# Numerisches Programmieren (IN0019)

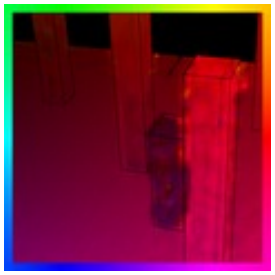
**Frank R. Schmidt**

Winter Semester 2016/2017

## Computer Vision

**Wir suchen immer nach Bachelor-Studenten!**

3D-Rekonstruktionen



Optischer Fluß

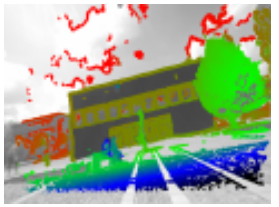
Shape Analyse

Echtzeit-Anwendungen

RGBD-Daten

Bildsegmentierung

Konvexe Vereinfachungen



Visuelles SLAM

Bitte sprechen Sie die entsprechenden Kontaktpersonen direkt an.

# 1. Einleitung



## Ablauf



**IN0019**

Die Vorlesung **Numerisches Programmieren** ist wie folgt organisiert:

- **Vorlesung:** Dienstag, 15:30–17:00 im Hörsaal 00.02.001
- **Tutorübung:** Montag – Freitag (eine Woche später)

Die Übungen kombinieren theoretische und praktische Aufgaben.

- **Aufgabenverteilung:** Die Aufgaben werden in die Übungen verteilt
- **Onlinezugang:** Die Aufgaben werden auch online zur Verfügung gestellt.
- **Aufgabenbesprechung:** Aufgaben werden in den Übungen besprochen

**Oktober**

| Oktober 2016 | | | | | | |
|---|---|---|---|---|---|---|
| Montag | Dienstag | Mittwoch | Donnerstag | Freitag | Samstag | Sonntag |
| | | | | | 1 | 2 |
| 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 10 | 11 | 12 | 13 | 14 | 15 | 16 |
| 17 | 18 | 19 | 20 | 21 | 22 | 23 |
| 24 | 25<br>Vorlesung 1 | 26 | 27 | 28 | 29 | 30 |
| 31 | | | | | | |

**November**

| November 2016 | | | | | | |
|---|---|---|---|---|---|---|
| Montag | Dienstag | Mittwoch | Donnerstag | Freitag | Samstag | Sonntag |
| | 1 Allerheiligen | 2 | 3 | 4 | 5 | 6 |
| 7 | 8 Vorlesung 2 | 9 | 10 | 11 | 12 | 13 |
| 14 | 15 Vorlesung 3 | 16 | 17 | 18 | 19 | 20 |
| 21 | 22 Vorlesung 4 | 23 | 24 | 25 | 26 | 27 |
| 28 | 29 Vorlesung 5 | 30 | | | | |

**Dezember**

| Dezember 2016 | | | | | | |
|---|---|---|---|---|---|---|
| Montag | Dienstag | Mittwoch | Donnerstag | Freitag | Samstag | Sonntag |
| | | | 1 | 2 | 3 | 4 |
| 5 | 6<br>Vorlesung 6 | 7 | 8 | 9 | 10 | 11 |
| 12 | 13<br>Vorlesung 7 | 14 | 15 | 16 | 17 | 18 |
| 19 | 20 | 21 | 22 | 23 | 24 | 25 |
| 26 | 27 | 28 | 29 | 30 | 31 | |

**Januar**

| Januar 2017 | | | | | | |
|---|---|---|---|---|---|---|
| Montag | Dienstag | Mittwoch | Donnerstag | Freitag | Samstag | Sonntag |
| | | | | | | 1 |
| 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 9 | 10 Vorlesung 8 | 11 | 12 | 13 | 14 | 15 |
| 16 | 17 Vorlesung 9 | 18 | 19 | 20 | 21 | 22 |
| 23 | 24 Vorlesung 10 | 25 | 26 | 27 | 28 | 29 |
| 30 | 31 Vorlesung 11 | | | | | |

## Februar

| Februar 2017 | | | | | | |
|---|---|---|---|---|---|---|
| Montag | Dienstag | Mittwoch | Donnerstag | Freitag | Samstag | Sonntag |
| | | 1 | 2 | 3 | 4 | 5 |
| 6 | 7 Vorlesung 12 | 8 | 9 | 10 | 11 | 12 |
| 13 | 14 | 15 | 16 | 17 | 18 | 19 |
| 20 | 21 | 22 Klausur | 23 | 24 | 25 | 26 |
| 27 | 28 | 29 | 30 | 31 | | |



## Klausur

Voraussetzung zur Klausurzulassung:

- **Anmeldung:** Jede(r) muss sich über **TUM Online** anmelden.
- **Termin:** Klausur findet am **22. Februar 2017** statt.
- **Wiederholung:** Es findet **keine Wiederholungsprüfung** statt, da die Vorlesung (inkl. Klausur) jedes Semester angeboten wird.

