Numerical Methods for Computer Science

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

16. Oktober 2025

TITLE PAGE COMING SOON

"*Denken vor Rechnen*" "*Wer in Python Type annotation benötigt, der soll kein Python verwenden*" (2025-10-09T10:43)

- Vasile Gradinaru, 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 (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.

Many of the figures in this summary were taken directly from the Script or Lecture notes created by Professor Vasile Gradinaru.

To add to the one quote regarding Python and type annotation: This is objectively wrong and a really hot take. Yes, this applies for small projects, but libraries *DO* need type annotation, as you can't possibly read the entire library's code to use it. The reason this quote was even included here is that his coding style is really awful (yes, there were semicolons in his Python-code sometimes) and he rambled about bad coding style for about 10 minutes in this lecture.

Meanwhile his code has variable names that neither future him, nor anybody else can make much sense of intuitively. You can get away without type annotation in Python, even in larger projects, but only if you give variables proper names!

Moral of the story: Use descriptive variable names and do NOT use t, tt, ttt, ...

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. **In `numpy`** können wir 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
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 $B \in \mathbb{R}^{l \times k}$.

In `numpy` können wir dieses einfach mit `np.kron(A, B)` berechnen (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
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 Polynominterpolation

2.1 Interpolation und Polynome

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 2.1.2: 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 2.1.5: *(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)* $p(x) \in (P)_k$ 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 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)* $p(x) = (x \dots x(x(\alpha_n x + \alpha_{n-1}) + \dots + \alpha_1) + \alpha_0)$

In `numpy` liefert `polyfit` 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.

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

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

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

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:

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

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

Definition 2.2.5: *(Dividierte Differenzen)*

$$
\begin{aligned}
y[x_i] &:= y_i \\
y[x_i, \dots, x_{i+k}] &\overset{\text{Rec.}}{:=} \frac{y[x_{i+1}, \dots, x_{i+k}] - y[x_i, \dots, x_{i+k-1}]}{x_{i+k} - x_i}
\end{aligned}
$$

$$
\begin{array}{c|c|c|c}
x_0 & y[x_0] & & \\
 & & > y[x_0, x_1] & \\
x_1 & y[x_1] & & > y[x_0, x_1, x_2] \\
 & & > y[x_1, x_2] & \\
x_2 & y[x_2] & & > y[x_1, x_2, x_3] \\
 & & > y[x_2, x_3] & \\
x_3 & y[x_3] & &
\end{array}
$$

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.)

Beispiel 2.2.9: *(Runge-Funktion)* Die Runge-Funktion kann am Rand des gewählten Intervalls starke Oszillationen in der Interpolation verursachen, wenn bspw. die Stützstellen nicht gut gewählt sind oder das Polynom einen zu hohen Grad hat. Sie ist definiert durch $f(x) = \frac{1}{1 + x^2}$

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

```python
# Slow matrix approach
def divdiff_slow(x,y):
    n = y.size
    T = np.zeros((n,n))
    T[:,0] = y

    for l in range(1,n):
        for i in range(n-l):
            T[i, l] = (T[i+1,l-1] - T[i, l-1])
            T[i, l] /= (x[i+l] - x[i])

    return T[0,:]
```

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

```python
# Fast vectorized approach
def divdiff_fast(x,y):
    n = y.shape[0]

    for k in range(1, n):
        y[k:] = (y[k:] - y[(k-1):n-1])
        y[k:] /= (x[k:] - x[0:n-k])

    return y
```

2.2.2 Auswertung

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

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

```python
def evalNewton(x_data, beta, x):
    p = np.zeros(x.shape[0])
    p += beta[beta.shape[0]-1]

    for i in range(1, n+1):
        p = (x - x_data[n-i])*p + beta[n-i]

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

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

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 gennannt 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 $n = 2$):

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

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 $\mathcal{O}\left(n^3\right)$ 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 $\mathcal{O}\left(n^2\right)$ sinkt ($\mathcal{O}(n)$ für die Auswertung der Formel und $\mathcal{O}\left(n^2\right)$ 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:

```python
def barycentric_weights(x: np.ndarray) -> np.ndarray:
    n = len(x)
    # Initialize to zeros
    barweight = np.ones(n)
    for k in range(n):
        # Vectorized differences between x_k and all xs
        differences = x[k] - x
        # Remove the k-th element (and handle edge cases for k = 0 and k = n - 1)
        if k < n - 1 and k > 0:
            diff_processed = np.concatenate((differences[:k], differences[(k + 1) :]))
            barweight[k] = 1 / np.prod(diff_processed)
        elif k == 0:
            barweight[k] = 1 / np.prod(differences[1:])
        else:
            barweight[k] = 1 / np.prod(differences[:k])
    return barweight
```

Gleiche funktion, etwas kürzer:

```python
def barycentric_weights(x: np.ndarray) -> np.ndarray:
    n = len(x)
    w = np.ones(n) # = barweight
    # Compute the (non-inverted) product, avoiding case (x[i] - x[i]) = 0
    for i in range(0, n, 1):
        if (i-1 > 0): w[0:(i-1)] *= (x[0:(i-1)] - x[i])
        if (i+1 < n): w[i+1:n]   *= (x[i+1:n]   - x[i])
    # Invert all at once
    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 Ausganspunkt für die Spektralmethode für Differenzialgleichungen.

```python
def interp_barycentric(
    data_point_x: np.ndarray,
    data_point_y: np.ndarray,
    barweight: np.ndarray,
    x: np.ndarray
):
    p_x = np.zeros_like(x)
    n = data_point_x.shape[0]

    for i in range(x.shape[0]):
        # Separate sums to divide in the end
        upper_sum = 0
        lower_sum = 0
        for k in range(n):
            frac = barweight[k] / (x[i] - data_point_x[k])
            upper_sum += frac * data_point_y[k]
            lower_sum += frac
        p_x[i] = upper_sum / lower_sum

    return p_x
```

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 Λ_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 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. $T_n\left(\cos\left(\frac{k\pi}{n}\right)\right) = (-1)^k$ für $k = 0, \ldots, n$
4. $T_n\left(\cos\left(\frac{(2k+1)\pi}{2n}\right)\right) = 0$ für $k = 0, \ldots, n-1$

Definition 2.4.4: *(Chebyshev-Knoten)* Die $(n+1)$ Chebyshev-Knoten $x_0, \ldots, x_n$ 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:

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

Definition 2.4.6: *(Chebyshev-Abszissen)* Die $(n-1)$ Chebyshev-Abszissen $x_0, \ldots, x_{n-2}$ 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:

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

Oder $k = 1, \ldots, n-1$ 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

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

Sie $(T_0, \ldots, T_n)$ 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 $(x_0, \ldots, x_n)$ 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)

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

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 $n \to \infty$

Interpolationspolynom **Satz 2.4.15**

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

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

wobei für die c_k gilt:

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

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

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	$\mathcal{O}\left((n+1)^2\right)$ Operationen
Dividierte Differenzen	$\mathcal{O}\left(\frac{n(n+1)}{2}\right)$ Operationen (zum Vergleich)
c_k mittels FFT	$\mathcal{O}\left(n\log(n)\right)$ 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 das Interpolationspolynom $p(x)$ mit Hilfe einer Rückwärtsrekursion berechnen

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

Auf der nächsten Seite findet sich eine saubere, effiziente Implementation des Clenshaw-Algorithmus:

```python
def clenshaw(coeffs: np.ndarray, x: np.ndarray):
    n = len(coeffs) - 1
    # initialise temporary variables
    d_prev_prev, d_prev, d_curr = (
        np.zeros_like(x),
        np.zeros_like(x),
        np.zeros_like(x),
    )

    for k in range(n, -1, -1):  # backward recursion
        d_curr = coeffs[k] + 2 * x * d_prev - d_prev_prev
        d_prev_prev, d_prev = d_prev, d_curr

    return d_prev - x * d_prev_prev
```

In `numpy` kann man mit `np.polynomial.chebyshev.chebfit` ein polyfit für Chebyshev-Polynome durchführen und mit `np.polynomial.chebyshev.chebder` die Ableitungen der Approximation berechnen. Die `chebder`-Funktion nimmt die normalen Chebyshev-Koeffizienten als Argument, die man einfach mit folgendem Code berechnen kann:

```python
def get_cheb_coeffs(abscissa: np.ndarray)
    n = len(abscissa) - 1
    dct_vals = scipy.fft.dct(abscissa, type=1)

    coeffs = dct_vals / n
    coeffs[0] /= 2
    self.coeffs = coeffs
```

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.

