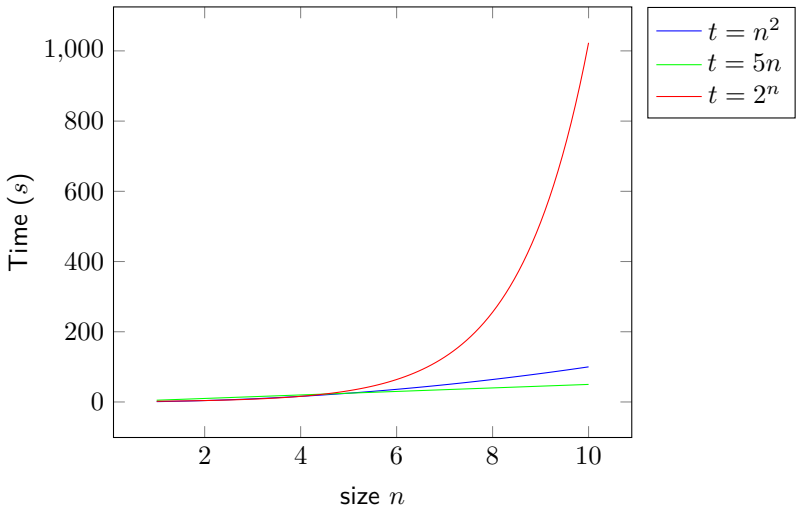
Numerical Methods for Computer Science

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

16. Januar 2026



"*Wenn ich keine Lust habe, das zu berechnen, dann wende ich einfach Gewalt an*"

- Prof. Dr. Vasile Gradinaru, 2025

HS2025, ETHZ

Summary of the Script and Lectures

Inhaltsverzeichnis

0 Introduction

This summary is intended to give you a broad overview of the topics relevant for the exam. While it aims to serve as a full on replacement for the script, please do not fully rely on it, as there may be mistakes, inaccuracies and missing details as compared to the script. Furthermore, you will only have access to the script during the exam, so getting familiar with the script is a good idea. 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. (Link will be updated if we are to get a new website link from him, as n.ethz.ch is down now)

A few things to be familiar with for the exam to be quicker at solving exercises:

- The script. You do not have to know basically anything by heart, so knowing where to find things in the script quickly (and knowing quirks of naming in the script) will make you much quicker. When searching, always use as short of a keyword as possible while not having hundreds of results.
- Be aware that some things might not be called the same way they are in the exercises. If you don't find it immediately, go to the appropriate section in the script (using the index in the PDF reader (that is likely PDF.js, Firefox's PDF reader)) and manually search for it there.
- Get familiar with the concepts that `NumPy` uses (i.e. how to do slicing, etc). See section 9 for an overview over some of the concepts.
- Learn the basics of `Sympy`, as that will spare you having to do integrals. See section 9.4 for an introduction to `Sympy`.

1 Einführung

1.1 Rundungsfehler

Absoluter & Relativer Fehler **Definition 1.1.1**

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

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

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

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

Auslöschung **Bemerkung 1.1.9**

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

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

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

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

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

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

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

Beispiel 1.1.20: *(Konvergenzbeschleunigung nach Richardson)*

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

Schema

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

wobei im Schema dann

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

und

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

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

1.2 Rechenaufwand

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

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

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

1.3 Rechnen mit Matrizen

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

In `numpy` haben wir folgende Funktionen:

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

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

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

Beispiel 1.3.8: `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 $A = ab^\top$. Dann gilt $y = Ax \Leftrightarrow y = a(b^\top x)$, was dasselbe Resultat ergibt, aber nur $\mathcal{O}(m+n)$ Operationen und nicht $\mathcal{O}(mn)$ benötigt wie links.

Beispiel 1.3.18: Für zwei Matrizen $A, B \in \mathbb{R}^{n \times p}$ mit geringem Rang $p \ll n$, dann kann mithilfe eines Tricks die Rechenzeit von `np.triu(A @ B.T) @ x` von $\mathcal{O}(pn^2)$ auf $\mathcal{O}(pn)$ reduziert werden. Die hier beschriebene Operation berechnet $\text{Upper}(AB^\top)x$ wobei $\text{Upper}(X)$ das obere Dreieck der Matrix X zurück gibt. Wir nennen diese Matrix hier R.

In numpy können wir den folgenden Ansatz verwenden, um die Laufzeit zu verringern: Da die Matrix R eine obere Dreiecksmatrix ist, ist das Ergebnis die Teilsummen von unserem Umgekehrten Vektor x, also können wir mit `x[::-1].cumsum(axis=0)[::-1]` die Kummulative Summe berechnen. Das `[::-1]` dient hier lediglich dazu, den Vektor x umzudrehen, sodass das richtige Resultat entsteht und die `axis=0` muss nur spezifiziert werden, falls wir nicht den Default von `None` wollen, welcher die `cumsum` auf `x.flat` ausführt. Die vollständige Implementation sieht so aus:

```python
def low_rank_matrix_vector_product(A: np.ndarray, B: np.ndarray, x: np.ndarray):
    n = A.shape[0]
    y = np.zeros(n) # Results vector

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

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

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

Definition 1.3.21: *(Kronecker-Produkt)* Das Kronecker-Produkt ist eine $(ml) \times (nk)$-Matrix, für $A \in \mathbb{R}^{m \times n}$ und $B \in \mathbb{R}^{l \times k}$.

In numpy können wir dieses einfach mit `np.kron(A, B)` berechnen (ist jedoch nicht immer ideal):

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

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

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

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

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

2 Polynominterpolation

2.1 Interpolation und Polynome

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

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

Normalerweise stellt f eine echte Messung dar, d.h. es macht 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 also $\widetilde{f}$ in einem Unterraum $\mathcal{V}_n$ mit endlicher $\dim(\mathcal{V}_n) = n$. Sei $B_n = \{b_1, \dots, b_n\}$ eine Basis für $\mathcal{V}_n$. Dann lässt sich der Bezug zwischen f und $\widetilde{f} = f_n(x)$ so ausdrücken:

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

Bemerkung 2.1.1: Unterräume $\mathcal{V}_n$ existieren nicht nur für Polynome, wir beschränken uns aber auf $b_j(x) = x^{i-1}$. Andere Möglichkeiten: $b_j = \cos((j-1)\cos^{-1}(x))$ *(Chebyshev)* oder $b_j = e^{i2\pi jx}$ *(Trigonometrisch)*

Satz 2.1.2: *(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)* $p(x) \in (P)_k$ ist durch $k+1$ Punkte $y_i = p(x_i)$ eindeutig bestimmt.

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

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

```python
def coeffs_monomial(x: np.ndarray, y: np.ndarray):
    """ Solve Vandermonde matrix for monomial coeffs (very unstable) """
    A = np.vander(x)
    coeffs = np.linalg.solve(A, y)
    return coeffs
```

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

```python
def eval_horner(coeffs: np.ndarray, vals: np.ndarray):
    """ Evaluate polynomial using Horner scheme """
    h = coeffs[0]
    for i in range(1, len(coeffs)): h = vals * h + coeffs[i]
    return h
```

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

2.2 Newton Basis

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

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

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

Satz 2.2.2: *(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)*

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

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

Bemerkung 2.2.5: *(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 $f(x) = \dfrac{1}{1+x^2}$

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

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

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

    return T[0,:]
```

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

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

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

    return y
```

2.2.2 Auswertung

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

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

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

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

    return p
```

2.2.3 Fehler

Satz 2.2.10: 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)* $f : [a,b] \to \mathbb{R}$ ist $(n+1)$-mal diff.-bar, p ist das Polynom zu f in $x_0, \dots, x_n \in [a,b]$.

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

2.3 Lagrange- und Baryzentrische Interpolationsformeln

Lagrange Polynome **Definition 2.3.1**

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

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

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 $n = 2$):

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

Lagrange-Interpolationsformel **Satz 2.3.3**

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

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

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

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

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

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

oder das ganze mithilfe von Numpy:

```python
def weights_barycentric(x: np.ndarray):
    """ All x should be pairwise distinct, else zero-divison error. """
    n = x.size
    w = np.zeros(n)
    for i in range(n):
        w[i] = 1/( np.prod(x[i] - x[0:i]) * np.prod(x[i] - x[i+1:n]) )
    return w
```

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

Baryzentrische Interpolationsformel

Formel

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

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

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

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

```python
def eval_barycentric(w: np.ndarray, data: np.ndarray, y: np.ndarray, x: np.ndarray):
    """  Sequentially calculate the barycentric formula """
    n = x.size
    tmp = np.ones(n)
    for i in range(n): tmp[i] = eval_barycentric_scalar(w, data, y, x[i])
    return tmp


def eval_barycentric_scalar(w: np.ndarray, data: np.ndarray, y: np.ndarray, x):
    """ Barycentric interpolation formula for a single value x """
    n = data.size;
    bottom = np.sum( w / (x - data) )
    top = np.sum( w / (x - data) * y)
    return top / bottom
```

2.3.1 Fehler

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

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

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

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

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

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

Fehler **Satz 2.3.8**

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

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

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

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

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

2.4 Chebyshev Interpolation

Chebyshev-Polynome **Definition 2.4.1**

Erster Art

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

Zweiter Art

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

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

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

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

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

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

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

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

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

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

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

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

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

Orthogonalität **Satz 2.4.9**

Die Chebyshev-Polynome sind orthogonal bezüglich des Skalarprodukts

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

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

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

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

2.4.1 Fehler

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

Fehlerabschätzung **Satz 2.4.11**

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

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

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

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

Interpolationspolynom **Satz 2.4.15**

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

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

wobei für die c_k gilt:

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

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

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

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

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

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

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

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

```python
def clenshaw(c: np.ndarray, x: np.ndarray):
    """ Clenshaw algorithm to evaluate polynomial using chebyshev coeffs """
    n = c.size; m = x.size
    d = np.zeros((m, 3))    # Save vectors [curr, prev1, prev2] as matrix

    for i in range(n-1, -1, -1):
        d[:, 2] = d[:, 1]; d[:, 1] = d[:, 0]
        d[:, 0] = c[i] + (2*x)*d[:, 1] - d[:, 2]
    return d[:, 0] - x*d[:, 1]
```

