(1,n):
8         for i in range(n-l):
9             T[i, l] = (T[i+1,l-1] - T[i, l-1]) / (x[i+1] - x[i])
10
11
12     return T[0,:]
```

Vektorisierter Ansatz in $\mathcal{O}(n^2)$, Speicher $\mathcal{O}(n)$

```

1 # Fast vectorized approach
2 def divdiff_fast(x,y):
3     n = y.shape[0]
4
5     for k in range(1, n):
6         y[k:] = (y[k:] - y[(k-1):n-1]) / (x[k:] - x[0:n-k])
7
8     return y
```

2.2.2 Auswertung

Auswertung eines Newton-Polynoms funktioniert in $\mathcal{O}(n)$ durch ein modifiziertes Horner-Schema:

$$\begin{aligned}
 p_0 &:= \beta_n \\
 p_1 &:= (x - x_{n-1})p_0 + \beta_{n-1} \\
 p_2 &:= (x - x_{n-2})p_1 + \beta_{n-2} \\
 &\vdots \\
 p_n &= p(x)
 \end{aligned}$$

```

1 def evalNewton(x_data, beta, x):
2     p = np.zeros(x.shape[0])
3     p += beta[beta.shape[0]-1]
4
5     for i in range(1, n+1):
6         p = (x - x_data[n-i])*p + beta[n-i]
7
8     return p
```

2.2.3 Fehler

Satz 2.2.11: f n -mal diff.-bar, $y_i = f(x_i) \implies \exists \xi \in (\min_i x_i, \max_i x_i)$ s.d. $y[x_0, x_1, \dots, x_n] = \frac{f^{(n)}(\xi)}{(n+1)!}$

Satz 2.2.12: (Fehler) $f: [a, b] \rightarrow \mathbb{R}$ ist $(n+1)$ -mal diff.-bar, p ist das Polynom zu f in $x_0, \dots, x_n \in [a, b]$.

$$\forall x \in [a, b] \exists \xi \in (a, b) : \underbrace{f(x) - p(x)}_{\text{Fehler}} = \prod_{i=0}^n (x - x_i) \cdot \frac{f^{(n+1)}(\xi)}{(n+1)!}$$

Man bemerke: Die Wahl der Stützpunkte hat direkten Einfluss auf den Fehler.

2.3 Lagrange- und Baryzentrische Interpolationsformeln

Lagrange Polynome

Definition 2.3.1

Für Knoten (auch genannt Stützstellen) $x_0, x_1, \dots, x_n \in \mathbb{R}$ definieren wir die Lagrange-Polynome für $n =$ Anzahl Stützstellen, also haben wir $n - 1$ Brüche, da wir eine Iteration überspringen, weil bei dieser $j = i$ ist:

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

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 $n = 2$):

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

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. $l_i(x_j) = 0 \quad \forall j \neq i$
2. $l_i(x_i) = 1 \quad \forall i$
3. $\deg(l_i) = n \quad \forall i$
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 λ_k). Man berechnet die baryzentrischen Gewichte λ_k folgendermassen:

$$\lambda_k = \prod_{j \neq k} \frac{1}{x_k - x_j}$$

oder das ganze mithilfe von Numpy:

```

1 def barycentric_weights(x: np.ndarray) -> np.ndarray:
2     n = len(x)
3     # Initialize to zeros
4     barweight = np.ones(n)
5     for k in range(n):
6         # Vectorized differences between x_k and all xs
7         differences = x[k] - x
8         # Remove the k-th element (and handle edge cases for k = 0 and k = n - 1)
9         if k < n - 1 and k > 0:
10             diff_processed = np.concatenate((differences[:k], differences[(k + 1) :]))
11             barweight[k] = 1 / np.prod(diff_processed)
12         elif k == 0:
13             barweight[k] = 1 / np.prod(differences[1:])
14         else:
15             barweight[k] = 1 / np.prod(differences[:k])
16     return barweight

```

Gleiche funktion, etwas kürzer:

```

1 def barycentric_weights(x: np.ndarray) -> np.ndarray:
2     n = len(x)
3     w = np.ones(n) # = barweight
4     # Compute the (non-inverted) product, avoiding case (x[i] - x[i]) = 0
5     for i in range(0, n, 1):
6         if (i-1 > 0): w[0:(i-1)] *= (x[0:(i-1)] - x[i])
7         if (i+1 < n): w[i+1:n]   *= (x[i+1:n]   - x[i])
8     # Invert all at once
9     return 1/w

```

Mit dem können wir dann ein Polynom mit der baryzentrischen Interpolationsformel interpolieren:

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 λ_k gegeben durch $\lambda_k = (-1)^k \delta_k$ mit $\delta_0 = \delta_n = 0.5$ und $\delta_i = 1$.