Intuition: Wir haben eine Datenmenge D, die die y-Werte einer Frequenzmessung an N äquidistanten Punkten enthält. Die Fourier-Transformation dieser Datenmenge ergibt eine neue Datenmenge, nennen wir sie F, die, wenn geplottet, einem Plot der Frequenzanalyse entsprechen. Dies ist auch korrekt, denn die Fourier-Transformation macht (vereinfacht) genau das; Sie macht einen Basiswechsel auf der Datenmenge D, so dass die Frequenz auf der x-Achse und die "Häufigkeit" deren auf der y-Achse aufgetragen werden, oder formaler, so dass wir statt einer Funktion der Zeit eine Funktion der Frequenz haben.

Das Inverse davon nimmt eine Funktion der Frequenz und transformiert diese in eine Funktion der Zeit

---

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: $p_m : \mathbb{R} \to \mathbb{C}$ 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 **Definition 3.1.3**

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

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

während die L^2-Norm (= Euklidische Norm, also die normale Vektornorm) auf $(0,1)$ durch das Skalarprodukt

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

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

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

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

In Anwendungen findet sich oft das Intervall $\left[-\frac{T}{2}, \frac{T}{2}\right]$. Dann verwandeln sich die Integrale in die Form $\frac{1}{T}\int_{\frac{T}{2}}^{-\frac{T}{2}} (\ldots) \, \mathrm{d}t$ 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 $n \to \infty$ gilt, dass $||f - f_n||_{L^2(0,1)} \to 0$

Fourier-Reihe **Satz 3.1.8**

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

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

wobei die Fourier-Koeffizienten

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

definiert sind. Es gilt die Parseval'sche Gleichung:

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

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 $\widehat{f}(k)$. 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

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

Fourier-Reihe **Satz 3.1.15**

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

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

Satz 3.1.16: Wenn $\int_0^1 |f^{(n)}(t)| \, \mathrm{d}t < \infty$, dann ist $\widehat{f}(k) = \mathcal{O}\left(k^{-n}\right)$

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*** gennant 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:

$$b - a + \frac{1}{\pi} \sum_{k \neq 0} e^{-ikc} \frac{\sin(kd)}{k} e^{i2\pi kt}, \qquad t \in [0,1]$$

Mit $c = \pi(a+b)$ und $d = \pi(b-a)$

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 $l = 0, 1 \ldots, N-1$ und N die Anzahl der Intervalle ist:

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

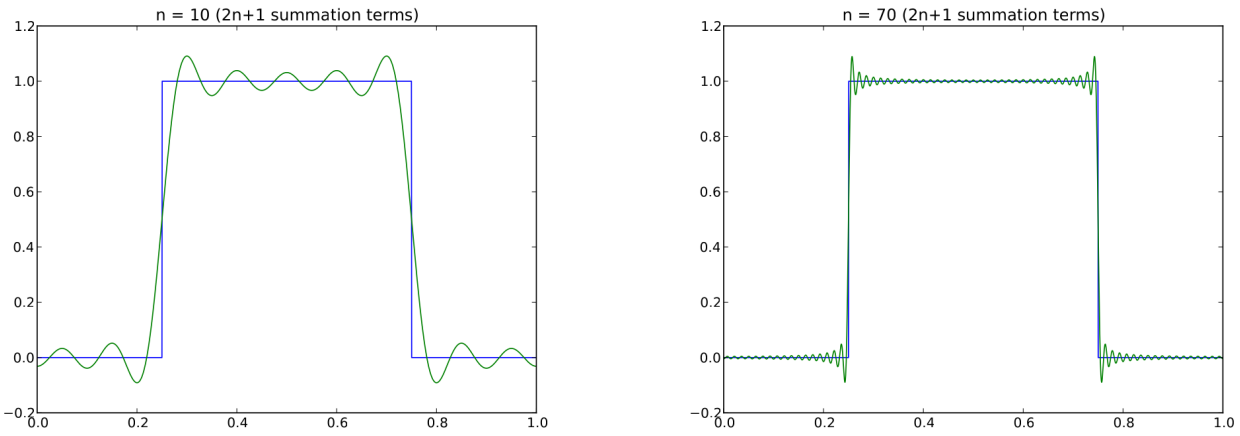


Abbildung 1: Überschwingungen der Fourier-Reihe der charakteristischen Funktion des Intervalls $[a, b] \subseteq]0, 1[$. (Abbildung aus dem Vorlesungsdokument von Prof. V. Gradinaru, Seite 69)

3.2 Diskrete Fourier Transformation

3.2.1 Motivation

Nutzen wir die Trapezregel um approximativ die Fourierkoeffizienten $\widehat{f}_N(k)$ auf äquidistanten Punkten $l_t = \frac{l}{N}$ $(0 \leq l \leq N-1)$ zu bestimmen, erhalten wir tatsächlich ein Polynom p_{N-1} welches die Interpolationsbedingung erfüllt:

$$p_{N-1}(t) = \sum_{k=-\frac{N}{2}}^{\frac{N}{2}-1} \widehat{f}_N(k) e^{2\pi i k t}$$

Der Beweis hierfür ist im Skript auf p. 71. Die N-te Einheitswurzel wird hier definiert:

Definition 3.2.1: *(N-te Einheitswurzel)* $\omega_N := \exp(\frac{-2\pi i}{N})$

Bemerkung 3.2.2: *(Eigenschaften von ω_N)*

$$\forall j, k \in \mathbb{Z}: \quad \omega_N^{k+jN} = \omega_N^k \qquad \omega_N^N = 1 \qquad \sum_{k=0}^{N-1} \omega_N^{kj} = \begin{cases} N, & j \equiv_N 0 \\ 0, & \text{sonst} \end{cases}$$
$$\forall k \in \mathbb{Z}, t \in \mathbb{R}: \quad \omega_N^{t+kN} = \omega_N^t \qquad \omega_N^{N/2} = -1$$

3.2.2 Konstruktion

Wir definieren die Trigonometrische Basis. Den Basiswechsel zu dieser Basis nennen wir diskrete Fourier Transformation.

