Numerik

by

Dr. Günter Kolousek

Numerik

- Numerische Mathematik
  - zahlenmäßige Berechnung von Lösungen mathematischer Modelle, die durch Formeln, Gleichungen, als Grenzwerte,… gegeben sind
  - Zahlendarstellung und Zahlenarithmetik
- Warum?
  - auf dem Papier nicht exakt berechenbar
  - zwar exakt berechenbar, aber
    - dies muss oftmalig passieren oder
    - auf diese Weise zu langsam oder zu große Rechenfehler $\rightarrow$ Näherungslösung u.U. sinnvoller (wenn ausreichend genau)

Akteure

- Numerische Mathematiker (Numeriker)
  - Aufbereitung der Modelle zur numerischen Behandlung
- Informatiker
  - Umsetzung
  - Stichworte: Rechenzeit, Speicherbedarf, Cache-Effekte, Parallelrechner, verwendete Rechnerarchitekturen, Compiler, Programmiertechniken

Anwendungen

- Meteorologie, Strömungsberechnungen, Brückenbau,…
- Simulationen (Crash-Tests, Flugzeug, Atomkraftanlagen, Chemieanlagen, Berechnungen für Energienetze,…)
- Computergraphik, Bildverarbeitung (Kompression, Analyse, Bearbeitung)
- Neuronale Netze (Lernverfahren)
- Steuerung von Raketen, Industrieroboter, Medizintechnik (z.B. Infusionspumpen), Eisenbahnsicherungsanlagen, Antiblockiersysteme,…
- Chip Design
- Kryptographie

Probleme der Rechnerarithmetik

- Die *endliche* Menge $M \subset \mathbb{R}$, der in einem Rechner darstellbaren Zahlen heißt Menge der Maschinenzahlen
  - Es muss eine größte (kleinste) Maschinenzahl geben
  - Nicht jede Zahl zwischen größter und kleinster Zahl aus M ist in M enthalten
    - z.B. $M = \{0, 1, 2, 3, 4, 5\}$, dann ist $\frac{2+3}{2} \notin M$
- Prozessor kann Rechnungen nicht exakt ausführen

Rechnen mit ganzen Zahlen

- Vergleich: VZ-behaftet und VZ-los
- Überlauf und Unterlauf
  - sowohl bei VZ-losen als auch bei VZ-behafteten Zahlen
  - Addition, Substraktion, Multiplikation
- Division
  - Division liefert im Allgemeinen keine ganzzahligen Ergebnisse
  - Division durch 0

Unsicherer Vergleich

```cpp
#include <iostream>

using namespace std;

int main() {
    int x{-3};
    unsigned int y{7};

    cout << boolalpha;
    cout << "-3 < 7: " << (x < y) << endl; // false
    cout << "-3 <= 7: " << (x <= y) << endl; // false
    cout << "-3 > 7: " << (x > y) << endl; // true
    cout << "-3 => 7: " << (x >= y) << endl; // true
} // x will be casted to unsigned!!!
```

Sicherer Vergleich (in C++ 20)

```cpp
#include <iostream>
#include <utility>

using namespace std;

int main() {
  int x{-3};
  unsigned int y{7};

  cout << boolalpha;
  cout << "3 == 7: " << cmp_equal(x, y) << endl;
  cout << "3 != 7: " << cmp_not_equal(x, y) << endl;
  cout << "-3 < 7: " << cmp_less(x, y) << endl;
  cout << "-3 <= 7: " << cmp_less_equal(x, y) << endl;
  cout << "-3 > 7: " << cmp_greater(x, y) << endl;
  cout << "-3 => 7: " << cmp_greater_equal(x, y)<< endl;
} // expected results!
```

