# TUM NumProg, WiSe 2022/2023

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

Zuletzt aktualisiert: 27. November 2022

# 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

# 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
- $\mathbb{F}_{B,t} = \{M \cdot B^E \mid M, E \in \mathbb{Z} \land \ M \text{ ohne führende Nullen bzw: } B^{t-1} \leq M < B^t\}$
- $\mathbb{F}_{B,t,\alpha,\beta} = \{M \cdot B^E \mid M, E \in \mathbb{Z} \ \& \ \ M \text{ ohne führende Nullen} \land \alpha \leq E < \beta\}$
- Wobei gilt:
- $B$ Basis - $t$ Anzahl der Bits - $\alpha$ 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: - $\sigma = 10^{4-1} \cdot 10^{-2} = 10$ - $\lambda = (10^4 - 1) \cdot 10^1 = 99990$

### Formel zur Berechnung des maximalen relativen Abstands zwei Float-Zahlen

Die Resolution 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:

- Rundet immer 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 gerade Zahl 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. - Beim korekten 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)

- $a \dot{*} b = f(a * b) = (a * b) \cdot \epsilon(a, b)$ - 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 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 eh 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 sind schlecht konditioniert, da sie bei ca. gleich großen Zahlen zu einem extremen Rundungsfehler führt.

Beispiele für gute und schlechte Kondition:

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

## Anwendung der Kondition bei konkreten Algorithmen

### 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.

Beispiele:

- 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

Given a potentially complicated function $f(x)$, find a function $p(x)$ that is ease to construct and handle. $p(x)$ needs to approximates $f(x)$ and should have a *small* error.

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:

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

Damit lassen sich die dazu gehörigen Koeffizienten berechnen:

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

Die Interpolation ist dann gegeben durch:

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

### Bernstein Polynomials

Mithilfe der Bersntein 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.

## Schema von Aitken und Neville

Wenn man nicht an der expliziten Representation des Interpolationspolynoms interessiert ist, sonder 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.
   - $p_{i,j} = \frac{x_{i+k} - x}{x_{i+k} - x_i} \cdot p_{i,k-1}(x) + \frac{x - x_i}{x_{i+k} - x_i} \cdot p_{i+1,k-1}(x)$

Als Psuedo Code:

```
def neville(x, x_values, y_values):
    n = len(x_values)

    p = np.zeros((n, n))
    p[:, 0] = y_values

    for k in range(1, n):
        for i in range(n - k):
            p[i, k] =   (x[i + k] - x) / (x[i + k] - x[i]) * p[i, k - 1] +
                        (x - x[i])     / (x[i + k] - x[i]) * p[i + 1, k - 1]
    return p[0, n - 1]
```

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

# 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 $\mathcal{O}(h^{n+1})$. Wo $h$ die Distanz zwischen den Stützstellen ist.

# 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 oszilieren kann.

# Polynom - Splines