

Numerical Methods for Computer Science

Robin Bacher, Janis Hutz

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

23. Dezember 2025

TITLE PAGE COMING SOON

*“Wer in Python Type annotation benötigt, der soll kein Python verwenden”
(2025-10-09T10:43) “Wenn ich keine Lust habe, das zu berechnen, dann wende ich
einfach Gewalt an”*

- Vasile Gradinaru, 2025

HS2025, ETHZ

Summary of the Script and Lectures

Inhaltsverzeichnis

0	Introduction	4
1	Einführung	5
1.1	Rundungsfehler	5
1.2	Rechenaufwand	7
1.3	Rechnen mit Matrizen	8
2	Polynominterpolation	10
2.1	Interpolation und Polynome	10
2.1.1	Monombasis	10
2.2	Newton Basis	11
2.2.1	Koeffizienten	11
2.2.2	Auswertung	12
2.2.3	Fehler	12
2.3	Lagrange- und Baryzentrische Interpolationsformeln	13
2.3.1	Fehler	15
2.4	Chebyshev Interpolation	16
2.4.1	Fehler	17
3	Trigonometrische Interpolation	19
3.1	Fourier-Reihen	19
3.2	Diskrete Fourier Transformation	21
3.2.1	Motivation	21
3.2.2	Konstruktion	21
3.2.3	DFT in Numpy	22
3.2.4	DFT & Lineare Algebra	23
3.3	Schnelle Fourier Transformation	24
3.4	Trigonometrische Interpolation	25
3.4.1	Von Approximation zur Interpolation	25
3.4.2	Zero-Padding-Auswertung	26
3.5	Fehlerabschätzungen	27
3.6	DFT und Chebyshev-Interpolation	30
4	Stückweise Polynomiale Interpolation	31
4.1	Stückweise Lineare Interpolation	31
4.2	Kubische Hermite-Interpolation	31
4.3	Splines	32
5	Numerische Quadratur	34
5.3	Grundbegriffe und -Ideen	34
5.4	Äquidistante Punkte	36
5.5	Nicht äquidistante Stützstellen	37
5.5.1	Gauss Quadratur	37
5.5.2	Clenshaw-Curtis Quadraturformel	38
5.6	Adaptive Quadratur	39
5.7	Quadratur in \mathbb{R}^d und dünne Gitter	39
5.8	Monte-Carlo Quadratur	40
5.9	Methoden zur Reduktion der Varianz	42
5.9.1	Control Variates	42
5.9.2	Importance Sampling	42
5.9.3	Quasi-Monte-Carlo	42
6	Nullstellensuche	43
6.1	Iterative Verfahren	43
6.2	Abbruchkriterien	44
6.3	Fixpunktiteration	44
6.4	Intervallhalbierungsverfahren	45

6.5	Newtonverfahren in 1D	45
6.6	Sekantenverfahren	45
6.7	Newton-Verfahren in n Dimensionen	45
6.8	Gedämpftes Newton-Verfahren	46
6.9	Quasi-Newton-Verfahren	46
7	Intermezzo: Lineare Algebra	47
8	Ausgleichsrechnung	48
8.1	Lineare Ausgleichsrechnung	48
8.1.1	Normalengleichung	48
8.1.2	Lösung mittels orthogonaler Transformation	48
8.1.3	Totale Ausgleichsrechnung	49
8.2	Nichtlineare Ausgleichsrechnung	50
8.2.1	Newton-Verfahren	50
8.2.2	Gauss-Newton Verfahren	50
8.2.3	Weitere Methoden: BFGS, GD, SGC, CG, LM, ADAM	52

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.

We have also taken some explanations and code examples from the slides of our TA, Nils Müller, whose slides can be found [here](#)

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

▪ **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 1.3.8: $D @ A$ 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 . **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:

```

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 $B \in \mathbb{R}^{l \times k}$.

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:

```

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 2.1.1: 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 2.1.4: (Peano) f stetig $\implies \exists p(x)$ welches f in $\|\cdot\|_\infty$ beliebig gut approximiert.

Definition 2.1.5: (Raum der Polynome) $\mathcal{P}_k := \{x \mapsto \sum_{j=0}^k \alpha_j x^j\}$ **Definition 2.1.6:** (Monom) $f : x \mapsto x^k$

Satz 2.1.7: (Eigenschaft von \mathcal{P}_k) \mathcal{P}_k ist ein Vektorraum mit $\dim(\mathcal{P}_k) = k+1$.

2.1.1 Monombasis

Satz 2.1.8: (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 2.1.9: (Horner Schema) $p(x) = (x \dots x(\alpha_n x + \alpha_{n-1}) + \dots + \alpha_1) + \alpha_0$

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.

$$p_0(x) = y_0 \quad (\text{Anfang: triviales Polynom})$$

$$p_1(x) = p_0(x) + (x - x_0) \frac{(y_1 - y_0)}{(x_1 - x_0)} \quad (\text{Addition des zweiten Datenpunktes})$$

$$p_2(x) = p_1(x) + \frac{\frac{(y_2 - y_1)}{(x_2 - x_1)} - \frac{(y_1 - y_0)}{(x_1 - x_0)}}{x_2 - x_0} (x - x_0)(x - x_1) \quad (\text{Schema lässt sich beliebig weiterführen})$$

$$p_3(x) = p_2(x) + \dots$$

Satz 2.2.2: (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 2.2.4: (Dividierte Differenzen)

$$y[x_i] := y_i$$

$$y[x_i, \dots, x_{i+k}] \stackrel{\text{Rec.}}{:=} \frac{y[x_{i+1}, \dots, x_{i+k}] - y[x_i, \dots, x_{i+k-1}]}{x_{i+k} - x_i}$$

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

Bemerkung 2.2.5: (Äquidistante Stellen)

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

$$y[x_0, x_1] = \frac{1}{h} \Delta y_0$$

$$y[x_0, x_1, x_2] = \frac{1}{2!h} \Delta^2 y_0$$

$$y[x_0, \dots, x_n] = \frac{1}{n!h^n} \Delta^n y_0$$

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 $\mathcal{O}(n^2)$

```

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 2.2.10: 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 2.2.11: (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:

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

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.zeros(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 das Interpolationspolynom $p(x)$ mit Hilfe einer Rückwärtsrekursion berechnen.

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

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

```
1 def clenshaw(coeffs: np.ndarray, x: np.ndarray):
2     n = len(coeffs) - 1
3     # initialise temporary variables
4     d_prev_prev, d_prev, d_curr = (
5         np.zeros_like(x),
6         np.zeros_like(x),
7         np.zeros_like(x),
8     )
9
10    for k in range(n, -1, -1): # backward recursion
11        d_curr = coeffs[k] + 2 * x * d_prev - d_prev_prev
12        d_prev_prev, d_prev = d_prev, d_curr
13
14    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:

```
1 def get_cheb_coeffs(abscissa: np.ndarray)
2     n = len(abscissa) - 1
3     dct_vals = scipy.fft.dct(abscissa, type=1)
4
5     coeffs = dct_vals / n
6     coeffs[0] /= 2
7     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: $p_m : \mathbb{R} \rightarrow \mathbb{C}$ 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

Definition 3.1.3

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 (= Euklidische Norm, also die normale Vektornorm) auf $(0, 1)$ durch das Skalarprodukt

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

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

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

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 $\widehat{f'}(k) = 2\pi i k \hat{f}(k)$ für $k \in \mathbb{Z}$.

Falls die Operationen erlaubt sind, dann gilt zudem:

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

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 3.1.18 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:

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

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

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

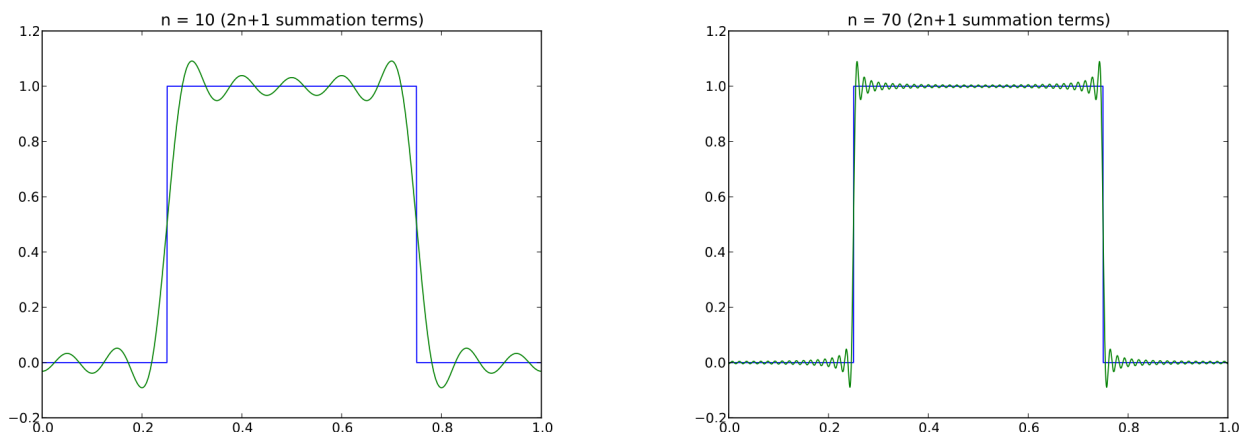


Abbildung 3.1.18: Überschwingungen der Fourier-Reihe der charakteristischen Funktion des Intervalls $[a, b] \subseteq]0, 1[$. (Abbildung aus dem Vorlesungsdokument von Prof. V. Gradinaru, Seite 69)

3.2 Diskrete Fourier Transformation

3.2.1 Motivation

Nutzen wir die Trapezregel um approximativ die Fourierkoeffizienten $\hat{f}_N(k)$ 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:

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

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)

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

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)

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

Die symmetrische, nicht hermitesche Matrix $V = [v_0, \dots, v_{N-1}]$ 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 entspricht 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.

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

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

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

Definition 3.2.4: (Fourier-Matrix)

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

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) $\mathcal{F}_N : \mathbb{C} \rightarrow \mathbb{C}$ s.d. $\mathcal{F}_N(y) = F_N y$

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

c lässt sich als Repräsentation von y im Frequenzbereich interpretieren. 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.

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

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

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

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:

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

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

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

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 $p(z) = c_0 z^0 + \dots + c_{N-1} z^{N-1}$.

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

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

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

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: $(g * h)(x) = \frac{1}{T} \int_0^T g(t)h(x-t)dt$.

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

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

Man erhält so letztendlich das Faltungs-Theorem: Die F_N -Transformierte einer Faltung ist genau das gleiche wie die Multiplikation zweier F_N -Transformierten. 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 $\mathcal{O}(N^2)$ 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 3.3.3 findet sich, zusammen mit dem Code, mit der sie produziert wurde im Skript auf Seite 86-88

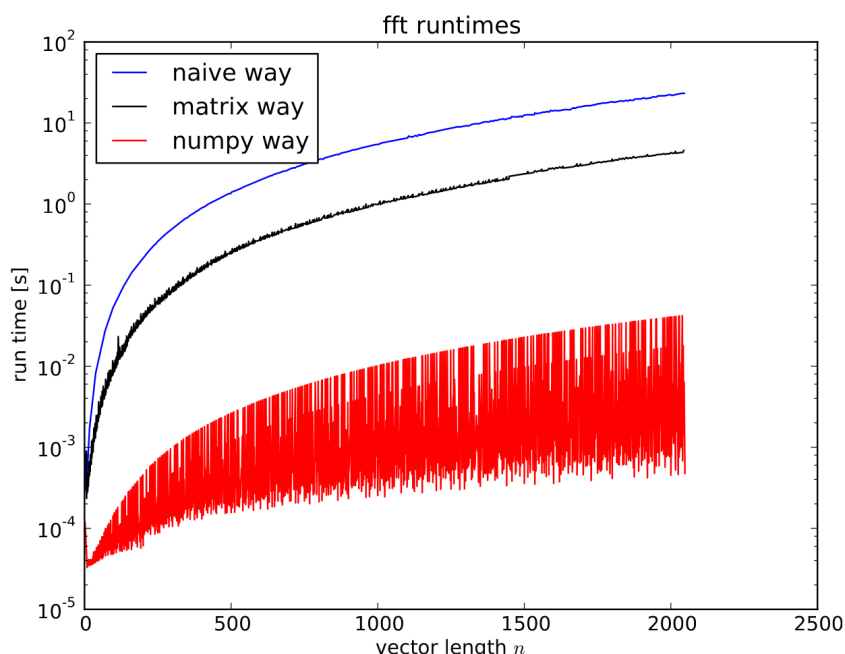


Abbildung 3.3.3: Vergleich der Laufzeit von verschiedenen Fourier-Transformations-Algorithmen. (Abbildung 3.3.3 aus dem Vorlesungsdokument von Prof. V. Gradinaru, Seite 88)

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. $\mathcal{O}(N^2)$, 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.

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 $\gamma_k = \hat{f}_N(k)$ und für $k < 0$ ist $\gamma_k = \hat{f}_N(N+k)$)

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

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

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 $p_{N-1}\left(\frac{l}{N}\right) = f\left(\frac{l}{N}\right)$ für $l = 0, 1, \dots, N-1$ 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: $\mathcal{O}(N^3)$
- (2) Mittels FFT in $\mathcal{O}(N \log(N))$ 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.