Mit anderen λ_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 Ausgangspunkt für die Spektralmethode für Differenzialgleichungen.

```

1 def interp_barycentric(
2     data_point_x: np.ndarray,
3     data_point_y: np.ndarray,
4     barweight: np.ndarray,
5     x: np.ndarray
6 ):
7     p_x = np.zeros_like(x)
8     n = data_point_x.shape[0]
9
10    for i in range(x.shape[0]):
11        # Separate sums to divide in the end
12        upper_sum = 0
13        lower_sum = 0
14        for k in range(n):
15            frac = barweight[k] / (x[i] - data_point_x[k])
16            upper_sum += frac * data_point_y[k]
17            lower_sum += frac
18        p_x[i] = upper_sum / lower_sum
19
20    return p_x

```

2.3.1 Fehler

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

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

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 Λ_n die beste Lebesgue-Konstante für die Ungleichung ist):

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

Fehler

Satz 2.3.8

Sei $f : [a, b] \rightarrow \mathbb{R}$ 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 $\Lambda_n \approx \frac{2^{n+1}}{en \log(n)}$

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 Chebyshev Interpolation

Chebyshev-Polynome

Definition 2.4.1

Erster Art

$$T_n(x) = \cos(n \arccos(x)), \quad x \in [-1, 1]$$

Zweiter Art

$$U_n(x) = \frac{\sin((n+1) \arccos(x))}{\sin(\arccos(x))}, \quad x \in [-1, 1]$$

$T_n(x)$ scheint erst nicht ein Polynom zu sein, aber wir haben einen \arccos in einem \cos . Zudem:

Satz 2.4.3: (*Eigenschaften*) Das n -te Chebyshev-Polynom ist ein Polynom von Grad n und für $x \in [-1, 1]$ gilt:

1. $T_0(x) = 1, T_1(x) = x,$
 $T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x)$
2. $|T_n(x)| \leq 1$
3. $T_n(\cos(\frac{k\pi}{n})) = (-1)^k$ für $k = 0, \dots, n$
4. $T_n(\cos(\frac{(2k+1)\pi}{2n})) = 0$ für $k = 0, \dots, n-1$

Definition 2.4.4: (*Chebyshev-Knoten*) Die $(n+1)$ Chebyshev-Knoten x_0, \dots, x_n im Intervall $[-1, 1]$ sind die Nullstellen von $T_{n+1}(x)$

Bemerkung 2.4.5: (*Chebyshev-Knoten für beliebiges Intervall*) Für $I = [a, b]$ sind die Chebyshev-Knoten:

$$x_k = a + \frac{1}{2}(b-a) \left(\cos\left(\frac{2k+1}{2(n+1)}\pi\right) + 1 \right) \quad k = 0, \dots, n$$

Definition 2.4.6: (*Chebyshev-Abszissen*) Die $(n-1)$ Chebyshev-Abszissen x_0, \dots, x_{n-2} im Intervall $[-1, 1]$ sind die Extrema des Chebyshev-Polynoms $T_n(x)$ und zeitgleich die Nullstellen von $U_{n-1}(x)$. Je nach Kontext nimmt man noch die Grenzen des Intervalls (1 und -1) hinzu und hat dann $(n+1)$ Abszissen.

Die Baryzentrischen Gewichte sind dann viel einfacher zu berechnen: $\lambda_k = (-1)^k$ (siehe Bemerkung unterhalb der Baryzentrischen Interpolationsformel, Kapitel 2.3)

Bemerkung 2.4.7: (*Chebyshev-Abszissen für beliebiges Intervall*) Für $I = [a, b]$ sind die Chebyshev-Abszissen:

$$x_k = a + \frac{1}{2}(b-a) \left(\cos\left(\frac{k}{n}\pi\right) + 1 \right) \quad k = 0, \dots, n$$

Oder $k = 1, \dots, n-1$ bei ausgeschlossenen Endpunkten a und b

Bemerkung 2.4.8: Gegen die Ränder des Intervalls werden die Chebyshev-Knoten dichter.