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

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

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

3 Trigonometrische Interpolation

3.1 Fourier-Reihen

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

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

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

---

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

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

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

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

L^2-Funktionen **Definition 3.1.3**

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

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

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

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

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

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

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

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

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

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

Fourier-Reihe **Satz 3.1.8**

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

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

wobei die Fourier-Koeffizienten

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

definiert sind. Es gilt die Parseval'sche Gleichung:

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

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

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

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

Fourier-Reihe **Satz 3.1.15**

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

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

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

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

Bemerkung 3.1.17: Diese Überschwingungen entstehen durch die Definition der Fourier-Reihe und sind in der untenstehenden Abbildung 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:

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

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

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

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

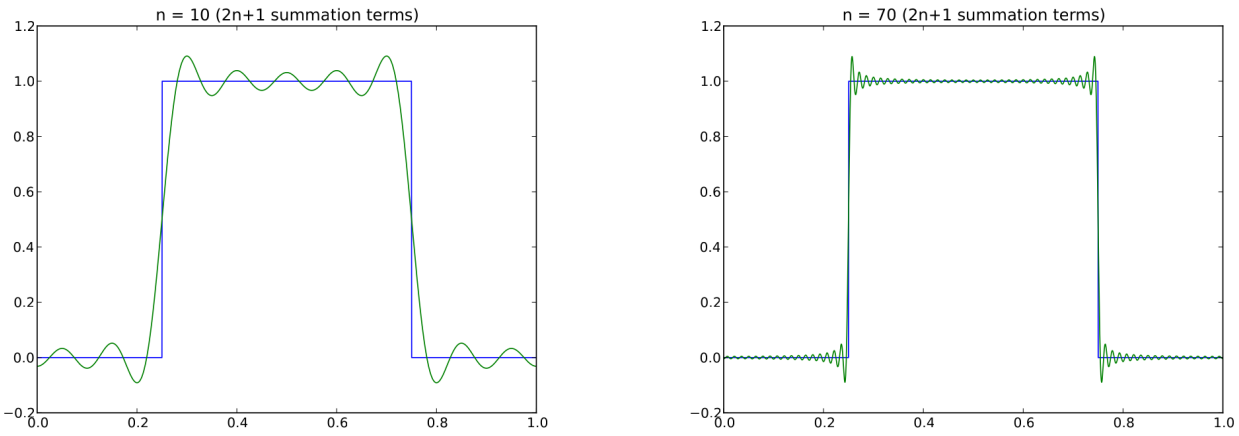


Abbildung 3.1.18: Überschwingungen der Fourier-Reihe der charakteristischen Funktion des Intervalls $[a, b] \subseteq]0, 1[$. (Abbildung aus dem Vorlesungsdokument von Prof. V. Gradinaru, Seite 69)

3.2 Diskrete Fourier Transformation

3.2.1 Motivation

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

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

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

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

Bemerkung 3.2.2: *(Eigenschaften von ω_N)*

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

3.2.2 Konstruktion

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

Definition 3.2.3: *(Trigonometrische Basis)*

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

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

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

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

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

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

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

Definition 3.2.4: *(Fourier-Matrix)*

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

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

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

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

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

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

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

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

3.2.3 DFT in Numpy

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

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

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

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

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

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

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

3.2.4 DFT & Lineare Algebra

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

3.3 Schnelle Fourier Transformation

Da es viele Anwendungen für die Fourier-Transformation gibt, ist ein Algorithmus mit guter Laufzeit sehr wichtig. Während eine naive version des DFT-Algorithmus eine Laufzeit von $\mathcal{O}\left(N^2\right)$ hat, so hat der Fast Fourier Transform Algorithmus nur eine Laufzeit von $\mathcal{O}\left(N \log(N)\right)$, was bei $N = 1024$ bereits eine Laufzeitsverbesserung von $100\times$ mit sich bringt ($\mathcal{O}\left(10\,000\right)$ vs $\mathcal{O}\left(1\,000\,000\right)$ Operationen)! Die untenstehende Abbildung 3.3.3 findet sich, zusammen mit dem Code, mit der sie produziert wurde im Skript auf Seite 86-88

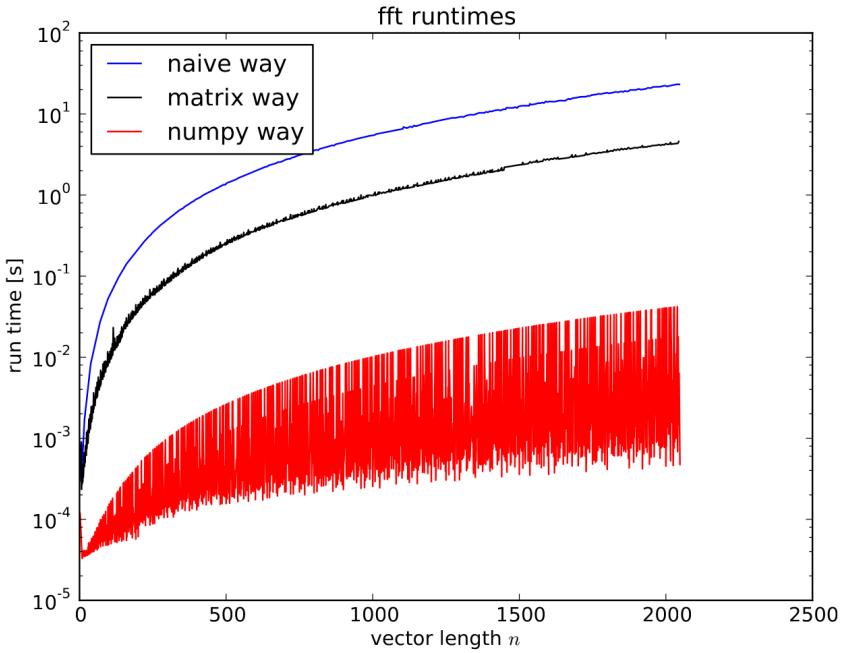


Abbildung 3.3.3: Vergleich der Laufzeit von verschiedenen Fourier-Transformations-Algorithmen. (Abbildung 3.3.3 aus dem Vorlesungsdokument von Prof. V. Gradinaru, Seite 88)

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

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

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

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

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

3.4 Trigonometrische Interpolation

3.4.1 Von Approximation zur Interpolation

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

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

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

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

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

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

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

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

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

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

folgendermassen finden können:

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

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

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

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

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

3.4.2 Zero-Padding-Auswertung

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

In numpy Dieses Verfahren lässt sich in Python leicht via `slices` umsetzen:

```python
def zero_pad(v: np.ndarray, N: int):
    """Apply zero-padding to size N to a vector v """
    n = v.size
    if (N < n): raise ValueError(f"ERROR: Zeropadding for N smaller than vector length: {N} < {n}")

    u = np.zeros(N, dtype=complex)
    u[:n//2] = v[:n//2]
    u[N-n//2:] = v[n//2:]
    return u


def eval_trig_poly(y: np.ndarray, N: int):
    """ Evaluate trig poly generated using y on N points """
    n = y.size
    if (n % 2 != 0): raise ValueError(f"ERROR: y must be of even length, len(y)={n}")

    coeffs = np.fft.fft(y) * 1/n
    coeffs = zero_pad(coeffs, N)
    return np.fft.ifft(coeffs) * N
```

3.4.3 Ableitungen mit Zero-Padding

Mit dem Trick aus Bemerkung 3.2.13 lassen sich auch direkt die Ableitungen berechnen.

In numpy Geht dies direkt durch leichte Modifikation der obigen Funktionen:

```python
def eval_trig_poly_d1(y: np.ndarray, N: int):
    """ Evaluates first der. of trig poly generated using y on N points """
    n = y.size
    if (n % 2 != 0): raise ValueError(f"ERROR: y must be of even length, len(y)={n}")

    coeffs = np.fft.fft(y) * 1/n

    for i in range(0, n//2):
        coeffs[i] *= (2.0j * np.pi * i)
    for i in range(n//2, n):
        coeffs[i] *= (2.0j * np.pi * (i - n))

    coeffs = zero_pad(coeffs, N)
    return np.fft.ifft(coeffs) * N
```

3.5 Fehlerabschätzungen

Konvergenz **Definition 3.5.1**

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

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

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

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

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

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

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

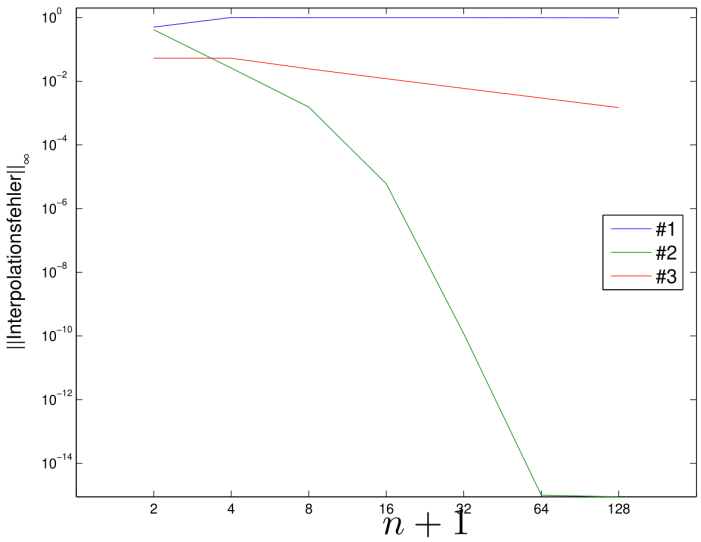


Abbildung 3.5.2: Interpolierungsfehler der Beispiele. Algebraische Konvergenz für (I) und (III), exponentielle für (II). (Abbildung 3.5.2 aus dem Vorlesungsdokument von Prof. V. Gradinaru, Seite 96)

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.

Eine mögliche Aufgabe in der Prüfung kann sein, dass man solchen Fehlergraphen eine Funktion zuordnen muss. Bei der L_2-Norm sind oft die Funktionen von Graphen mit Knicken aufwärts Rationale Funktionen (die Position des Knicks verrät dann den Nenner). Oft sind die Konvergenzfunktionen auch exponentiell bis zum Knick. Graphen mit langsamer linearer Konvergenz sind oft von linearen Funktionen. Graphen mit langsamer Divergenz sind oft von konstanten Funktionen, während Graphen mit Sprüngen nach unten oft von trigonometrischen Funktionen stammen, wobei diese jedoch auch ausserhalb des Graphes liegen können. Dann sind sie an der relativ schnellen Konvergenz zu erkennen.