```

1 def get_coeff_trig_poly(t: np.ndarray, y: np.ndarray):
2     N = y.shape[0]
3     if N % 2 == 1:
4         n = (N - 1.0) / 2.0
5         M = np.exp(2 * np.pi * 1j * np.outer(t, np.arange(-n, n + 1)))
6     else:
7         n = N / 2.0
8         M = np.exp(2 * np.pi * 1j * np.outer(t, np.arange(-n, n)))
9     c = np.linalg.solve(M, y)
10    return c
11
12 N = 2**12
13 t = np.linspace(0, 1, N, endpoint=False)
14 y = np.random.rand(N)
15 direct = get_coeff_trig_poly(t, y)
16 using_fft = np.fft.fftshift(np.fft.fft(y) / N)
17 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.

```

1 def evaluate_trigonometric_interpolation_polynomial(y: np.ndarray, N: int):
2     n = len(y)
3     if (n % 2) == 0:
4         c = np.fft.ifft(y) # Fourier coefficients
5         a = np.zeros(N, dtype=complex)
6
7         # Zero padding
8         a[: n // 2] = c[: n // 2]
9         a[N - n // 2 :] = c[n // 2 :]
10        return np.fft.fft(a)
11    else:
12        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 $E(n) = \mathcal{O}(q^n)$ mit $0 \leq q < 1$

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

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

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

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

Die untenstehende Abbildung 3.5.2 beinhaltet einen Plot, auf dem die Konvergenz in Abhängigkeit des Grades des Interpolationspolynoms aufgetragen ist.

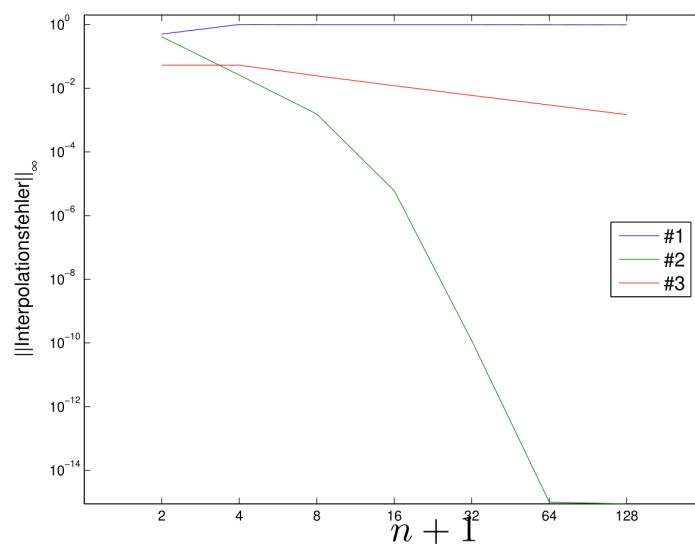


Abbildung 3.5.2: Interpolierungsfehler der Beispiele. Algebraische Konvergenz für (I) und (III), exponentielle für (II). (Abbildung 3.5.2 aus dem Vorlesungsdokument von Prof. V. Gradinaru, Seite 96)

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 $\alpha \in [0, 1)$ $f(t) = \frac{1}{\sqrt{1 - \alpha \sin(2\pi t)}}$. Die Konvergenz ist exponentiell in n und je kleiner α , desto schneller ist sie. In der untenstehenden Abbildung 3.5.5 sind einige Beispiele aufgetragen:

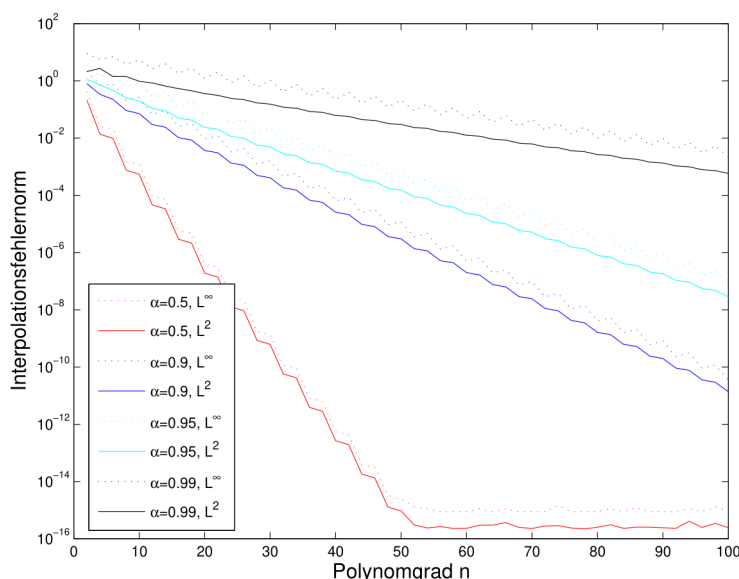


Abbildung 3.5.5: Fehler bei der trigonometrischen Interpolation. (Abbildung 3.5.5 aus dem Vorlesungsdokument von Prof. V. Gradinaru, Seite 98)

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:

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

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

Das heisst also, dass die Fourier-Koeffizienten von f bei kleinen Frequenzen (hier $|k| < \frac{N}{2}$) 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 $\sum_{k \in \mathbb{Z}} |\hat{f}(k)|$ absolut konvergiert, dann ist der Approximationsfehler definiert als:

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

Da durch diesen Satz die obere Schranke für den Approximationsfehler durch die schwer approximierbaren Fourier-Koeffizienten $\hat{f}(k)$ 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 $\hat{f}(k) = 0 \quad \forall |k| > m$. Falls $N > 2m$, dann gilt $p_N(x) = f(x) \quad \forall x$

Beispiel: 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 3.5.10 zu sehen: Für unser

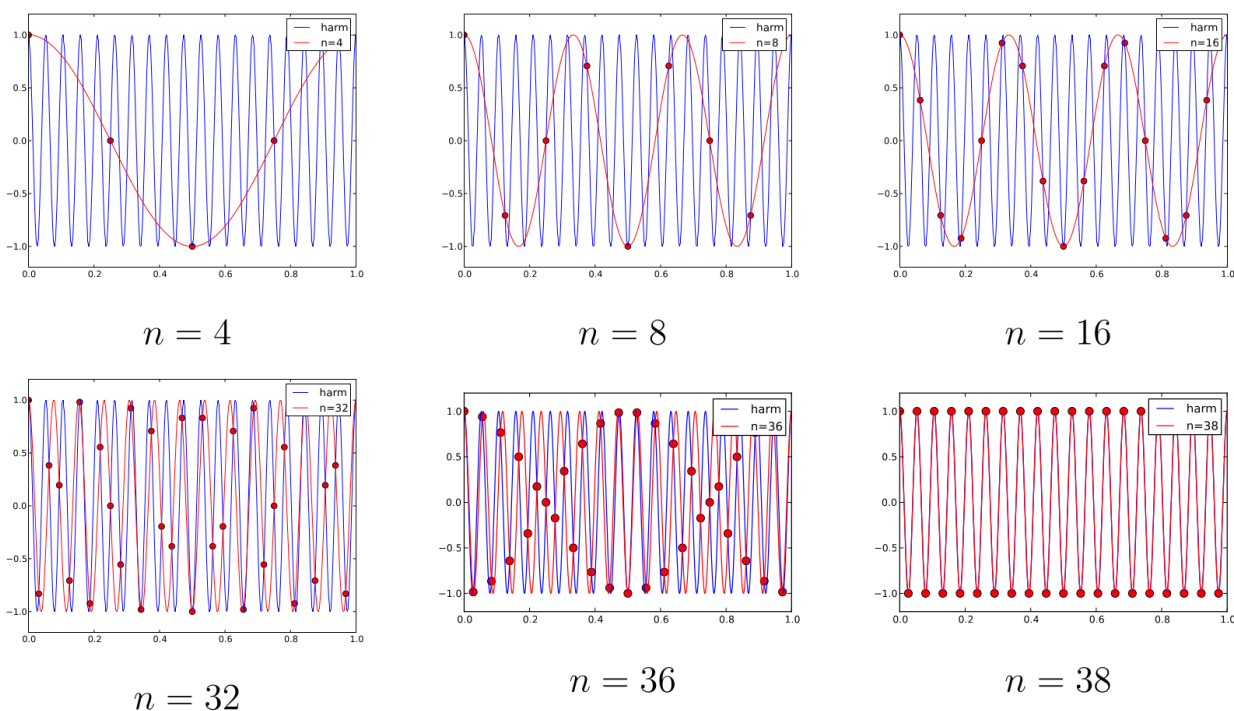


Abbildung 3.5.10: Aliasing für $f(t) = \cos(2\pi \cdot 19t)$. (Abbildung 3.5.10 aus dem Vorlesungsdokument von Prof. V. Gradinaru, Seite 100)

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

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

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 $g: [-1, 1] \rightarrow \mathbb{C}, s \mapsto f(\cos(2\pi s))$ und $q: [-1, 1] \rightarrow \mathbb{C}, s \mapsto p(\cos(2\pi s))$, 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 $s^* = s + \frac{1}{4n+4}$ 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:

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

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

Und mit weiteren 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 Messungen 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:

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

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* genannt. 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 Paweise 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 $\chi(t) = t \quad \forall t \in [0, 1]$. 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 Variablen 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 "formerhaltender" 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 höchstens 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 bleiben noch zwei Freiheitsgrade übrig.

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

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

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

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

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 erzwingt 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:

$$\int_a^b |s''(x)|^2 dx \leq \int_a^b |g''(x)|^2 dx$$

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

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 Instanzieren 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 CubicSplines.

5 Numerische Quadratur

5.3 Grundbegriffe und -Ideen

Es ist oft nicht möglich oder sinnvoll ein Integral analytisch zu berechnen. Mit Methoden der Quadratur können wir Integrale numerisch berechnen.

In `numpy` kann `scipy.integrate.quad` verwendet werden. Falls man jedoch eine manuelle Implementation erstellen will, so nutzt man oft die Trapez- oder Simpson-Regel, da sie sowohl einfach zu implementieren, wie auch effizient sind. In gewissen Anwendungen sind Gauss-Quadratur-Formeln nützlich, welche man durch Spektralmethoden ersetzen kann, welche die FFT verwenden und effizienter sind.

Quadratur

Ein Integral kann durch eine gewichtete Summe von Funktionswerten der Funktion f an verschiedenen Stellen c_i^n approximiert werden:

$$\int_a^b f(x) \, dx \approx Q_n(f; a, b) := \sum_{i=1}^n \omega_i^n f(c_i^n)$$

wobei ω_i^n die Gewichte und $c_i^n \in [a, b]$ die Knoten der Quadraturformel sind.

Definition 5.3.1

Wir wollen natürlich wieder $c_i^n \in [a, b]$ und w_i^n so wählen, dass der Fehler minimiert wird.

Fehler

Der Fehler der Quadratur $Q_n(f)$ ist

$$E(n) = \left| \int_a^b f(x) \, dx - Q_n(f; a, b) \right|$$

Wir haben **algebraische Konvergenz** wenn $E(n) = \mathcal{O}\left(\frac{1}{n^p}\right)$ mit $p > 0$ und **exponentielle Konvergenz** wenn $E(n) = \mathcal{O}(q^n)$ mit $0 \leq q < 1$

Definition 5.3.2