Übungsteilnahme:

- **Keine Klausurvoraussetzung, aber stark empfohlen:**
  Theoretische Übungen helfen, den Vorlesungsstoff besser zu verstehen.
  Programmieraufgaben helfen, die Theorie in die Praxis umzusetzen.
- **Bonus:** Aktive Studenten, die mindestens 70% der Programmieraufgaben lösen, erhalten einen Bonus.
- **Klausur:** Jede(r), der in der Klausur eine Note zwischen 1.3 und 4.0 erhält, erhält eine um 0.3 verbesserte Note, falls er/sie sich für den Bonus qualifiziert hat.

## Bonus

Um den Bonus zu erhalten, müssen die folgenden Voraussetzungen erfüllt sein:

**Theorie**

- Anwesenheit bei mindestens 8 der Tutorübungen.

**Programmieren**

- 70% der Punkte aller Programmieraufgaben müssen erreicht werden.

Um Team Work zu unterstützen, empfehlen wir in Gruppen von **zwei** oder **drei** StudentInnen zusammen zuarbeiten.

## Ansprechpartner

**Vorlesung**

Dr. Frank R. Schmidt

**Übungenzentralverwaltung**

Nikola Tchipev

Michael Rippl

Falls Fragen in den Tutorien nicht gelöst werden können, können Sie gerne per Email (mit CC an ihren Tutor) einen Termin ausmachen:

- `f.schmidt@in.tum.de`
- `n.tchipev@tum.de`
- `ripplm@in.tum.de`

## Digital

Kursmaterialien werden auf folgender Seite verwaltet
`http://www5.in.tum.de/wiki/index.php/Teaching`

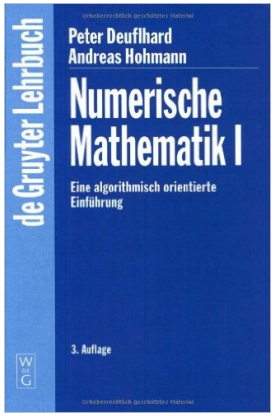
- Vorlesungsfolien (Vor der Vorlesung verfügbar)
- Übungsaufgaben (Nach der Vorlesung verfügbar)
- Musterlösungen (Nach der Freitagübung verfügbar)

**Diese Seite wird auch für außerordentliche Ankündigungen benutzt.**

Darüber hinaus werden die Vorlesungsfolien auf folgender Seite gespiegelt:
`https://vision.in.tum.de/teaching/ws2016/numerics`

## Analog


Huckle/Schneider:
"Numerische Methoden"



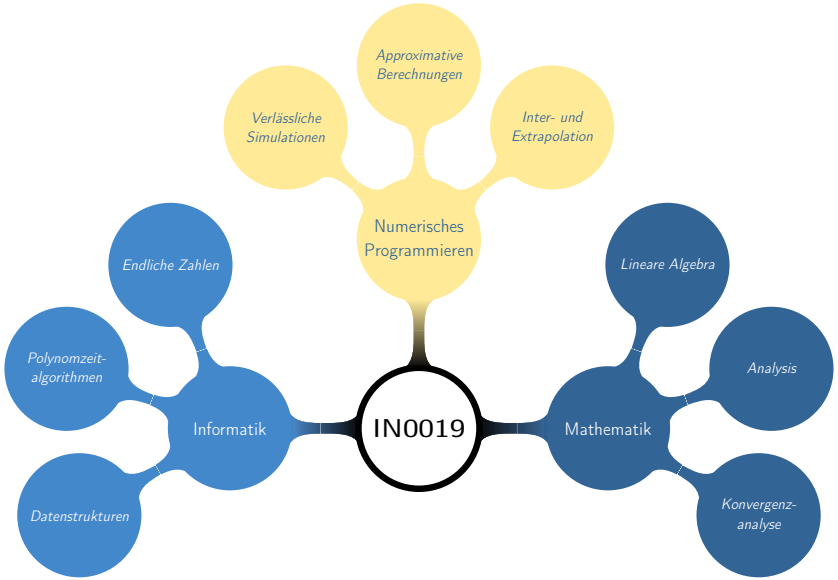

Deuflhard/Hohmann:
"Numerische Mathematik I"