Orthogonalität

Satz 2.4.9

Die Chebyshev-Polynome sind orthogonal bezüglich des Skalarprodukts

$$\langle f, g \rangle = \int_{-1}^1 f(x)g(x) \frac{1}{\sqrt{1-x^2}} dx$$

Sie (T_0, \dots, T_n) sind zudem orthogonal bezüglich des diskreten Skalarprodukts im Raum der Polynome von Grad $\leq n$

$$(f, g) = \sum_{l=0}^n f(x_l)g(x_l)$$

wobei (x_0, \dots, x_n) die Nullstellen von T_{n+1} sind.

2.4.1 Fehler

Was hat die neue Verteilung für einen Einfluss auf den Fehler?

Fehlerabschätzung

Satz 2.4.11

Unter allen (x_0, \dots, x_n) mit $x_i \in \mathbb{R}$ wird (wobei x_k die Nullstellen von T_{n+1} sind)

$$\max_{x \in [-1, 1]} |(x - x_0) \cdot \dots \cdot (x - x_n)| \quad \text{minimal für } x_k = \cos\left(\frac{2k+1}{2(n+1)}\pi\right)$$

Folglich sind also die Nullstellen der Chebyshev-Polynome T_n die bestmögliche Wahl für die Stützstellen. Da die Abszissen mit FFT einfacher zu berechnen sind, werden diese oft bevorzugt berechnet. Dies, da die Nullstellen von T_n in den Extrema von T_{2n} enthalten sind, während zudem zwischen zwei nebeneinanderliegenden Chebyshev-Abszissen jeweils eine Nullstelle von T_{2n} liegt

Satz 2.4.13: (*Lebesgue-Konstante*) Für die Chebyshev-Interpolation: $\Lambda_n \approx \frac{2}{\pi} \log(n)$ für $n \rightarrow \infty$

Interpolationspolynom

Satz 2.4.15

Das Interpolationspolynom p zu f mit Chebyshev-Knoten gleich der Nullstellen von T_{n+1} ist gegeben durch

$$p(x) = c_0 + c_1 T_1(x) + \dots + c_n T_n(x)$$

wobei für die c_k gilt:

$$c_k = \frac{2}{n+1} \sum_{l=0}^n f\left(\underbrace{\cos\left(\frac{2l+1}{n+1}\frac{\pi}{2}\right)}_{=x_l(\text{Knoten})}\right) \cos\left(k \frac{2l+1}{n+1} \frac{\pi}{2}\right) \quad \text{für } k = 1, \dots, n$$

$$c_k = \frac{1}{n+1} \sum_{l=0}^n f\left(\underbrace{\cos\left(\frac{2l+1}{n+1}\frac{\pi}{2}\right)}_{=x_l(\text{Knoten})}\right) \cos\left(k \frac{2l+1}{n+1} \frac{\pi}{2}\right) \quad \text{für } k = 0$$

Für $n \geq 15$ berechnet man c_k mit der Schnellen Fourier Transformation (FFT).

Bemerkung 2.4.16: (*Laufzeit*) Für die Interpolation ergibt sich folgender Aufwand:

Direkte Berechnung der c_k	$\mathcal{O}((n+1)^2)$ Operationen
Dividierte Differenzen	$\mathcal{O}\left(\frac{n(n+1)}{2}\right)$ Operationen (zum Vergleich)
c_k mittels FFT	$\mathcal{O}(n \log(n))$ Operationen

Satz 2.4.17: (*Clenshaw-Algorithmus*) Seien $d_{n+2} = d_{n+1} = 0$. Sei $d_k = c_k + (2x)d_{k+1} - d_{k+2}$ für $k = n, \dots, 0$. Dann gilt: $p(x) = \frac{1}{2}(d_0 - d_2)$ und man kann $p(x)$ mit Hilfe einer Rückwärtsrekursion berechnen

Der Clenshaw-Algorithmus ist sehr stabil, auch wenn er mit (oft) instabilen Rekursionen implementiert ist.

2.5 DFT und Chebyshev-Interpolation

Der folgende Abschnitt wurde ausgelassen, da dieser (noch) nicht während den Vorlesungen behandelt wurde. Er behandelt primär die Implementation der Chebyshev-Interpolation.

3 Trigonometrische Interpolation

3.1 Fourier-Reihen

Eine Anwendung der Schnellen Fourier-Transformation (FFT) ist die Komprimierung eines Bildes und sie wird im JPEG-Format verwendet.