Definition 3.2.3: *(Trigonometrische Basis)*

$$\{v_0, \ldots, v_{N-1}\} \text{ ist eine Basis von } \mathbb{C}^N, \text{ wobei } v_k = \begin{bmatrix} \omega_N^{0 \cdot k} \\ \omega_N^{1 \cdot k} \\ \vdots \\ \omega_N^{(N-1) \cdot k} \end{bmatrix} \in \mathbb{C}^N$$

Die symmetrische, nicht hermitesche Matrix $V = [v_0, \ \ldots \ , v_{N-1}]$ ist eine orthogonale Basis für $\mathbb{C}^N$: $V^H V = N \cdot I_N$.
Ebenfalls ist V die Basiswechsel Matrix Trigonometrische Basis $(z) \mapsto$ Standardbasis (y).
An Hand von V definieren wir gleich die Fourier-Matrix F_N.

$$y = Vz \implies z = V^{-1} y = \frac{1}{N} V^H y = \frac{1}{N} \underbrace{F_N}_{:=V^H} y$$

Der Eintrag y_l enstspricht einem Glied der Fourier-Reihe ausgewertet in $\frac{l}{N} \in [0, 1)$.
Die diskreten Fourier-Koeffizienten γ_k sind eine Umsortierung der Koeffizienten der trigonometrischen Basis.

$$y = \underbrace{\sum_{k=0}^{N-1} y_k e_{k+1}}_{y \text{ in Komponenten}} = \underbrace{\sum_{k=0}^{N-1} z_k v_k}_{\text{in Trig. Basis}} = \sum_{k=0}^{N-1} z_k \begin{bmatrix} \omega_N^{0\cdot k} \\ \omega_N^{1\cdot k} \\ \omega_N^{2\cdot k} \\ \vdots \\ \omega_N^{(N-1)\cdot k} \end{bmatrix}$$

$$y_l = \sum_{k=0}^{N-1} z_k \omega_N^{l\cdot k} \overset{\text{S. 75}}{=} \sum_{k=-N/2}^{N/2-1} \gamma_k \cdot \exp(\frac{2\pi i}{N} lk)$$

$$\text{wobei } \gamma_k = \begin{cases} z_k, & 0 < k \leq \frac{N}{2} - 1 \\ z_k + N, & -\frac{N}{2} \leq k < 0 \end{cases}$$

Definition 3.2.4: *(Fourier-Matrix)*

$$F_N := V^H = [v_0, \dots, v_{N-1}]^H = \begin{bmatrix} \omega_N^0 & \omega_N^0 & \cdots & \omega_N^0 \\ \omega_N^0 & \omega_N^1 & \cdots & \omega_N^{N-1} \\ \omega_N^0 & \omega_N^2 & \cdots & \omega_N^{2(N-1)} \\ \vdots & \vdots & & \vdots \\ \omega_N^0 & \omega_N^{N-1} & \cdots & \omega_N^{(N-1)^2} \end{bmatrix} = \left[\omega_N^{jk}\right]_{j,k=0}^{N-1} \in \mathbb{C}^{N\times N}$$

Die skalierte Fourier-Matrix $\frac{1}{\sqrt{N}} F_N$ hat einige besondere Eigenschaften.

Satz 3.2.6: Die skalierte Fourier-Matrix $\frac{1}{\sqrt{N}} F_N$ ist unitär: $F_N^{-1} = \frac{1}{N} F_N^H = \frac{1}{N} \overline{F_N}$

Bemerkung 3.2.7: *(Eigenwerte von $\frac{1}{\sqrt{N}} F_N$)* Die λ von $\frac{1}{\sqrt{N}} F_N$ liegen in $\{1, -1, i, -i\}$.

Die diskrete Fourier-Transformation ist nun einfach die Anwendung der Basiswechsel-Matrix F_N.

Definition 3.2.5: *(Diskrete Fourier-Transformation)* $\mathcal{F}_N : \mathbb{C} \to \mathbb{C}$ s.d. $\mathcal{F}_N(y) = F_N y$

$$\text{Für } c = \mathcal{F}_N(y) \text{ gilt:} \quad c_k = \sum_{j=0}^{N-1} y_j \omega_N^{kj}$$

c lässt sich als Repräsentation von y im Frequenzbereich interpetieren. Durch die DFT können wir nun jederzeit zwischen der normalen und der Frequenz-perspektive wechseln. Das ermöglicht einige interessante Anwendungen.

3.2.3 DFT in Numpy

Sei y in der Standardbasis, und $c = \mathcal{F}_N(y)$, also y in der trig. Basis.

$$c = F_N \times y = \texttt{fft}(y) \quad \textit{(DFT in numpy)} y = \frac{1}{N} F_N^H c = \texttt{ifft}(c) \quad \textit{(Inverse DFT in numpy)}$$

Um zur ursprünglichen Darstellung des trig. Polynoms zurück zu kommen, müssen wir die Koeffizienten umsortieren: Seien $z = \frac{1}{N} F_N y$ und $\zeta = \texttt{fft.fftshift}(z)$.

$$f(x) \approx \underbrace{\sum_{k=-N/2}^{N/2-1} \zeta_k \cdot e^{2\pi i k x}}_{\text{Form des trig. Polynoms}}$$

Bemerkung 3.2.13: Man kann mit dieser Approximation einfach die L^2-Norm und Ableitungen berechnen:

$$||f||_{L^2}^2 \approx \left\| \sum_{k=-N/2}^{N/2-1} \zeta_k \cdot e^{2\pi i k x} \right\|_{L^2}^2 = \sum_{k=-N/2}^{N/2-1} |\zeta_k|^2 = \|z\|_{L^2}^2$$

$$f'(t) \approx \sum_{k=-N/2}^{N/2-1} (2\pi i k) \zeta_k \cdot e^{2\pi i k x}$$

3.2.4 DFT & Lineare Algebra

Definition 3.2.25: *(Zirkulant)* Für einen vektor $c \in \mathbb{R}^N$ hat der Zirkulant $C \in \mathbb{R}^{N \times N}$ die Form:

$$C = \begin{bmatrix} c_0 & c_{N-1} & c_{N-2} & \cdots & c_3 & c_2 & c_1 \\ c_1 & c_0 & c_{N-1} & \cdots & c_4 & c_3 & c_2 \\ c_2 & c_1 & c_0 & \cdots & c_5 & c_4 & c_3 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ c_{N-3} & c_{N-4} & c_{N-5} & \cdots & c_0 & c_{N-1} & c_{N-2} \\ c_{N-2} & c_{N-3} & c_{N-4} & \cdots & c_1 & c_0 & c_{N-1} \\ c_{N-1} & c_{N-2} & c_{N-3} & \cdots & c_2 & c_1 & c_0 \end{bmatrix} \qquad S_N = \begin{bmatrix} 0 & 0 & \cdots & \cdots & 0 & 1 \\ 1 & 0 & \cdots & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & \cdots & 0 & 0 \\ 0 & 0 & \cdots & \cdots & 1 & 0 \end{bmatrix}$$

Die Shift Matrix S_N ist der Zirkulant für $c = e_2$. S_N ist eine Permutationsmatrix, die alle Einträge nach vorne schiebt.

$$S_N \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{N-1} \end{bmatrix} = \begin{bmatrix} x_{N-1} \\ x_0 \\ \vdots \\ x_{N-2} \end{bmatrix} \qquad S_N^\top \begin{bmatrix} x_{N-1} \\ x_0 \\ \vdots \\ x_{N-2} \end{bmatrix} = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{N-1} \end{bmatrix}$$

Die Shift-Matrix hat einen speziellen Bezug zu den Spaltenvektoren v_k von F_N, und auch allen anderen Zirkulanten C.

Bemerkung 3.2.26: Der k-te Fourier-Vektor v_k ist ein Eigenvektor von S_N zu $\lambda_k = e^{2\pi i \frac{k}{N}}$.

Satz 3.2.27: *(Diagonalisierung von Zirkulanten)* Die Eigenvektoren von S_N diagonalisieren jeden Zirkulanten C, und sind d.h. auch die Eigenvektoren von C. Die Eigenwerte erhält man aus $p(z) = c_0 z^0 + \ldots + c_{N-1} z^{N-1}$.

Eine Operation mit vielen Anwendungen ist die Faltung. Sie hat einige Beziehungen zur Fourier-Transformation.

Definition 3.2.28: *(Faltung)* $a * b := (c_k)_{k \in \mathbb{Z}} = \sum_{n=-\infty}^{\infty} a_n b_{k-n}$, wobei $(a_k)_{k \in \mathbb{Z}}$, $(b_k)_{k \in \mathbb{Z}}$ unendliche Folgen sind.