Über/Unterlauf

Überlauf (overflow) und Unterlauf (underflow) bei ganzen Zahlen!

```cpp
#include <iostream>
#include <cstdint>
using namespace std;
int main() {
    // sadly, it's an alias for unsigned char...
    uint8_t i{253};
    cout << +i++ << endl;  // -> 253
    cout << +i++ << endl;  // -> 254
    cout << +i++ << endl;  // -> 255
    cout << +i-- << endl;  // -> 0
    cout << +i << endl;    // -> 255
    // ... so you can't omit the '+'!
    cout << i + i << endl; // -> 510
    i = i + i;
    cout << +i << endl;    // -> 254
}
```

Über/Unterlauf – 2

- meistens nicht gewünscht, d.h.
  - erkennen
  - vermeiden
- manchmal erwünscht, z.B.:
  - Timer, Clocks
  - gewisse Zähler (wie bei Ringpuffer)

Überlauf erkennen

- ganze Zahlen
  - Addition/Subtraktion positiver Zahlen → darf nicht kleiner/größer sein

```cpp
#include <iostream>
#include <cstdint>
using namespace std;

int main() {
    uint8_t i{254};
    uint8_t res;
    res = i + i;
    cout << +res << endl;  // -> 252
    res = res - i;
    cout << +res << endl;  // -> 254
}
```

Überlauf erkennen – 2

- Multiplikation positiver Zahlen
  - Produkt kann größer sein, aber trotzdem falsch:

```cpp
#include <iostream>
#include <cstdint>
using namespace std;

int main() {
    uint8_t i{90};
    uint8_t res;
    res = i * i;
    cout << +res << endl;  // -> 164
```

Überlauf erkennen – 3

- $\rightarrow$ in größerem Datentyp rechnen:

```
    uint16_t tmp;
    tmp = i * i;
    // erkennen und auf max setzen
    res = (tmp < 255) ? tmp : 255;
    cout << +res << endl;  // -> 255

    tmp = i * 2;
    res = (tmp < 255) ? tmp : 255;
    cout << +res << endl;  // -> 180
}
```

Überlauf erkennen – 4

- Gleitkommazahlen

```cpp
#include <iostream>
#include <limits>
#include <cmath>
using namespace std;

int main() {
    double x{};
    x = numeric_limits<double>::max();
    cout << x << " is inf: " << isinf(x) << end
    // -> 1.79769e+308 is inf: 0
    x = numeric_limits<double>::infinity();
    cout << x << " is inf: " << isinf(x) << end
    // -> inf is inf: 1
    cout << x + 1 << endl;  // -> inf
}
```

Überlauf vermeiden

- abhängig von der jeweiligen Aufgabenstellung!
- Mittelwert zweier Zahlen?
  - ganze Zahlen
    - VZ-lose vs VZ-behanftete Zahlen
  - Gleitkommazahlen
- Interpolation

Mittelwert zweier ganzen Zahlen?

…Mittelpunkt zwischen zwei Zahlen am Zahlenstrahl

Mittelwert zweier ganzen Zahlen?

…Mittelpunkt zwischen zwei Zahlen am Zahlenstrahl

- Lösung

```
c = (a + b) / 2;
```

  - funktioniert für `int`, `unsigned` und Gleitkommazahlen

Mittelwert zweier ganzen Zahlen?

…Mittelpunkt zwischen zwei Zahlen am Zahlenstrahl

- Lösung

```
c = (a + b) / 2;
```

  - funktioniert für `int`, `unsigned` und Gleitkommazahlen
    - für ganze Zahlen: nicht unbedingt ohne Abschneiden der Nachkommastellen beim Ergebnis
  - aber: Überlauf kann auftreten!

Mittelwert zweier ganzen Zahlen?

…Mittelpunkt zwischen zwei Zahlen am Zahlenstrahl

- Lösung

```
c = (a + b) / 2;
```

  - funktioniert für `int`, `unsigned` und Gleitkommazahlen
    - für ganze Zahlen: nicht unbedingt ohne Abschneiden der Nachkommastellen beim Ergebnis
  - aber: Überlauf kann auftreten!
- Lösung für VZ-lose Zahlen, $b \geq a$

```
c = a + (b - a) / 2;
```

  - für VZ-behaftete Zahlen?