Dahmen/Reusken:
"Numerik für Ingenieure
und Naturwissenschaftler"

Ergänzende Literatur:

- Herzberger: "Wissenschaftliches Rechnen"
- Opfer: "Numerik für Anfänger"
- Überhuber: "Computer-Numerik"
- Kahaner et al.: "Numerical Methods and Software"

## Einordnung der Vorlesung



## Numerik

### Numerische Mathematik

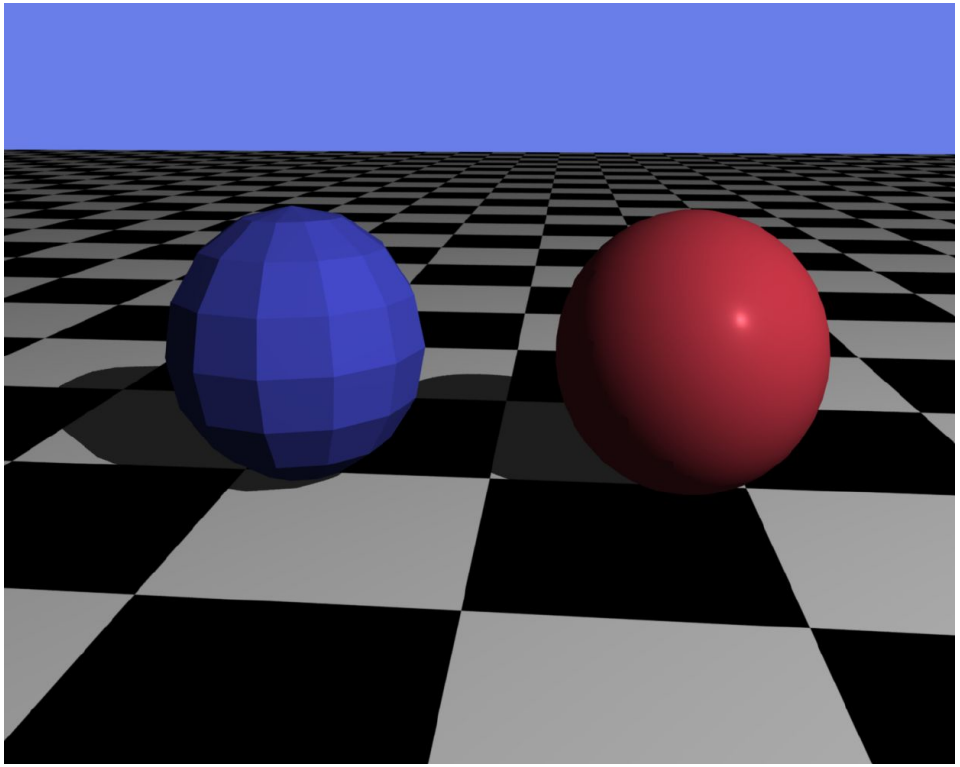
- Teil der angewandten Mathematik
- Entwicklung von Algorithmen für kontinuierliche Methoden
  Berechnung von Nullstellen, Lösen Linearer Gleichungssysteme,
  Integral- und Differenzialrechnung, etc.
- Analyse numerischer Algorithmen
  Speicher- und Zeitverbrauch, Genauigkeit, Konvergenzgeschwindigkeit

### Numerisches Programmieren

- Teil der praktischen Informatik
- Effiziente Implementierung von numerischen Algorithmen

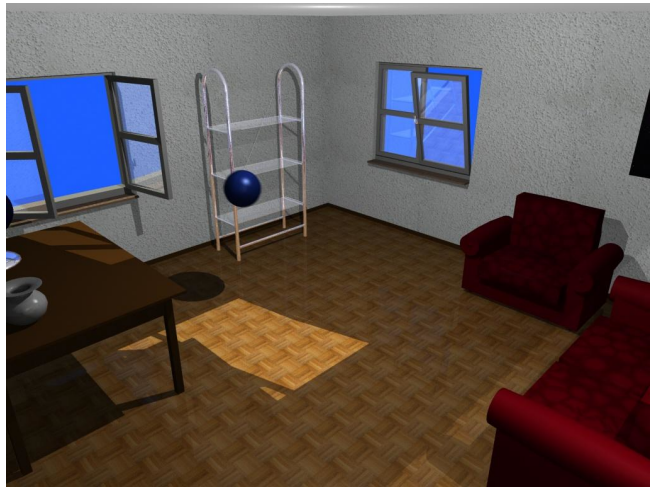
## Geometrische Modellierung

- Befasst sich mit geometrischen Objekten
- Vor allem bei glatten Oberflächen werden numerische Methoden benötigt.