Die Faltung von $a = [a_0, \ldots, a_{N-1}]^\top, b = [b_0, \ldots, b_{N-1}]^\top$ ist leicht: Man erweitert beide Vektoren mit Nullen.

Definition 3.2.29: *(Zyklische Faltung)* Für N-periodische Folgen oder Vektoren der Länge N:

$$c = a \circledast b \qquad \text{s.d.} \sum_{n=0}^{N-1} a_n b_{k-n} \equiv_N \sum_{n=0}^{N-1} b_n a_{n-k}$$

Bemerkung 3.2.32: Zyklische Faltungen von Vektoren kann man mit Zirkulanten berechnen.

$$c = a \circledast b = Ab = \underbrace{\begin{bmatrix} a_0 & \cdots & a_{N-1} \\ \vdots & \ddots & \vdots \\ a_{N-1} & \cdots & a_0 \end{bmatrix}}_{\text{Zirkulant von } a} b$$

Bemerkung 3.2.30: Eine Multiplikation von Polynomen g, h entspricht einer Faltung im Frequenzbereich.

$$\mathcal{F}_N(\underbrace{g * h}_{\text{Standard Basis}}) = \underbrace{\mathcal{F}_N(g) \cdot \mathcal{F}_N(h)}_{\text{Trigonometrische Basis}}$$

Im Fall von T-periodischen Funktionen gilt: $(g * h)(x) = \frac{1}{T} \int_0^T g(t) h(x - t)$.

Bemerkung 3.2.31: Da F_N jeden Zirkulant C diagonalisiert (Satz 3.4.27), gilt sogar:

$$c = a \circledast b = Ab = F_N^{-1} p(D) F_N b \qquad (p(D) \text{ ist Diagonalmatrix der } \lambda \text{ von } C)$$

Man erhält so letzendlich das Faltungs-Theorem: Die F_N-Transformierte einer Faltung ist genau das gleiche wie die Multiplikation zweier F_N-Transormierten. Da die DFT in $\mathcal{O}(n \log(n))$ (Kap. 3.3) geht, gilt dies nun auch für die Faltung.

$$F_N c = \text{diag}(F_N a) F_N b$$

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 $\mathcal{O}\left(N^2\right)$ hat, so hat der Fast Fourier Transform Algorithmus nur eine Laufzeit von $\mathcal{O}\left(N \log(N)\right)$, was bei $N = 1024$ bereits eine Laufzeitsverbesserung von $100\times$ mit sich bringt ($\mathcal{O}\left(10\,000\right)$ vs $\mathcal{O}\left(1\,000\,000\right)$ Operationen)! Die untenstehende Abbildung 2 findet sich, zusammen mit dem Code, mit der sie produziert wurde im Skript auf Seite 86-88

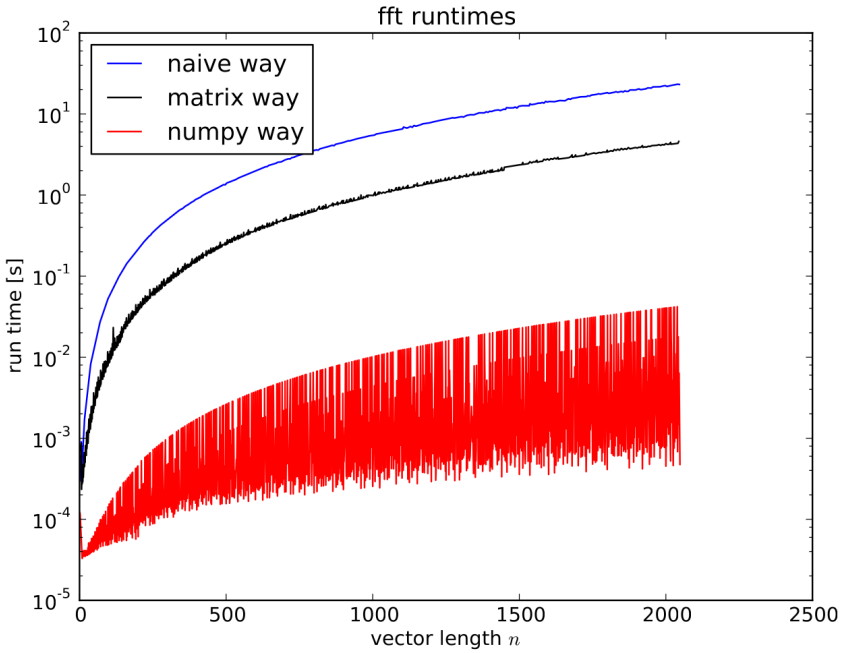


Abbildung 2: Vergleich der Laufzeit von verschiedenen Fourier-Transformations-Algorithmen. (Abbildung 3.3.3 aus dem Vorlesungsdokument von Prof. V. Gradinaru, Seite 88)

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 gut

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. $\mathcal{O}\left(N^2\right)$, besonders für N gross

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

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

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.

In `numpy` gibt es die Funktionen `np.fft.fft` (Vorwärts FFT), `np.fft.ifft` (Rückwärts FFT). `scipy.fft` liefert dieselben Funktionen und sie sind oft etwas schneller als die von `numpy`

3.4 Trigonometrische Interpolation

3.4.1 Von Approximation zur Interpolation

Wir erinnern uns daran, dass wir die Fourier-Approximation durch den Abbruch der unendlichen Fourier-Reihe erhalten, oder in anderen Worten, wir verkleinern die Limiten der Summe.

Bemerkung 3.4.1: *(DFT mit $N = 2n$ Koeffizienten an Punkten $\frac{l}{N}$ für $l = 0, 1, \dots, N-1$)*

Der Shift ist hier gegeben durch (für $k \geq 0$ ist $\gamma_k = \widehat{f}_N(k)$ und für $k < 0$ ist $\gamma_k = \widehat{f}_N(N+k)$)

$$f_{N-1}(x) = \sum_{k=-n}^{n-1} \gamma_k e^{2\pi i k x} = \sum_{k=0}^{n-1} \gamma_k e^{2\pi i k x} + \sum_{k=-n}^{-1} \gamma_k e^{2\pi i k x}$$

$$\Leftrightarrow f_{N-1}(x) = \frac{1}{N} \left(\sum_{j=0}^{N-1} \left(f\left(\frac{j}{n}\right) \sum_{k=-n}^{n-1} e^{2\pi i k \left(x - \frac{j}{N}\right)} \right) \right)$$

Wenn wir die Funktion nun an der Stelle $\frac{l}{N}$ auswerten so erhalten wir:

$$f_{N-1}\left(\frac{l}{N}\right) = \dots = f\left(\frac{l}{N}\right)$$

was aufgrund der Orthogonalität der diskreten Fourier-Vektoren funktioniert, welche besagt, dass $\sum_{k=-n}^{n-1} \omega_N^{k(j-l)} = 0$, für alle $j \neq l$. Für $j = l$ ergibt die Summe N.