Beispiel 3.5.4: Sei für $\alpha \in [0,1)$ $f(t) = \dfrac{1}{\sqrt{1-\alpha\sin(2\pi t)}}$. Die Konvergenz ist exponentiell in n und je kleiner α, desto schneller ist sie. In der untenstehenden Abbildung 3.5.5 sind einige Beispiele aufgetragen:

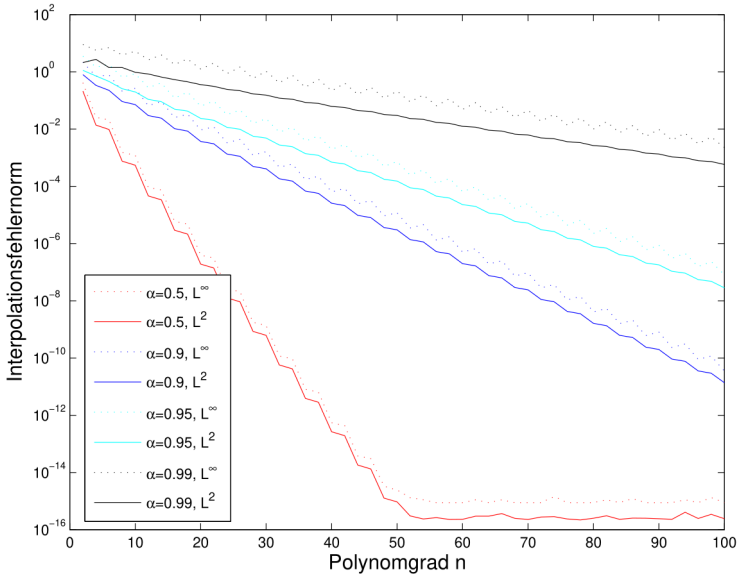


Abbildung 3.5.5: Fehler bei der trigonometrischen Interpolation. (Abbildung 3.5.5 aus dem Vorlesungsdokument von Prof. V. Gradinaru, Seite 98)

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:

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

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

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

Fehler der trigonometrischen Interpolation

Satz 3.5.8

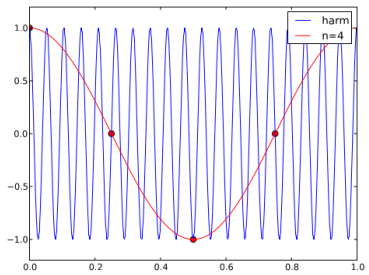
Falls f 1-periodisch ist und die Reihe $\sum_{k\in\mathbb{Z}} |\widehat{f}(k)|$ absolut konvergiert, dann ist der Approximationsfehler definiert als:

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

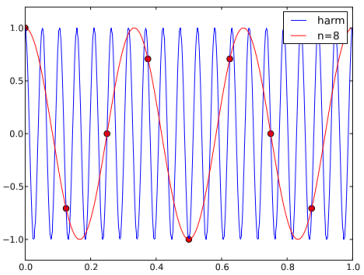
Da durch diesen Satz die obere Schranke für den Approximationsfehler durch die schwer approximierbaren Fourier-Koeffizienten $\widehat{f}(k)$ gegeben ist, heisst das folgendes für die Approximation von Polynomen von Grad $\deg(P(x)) < n$ für unser Approximationspolynom von Grad $\deg(P_N(x)) = n$:

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

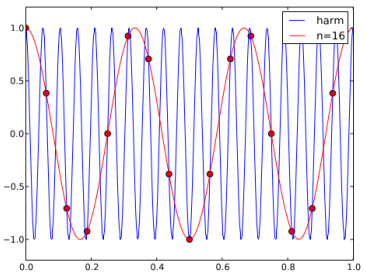
Beispiel: Ein Beispiel aus der Musik: Wir haben ein analoges Signal und wollen es digitalisieren. Wir messen die Spannungswerte in äquidistanten Punkten. Falls wir jedoch die Frequenz der Messung zu niedrig wählen, so kann ein total falsches Interpolationspolynom entstehen, wie in der untenstehenden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



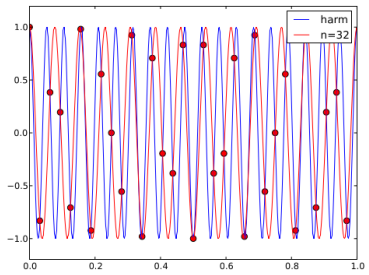
$n = 4$



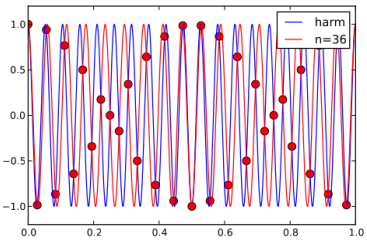
$n = 8$



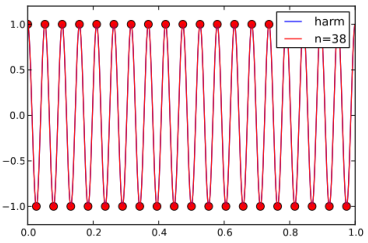
$n = 16$



$n = 32$



$n = 36$



$n = 38$

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)

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

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

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

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

3.6 DFT und Chebyshev-Interpolation

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

Die Chebyshev-Knoten sind folgendermassen definiert:

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

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

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

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

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

Um das ganze zu implementieren ist eine andere Darstellung nützlich:

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

Durch Umformungen (Seite 101 im Skript) erhalten wir:

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

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

Und mit weitern Umformungen erhalten wir

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

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

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

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

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

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

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

4 Stückweise Polynomiale Interpolation

4.1 Stückweise Lineare Interpolation

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

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

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

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

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

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

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

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

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

In numpy Stückweise lineare interpolation lässt sich leicht umsetzen via numpy `piecewise` Funktionen:

```python
def slope(x: np.ndarray, j: int, x_data: np.ndarray, y: np.ndarray):
    """ Slope on j-th sub-interval, for x falling inside that sub-interval """
    h = np.abs(x_data[j] - x_data[j-1])   # size of sub-interval
    return y[j-1]*(x_data[j] - x)/h + y[j]*(x - x_data[j-1])/h


def eval_linear_interp(x: np.ndarray, x_data: np.ndarray, y: np.ndarray):
    """ Evaluate linear interpolation on x, using data points (x_data, y) """
    n = x_data.size; m = x.size

    condlist = [
        (x_data[j-1] <= x) & (x <= x_data[j])
        for j in range(1, n)
    ]
    funclist = [
        lambda x, ind=j: slope(x, ind, x_data, y)
        for j in range(1, n)
    ]

    return np.piecewise(x, condlist, funclist)
```

4.2 Kubische Hermite-Interpolation

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

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

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

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:

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

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

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 höchtens d bestehen und wir der Raum der Splines vom Grad d, oder der Ordnung $(d+1)$ genannt

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

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

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

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

Wobei im Resultatvektor Einträge der Form

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

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

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

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

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

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

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

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

Bemerkung 4.3.8: Bei der "not-a-knot"-Bedingung muss s''' stetig in x_1 und x_{N-1} sein und man braucht 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:

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

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

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 $k = 0, 1, 2, 3$:

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

Zur Erinnerung, C^4 ist die Klasse aller vier (oder mehr) mal ableitbaren Funktionen.

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

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

5 Numerische Quadratur

5.3 Grundbegriffe und -Ideen

Es ist oft nicht möglich oder sinnvoll einen Integral analytisch zu berechnen. Mit Methoden der Quadratur können wir Integrale nummerisch 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 **Definition 5.3.1**

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

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

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

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

Fehler **Definition 5.3.2**

Der Fehler der Quadratur $Q_n(f)$ ist

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

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

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:

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

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:

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

Lagrange-Polynome **Definition 5.3.3**

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 $x_0, x_1, \dots, x_n$

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. $l_i(x_j) = 0 \ \ \forall j \neq i$
2. $l_i(x_i) = 1 \ \ \forall i$
3. $\deg(l_i) = n \ \ \forall i$
4. $\sum_{k=0}^{n} l_k(x) = 1 \ \ \forall x \in \mathbb{R}$
5. $\sum_{k=0}^{n} l_k^{(m)}(x) = 0$ für $m > 0$
6. $l_0, l_1, \ldots, l_n$ 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:

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

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:

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

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 $k = 0, \ldots, N$:

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

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

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

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 $\mathcal{O}\left(h^{n+1}\right)$ und mit $N = (b-a) \div h$ Intervallen der Grösse h haben wir einen globalen Fehler in $\mathcal{O}\left(h^n\right)$. 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 $[-1, 1]$:

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

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

$$\int_a^b f(t) \, \mathrm{d}t \approx \frac{1}{2}(b-a) \sum_{j=1}^{n} \widehat{w}_j \widehat{f}(\widehat{c}_j) = \sum_{j=1}^{n} w_j f(c_j) \qquad \text{mit } \begin{matrix} c_j = \frac{1}{2}(1-\widehat{c}_j)a + \frac{1}{2}(1+\widehat{c}_j)b \\ w_j = \frac{1}{2}(b-a)\widehat{w}_j \end{matrix}$$

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

Untenstehend eine Liste verschiedener Quadraturverfahren (Reminder: Eine Funktion der Orgnung 2 ist eine exakte Approximation einer konstanten oder linearen Funktion):

Eigenschaft	Mittelpunkt	Trapez	Simpson
Knoten	1	2	3
Ordnung	2	2	4
Fehler	$\mathcal{O}\left(h^2\right)$	$\mathcal{O}\left(h^2\right)$	$\mathcal{O}\left(h^4\right)$
Symmetrisch	Ja	Ja	Ja

Mittelpunkt-Regel $Q^M(f;a,b) = (b-a)f\left(\frac{a+b}{2}\right)$. Gewicht: $\omega = b-a$

Trapez-Regel $Q^T(f;a,b) = \frac{b-a}{2}(f(a)+f(b))$. Fehler: $E(n) = \left|-\frac{1}{12}(b-a)^3 f^{(2)}(\xi)\right|$ mit $\xi \in [a,b]$

Fehlerabschätzung: $|E_n| \leq \frac{(b-a)^3}{12}||f''||_\infty$. Gewichte: $\omega_1 = \omega_2 = \frac{b-a}{2}$

