TUM NumProg, WiSe 2022/2023

Mitschriften basierend auf der Vorlesung von Dr. Hans-Joachim Bungartz

Zuletzt aktualisiert: 28. Februar 2023

Introduction

About

Hier sind die wichtigsten Konzepte der NumProg Vorlesung von Dr. Hans-Joachim Bungartz im Wintersemester 2022/2023 zusammengefasst.

Die Mitschriften selbst sind in Markdown geschrieben und werden mithilfe einer GitHub-Action nach jedem Push mithilfe von Pandoc zu einem PDF konvertiert.

Eine stets aktuelle Version der PDFs kann über https://manuellerchner.github.io/numprog/numerisches_programmieren_IN0019_WS2223_merge.pdf heruntergeladen werden.

Implementation

Außerdem befindet sich eine Implementation von verschiedenen Algorithmen im Ordner `/algorithms` auf GitHub. Diese sind in Python und unter der Verwendung von NumPy geschrieben.

How to Contribute

1. Fork this Repository
2. Commit and push your changes to **your** forked repository
3. Open a Pull Request to this repository
4. Wait until the changes are merged

Contributors

Inhaltsverzeichnis

Floating-Point Arithmetik

Fixed-Point

Bei Fixed-Point wird die Zahl in eine ganze Zahl und eine Bruchzahl aufgeteilt. Diese werden jeweils "normal" kodiert.

- Nachteile:
  - Schneller Overflow, da nur kleine Zahlen dargestellt werden können
  - Konstanter Abstand zwischen zwei Zahlen. Oft nicht benötigt.

Floating-Point

Eine Floating-Point Zahl wird in Mantisse und Exponent aufgeteilt. Zusammen mit einem Vorzeichenbit, lässt sich so ein sehr großer Wertebereich darstellen.

- Definition normalisierte, t-stellige Float-Zahl
- $\mathbb{F}_{B,t} = \{M \cdot B^E \mid M, E \in \mathbb{Z} \wedge \ M \text{ ohne führende Nullen bzw: } B^{t-1} \leq M < B^t\}$
- $\mathbb{F}_{B,t,\alpha,\beta} = \{M \cdot B^E \mid M, E \in \mathbb{Z} \ \ \& \quad M \text{ ohne führende Nullen} \wedge \alpha \leq E < \beta\}$
- Wobei gilt:
- B Basis
  - t Anzahl der signifikanten Stellen
  - α kleinster möglicher Exponent
  - β größter möglicher Exponent
- Vorteile:
  - Großer Wertebereich, da variable Abstände zwischen zwei Zahlen

Kleinste bzw. größte Zahl

In einem solchen System ist:

- $\sigma = B^{t-1} \cdot B^{\alpha}$ die kleinste positive Zahl, die dargestellt werden kann.
- $\lambda = (B^t - 1) \cdot B^{\beta}$ die größte positive Zahl, die dargestellt werden kann.

Beispiel:

- Mit $B = 10$ und $t = 4$ und $\alpha = -2$ und $\beta = 1$ ergibt sich:
  - $\sigma = 10^{4-1} \cdot 10^{-2} = 10$
  - $\lambda = (10^4 - 1) \cdot 10^1 = 99990$

Maximalen relativen Abstand zweier Float-Zahlen

Die Auflösung einer Float-Zahl ist der maximale relative Abstand zu einer anderen Float-Zahl. Sie berechnet sich wie folgt:

- $\varrho = \frac{1}{M} \leq B^{1-t}$

Beispiel:

- Mit einer Basis von $B = 2$ und $t = 4$ Stellen ergibt sich; $\varrho \leq 2^{-3} = 0.125$. Damit ist der maximale relative Abstand zwischen zwei Float-Zahlen 0.125.

Rundung

Da nur eine endliche Anzahl von Float-Zahlen existieren, muss grundsätzlich nach jeder Operation gerundet werden.

Diese Rundungsfunktion wird als `rd(x)` bezeichnet: Es gilt:

- $rd : \mathbb{R} \to \mathbb{F}$
- surjektiv: $\forall f \in \mathbb{F}\ \exists x \in \mathbb{R} \Rightarrow rd(x) = f$
- idempotent: $rd(rd(x)) = rd(x)$
- monoton: $x \leq y \Rightarrow rd(x) \leq rd(y)$

Rundungsmodi

1. Abrunden:
   - $rd_-(x) = f_l(x)$ wobei $f_l(x)$ die nächstkleinere Float-Zahl ist.
2. Aufrunden:
   - $rd_+(x) = f_r(x)$ wobei $f_r(x)$ die nächstgrößere Float-Zahl ist.
3. Abschneiden (Runden in Richtung 0):
   - $rd_0(x) = f_-(x)$ wenn $x \geq 0$ und $f_+(x)$ wenn $x \leq 0$.
4. Korrektes Runden:
   - Rundet immer zur nächstgelegenen Float-Zahl.
   - Falls die nächstgelegene Zahl gleich weit entfernt ist, wird die Zahl mit gerader Mantisse gewählt.

Rundungsfehler

Durch jeden Rundungsschritt entsteht zwangsläufig ein Rundungsfehler.

- Absolute Rundungsfehler:
  - $rd(x) - x$
- Relative Rundungsfehler:
  - $\epsilon = \frac{rd(x)-x}{x}$
  - Dieser Rundungsfehler kann bei direktem Runden mit: $|\epsilon| \leq \varrho$ abgeschätzt werden.
  - Bei korrektem Runden gilt: $|\epsilon| \leq \frac{1}{2} \cdot \varrho$

Durch diese Konstruktion des relativen Rundungsfehlers, gilt:

- $rd(x) = x * (1 + \epsilon)$

Um die Operationen, welche Rundungsfehler verursachen, von den "sauberen" Operationen zu unterscheiden, wird eine neue Notation eingeführt:

- $a * b$ bezeichnet den Wert der Multiplikation ohne Rundung
- $a \dot{*} b$ bzw. $rd(a * b)$ bezeichnet den Wert nach der Rundung

Es gibt zwei Möglichkeiten, die entstehenden Rundungsfehler zu modellieren:

1. Als Funktion des exakten Ergebnisses: (Starke Hypothese)

- $a \dot{*} b = f(a * b) = (a * b) \cdot (1 + \epsilon)$
  - Diese Variante wird von fast allen Systemen unterstützt.

2. Als Funktion der Rundungsfehler der Operanden: (Schwache Hypothese)
   - $a \dot{*} b = f(a, b) = (a \cdot (1 + \epsilon_1)) * (b \cdot (1 + \epsilon_2))$

Wobei alle ϵ-Werte betragsmäßig durch die Maschinengenauigkeit $\bar{\epsilon}$ begrenzt sind. Diese entspricht je nach verwendetem Rundungsmodus entweder ϱ oder $\frac{1}{2} \cdot \varrho$.

Achtung:

Die gerundeten Varianten der Operatoren sind nicht mehr assoziativ!

- $(a \dot{*} b) \dot{*} c \neq a \dot{*} (b \dot{*} c)$

Außerdem findet Absorption statt. Das bedeutet, dass z.B. bei der Subtraktion von ähnlich großen Zahlen, die Anzahl der signifikanten Stellen deutlich abnimmt. Und dadurch ein extrem hoher Rundungsfehler entsteht.

Fehleranalyse

Es gibt die Möglichkeit der Vorwärts- und Rückwärtsfehleranalyse.

Vorwärts Fehleranalyse

Hierbei wird das Ergebnis als Funktion des exakten Ergebnisses modelliert.

- $a \dot{+} b = (a + b) \cdot (1 + \epsilon)$
- $a \dot{*} b = (a * b) \cdot (1 + \epsilon)$

Diese Modellierung ist einfach, jedoch in der Praxis nur schwer berechenbar, da die Fehler korreliert sind.

Rückwärts Fehleranalyse

Hierbei wird das Ergebnis als Funktion der Rundungsfehler der Operanden modelliert.

- $a \dot{+} b = (a \cdot (1 + \epsilon)) + (b \cdot (1 + \epsilon))$
- $a \dot{*} b = (a \cdot \sqrt{1 + \epsilon}) * (b \cdot \sqrt{1 + \epsilon})$

Kondition

Die Kondition eines Problems ist ein Maß für die Sensitivität des Problems gegenüber Änderungen der Eingabedaten. Diese ist unabhänging vom verwendeten Algorithmus.

Ist ein Problem gut konditioniert, so ist es sehr stabil gegenüber kleinen Änderungen der Eingabedaten. Bei solchen Problemen lohnt sich die Verwendung eines guten Algorithmus.