Mittelwert zweier ganzen Zahlen?

…Mittelpunkt zwischen zwei Zahlen am Zahlenstrahl

- Lösung

```
c = (a + b) / 2;
```

  - funktioniert für `int`, `unsigned` und Gleitkommazahlen
    - für ganze Zahlen: nicht unbedingt ohne Abschneiden der Nachkommastellen beim Ergebnis
  - aber: Überlauf kann auftreten!

- Lösung für VZ-lose Zahlen, $b \geq a$

```
c = a + (b - a) / 2;
```

  - für VZ-behaftete Zahlen?
  - Annahme: 4 Bit VZ-behaftet in 2er Komplement
    - Zahlenbereich: $[-8, 7]$
    - $a = 5, b = -7 \rightarrow b - a = -12$!!!

Mittelwert zweier ganzen Zahlen?

…Mittelpunkt zwischen zwei Zahlen am Zahlenstrahl

- Lösung

```
c = (a + b) / 2;
```

  - funktioniert für `int`, `unsigned` und Gleitkommazahlen
    - für ganze Zahlen: nicht unbedingt ohne Abschneiden der Nachkommastellen beim Ergebnis
  - aber: Überlauf kann auftreten!

- Lösung für VZ-lose Zahlen, $b \geq a$

```
c = a + (b - a) / 2;
```

  - für VZ-behaftete Zahlen?
  - Annahme: 4 Bit VZ-behaftet in 2er Komplement
    - Zahlenbereich: $[-8, 7]$
    - $a = 5, b = -7 \rightarrow b - a = -12$!!!
  - $\rightarrow$ funktioniert, wenn nicht (immer) wenn verschiedene VZ
    - auch wenn $b < a$

Mittelwert zweier ganzen Zahlen? –

```cpp
#include <iostream>
#include <limits>
#include <type_traits>

using namespace std;

// iff conversion from unsigned to signed preserves bit patter
template <typename Integer>  // iff int is two-complement
constexpr Integer midpoint(Integer a, Integer b) noexcept {
    using U = make_unsigned_t<Integer>;  // -> type_traits
    return a > b ? a - (U(a) - b) / 2 : a + (U(b) - a) / 2;
}

int main() {
   cout << numeric_limits<int>::min() << endl; // -2147483648
   int a{-2147483640};
   int b{10};
   cout << a + (b - a) / 2 << endl;  // 1073741833
   cout << midpoint(a, b) << endl;  // -1073741815
   a = 11;
   cout << a + (b - a) / 2 << endl;  // 11
   cout << midpoint(a, b) << endl;  // 11
}
```

Mittelwert zweier Gleitkommazahlen

- keine der obigen Lösungen funktioniert!
- Überlauf bzw. nicht korrekte Rundung in Subtraktion und Addition

Mittelwert zweier Gleitkommazahlen

- keine der obigen Lösungen funktioniert!
  - Überlauf bzw. nicht korrekte Rundung in Subtraktion und Addition
- Lösung für Gleitkommazahlen:

```
c = a / 2 + b / 2;
```

aber: in Spezialfällen → Rundungsfehler bei subnormalen Zahlen

- daher:

```
#include <iostream>
#include <cmath>
using namespace std;
int main() {
    double a{numeric_limits<double>::max()};
    cout << a << endl; // 1.79769e+308
    double b{a};
    double c{(a + b) / 2};
    cout << c << endl;  // inf
    a = numeric_limits<double>::denorm_min();
    b = a;
    c = a / 2 + b / 2;
    cout << c << endl;  // 0
    // -> Unterlauf
```

Interpolation

- i.A.: $a + t * (b - a) \neq b$ wenn $t = 1$

```cpp
#include <iostream>
#include <cmath>  // -> M_PI

using namespace std;

int main() {
    double b{0.1};
    double a{M_PI};
    double t{1};
    cout << M_PI << endl; // 3.14159
    cout << (a + t * (b - a)) << endl;  // 0.1
    cout << (a + t * (b - a) == b) << endl;  /
}
```

Interpolation – 2

- Überlauf, wenn a, b verschiedene Vorzeichen und größter Exponent
  - $\rightarrow b - a$!