Simpson-Regel $Q^S(f;a,b) = \frac{b-a}{6}\left(f(a) + 4f\left(\frac{a+b}{2} + f(b)\right)\right)$. Fehler: $E(n) = \left|-\frac{1}{90}\left(\frac{b-a}{2}f^{(4)}\right)f^{(4)}(\xi)\right|$

Fehlerabschätzung: $|E_n| \leq \left(\frac{b-a}{2}\right)^2 f^{(4)}(\xi)$. Gewichte: $\frac{b-a}{6}, \frac{4(b-a)}{6}, \frac{b-a}{6}$

Bemerkung 5.4.1: Die Schranken für den Fehler erhält man aus den Lagrange-Polynomen vom Grad $n-1$:

$$f \in \mathbb{C}^n([a,b]) \Rightarrow |f(t)\ \mathrm{d}t - Q_n(f)| \leq \frac{1}{n!}(b-a)^{n+1}||f^{(n)}||_{L^\infty([a,b])}$$

5.4.1 Summierte Quadratur

Mit Anwendung von Divide and Conquer kann die Präzision der Integration verbessert werden, man unterteilt dazu einfach das Integrationsintervall in viele kleine Intervalle:

$$\int_a^b f(x)\ \mathrm{d}x = \sum_{i=0}^{N-1}\int_{x_i}^{x_{i+1}} f(x)\ \mathrm{d}x = \sum_{i=0}^{N-1} Q_n(f;x_i,x_{i+1})$$

Im Folgenden ist $h = \frac{b-a}{N}$, $x_0 = a$, $x_i = x_0 + ih$ und $x_N = b$

Summierte Mittelpunkt-Regel $I(f;a,b) \approx \sum_{i=0}^{N-1} h \cdot f\left(\frac{x_i + x_{i+1}}{2}\right)$

Summierte Trapez-Regel $I(f;a,b) \approx \sum_{i=0}^{N-1}\frac{h}{2}(f(x_i)+f(x_{i+1})) = \frac{h}{2}\left(f(a) + 2\sum_{i=1}^{N-1} f(x_i) + f(b)\right)$

Fehler: $E(n) \leq \frac{h^2}{12}(b-a)\max_{x\in[a,b]}|f''(x)|$. Ist exakt bei periodischen, unendlich oft differenzierbaren Funktionen. Untenstehend eine Implementation der Trapez-Regel in Numpy

```
def trapezoidal(f, a, b, N):
    x, h = np.linspace(a, b, int(N) + 1, retstep=True)
    I = h / 2.0 * (f(x[0]) + 2.0 * np.sum(f(x[1:-1])) + f(x[-1]))
    return I
```

Summierte Simpson-Regel

$$I(f;a,b) \approx \frac{h}{6}\left(f(a) + 2\sum_{i=1}^{N-1} f(x_i) + 4\sum_{i=1}^{N} f\left(\frac{x_{i-1}+x_i}{2}\right) + f(b)\right)$$
$$= \frac{\widetilde{h}}{3}\left(f(\widetilde{x}_0) + 2\sum_{i=1}^{N-1} f(\widetilde{x}_{2i}) + 4\sum_{i=1}^{N} f(\widetilde{x}_{2i-1}) + f(\widetilde{x}_{2N})\right) \quad \text{mit } \widetilde{h} = \frac{h}{2},\ \widetilde{x}_{2i} = x_i \text{ und } \widetilde{x}_{2i-1} = \frac{x_{i-1}+x_i}{2}$$

Untenstehend eine Implementation der Simpson-Regel

```python
def simpson(f, a, b, N):
    x, h = np.linspace(a, b, 2 * int(N) + 1, retstep=True)
    I = h / 3.0 * (np.sum(f(x[::2])) + 4.0 * np.sum(f(x[1::2])) + f(x[0]) - f(x[-1]))
    return I
```

5.4.2 Romberg Schema

Für glatte Funktionen haben wir: $T(h) = I[f] + c_1h^2 + c_2h^4 + \ldots + c_ph^{2p} + \mathcal{O}\left(h^{2p+2}\right)$

Die Idee des Romberg-Schemas ist es, die führenden Fehlerterme durch Linearkombinationen zu eliminieren

Schritt 1 Berechnung von $T(h)$ und $T(\frac{h}{2})$:

$$T(h) = I + c_1h^2 + c_2h^4 + \ldots$$
$$T\left(\frac{h}{2}\right) = I + c_1\frac{h^2}{4} + c_2\frac{h^4}{16} + \ldots$$

Schritt 2 Linearkombination zur Elimination des h^2-Terms (Ordnung dann 4):

$$R_{1,1} = \frac{4T(h/2) - T(h)}{3} = I + c_2'h^4 + \ldots$$

Schritt 3 Wiederholen bis zur gewünschten Präzision mit der allgemeinen Rekursionsformel:

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

Der Einfachheit halber können die Terme auch in das sogenannte "Romberg-Tableau" eingefüllt werden: Das Romberg-

k	0	1	2	3
0	$T(h)$	$R_{0,1}$		
1	$T(h/2)$	$R_{1,1}$	$R_{1,2}$	
1	$T(h/4)$	$R_{2,1}$	$R_{2,2}$	$R_{2,3}$
1	$T(h/8)$	$R_{3,1}$	$R_{3,2}$	$R_{3,3}$

Schema konvergiert sehr schnell für glatte Funktionen.

5.4.3 Anwendung

In der Praxis keine Newton-Cotes höherer Ordnung mit äquidistanten Stützpunkten

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 $\langle M, g\rangle = \int_0^1 M(t)g(t)\,\mathrm{d}t$ (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 $\deg(g) \le m-1$ und $M(t) = (t-c_1)\cdot(t-c_2)\cdot\ldots\cdot(t-c_s)$ 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 $\le 2s$

Orthogonale Polynome

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

Orthogonale Polynome — **Satz 5.5.3**

Im Raum $V = \{f : I \to \mathbb{R} \text{ stetig}, \int_a^b |f(x)|^2 w(x)\,\mathrm{d}x \text{ existiert}\}$ existiert eine eindeutige Folge von Polynomen $p_0, p_1, \dots$ mit $p_k(x) = x^k + P(x)$ mit $\deg(P(x)) \le k-1$ für $k \ge 0$, 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 $\beta_{k+1} = \frac{\langle p_k, p_k\rangle}{\langle p_{k-1}, p_{k-1}\rangle}$, wobei hier $\langle f, g\rangle = \int_a^b f(x)g(x)w(x)\,\mathrm{d}x$ das Skalarprodukt ist.

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

$$p_0(x) = 1 \qquad p_1(x) = x$$
$$p_2(x) = \frac{1}{2}(3x^2 - 1) \qquad p_3(x) = \frac{1}{2}(5x^3 - 3x)$$

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 $b = \infty$:

$$p_0(x) = 1 \qquad p_1(x) = 2x$$
$$p_2(x) = 4x^2 - 2 \qquad p_3(x) = 8x^3 - 12x$$

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 (also entweder $c_1 = a$ oder $c_s = b$), 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 (i.e. man setzt $c_1 = a$ und $c_s = b$), und man hat Ordnung $2s - 2$ und wir benötigen nur noch die Knoten $c_2, \dots, c_{s-1}$
- Die Lombatto- und Radau-Quadratur werden häufig bei der Lösung gewöhnlicher DGL verwendet.

Die gefundenen Gewichte und Punkte können dann in die normale Quadraturformel eingesetzt werden (Definition 5.3.1)

Der Fehler der Gauss-Quadratur ist:

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

Und eine obere Schranke für den Fehler ist dann

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

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:

$$b_i = \int_0^1 l_i(t)^2\ \mathrm{d}t$$

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 $x_k = \cos\left(k\frac{\pi}{n}\right)$ 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:

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

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

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

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 $\mathcal{M} := \{a = x_0 < x_1 < \cdots < x_m = b\}$ ist (für $f \in C^2([a,b])$):

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

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

In `numpy` Lässt sich eine simple adaptive Quadratur folgendermassen implementieren. Die Idee ist hierbei einfach das Gitter an Problemstellen zu verdichten, was iterativ die Konvergenz der Quadratur verbessern sollte.

```python
def adaptive_quad(f, M: np.ndarray, rtol: float = 1e-6, atol: float = 1e-10):
    """ Calculate adaptive quad. of f on given grid M"""
    h = np.diff(M)
    midp = 0.5 * ( M[:-1] + M[1:] )
    fx   = f(M); fmid = f(midp)

    # Local quadratures to estimate local errors further down
    trap_local = h/4 * ( fx[:-1] + 2*fmid + fx[1:] )
    simp_local = h/6 * ( fx[:-1] + 4*fmid + fx[1:] )
    Q = np.sum(simp_local)

    # Estimate local & total error
    err_loc = np.abs( simp_local - trap_local )
    err = np.sum(err_loc)
    err_avg = err / len(err_loc)

    # Refine grid in high-error regions, if needed
    if (err > rtol*np.abs(Q) and err > atol):
        refcells = np.nonzero( err_loc > 0.9 * err_avg )[0]      # Find problematic cells
        M_new = np.sort( np.append(M, midp[refcells]) )          # Update grid
        Q = adaptive_quad(f, M_new, rtol, atol, results)         # Try again

    return Q
```

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

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

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

In numpy Lassen sich so die bekannten Verfahren wie die Trapez-Regel oder Simpson-Regel leicht auf höhere Dimensionen anwenden

```python
def trapezoid_2d_mesh(f, a: float, b: float, Nx: int, c: float, d: float, Ny: int):
    """ Trapezoidal rule on a 2d function via np.meshgrid """
    x, hx = np.linspace(a, b, Nx+1, retstep=True)
    y, hy = np.linspace(c, d, Ny+1, retstep=True)

    X, Y = np.meshgrid(x, y)
    F = f(X, Y)

    # Apply once along x-axis, once along y-axis
    Q_x = hx/2 * ( 2.0 * np.sum(F[:, 1:-1], axis=1) + F[:, 0] + F[:, -1] )
    Q_y = hy/2 * ( 2.0 * np.sum( Q_x[1:-1] ) + Q_x[0] + Q_x[-1] )

    return Q_y

def simpson_2d_weights(N: int):
    """ Generate weights for simpson rule """
    w = np.zeros(N+1)
    w[0] = 1.0; w[-1] = 1.0
    for i in range(1, N): w[i] = 2.0 if i % 2 == 0 else 4.0
    return w

def simpson_2d_mesh(f, a: float, b: float, Nx: int, c: float, d: float, Ny: int):
    """ Simpson rule on a 2d function via np.meshrgdi """
    x, hx = np.linspace(a, b, Nx+1, retstep=True)
    y, hy = np.linspace(c, d, Ny+1, retstep=True)

    X, Y = np.meshgrid(x, y)
    F = f(X, Y)
    wx, wy = simpson_2d_weights(Nx), simpson_2d_weights(Ny)
    W = np.outer(wx, wy)

    scale = hx*hy / 3**2
    Q = scale * np.sum( W * F )
    return Q
```

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.