Ist ein Problem schlecht konditioniert, so ist es sehr empfindlich gegenüber kleinen Änderungen der Eingabedaten. Somit hat sogar der bestmögliche Algorithmus keinen signifikanten Einfluss auf die Genauigkeit des Ergebnisses. Da dieses ohnehin durch die Fehlerfortpflanzung dominiert wird.

Man betrachtet wiederum den absoluten und den relativen Fehler:

- $err_{abs} = f(x + \delta x) - f(x)$
- $err_{rel} = \frac{f(x+\delta x)-f(x)}{f(x)}$

Die Konditionszahl wird nun wie folgt definiert:

- $kond_{abs} = \frac{err_{abs}}{\delta x}$
- $kond_{rel} = \frac{err_{rel}}{\frac{\delta x}{x}}$

Im Algemeinen ist die Konditionszahl eines Problems $p(x)$ bei der Eingabe x als:

- $kond(p(x)) = \frac{\partial p(x)}{\partial x}$

Laut dieser Definition haben alle Grundrechenarten, außer die Addition / Subtraktion, eine Konditionszahl von ca. 1 und sind somit gut konditioniert.

Die Subtraktion ist schlecht konditioniert, da sie bei ca. gleich großen Zahlen zu einem extrem hohen relativen Fehler führen kann.

Beispiele für gute und schlechte Kondition:

- Gut konditionierte Probleme:
  - Berechnung der Fläche eines Rechtecks
  - Berechnung von Schnittpunkten von fast orthogonalen Geraden
- Schlecht konditionierte Probleme:
  - Berechnung von Nullstellen von Polynomen
  - Berechnung von Schnittpunkten zweier fast paralleler Geraden

Akzeptable Ergebnisse

Ein numerisch akzeptables Ergebnis ist dann gegeben, wenn das berechnete Ergebnis, auch als exaktes Ergebnis von nur leicht gestörten Eingabedaten erklärt werden kann.

- $\tilde{y}$ ist akzeptables Ergebnis für $y = f(x)$, wenn $\tilde{y} \in \{f(\tilde{x}) \mid \tilde{x}$ nahe von x$\}$

Numerische Stabilität

Ein Algorithmus ist numerisch stabil, wenn für alle erlaubten Eingabedaten ein *akzeptables* Ergebnis berechnet wird.

- Die Grundrechenarten sind numerisch stabil (Basierend auf schwacher Hypothese)

- Die Komposition von numerisch stabilen Algorithmen ist nicht zwingend stabil

Beispiel:

- Numerisch instabil:
  - Berechnung der Wurzel als: $x = \sqrt{\left(\frac{p}{2}\right)^2 - q} - \frac{p}{2}$
- Numerisch stabil:
  - Berechnung der Wurzel als: $x = \frac{-q}{\sqrt{\left(\frac{p}{2}\right)^2 - q} + \frac{q}{2}}$

Polynom-Interpolation

Problem

Für eine gegebene Funktion $f(x)$, suchen wir eine Funktion $p(x)$ welche einfach zu konstruieren und für weitere Anwendungen nutzbar ist. $p(x)$ soll dabei $f(x)$ annähern und einen *geringen* Fehler aufweisen.

Der resultierende Fehler ist ein Maß für die Qualität der Approximation. Dieser wird als *Fehlerterm* bzw. *Remainder* bezeichnet.

$$f(x) - p(x) = \frac{D^{(n+1)} f(\xi)}{(n+1)!} \prod_{i=0}^{n} (x - x_i)$$

Wobei ξ ein Punkt zwischen x_0 und x_n ist.

Lagrange Polynomials

Bei der Lagrange Interpolation werden Basisfunktionen verwendet, welche jeweils an allen Stützstellen, bis auf der Stelle x_i, den Wert 0 annehmen.

$$L_k(x) = \prod_{i=0, i \neq k}^{n} \frac{x - x_i}{x_k - x_i}$$

Das Resultierende Polynom ergibt sich dann als:

$$p(x) = \sum_{i=0}^{n} y_i \cdot L_i(x)$$

Chebyshev Polynomials

Die Basisfunktionen für die Interpolation sind die Chebyshev Polynome. Diese sind definiert als:

$$\begin{aligned} T_0(x) &= 1 \\ T_1(x) &= x \\ T_{k+1}(x) &= 2x \cdot T_k(x) - T_{k-1}(x) \end{aligned}$$

Damit lassen sich die dazu gehörigen Koeffizienten berechnen:

$$\begin{aligned} a_0 &= \frac{1}{2\pi} \int_{-1}^{1} \frac{f(x)}{\sqrt{1 - x^2}} dx \\ a_k &= \frac{2}{\pi} \int_{-1}^{1} \frac{f(x) \cdot T_k(x)}{\sqrt{1 - x^2}} dx \end{aligned}$$

Die Interpolation ist dann gegeben durch:

$$p(x) = \sum_{k=0}^{n} a_k T_k(x)$$

Bernstein Polynomials

Mithilfe der Bernstein Polynome lässt sich die Interpolation durch Bezier Kurven realisieren. Diese sind definiert als:

$$B_i^n = \binom{n}{i}(1-t)^{n-i} \cdot t^i$$

Die Interpolation ist dann gegeben durch:

$$p(t) = \sum_{i=0}^{n} b_i B_i^n(t)$$

Hierbei stellen b_i die Kontrollpunkte dar. (Diese können auch höherdimensional sein)

Variationen der Problemstellung

Es gibt zwei verschidenen Anwendungen der Interpolation welche auftreten können:

1. Simple Nodes
   - Man hat eine Menge von Punkten $P = \{(x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n)\}$, welche durch ein Polynom interpoliert werden sollen.
   - Diese Variante wird auch als *Lagrange Interpolation* bezeichnet.
2. Multiple Nodes
   - Man hat heine Menge von Knoten $P = \{(x_0, y_0, y_0'), (x_1, y_1, y_1'), \ldots, (x_n, y_n, y_n')\}$ welche durch ein Polynom interpoliert werden sollen.
   - Hierbei ist y_i' die Ableitung von y_i.
   - Diese Variante wird als *Hermit Interpolation* bezeichnet.

Algorithmen

Schema von Aitken und Neville

Wenn man nicht an der expliziten Representation des Interpolationspolynoms interessiert ist, sondern nur den Funktionswert an einem festen x Wert bestimmen möchte eignet sich das Schema von Aitken und Neville.

Algorithmus:

1. Initialisiere konstante Polynome welche den Funktionswert an den Stützstellen annehmen.
   - $p_{0,0} = y_0, p_{1,0} = y_1, \ldots, p_{n,0} = y_n$
2. Verfeinere rekursiv die Polynome durch die Kombibation mehrerer Polynome.
   - $p_{i,j} = \frac{x_{i+k} - x}{x_{i+k} - x_i} \cdot p_{i,k-1}(x) + \frac{x - x_i}{x_{i+k} - x_i} \cdot p_{i+1,k-1}(x)$

Zur Berechnung mit Hand kann folgendes Schema herangezogen werden:

Als Pseudo-Code:

Diese Form der Interpolation eignet sich nur, wenn nur relative wenige Werte ausgewertet werden müssen. Ansonsten lohnt sich die Bestimmung des expliziten Polynoms.

x_i	$i \setminus k$	0		1		2		$\cdots$
x_0	0	$p[0,0] = y_0$	$\nearrow$	$p[0,1]$	$\nearrow$	$p[0,2]$	$\rightarrow$	$\cdots$
x_1	1	$p[1,0] = y_1$	$\nearrow$	$p[1,1]$	$\rightarrow$	$\vdots$		
x_2	2	$p[2,0] = y_2$	$\rightarrow$	$\vdots$				
$\vdots$	$\vdots$	$\vdots$						

Abbildung 1: Dreiecks-Schema für Aitken-Neville

```
for i=0:n; p[i,0]:=f_x[i]; end
for k=1:n
    for i=0:n-k
        p[i,k] := p[i,k-1] + (x-x[i])/(x[i+k]-x[i])*(p[i+1,k-1] - p[i,k-1]);
    end
end
```

Abbildung 2: aitken_neville_code

Newton Interpolation

Die Newton Interpolation ist eine spezielle Form der Lagrange Interpolation. Hierbei werden die Koeffizienten des Polynoms durch die Differenzenquotienten der Stützstellen bestimmt.