Dies heisst also, dass die Fourier-Approximation die Interpolationsbedingungen an den Punkten $\frac{l}{N}$ erfüllt, also können wir die Lösung der Interpolationsaufgabe $p_{N-1}\left(\frac{l}{N}\right) = f\left(\frac{l}{N}\right)$ f $l = 0, 1, \dots, N-1$ im Raum

$$\mathcal{T}_N = \text{span}\{e^{2\pi i j t} \mid j = -\left\lfloor \frac{N-1}{2} \right\rfloor, \dots, \left\lfloor \frac{N}{2} \right\rfloor\}$$

folgendermassen finden können:

(1) Mittels Gleichungssystem $\sum_j \gamma_j e^{2\pi i j t_l} = f(t_l)$ für $l = 0, \dots, N-1$. Operationen: $\mathcal{O}\left(N^3\right)$

(2) Mittels FFT in $\mathcal{O}\left(N \log(N)\right)$ Operationen, aber nur falls die Punkte äquidistant sind, also $t_l = \frac{l}{N}$. Dann ist die Matrix des obigen Gleichungssystems F_N^{-1}

Unten findet sich Python code der mit den unterschiedlichen Methoden die Koeffizienten des Trigonometrischen Polynoms bestimmt.

```python
def get_coeff_trig_poly(t: np.ndarray, y: np.ndarray):
    N = y.shape[0]
    if N % 2 == 1:
        n = (N - 1.0) / 2.0
        M = np.exp(2 * np.pi * 1j * np.outer(t, np.arange(-n, n + 1)))
    else:
        n = N / 2.0
        M = np.exp(2 * np.pi * 1j * np.outer(t, np.arange(-n, n)))
    c = np.linalg.solve(M, y)
    return c

N = 2**12
t = np.linspace(0, 1, N, endpoint=False)
y = np.random.rand(N)
direct = get_coeff_trig_poly(t, y)
using_fft = np.fft.fftshift(np.fft.fft(y) / N)
using_ifft = np.conj(np.fft.fftshift(np.fft.ifft(y)))
```

3.4.2 Zero-Padding-Auswertung

Ein trigonometrisches Polynom $p_{N-1}(t)$ kann effizient an den äquidistanten Punkten $\frac{k}{M}$ mit $M > N$ ausgewertet werden, für $k = 0, \dots, M-1$. Dazu muss das Polynom $p_{N-1} \in \mathcal{T}_N \subseteq \mathcal{T}_M$ in der trigonometrischen Basis $\mathcal{T}_M$ neugeschrieben werden, in dem man ***Zero-Padding*** verwendet, also Nullen im Koeffizientenvektor an den Stellen höheren Frequenzen einfügt.

TODO: Insert cleaned up code from Page 95 (part of exercises)

Die folgende Funktion wird im Script `evaliptrig` genannt.

```python
def evaluate_trigonometric_interpolation_polynomial(y: np.ndarray, N: int):
    n = len(y)
    if (n % 2) == 0:
        c = np.fft.ifft(y)  # Fourier coefficients
        a = np.zeros(N, dtype=complex)

        # Zero padding
        a[: n // 2] = c[: n // 2]
        a[N - n // 2 :] = c[n // 2 :]
        return np.fft.fft(a)
    else:
        raise ValueError("odd length")
```

3.5 Fehlerabschätzungen

Konvergenz **Definition 3.5.1**

Algebraische Konvergenz
Wenn der Fehler $E(n) = \mathcal{O}\left(\frac{1}{n^p}\right)$ mit $p > 0$ ist

Exponentielle Konvergenz
Wenn der Fehler $E(n) = \mathcal{O}\left(q^n\right)$ mit $0 \leq q < 1$

Beispiel 3.5.2: Zur Fehlerbetrachtung verwenden wir drei Funktionen $f : [0,1] \to \mathbb{R}$, welche wir mit trigonometrischer Interpolation an den Punkten $\frac{k}{N}$ approximieren:

(I) Stufenfunktion (periodische Fortsetzung von f) $f : [0,1] \to \mathbb{R}$ mit $f(t) = \begin{cases} 0 & \text{für } \left|t - \frac{1}{2}\right| > \frac{1}{4} \\ 1 & \text{für } \left|t - \frac{1}{2}\right| \leq \frac{1}{4} \end{cases}$

(II) Periodische, glatte Funktion $h : \mathbb{R} \to \mathbb{R}$ mit $h(t) = \dfrac{1}{\sqrt{1 + \frac{1}{2}\sin(2\pi t)}}$

(III) Hutfunktion (periodische Fortsetzung von h) $g : [0,1] \to \mathbb{R}$ mit $g(t) = \left|t - \frac{1}{2}\right|$

Die untenstehende Abbildung 3 beinhaltet einen Plot, auf dem die Konvergenz in Abhängigkeit des Grades des Interpolationspolynoms aufgetragen ist.

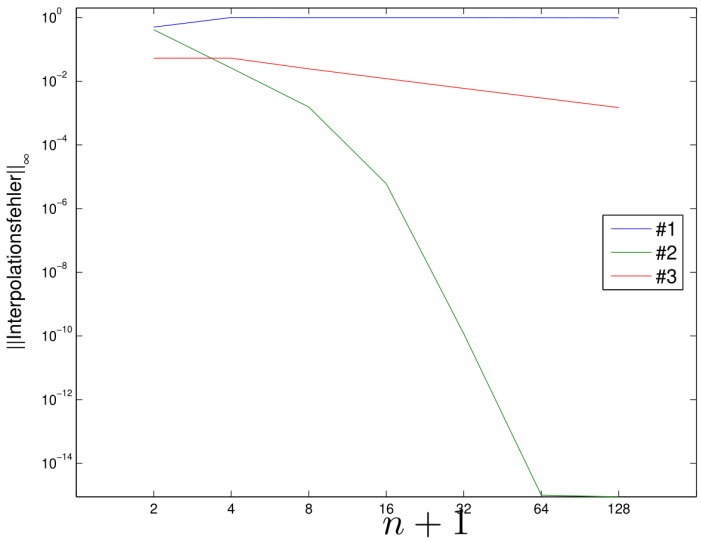


Abbildung 3: Interpolierungsfehler der Beispiele. Algebraische Konvergenz für (I) und (III), exponentielle für (II). (Abbildung 3.5.2 aus dem Vorlesungsdokument von Prof. V. Gradinaru, Seite 96)

Auch hier tritt das Gibbs-Phänomen wieder an den Sprungstellen von $f(t)$ auf. Dies verursacht die Verlangsamung der Konvergenz in den Stellen, in welchen die Funktion nicht glatt ist.

Beispiel 3.5.4: Sei für $\alpha \in [0,1)$ $f(t) = \dfrac{1}{\sqrt{1-\alpha\sin(2\pi t)}}$. Die Konvergenz ist exponentiell in n und je kleiner α, desto schneller ist sie. In der untenstehenden Abbildung 4 sind einige Beispiele aufgetragen:

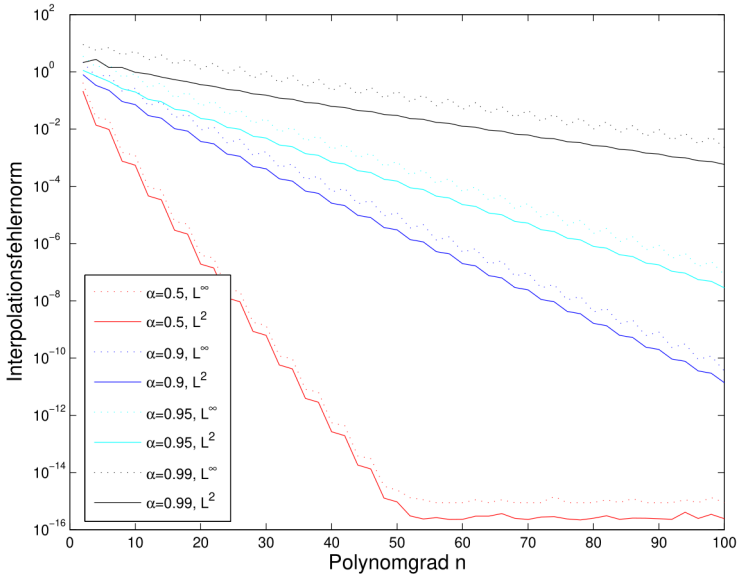


Abbildung 4: Fehler bei der trigonometrischen Interpolation. (Abbildung 3.5.5 aus dem Vorlesungsdokument von Prof. V. Gradinaru, Seite 98)

Aliasing — Satz 3.5.6

Der k-te Fourier-Koeffizient des N-ten trigonometrischen Interpolationspolynoms unterscheidet sich vom k-ten Fourier-Koeffizienten von f gerade um die Summe aller Fourier-Koeffizienten, die um ganze Vielfache von N vom k-ten Fourier-Koeffizienten verschoben sind:

$$\widehat{p}_N(k) - \widehat{f}(k) = \sum_{j\neq 0\in\mathbb{Z}} \widehat{f}(k+jN)$$

Korollar 3.5.7: Für $f \in \mathbb{C}^p([0,1])$ mit $p \geq 1$ und f 1-periodisch, dann gilt: $|\widehat{p}_N(k) - \widehat{f}(k)| = \mathcal{O}\left((N^{-p})\right)$ für $|k| \leq \frac{N}{2}$

Das heisst also, dass die Fourier-Koeffizienten von f bei kleinen Frequenzen $\left(\text{hier } |k| < \frac{N}{2}\right)$ sehr gut durch die Fourier-Koeffizienten des trigonometrischen Interpolationspolynoms approximiert werden.