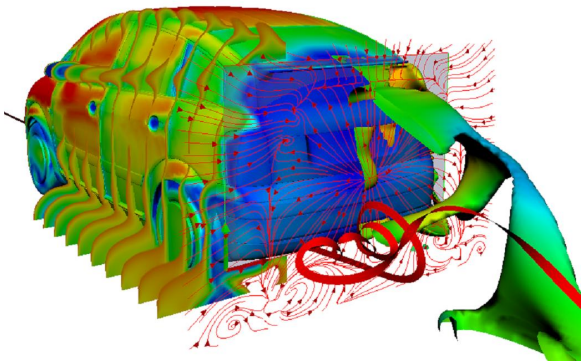
## Computergrafik

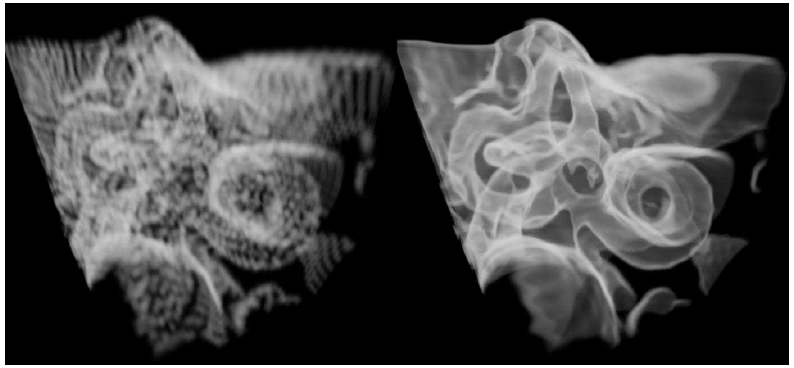
- Beim Ray-Tracing werden Reflexionen aus Schnittpunkten von Strahlen und Objekten berechnet.
- Berechnung von "diffuser Beleuchtung" löst lineare Gleichungssysteme.



## Visualisierung

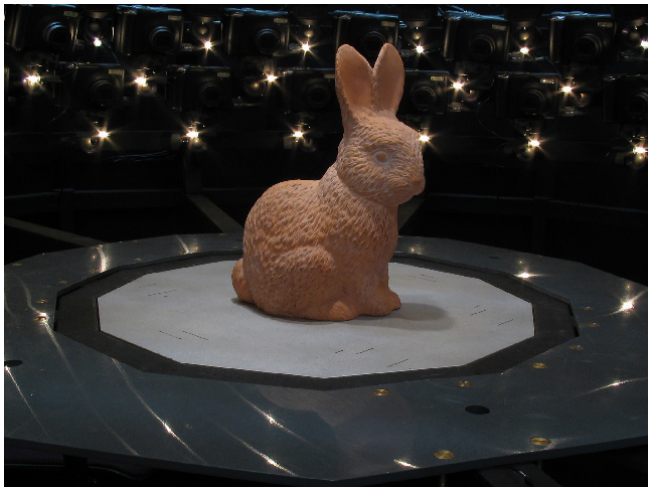
- Particle-Tracing als Visualisierung von simulierten Flüssen.
- Volumenvisualisierung wird z.B. in der Medizin verwendet.





## Computer Vision

- Computer Vision beschäftigt sich mit den inversen Problemen der Computergrafik. Ausgehend von gewissen Beobachtungen möchte man die 3D-Welt modellieren.
- Eine Interaktion von Computergrafik und Computer Vision findet z.B. im Bereich der Virtual Reality statt.



## Maschinenzahlen

Da wir nur endlichen Speicherplatz zur Verfügung haben, können wir nur endlich viele Zahlen speichern.

Ein Bit $b \in \mathbb{B} := \{0, 1\}$ kann nur zwei unterschiedliche Zahlen speichern.