Die Idee, den Integral einer schweren Funktion zu berechnen, ist diese mit einer einfachen Funktion, die analytisch integrierbar ist, zu approximieren. Wenn wir diese Funktion geschickt wählen, dann ist es sogar möglich, dass wir nur eine solche Funktion für alle Funktionen f benötigen.

Wir ersetzen also f durch $f_n \in \text{span}\{c_0, c_1, \dots, c_n\}$, wobei die c_i eine Basis des Raums der Funktionen auf $[a, b]$ bilden:

$$\int_a^b f(x) \, dx \approx \int_a^b f_n(x) \, dx = \int_a^b \left(\sum_{k=0}^n \alpha_k c_k(x) \right) \, dx = \sum_{k=0}^n \alpha_k \int_a^b c_k(x) \, dx$$

Falls wir $c_k(x) = x^k$ haben (was oft der Fall ist, je nach Funktion aber könnte eine rationale Funktion oder andere Arten besser geeignet sein), dann erhalten wir:

$$\int_a^b c_k(x) \, dx = \frac{b^{k+1} - a^{k+1}}{k+1}$$

Lagrange-Polynome

Für die Knoten $x_0, x_1, \dots, x_n \in \mathbb{R}$ definieren wir die Polynome

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

als die Lagrange-Polynome zu den Stützstellen x_0, x_1, \dots, x_n

Definition 5.3.3

Für ein Beispiel verweisen wir auf Beispiel 2.3.2

Bemerkung 5.3.6: (*Eigenschaften der Lagrange-Polynome*) Zu den Eigenschaften aus 2.3.4 fügen wir an (die Eigenschaften aus Bemerkung 2.3.4 sind hier erneut aufgeführt)

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. $\sum_{k=0}^n l_k(x) = 1 \quad \forall x \in \mathbb{R}$
 5. $\sum_{k=0}^n l_k^{(m)}(x) = 0$ für $m > 0$
 6. l_0, l_1, \dots, l_n bilden Basis von \mathcal{P}_{n+1}
- wobei \mathcal{P}_{n+1} der Raum der Polynome von Grad maximal n ist.

Bemerkung 5.3.7: (*Quadraturgewichte aus den Lagrange-Polynomen*) Das Interpolationspolynom ist gegeben durch:

$$p(x) = \sum_{j=0}^n f(x_j) l_j(x)$$

Durch die Eigenschaften der Lagrange-Polynome haben wir $p(x_j) = f(x_j)$ und die Konstruktion von $p(x)$ ist eindeutig in \mathcal{P}_{n+1} . Wir erhalten nun eine Quadraturformel, wenn wir p als Approximation von f verwenden:

$$w_j = \int_a^b l_j(x) dx, \quad j = 0, 1, \dots, n$$

Diese Gewichte werden für die Trapez- und Simpson-Regeln verwendet, genau genommen, im Falle der Trapezregel haben wir w_2 und für die Simpsonregel w_3 , also müssen wir die entsprechenden Lagrange-Polynome integrieren. Durch die Konstruktion der Formel ist sie exakt für alle Polynome aus \mathcal{P}_{n+1} und der Fehler ist:

$$\left| \int_a^b f(x) dx - \int_a^b p_n(x) dx \right| \leq \frac{1}{n!} (b-a)^{n+1} \max |f^{(n)}(z)|$$

Wir wollen also ein kleines Intervall (oft $b-a < 1$ da wir so das Integral besser approximieren können) und wir setzen voraus, dass f glatt ist.

Da wir aber oft ein grösseres Intervall betrachten möchten, ist ein möglicher Ansatz, das grosse Intervall in kleinere Intervalle zu zerlegen. Wir nehmen ein äquidistantes Gitter, mit $x_k = x_0 + k \cdot h$ für $h = \frac{b-a}{N}$ und $k = 0, \dots, N$:

$$\int_a^b f(x) dx = \sum_{k=0}^{N-1} \int_{x_k}^{x_{k+1}} f(x) dx$$

Die obige Formel wird auch die *summierte* Quadraturformel genannt. Der Fehler ist dann also:

$$\left| \int_a^b f(x) dx - \sum_{k=0}^{N-1} Q(f, x_k, x_{k+1}) \right| \leq \dots \leq C \frac{h^n}{n!} (b-a) \quad \text{mit } C = \max_{z \in [a,b]} |f^{(n)}(z)| = \|f^{(n)}\|_{\max}$$

Der obige Ansatz ist gewissermassen “divide and conquer” (zu Deutsch: “Teile und Herrsche”, wir werden aber DnC verwenden) und wir der lokale Fehler liegt in $\mathcal{O}(h^{n+1})$ und mit $N = (b-a) \div h$ Intervallen der Grösse h haben wir einen globalen Fehler in $\mathcal{O}(h^n)$. Folglich ist also der Fehler kleiner, je kleiner h ist.

Wir benutzen erneut einen Variablenwechsel, um von einem Referenzintervall $[-1, 1]$ auf eines unserer Teilintervalle $[x_k, x_{k+1}]$ zu wechseln. Dies heisst also allgemein für Intervall $[a, b]$ nach $[-1, 1]$:

$$\int_a^b f(t) dt = \frac{1}{2}(b-a) \int_{-1}^1 \hat{f}(\tau) d\tau \quad \text{mit } \hat{f}(\tau) := f\left(\frac{1}{2}(1-\tau)a + \frac{1}{2}(\tau+1)b\right)$$

Für dieses Referenzintervall können wir die Gewichte \hat{w}_j und die Knoten \hat{c}_j bestimmen.

$$\int_a^b f(t) dt \approx \frac{1}{2}(b-a) \sum_{j=1}^n \hat{w}_j \hat{f}(\hat{c}_j) = \sum_{j=1}^n w_j f(c_j) \quad \text{mit } \begin{aligned} c_j &= \frac{1}{2}(1-\hat{c}_j)a + \frac{1}{2}(1+\hat{c}_j)b \\ w_j &= \frac{1}{2}(b-a)\hat{w}_j \end{aligned}$$

Definition 5.3.8: Die Ordnung einer Quadraturformel ist n wenn sie Polynome vom Grad $(n-1)$ exakt integriert.

Dies folgt natürlich direkt davon, dass wir ein Polynom n -ten Grades mit $n+1$ Koeffizienten darstellen können.

Definition 5.3.9: (*Symmetrie*) Eine Quadraturformel auf $[-1, 1]$ heisst symmetrisch, falls $\omega_i = \omega_{n+1-i}$ und $c_i = -c_{n+1-i}$ gilt für die Gewichte ω_i und Knoten c_i

Bemerkung 5.3.10: Die Mittelpunkts-, Trapez- und Simpson-Regeln aus Abschnitt 5.4 sind symmetrisch

Satz 5.3.11: Die Ordnung einer symmetrischen Quadraturformel ist gerade

Beweis: Kann mittels Induktion bewiesen werden, siehe dazu Seite 123 im Skript

5.4 Äquidistante Punkte

5.5 Nicht äquidistante Stützstellen

Alternativ zur Unterteilung des Intervalls können wir andere Quadraturpunkte erlauben

5.5.1 Gauss Quadratur

In diesem Kapitel werden die Gewichte mit b_1, b_2, \dots, b_s und die Knoten auf unserem Referenzintervall, welches hier $[0, 1]$ ist, mit $c_1, c_2, \dots, c_s \in [0, 1]$ bezeichnet.

Wir möchten unsere Gewichte b_i und Knoten c_i so bestimmen, dass die Quadraturordnung maximal ist.

Wir definieren die Notation $\langle M, g \rangle = \int_0^1 M(t)g(t) dt$ (also das Skalarprodukt).

Ordnung der Quadraturformel

Satz 5.5.1

Die Ordnung ist $s + m$ genau dann, wenn $\langle M, g \rangle = 0$ für alle Polynome g mit $\deg(g) \leq m - 1$ und $M(t) = (t - c_1) \cdot (t - c_2) \cdot \dots \cdot (t - c_s)$ für s . Also steht M senkrecht zu allen g .

Satz 5.5.2: (Maximale Ordnung einer Quadraturformel) Die Ordnung einer Quadraturformel mit s Knoten ist $\leq 2s$

Orthogonale Polynome

Für $I =]a, b[$ sei $w : I \rightarrow \mathbb{R}$ eine stetige Gewichtsfunktion mit $w(x) > 0 \ \forall x \in I$, so dass für alle $k = 0, 1, 2, \dots$ $\int_a^b |x|^k w(x) dx$ existiert.

Orthogonale Polynome

Satz 5.5.3

Im Raum $V = \{f : I \rightarrow \mathbb{R} \text{ stetig, } \int_a^b |f(x)|^2 w(x) dx \text{ existiert}\}$ existiert eine eindeutige Folge von Polynomen p_0, p_1, \dots mit $p_k(x) = x^k + P(x)$ mit $\deg(P(x)) \leq k - 1$ für $k \geq 0$, so dass $p_k \perp \text{span}\{p_0, p_1, \dots, p_{k-1}\}$. Sie können mit der **3-Term-Rekursion** gebaut werden:

$$p_{k+1}(x) = (x - \beta_{k+1}) \cdot p_k(x) - \gamma_k p_{k-1}(x)$$

mit $p_0(x) = 1$, $p_{-1}(x) = 0$, $\beta_{k+1} = \frac{\langle x \cdot p_k, p_k \rangle}{\langle p_k, p_k \rangle}$ und $\gamma_{k+1} = \frac{\langle p_k, p_k \rangle}{\langle p_{k-1}, p_{k-1} \rangle}$, wobei hier $\langle f, g \rangle = \int_a^b f(x)g(x)w(x) dx$ das Skalarprodukt ist.

Beispiel 5.5.4: (Legendre-Polynome) sind definiert für $w(x) = 1$, $a = -1$ und $b = 1$ (sie sind orthogonal):

$$\begin{aligned} p_0(x) &= 1 & p_1(x) &= x \\ p_2(x) &= \frac{1}{2}(3x^2 - 1) & p_3(x) &= \frac{1}{2}(5x^3 - 3x) \end{aligned}$$

Die Normierung der Legendre-Polynome ist nicht standardisiert

In `numpy` können wir mit `scipy.special.eval_legendre` und `scipy.special.legendre` diese Polynome berechnen und mit `scipy.special.roots_legendre` die Knoten berechnen

Beispiel 5.5.5: (Hermite-Polynome) sind definiert für $w(x) = e^{-x^2}$, $a = -\infty$ und $b = \infty$:

$$\begin{aligned} p_0(x) &= 1 & p_1(x) &= 2x \\ p_2(x) &= 4x^2 - 2 & p_3(x) &= 8x^3 - 12x \end{aligned}$$

Bemerkung 5.5.7: Aus Theorem 5.5.3 folgt direkt, dass c_1, c_2, \dots, c_s die Nullstellen von p_s sind.

Bemerkung 5.5.11: (Knoten und Fehler der Gauss-Quadratur)

- Gauss-Knoten sind nicht äquidistant.
- Gauss-Knoten sind nicht verschachtelt (was er damit meint ist, dass wir sie nicht mit DnQ verwenden können —Wir können also nicht für eine Quadratur höherer Ordnung die Knotenpunkte der Gauss-Quadratur tieferer Ordnung verwenden)

- Die Gauss-Quadratur ist offen (da die Endpunkte des Intervalls keine Knoten sind)
- Bei der **Radau-Quadratur** fixiert man ein Ende als Randknoten, und man hat nun Ordnung $2s-1$. Die Berechnung ist ansonsten gleich, bis auf den Fakt, dass wir nur noch $(s-1)$ Knoten haben (1 bis und mit $s-1$). Sie können mit `scipy.special.roots_jacobi(s-1, alpha=1, beta=0)` berechnet werden.
- Bei der **Lobatto-Quadratur** fixiert man gleich beide Enden als Randknoten, und man hat Ordnung $2s-2$ und wir haben die Knoten c_2, \dots, c_{s-1}
- Die Lobatto- und Radau-Quadratur werden häufig bei der Lösung gewöhnlicher DGL verwendet.

Der Fehler der Gauss-Quadratur ist:

$$\int_a^b f(x) dx - \sum_{j=1}^s b_j \cdot f(c_j) = \frac{b-a}{(2s)!} f^{(2s)}(z) \text{ mit } z \in [a, b]$$

Und eine obere Schranke für den Fehler ist dann

$$\left| \int_a^b f(x) dx - \sum_{k=1}^N G_s(f, x_{k-1}, x_k) \right| \leq c \cdot h^{2s} \max_{z \in [a, b]} |f^{(2s)}(z)|$$

wobei $c \in \mathbb{R}$ eine Konstante ist und $h = b - a$ die Grösse des Intervalls ist.

Bemerkung 5.5.14: (Gewichte der Gauss-Legendre-Quadratur) Für die Knoten c_1, \dots, c_s und den entsprechenden Lagrange-Polynomen l_1, \dots, l_s mit $\deg(l_i) = s-1 \forall i \in \{1, \dots, s\}$. Die zugehörige Quadraturformel ist exakt für Polynome $2s-1$ -ten Grades. Die Gewichte sind:

$$b_i = \int_0^1 l_i(t)^2 dt$$

Satz 5.5.15: Die Gewichte der Gauss-Legendre-Quadraturformel sind positiv.

Algorithmus	Laufzeit	Genauigkeit Knoten	Genauigkeit Gewichte
GW (1969)	$\mathcal{O}(s^3) / \mathcal{O}(s^2)$	$\mathcal{O}(1)$	$\mathcal{O}(s^2)$
Bogaert-Townsend	$\mathcal{O}(s)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
CC (2s Knoten)	$\mathcal{O}(s \log(s))$	$\mathcal{O}(1)$	$\mathcal{O}(1)$

Die Gauss-Quadratur ist in der Messtechnik nicht besonders geeignet, da wir die zugrundeliegende Funktion nicht im Vorhinein kennen und die Kosten für die Anpassung der Ordnung aufgrund fehlender Verschachtelbarkeit sehr hoch sind (wir müssen alle vorigen Berechnungen komplett neu machen)

5.5.2 Clenshaw-Curtis Quadraturformel

Die erste Quadraturformel von Fejér benutzt die Chebyshev-Knoten (Nullstellen der Chebyshev-Polynome erster Art), welche aber nicht verschachtelt sind. Die zweite Quadraturformel von Fejér benutzt die Filippi-Knoten $x_k = \cos(k \frac{\pi}{n})$ für $k = 1, \dots, n-1$ und Clenshaw und Curtis haben dann zusätzlich noch die Endknoten hinzugefügt (also $k = 0, \dots, n$). Die Clenshaw-Curtis-Knoten sind die Chebyshev-Abszissen und die Formel verhält sich mit den entsprechenden Gewichten ähnlich gleich wie die Gauss-Quadratur.

Da die Clenshaw-Curtis-Quadratur mithilfe der DFT berechnet werden kann ist sie sehr effizient. Dazu müssen wir aber zuerst etwas umformen, mit $x = \cos(\theta)$, so dass das Integral eine periodische Funktion wird:

$$\int_{-1}^1 f(x) dx = \int_0^\pi f(\cos(\theta)) \sin(\theta) d\theta = f(\cos(\theta))$$

$F(\theta)$ ist 2π -periodisch und gerade, kann sich also in eine Kosinus-Reihe entwickeln, also: $F(\theta) = \sum_{k=0}^\infty a_k \cos(k\theta)$, woraus folgt, dass

$$\int_0^\pi F(\theta) \sin(\theta) d\theta = \dots = a_0 + \sum_{2 \leq k \text{ gerade}} \frac{2a_k}{1-k^2}$$

wobei sich die Koeffizienten a_k mit FFT oder DCT berechnen lassen

Eine wichtige Erkenntnis ist, dass die Newton-Cotes bei grösserer Ordnung komplett unbrauchbar werden, wie das in Abbildung 5.5.24 im Skript zu sehen ist, während die Clenshaw-Curtis-Quadratur ähnlich gut ist wie die Gauss-Quadratur (gleiche Konvergenzordnung).

Quadratur	Intervall	Gewichtsfunktion	Polynom	Notation	scipy.special.
Gauss	$(-1, 1)$	1	Legendre	P_k	roots_legendre
Chebyshev I	$(-1, 1)$	$\frac{1}{\sqrt{1-x^2}}$	Chebyshev I	T_k	roots_chebyt
Chebyshev II	$(-1, 1)$	$\sqrt{1-x^2}$	Chebyshev II	U_k	roots_chebyu
Jacobi $\alpha, \beta > 1$	$(-1, 1)$	$(1-x)^\alpha(1+x)^\beta$	Jacobi	$P_k^{(\alpha, \beta)}$	roots_jacobi
Hermite	\mathbb{R}	e^{-x^2}	Hermite	H_k	roots_hermite
Laguerre	$(0, \infty)$	$x^\alpha e^{-x^2}$	Laguerre	L_k	roots_genlaguerre

Tabelle 5.5.16: Gewichtsfunktionen für Quadraturformeln

5.6 Adaptive Quadratur

Der lokale Fehler einer zusammengesetzten Quadraturformel auf dem Gitter $\mathcal{M} := \{a = x_0 < x_1 < \dots < x_m = b\}$ ist (für $f \in C^2([a, b])$):

$$\left| \int_{x_k}^{x_{k+1}} f(t) dt - \frac{f(x_k) + f(x_{k+1})}{2} (x_{k+1} - x_k) \right| \leq (x_{k+1} - x_k)^3 \|f''\|_{L^\infty([x_k, x_{k+1}])}$$

Also ist es nur sinnvoll, das Gitter zu verfeinern wo $|f''|$ gross ist.

Auf Seiten 150 - 151 im Skript findet sich Code, um eine adaptive Quadratur durchzuführen.

Bemerkung 5.6.3: (*Adaptive Quadratur in Python*) Mit `scipy.integrate.quad` können wir einfach eine adaptive Quadratur durchführen und benutzt QUADPACK. Mit `scipy.integrate.quadrature` können wir die Gauss-Quadratur verwenden.

