

TUM NumProg, WiSe 2022/2023

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

Zuletzt aktualisiert: 11. Januar 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://github.com/ManuelLerchner/numprog/releases/download/Release/merge.pdf> heruntergeladen werden.

## 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

Introduction . . . . .	1
About . . . . .	1
How to Contribute . . . . .	1
Contributors . . . . .	1
<b>Floating-Point</b>	<b>4</b>
Fixed-Point . . . . .	4
Representation . . . . .	4
Floating-Point . . . . .	4
Representation . . . . .	4
Kleinste bzw. größte Zahl . . . . .	4
Maximalen relativen Abstand zweier Float-Zahlen . . . . .	4
Rundung . . . . .	5
Rundungsmodi . . . . .	5
Rundungsfehler . . . . .	5
<b>Fehleranalyse</b>	<b>7</b>
Vorwärts Fehleranalyse . . . . .	7
Rückwärts Fehleranalyse . . . . .	7
<b>Kondition</b>	<b>8</b>
Akzeptable Ergebnisse . . . . .	8
Numerische Stabilität . . . . .	8
<b>Polynom-Interpolation</b>	<b>10</b>
Problem . . . . .	10
Lagrange Polynomials . . . . .	10
Chebyshev Polynomials . . . . .	10
Bernstein Polynomials . . . . .	11
Variationen der Problemstellung . . . . .	11
Algorithmen . . . . .	11
Schema von Aitken und Neville . . . . .	11
Newton Interpolation . . . . .	12
Kondition von Interpolationspolynomen . . . . .	13
<b>Polynom - Splines</b>	<b>14</b>
Definition . . . . .	14
Kubische Splines . . . . .	14
<b>Trigonometrische Interpolation</b>	<b>16</b>
Definition . . . . .	16
Diskrete Fourier Transformation . . . . .	16
Inverse Diskrete Fourier Transformation . . . . .	16
Fast Fourier Transformation . . . . .	17
<b>Numerische Quadratur</b>	<b>18</b>
Problem . . . . .	18
Kondition der numerischen Quadratur . . . . .	18

Algorithmen . . . . .	18
Rechteckregel . . . . .	18
Trapezregel . . . . .	19
Kepler'sche Regel . . . . .	19
Trapezregel mit mehreren Teilintervallen . . . . .	19
Simpson'sche Regel . . . . .	19
Nicht gleichmäßige Gitter . . . . .	19
Extrapolation . . . . .	19
Monte Carlo Integration . . . . .	20
Gaussian Quadrature . . . . .	20
Archimedes Quadrature . . . . .	20
<b>Lineare Gleichungssysteme</b>	<b>21</b>
Arten von Matrizen . . . . .	21
Volle Matrizen . . . . .	21
Sparse Matrix . . . . .	21
Lösungsverfahren . . . . .	21
Vektor Normen . . . . .	22
Matrix Normen . . . . .	22
Kondition Lösen von Gleichungssystemen . . . . .	22
Das Residuum . . . . .	22
LR-Zerlegung . . . . .	23
Cholesky-Zerlegung . . . . .	23
Pivotsuche . . . . .	23
<b>Ordinary Differential Equations</b>	<b>24</b>
Definition . . . . .	24
Initial Conditions . . . . .	24
Boundary Conditions . . . . .	24
Analytical Solution . . . . .	24
Lipschitz Condition . . . . .	24
Condition of Differential Equations . . . . .	25
Differential Equations as Integration Problems . . . . .	25
Numerical Solutions . . . . .	25
Finite Difference Method / Euler Method . . . . .	25
Method of Heun . . . . .	25
Runge-Kutta Method . . . . .	26
Consistency and Convergence . . . . .	26
Local Discretization Error . . . . .	26
Global Discretization Error . . . . .	26
Multistep Methods . . . . .	26
Adams-Bashforth Method . . . . .	27
<b>Iterative Methods</b>	<b>28</b>
Introduction . . . . .	28
Relaxation Methods . . . . .	28
Richardson Iteration . . . . .	28
Jacobi Iteration . . . . .	28
Gauss-Seidel Iteration . . . . .	29
SOR Iteration . . . . .	29
Konvergence of those Methods . . . . .	30
Minimization Methods . . . . .	30
Method of Steepest Descent . . . . .	30
Conjugate Direction Method . . . . .	30