Definition 3.1.1: (Trigonometrisches Polynom von Grad $\leq m$) Die Funktion:

$$p_m(t) := t \mapsto \sum_{j=-m}^m \gamma_j e^{2\pi i j t} \text{ wobei } \gamma_j \in \mathbb{C} \text{ und } t \in \mathbb{R}$$

Bemerkung 3.1.2: $p_m : \mathbb{R} \rightarrow \mathbb{C}$ ist periodisch mit Periode 1. Falls $\gamma_{-j} = \overline{\gamma_j}$ für alle j , dann ist p_m reellwertig und p_m kann folgendermassen dargestellt werden ($a_0 = 2\gamma_0$, $a_j = 2\Re(\gamma_j)$ und $b_j = -2\Im(\gamma_j)$):

$$p_m(t) = \frac{a_0}{2} + \sum_{j=1}^m (a_j \cos(2\pi j t) + b_j \sin(2\pi j t))$$

L^2 -Funktionen

Wir definieren die L^2 -Funktionen auf dem Intervall $(0, 1)$ als

$$L^2(0, 1) := \{f : (0, 1) \rightarrow \mathbb{C} \mid \|f\|_{L^2(0,1)} < \infty\}$$

während die L^2 -Norm auf $(0, 1)$ durch das Skalarprodukt

$$\langle g, f \rangle_{L^2(0,1)} := \int_0^1 \overline{g(x)} f(x) \, dx$$

über $\|f\|_{L^2(0,1)} = \sqrt{\langle f, f \rangle_{L^2(0,1)}}$ induziert wird

Definition 3.1.3

Bemerkung 3.1.4: $L^2(a, b)$ lässt sich analog definieren mit

$$\begin{aligned} \langle g, f \rangle_{L^2(a,b)} &:= \int_a^b \overline{g(x)} f(x) \, dx \\ &= (b-a) \int_0^1 \overline{g(a+(b-a)t)} f(a+(b-a)t) \, dt \end{aligned}$$

In Anwendungen findet sich oft das Intervall $[-\frac{T}{2}, \frac{T}{2}]$. Dann verwandeln sich die Integrale in die Form $\frac{1}{T} \int_{-\frac{T}{2}}^{\frac{T}{2}} (\dots) \, dt$ und $\exp(2\pi i j t)$ durch $\exp(i \frac{2\pi j}{T} t)$ ersetzt wird.

Bemerkung 3.1.6: Die Funktionen $\varphi_k(x) = \exp(2\pi i k x)$ sind orthogonal bezüglich des $L^2(0, 1)$ -Skalarprodukts, bilden also eine Basis für den Unterraum der trigonometrischen polynome.

Definition 3.1.7: Eine Funktion f ist der L^2 -Grenzwert von Funktionenfolgen $f_n \in L^2(0, 1)$, wenn für $n \rightarrow \infty$ gilt, dass $\|f - f_n\|_{L^2(0,1)} \rightarrow 0$

Fourier-Reihe

Satz 3.1.8

Jede Funktion $f \in L^2(0, 1)$ ist der Grenzwert ihrer Fourier-Reihe:

$$f(t) = \sum_{k=-\infty}^{\infty} \hat{f}(k) e^{2\pi i k t}$$

wobei die Fourier-Koeffizienten

$$\hat{f}(k) = \int_0^1 f(t) e^{-2\pi i k t} \, dt \quad k \in \mathbb{Z}$$

definiert sind. Es gilt die Parseval'sche Gleichung:

$$\sum_{k=-\infty}^{\infty} |\hat{f}(k)|^2 = \|f\|_{L^2(0,1)}^2$$

Bemerkung 3.1.9: Oder viel einfacher und kürzer: Die Funktionen $\varphi_k(x)$ bilden eine vollständige Orthonormalbasis in $L^2(0, 1)$.

Bemerkung 3.1.14: Die Parseval'sche Gleichung beschreibt einfach gesagt einen "schnellen" Abfall der $\hat{f}(k)$. Genauer gesagt, klingen die Koeffizienten schneller als $\frac{1}{\sqrt{k}}$ ab. Sie sagt zudem aus, dass die L^2 -Norm der Funktion aus einer Summe berechnet werden kann (nicht nur als Integral). Wenn wir die Fourier-Reihe nach t ableiten, erhalten wir

$$f'(t) = \sum_{k=-\infty}^{\infty} 2\pi i k \hat{f}(k) e^{2\pi i k t}$$

Fourier-Reihe

Satz 3.1.15

Seien f und f' integrierbar auf $(0, 1)$, dann gilt $\hat{f}'(k) = 2\pi i k \hat{f}(k)$ für $k \in \mathbb{Z}$.