Für $x \in \mathbb{R}^d$, also eine mehrdimensionale Funktion der Dimension d können wir `scipy.integrate.nquad` verwenden. Mehr dazu im nächsten Kapitel

5.7 Quadratur in \mathbb{R}^d und dünne Gitter

Eine einfache Option wäre natürlich, zwei eindimensionale Quadraturformeln aneinander zu hängen. Für zweidimensionale Funktionen sieht dies so aus:

$$I = \int_{j_1}^{n_1} \sum_{j_2}^{n_2} \omega_{j_1}^1 \omega_{j_2}^2 f(c_{j_1}^1, c_{j_2}^2)$$

und für beliebige d haben wir

$$(w_{j_k}^k, c_{j_k}^k)_{1 \leq j_k \leq n_k} \quad k = 1, \dots, d$$

Was dasselbe ist, wie oben, aber mit d Summen und d -mal ein w_{j_k} und eine d -dimensionale Funktion f

Tensor-Produkt

Repetition

TODO: Write this section

Die wichtigste Erkenntnis aus diesem Abschnitt ist die Idee, ein **Sparse-Grid** zu verwenden, um die Rechenarbeit zu reduzieren.

In numpy Gibt es die Möglichkeit Sparse-Grid arrays mit `scipy.sparse` zu erstellen.

5.8 Monte-Carlo Quadratur

Bei der Monte-Carlo Quadratur wird, wie bei anderen Monte-Carlo-Algorithmen der Zufall genutzt

Grundrezept Wir nehmen N Zahlen t_i die zufällig aus der uniformen Verteilung auf $[0, 1]$ gewählt werden.

$$I = \int_0^1 z(t) dt \approx \frac{1}{N} \sum_{i=1}^N z(t_i)$$

Auf einem anderen Intervall $[a, b]$ haben wir dann für $s_i = a + (b - a) \cdot t_i$

$$I = \int_a^b z(s) ds \approx |b - a| \frac{1}{N} \sum_{i=1}^N z(s_i) =: I_N$$

Bemerkung 5.8.1: Die Konvergenz ist sehr langsam (\sqrt{N}), aber nicht abhängig von der Dimension oder Glattheit. Zudem kann das Ergebnis falsch sein, da es probabilistisch ist.

Jede Monte-Carlo-Methode benötigt folgendes mit $X = [I_N - \tilde{\sigma}_N, I_N + \tilde{\sigma}_N]$ und X enthält den wahren Wert $\int_{[0,1]^d} z(t) dt$ in ungefähr 68.3% der Fälle für N t_i uniform Verteilt in $[0, 1]^d$:

- ein Gebiet für das "Experiment", hier $[0, 1]^d$
- gute deterministische Berechnungen, hier $\tilde{\sigma}_N$ und I_N
- gute Zufallszahlen
- Darstellung des Ergebnis, hier $\Pr[I \in X] = 0.683$

$$\text{Sei } \tilde{\sigma}_N = \sqrt{\frac{\frac{1}{N} \sum_{i=1}^N z(t_i)^2 - \left(\frac{1}{N} \sum_{i=1}^N z(t_i)\right)^2}{N - 1}} = \frac{\sigma_N}{\sqrt{N}}$$

Das Monte-Carlo-Verfahren beruht auf folgendem:

$$\int_{[0,1]^d} z(x) dx = \mathbb{E}z(\mathcal{X}) \text{ mit } \mathcal{X} \sim \mathcal{U}([0, 1]^d)$$

wobei $\mathcal{U}([0, 1]^d)$ die uniforme Verteilung der Zufallsvariable auf dem d -dimensionalen Intervall $[0, 1]^d$ ist.

Das Ziel der Monte-Carlo-Methode ist es, den Erwartungswert durch den Mittelwert der Funktionswerte der simulierten Zufallsvariable mit einem Schätzer $m_N(z(\mathcal{X}))$, bzw. einer Schätzung $m_N(z(x))$ zu approximieren:

$$m_N(z(\mathcal{X})) := \frac{1}{N} \sum_{i=1}^N z(\mathcal{X}_i) \qquad m_N(z(x)) := \frac{1}{N} \sum_{i=1}^N z(x_i)$$

Bemerkung 5.8.16: Wir verwenden $m_N(z(x))$ für das $z(x)$ im obigen Integral:

$$\mathbb{E}m_N(z(\mathcal{X})) = N \frac{1}{N} \mathbb{E}z(\mathcal{X}) = \int_{[0,1]^d} z(x) dx$$

Die Approximation von $\mathbb{E}z(\mathcal{X})$ durch $m_N(x)$ ist besser, je kleiner die Varianz ist:

$$\mathbb{V}m_N(z(\mathcal{X})) = \mathbb{V}\left(\frac{1}{N} \sum_{i=1}^N z(\mathcal{X}_i)\right) = \frac{1}{N^2} N \mathbb{V}(z(\mathcal{X})) = \frac{1}{N} \mathbb{V}(z(\mathcal{X})) \rightarrow 0$$

Der Zentrale Grenzwertsatz (Central Limit Theorem) besagt, dass für grosse N

$$\frac{m_N(z(\mathcal{X})) - \mathbb{E}z(\mathcal{X})}{\sqrt{\frac{1}{N} \mathbb{V}(z(\mathcal{X}))}}$$

sich fast wie eine normalverteilte Zufallsvariable $\mathcal{Y} \sim \mathcal{N}(0, 1)$ verhält. Daraus folgt mit $\sigma(z(\mathcal{X})) = \sqrt{\mathbb{V}(z(\mathcal{X}))}$:

$$\Pr\left[|m_N(z(\mathcal{X})) - \mathbb{E}z(\mathcal{X})| \leq \frac{\lambda \sigma}{\sqrt{N}}\right] = \frac{1}{\sqrt{2\pi}} \int_{-\lambda}^{\lambda} e^{-\frac{x^2}{2}} dx + \mathcal{O}\left(\frac{1}{\sqrt{N}}\right) = p(\lambda) + \mathcal{O}\left(\frac{1}{\sqrt{N}}\right)$$

wobei 2λ die Länge des Integrationsintervalls ist, mit λ definiert als die Länge des untersuchten Intervalls. Wir haben also eine langsame Konvergenz mit Fehler in $\mathcal{O}\left(\frac{1}{\sqrt{N}}\right)$.

Oben (Bemerkung 5.8.1) wurde bereits erwähnt, dass wir 68.3% als die Wahrscheinlichkeit haben, dass wir den exakten Wert in unserem Intervall haben. Woher kommt aber dieser Wert? Wenn wir $\lambda = 1$ wählen (wie es der Fall ist für das gewählte Intervall), dann erhalten wir $p(1) \approx 0.683$. Dies trifft allerdings nur zu, wenn N genügen gross ($N \geq 30$) ist.

Um die Präzision abzuschätzen, benötigen wir einen Schätzer für $\mathbb{V}(z(\mathcal{X})) = \mathbb{E}(z(\mathcal{X})^2) - (\mathbb{E}(z(\mathcal{X})))^2$:

$$d_2^*(z(\mathcal{X})) := \frac{1}{N-1} \sum_{j=1}^N (z(\mathcal{X}_j) - m_N(z(\mathcal{X})))^2 = \frac{N}{N-1} (m_N(z(\mathcal{X}))^2 - (m_N(z(\mathcal{X})))^2)$$

Aus dem kann die Schätzung für $\tilde{\sigma}_N$ von oben hergeleitet werden:

$$\sqrt{\frac{N}{N-1} (m_N(z(x)^2)) - (m_N(z(x)))^2}$$

für $y = \sqrt{\mathbb{V}(z(\mathcal{X}))}$ (oder ohne das N im Zähler für $\frac{y}{\sqrt{N}}$) und $\mathbb{V}(z(\mathcal{X})) = \mathbb{E}d_2^*(z(\mathcal{X}))$

Vertrauensintervall

Satz 5.8.17

Das Intervall mit $y = \lambda d_2^*(z(\mathcal{X}))$

$$\left[m_N(z(\mathcal{X})) - \frac{y}{\sqrt{N}}, m_N(z(\mathcal{X})) + \frac{y}{\sqrt{N}} \right]$$

enthält den wahren Wert $I = \mathbb{E}(z(\mathcal{X}))$ mit Wahrscheinlichkeit $p(\lambda)$ bis auf einen Fehler in $\mathcal{O}\left(\frac{1}{\sqrt{N}}\right)$ und das Vertrauensintervall ist:

$$[m_N(z(\mathcal{X})) - \lambda \tilde{\sigma}_N, m_N(z(\mathcal{X})) + \lambda \tilde{\sigma}_N]$$

Wiederholen wir das Experiment M mal, mit M gross, so enthält das Vertrauensintervall den Wert I in ungefähr $100p(\lambda)$ Prozent der M Fälle.

Bemerkung 5.8.21: Ein kurzes Vertrauensintervall entspricht einer hohen Wahrscheinlichkeit, dass das Vertrauensintervall den wahren Wert enthält. Im Grundrezept wird $\lambda = 1$ mit $p(1) = 0.683$ verwendet. Um ein kürzeres Vertrauensintervall zu erzielen können wir N erhöhen, oder die Berechnungen so reorganisieren, dass die Varianz kleiner wird. Wir brauchen also für einen Fehler ε ungefähr $\frac{1}{\varepsilon^2}$ Evaluierungen.

Seiten 171 bis 176 im Skript enthalten eine Implementation. Unten eine simple Implementation ohne Plotting:

```

1 import numpy as np
2 def monte_carlo_integral(func, a, b, N):
3     t = np.random.uniform(a, b, N)
4     fx = func(t)
5     I = np.mean(fx) * (b - a)
6     var = np.std(fx, ddof=1) / np.sqrt(N)
7     return I, var

```

5.9 Methoden zur Reduktion der Varianz

Bei höheren Dimensionen d ist die Monte-Carlo-Methode oft die einzige praktikable Option. Deshalb ist es wichtig, Methoden zu haben, um die Varianz zu verringern.

5.9.1 Control Variates

Die Idee ist hier, bekannte Integrale zu verwenden, um die Varianz zu reduzieren. Wir schreiben unser Integral unter Verwendung eines bekannten, exakten Integrals $\varphi(x)$ neu:

$$\iiint f(x) = \iiint (z(x) - \varphi(x)) + \iiint \varphi(x)$$

Das Ganze funktioniert natürlich für jedes $d \in \mathbb{N}$. Oft wird die Taylor-Entwicklung von $f(x)$ gewählt, da diese einfach analytisch integrierbar ist.

Die Varianz wird dadurch reduziert, dass wir nur noch für $z(x)$ einen Fehler haben.

In `numpy` können wir dies folgendermassen implementieren:

```

1 def control_variate_mc(func, phi, analytic_int_phi, a, b, N):
2     t = np.random.uniform(a, b, N)
3     val = func(t) - phi(t)
4     I = np.mean(val) * (b - a) + analytic_int_phi
5     return I

```

5.9.2 Importance Sampling

Importance Sampling

Intuition

- Nicht alle Punkte, die während der Monte-Carlo Integration gezogen werden sind, sind gleich wichtig
- Importance Sampling optimiert die Verteilung der Punkte
- Man gewichtet die Punkte mit einer Dichtefunktion $g(x)$, die wichtige Bereiche betont
- Der Erwartungswert wird als gewichteter Mittelwert berechnet, sodass keine Verzerrung auftritt

Wir schreiben unser zu berechnendes Integral mit $D = [0, 1]^d$ ein Intervall

$$I = \int_D f(x) \, dx$$

mit der Hilfsdichte $g(x)$ (für welche gilt $\int_D g(x) \, dx = 1$)

$$I = \int_D \frac{f(x)}{g(x)} g(x) \, dx = \mathbb{E}_g \left(\frac{f(\mathcal{X})}{g(\mathcal{X})} \right)$$

Der entsprechende Monte-Carlo-Schätzer mit N Stichproben $\mathcal{X}_i \sim g$ ist

$$\hat{I}_N = \frac{1}{N} \sum_{i=1}^N \frac{f(\mathcal{X}_i)}{g(\mathcal{X}_i)}$$

und dessen Varianz ist

$$\mathbb{V}_g \left(\frac{f(\mathcal{X})}{g(\mathcal{X})} \right) = \int_D \frac{f^2(x)}{g(x)} \, dx - I^2$$

Ideal ist $g(x) \propto |f(x)|$, also proportional zum Betrag von $f(x)$

5.9.3 Quasi-Monte-Carlo

Oft ist ein deterministischer Fehler nützlich, weshalb man bei der Quasi-Monte-Carlo-Methode die Zufallszahlen durch quasi-zufällige Folgen ersetzt. Diese Folgen decken den Integrationsbereich systematisch ab.

Dies führt dazu, dass unser Fehler mit $\mathcal{O}(N^{-1} \cdot (\log(N))^d)$ abnimmt. Für kleine d haben wir ungefähr eine Abnahme in $\mathcal{O}(N^{-1})$, aber bei grossen d ist die Verbesserung kaum mehr sichtbar.

In der Realität sind diese Methoden (besonders die Sobol-Sequenzen) trotzdem effektiv, da viele Integrale "effektiv niedrigdimensional" sind.

6 Nullstellensuche

6.1 Iterative Verfahren

Definition 6.1.1: Ein iteratives Verfahren ist ein Algorithmus ϕ_F , der die Folge $x^{(0)}, x^{(1)}, \dots$ von approximativen Lösungen $x^{(j)}$ generiert. Die Definition ist dabei rekursiv: $x^{(k)} := \phi_F(x^{(k-1)})$, sofern $x^{(0)}$ und ϕ gegeben sind.

Definition 6.1.5: (Konvergenz) ϕ_F zur Lösung $F(x^*) = 0$ konvergiert, wenn $x^{(k)} \rightarrow x^*$, mit x^* die Nullstelle.

Definition 6.1.8: (Norm)

In numpy haben wir `numpy.linalg.norm`, welches zwei Argumente nimmt. Dabei ist das erste Argument der Vektor und das Zweite die Art der Norm. Ohne zweites Argument wird die Euklidische Norm $\|x\|_2$, mit Argument 1 wird die 1-Norm $\|x\|_1 := |x_1| + \dots + |x_n|$ und mit `inf` als Argument wird die ∞ -Norm, bzw. die Max-Norm $\|x\|_\infty := \max\{|x_1|, \dots, |x_n|\}$ berechnet.

Definition 6.1.10: Zwei Normen $\|\cdot\|_1$ und $\|\cdot\|_2$ sind äquivalent auf \mathcal{V} , falls es Konstanten \underline{C} und \overline{C} gibt so dass

$$\underline{C} \cdot \|v\|_1 \leq \|v\|_2 \leq \overline{C} \cdot \|v\|_1 \quad \forall v \in \mathcal{V}, \text{ mit } \mathcal{V} \text{ ein linearer Raum}$$

Satz 6.1.11: Falls $\dim(\mathcal{V}) < \infty$, dann sind alle Normen auf \mathcal{V} äquivalent

Definition 6.1.13: (Lineare Konvergenz) $x^{(k)}$ konvergiert linear gegen x^* , falls es ein $L < 1$ gibt, so dass

$$\|x^{(k+1)} - x^*\| \leq L \|x^{(k)} - x^*\| \quad \forall k \geq k_0, \quad L \text{ genannt Konvergenzrate}$$

Definition 6.1.15: (Konvergenzordnung) p für das Verfahren, falls es ein $C > 0$ gibt, so dass

$$\|x^{(k+1)} - x^*\| \leq C \|x^{(k)} - x^*\|^p \quad \forall k \in \mathbb{N} \text{ mit } C < 1 \text{ für } p = 1$$

Wir nehmen dabei an, dass $\|x^{(0)} - x^*\| < 1$, damit wir eine konvergente Folge haben. Man kann die Konvergenzordnung folgendermassen abschätzen, mit $\varepsilon_k := \|x^{(k)} - x^*\|$ (Konvergenzrate in Bemerkung 6.1.19):

$$p \approx \frac{\log(\varepsilon_{k+1}) - \log(\varepsilon_k)}{\log(\varepsilon_k) - \log(\varepsilon_{k-1})}$$

Intuitiv haben wir Quadratische (oder Kubische, etc.) Konvergenzordnung, wenn sich die Anzahl Nullen im Fehler jede Iteration verdoppeln (verdreifachen, etc.)

Bemerkung: Eine höhere Konvergenzordnung ist in Lin-Log-Skala an einer gekrümmten Konvergenzkurve erkennbar.

Bemerkung 6.1.19: (Abschätzung der Konvergenzrate) Sei $\varepsilon_k := \|x^{(k)} - x^*\|$ die Norm des Fehlers im k -ten Schritt.

$$\varepsilon_{k+1} \approx L \cdot \varepsilon_k \implies \log(\varepsilon_{k+1}) \approx \log(L) + \log(\varepsilon_k) \implies \varepsilon_{k+1} \approx k \log(L) + \log(\varepsilon_0)$$

Untenstehender Code berechnet den Fehler und die Konvergenzrate von $x^{(k+1)} = x^{(k)} + \frac{\cos(x^{(k)}) + 1}{\sin(x^{(k)})}$. Dabei verwenden wir $x^{(15)}$ anstelle von x^* zur Berechnung der Konvergenzrate, da x^* meist unbekannt ist.

```

1 def linear_convergence(x):
2     y = [] # container for the x(j)
3     for k in range(15):
4         x = x + (np.cos(x) + 1) / np.sin(x) # apply the iteration formula
5         y += [x] # store the value in the container
6     err = abs(np.array(y) - x) # estimation for the error
7     rate = err[1:] / err[:-1]
8     # estimation for convergence rate
9     return err, rate

```

6.2 Abbruchkriterien

Wir müssen irgendwann unsere Iteration abbrechen können, dazu haben wir folgende Möglichkeiten:

Typ	Idee	Vorteile	Nachteile
A priori	Fixe Anzahl k_0 Schritte	Einfach zu implementieren	Zu ungenau
A posteriori Ungefähr gleich	Berechnen bis Toleranz $\varepsilon < \tau$ erreicht Iteration bis $x^{(k+1)} \approx x^{(k)}$	Präzise Keine Voraussetzungen	Man kennt x^* nicht Ineffizient
Residuum	Abbruch wenn $\ F(x^{(k)})\ < \tau$ (wir also fast bei 0 sind mit dem Funktionswert)	Einfach zu implementieren	Bei flachen Funktionen kann $\ F(x^{(k)})\ $ klein sein, aber ε gross

Tabelle 6.2.1: Vergleich der Abbruchkriterien

Bemerkung 6.2.2: Für das *a posteriori* Abbruchkriterium mit linearer Konvergenz und bekanntem L gilt die Abschätzung aus Lemma 6.3.6 mit Korollar 6.3.17

6.3 Fixpunktiteration

Ein 1-Punkt-Verfahren benötigt nur den vorigen Wert: $x^{(k+1)} = \phi(x^{(k)})$

Definition 6.3.1: Eine Fixpunktiteration heisst konsistent mit $F(x) = 0$ falls $F(x) = 0 \Leftrightarrow \phi(x) = x$

Beispiel 6.3.2: Für $F(x) = xe^x - 1$ mit $x \in [0, 1]$ liefert $\phi_1(x) = e^{-x}$ lineare Konvergenz, $\phi_2(x) = \frac{1+x}{1+e^x}$ quadratische Konvergenz und $\phi_3(x) = x + 1 - xe^x$ eine divergente Folge.

Definition 6.3.5: (Kontraktion) ϕ falls es ein $L < 1$ gibt, so dass $\|\phi(x) - \phi(y)\| \leq L\|x - y\| \forall x, y$

Bemerkung 6.3.6: Falls x^* ein Fixpunkt der Kontraktion ϕ ist, dann ist

$$\|x^{(k+1)} - x^*\| = \|\phi(x^{(k)}) - \phi(x^*)\| \leq L\|x^{(k)} - x^*\|$$

Banach'scher Fixpunktsatz

Satz 6.3.7

Sei $D \subseteq \mathbb{K}^n$ ($\mathbb{K} = \mathbb{R}, \mathbb{C}$) mit D abgeschlossen und $\phi : D \rightarrow D$ eine Kontraktion. Dann existiert ein eindeutiger Fixpunkt x^* , für welchen also gilt, dass $\phi(x^*) = x^*$. Dieser ist der Grenzwert der Folge $x^{(k+1)} = \phi(x^{(k)})$.

Lemma 6.3.8: Für $U \subseteq \mathbb{R}^n$ konvex und $\phi : U \rightarrow \mathbb{R}^n$ stetig differenzierbar mit $L := \sup_{x \in U} \|D_\phi(x)\| < 1$ ($D_\phi(x)$ ist die Jacobi-Matrix von $\phi(x)$). Wenn $\phi(x^*) = x^*$ für $x^* \in U$, dann konvergiert $x^{(k+1)} = \phi(x^{(k)})$ gegen x^* lokal mindestens linear. Dies ist eine hinreichende (= sufficient) Bedingung.

Lemma 6.3.11: Für $\phi : \mathbb{R}^n \rightarrow \mathbb{R}^n$ mit $\phi(x^*) = x^*$ und ϕ stetig differenzierbar in x^* . Ist $\|D_\phi(x^*)\| < 1$, dann konvergiert $x^{(k+1)} = \phi(x^{(k)})$ lokal und mindestens linear mit $L = \|D_\phi(x^*)\|$

Satz 6.3.13: (Satz von Taylor) Sei $I \subseteq \mathbb{R}$ ein Intervall, $\phi : I \rightarrow \mathbb{R}$ $(m+1)$ -mal differenzierbar und $x \in I$. Dann gilt für jedes $y \in I$

$$\phi(y) - \phi(x) = \sum_{k=1}^m \frac{1}{k!} \left(\phi^{(k)}(x)(y-x)^k \right) + \mathcal{O}(|y-x|^{m+1})$$

Lemma 6.3.14: Sei I und ϕ wie in Satz 6.3.13. Sei zudem $\phi^{(l)}(x^*) = 0$ für $l \in \{1, \dots, m\}$ mit $m \geq 1$. Dann konvergiert $x^{(k+1)} = \phi(x^{(k)})$ lokal gegen x^* mit Ordnung $p \geq m+1$

Lemma 6.3.16: Konvergiert ϕ linear mit $L < 1$, dann gilt:

$$\|x^* - x^{(k)}\| \leq \frac{L^{k-l}}{1-L} \|x^{(l+1)} - x^{(l)}\|$$

Korollar 6.3.17: für $l = 0$ haben wir ein *a priori* und für $l = k-1$ ein *a posteriori* Abbruchkriterium:

$$\|x^* - x^{(k)}\| \leq \frac{L^k}{1-L} \|x^{(1)} - x^{(0)}\| \leq \tau \qquad \|x^* - x^{(k)}\| \leq \frac{L}{1-L} \|x^{(k)} - x^{(k-1)}\| \leq \tau$$

6.4 Intervallhalbierungsverfahren

Die Idee hier ist, das Intervall immer weiter zu halbieren und ein bekannterer Namen für dieses Verfahren ist **Bisektionsverfahren**.

In `numpy` haben wir `scipy.optimize.bisect` und `scipy.optimize.fsolve`, wobei `fsolve` ein alter Algorithmus ist.

Im Skript auf Seiten 206 - 207 findet sich eine manuelle implementation des Bisektionsverfahren. Der Code ist jedoch (at the time of writing) nicht ausführbar aufgrund von `IndentationErrors`

Das Bisektionsverfahren konvergiert linear und kann nur für Funktionen verwenden, bei welchen die Nullstellen auf beiden Seiten jeweils ungleiche Vorzeichen haben.

Für jeden Iterationsschritt ermitteln wir die Mitte des Intervalls und berechnen die Funktionswerte an den Rändern, wie auch dem Mittelpunkt. Dann ersetzen wir den Rand des Intervalls, dessen Funktionswert dasselbe Vorzeichen hat, wie der Funktionswert des Mittelpunkts.

6.5 Newtonverfahren in 1D

Beim Newtonverfahren verwendet man für jeden Iterationsschritt die lineare Funktion $\tilde{F} = F(x^{(k)}) + F'(x^{(k)})(x - x^{(k)})$. Die Nullstelle ist dann:

$$x^{(k+1)} := x^{(k)} - \frac{F(x^{(k)})}{F'(x^{(k)})}, \quad \text{falls } F'(x^{(k)}) \neq 0$$

Bemerkung 6.5.2: Die Newton-Iteration ist eine Fixpunktiteration mit quadratischer lokaler Konvergenz, mit

$$\phi(x) = x - \frac{F(x)}{F'(x)} \implies \phi'(x) = \frac{F(x)F''(x)}{(F'(x))^2} \implies \phi'(x^*) = 0$$

falls $F(x^*) = 0$ und $F'(x^*) \neq 0$

6.6 Sekantenverfahren

Falls die Ableitung zu teuer oder nicht verfügbar ist, kann man sie durch $q^{(k)} := \frac{F(x^{(k)}) - F(x^{(k-1)})}{x^{(k)} - x^{(k-1)}}$. Dann ist ein Schritt:

$$\tilde{F}(x) = F(x^{(k)}) + q^{(k)}(x - x^{(k)}) \implies x^{(k+1)} := x^{(k)} - \frac{F(x^{(k)})}{q^{(k)}}, \quad \text{falls } q^{(k)} \neq 0$$

6.7 Newton-Verfahren in n Dimensionen

Sei $D \subseteq \mathbb{R}^n$ und $F : D \rightarrow \mathbb{R}^n$ stetig differenzierbar. Die Nullstelle ist

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

wobei $DF(x^{(k)}) = \left[\frac{\partial F_i}{\partial x_k}(x) \right]_{j,k=1,2,\dots,n}$ die Jacobi-Matrix von F ist.

Wichtig ist dabei, dass wir **niemals** das Inverse der Jacobi-Matrix (oder irgend einer anderen Matrix) von der Form $s = A^{-1}b$, sondern immer das Gleichungssystem $As = b$ lösen sollten, da dies effizienter ist:

```

1 def newton(x, F, DF, tol=1e-12, maxit=50):
2     x = np.atleast_2d(x) # 'solve' erwartet x als 2-dimensionaler numpy array
3     # Newton Iteration
4     for _ in range(maxit):
5         s = np.linalg.solve(DF(x), F(x))
6         x -= s
7         if np.linalg.norm(s) < tol * np.linalg.norm(x):
8             return x

```

Wollen wir aber garantiert einen Fehler kleiner als unsere Toleranz τ können wir das Abbruchkriterium

$$\|DF(x^{(k-1)})^{-1}F(x^{(k)})\| \leq \tau$$

verwenden. Code, welcher dies implementiert findet sich auf Seite 213-216 im Skript.

6.8 Gedämpftes Newton-Verfahren

Wir wenden einen Dämpfungsfaktor $\lambda^{(k)}$ an, welcher heuristisch gewählt wird:

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

Wir wählen $\lambda^{(k)}$ so, dass für $\Delta x^{(k)} = DF(x^{(k)})^{-1} F(x^{(k)})$ und $\Delta(\lambda^{(k)}) = DF(x^{(k)})^{-1} F(x^{(k)} - \lambda^{(k)} \Delta x^{(k)})$

$$\|\Delta x(\lambda^{(k)})\|_2 \leq \left(1 - \frac{\lambda^{(k)}}{2}\right) \|\Delta x^{(k)}\|_2$$

6.9 Quasi-Newton-Verfahren

Falls $DF(x)$ zu teuer ist oder nicht zur Verfügung steht, können wir im Eindimensionalen das Sekantenverfahren verwenden.

Im höherdimensionalen Raum ist dies jedoch nicht direkt möglich und wir erhalten die Broyden-Quasi-Newton Methode:

$$J_{k+1} := J_k + \frac{F(x^{(k+1)})(\Delta x^{(k)})^\top}{\|\Delta x^{(k)}\|_2^2}$$

Dabei ist J_0 z.B. durch $DF(x^{(0)})$ definiert.

7 Intermezzo: Lineare Algebra

Das Skript-Kapitel hierzu existiert nicht. Die Vorlesung orientiert sich an dem äquivalenten Teil des NumCSE Dokuments.

Perturbierte LGS

Statt $Ax = b$ ist das LGS ungenau gegeben: $(A + \Delta A)(\tilde{x} - x) = \Delta b - \Delta Ax$.

$\text{cond}(A) := \|A^{-1}\| \cdot \|A\| \in \mathbb{R}$ (Konditionszahl von A)

$\text{cond}(A) \gg 1$ bedeutet intuitiv: kleine Änderung der Daten \mapsto grosse Änderung in der Lösung

Gauss Elimination / LU Zerlegung

$A \in \mathbb{R}^{n \times m} = PLU$ wie bekannt.

Cholesky Zerlegung (A pos. def. und hermitesch)

$$A = LDL^T = \underbrace{L\sqrt{D}}_{R^T} \underbrace{\sqrt{D}L^T}_R = R^T R$$

Kann $Ax = b$ potenziell schneller lösen als LU.

```

1 L = np.linalg.solve(A)           # A = L @ L.T
2 y = np.linalg.solve(L, b)
3 x = np.linalg.solve(L.T, y)

```

Grosse Matrizen

Passen oft nicht (direkt) in den Speicher: effizientere Speicherung nötig, möglich für z.B. Diagonalmatrizen, Dreiecksmatrizen. Auch für Cholesky möglich.

Dünnbesetzte Matrizen

$\text{nnz}(A) := |\{(i, j) \mid a_{ij} \in A, a_{ij} \neq 0\}| \ll m \cdot n$

$$\lim_{l \rightarrow \infty} \frac{\text{nnz}(A^{(l)})}{n_l m_l} = 0$$

Einfacher zu speichern: `val`, `col`, `row` vektoren s.d. `val[k] = aij` wobei $i = \text{row}[k]$, $j = \text{col}[k]$. (nur $a_{ij} \neq 0$)

Viele Formate, je nach Anwendung gewisse sinnvoller als andere. (Siehe Tabelle, NumCSE)

`scipy.sparse.csr_matrix(A)` \mapsto Dramatische Speichereinsparung.

Deprecated: `bsr_array` und `coo_array` verwenden, kompatibel mit `numpy arrays`.

CSC, CSR erlauben weitere Optimierungen, je nach Gewichtung der a_{ij} auf Zeilen, Spalten.

8 Ausgleichsrechnung

Der Begriff "Ausgleichsrechnung" mag vielleicht nicht bekannt sein, jedoch macht die Englische Übersetzung klar, was der Inhalt dieses Abschnitts ist: **Curve Fitting**.

Wir werden also (unter anderem) Least-Squares-Probleme behandeln

8.1 Lineare Ausgleichsrechnung

Die Ansatz der Methode der kleinsten Quadrate ist (ausgedrückt mit Matrizen) ist $\min_{\hat{x} \in \mathbb{R}^n} \|A\hat{x} - b\|^2$ und als Summe:

$$(a, c) = \operatorname{argmin}_{p \in \mathbb{R}^n, q \in \mathbb{R}} \sum_{i=1}^m |y_i - p^\top x_i - q|^2$$

Wobei y_i die y -Koordinaten der Messpunkte zugehörig zu x_i sind.

In numpy haben wir die Funktionen `numpy.polyfit` (um ein Polynom zu fitten), oder die allgemeinere Methode `numpy.linalg.lstsq`. Um eine eindeutige Lösung zu erhalten können wir die Moore-Penrose (eine Art der Pseudoinversen) verwenden, wofür `numpy.linalg.pinv` und `numpy.linalg.pinv2` zur Verfügung stehen

8.1.1 Normalengleichung

Definition 8.1.9: (Normalengleichung) $A^H A x = A^H b$

Bemerkung 8.1.10: $A^H A$ ist Hermite-Symmetrisch, und falls A vollen Rank hat, dann ist $A^H A$ positiv-definit und die Normalengleichung hat eine eindeutige Lösung. Jedoch ist die Normalengleichung schlecht konditioniert (es gilt: $\operatorname{cond}(A^H A) = \operatorname{cond}(A)^2$). Für gut konditionierte Matrizen ist dies kein Problem, jedoch ist die Normalengleichung für schlecht konditionierte Matrizen ungeeignet.

Bemerkung 8.1.11: Man kann die Normalengleichung auch ohne die Berechnung von $A^H A$ berechnen:

$$A^H A x = A^H b \iff B \begin{bmatrix} r \\ x \end{bmatrix} := \begin{bmatrix} -I & A \\ A^H & 0 \end{bmatrix} \begin{bmatrix} r \\ x \end{bmatrix} = \begin{bmatrix} b \\ 0 \end{bmatrix}$$

für $r := \frac{1}{a}(Ax - b)$ mit $a > 0$, dann können wir B in obiger Gleichung durch $B_a = \begin{bmatrix} -aI & A \\ A^H & 0 \end{bmatrix}$ ersetzen, wobei wir a so wählen, dass $\kappa(B_a)$ minimal wird.

8.1.2 Lösung mittels orthogonaler Transformation

Nicht nur die Normalengleichungen, aber auch das LU-Verfahren kann für gewisse Matrizen (im Falle von LU sind es Matrizen mit $m > n$) ungeeignet sein.

Wir versuchen wieder $\|r\|_2^2$ zu minimieren, mit $r = Ax - b$. Mithilfe der QR -Zerlegung lässt sich ein Lösungsansatz herleiten, der höhere numerische Stabilität aufweist, als die Normalengleichungen. Sei $A = QR = Q \begin{bmatrix} \tilde{R} \\ 0 \end{bmatrix}$. Dann, nach Umformungen erhalten wir $\|Rx - \tilde{b}\|_2^2$ mit $\tilde{b} = Q^H b$.

Nutzt man beispielsweise Housholder-Spiegelungen zur Berechnung der QR -Zerlegung, so kann man die Transformationen direkt auf b anwenden und so kann man sich das Abspeichern der Matrix Q komplett sparen.

Falls jedoch die Matrix A nicht vollen Rang hat (was sehr oft der Fall ist), dann ist es besser, die Singulärwertzerlegung zu verwenden. Dann ist:

$$\|Ax - b\|_2 = \|U\Sigma V^H x - b\|_2 = \|\Sigma V^H x - U^H b\|_2$$

In numpy verwendet `numpy.linalg.lstsq` die SVD für das Lösen

Definition 8.1.15: (Pseudoinverse) $A^+ = (A^H A)^{-1} A^H = V_1 \Sigma_r^+ U_1^H$

Die drei bisher besprochenen Verfahren lassen sich in zwei Kategorien einordnen:

1. $A \in K^{m \times n}$ ist voll besetzt und n ist klein ($m \gg n$)
2. $A \in K^{m \times n}$ ist dünn besetzt und m, n sind gross

Im ersten Fall wird aufgrund der numerischen Stabilität die QR oder SVD-Methode verwendet. Im zweiten Fall verwendet man die Normalengleichungen, da diese die Struktur der dünn besetzten Matrizen verwenden können.