Nutzen wir $n$ Bits, dann kann in dem Vektor $b = (b_{n-1}, \ldots, b_1, b_0) \in \mathbb{B}^n$ nur eine von maximal $2^n$ verschiedene Zahlen gespeichert werden.

Das heißt insbesondere

- Zahlen wie $\frac{1}{3}$, $\sqrt{2}$ oder $\pi$ können üblicherweise nur angenähert werden.
- Funktionen wie z.B. $\sin(\cdot)$, $\cos(\cdot)$ oder $\exp(\cdot)$ können nur ungefähr berechnet werden.
- Geometrische Objekte wie Kreise, Kugeln, etc. können nur durch endlich viele Punkte beschrieben werden. Die Koordinaten dieser Punkte wiederum können nur angenähert gespeichert werden.



## Maschinenzahlen

**Zusammengefasst:** Die Menge der reellen Zahlen $\mathbb{R}$ besteht aus unendlich vielen Zahlen, aber **jeder Computer ist nur endlich**.

**Definition 1.** Die endliche Menge $\mathbb{M} \subset \mathbb{R}$ der in einem Rechner darstellbaren Zahlen heißt **Menge der Maschinenzahlen**.
Der **Bereich** von $\mathbb{M}$ ist das abgeschlossene Intervall $I_{\mathbb{M}} := [\min \mathbb{M}; \max \mathbb{M}]$.

Wir unterscheiden **ganzzahlige** Maschinenzahlen
und Maschinenzahlen für reelle (nicht ganzzahligen) Zahlen.

Im Folgenden werden wir uns mit folgenden Datentypen befassen:

- `uint`$\subset \mathbb{N}$ kann nur nicht-negative, ganze Zahlen speichern.
- `int`$\subset \mathbb{Z}$ kann auch negative Zahlen speichern.
- `float`$\subset \mathbb{R}$ speichert auch nicht-ganzzahlige Zahlen.

## `uint`

Benutzen wir $n$ Bits, dann beschreibt der Vektor $b = (b_{n-1}, \ldots, b_1, b_0) \in \mathbb{B}^n$ die nicht-negative Zahl

$$n = \sum_{i<n} b_i \cdot 2^i \in \mathbb{N}$$

Die kleinste Zahl mit $n$ **Binärstellen** ist $0$ und wird durch $b = (0, \ldots, 0) \in \mathbb{B}^n$ dargestellt. Die größte Zahl mit $n$ Binärstellen ist $2^n - 1$ und wird durch $b = (1, \ldots, 1) \in \mathbb{B}^n$ dargestellt.

| Datentyp | `uint8` | `uint16` | `uint32` | `uint64` |
|---|---|---|---|---|
| Größte Zahl | $255$ | $65535$ | $\approx 4.294 \cdot 10^9$ | $\approx 18.466 \cdot 10^{18}$ |



## Besonderheiten von `uint`

Beim Rechnen mit `uint` sollte man Folgendes beachten:

**Überlauf** Fehler bei Bereichsüberschreitung, d.h. in `uint8` ist `128*2==0`.
Dies kann `for`-Schleifen in Endlosschleifen verwandeln:

```
for (uint8_t i=1; i<200; i*=2); // infinite loop in C++11
for (uint8_t i=1; i<200; i*=3); // finite but useless
```

**Division** Bei Divisionen werden Nachkommastellen abgeschnitten.
Es kann daher gelten `(x/n)*n != x`

**Null-Division** Divisionen durch 0 sind nicht definiert und führen zu Programmabrüchen bzw. Ausnahmesituationen (Exceptions)

**Negative Zahlen** Negative Zahlen können nicht dargestellt werden.

`int`

Um auch negative Zahlen zu speichern, gibt es zwei verschiedene Ansätze:

**Definition 2.** Benutzen wir $n$ Bits, dann beschreibt der Vektor $b = (b_{n-1}, \ldots, b_1, b_0) \in \mathbb{B}^n$ die **Integer-Zahl**

$$z = \left[ \sum_{i<n} b_i \cdot 2^i \right] - 2^{n-1} \in \mathbb{Z} \qquad \text{(mit Bias)}$$

$$z = \left[ \sum_{i<n-1} b_i \cdot 2^i \right] - b_{n-1} 2^{n-1} \in \mathbb{Z} \qquad \text{(Zweierkomplement)}$$