Algorithmus:

1. Initialisiere die Differenzenquotienten
   - $[x_i]f = f(x_i) = y_i$
2. Rekursiv berechne die Differenzenquotienten
   - $[x_i, x_{i+1}, \ldots, x_{i+k}]f = \frac{[x_{i+1},\ldots,x_{i+k}]f - [x_i,\ldots,x_{i+k-1}]f}{x_{i+k} - x_i}$

Die Interpolation ist dann gegeben durch:

$$p(x) = \sum_{i=0}^{n} [x_0, x_1, \ldots, x_i]f \cdot \prod_{j=0}^{i-1} (x - x_j)$$

Diese Methode eignet sich gut, um die explizite Form des Polynoms zu erhalten. Außerdem ist es leicht möglich, weitere Stützstellen hinzuzufügen.

Der entstehende Fehler ist hierbei $\mathcal{O}(h^{n+1})$. Wobei h die Distanz zwischen den Stützstellen ist.

Auch für die Newton Interpolation gibt es ein Schema für die händische Berechnung:

x_i	$i \setminus k$	0		1		2		$\cdots$
x_0	0	$c_{0,0} = y_0$	$\nearrow$	$c_{0,1}$	$\nearrow$	$c_{0,2}$	$\rightarrow$	$\cdots$
x_1	1	$c_{1,0} = y_1$	$\nearrow$	$c_{1,1}$	$\rightarrow$	$\vdots$		
x_2	2	$c_{2,0} = y_2$	$\rightarrow$	$\vdots$				
$\vdots$	$\vdots$	$\vdots$						

Abbildung 3: newton_interpolation_schema

Dementsprechend sind auch die Variablen in den Formeln anders benannt:

$$c_{i,0} = f(x_i) = y_i$$

$$c_{i,k} = \frac{c_{i+1,k-1} - c_{i,k-1}}{x_{i+k} - x_i}.$$

$$p(x) = c_{0,0} + c_{0,1} \cdot (x - x_0) + \ldots + c_{0,n} \cdot \prod_{i=0}^{n-1} (x - x_i) \ .$$

Abbildung 4: newton_interpolation_formeln

Kondition von Interpolationspolynomen

Die Kondition der Polynomialen Interpolation ist besonders bei einer großen Anzahl von Stützstellen ($n > 7$) ein Problem. Da das entstehende Polynom besonders an den Randstellen extrem oszillieren kann.

Polynom - Splines

Definition

Anstatt alle Punkte durch ein gemeinsames Polynom zu interpolieren, wird der Bereich in mehrere Intervalle unterteilt und für jedes Intervall ein eigenes Polynom erstellt, welches dann an den Intervallgrenzen mit den anderen Polynomen "zusammengeklebt" wird.

Ein Spline $s(x)$ von der Ordnung m bzw. mit Grad $m-1$ ist eine Kette von Polynomen mit Grad $m-1$, welche jeweils zwischen zwei Stützstellen die Funktion interpolieren. Außerdem ist $s(x)$ auf dem gesamten Intervall jeweils $m-2$ mal stetig differenzierbar ist.

Beispiel:

- $m = 1 \rightarrow$ Stückweise konstante Funktion, Treppenfunktion
- $m = 2 \rightarrow$ Stückweise lineare Funktion, stetig
- $m = 3 \rightarrow$ Stückweise quadratische Funktion, stetig und einmal stetig differenzierbar

Kubische Splines

Für den Fall $m = 4$ erhält man kubische Splines. Diese eignen sich gut für die Interpolation von Datenpunkten, da sie einfach zu berechnen sind und eine gute Approximation liefern.

Durch geeignete Herleitung, erhält man für jedes Teilintervall folgende Basisfunktionen:

$$\begin{aligned}
\alpha_1(t) &= 1 - 3t^2 + 2t^3 \\
\alpha_2(t) &= 3t^2 - 2t^3 \\
\alpha_3(t) &= t - 2t^2 + t^3 \\
\alpha_4(t) &= t^3 - t^2
\end{aligned}$$

Damit erhält man für die Funktion $s(x)$ folgende Form:

$$\begin{aligned}
s(x) &= p * i \left(\frac{x - x_i}{h_i} \right) := p_i(t) \\
&= y_i \cdot \alpha_1(t) + y * i + 1 \cdot \alpha * 2(t) + h_i \cdot y_1' \cdot \alpha_3(t) + h_i \cdot y * i + 1' \cdot \alpha_4(t)
\end{aligned}$$

Diese Formel garantiert, dass:

$$\begin{aligned}
s(x_i) &= y_i & \forall i \\
s(x_{i+1}) &= y_{i+1} & \forall i \\
s'(x_i) &= y_i' & \forall i \\
s'(x_{i+1}) &= y_{i+1}' & \forall i
\end{aligned}$$

Der Fehler für kubische Splines kann durch $|f(x) - s(x)| = \mathcal{O}(h^4)$ abgeschätzt werden. Dies ist wesentlich besser als bei der Interpolation durch ein einziges Polynom.

Hierbei benötigt man allerdings die Ableitung des gewünschten Polynoms an den Stützstellen. Sollten diese aber nicht bekannt sein, können diese unter der Annahme von 2-mal stetiger Differenzierbarkeit folgendermaßen ermittelt werden:

$$\begin{pmatrix} 4 & 1 & & \\ 1 & 4 & \ddots & \\ & \ddots & \ddots & 1 \\ & & 1 & 4 \end{pmatrix} \begin{pmatrix} y'_1 \\ y'_2 \\ \vdots \\ y'_{n-2} \\ y'_{n-1} \end{pmatrix} = \frac{3}{h} \begin{pmatrix} y_2 - y_0 - \frac{h}{3} y'_0 \\ y_3 - y_1 \\ \vdots \\ y_{n-1} - y_{n-3} \\ y_n - y_{n-2} - \frac{h}{3} y'_n \end{pmatrix}.$$

Abbildung 5: Berechnung der Ableitungen

Somit müssen lediglich die Ableitungen in den Randpunkten des Interpolationsintervalls müssen angegeben werden.

Trigonometrische Interpolation

Definition

Bei dieser Form von Interpolation werden die Basisfunktionen durch Sinus- und Kosinus-Funktionen ersetzt. Diese Form der Interpolation ist besonders gut geeignet, wenn die zu interpolierenden Datenpunkte periodisch sind.

Um den Rechenaufwand zu minimieren behilft man sich der komplexen Darstellung von Sinus- und Kosinus-Funktionen. $e^{i\theta} = \cos(\theta) + i\sin(\theta)$

Die verwendeten Stützstellen liegen gleichverteilt auf dem Einheitskreis. Es gilt: $z_j = e^{\frac{2\pi i}{n}j}$

Der kontinuierliche Interpolant ist gegeben durch: $z_t = e^{2\pi it} \quad t \in [0,1]$

Das resultierende Polynom hat die Form:

$$p(t) = \sum_{k=0}^{n} c_k \cdot z^k = \sum_{k=0}^{n} c_k \cdot e^{2\pi ikt}$$

Diskrete Fourier Transformation

Es soll eine Interpolation gefunden werden die die gleichverteilten Punkte $P = [(x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n)]$ interpoliert. Hierbei ist $\omega = e^{\frac{2\pi i}{n}}$ die n-te Wurzel von 1.

$$\begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_{n-1} \end{bmatrix} = \frac{1}{n} \begin{bmatrix} 1 & 1 & 1 & \ldots & 1 \\ 1 & \bar{\omega} & \bar{\omega}^2 & \ldots & \bar{\omega}^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \bar{\omega}^{n-1} & \bar{\omega}^{2(n-1)} & \ldots & \bar{\omega}^{(n-1)^2} \end{bmatrix} \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{bmatrix}$$

Man erhält jetzt also die Werte $[(f_0, c_0), (f_1, c_1), \ldots, (f_{n-1}, c_{n-1})]$. Welche jeweils die Frequenz und die Amplitude der jeweiligen Basisfunktionen darstellen.

Damit kann das Polynom $p(t)$ berechnet werden.

Inverse Diskrete Fourier Transformation

Die Inverse Diskrete Fourier Transformation ist die Umkehrung der Diskreten Fourier Transformation. Sie berechnet die Funktionswerte y_i aus den Koeffizienten c_i.

$$\begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & \ldots & 1 \\ 1 & \omega & \omega^2 & \ldots & \omega^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \ldots & \omega^{(n-1)^2} \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_{n-1} \end{bmatrix}$$

Fast Fourier Transformation

Die Fast Fourier Transformation ist eine effiziente Methode zur Berechnung der Diskreten Fourier Transformation. Sie ist eine rekursive Methode, die die Berechnung der Diskreten Fourier Transformation in $\mathcal{O}(n \log n)$ durchführt.