# Floating-Point

## Fixed-Point

### Representation

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

### Representation

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
- $\mathbb{F}_{B,t} = \{M \cdot B^E \mid M, E \in \mathbb{Z} \wedge M \text{ ohne führende Nullen bzw: } B^{t-1} \leq M < B^t\}$
- $\mathbb{F}_{B,t,\alpha,\beta} = \{M \cdot B^E \mid M, E \in \mathbb{Z} \ \& \ M \text{ ohne führende Nullen} \wedge \alpha \leq E < \beta\}$
- Wobei gilt:
- $B$  Basis
  - $t$  Anzahl der signifikanten Stellen
  - $\alpha$  kleinster möglicher Exponent
  - $\beta$  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:

- $rd : \mathbb{R} \rightarrow \mathbb{F}$
- 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  $\epsilon$ -Werte betragsmäßig durch die Maschinengenauigkeit  $\bar{\epsilon}$  begrenzt sind. Diese entspricht je nach verwendetem Rundungsmodus entweder  $\varrho$  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 unabhängig 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 Allgemeinen 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  $\tilde{y} \in \{f(\tilde{x}) \mid \tilde{x} \text{ nahe von } x\}$

## 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  $\xi$  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 verschiedenen Anwendungen der Interpolation welche auftreten können:

### 1. Simple Nodes

- Man hat eine Menge von Punkten  $P = \{(x_0, y_0), (x_1, y_1), \dots, (x_n, y_n)\}$ , welche durch ein Polynom interpoliert werden sollen.
- Diese Variante wird auch als *Lagrange Interpolation* bezeichnet.

### 2. Multiple Nodes

- Man hat eine Menge von Knoten  $P = \{(x_0, y_0, y'_0), (x_1, y_1, y'_1), \dots, (x_n, y_n, y'_n)\}$  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.
  - $p_{0,0} = y_0, p_{1,0} = y_1, \dots, p_{n,0} = y_n$
2. Verfeinere rekursiv die Polynome durch die Kombination 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	...
$x_0$	0	$p[0,0] = y_0$	$\rightarrow p[0,1]$	$\rightarrow p[0,2]$	$\rightarrow \dots$
			$\nearrow$	$\nearrow$	
$x_1$	1	$p[1,0] = y_1$	$\rightarrow p[1,1]$	$\rightarrow \vdots$	
			$\nearrow$		
$x_2$	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
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
  - $[x_i, x_{i+1}, \dots, x_{i+k}]f = \frac{[x_{i+1}, \dots, x_{i+k}]f - [x_i, \dots, x_{i+k-1}]f}{x_{i+k} - x_i}$

Die Interpolation ist dann gegeben durch:

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

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	...
$x_0$	0	$c_{0,0} = y_0$	$\rightarrow c_{0,1}$	$\rightarrow c_{0,2}$	$\rightarrow \dots$
			$\nearrow$	$\nearrow$	
$x_1$	1	$c_{1,0} = y_1$	$\rightarrow c_{1,1}$	$\rightarrow \vdots$	
			$\nearrow$		
$x_2$	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:

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

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

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

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 hilft 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:  $z_t = e^{2\pi i t} \quad t \in [0, 1]$

Das resultierende Polynom hat die Form:

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

## Diskrete Fourier Transformation

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

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

Man erhält jetzt also die Werte  $[(f_0, c_0), (f_1, c_1), \dots, (f_{n-1}, c_{n-1})]$ . 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  $c_i$ .

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

## 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 numerische 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:

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

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:

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

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 Funktion in Jedem Intervall durch ein Parabel-Polynom abgeschätzt.

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

$$Q_{SS}(f) := \frac{h}{6} \cdot (f_0 + 4 \cdot f_1 + 2 \cdot f_2 + 4 \cdot f_3 + \dots + 2 \cdot f_{n-2} + 4 \cdot f_{n-1} + f_n)$$

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.