In beiden Darstellungen sind die **größten** und **kleinsten** darstellbaren Zahlen:

$$\texttt{INT\_MIN} = -2^{n-1} \qquad \texttt{INT\_MAX} = +2^{n-1} - 1$$

## Gewollter Überlauf beim Zweierkomplement

Angenommen, wir haben die Zahlen $z_1 = -13$ und $z_2 = 116$ im Zweierkomplement mit $8$ Bits gespeichert. Dann werden diese Maschinenzahlen wie folgt repräsentiert:

$$b_1 = (1\ 1\ 1\ 1\ 0\ 0\ 1\ 1)$$
$$b_2 = (0\ 1\ 1\ 1\ 0\ 1\ 0\ 0)$$

Interpretiert man diese Zahlen als `uint8`, so erhält man

$$u_1 = 243 \qquad u_2 = 116$$

Addiert man diese Zahlen, so erhält man nicht 359 sondern

$$359 - 256 = 103 = z_1 + z_2$$

Daher kann man für `int` die gleiche Addition benutzen wie für `uint`.

## Festkommazahlen

Festkommazahlen können als einfache Erweiterungen von `int` angesehen werden. Es wird das Vielfache der kleinsten positiven Zahl gespeichert.

**Definition 3.** Benutzen wir $n + v + 1$ Bits, dann beschreibt der Vektor $b = (s, b_{v-1}, \ldots, b_0, \ldots, b_{-n}) \in \mathbb{B}^{n+v+1}$ die **Festkomma-Zahl**

$$z = (-1)^s \sum_{i=-n}^{v-1} b_i \cdot 2^i \in \mathbb{R}$$

mit $v$ **Vorkommastellen** und $n$ **Nachkommastellen**.

Dies kann hilfreich sein bei der Speicherung von

- Geldbeträgen ($n = 2$)
- Entfernungen in km ($n = 3$)



## Probleme von Festkommazahlen

Festkommazahlen sind für praktische Berechnungen ungenügend!

Sehr große Zahlen und sehr kleine Zahlen sind oft nicht gleichzeitig darstellbar.

Die Verfügbarkeit von großen Zahlen schränkt die Nachkommastellen für kleine Zahlen ein.

Die Verfügbarkeit von vielen Nachkommastellen schränkt die maximal darstellbare Zahl ein.

Festkommazahlen sind daher nur eingeschränkt nutzbar.

Im Folgenden werden wir diese Zahlendarstellung **nicht benutzen**.

## Fließkommazahlen

Die Idee der Fließkommazahlen ist die folgende Zahlendarstellung

$$x = (-1)^s \cdot m \cdot b^e,$$

wobei

| | |
|---|---|
| $s \in \mathbb{B}$ | das Vorzeichen, |
| $m \in [0, b[$ | die Mantisse, |
| $e \in \mathbb{N}$ | den Exponenten und |
| $b \geqslant 2$ | die Basis |

kodieren.

Im Folgenden betrachten wir nur $b = 2$, da $b = 8, 10, 16$ eher selten vorkommen.



## Normierte Fließkommazahlen

Fließkommazahlen können dazu führen, dass eine Zahl verschiedene Darstellungen besitzt.

Die Zahl $1.0$ läßt sich wie folgt darstellen ($b = 2$)

| | | |
|---|---|---|
| $s = 0$ | $e = 0$ | $m = 1.000$ |
| $s = 0$ | $e = 1$ | $m = 0.500$ |
| $s = 0$ | $e = 2$ | $m = 0.250$ |

**Definition 4.** Eine Fließkommazahl $x = (-1)^s \cdot m \cdot b^e$ heißt **normierte Fließkommazahl**, wenn $m \in [1; b[$ gilt.

Für $b = 2$ gilt somit $m \in [1; 2[$. Da die Stelle vor dem Komma immer die "1" ist, müssen nur die Nachkommastellen gespeichert werden.

`float`

Nach dem IEEE-Standard werden auch $\pm\infty$ und `NaN` gespeichert.