Algorithmus:

- Divide:
  1. Teile die Daten in zwei Teile auf. Einmal die geraden und einmal die ungeraden Indizes.
  2. Wiederhole Schritt 1 für die beiden Teile, solange bis nur noch ein Wert übrig ist.
- Konquer:
  1. Kombiniere jeweils die geraden und ungeraden Werte eines Teils zu einem neuen Wert.
  2. Wende dazu den *Butterfly* Operator an.
     - $[a_j, b_j] \mapsto [a_j + \omega^j b_j, a_j - \omega^j b_j]$
       - j bezeichnet dabei den Index des der Operation im jeweiligen "Batch".

Numerische Quadratur

Problem

Gegeben sei eine Funktion $f(x)$, die auf einem Intervall $[a, b]$ definiert ist. Wir wollen nun die Fläche unter der Kurve numerisch $f(x)$ berechnen. Die Integration kann auch als gewichtete Summe der Funktionswerte auf einem Gitter gesehen werden.

$$I(f) \approx Q(f) := \sum_{i=0}^{n} g_i \cdot y_i$$

Idee: Anstatt eine komplizierte Funktion zu integrieren, und diese womöglich sehr oft ausrechnen zu müssen, um die benötigten Stützpunkte zu erhalten, können wir die Funktion durch Polynome interpolieren und dann die Fläche unter dieser Polynome exakt berechnen.

$$Q(f) = \int_a^b \tilde{f}(x)dx := \int_a^b p(x)dx$$

Hierbei stellt $\tilde{f}(x) = p(x)$ die Interpolationsfunktion dar.

Sonderfall: Wenn man $p(x)$ mittels Lagrange Interpolation berechnet, kann man die Resultierenden Faktoren vorberechnen.

Kondition der numerischen Quadratur

Wenn alle gewichte g_i der Interpolation positiv sind, dann ist die die numerischen Quadratur gut konditioniert. Der Fehler des Ergebnisses ist dann proportional zum Fehler der Eingabedaten.

Sollten jedoch manche Gewichte negativ sein, dann ist die numerische Quadratur schlecht konditioniert.

Algorithmen

Rechteckregel

Die Rechteckregel ist die einfachste Form der numerischen Quadratur. Hierbei wird die Fläche unter der Kurve durch ein Rechtecke abgeschätzt. Im gesamten Intervall $[a, b]$ ergibt sich dann:

$$Q_R(f) := H \cdot f\left(\frac{a+b}{2}\right)$$

Für das Integrationsintervall der Länge H wird also der Mittelpunkt berechnet und eine Konstante Funktion durch diesen Punkt gelegt. Dieses einfache Polynom wird nun exakt integriert.

Der entstandene Fehler bei dieser Variante ist in $O(H^3 \cdot f''(\xi))$.

Trapezregel

Ansatz: Die Fläche unter der Kurve kann im Intervall $[a, b]$ auch durch ein Trapez abgeschätzt werden. Dieses Polynom wird nun exakt integriert.

$$Q_T(f) := H \cdot \frac{(f(a) + f(b))}{2}$$

Der entstandene Fehler bei dieser Variante ist immer noch in $O(H^3 \cdot f''(\xi))$.

Kepler'sche Regel

Anstatt die Funktion $f(x)$ durch ein lineares Polynom abzuschätzen verwenden wir ein quadratisches Polynom. Dieses Polynom wird nun exakt integriert.

Für das Intervall $[a, b]$ ergibt sich:

$$Q_F(f) := H \cdot \frac{\left(f(a) + 4 \cdot f\left(\frac{a+b}{2}\right) + f(b)\right)}{6}$$

Die Fehlerordnung ist hier in: $O(H^5 \cdot f^{(4)}(\xi))$.

Trapezregel mit mehreren Teilintervallen

Die Trapezregel kann auch auf mehrere Teilintervalle der Länge $h = \frac{b-a}{n}$ angewendet werden. Hierbei wird die Fläche unter der Kurve in Jedem Intervall durch ein Trapez abgeschätzt.

Auf dem gesamten Intervall $[a, b]$ ergibt sich:

$$Q_{TS}(f) := h \cdot (\frac{f_0}{2} + f_1 + f_2 + \cdots + f_{n-1} + \frac{f_n}{2})$$

Der Fehler ist hier in $O(H \cdot h^2 \cdot f''(\xi))$.

Simpson'sche Regel

Die Simpson'sche Regel ist eine Erweiterung der Trapezregel. Hierbei wird die Funktionin Jedem Intervall durch ein Parabel-Polynom abgeschätzt.

Auf dem gesamten Intervall $[a, b]$ ergibt sich:

$$Q_{SS}(f) := \frac{h}{3} \cdot (f_0 + 4 \cdot f_1 + 2 \cdot f_2 + 4 \cdot f_3 + \cdots + 2 \cdot f_{n-2} + 4 \cdot f_{n-1} + f_n)$$

Der Fehler ist hier in $O(H \cdot h^4 \cdot f^{(4)}(\xi))$.

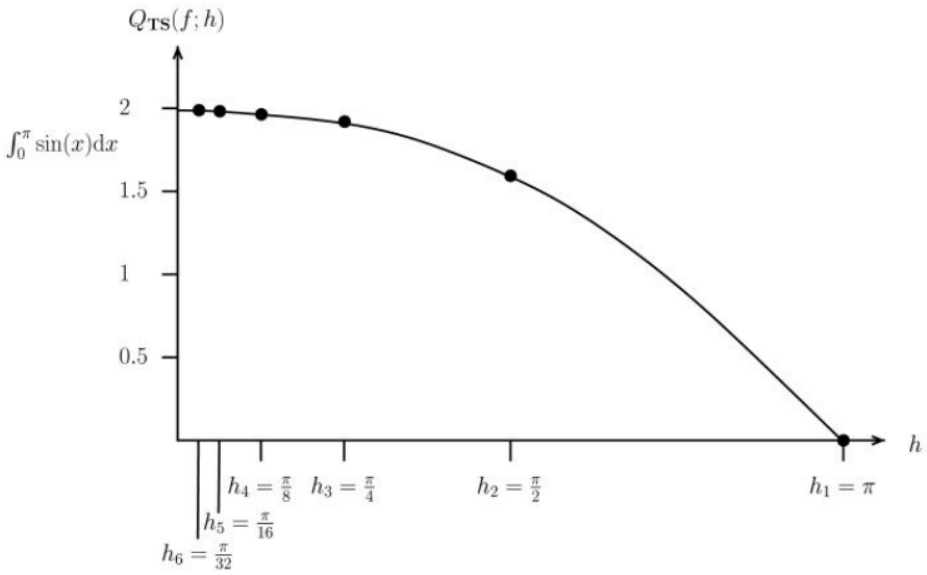
Nicht gleichmäßige Gitter

Um mehrere Abtastpunkte am Rand des Intervalls können die Abtastpunkte auch anders gewählt werden.

$x_i = a + H \cdot \frac{1-\cos\left(\frac{i \cdot \pi}{n}\right)}{2}$

Extrapolation

Bei der Extrapolation wird die numerische Quadratur mit verschider Anzahl von Teilintervallen berechnet. Die daraus resultierenden Ergebnisse werden dann intelligent miteinander kombiniert (extrapoliert), um so ein um ein Vielfaches genaueres Ergebnis zu erhalten.



h	h_1	h_2	h_3	h_4	h_5	h_6
$Q_{TS}(f;h)$	0	1.571	1.896	1.974	1.993	1.998
error	2	0.429	0.104	$2.6 \cdot 10^{-2}$	$7 \cdot 10^{-3}$	$2 \cdot 10^{-3}$

Abbildung 6: Extrapolation

Monte Carlo Integration

Bei der Monte Carlo Integration wird die Fläche unter der Kurve durch eine große Anzahl von Zufallszahlen abgeschätzt. Dabei wird die zu integrierende Funktion $f(x)$ in einen Bereich eingebettet, der die Fläche unter der Kurve umfasst. Die Zufallszahlen werden nun in diesem Bereich verteilt. Die Anzahl der Zufallszahlen, die unter der Kurve liegen, ist proportional zur Fläche unter der Kurve.

Gaussian Quadrature

Bei der Gaussian Quadrature werden die Gewichte und die Stützstellen der Interpolation so gewählt, dass die Interpolation exakt ist.