```

1 import numpy as np
2
3 A = np.array([[98.269, 1.0], [0.0, 1.0], [-194.96, 1.0]])
4 b = np.array([852.7, 624.5, 172.7])
5
6 def least_squares_svd(A, b, epsilon=1e-6):
7     U, s, Vh = np.linalg.svd(A)
8     r = 1 + np.where(s / s[0] > epsilon)[0].max() # numerical rank
9     y = np.dot(Vh[:r, :].T, np.dot(U[:, :r].T, b) / s[:r])
10    return y
11
12 # qr-decomposition:
13 def least_squares_qr(A, b):
14     Q, R = np.linalg.qr(A)
15     b_tilde = np.dot(Q.T, b)
16     return np.linalg.solve(R, b_tilde)
17
18 np.linalg.lstsq(A, b)

```

8.1.3 Totale Ausgleichsrechnung

Es kann vorkommen, dass sowohl die Matrix A , wie auch der Vektor b fehlerhaft sind. Dann ersetzen wir das System $Ax = b$ durch ein neues System $\hat{A}\hat{x} = \hat{b}$, welches so nah wie möglich am ursprünglichen System liegt und so für welches gilt $\hat{b} \in \text{Bild}(\hat{A})$.

Wir versuchen also die folgende Norm zu minimieren:

$$\|C - \hat{C}\|_F = \left\| \begin{bmatrix} A & b \end{bmatrix} - \begin{bmatrix} \hat{A} & \hat{b} \end{bmatrix} \right\|_F$$

Das Problem lässt sich umschreiben als

$$\min_{\text{Rang}(\hat{C})=n} \|C - \hat{C}\|_F$$

Theorem ?? liefert die Lösung. Die Singulärwertzerlegung

$$C = U\Sigma V^H = \sum_{j=1}^{n+1} \sigma_j(u)_j(v)_j^H$$

gibt das Optimum

$$\hat{C} = \sum_{j=1}^n \sigma_j(u)_j(v)_j^H$$

8.2 Nichtlineare Ausgleichsrechnung

Es ist natürlich auch möglich, dass das Modell für das Ausgleichsproblem nicht linear ist.

8.2.1 Newton-Verfahren

Aus der Analysis ist bekannt, dass für gesuchtes $x \in \mathbb{R}^n$, so dass $\Phi(x)$ minimal ist, eine notwendige Bedingung durch $\text{grad}(\Phi(x)) = 0$ gegeben ist.

Da wir also eine Nullstellensuche in \mathbb{R}^n haben, können wir dies mit dem Newton-Verfahren lösen:

$$x^{(k+1)} = x^{(k)} - (D\text{grad}(\Phi(x)))^{-1} \text{grad}(\Phi(x))$$

Da $H_\Phi(x) := D((\text{grad})(\Phi(x)))$ ist, haben wir also:

$$x^{(k+1)} = x^{(k)} - (H_\Phi(x))^{-1} \text{grad}(\Phi(x))$$

8.2.2 Gauss-Newton Verfahren

Direkt das Newton-Verfahren auf ein Problem anzuwenden kann unmöglich oder schwer praktikabel sein.

Die Idee des Gauss-Newton Verfahrens ist es, die komplizierte Funktion $F(x)$ lokal durch eine lineare Funktion approximiert, also:

$$F(x) \approx F(y) + DF(y)(x - y) = F(y) + DF(y)x - DF(y)y$$

Falls man $A := DF(y)$ und $b = DF(y)y - F(y)$ definiert, so erhält man ein lineares Ausgleichsproblem:

$$\underset{x \in \mathbb{R}^n}{\text{argmin}} \frac{1}{2} \|F(x)\|_2^2 \approx \underset{x \in \mathbb{R}^n}{\text{argmin}} \frac{1}{2} \|F(y) + DF(y)x\|_2^2 = \underset{x \in \mathbb{R}^n}{\text{argmin}} \frac{1}{2} \|Ax - b\|_2^2$$

wobei y eine Näherung der Lösung x ist. Die Iterationsvorschrift ist gegeben durch:

$$x^{(k+1)} = x^{(k)} - s \quad \text{mit } s := \underset{z \in \mathbb{R}^n}{\text{argmin}} \|F(x^{(k)}) - DF(x^{(k)})z\|_2^2$$

```

1 import numpy as np
2
3 def gauss_newton(start_vec, Func, Jacobian, tolerance):
4     # Start vector has to be chosen intelligently
5     s = np.linalg.lstsq(Jacobian(start_vec), Func(start_vec))[0]
6     start_vec = start_vec - s
7     # now we perform the iteration
8     while np.linalg.norm(s) > tolerance * np.linalg.norm(start_vec):
9         # every time we update x by subtracting s, found with the least square method
10        s = np.linalg.lstsq(Jacobian(start_vec), Func(start_vec))[0]
11        start_vec = start_vec - s
12    return start_vec

```

Der Vorteil ist, dass die zweite Ableitung nicht benötigt wird, jedoch ist die Konvergenzordnung niedriger ($p \leq 2$)

Beispiel 8.2.3: Wir haben zwei Modellfunktionen, $F_1(t) = a_1 + b_1 e^{-c_1 t}$ and $F_2(t) = a_2 - b_2 e^{-c_2 t}$. (F_1 ist ein Heizvorgang, F_2 ist ein Abkühlvorgang). Untenstehender code berechnet die Lösung des nichtlinearen Ausgleichsproblems

```

1  import numpy as np
2
3  t = np.arange(0, 30, 5); n = len(t)
4  curr_heating = np.array([24.34, 18.93, 17.09, 16.27, 15.97, 15.91])
5  curr_cooling = np.array([9.66, 18.8, 22.36, 24.07, 24.59, 24.91])
6  # define the functions that have to be minimized
7  F_1 = lambda a: a[0] + a[1] * np.exp(-a[2] * t) - curr_heating
8  F_2 = lambda a: a[0] - a[1] * np.exp(-a[2] * t) - curr_cooling
9
10 # define the corresponding Jacobi matrices
11 def J_1(a):
12     mat = np.zeros((n, 3))
13     for k in range(n):
14         mat[k, 0] = 1.0
15         mat[k, 1] = np.exp(-t[k] * a[2])
16         mat[k, 2] = -t[k] * a[1] * np.exp(-t[k] * a[2])
17     return mat
18
19 def J_2(a):
20     mat = np.zeros((n, 3))
21     for k in range(n):
22         mat[k, 0] = 1.0
23         mat[k, 1] = -np.exp(-t[k] * a[2])
24         mat[k, 2] = t[k] * a[1] * np.exp(-t[k] * a[2])
25     return mat
26
27 # guess starting vector
28 x_1 = np.array([10.0, 5.0, 0.0])
29 x_2 = np.array([30.0, 10.0, 0.0])
30
31 # use the Gauss-Newton algorithm declared above
32 a_1 = gauss_newton(x_1, F_1, J_1, tolerance=10e-6)
33 a_2 = gauss_newton(x_2, F_2, J_2, tolerance=10e-6)
34 print("Heating ", a_1)
35 print("Cooling ", a_2)

```

8.2.3 Weitere Methoden: BFGS, GD, SGC, CG, LM, ADAM

Für unterschiedliche Probleme können andere Funktionen günstiger oder besser geeignet sein. Eine Liste einiger bekannter Methoden:

- **BFGS** (basiert auf Broyden): $D\Phi(x^{(k)}) = DF(x^{(k)})^\top F(x^{(k)})$, oder günstiger mit $D\Phi(x^{(k)})^\top DF(x^{(k)})s = DF(x^{(k)})^\top F(x^{(k)})$
- **GD** (Gradient Descent): $s = \lambda_k D\Phi(x^{(k)})$ (in ML wird λ_k als "Learning rate" bezeichnet)
- **LM** (Levenberg-Marquant): wir minimieren $\|F(x^{(k)}) + DF(x^{(k)})s\|_2^2 + \lambda \|s\|_2^2$ (also werden kleine Schritte bevorzugt)
- **CG** (Conjugated Gradient): GD ist sehr langsam, aber auch günstig. Mit höheren Kosten kann durch Wahl von $s = \lambda z^k$ und $z^k = D\Phi(x^{(k)}) + \beta z^{(k-1)}$ eine schnellere Konvergenz erreicht werden (Dämpfung)
- **ADAM** Hier werden spezielle λ und β gewählt und liefert die einfache Iterationen

$$x^{(k+1)} = x^k - \lambda z^k$$

$$z^{(k+1)} = D\Phi(x^{(k)}) + \beta z^{(k)}$$

In numpy gibt es via `scipy.optimize.leastsq` eine Implementation mit verschiedenen Iterationsmethoden, oder alternativ `scipy.optimize.minimize`