Fehler der trigonometrischen Interpolation — Satz 3.5.8

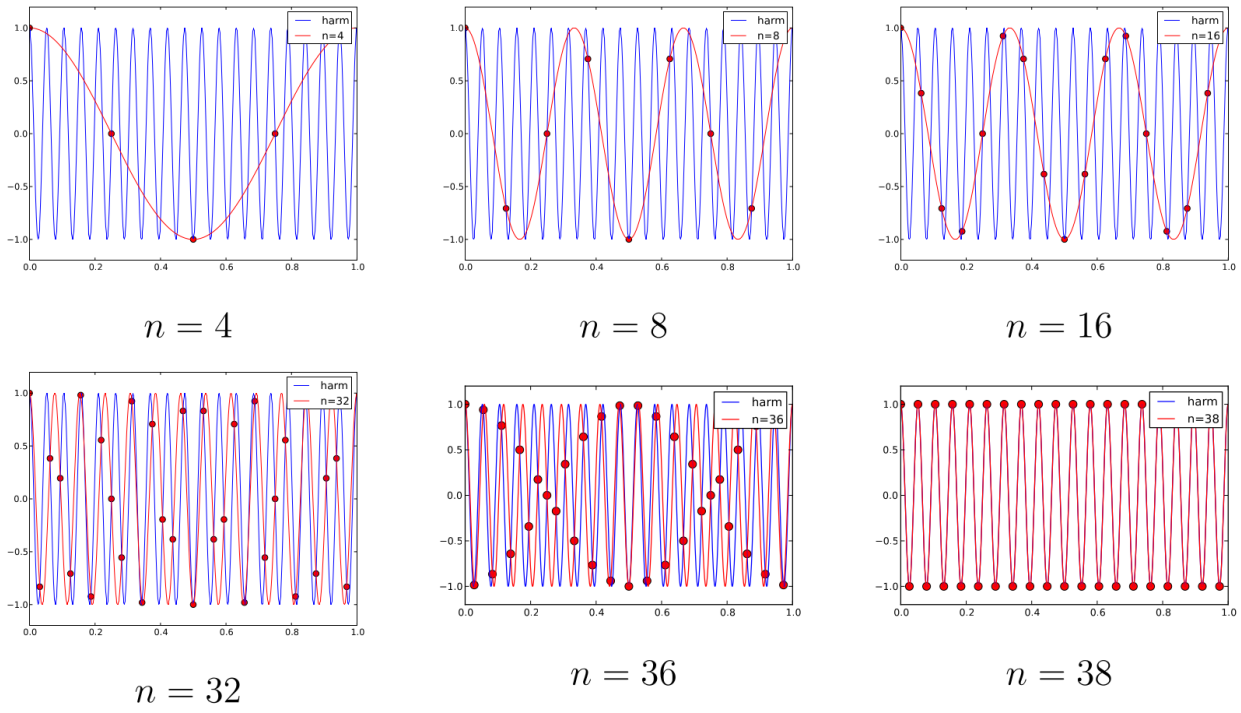
Falls f 1-periodisch ist und die Reihe $\sum_{k\in\mathbb{Z}} |\widehat{f}(k)|$ absolut konvergiert, dann ist der Approximationsfehler definiert als:

$$|p_N(x) - f(x)| \leq 2 \sum_{|k|\geq\frac{N}{2}} |\widehat{f}(k)| \qquad \forall x \in \mathbb{R}$$

Da durch diesen Satz die obere Schranke für den Approximationsfehler durch die schwer approximierbaren Fourier-Koeffizienten $\widehat{f}(k)$ gegeben ist, heisst das folgendes für die Approximation von Polynomen von Grad $\deg(P(x)) < n$ für unser Approximationspolynom von Grad $\deg(P_N(x)) = n$:

Korollar 3.5.9: *(Abtasttheorem)* Sei f 1-periodisch mit maximaler Frequenz m, also $\widehat{f}(k) = 0\ \forall |k| > m$. Falls $N > 2m$, dann gilt $p_N(x) = f(x)\ \forall x$

Beispiel 3.5.10: Ein Beispiel aus der Musik: Wir haben ein analoges Signal und wollen es digitalisieren. Wir messen die Spannungswerte in äquidistanten Punkten. Falls wir jedoch die Frequenz der Messung zu niedrig wählen, so kann ein total falsches Interpolationspolynom entstehen, wie in der untenstehenden Abbildung 5 zu sehen: Für unser Signal



$n = 4$ $n = 8$ $n = 16$

$n = 32$ $n = 36$ $n = 38$

Abbildung 5: Aliasing für $f(t) = \cos(2\pi \cdot 19t)$. (Abbildung 3.5.10 aus dem Vorlesungsdokument von Prof. V. Gradinaru, Seite 100)

bedeutet das also, dass wir eine Art Verzerrung auf der Aufnahme haben, oder für Autoräder, dass es so scheint, als würden sich die Räder rückwärts drehen.

Fehlerabschätzung **Satz 3.5.11**

Sei $f^{(k)} \in L^2(0,1) \ \ \forall k \in \mathbb{N}$, dann gilt:

$$||f - p_N(f)||_{L^2(0,1)} \leq \sqrt{1 + c_k} N^{-k} ||f^{(k)}||_{L^2(0,1)} \text{ wobei } c_k = 2 \sum_{l=1}^{\infty} (2l-1)^{-2k}$$

Also, je mehr Ableitungen in $L^2(0,1)$ liegen, desto kleiner ist der Fehler.

Im Skript auf Seiten 101 und 102 gibt es einige Abbildungen die eine gewisse Intuition hinter der Approximation und den entstandenen Fehlern gibt.

3.6 DFT und Chebyshev-Interpolation

Mithilfe der DFT können günstig und einfach die Chebyshev-Koeffizienten (c_k) berechnet werden. Die Idee basiert auf dem Satz 2.4.16, durch welchen schon schnell klar wird, dass es eine Verbindung zwischen den Fourier-Koeffizienten und Chebyshev-Koeffizienten gibt.