Bei n Stützstellen ergibt sich eine maximale exakte Interpolation von Polynomen der Ordnung $2n - 1$.

Zum Beispiel:

$$\int_{-1}^{1} f_k(x) \cdot dx = \sum_{i=1}^{n} w_i \cdot f_k(x_i)$$

Erstelle Gleichungsystem für $f_k \in \{1, x, x^2, x^3, \dots, \}$

Archimedes Quadrature

Eine Divide-and-Conquer Variante der Integration. Hierbei wird iterativ die Fläche unter einer Kurve bestimmt, indem man Teilflächen konstruiert, die sich immer weiter an die Fläche annähern.

Lineare Gleichungssysteme

Lineare Gleichungssysteme sind ein sehr wichtiges Thema in der Mathematik. Mit ihnen lassen sich Matrix-Vektor-Produkte, Eigenwerte, Differentialgleichungen ... berechnen.

Arten von Matrizen

Volle Matrizen

Eine volle Matrix ist eine Matrix, in der die meisten Elemente nicht Null sind. Sie wird auch als dicht bezeichnet.

Sparse Matrix

Eine Sparse Matrix ist eine Matrix, bei der die meisten Elemente 0 sind. Diese Nullen tauchen oft in einem speziellen Muster auf.

Beispiel:

$$Diagonalmatrix = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

$$Tridiagonalmatrix = \begin{pmatrix} 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 \end{pmatrix}$$

$$BandedMatrix = \begin{pmatrix} 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 1 \end{pmatrix}$$

Lösungsverfahren

Es gibt zwei verschiedene Arten von Lösungsverfahren für lineare Gleichungssysteme:

1. Direkte Verfahren
    - Bei diesen Verfahren wird (bis auf Rundungsfehler) das exakte Ergebnis berechnet.
2. Indirekte Verfahren
    - Bei diesen Verfahren wird iterativ ein Näherungswert berechnet.

Die offensichtliche Lösungen, das Gleichungssystem einfach durch invertieren der Matrix zu lösen, sowie die Cramer-Regel sind für große Matrixen unbrauchbar, da sie sehr viel Rechenzeit benötigen.

Vektor Normen

Eine Vektor Norm besitzt folgende Eigenschaften:

$$\begin{aligned} \|x\| &\geq 0 && \text{(positivität)} \\ \|x\| &= 0 && \text{(nur wenn } x = 0\text{)} \\ \|x + y| \leq |x| + |y| && && \text{(Triangulare Ungleichung)} \\ \|ax\| &= |a|\|x\| && \text{(Skalierung)} \end{aligned}$$

Beispiele für Vektor Normen:

$$\begin{aligned} \text{Euklidische Norm:} && \|x\|_2 &= \sqrt{\sum_{i=1}^{n} x_i^2} \\ \text{Manhattan Norm:} && \|x\|_1 &= \sum_{i=1}^{n} |x_i| \\ \text{Maximum Norm:} && \|x\|_\infty &= \max_{i=1,\ldots,n} |x_i| \end{aligned}$$

Matrix Normen

Eine Matrix Norm wird von einer Vekor Norm abgeleitet. Sie wird definiert als:

$$||A|| = \max_{||x||=1} ||Ax||$$

Die Kondizionszahl einer Matrix ist definiert als:

$$\kappa(A) = \frac{\max_{||x||=1} ||Ax||}{\min_{||x||=1} ||Ax||}$$

Bei einer invertierbaren Matrix ist $\kappa(A) = ||A|| \cdot ||A^{-1}||$.

Kondition Lösen von Gleichungssystemen

Der relative Fehler der Lösung eines Gleichungssystems, bei denen alle relativen Fehler kleiner als ϵ sind, ist:

$$\frac{||\delta x||}{||x||} \leq \frac{2\epsilon\kappa(A)}{1 - \epsilon\kappa(A)}$$

Somit wird auch der Fehler sehr schnell groß, wenn die Kondition der Matrix groß ist.

Das Residuum

Das Residuum ist definiert als $r := b - A\tilde{x}$. Wobei $\tilde{x}$ die Annäherung von x ist.

Der Fehler hängt nicht umbedingt vom residuum ab. $\tilde{x}$ kann auch als exakte Lösung von einem gestörten Gleichungssystem verstanden werden.

$$r = b - A\tilde{x} \iff A\tilde{x} = b - r$$

LR-Zerlegung

Ziel: $Ax = b$ zu lösen.

Art: Führe Gauss-Verfahren durch und Speichere die Verwendeten Koeffizienten in einer Matrix L ab.

Speichert man sich die verwendeten Koeffizienten für jede Zeilen-Subtraktion Elementweise in einer neuen Matrix ab erhält man L. Zusammen mit R, der vom Gausverfahren übriggebliebenen Dreicksmatrix, kann man A mit $A = LR$ wiederherstellen. Man erhält also eine Faktorisierung von A.

Dadurch kann man $Ax = LRx = L(Rx) = b$ lösen.

Cholesky-Zerlegung

Bei der Cholesky-Zerlegung wird eine Matrix A in eine Matrix L und deren Transponierte L^T zerlegt.

Dadurch kann das Gleichungssystem $Ax = b$ mit halber Rechenzeit gelöst werden. Die asymptotische Komplexität ist jedoch immernoch $O(n^3)$.

Pivotsuche

Bei der Pivotsuche wird die Zeile mit dem größten Element in der Spalte, bzw. der restlichen Matrix gesucht.

Damit kann verhindert werden, dass bei der Berechnung der Zeilensubtraktions-Koeffizienten, durch 0 geteilt wird. Außerdem werden die benötigten Faktoren und Fehler kleiner.

- Partielle Pivotsuche
  - In der Spalte wird nach dem betragsmäßig größten Element gesucht.
  - Die Zeile wird mit der Zeile mit dem größten Element vertauscht.
- Totale Pivotsuche
  - Es wird nach dem größten Element in der gesamten Rest-Matrix gesucht.
  - Es müssen womöglich Zeilen und Spalten vertauscht werden.
    - Dabei müssen auch eventuell die Elemente im Lösungsvektor vertauscht werden.

Die Umordnung der Matrix und der Vektoren ändert nichts an der Lösung des Gleichungssystems.

Ordinary Differential Equations

Definition

A differential equation is an equation that relates one or more functions and their derivatives. An example of a differential equation is $\dot{y}(t) = y(t)$

The difference between ordinary and partial differential equations is that ordinary differential equations only have one independent variable, while partial differential equations have more than one independent variable.

The Heat Equation is an example of a partial differential equation.

$$u_t(x,t) = u_{xx}(x,t)$$

Nearly all physical phenomena can be described by differential equations. But a differential equation does not specify a unique solution. There is a need for additional constraints to determine a unique solution.

Such constraints appear in:

- Initial conditions
- Boundary conditions

Initial Conditions

Initial conditions describe the state of the system at the beginning of the simulation. For example:

$$y(0) = 1$$
$$\dot{y}(0) = 0$$

Boundary Conditions

Boundary conditions describe the state of the system at the boundary of the simulation. For example:

$$y(0) = 1$$
$$y(1) = 0$$

Analytical Solution

In simple cases, a differential equation can be solved analytically. This means that the solution can be expressed in terms of elementary functions. This is not always possible.

Lipschitz Condition

The Lipschitz condition is a necessary condition for the existence of a unique solution to a differential equation. It states that the derivative of the solution must be bounded.

$$\left|\frac{dy}{dt}\right| \leq L$$

Condition of Differential Equations

In general differential equations are ill conditioned. This means that small changes in the input can lead to large changes in the output. This is a problem for numerical methods.

Differential Equations as Integration Problems

A differential equation can be seen as an integration problem. The solution of the differential equation is the integral of the right hand side of the equation.

$$y(t+\delta t) = y(t) + \int_t^{t+\delta t} \dot{y}(t)dt$$

Numerical Solutions

Let $\dot{y}(t) = f(t, y(t))$ be a differential equation. With $t \in [a, b]$ and $y(a) = y_a$.

Finite Difference Method / Euler Method

The finite difference method is a numerical method for solving differential equations. It is based on the Taylor expansion of the solution.

The derivative of the solution is approximated by a finite difference.

$$\frac{dy}{dt} \approx \frac{y(t+\delta t) - y(t)}{\delta t}$$

This yields to the following approximation of the solution:

$$y(a+\delta t) \approx y(a) + \delta t \cdot f(a, y(a))$$
$$y_{k+1} \approx y_k + \delta t \cdot f(t_k, y_k)$$

The Euler method corrisponds with the rectangular rule for numerical integration.