Falls die Operationen erlaubt sind, dann gilt zudem:

$$\widehat{f^{(n)}} = (2\pi i k)^n \hat{f}(k) \text{ und } \|f^{(n)}\|_{L^2}^2 = (2\pi)^{2n} \sum_{k=-\infty}^{\infty} k^{2n} |\hat{f}(k)|^2$$

Satz 3.1.16: Wenn $\int_0^1 |f^{(n)}(t)| dt < \infty$, dann ist $\hat{f}(k) = \mathcal{O}(k^{-n})$

Falls die Funktion jedoch nicht glatt ist, dann entstehen *Überschwingungen* an den Sprungstellen, die näher und näher an die Sprünge herankommen, aber nicht kleiner werden, wenn wir mehr Terme der Fourier-Reihe aufsummieren. Das Phänomen wird das **Gibbs-Phänomen** genannt und wir haben L^2 -Konvergenz, aber keine punktweise Konvergenz an der Sprungstelle.

Bemerkung 3.1.17: Diese Überschwingungen entstehen durch die Definition der Fourier-Reihe und sind in der untenstehenden Abbildung 1 aus dem Skript sehr gut ersichtlich. Die dargestellte Funktion ist die Fourier-Reihe der charakteristischen Funktion des Intervalls $[a, b] \subseteq]0, 1[$, welche sich folgendermassen analytisch berechnen lässt:

$$b - a + \frac{1}{\pi} \sum_{k \neq 0} e^{-i k c} \frac{\sin(k d)}{k} e^{i 2 \pi k t}, \quad t \in [0, 1]$$

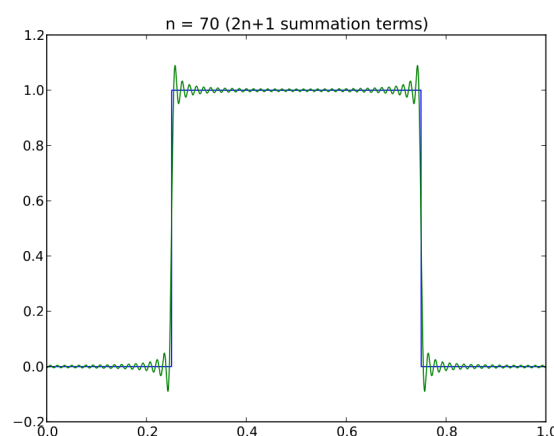
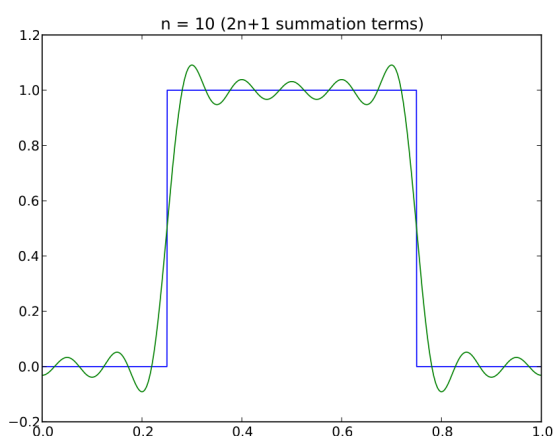


Abbildung 1: Überschwingungen der Fourier-Reihe der charakteristischen Funktion des Intervalls $[a, b] \subseteq]0, 1[$

Bemerkung 3.1.19: Meist ist es nicht möglich (oder nicht sinnvoll) die Fourier-Koeffizienten analytisch zu berechnen, weshalb man wieder zur Numerik und der Trapezformel greift, die folgendermassen definiert ist für $t_l = \frac{l}{N}$, wobei $l = 0, 1, \dots, N-1$ und N die Anzahl der Intervalle ist:

$$\hat{f}_N(k) := \frac{1}{N} \sum_{l=0}^{N-1} f(t_l) e^{-2\pi i k t_l} \approx \hat{f}(k)$$

3.2 Diskrete Fourier Transformation

3.3 Schnelle Fourier Transformation