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

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

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

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 $X = [I_N - \widetilde{\sigma}_N, I_N + \widetilde{\sigma}_N]$ und X enthält den wahren Wert $\int_{[0,1]^d} z(t)\ \mathrm{d}t$ 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 Zufallszahlen
- gute deterministische Berechnungen, hier $\widetilde{\sigma}_N$ und I_N
- Darstellung des Ergebnis, hier $\Pr[I \in X] = 0.683$

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

Das Monte-Carlo-Verfahren beruht auf folgendem:

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

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:

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

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

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

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 $\sigma(z(\mathcal{X})) = \sqrt{\mathbb{V}(z(\mathcal{X}))}$:

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

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 ($N \ge 30$) ist.

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

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

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

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

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:

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

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:

```python
import numpy as np
def monte_carlo_integral(func, a, b, N):
    t = np.random.uniform(a, b, N)
    fx = func(t)
    I = np.mean(fx) * (b - a)
    var = np.std(fx, ddof=1) / np.sqrt(N)
    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:

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

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:

```
def control_variate_mc(func, phi, analytic_int_phi, a, b, N):
t = np.random.uniform(a, b, N)
val = func(t) - phi(t)
I = np.mean(val) * (b - a) + analytic_int_phi
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

$$I = \int_D f(x) \, \mathrm{d}x$$

mit der Hilfsdichte $g(x)$ (für welche gilt $\int_D \mathrm{d} = 1$)

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

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

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

und dessen Varianz ist

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

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 $\mathcal{O}\left(N^{-1} \cdot (\log(N))^d\right)$ abnimmt. Für kleine d haben wir ungefähr eine Abnahme in $\mathcal{O}\left(N^1\right)$, 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 $x^{(k)} \to x^*$, 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 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

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

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

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

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):

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

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.

```python
def linear_convergance(x):
    y = []  # container for the x(j)
    for k in range(15):
        x = x + (np.cos(x) + 1) / np.sin(x)  # apply the iteration formula
        y += [x]  # store the value in the container
    err = abs(np.array(y) - x)  # estimation for the error
    rate = err[1:] / err[:-1]
    # estimation for convergence rate
    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	Berechnen bis Toleranz $\varepsilon < \tau$ erreicht	Präzise	Man kennt x^* nicht
Ungefähr gleich	Itaration bis$x^{(k+1)} \approx x^{(k)}$	Keine Voraussetzungen	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 $\phi : D \to D$ 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 $\phi : U \to \mathbb{R}^n$ 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 $\phi : \mathbb{R}^n \to \mathbb{R}^n$ 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, $\phi : I \to \mathbb{R}$ $(m+1)$-mal differenzierbar und $x \in I$. Dann gilt für jedes $y \in I$

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

Lemma 6.3.14: Sei I und ϕ wie in Satz 6.3.13. Sei zudem $\phi^{(l)}(x^*) = 0$ für $l \in \{1, \ldots, m\}$ 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.

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.

In numpy Lässt sich das Intervallhalbierungsverfahren (Bisektion) leicht direkt implementieren:

```python
def bisection_method(f, a: float, b: float, tol=1e-12, maxIter=100):
    """ Bisection method on f using initial bounds a,b """
    if a > b: a, b = b, a
    fa = f(a); fb = f(b)
    if fa*fb > 0: raise ValueError("f(a) & f(b) must have different signs for bisection")

    sgn_fb = 1
    if fa > 0: sgn_fb = -1
    x = 0.5 * (a + b)

    iter = 1
    while (b-a > tol and a<x and x<b and iter<maxIter):
        # Check on which side f(x) is, update bounds
        if sgn_fb*f(x) > 0: b = x
        else:               a = x

        x = 0.5 * (a + b)
        iter += 1

    return x, iter
```

6.5 Newtonverfahren in 1D

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

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

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

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

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

In numpy Ist das Newton-Verfahren mit sehr wenig code implementierbar:

```python
def newton_method(f, df, x: float, tol=1e-12, maxIter=100):
    """ Use Newton's method to find zeros, requires derivative of f """
    iter = 0
    while (np.abs(df(x)) > tol and iter < maxIter):
        x -= f(x)/df(x); iter += 1

    return x, iter
```

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:

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

In numpy Das Sekanten-Verfahren lässt sich im Prinzip ähnlich implementieren wie Newton:

```python
def secant_method(f, x0: float, x1: float, tol=1e-12, maxIter=100):
    """ Use secant method, which approximates the derivative """
    f0 = f(x0)
    for i in range(maxIter):
        fn = f(x1)
        secant = fn * (x1-x0) / (fn - f0)    # Approximate derivative via secant
        x0 = x1; x1 -= secant

        if np.abs(secant) < tol: return x1, i
        else: f0 = fn

    return None, maxIter
```

6.7 Newton-Verfahren in n Dimensionen

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

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

wobei $DF(x^{(k)}) = \left[\frac{\partial F_j}{\partial x_k}(x)\right]_{j,k=1,2,\dots,n}$ 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:

```python
def newton_2d(x: np.ndarray, F, DF, tol=1e-12, maxIter=50):
    """ Newton method in 2d using Jacobi Matrix of F"""
    for i in range(maxIter):
        s = np.linalg.solve(DF(x[0], x[1]), F(x[0], x[1]))
        x -= s
        if np.linalg.norm(s) < tol * np.linalg.norm(x): return x, i
    return x, maxIter
```

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

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

In numpy Das gedämpfte Newton-Verfahren lässt sich mit Funktionen aus `scipy.linalg` implementieren:

```python
def dampened_newton(x: np.ndarray, F, DF, q=0.5, rtol=1e-10, atol=1e-12):
    """ Dampened Newton with dampening factor q """

    lup    = lu_factor(DF(x))          # LU factorization for efficiency, direct works too
    s      = lu_solve(lup, F(x))       # 1st proper Newton correction
    damp   = 1                         # Start with no dampening
    x_damp = x - damp*s
    s_damp = lu_solve(lup, F(x_damp)) # 1st simplified Newton correction (Reuse Jacobian)

    while norm(s_damp) > rtol * norm(x_damp) and norm(s_damp) > atol:
        while norm(s_damp) > (1-damp*q) * norm(s):  # Reduce dampening if step aggresive
            damp *= q
            if damp < 1e-4: return x            # Conclude dampening doesn't work anymore
            x_damp = x - damp*s                 # Try weaker dampening instead
            s_damp = lu_solve(lup, F(x_damp))

        x = x_damp     # Accept this dampened iteration, continue with next proper step

        lup    = lu_factor(DF(x))
        s      = lu_solve(lup, F(x))         # Next proper Newton correction
        damp   = np.min( damp/q, 1 )
        x_damp = x - damp*s
        s_damp = lu_solve(lup, F(x_damp))    # Next simplified Newton correction

    return x_damp
```

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.

Bemerkung 6.9.1: *(Broyden-Update)* Das Broyden-Update ergibt bezüglich der $||\cdot||_2$-Norm die minimale korrektur der Jakobi-Matrix J_k an, so dass die Sekantenbedingung erfüllt ist. Die Implementierung erzielt man folgendermassen mit der ***Sherman-Morrison-Woodbury*** Update-Formel:

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

Das Broyden-Quasi-Newton-Verfahren konvergiert langsamer als das Newton-Verfahren, aber schneller als das vereinfachte Newton-Verfahren. (`sp` ist `Scipy` und `np` logischerweise `Numpy` im untenstehenden code)

```python
def fast_broyden(x0: np.ndarray, F, J, tol=1e-12, maxIter=20):
    x   = x0.copy()
    lup = lu_factor(J)

    s  = lu_solve(lup, F(x))
    sn = np.dot(s, s)
    x -= s

    # Book keeping, for Broyden Update
    dx     = np.zeros((maxIter, len(x)))
    dxn    = np.zeros(maxIter)
    dx[0]  = s
    dxn[0] = sn
    k = 1

    while sn > tol and k < maxIter:
        w = lu_solve(lup, F(x))     # Simplified Newton Update

        # Apply Broyden correction (Shermann-Morrison-Woodbury formula)
        for r in range(1, k):
            w += dx[r] * np.dot(dx[r-1], w) / dxn[r-1]
        z = np.dot(s, w)
        s = (1 + z/(sn-z)) * w
        x -= s      # Apply the iteration