Die Chebyshev-Knoten sind folgendermassen definiert:

$$t_k := \cos\left(\frac{2k+1}{2(n+1)}\pi\right), \quad k = 0, \dots, n$$

Mit den Hilfsfunktionen $g : [-1,1] \to \mathbb{C}, s \mapsto f(\cos(2\pi s))$ und $q : [-1,1] \to \mathbb{C}, s \mapsto p(\cos(2\pi s))$, können wir folgendes mit der Interpolationsbedingung $f(t_k) = p(t_k)$ tun:

$$f(t_k) = p(t_k) \iff g\left(\frac{2k+1}{4(n+1)}\right) = p\left(\frac{2k+1}{4(n+1)}\right)$$

Wir wenden nun die Translation $s^* = s + \frac{1}{4n+1}$ an, die Hilfsfunktionen sind dann $g*(s) = g(s^*)$ und $q^*(s) = q(s^*)$ und man kann zeigen (Seite 100 im Skript), dass q^* das trigonometrische Interpolationspolynom von g^* ist, also kann man eine Chebyshev-Interpolation durch eine DFT durchführen. Folglich überträgt sich auch die Fehlerabschätzung. Die Interpolationsbedingungen sind folgendermassen definiert:

$$q\left(\frac{k}{2(n+1)} + \frac{1}{4(n+1)}\right) = z_k := \begin{cases} y_k & \text{für } k = 0, \dots, n \\ y_{2n+1-k} & \text{für } k = n, \dots, 2n+1 \end{cases}$$

Um das ganze zu implementieren ist eine andere Darstellung nützlich:

$$\cos(2\pi\xi_k) \text{ mit } \xi_k = \frac{2k+1}{4(n+1)}$$

Durch Umformungen (Seite 101 im Skript) erhalten wir:

$$z_l = \sum_{-n}^{n} \zeta_j \exp\left(2\pi i j \widetilde{\xi}_l\right) \text{ mit } \widetilde{\xi}_l = \frac{l}{2n+2} \text{ für } 0, 1, \dots, 2n+1$$

$$\zeta_j = c_j \exp\left(\frac{2\pi i j}{4(n+1)}\right) \text{ für } j = -n, \dots, -1, 0, 1, \dots, n$$

Und mit weitern Umformungen erhalten wir

$$F_{2n+2}^{-1}\left[\exp\left(\frac{\pi i n l}{n+1}\right) z_l\right] = [\zeta_{k-n}]$$

Auf Seite 102 im Skript findet sich auch eine effiziente Implementation dessen.

Bemerkung 3.6.2: Die Formel in Satz 2.4.16 (und in der eben erwähnten Implementierung) sind nichts anderes als eine Version der DCT (Discrete Cosine Transform). Dies ist eine günstigere, aber beschränktere Variante der DFT, mit der nur reellwertige, gerade Funktionen interpoliert werden können.

In `numpy` benutzen wir `scipy.fft.dct`. Dazu müssen die Mesungen in den Punkten $x_j = \cos\left((j+0.5)\cdot\frac{\pi}{N}\right)$

Bemerkung 3.6.3: Die Chebyshev-Koeffizienten c_j können folgendermassen berechnet werden:

$$c_j = \frac{1}{\pi}\int_0^{2\pi} f(\cos(\varphi))\cos(j\varphi)\,\mathrm{d}\varphi$$

Eine weitere effiziente Interpolation findet sich auf Seiten 104 - 105 im Skript

4 Stückweise Polynomiale Interpolation

4.1 Stückweise Lineare Interpolation

Globale Interpolation (also Interpolation auf dem ganzen Intervall $]-\infty, \infty[$) funktioniert nur dann gut, wenn:
(a) die gegebenen Interpolationspunkte als Chebyshev-Knoten oder -Abszissen verwendet werden können
(b) die Funktion glatt ist

Es müssen beide obige Eigenschaften zutreffen. Eine Idee um die Einschränkungen zu reduzieren oder komplett zu entfernen ist es, das Intervall zu unterteilen, oder formaler, das Intervall $I = [a, b]$ in viele kleinere Intervalle zu zerlegen.

Wir haben dann ein Polynom vom Grad n auf jedem Teilintervall mit $n+1$ Punkten, was den Fehler verringert:

$$|f(x) - s(x)| < \frac{h^{n+1}}{(n+1)!}||f^{(n+1)}||_\infty$$

Seien $N+1$ Messpunkte gegeben. Wir verwenden sie als Knoten (im Englischen *breakpoints* gennant. Die Knoten sind also nicht dasselbe wie in den vorigen Kapiteln, es gibt aber keinen wirklich sinnvollen Namen im Deutschen) diese $N+1$ Messpunkte. Die Knoten dienen Paarweise als Abgrenzung der neuen, kleinen Intervalle, die wir erstellt haben. Die linearen Interpolanten für jedes Intervall sind (mit $h_j = x_j - x_{j-1}$):

$$s_j(x) = y_{j-1}\frac{x_j - x}{h_j} + y_j\frac{x - x_{j-1}}{h_j} \quad \text{für } x \in [x_{j-1}, x_j]$$

Wie man nun zu dieser Formel kommt: Sei $\chi(t) = t \ \ \forall t \in [0,1]$. Die Funktion $f(t) = y_0\chi(1-t) + y_1\chi(t)$ hat also die Interpolationseigenschaften $f(0) = y_0$ und $f(1) = y_1$ und ist linear in t. Die Interpolation $s_j(x)$ auf $[x_{j-1}, x_j]$ entsteht dann also aus f mit Variablenwechsel $t = \frac{x - x_{j-1}}{h_j} \in [0,1] \leftrightarrow x = x_{j-1} + h_j t$, also gilt:

$$s_j(x) = y_{j-1}\chi\left(\frac{x_j - x}{h_j}\right) + y_j\chi\left(\frac{x - x_{j-1}}{h_j}\right) \quad \text{für } x \in [x_{j-1}, x_j]$$

Dies ist eine lokale Interpolation und s_j ist 0 ausser im definierten Intervall. Die Idee des Variablenwechsel ist es, das Intervall, auf welchem die Funktion definiert ist von $[0,1]$ nach $[x_{j-1}, x_j]$ zu verschieben.

4.2 Kubische Hermite-Interpolation

Die Kubische Hermite-Interpolation (CHIP) produziert eine auf $[a,b]$ stetig differenzierbare Funktion, welche auf den Teilintervallen $[x_{j-1}, x_j]$ jeweils ein Polynom von Grad 3 ist. Wichtige Eigenschaft von Polynomen n-ten Grades ist, dass sie $n+1$ Freiheitsgrade haben (da sie $n+1$ freie Variabeln enthalten).

Nutzen wir wieder das Konzept von oben, und wählen eine Funktion $\varphi(t) = t^2(3-2t)$ für $t \in [0,1]$, so erfüllt $f(t) = y_0\varphi(1-t) + y_1\varphi(t)$ wieder unsere Interpolationseigenschaften $f(0) = y_0$ und $f(1) = y_1$ und wir vollziehen denselben Variablenwechsel wie oben. So erhalten wir:

$$p_j(x) = y_{j-1}\varphi\left(\frac{x_j - x}{h_j}\right) + y_j\varphi\left(\frac{x - x_{j-1}}{h_j}\right) \quad \text{für } x \in [x_{j-1}, x_j]$$

Wir haben folgende Ableitungen: $\varphi'(t) = 6t(1-t)$, also sind die Nullstellen dieser Funktion bei $t \in \{0,1\}$, weshalb auch die Ableitungen von p_j an den Stellen x_{j-1} und x_j verschwinden.