Da es viele Anwendungen für die Fourier-Transformation gibt, ist ein Algorithmus mit guter Laufzeit sehr wichtig. Während eine naive version des DFT-Algorithmus eine Laufzeit von $\mathcal{O}(N^2)$ hat, so hat der Fast Fourier Transform Algorithmus nur eine Laufzeit von $\mathcal{O}(N \log(N))$, was bei $N = 1024$ bereits eine Laufzeitsverbesserung von $100\times$ mit sich bringt ($\mathcal{O}(10\,000)$ vs $\mathcal{O}(1\,000\,000)$ Operationen)! Die untenstehende Abbildung 2 findet sich, zusammen mit dem Code, mit der sie produziert wurde im Skript auf Seite 86-88

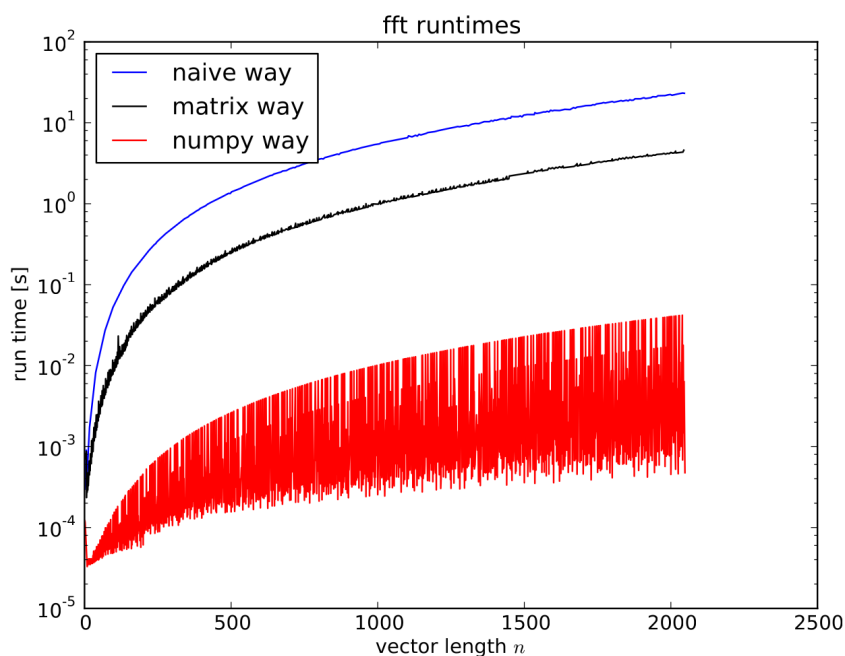


Abbildung 2: Vergleich der Laufzeit von verschiedenen Fourier-Transformations-Algorithmen

Der hier besprochene Cooley-Tukey-Algorithmus wurde ursprünglich von Gauss 1805 entdeckt, dann vergessen und schliesslich 1965 von Cooley und Tukey wiederentdeckt. Der Algorithmus verwendet einen "Divide and Conquer" Approach, also ist logischerweise die Idee, dass man die Berechnung einer DFT der Länge n auf die Berechnung vieler DFTs kleinerer Längen zurückführen kann.

Für den Algorithmus müssen folgende vier Optionen betrachtet werden:

- I Vektoren der Länge $N = 2m \implies$ Laufzeit ideal
- II Vektoren der Länge $N = 2^L \implies$ Laufzeit ideal
- III Vektoren der Länge $N = pq$ mit $p, q \in \mathbb{Z} \implies$ Etwas langsamer
- IV Vektoren der Länge N , mit N prim \implies ca. $\mathcal{O}(N^2)$, besonders für N gross

Wir formen die Fourier-Transformation um für den ersten Fall ($N = 2m$):

$$\begin{aligned}
 c_k &= \sum_{j=0}^{N-1} y_j e^{-\frac{2\pi i}{N} jk} \\
 &= \sum_{j=0}^{m-1} y_{2j} e^{-\frac{2\pi i}{N} 2jk} + \sum_{j=0}^{m-1} y_{2j+1} e^{-\frac{2\pi i}{N} (2j+1)k} \\
 &= \sum_{j=0}^{m-1} \left(y_{2j} e^{-\frac{2\pi i}{N/2} jk} \right) + e^{-\frac{2\pi i}{N} k} \left(\sum_{j=0}^{m-1} y_{2j+1} e^{-\frac{2\pi i}{N/2} jk} \right)
 \end{aligned}$$

Der zweite Fall ist einfach eine rekursive Weiterführung des ersten Falls, bei welchem dann das m kontinuierlich weiter dividiert wird bis zum Trivialfall mit einer 1×1 -Matrix.