        # Book keeping again
        sn = np.dot(s, s)
        dx[k] = s
        dxn[k] = sn
        k += 1

    return x, k
```

7 Intermezzo: Lineare Algebra

7.1 Grundlagen

Bemerkung 7.1.1: Eine Tabelle mit invertierbaren und nicht invertierbaren Matrizen findet sich unten:

Invertierbar	Nicht Invertierbar
A ist regulär	A ist singulär
Spalten sind linear unabhängig	Spalten sind linear abhängig
Zeilen sind linear unabhängig	Zeilen sind linear abhängig
$\det(A) \neq 0$	$\det(A) = 0$
$Ax = 0$ hat eine Lösung $x = b$	$Ax = 0$ hat unendlich viele Lösungen
$Ax = b$ hat eine Lösung $x = A^{-1}b$	$Ax = b$ hat keine oder unendlich viele Lösungen
A hat vollen Rang	A hat Rang $r < n$
A hat n non-zero Pivots	A hat $r < n$ Pivots
span$\{A_{:,1}, \dots, A_{:,n}\}$ hat Dimension n	span$\{A_{:,1}, \dots, A_{:,n}\}$ hat Dimension $r < n$
span$\{A_{1,:}, \dots, A_{n,:}\}$ hat Dimension n	span$\{A_{1,:}, \dots, A_{n,:}\}$ hat Dimension $r < n$
Alle Eigenwerte von A sind nicht Null	0 ist der Eigenwert von A
$0 \notin \sigma(A) =$ Spektrum von A	$0 \in \sigma(A)$
$A^H A$ ist symmetrisch positiv definit	$A^H A$ ist nur semidefinit
A hat n (positive) Singulärwerte	A hat $r < n$ (positive) Singulärwerte

Definition 7.1.2: *(Orthogonale Vektoren)* Vektoren $q_1, \dots, q_n$ heissen ***orthogonal***, falls

$$q_i^H \cdot q_j = 0 \quad \forall i, j \leq n \text{ with } i \neq j$$

Wenn sie zudem normiert sind (also $||q_i||_2 = 1 \quad \forall i \leq n$), dann heissen sie ***orthonormal***

Bemerkung 7.1.3: In der vorigen Definition wird die ***Euklidische Norm*** $||q||_2^2 = q^H \cdot q$ verwendet

Bemerkung 7.1.7: *(Rotationen)* Die Rotationsmatrix für eine Rotation um Winkel θ ist gegeben durch:

$$R_\theta = \begin{bmatrix} \cos(\theta) & 0 & -\sin(\theta) \\ 0 & 1 & 0 \\ \sin(\theta) & 0 & \cos(\theta) \end{bmatrix}$$

Perturbierte LGS

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

Definition 7.1.18: *(Konditionszahl)* $\text{cond}(A) := ||A^{-1}|| \cdot ||A||$. Manchmal auch mit $\kappa(A)$ notiert

Auch hier gibt es sie wieder für verschiedene Normen:

- $\kappa_2(A) = \frac{\sigma_{\max}(A)}{\sigma_{\min}(A)}$ (Spektralnorm mit Singulärwerten)
- $\kappa_\infty(A) = ||A||_\infty \cdot ||A^{-1}||_\infty$
- $\kappa_1(A) = ||A||_1 \cdot ||A^{-1}||_1$

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

Zudem haben wir folgende Eigenschaften:

- $\kappa(A) \geq 1$
- $\kappa(cA) = \kappa(A) \quad \forall c \neq 0$
- $\kappa(A) = \kappa(A^{-1})$
- Für orthogonale und unitäre Matrizen Q: $\kappa_2(Q) = 1$

Grosse Matrizen

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

Dünnbesetzte Matrizen

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

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

Einfacher zu speichern: `val, col, row` sind Vektoren so dass `val[k]` $= a_{ij}$, wobei $i =$ `row[k]`, $j =$ `col[k]`. (nur $a_{ij} \neq 0$)

Es gibt viele Formate, je nach Anwendung sind 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.

7.1.1 Gauss Elimination / LU Zerlegung

Das Anwenden der Gauss-Elimintation ergibt die LU-Zerlegung, gegeben durch $A \in \mathbb{R}^{n\times m} = PLU$, wobei U eine obere Dreiecksmatrix (die resultierende Matrix der Gauss-Elimintation), L eine untere Dreiecksmatrix (Matrix aller Schritte der Gauss-Elimintation) und P eine Permutationsmatrix ist.

In `numpy` können wir `P, L, U = scipy.linalg.lu(A)` (`Numpy` liefert keine LU-Zerlegung). Mit `scipy.linalg.lu_solve(P, L, U)` kann man dann das System lösen. Jedoch ist dies nicht sinnvoll, wenn wir die Dreiecksmatrizen gar nicht benötigen. In diesem Fall verwenden wir einfach `numpy.linalg.solve(A)`

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

Cholesky Zerlegung (A ist positiv defefinit und hermetisch)

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

Diese Zerlegung kann $Ax = b$ potenziell schneller lösen als LU und wir verwenden nur halb so viel Speicher. Zudem ist keine Pivotierung nötig, also ist das Verfahren für symmetrisch positiv definite Matrizen numerisch stabil.

Im Folgenden ist der Cholesky algorithmus in Pseudocode beschrieben:

Algorithm 1 CHOLESKY(A)

$n \leftarrow$ `A.shape[0]`
$l \leftarrow$ Initialisiere ein $n \times n$ array
for $j = 1, 2, \dots, n$ **do**
    $l_{jj} \leftarrow \sqrt{A_{jj} - \sum_{k=1}^{j-1} l_{jk}^2}$
    for $i = j+1, \dots, n$ **do**
        $l_{ij} \leftarrow \frac{1}{l_{jj}}\left(A_{ij} - \sum_{k=1}^{j-1} l_{ik}l_{jk}\right)$
return l

In `numpy` haben wir via `scipy.linalg` die Funktionen `cholesky`, `cho_factor` und `cho_solve`, wie auch bereits äquivalent für die LU-Zerlegung

7.1.2 QR-Zerlegung

Wir können eine Matrix $A \in \mathbb{R}^{m\times n}$ mit $m \geq n$ als $A = QR$ zerlegen, wobei $Q \in \mathbb{R}^{m\times n}$ orthonormale Spalten besitzt und $R \in \mathbb{R}^{n\times n}$ ist eine obere Dreiecksmatrix.

Die QR-Zerlegung ist numerisch stabiler bei schlecht konditionierten Problemen, ist die Basis vieler Eigenwertverfahren und ist ideal für Least Squares.

Wir können mit der QR-Zerlegung auch LGS $Ax = b$ lösen:

$$Ax = b \iff QRx = b \iff Rx = Q^\top b$$

Da Q orthogonal ist, haben wir $Q^{-1} = Q^\top$. Um das Ganze einfacher zu machen, lösen wir das System $Rx = y$, wobei $y := Q^\top b$. In numpy können wir direkt mit `np.linalg.solve()` dies lösen (nutzt automatisch Rückwärtssubstitution)

Givens-Rotations

Bei der Givens-Rotation generiert man eine Rotationsmatrix, die die (i,j)-Ebene um einen Winkel θ rotiert. Die dazu konstruierte Matrix hat dabei die folgende Form (rechts eine Beschreibung des Eintrags (k,l)):

$$G(i,j,\theta) = \begin{bmatrix} 1 & 0 & & \cdots & & & 0 \\ 0 & \ddots & & & & & \\ & & c & \cdots & s & & \\ \vdots & & \vdots & & \vdots & & \vdots \\ & & -s & \cdots & c & & \\ & & & & & \ddots & \\ 0 & & & \cdots & & & 1 \end{bmatrix} \quad \text{oder } G(i,j,\theta)_{k,l} = \begin{cases} k = l \land k \neq i,j & 1 \\ k = l \land (k = i \lor k = j) & c \\ k = i \land l = j & -s \\ k = j \land l = i & s \\ \text{sonst} & 0 \end{cases}$$

Dabei ist $c = \cos(\theta)$ und $s = \sin(\theta)$. Diese Matrix hat einige nützliche Eigenschaften: $G^\top G = I$ (also ist G orthogonal), also gilt auch $G^{-1} = G^\top$ und G modifiziert nur Zeilen i und j

Im Zweidimensionalen Raum können wir die Werte für c und s so bestimmen:

$$\begin{bmatrix} c & s \\ -s & c \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix}$$

$$r = \sqrt{a^2 + b^2}, \quad c = \frac{a}{r}, \quad s = \frac{b}{r}$$

Wir haben jedoch das Problem, dass die Berechnung von r überlaufen kann. Dies lösen wir, indem wir skalieren:

Falls $|b| > |a|$:

$$t = \frac{a}{b}, \quad s = \frac{1}{\sqrt{1+t^2}}, \quad c = s \cdot t$$

Falls $|a| \geq |b|$:

$$t = \frac{b}{a}, \quad s = c \cdot t, \quad c = \frac{1}{\sqrt{1+t^2}}$$

Es ist wichtig, dass wir das $r = \text{sign}(a)\sqrt{a^2+b^2}$ mit Vorzeichen berechnen, um Auslöschung zu vermeiden

Man kann nun mit der Givens-Rotation die QR-Zerlegung durchführen:

Algorithm 2 GivensQRDecomposition(A)

```
m ← A.shape[0]
n ← A.shape[1]
q ← Initialisiere ein n × n array
for j = 1, 2, ..., n do
    for i = m, ..., 2 do
        Nullsetze a_{m,j}, a_{m-1,j}, ..., a_{2,j} durch Givens-Rotationen G_{m,j}, G_{m-1,j}, ..., G_{2,j}
return l
```

Gram-Schmidt

Die Idee des Gram-Schmidt-Algorithmus ist es, orthonormale Vektoren zu konstruieren und diese dann zur Matrix Q zusammenzubasteln.

Es wurden zwei Algorithmen behandelt, beide unten in Pseudocode & Python:

Algorithm 3 CLASSICALGRAMSCHMIDT(A)

$n \leftarrow$ `A.shape[0]`
$q \leftarrow$ Initialisiere ein $n \times n$ array
$r \leftarrow$ Initialisiere ein $n \times n$ array
for $k = 1, 2, \dots, n$ **do**
  $v_k \leftarrow a_k$ ▷ *Der k-te Spaltenvektor*
  for $i = 1, \dots, k-1$ **do**
    $r_{ik} \leftarrow q_i^\top a_k$
    $v_k \leftarrow a_k - \sum_{i=1}^{k-1} r_{ik} q_i$
  $r_{kk} = ||v_k||$
  $q_k = v_k / r_{kk}$ ▷ *Vektor normieren*
return q, r

```python
def gram_schmidt(A: np.ndarray):
    """ Regular Gram-Schmidt, assumes lin. indep. columns in A """
    m, n = A.shape
    Q, R = np.zeros((m, n)), np.zeros((m, n))

    for j in range(0, n):
        Q[:, j] = A[:, j]
        for i in range(0, j):
            proj = np.dot(A[:, j], Q[:, i])
            R[i, j] = proj
            Q[:, j] = Q[:, j] - proj * Q[:, i]
        Q[:, j] = Q[:, j] / np.linalg.norm(Q[:, j], 2)

    return Q, R
```

Die modifizierte Variante ist deutlich stabiler.

Algorithm 4 MODIFIEDGRAMSCHMIDT(A)

```
n ← A.shape[0]
q ← Initialisiere ein n × n array
r ← Initialisiere ein n × n array
for k = 1, 2, ..., n do
    v_k ← a_k                                   ▷ Der k-te Spaltenvektor
    for i = 1, ..., k − 1 do
        r_ik ← q_i^⊤ v_k
        v_k ← v_k − r_ik q_i
    q_k = v_k / r_kk                            ▷ Vektor normieren
return q, r
```