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

## Extrapolation

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

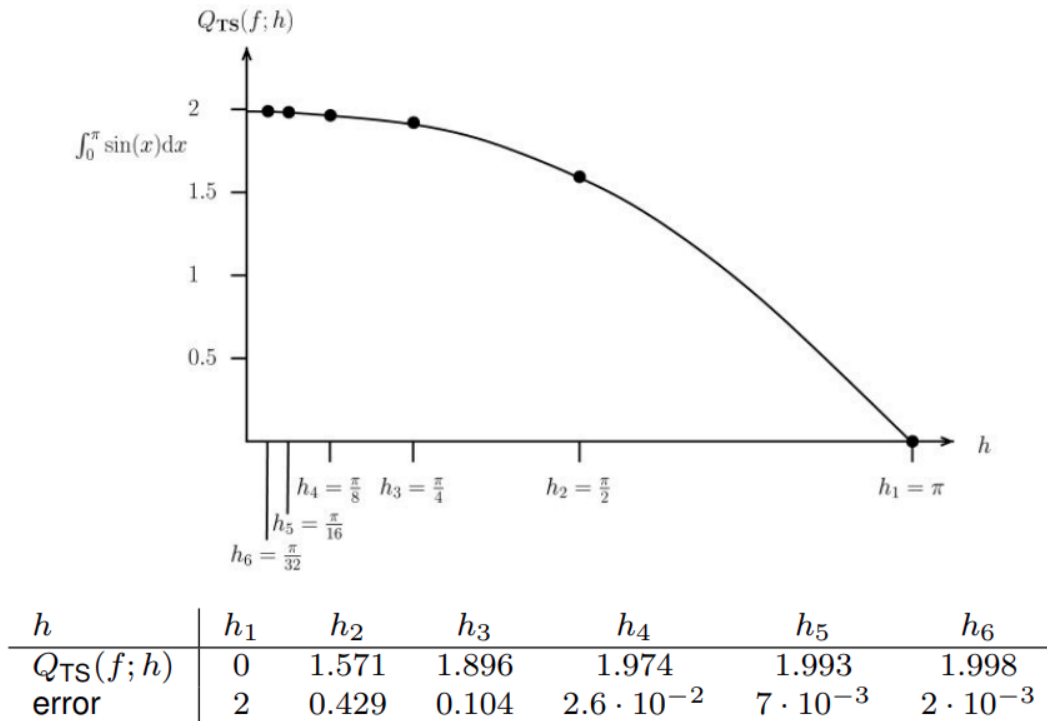


Abbildung 6: Extrapolation

## 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 Gleichungssystem 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:

$$\begin{aligned} \textit{Diagonalmatrix} &= \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \\ \textit{Tridiagonalmatrix} &= \begin{pmatrix} 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 \end{pmatrix} \\ \textit{BandedMatrix} &= \begin{pmatrix} 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 1 \end{pmatrix} \end{aligned}$$

## 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:

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

Beispiele für Vektor Normen:

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

## Matrix Normen

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

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

Die Konditionszahl 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  $\epsilon$  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 unbedingt 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 Dreiecksmatrix, 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:

$$\begin{aligned}y(0) &= 1 \\ \dot{y}(0) &= 0\end{aligned}$$

## Boundary Conditions

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

$$\begin{aligned}y(0) &= 1 \\ y(1) &= 0\end{aligned}$$

## 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:

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

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

The Error of the Euler method is:

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

### 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 corresponds with the trapezoidal rule for numerical integration.

The Error of the method of Heun is:

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

## 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

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

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

The Error of the Runge-Kutta method is:

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

## 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 over 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 arises 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  $\lim_{\delta t \rightarrow 0} l_{\delta t} = 0$  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  $\lim_{\delta t \rightarrow 0} e_{\delta t} = 0$  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 occurring 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 appealing matrices also have a specific shape (sparse, diagonal-dominant,...) which can be exploited to improve the performance of the methods.

## 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:

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

```
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, because the 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  $A$ .

```
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 doesn't use a **frozen** 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  $A$ .

```
def gauss_seidel(A, b, tol=1e-6, max_iter=1000):
    x = np.zeros_like(b)
    for i in range(max_iter):
        for k in range(len(b)):
            r = b[k] - A[k, :k] @ x[:k] - A[k, k + 1 :] @ x[k + 1 :]
            y = r / A[k, k]
            x[k] = x[k] + y
        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  $\alpha$  to improve the convergence speed.

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

## Konvergence of those Methods

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

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

In generell, 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:

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

## 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.

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

```
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 * A @ r
        r = b - A @ x
        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.)

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

```

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 / (r.T @ A @ r)
        x = x - alpha * A @ d
        r = b - A @ x
        beta = r.T @ r / (r.T @ A @ r)
        d = r + beta * d
        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  $Ax = b$