Division ganzer Zahlen

```cpp
#include <iostream>
using namespace std;

int main() {
    cout << 4 / 2 << endl;  // -> 2
    cout << 5 / 2 << endl;  // -> 2
    cout << -3 / 2 << endl; // -> -1
    cout << 1 / 0 << endl;
    // -> ...terminated by signal SIGFPE
}
```

- $\rightarrow$ kann als Rundung zur Null interpretiert werden!
- $\rightarrow$ Division durch 0 $\rightarrow$ Programmabsturz
  - in diesem Fall: Warnung durch Compiler: `warning: division by zero [-Wdiv-by-zero]`
  - daher: Divisor auf 0 überprüfen!
- Achtung in Python wie in Mathematik!
  - daher: eigener Operator `//`

Rest ganzer Zahlen

Def.: Rest der Division ganzer Zahlen a (Dividend) und b (Divisor):

$$a = b \cdot q + r, \qquad 0 \leq r < |b|$$

- $a, b \in \mathbb{N}$ … eindeutig definiert
- $a, b \in \mathbb{Z}$ … nicht eindeutig
  - $5 \div -2 = -3R - 1$
    - Definition in der Mathematik: Rest hat VZ vom Divisor

Rest ganzer Zahlen

Def.: Rest der Division ganzer Zahlen a (Dividend) und b (Divisor):

$$a = b \cdot q + r, \qquad 0 \leq r < |b|$$

- $a, b \in \mathbb{N}$ … eindeutig definiert
- $a, b \in \mathbb{Z}$ … nicht eindeutig
  - $5 \div -2 = -3R - 1$
    - Definition in der Mathematik: Rest hat VZ vom Divisor
  - aber: $5 \div -2 = -2R1$!!

Rest ganzer Zahlen

```
print(5 // 2, 5 % 2)
print(-5 // 2, -5 % 2)
print(5 // -2, 5 % -2)
print(-5 // -2, -5 % -2)
```

Rest ganzer Zahlen

```
print(5 // 2, 5 % 2)
print(-5 // 2, -5 % 2)
print(5 // -2, 5 % -2)
print(-5 // -2, -5 % -2)
```

```
2 1
-3 1
-3 -1
2 -1
```

d.h. wie in der Mathematik!

Rest ganzer Zahlen – 2

```cpp
#include <iostream>
using namespace std;

int main() {
    cout << 5 / 2 << ' ' << 5 % 2 << endl;
    cout << -5 / 2 << ' ' <<  -5 % 2 << endl;
    cout << 5 / -2 << ' ' <<  5 % -2 << endl;
    cout << -5 / -2 << ' ' <<  -5 % -2 << endl;
}
```

Rest ganzer Zahlen – 2

```cpp
#include <iostream>
using namespace std;

int main() {
    cout << 5 / 2 << ' ' << 5 % 2 << endl;
    cout << -5 / 2 << ' ' <<  -5 % 2 << endl;
    cout << 5 / -2 << ' ' <<  5 % -2 << endl;
    cout << -5 / -2 << ' ' <<  -5 % -2 << endl;
}
```

```
2 1
-2 -1
-2 1
2 -1
```

laut Spezifikation: ”(a/b)*b + a%b is equal to a.”

Rest ganzer Zahlen – 3

```cpp
#include <iostream>
using namespace std;

int mod(int a, int b) { return ((a % b) + b) % b; }

int main() {
    cout << mod(5, 2) << endl;
    cout << mod(-5, 2) << endl;
    cout << mod(5, -2) << endl;
    cout << mod(-5, -2) << endl;
}
```

```
1
1
-1
-1
```

Rechnen mit Gleitkommazahlen

- Darstellung von Gleitkommazahlen
  - Abspeichern von Gleitkommazahlen
  - $\rightarrow$ Foliensatz ”Gleitkommazahlen”
- Fehler beim Rechnen
- Überlauf
- Unterlauf
- NaN

Darstellung GKZ – Python

$$0.1 + 0.1 + 0.1 = 0.3$$

Darstellung GKZ – Python

$0.1 + 0.1 + 0.1 = 0.3$ aber:

```
>>> 0.1 + 0.1 + 0.1 == 0.3
False
```

???