```python
def gram_schmidt_mod(A: np.ndarray):
    """ Gram-Schmidt in place, assumes lin. indep. columnes in A  """
    m, n = A.shape
    Q, R = np.zeros((m, n)), np.zeros((m, n))

    for i in range(1, n):
        norm = np.linalg.norm(A[:, i])
        Q[:, i] = A[:, i] / norm

        for j in range(i+1, n):
            proj = np.dot(A[:, j], Q[:, i])
            A[:, j] = A[:, j] - proj * Q[:, i]
            R[i, j] = proj

        R[i, i] = np.dot( Q[:, i], A[:, i] )

    return Q, R
```

Falls R nicht benötigt wird, kann viel Speicher gespart werden, indem man das r_{ik} als eine scoped variable verwendet.

Householder-Reflektor

Wir konstruieren eine Matrix $H = I - 2\frac{vv^\top}{v^\top v} = I - 2uu^\top$ mit $u = \frac{v}{||v||}$.

Dabei ist die Matrix H orthogonal ($H^\top H = I$) und symmetrisch ($H^\top = H$).

Um nun die QR-Zerlegung durchzuführen mit der Householder-Reflektion fehlen uns die Householder-Reflektoren. Um diese zu erstellen wollen wir das v so wählen, dass $Hx = ||x||e_1$ gilt. So werden also $m-1$ Elemente auf einmal auf Null gesetzt.

Der Ansatz dazu ist entsprechend $v = x - \alpha e_1$ mit $\alpha = -\text{sign}(x_1)||x||$ (minus, um numerische Stabilität zu erhalten) und wir haben dann:

$$Hx = \alpha e_1 \iff Hx = x - 2\frac{v^\top x}{v^\top v}v$$

Dann müssen wir nur noch $v^\top x$ und $v^\top v$ berechnen und auflösen. Der vollständige QR-Algorithmus lautet:

Algorithm 5 HouseholderQR(A)

```
n ← A.shape[0]
H ← Initialisiere ein n × n array
for k = 1, 2, ..., n do
    x ← A(k : m, k                          ▷ Wähle subvektor der k-ten Spalte
    H_k ← Konstruiere Householder-Reflektor für x
    A(k : m, k : n) ← H_k A(k : m, k : n)   ▷ Update
Q ← H_1 H_2 ··· H_n
R ← H_n H_{n-1} ··· H_1 A
return Q, R
```

Die Laufzeiten der verschiedenen Methoden im Vergleich:

- Householder-QR: $\approx 2mn^2$ Flops
- Gram-Schmidt: $\approx 2mn^2$ Flops
- Givens: $\approx 3mn^2$ Flops

Jedoch ist die Householder-Methode bedeutend stabiler als die anderen beiden.

7.1.3 Singulärwertzerlegung

Satz 7.1.35: Jede Matrix $A \in \mathbb{C}^{m\times n}$ kann in unitäre Matrizen $U \in \mathbb{C}^{m\times m}$ und $V \in \mathbb{C}^{n\times n}$ und die Diagonalmatrix $\Sigma = \text{diag}(\sigma_1, \dots, \sigma_p) \in \mathbb{C}^{m\times n}$, wobei $p = \min\{m, n\}$ und $\sigma_1 \geq \dots \geq \sigma_p \geq 0$, wobei σ_i der i-te Eigenwert ist, so dass

$$A = U\Sigma V^H$$

Die PCA (principal component analysis, zu Deutsch Hauptkomponentenanalysis) setzt sich zum Ziel, die Menge an Informationen so zu reduzieren, so dass nur das Notwendige übrig bleibt.

Idealerweise sind die Daten frei von jeglichem Rauschen:

$$a_j \approx \text{span}\{u_1, \dots, u_p\} \text{ mit } p \ll n$$

In der Realität haben wir jedoch oft ein Rauschen in den Daten:

$$a_j = \sum_{i=1}^{p} \sigma_i u_i v_{ij} + \text{ kleine Störungen}$$

Die PCA versucht nun das p zu bestimmen und die orthonormalen Trendvektoren $u_1, \dots, u_p$ zu finden. Die Spalten von U aus der SVD sind genau die gesuchten Trendvektoren (können geordnet werden nach den zugehörigen Singulärwerten):

$$A_{:,j} \approx \sum_{i=1}^{p} \sigma_i u_i v_{ij}$$

Hierbei ist A eine Datenmatrix, bei welcher die Spalten die Datenpunkte oder Messungen sind und die Zeilen die verschiedenen Merkmale oder Zeitpunkte in den Messungen sind.

Die Matrix V^H enthält in ihrer j-ten Spalte die Gewichte, die die p-Trends zum j-ten Datenpunkt beitragen.

Die ersten p Komponenten erfassen ca. $\frac{\sum_{i=1}^{p} \sigma_i^2}{\sum_{i=1}^{m} \sigma_i^2}$

Varianzkriterium $p = \min \left\{ q : \sum_{j=1}^{q} \sigma_j^2 \geq \varepsilon \sum_{j=1}^{\min\{m,n\}} \sigma_j^2 \right\}$. Oft wird $\varepsilon = 0.90$ verwendet (oder $\varepsilon = 0.95$, dies ist jedoch konservativ, also kann es sein, dass es mehr Komponenten benötigt). $\varepsilon \in (0, 1)$

In `numpy` können wir sowohl die vollständige Singulärwertzerlegung durchführen, wie auch nur die Singulärwerte berechnen.

- Zur vollständigen Berechnung nutzen wir `numpy.linalg.svd` (Option `full_matrices=False` führt eine sparsamere Version der SVD durch) oder `scipy.linalg.svd`,
- Falls wir nur die Singulärwerte benötigen, dann liefert `scipy.linalg.svdvals` eine günstigere Alternative.

Algorithm 6 $\mathrm{PCA}(A, \varepsilon)$

$n \leftarrow$ `A.shape[0]`
$m \leftarrow$ `A.shape[1]`
$U, \Sigma, V^H \leftarrow$ Singulärwertzerlegung von A
$p \leftarrow$ berechnet wie oben mit Varianzkriterium
$U_p \leftarrow U_{:,1:p}$ ▷ *Wähle erste p Spalten von U*
return U_p, Singulärwerte $\sigma_1, \ldots, \sigma_p$

```python
import numpy as np

# SVD of the data matrix
U, sigma, Vh = np.linalg.svd(A, full_matrices=False)
threshold = 0.90 # Threshold for variance (e.g. 90%, the epsilon as discussed previously)
total_var = np.sum(sigma**2)
cumsum = np.cumsum(sigma**2)
p = np.argmax(cumsum >= threshold * total_var) + 1
U_p = U[:, :p] # Primary components
scores = A.T @ U_p # Projection of the data
```

A word of caution:

- Zu niedriges ε kann zu Informationsverlust führen
- Zu hohes ε (e.g. $\varepsilon = 0.99$) kann zu overfitting führen

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 $\min_{\widehat{x}\in\mathbb{R}^n} ||A\widehat{x}-b||^2$ und als Summe:

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

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)* $A^HAx = A^Hb$

Bemerkung 8.1.10: A^HA ist Hermite-Symmetrisch, und falls A vollen Rank hat, dannn ist A^HA positiv-definit und die Normalengleichung hat eine eindeutige Lösung. Jedoch ist die Normalengleichung schlecht konditioniert (es gilt: $\text{cond}(A^HA) = \text{cond}(A)^2$). 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^HA berechnen:

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

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 (Zur Erinnerung, κ ist die Konditionszahl der Matrix).

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 $A = QR = Q\begin{bmatrix}\widetilde{R} \\ 0\end{bmatrix}$. Dann, nach Umformungen, erhalten wir $||Rx-\widetilde{b}||_2^2$ mit $\widetilde{b} = Q^Hb$.

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:

$$||Ax-b||_2 = ||U\Sigma V^Hx - b||_2 = ||\Sigma V^Hx - U^Hb||_2$$

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

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

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.

```python
import numpy as np

A = np.array([[98.269, 1.0], [0.0, 1.0], [-194.96, 1.0]])
b = np.array([852.7, 624.5, 172.7])

def least_squares_svd(A, b, epsilon=1e-6):
    U, s, Vh = np.linalg.svd(A)
    r = 1 + np.where(s / s[0] > epsilon)[0].max()  # numerical rank
    y = np.dot(Vh[:r, :].T, np.dot(U[:, :r].T, b) / s[:r])
    return y

# qr-decomposition:
def least_squares_qr(A, b):
    Q, R = np.linalg.qr(A)
    b_tilde = np.dot(Q.T, b)
    return np.linalg.solve(R, b_tilde)

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 $\widehat{A}\widehat{x} = \widehat{b}$, welches so nah wie möglich am ursprünglichen System liegt und so für welches gilt $\widehat{b} \in \text{Bild}(\widehat{A})$.

Wir versuchen also die folgende Norm zu minimieren:

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

Das Problem lässt sich umschreiben als

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

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

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

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 $H_\Phi(x) := D((grad)(\Phi(x)))$ 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:

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

```python
import numpy as np

def gauss_newton(start_vec, Func, Jacobian, tolerance):
    # Start vector has to be chosen intelligently
    s = np.linalg.lstsq(Jacobian(start_vec), Func(start_vec))[0]
    start_vec = start_vec - s
    # now we perform the iteration
    while np.linalg.norm(s) > tolerance * np.linalg.norm(start_vec):
        # every time we update x by subtracting s, found with the least square method
        s = np.linalg.lstsq(Jacobian(start_vec), Func(start_vec))[0]
        start_vec = start_vec - s
    return start_vec
```