**Definition 5.** Benutzen wir $1 + r + p$ Bits, dann kodiert der Vektor $b = (s, E, M) \in \mathbb{B} \times \mathbb{B}^r \times \mathbb{B}^p$ die Mantisse und den Exponenten

$$m = 1 + \sum_{i=1}^{p} M_i 2^{-i} \qquad e = \sum_{i<r} E_i 2^i - (2^{r-1} - 1)$$

Die Fließkommazahl $x$ ist dann

$$x = \begin{cases} (-1)^s \cdot \infty & \text{wenn } E = (1, \ldots, 1),\ M = (0, \ldots, 0) \\ \texttt{NaN} & \text{wenn } E = (1, \ldots, 1),\ M \neq (0, \ldots, 0) \\ 0 & \text{wenn } E = (0, \ldots, 0),\ M = (0, \ldots, 0) \\ (-1)^s \cdot m \cdot 2^e & \text{sonst} \end{cases}$$



## Genauigkeit

Die Genauigkeit einer Fließkommazahl hängt von der Anzahl der Bits ab, die zur Darstellung verwendet werden.

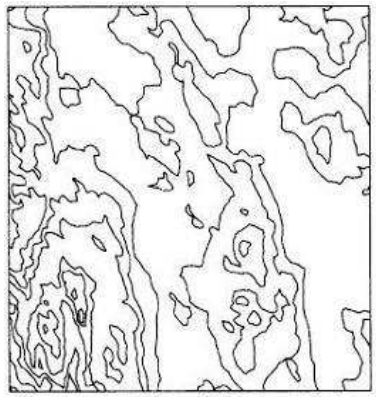
Der Standard *IEEE 754* definiert `float` mit einfacher Genauigkeit von 32 Bit und `double` mit doppelter Genauigkeit von 64 Bit. Darüber hinaus wird auf GPUs teilweise mit halber Genauigkeit von 16 Bit gearbeitet:

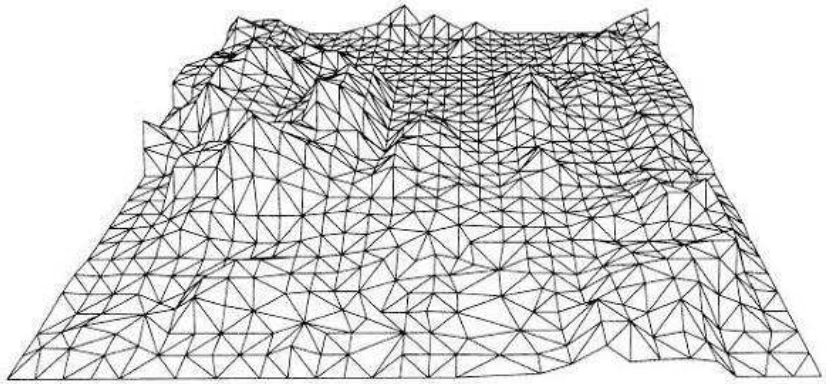
| Typ | Bit-Verteilung | $2^{e_{\min}}$ | $2^{e_{\max}}$ | $\Delta_m$ |
|---|---|---|---|---|
| `float` | $1 + 8 + 23$ | $2^{-126} \approx 1.18 \cdot 10^{-38}$ | $2^{127} \approx 1.70 \cdot 10^{38}$ | $2^{-23} \approx 1.19 \cdot 10^{-7}$ |
| `double` | $1 + 11 + 52$ | $2^{-1022} \approx 2.22 \cdot 10^{-308}$ | $2^{1023} \approx 8.99 \cdot 10^{307}$ | $2^{-52} \approx 2.22 \cdot 10^{-16}$ |
| `half` | $1 + 5 + 10$ | $2^{-14} \approx 6.10 \cdot 10^{-5}$ | $2^{15} \approx 3.28 \cdot 10^{4}$ | $2^{-10} \approx 9.77 \cdot 10^{-4}$ |

## Diskretisierung

Der Übergang von kontinuierlichen Zahlenraum zum endlichen Zahlenraum der Maschinenzahlen heißt **Diskretisierung**, z.B.

- **Runden** von reellen Zahlen auf Maschinenzahlen
- Darstellen von Oberflächen durch **Gitterpunkte**
- Ableiten durch Differenzenquotienten
- Integrieren durch Summieren





## Rundungen

**Definition 6** (Rundung). Es seien die Maschinenzahlen $\mathbb{M}$ gegeben. Dann heißt eine Abbildung $r\colon I_{\mathbb{M}} \to \mathbb{M}$ Rundung, falls gilt:

| | | |
|---|---|---|
| $r(m) = m$ | für alle $m \in \mathbb{M}$ | (Projektion) |
| $x \leqslant y \Rightarrow r(x) \leqslant r(y)$ | für alle $x, y \in I_{\mathbb{M}}$ | (Ordnungserhaltend) |