Für die Ableitungen definieren wir eine zweite Funktion $\psi(t) = t^2(t-1)$, welche offensichtlich die Nullstellen an $t \in \{0,1\}$ hat und deren Ableitung $\psi'(t) = t(3t-2)$. Mit demselben Variablenwechsel müssen wir die Kettenregel beachten:

$$q_j(x) = c_{j-1}h_j\psi\left(\frac{x - x_{j-1}}{h_j}\right) - c_j h_j\psi\left(\frac{x_j - x}{h_j}\right) \quad \text{für } x \in [x_{j-1}, x_j]$$

Die Interpolationsfunktion ist dann einfach die Summe $s_j(x) = p_j(x) + q_j(x) \quad \text{für } x \in [x_{j-1}, x_j]$

In `numpy` verwendet man `scipy.interpolate.Akima1DInterpolator` oder `PchipInterpolator`, welcher "former-haltender" ist, also wenn eine Funktion lokal monoton ist, so ist der Interpolant dort auch monoton. Bei anderen Interpolationsmethoden ist dies nicht garantiert (so auch nicht beim `Akima1DInterpolator`)

Wenn man den Parameter `method="makima"` bei `Akima1DInterpolator` mitgibt, wird eine neuere modifizierte Variante davon ausgeführt

Fehler der CHIP **Satz 4.2.2**

Sei $f \in C^4[a,b]$ und s der stückweise CHIP mit exakten Werten der Ableitungen $s'(x_j) = f'(x_j), s(x_j) = f(x_j)$ für $j = 0, \dots, N$ und sei s_j ein Polynom vom Grad 3, für $j = 1, \dots, N$. Dann gilt:

$$||f^{(k)} - s^{(k)}||_{L^\infty} \leq \frac{1}{384} h^{4-k} ||f^{(4)}||_{L^\infty}$$

mit $h = \max_{j=1,\dots,N}(x_j - x_{j-1})$ und $k = 0, 1$

4.3 Splines

Raum der Splines **Definition 4.3.1**

Sei $[a,b] \subseteq \mathbb{R}$ ein Intervall, sei $\mathcal{G} = \{a = x_0 < x_1 < \dots < x_N = b\}$ und sei $d \geq 1 \in \mathbb{N}$. Die Menge

$$\mathcal{S}_{d,\mathcal{G}} = \{s \in C^{d-1}[a,b], \quad s_j := s_{|[x_{j-1}, x_j]|} \text{ ist ein polynom von Grad höchstens } d\}$$

ist die Menge aller auf $[a,b]$ $(d-1)$ mal stetig ableitbaren Funktionen, die auf $\mathcal{G}$ aus stückweisen Polynomen von Grad höchtens d bestehen und wir der Raum der Splines vom Grad d, oder der Ordnung $(d+1)$ genannt

Bemerkung 4.3.2: Obige Definition ist undefiniert für $d = 0$, aber $\mathcal{S}_{d,\mathcal{G}}$ kann als die Menge der stückweise Konstanten Funktionen betrachtet werden. Im Vergleich zu den Kubischen Hermite-Interpolanten sind die Kubischen-Splines (für $d = 3$) *zweimal* Ableitbar statt nur *einmal*

Bemerkung 4.3.3: $\dim(\mathcal{S}_{d,\mathcal{G}}) = N + d$. Es werden oft kubische Splines in Anwendungen verwendet, also ist $\dim(\mathcal{S}_{d,\mathcal{G}}) = N + 3$, wir haben aber nur $N + 1$ Funktionswerte, also beleiben noch zwei Freiheitsgrade übrig.

Dies bedeutet, dass wir ein underdeterminiertes lineares Gleichungssystem haben für $h_j = x_j - x_{j-1}$:

$$\begin{bmatrix} b_0 & a_1 & b_1 & 0 & \dots & & \dots & 0 \\ 0 & b_1 & a_2 & b_2 & & & & \\ & 0 & \ddots & \ddots & \ddots & & & \vdots \\ \vdots & & & \ddots & \ddots & \ddots & & \\ & & & & \ddots & a_{N-2} & b_{N-2} & 0 \\ 0 & \dots & & \dots & 0 & b_{N-2} & a_{N-1} & b_{N-1} \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ \\ \vdots \\ \\ c_{N-1} \\ c_N \end{bmatrix} = \begin{bmatrix} 3\left(\frac{y_1 - y_0}{h_1^2} + \frac{y_2 - y_1}{h_2^2}\right) \\ \\ \vdots \\ \\ 3\left(\frac{y_{N-1} - y_{N-2}}{h_{N-1}^2} + \frac{y_N - y_{N-1}}{h_N^2}\right) \end{bmatrix}$$

Wobei im Resultatvektor Einträge der Form

$$3\left(\frac{y_j - y_{j-1}}{h_j^2} + \frac{y_{j+1} - y_j}{h_{j+1}^2}\right)$$

enthalten sind und mit $a_j := \frac{2}{h_j} + \frac{2}{h_{j+1}}$ und $b_j := \frac{1}{h_{j+1}}$ für $j = 0, 1, \dots, N-1$

Wir müssen also zwei weitere Gleichungen finden (oder zwei Freiheitsgrade eliminieren).

Definition 4.3.4: *(Vollständige kubische Spline-Interpolation)* Falls wir die zusätzlichen Bedingungen $s'(x_0) = c_0$ und $s'(x_N) = c_N$ mit gegebenen c_0 und c_N haben. Sie ist auch bekannt als *clamped cubic spline*. In der obigen Matrix können dann die erste und letzte Spalte weggelassen werden.

Definition 4.3.5: *(Natürliche kubische Spline-Interpolation)* Falls wir die zusätzlichen Bedingungen $s''(x_0) = 0$ und $s''(x_N) = 0$ haben. Dann fügen wir obigem SLE zwei Zeilen hinzu (1. und $(N+1)$-te), die $2, 1, 0, 0, \dots = \frac{y_1 - y_0}{h_1}$ und $0, \dots, 0, 1, 2 = \frac{y_N - y_{N-1}}{h_N}$. Die Matrix ist nun also positive-definite und symmetrisch

Definition 4.3.6: *(Periodische kubische Spline-Interpolation)* Falls wir die zusätzlichen Bedingungen $s'(x_0) = s'(x_N)$ und $s''(x_0) = s''(x_N)$ haben. Dies macht nur Sinn, wenn $y_0 = y_N$, also nehmen wir das an und wir haben eine Spalte weniger und eine Reihe mehr, also ist die Systemmatrix rechts

$$A := \begin{bmatrix} a_1 & b_1 & 0 & \dots & 0 & b_0 \\ b_1 & a_2 & b_2 & & & 0 \\ 0 & \ddots & \ddots & \ddots & & \vdots \\ 0 & & & \ddots & a_{N-1} & b_{N-1} \\ b_0 & 0 & \dots & 0 & b_{N-1} & a_0 \end{bmatrix}$$

Bemerkung 4.3.7: Die SLE können in $\mathcal{O}(n)$ gelöst werden.

Bemerkung 4.3.8: Mit der "not-a-knot"-Bedingung s''' ist stetig in x_1 und x_{N-1} braucht man mindestens 4 Knoten. Da wir kubische Splines betrachten erzwing die Bedingung dass ein Polynom nur in den ersten beiden und ein anderes in den letzten beiden Subintervallen erscheint, also gilt $s_1 = s_2$ und $s_{N-1} = s_N$

Bemerkung 4.3.9: Der natürliche Spline minimiert die Gesamtkrümmung des Funktionsgraphen:

$$\int_a^b |s''(x)|^2 \, \mathrm{d}x \leq \int_a^b |g''(x)|^2 \, \mathrm{d}x$$

für alle Funktionen zweimal stetig differenzierbaren Funktionen g, für welche $g(x_j) = y_j$ gilt für jedes $j = 0, \ldots, N$

Interpolationsfehler vollständiger kubischer Splines — **Satz 4.3.10**

Wenn $f \in C^4[a,b]$ und s der vollständige kubische Spline-Interpolation von f auf einem äquidistantem Gitter mit Gitterweite h ist, dann ist der Fehler für $k = 0, 1, 2, 3$:

$$||f^{(k)} - s^{(k)}||_{L^\infty} \leq \frac{5}{384} h^{4-k} ||f^{(4)}||_{L^\infty}$$

In `numpy` verwendet `scipy.interpolate.CubicSpline` aktuell die "not-a-knot"-Bedingung. Es ist möglich mithilfe von `bc_type` beim Instanziieren der Klasse die Art des Splines zu ändern. Folgende (relevante) Optionen stehen laut Dokumentation zur Verfügung: `"not-a-knot"` (was der Default ist), `"periodic"`, `"clamped"` und `"natural"`

Auf Seite 114-115 im Skript finden sich einige Abbildungen zur Konvergenz der verschiedenen Varianten des `CubicSplices`

5 Quadratur