The Error of the Euler method is:

- $l_{\delta t} = \mathcal{O}(\delta t)$
- $e_{\delta t} = \mathcal{O}(\delta t)$

Euler Method in Python

```
def euler(f, t0, y0, dt, n):
    t = t0
    y = y0

    for i in range(n):
        y = y + dt*f(t,y)
        t = t + dt
    return y
```

Method of Heun

The method of Heun uses two steps of the Euler method to improve the accuracy of the solution.

$$y_{k+1} \approx y_k + \frac{\delta t}{2} \cdot (f(t_k, y_k) + f(t_{k+1}, y_k + \delta t \cdot f(t_k, y_k)))$$

The method of Heun corrisponds with the trapezoidal rule for numerical integration.

The Error of the method of Heun is:

- $l_{\delta t} = \mathcal{O}((\delta t)^2)$
- $e_{\delta t} = \mathcal{O}((\delta t)^2)$

Method of Heun in Python

```
def heun(f, t0, y0, dt, n):
    t = t0
    y = y0

    for i in range(n):
        y = y + dt/2*(f(t,y) + f(t+dt, y+dt*f(t,y)))
        t = t + dt
    return y
```

Runge-Kutta Method

The Runge-Kutta method is a generalization of the Euler method. It uses multiple steps of the Euler method to improve the accuracy of the solution.

$$y_{k+1} \approx y_k + \frac{\delta t}{6} \cdot (T_1 + 2 \cdot T_2 + 2 \cdot T_3 + T_4)$$

where

$$\begin{aligned} T_1 &= f(t_k, y_k) \\ T_2 &= f(t_k + \frac{\delta t}{2}, y_k + \frac{\delta t}{2} \cdot T_1) \\ T_3 &= f(t_k + \frac{\delta t}{2}, y_k + \frac{\delta t}{2} \cdot T_2) \\ T_4 &= f(t_k + \delta t, y_k + \delta t \cdot T_3) \end{aligned}$$

The Runge-Kutta method corrisponds with the Kepler rule for numerical integration.

The Error of the Runge-Kutta method is:

- $l_{\delta t} = \mathcal{O}((\delta t)^4)$
- $e_{\delta t} = \mathcal{O}((\delta t)^4)$

Runge-Kutta Method in Python

```
def rungeKutta(f, t0, y0, dt, n):
    t = t0
    y = y0

    for i in range(n):
        T1 = f(t,y)
        T2 = f(t+dt/2, y+dt/2*T1)
        T3 = f(t+dt/2, y+dt/2*T2)
        T4 = f(t+dt, y+dt*T3)

        y = y + dt/6*(T1 + 2*T2 + 2*T3 + T4)
        t = t + dt
    return y
```

Consistency and Convergence

- Local Discretization Error:
  - This is the error which happens in a single step of the numerical method, if we suppose that the starting conditions are known exactly.
- Global Discretization Error:
  - This is the error which accumulates ver the whole simulation, if we suppose that the starting conditions are known exactly.

Local Discretization Error

The local discretization error is the maximum error which arrises from a single step of the numerical method.

$$l_{\delta t} = \max_{t \in [a,b]} \left| \frac{y(t+\delta t) - y(t)}{\delta t} - \dot{y}(t) \right|$$

If $\lim_{\delta t \to 0} l_{\delta t} = 0$ then the method is called consistent.

Global Discretization Error

The global discretization error is the maximum error between the numerical solution and the exact solution.

$$e_{\delta t} = \max_{k=0,\dots,N} |y_k - y(t_k)|$$

If $\lim_{\delta t \to 0} e_{\delta t} = 0$ then the method is called convergent. This means that increasing the number of steps leads to a better approximation of the solution.

Multistep Methods

Multistep methods are numerical methods for solving differential equations. They use the solution of the previous steps to improve the accuracy of the solution.

Adams-Bashforth Method

The Adams-Bashforth method works by replacing f with a polynomial approximation of f. This polynomial is then integrated exactly.

$$y_{k+1} \approx y_k + \frac{\delta t}{2} \cdot (3 \cdot f(t_k, y_k) - f(t_{k-1}, y_{k-1}))$$

Iterative Methods

Introduction

Many Problems occuring in real life require solving a system of linear equations. However, in many cases, the system of equations is too large to be solved analytically (Gauss-Elimination: $\mathcal{O}(n^3)$). In this case, iterative methods are used to solve the system of equations numerically. Most appearing matrices also have a specific shape (sparse, diagonal-dominant,...) which can be exploited to improve the performance of the methods.

Terminology

The convergence of an iterative method is defined as the following:

$$|x^{(i+1)} - \hat{x}| \leq c \cdot |x^{(i)} - \hat{x}|^{\alpha}$$

If $\alpha = 1 \wedge 0 < c < 1$ the method converges linearly.

If $\alpha = 2 \wedge 0 < c < 1$ the method converges quadratically and so on.

There exist methods which don't converge globaly. That means that there exists a solution x which is not found by the method. Such methods only converge if the starting guess $x^{(0)}$ is close enough to the solution.

All linear iterative methods converge globaly.

Relaxation Methods

Relaxation methods are iterative methods for solving linear systems of equations. They use the residual to improve the solution.

$$r^{(i)} = b - Ax^{(i)} = -Ae^{(i)}$$

Where $r^{(i)}$ is the residual at Iteration-Step i, b is the right hand side of the equation, and $x^{(i)}$ is the approximation of the solution at Iteration-Step i.

Richardson Iteration

The Richardson Iteration is the easiest way of updating the guessed solution. It works by just adding the Residual each step.

$$x^{(i+1)} = x^{(i)} + r^{(i)}$$

The Update can also be written in Matrix form:

$$x^{(i+1)} = x^{(i)} + Ir^{(i)}$$

Richardson Iteration in Python

```
def richardson(A, b, tol=1e-6, max_iter=1000):
    x = np.zeros_like(b)
    for i in range(max_iter):
        r = b - A @ x
        x = x + r
        if np.linalg.norm(r) < tol:
            break
    return x
```

Jacobi Iteration

The Jacobi Iteration is an improvement of the Richardson Iteration. It uses the diagonal-dominant property of the matrix to improve the convergence speed.

If you use a for-loop based implementation, in order to not overwrite the values of the previous iteration, the update-vector needs to be precomputed. This needs to be done, becausethe residual $r = b - Ax^{(i)}$ changes with every step.

$$y_k = \frac{1}{a_{kk}} r_k^{(i)}$$
$$x_k^{(i+1)} = x_k^{(i)} + y_k$$

The Update can also be written in Matrix Form:

$$x^{(i+1)} = x^{(i)} + D^{-1} r^{(i)}$$

where D is the diagonal-row part of matrix A.

Jacobi Iteration in Python

```
def jacobi(A, b, tol=1e-6, max_iter=1000):
    x = np.zeros_like(b)
    for i in range(max_iter):
        r = b - A @ x
        x = x + np.diag(1 / np.diag(A)) @ r
        if np.linalg.norm(r) < tol:
            break
    return x
```

Gauss-Seidel Iteration

The Gauss-Seidel Iteration is an improvement of the Jacobi Iteration. It uses doesnt use a `freezed` residual during each update step. Instead it recalculates the residual after each improvement and uses the improved residual for future steps.

$$r_k^{(i)} = b_k - \sum_{j=1}^{k-1} a_{kj} x_j^{(i)} - \sum_{j=k}^{n} a_{kj} x_j^{(i)}$$
$$y_k = \frac{1}{a_{kk}} r_k^{(i)}$$
$$x_k^{(i+1)} = x_k^{(i)} + y_k$$

The update can also be written in Matrix Form:

$$x^{(i+1)} = x^{(i)} + (D_A + L_A)^{-1} r^{(i)}$$

where D_A is the diagonal-row part of matrix A and L_A is the lower-triangular part of matrix A.

Gauss-Seidel Iteration in Python

```python
def gauss_seidel(A, b, tol=1e-6, max_iter=1000):
    x = np.zeros_like(b)
    for i in range(max_iter):
        r = b - A @ x
        x = x + np.diag(1 / np.diag(A)) @ r
        if np.linalg.norm(r) < tol:
            break
    return x
```

In all of the Methods above a damping-factor $0 < \alpha < 2$ can be used to improve the convergence speed. $x^{(i+1)} = x^{(i)} + \alpha y$.