Folgende Funktionen sind Rundungen

$$\lfloor x \rfloor := \max\{m \in \mathbb{M} | m \leqslant x\}$$
$$\lceil x \rceil := \min\{m \in \mathbb{M} | m \geqslant x\}$$
$$\text{round}(x) := \begin{cases} \lfloor x \rfloor & \text{wenn } x < \frac{\lfloor x \rfloor + \lceil x \rceil}{2} \\ \lceil x \rceil & \text{wenn } x \geqslant \frac{\lfloor x \rfloor + \lceil x \rceil}{2} \end{cases}$$

## Rundungsfehler

Bei Rundungen treten Fehler auf:

$$\begin{aligned} f\colon I_{\mathbb{M}} &\to \mathbb{R} \\ x &\mapsto x - r(x) \end{aligned}$$

Diese Fehler entstehen bereits beim Speichern von Daten.

Bei jeder Berechnung entstehen weitere Rundungsfehler. Der Code `z = x * y;` erzeugt einen Rundungsfehler von

$$f(x \cdot y) \qquad \text{für alle } x, y \in \mathbb{M}.$$

Das heißt jede Rechenoperation erzeugt Rundungsfehler.



## Runden bei Fließkommazahlen

Da der Exponent ganzzahlig ist, gibt es für den Exponenten nur Überlauffehler:

- $e > e_{\max}$: Überlauffehler, der üblicherweise einfach abzufangen ist.
- $e < e_{\min}$: Unterlauffehler: Abfangen und Abrunden auf 0.

Der Fehler, der bei jeder Operation auftritt entsteht beim Speichern der Mantisse. Daher benutzt die Mantisse den größten Anteil des Speichers.

Üblicherweise wird intern mit einer größeren Mantisse gerechnet, so dass die Rundung exakt durchgeführt werden kann.

Der Mantissenfehler, der auftritt, ist durch $\Delta_m = \pm\frac{1}{2}2^{-p}$ beschränkt.

Der tatsächliche Fehler hängt natürlich noch vom Exponenten ab.

## Rundungsfehler

Sei $x = (-1)^s \cdot m \cdot 2^e$ eine Fließkommazahl.
Dann gilt für den **absoluten Rundungsfehler** $f(x)$ gerade

$$|f(x)| = |x - \text{round}(x)| \leqslant 2^{-(p+1)} 2^e = 2^{e-(p+1)}$$

Diese Fehleranalyse ist in der Praxis nicht sehr aussagekräftig.
Ein absoluter Fehler von $0.1$ ist bei $0.9$ sehr groß ($> 10\%$).
Bei $12345.67$ ist er eher klein ($< 0.001\%$).

Daher sind wir am **relativen Rundungsfehler**

$$\varepsilon_x := \frac{f(x)}{x}$$

interessiert.

## Relativer Rundungsfehler von Fließkommazahlen

Sei $x = (-1)^s \cdot m \cdot 2^e$ eine Fließkommazahl.
Dann gilt für den relativen Rundungsfehler $\varepsilon_x$ gerade

$$|\varepsilon_x| \leqslant \frac{2^{e-(p+1)}}{m \cdot 2^e} \leqslant 2^{-(p+1)}$$

Das können wir wie folgt umformen

$$\text{round}(x) = x - x \cdot \varepsilon_x = x \cdot (1 - \varepsilon_x),$$

wobei $|\varepsilon_x| \leqslant \frac{1}{2} 2^{-p}$.

**Die gerundete Zahl ist die Ausgangszahl bis auf $1 \pm \varepsilon$.**

## Maschinengenauigkeit

Wir wollen die Maschinengenauigkeit als den größten relativen Fehler definieren, der bei Rundungen auftreten kann.

**Definition 7.** Die **Maschinengenauigkeit** $\varepsilon_{\mathbb{M}} > 0$ ist die kleinste obere Schranke des Betrags des relativen Fehlers, der bei der Rundungen von Zahlen $x \in I_{\mathbb{M}}$ auftreten, d.h.

$$\varepsilon_{\mathbb{M}} = \sup_{x \in I_{\mathbb{M}}} \left| \frac{x - \mathsf{round}(x)}{x} \right|$$

**Lemma 1.** *Wird die Mantisse einer Fließkommazahl in $p$ Bits gespeichert, ist die Maschinengenauigkeit gerade*

$$\varepsilon_{\mathbb{M}} = 2^{-(p+1)}.$$