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

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

```python
import numpy as np

t = np.arange(0, 30, 5); n = len(t)
curr_heating = np.array([24.34, 18.93, 17.09, 16.27, 15.97, 15.91])
curr_cooling = np.array([9.66, 18.8, 22.36, 24.07, 24.59, 24.91])
# define the functions that have to be minimized
F_1 = lambda a: a[0] + a[1] * np.exp(-a[2] * t) - curr_heating
F_2 = lambda a: a[0] - a[1] * np.exp(-a[2] * t) - curr_cooling

# define the corresponding Jacobi matrices
def J_1(a):
    mat = np.zeros((n, 3))
    for k in range(n):
        mat[k, 0] = 1.0
        mat[k, 1] = np.exp(-t[k] * a[2])
        mat[k, 2] = -t[k] * a[1] * np.exp(-t[k] * a[2])
    return mat

def J_2(a):
    mat = np.zeros((n, 3))
    for k in range(n):
        mat[k, 0] = 1.0
        mat[k, 1] = -np.exp(-t[k] * a[2])
        mat[k, 2] = t[k] * a[1] * np.exp(-t[k] * a[2])
    return mat

# guess starting vector
x_1 = np.array([10.0, 5.0, 0.0])
x_2 = np.array([30.0, 10.0, 0.0])

# use the Gauss-Newton algorithm declared above
a_1 = gauss_newton(x_1, F_1, J_1, tolerance=10e-6)
a_2 = gauss_newton(x_2, F_2, J_2, tolerance=10e-6)
print("Heating ", a_1)
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 $||F(x^{(k)})+DF(x^{(k)})||^2+\lambda||s||_2^2$ (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`

9 Introduction to Python

Python is a high-level dynamically and strongly typed, multi-paradigm, interpreted programming language. Its syntax might remind you of pseudocode, which allows very quick writing, but lacks some control that other lower level programming languages might offer. Be aware that Python likes to call things differently (`try ... except` for example instead of `try ... catch` or `True, False` instead of `true, false`).

9.1 Basics

9.1.1 Variables

While Python supports many different paradigms (thus making it multi-paradigm), Python still is first and foremost an object oriented programming language and all variables are just references to objects.

What this means is that the variables aren't *technically* conventional variables, but rather pointers and whenever you reassign a value, the object is actually deleted and a new object is created in its place and the variable's reference is updated. This is one of the reasons as to why Python is so slow.

Assigning and initializing variables uses the same syntax and you cannot have unassigned variables:

```
x # This will not work
x = 1 # Notice that there is no type annotated (Python is dynamically typed)
x = 10 # Assign another value to x
x = "Hello World" # Is actually possible, due to dynamic typing
x += 10 # This will cause a TypeError
arr = [] # Creates an empty list. They are dynamically sized
d = {} # Creates an empty dict (Dictionary)
arr_list = [ "Hello", "World", 1 ] # Can contain multiple different elements
```

Python supports the same data types as most other programming languages, but gives some of them funny names (`bool`, `int`, `float`, `str`, `List`, `Dict` (`Map` in Java), `None` (`void` in basically all other languages))

To cast a type in python, the basic types are callable, i.e. to cast an `int` to a `str`, do `str(my_int)`

9.1.2 Operations

Python uses the normal operators for basic arithmetic operations, however be aware that Python implicitly casts `int` to `float` when dividing. To do integer division use the `a // b` syntax. For exponentiation, Python supports the `a ** b` syntax (which computes a^b). Increment and decrement operators are not supported, however `+=`, etc are supported.

9.1.3 Control flow

Python uses indents (that are consistent, i.e. always use `n` spaces or a tab character) to indicate blocks. You cannot use curly braces. In other ways though, Python is fairly lenient, i.e. you are allowed to write parenthesis around the if statement's condition

```
# Can also use range(start, stop, step) for a traditional for-loop.
# Parameters start and step can be both (or just one of them) omitted.
for i in iterable:
    print(i)

for i, val in enumerate(iterable): # Get index and value (or key and value)
    print(i, val)

while True:
    break # Break statement to exit loop

if x > 1: # All blocks start with :
```

```
13      print(1)
14  elif x > 0:
15      print(2)
16  else:
17      print(3)
```

9.1.4 Imports

Python has multiple ways of importing other libraries and from your own files. Your own imports are always relative to the run file (i.e. not to the current file, but the root file of either the library or your program).

```
1  import lib.mylib # Local import from file lib/mylib.py
2  import numpy # Import numpy. Access numpy functions using numpy.<method>(...)
3  import numpy as np # Alias numpy to np. Access using np.<method>(...)
4  from numpy import array # Only import array method from numpy. Call using array(...)
5  # The below is the biggest Python sin. DO NOT DO THIS!
6  from numpy import * # Wildcard import (import all functions from the library as <method>(...))
```

What you should always refrain from doing is a wildcard import. Every function from that library is then imported directly and can be called using `<method>(...)`. Thus, if two libraries have a function that has the same name, your program will stop working, thus always either import a specific function, or better still, use the options on line 2 or 3 for library imports, as it is also much easier for other people to understand where the function is defined.

9.2 Numpy

Numpy is a library that, according to its website, is "The fundamental package for scientific computing with Python".

If you prefer to read an online guide, the official quick start guide is excellent.

9.2.1 Arrays

The heart of numpy is the `numpy.ndarray` class (it has the alias `numpy.array`). As the name would imply, it can be any dimension you would like. The most important attributes for this course are: `shape` (returns a tuple that indicates the number of elements for each dimension), `dtype` (indicates the data type of elements in the array), `ndim` (the number of dimensions, equal to `len(shape)`)

To create an array, we can use a few functions:

- `np.array([...])`
- `np.zeros(shape)`
- `np.zeros_like(arr)`
- `np.ones(shape)`
- `np.ones_like(arr)`
- `np.empty_like(arr)`
- `np.arange(start, stop, step)`
- `np.linspace(start, stop, num_el)`
- `np.logspace(start, stop, num_el)`
- `np.eye(N, M)` (`N` is rows, `M` is cols)
- `np.identity(N, M)`
- `np.fromfunction(f, (dim1, ...))`

To use complex numbers, we can write `a + b * 1.j`

Another useful method is `np.meshgrid(x, y, ...)`, which returns a coordinate grid and treats the input vectors as `x`, `y`, etc coordinates of point `i`.

Arrays aren't usually copied, but you get a view. The `reshape` method below is an example of a case where you get a shallow copy (that still technically is a view). The values are still in the same object, but you get access to it in a different shape. To deep copy an array (i.e. create a new array), use `arr.copy()`

9.2.2 Operations

The same basic operations that Python supports are also supported by numpy, though they are executed on each element.

- You can subtract a number from an ndarray
- You can subtract an ndarray from another ndarray (vector, matrix, ... difference)
- To compute a matrix product (or matrix-vector product), you can do `A @ B` or `np.dot(A, B)`
- `arr.sum()` sums up all elements and `arr.sum(axis=n)` sums up all elements on that axis (0 is column, 1 is row)
- `arr.cumsum()` computes the cumulative sum (i.e. each element in the output is the sum of all preceding elements). You can also use the axis argument again.
- `np.exp`, `np.sqrt`, etc operate element-wise on the array
- `np.where(condition, arr_true, arr_false)` returns a numpy array where if `condition` is true, element `i` is chosen from `arr_true`, else from `arr_false`

A useful trick to create a mask is to use `a < b` (or any other comparison), as that will return an array of booleans.

For piecewise interpolation, a useful method is `np.searchsorted(arr.flat, vals_insert, side='right')`, where `vals_insert` are the values to be inserted and the `side` argument indicates on which side of the match they are to be inserted.

For that though, the array needs to be sorted, for which we can use `np.argsort(arr, axis=-1)`. This will return the indices in the order that would sort the array along the given axis. If axis is unspecified, `-1` is used. To use these indices to sort an array, we can simply use `arr[np.argsort(arr)]`.

Slicing and indexing works just as in Python (assume `a` is a numpy array):

```
a = np.arange(10)
print(a[2]) # Outputs 2 (third element)
print(a[-1]) # Outputs 9 (last element)
print(a[-2]) # Outputs 8 (2nd to last element)
print(a[2:5]) # Outputs [2, 3, 4] (elements 2, 3, and 4)
print(a[2:5:-1]) # Outputs [4, 3, 2] (reversed)
print(a[::-1]) # Reverses the array
```

So, the basic syntax for slicing is `a[start : stop : step]` and any of them can be omitted, though the corresponding colons cannot be omitted if you omit start.

We can also iterate over a numpy array normally (the iterator variable will be the `i`-th element of the outermost array of the numpy array). To iterate over all elements of the array (i.e. the actual data values), we can use `arr.flat` to get a view (similar to a reference, not copied) that has length that corresponds to the sum of all elements of `arr.shape`.

For `n`-d arrays, we can use `a[0, 1]` to access element `a[0][1]`, which is more efficient.

9.2.3 Shape manipulation

We can reshape an array by using `arr.reshape(dim1, dim2, ...)`. This however returns the array with modified shape, whereas `arr.resize((dim1, dim2, ...))` modifies the array directly (notice that here we have to pass a tuple!)

To get a one-dimensional array, we can use `arr.ravel()`, after which the array looks the same as `arr.flat`, but the change is permanent

Stacking arrays

`np.vstack((a, b))` adds array `b`'s elements to array `a` (i.e. stacks along `axis=0`). `np.hstack((a, b))` adds array `b`'s elements to the inner arrays of `a` (i.e. stacks along `axis=1`). `np.concatenate((a, b, ...), axis=n)` does as the above, but applies it to axis `n`

Splitting arrays

`np.hsplit(a, count)` splits `a` into `count` arrays (along `axis=0`) `np.vsplit(a, count)` splits `a` into `count` arrays (along `axis=1`) `np.array_split(a, count, axis=n)` splits `a` into `count` arrays (along `axis=n`)

9.3 Scipy

Scipy is a library that, according to its website, provides "Fundamental algorithms for scientific computing in Python". In this course it is primarily used for cases where Numpy either does not have an implementation or SciPy's implementation is faster.

Scipy needs Numpy arrays as inputs in most cases, thus not requiring any additional syntax.

You can find the scipy reference docs here

9.4 Sympy

Sympy can be used to do symbolic operations, i.e. similar to what we would do in Analysis or the like.

You usually want to follow something like the below code:

```
import sympy as sp

a, b = (0, 1);
x, y = sp.symbols("x, y")  # Create sympy symbols
f = x**2 + 3 * x - 2  # Create your function
# In the below, for 1D differentiation, we can omit the symbol and just use sp.diff(f)
df = sp.diff(f, x)  # Add more arguments for derivatives in other variables
F = sp.integrate(f, x)  # Integrates the function analytically in x
F = sp.integrate(f, (x, a, b))  # Indefinite integral in x in interval [a, b]
lambda_f = sp.lambdify(x, F)  # Creates a lambda function of F in variable x
lambda_f = sp.lambdify([x, y], F)  # Creates a lambda function of F in variables x and y
roots = sp.roots(f)  # Computes the roots of function f analytically
sp.hermite_poly(5)  # Creates hermite poly of degree 5
sp.chebyshevt_poly(5)  # Creates chebychev T (first kind) poly of degree 5
sp.chebyshevu_poly(5)  # Creates chebychev U (second kind) poly of degree 5
```