SOR Iteration

The SOR Iteration is an improvement of the Gauss-Seidel Iteration. It uses a damping-factor α to improve the convergence speed.

$$x^{(i+1)} = x^{(i)} + \alpha(D_A + L_A)^{-1} r^{(i)}$$

convergence of those Methods

- Due to construction of the methods, there only exists one possible solution for x if the iteration converges
- If the spectral radius of the iteration matrix is less than 1, the iteration converges
- If A is positive-definite, the SOR method (and therefore the Gauss-Seidel method) converges
- If A is strict diagonal dominant, the Jacobi and Gauss-Seidel methods converges

The smaller the spectral radius (|smallest eigenvalue|) of the iteration matrix, the faster the iteration converges.

In generall, the finer the mesh, the bigger the spectral radius. This is bad for big simulations.

Minimization Methods

Linear systems of equations can also be solved by minimizing a function.

For example:

$$\begin{aligned} f(x) &= \frac{1}{2} x^T A x - b^T x \\ f'(x) &= Ax - b = -r(x) \\ \implies f'(x) = 0 &\iff Ax = b \end{aligned}$$

Method of Steepest Descent

The Method of Steepest Descent is an iterative method for minimizing a function. It works by taking the steepest descent of the function at each step.

$$\begin{aligned} r^{(0)} &= b - Ax^{(0)} \\ \alpha_i &= \frac{{r^{(i)}}^T r^{(i)}}{{r^{(i)}}^T A r^{(i)}} \\ x^{(i+1)} &= x^{(i)} - \alpha_i A r^{(i)} \end{aligned}$$

Steepest Descent in Python

```
def steepest_descent(A, b, tol=1e-6, max_iter=1000):
    x = np.zeros_like(b)
    r = b - A @ x
    for i in range(max_iter):
        alpha = (r.T @ r) / (r.T @ A @ r)
        x = x + alpha * r
        r = r - alpha*A @ r
        if np.linalg.norm(r) < tol:
            break
    return x
```

Discussion Steepest Descent

- The convergence can take arbitrary long. For Example $A = I$ and $b = 0$.

Conjugate Direction Method

The Conjugate Direction Method is an iterative method for minimizing a function. It works by taking a step in the correct direction at each step. (The error in the $i+1$-step is orthogonal to all the errors in the previous steps.)

$$
\begin{aligned}
d^{(0)} &= b - Ax^{(0)} \\
\beta_i &= \frac{r^{(i+1)^T} r^{(i+1)}}{r^{(i)^T} r^{(i)}} \\
\alpha_i &= \frac{r^{(i)^T} r^{(i)}}{r^{(i)^T} A r^{(i)}} \\
d^{(i+1)} &= r^{(i+1)} + \beta_i d^{(i)} \\
x^{(i+1)} &= x^{(i)} - \alpha_i A d^{(i)} \\
r^{(i+1)} &= r^{(i)} - \alpha_i A d^{(i)}
\end{aligned}
$$

Conjugate Direction in Python

```
def conjugate_direction(A, b, tol=1e-6, max_iter=1000):
    x = np.zeros_like(b)
    r = b - A @ x
    d = r
    for i in range(max_iter):
        alpha = (r.T @ r) / (d.T @ A @ d)
        x = x + alpha * d
        r_new = r - alpha * A @ d
        beta = (r_new.T @ r_new) / (r.T @ r)
        d = r_new + beta * d
        r = r_new
        if np.linalg.norm(r) < tol:
            break
    return x
```

Discussion Conjugate Direction

- Is a direct solver: Korrekt solution after n iterations
- The finer the Mesh, the slower the convergence
- Can be improved with a Precondition-Matrix
  - Solve $M^{-1}Ax = M^{-1}b$ instead of $Ax = b$

Root finding of Nonlinear Systems of Equations

Bisecting Method

If a function $f(x)$ is continuous and has a root in the interval $[a, b]$, then the root can be found by bisecting the interval.

Algorithm:

1. Choose a and b such that $f(a) \cdot f(b) < 0$
2. Choose $c = \frac{a+b}{2}$ and calculate $f(c)$
3. Determine the new interval, where the root is located. $[a, c]$ or $[c, b]$
4. Continue until $|f(c)| < \epsilon$

Bisecting Method in Python

```
def bisect(f, a, b, tol=1e-6, max_iter=1000):
    for i in range(max_iter):
        c = (a + b) / 2
        if f(c) == 0 or (b - a) / 2 < tol:
            break
        if f(c) * f(a) < 0:
            b = c
        else:
            a = c
    return c
```

This method converges globaly-linearly. But its easy to compute.

Regula Falsi Method

The Regula Falsi Method works similar to the Bisecting Method, but uses a different formula to calculate the midpoint.

The new midpoint is calculated by the intersection of the line between $(a, f(a))$ and $(b, f(b))$ and the x-axis.

Regula Falsi Method in Python

```
def regula_falsi(f, a, b, tol=1e-6, max_iter=1000):
    for i in range(max_iter):
        c = (a * f(b) - b * f(a)) / (f(b) - f(a))
        if f(c) == 0 or (b - a) / 2 < tol:
            break
        if f(c) * f(a) < 0:
            b = c
        else:
            a = c
    return c
```

This method converges globaly-linearly. But its easy to compute.

Secant Method

The Secant Method works by calculating the tangent of the function at the last two points and calculating the intersection with the x-axis.

Algorithm:

1. Choose x_0 and x_1 and calculate $f(x_0)$ and $f(x_1)$
2. Draw a line between $(x_0, f(x_0))$ and $(x_1, f(x_1))$. Let x_2 be the intersection with the x-axis.
3. Continue with x_1 and x_2 ...

Secant Method in Python

```
def secant(f, x0, x1, tol=1e-6, max_iter=1000):
    for i in range(max_iter):
        x2 = x1 - f(x1) * (x1 - x0) / (f(x1) - f(x0))
        if f(x2) == 0 or abs(x2 - x1) < tol:
            break
        x0 = x1
        x1 = x2
    return x2
```

This method converges globaly-linearly. And locally with a rate of 1.618.

Newton-Raphson Method

The Newton-Raphson Method works by using the derivative of the function to calculate the next point.

Algorithm:

1. Choose x_0 and calculate $f(x_0)$ and $f'(x_0)$
2. Draw a line through $(x_0, f(x_0))$ with slope $f'(x_0)$. Let x_1 be the intersection with the x-axis.
3. Continue with x_1 and $f(x_1)$ and $f'(x_1)$ ...

Newton-Raphson Method in Python

```
def newton_raphson(f, df, x0, tol=1e-6, max_iter=1000):
    for i in range(max_iter):
        x1 = x0 - f(x0) / df(x0)
        if f(x1) == 0 or abs(x1 - x0) < tol:
            break
        x0 = x1
    return x1
```

This method converges localy-quadraticly. But it requires the derivative of the function. So it requires two function evaluations per iteration.

Multidimensional Root Finding

When dealing with multiple dimensions, the concept of a Derivative needs to be extended to a Jacobian Matrix.

Jacobian Matrix

The Jacobian Matrix is a matrix of partial derivatives of a function.

$$f\left(\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}\right) = \begin{bmatrix} f_1(x_1, x_2) \\ f_2(x_1, x_2) \end{bmatrix} = \begin{bmatrix} x^2 y \\ 5x + \sin(y) \end{bmatrix}$$

$$F' = J = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} \end{bmatrix} = \begin{bmatrix} 2xy & x^2 \\ 5 & \cos(y) \end{bmatrix}$$

Newton-Raphson Method for Multidimensional Systems

Analog to the 1D Newton-Raphson Method, the multidimensional Newton-Raphson Method works by using the Jacobian Matrix to calculate the next point.

In 1D, the next point was calculated by:

$$x^{(i+1)} = x^{(i)} - \frac{f(x^{(i)})}{f'(x^{(i)})}$$

In Higher Dimensions, the next point is calculated by:

$$x^{(i+1)} = x^{(i)} - (F'(x^{(i)}))^{-1} \cdot F(x^{(i)})$$

But inverting a matrix is computationally expensive. So the method uses LU-Decomposition to solve the equation.

Algorithm:

1. Choose x_0 and calculate $F(x_0)$ and $F'(x_0)$
2. Factorize $F'(x_0) = LU$.
3. Solve $Ly = -F(x_0)$ and $Us = y$.
4. Update x_0 with $x_0 + s$.

Newton-Raphson Method for Multidimensional Systems in Python

```
def newton_raphson_multidim(f, J, x0, tol=1e-6, max_iter=1000):
    for i in range(max_iter):
        L, U, _ = scipy.linalg.lu(J(x0))
        y = scipy.linalg.solve_triangular(L, -f(x0), lower=True)
        s = scipy.linalg.solve_triangular(U, y)
        x0 = x0 + s
        if np.allclose(f(x0), 0, atol=tol):
            break
    return x0
```

This method converges localy-quadraticly. But it requires the Jacobian Matrix of the function.

It is very expensive as it needs to solve a linear system for each iteration.

Improved Newton-Raphson Methods for Multidimensional Systems

1. Newton chords method
   - Idea: Reuse the Jacobian Matrix for several iterations.
2. inexact Newton method
   - Idea: Instead of solving the linear system exactly, solve it approximately. with a inner loop.
3. quasi-Newton method
   - Idea: Approximate $F'(x)$ and update it in each iteration.

Multigrid Methods

Multigrid Methods are a class of iterative methods that are used to solve linear systems.

It is used to improve Relaxation Methods.

Our previous Relaxation Methods were pretty good at reducing high frequencies noise, but failed to reduce low frequencies noise efficiently.

Main Idea: Reduce high frequencies noise with a coarse grid and few iteration steps. The create a finder grid and use the coarse grid as a starting point. On this finer grid, the low-frequency noise from bevore becomes high-frequency noise. And can again be reduced with a few iteration steps. Repeat this process until the desired accuracy is reached. And prolongate the solution up to the finer grids.

Algorithm:

Given:

- A_f, A_c describing the linear system on the fine and coarse grid.
- b_f, b_c describing the right hand side on the fine and coarse grid.

1. Create a coarse grid Ω_c and a finer grid Ω_f. with width $h_c = 2h_f$.
2. Smooth the solution x_c on Ω_c with a few iteration steps.

3. Compute the residual $r_f = b_f - A_f x_c$.
4. Interpolate r_f to Ω_c.
5. Find e_c such that $A_c e_c = r_c$.
6. prolongate e_c to Ω_f.
7. subtract e_f from x_f. And repeat from step 2.

Eigenvalue Problem

Eigenvalues are very important in numerics. For example when solving Linear Systems of Equations, the eigenvalues of the matrix are used to determine the condition of the system.

For symmetric A, the kondition number is given by:

$$\kappa(A) = \left| \frac{\lambda_{\max}(A)}{\lambda_{\min}(A)} \right|$$

it measures the maximum ratio of the relative error accumulated in the computation.

Symmetric Eigenvalue Problem

The symmetric eigenvalue problem is defined as:

$$A\mathbf{x} = \lambda\mathbf{x}$$

where A is a symmetric matrix and $\mathbf{x}$ is a vector. It is well-conditioned.

Naive Algorithm

The Standard Algorithm for solving eigenvalue problems is the characteristic polynomial method:

$$p(\lambda) = \det(A - \lambda I) = 0$$

The roots of the polynomial are the eigenvalues of A.

This way of solving is ill-conditioned.

Vector Iteration

Power Method

The power method is a simple iterative method for finding the largest eigenvector of a matrix A.

It works by calculating the matrix-vector product $A^k\mathbf{x}_0$ and normalizing the result.

The algorithm is as follows:

1. Initialize $\mathbf{x}_0$ to a random unit-vector
2. $\omega^{(k)} = Ax^{(k)}$
3. $x^{(k+1)} = \frac{\omega^{(k)}}{\|\omega^{(k)}\|}$ (normalize)
4. Repeat steps 2 and 3 until convergence

The korrisponding eigenvalue is given by:

$$\lambda^{(k)} = (x^{(k)})^T Ax^{(k)}$$

This method requires $O(n^2)$ operations per iteration. The solution konvergess **linearly** (q=$\frac{\lambda_2}{\lambda_1}$) to the eigenvektor with the largest eigenvalue. The convergence of the eigenvalue is **quadratic**.

The Convergence rate can be improved by shifting the eigenvalues. This is done by calculating the eigenvalues of $A - \sigma I$ and shifting the eigenvalues by σ.

Ideally $\sigma \approx (\lambda_2 + \lambda_f)/2$. Where λ_f is the eigenvalue furthest from λ_2.

Inverse Iteration

The inverse iteration is a simple iterative method for finding **specific** eigenvalue of a matrix A. Given a guess μ.

Idea: Perform the power method on the $(A - \mu I)^{-1}$ matrix. After reshifting, this finds the closest eigenvector to μ.

The algorithm is as follows:

1. Initialize $\mathbf{x}_0$ to a random unit-vector
2. Solve $(A - \mu I)\omega^{(k)} = x^{(k)}$
3. $x^{(k+1)} = \frac{\omega^{(k)}}{\|\omega^{(k)}\|}$ (normalize)
4. Repeat steps 2 and 3 until convergence
5. Calculate $\lambda^* = (x^{(k)})^T A x^{(k)} + \mu$

This method is dominated by the cost of solving the linear system. However: The matrix stays the same, so using LU decomposition or Cholesky decomposition the cost of solving the linear system can be reduced to $O(n^2)$.

It converges **linearly** to the eigenvektor with the eigenvalue closest to μ. The convergence of the eigenvalue is **quadratic**.

Rayleigh Quotient Iteration

The Rayleigh Quotient Iteration is a simple iterative method for improving a guess for the eigenvalue of a matrix A.

Idea: Use the Rayleigh Quotient to compute a better guess for the eigenvalue. It uses the incresingly better approximation of the eigenvalue to improve the guess.

The algorithm is as follows:

1. Initialize $\mathbf{x}_0$ to a random unit-vector
2. $\mu^{(k)} = (x^{(k)})^T A x^{(k)}$
3. Solve $(A - \mu^{(k)} I)\omega^{(k)} = x^{(k)}$
4. $x^{(k+1)} = \frac{\omega^{(k)}}{\|\omega^{(k)}\|}$ (normalize)
5. Repeat steps 2 and 3 until convergence

This method converges **linearly** to the eigenvektor with the eigenvalue closest to μ. But the convergence of the eigenvalue is now **qubically**.

However, the Matrix changes at every iteration, so the cost of solving the linear system is $O(n^3)$.

QR Iteration

The QR Iteration is a simple iterative method for finding all eigenvalues of a matrix A.

Idea: Compute similar matrices A_k wich converge to a diagonal matrix. The diagonal elements are the eigenvalues.

The algorithm is as follows:

1. Initialize $A_0 = A$
2. Calculate $A = QR$ with Q orthogonal and R upper triangular
3. $A_{k+1} = RQ$
4. Repeat steps 2 and 3 until convergence
5. The eigenvalues are the diagonal elements of A_k

QR Decomposition

The Idea of the QR Decomposition is to decompose a matrix A into a product of an orthogonal matrix Q and an upper triangular matrix R.

This is equivalent to finding an orthogonal Matrix G such that $GA = R$, because it follows that $A = G^T R$.

It works by applying a sequence of rotations (Givens-Rotations) to zero out subdiagonal elements of A.

Example $\mathbb{R}^2$:

- $A = \begin{bmatrix} x_1 & x_2 \\ x_3 & x_4 \end{bmatrix}$
- $G = \frac{1}{\sqrt{x_1^2+x_2^2}} \begin{bmatrix} x_1 & x_2 \\ -x_2 & x_1 \end{bmatrix}$
  - G describes the rotation of the first column of A to the x_1-axis ($\theta = \arctan\left(\frac{x_2}{x_1}\right)$)

In higher dimensions the Rotation works similar. In each step the Matrix is rotated by a 2d-rotation, to zero out the subdiagonal elements.

Cost

The Naive way of performing QR-Decomposition is $O(n^5)$. Because you need to apply $O(n^2)$ matrix-multiplications. But you can reduce the cost to $O(n^3)$ because only two rows are changed at a time due to the rotations.

The Convergence of the QR-Iteration to the eigenvalues is **linear**. If you combine this method with shift-operations you can improve the convergence to **quadratic**.

The corrisponding Eigenvectors can be computed by using inverse iteration with the corresponding eigenvalue.

Improving the Convergence

All the Methods above, are relatively slow, because the Matrix has many non-zero elements. So we can improve the convergence by transforming the problem into a sparse matrix.

Householder Transformations

Using reflections, we can transform a matrix into a matrix with a lot of zeros. This is done by applying a sequence of Householder transformations to A.

New Algorithm:

1. Transfom A into a triangular matrix A' using Householder transformations ($O(n^3)$)
2. Perform QR-Iteration on A' ($O(n^2)$ because only one diagonal row needs to be zeroed out)