Darstellung GKZ – Python

$0.1 + 0.1 + 0.1 = 0.3$ aber:

```
>>> 0.1 + 0.1 + 0.1 == 0.3
False
```

???

```
>>> 0.1
0.1
>>> 0.1 + 0.1 + 0.1
```

Darstellung GKZ – Python

$0.1 + 0.1 + 0.1 = 0.3$ aber:

```
>>> 0.1 + 0.1 + 0.1 == 0.3
False
```

???

```
>>> 0.1
0.1
>>> 0.1 + 0.1 + 0.1
0.30000000000000004
```

0.1_{10} im Binärsystem

- Rechnet Python falsch?

0.1_{10} im Binärsystem

- Rechnet Python falsch? nein! $\rightarrow$ Darstellung von `0.1` ist **nicht** exakt möglich! D.h. $0.1 \notin M$!
  - 0.1 kann kein Element von M sein… warum?

0.1_{10} im Binärsystem

- Rechnet Python falsch? nein! $\rightarrow$ Darstellung von `0.1` ist **nicht** exakt möglich! D.h. $0.1 \notin M$!
  - 0.1 kann kein Element von M sein… warum?unabhängig davon wie Gleitkommazahlen abgespeichert werden…

0.1_{10} im Binärsystem

- Rechnet Python falsch? nein! $\rightarrow$ Darstellung von `0.1` ist **nicht** exakt möglich! D.h. $0.1 \notin M$!
  - 0.1 kann kein Element von M sein… warum?unabhängig davon wie Gleitkommazahlen abgespeichert werden…
- $0.1_{10} = 0.0001100110011001100110011... = 0.0\overline{0011}$

$$
\begin{array}{ll}
0.1 \cdot 2 & \text{-} \\
0.2 \cdot 2 & 0 \\
0.4 \cdot 2 & 0 \\
0.8 \cdot 2 & 0 \\
\not{1}.6 \cdot 2 & 1 \\
\not{1}.2 \cdot 2 & 1 \\
0.4 \cdot 2 & 0 \\
\ldots &
\end{array}
$$

Darstellung GKZ – C++

- Beispiel – C++
  - $0.1 + 0.1 + 0.1 = 0.3$ in C++?

```cpp
#include <iostream>
#include <cmath>

using namespace std;

int main() {
    cout << 0.1 << endl;
```

Darstellung GKZ – C++

- Beispiel – C++
  - $0.1 + 0.1 + 0.1 = 0.3$ in C++?

```
#include <iostream>
#include <cmath>

using namespace std;

int main() {
    cout << 0.1 << endl;
```

$\rightarrow$ `0.1`

```
    cout << 0.1 + 0.1 + 0.1 << endl;
```

Darstellung GKZ – C++

- Beispiel – C++
  - $0.1 + 0.1 + 0.1 = 0.3$ in C++?

```
#include <iostream>
#include <cmath>

using namespace std;

int main() {
    cout << 0.1 << endl;
```

$\rightarrow$ `0.1`

```
    cout << 0.1 + 0.1 + 0.1 << endl;
```

$\rightarrow$ `0.3`

Darstellung GKZ – C++

- Beispiel – C++
  - $0.1 + 0.1 + 0.1 = 0.3$ in C++?

```
#include <iostream>
#include <cmath>

using namespace std;

int main() {
    cout << 0.1 << endl;
```

$\rightarrow$ `0.1`

```
    cout << 0.1 + 0.1 + 0.1 << endl;
```

$\rightarrow$ `0.3`

rechnet C++ besser als Python?

```
    cout << (0.1 + 0.1 + 0.1 == 0.3) << endl;
```

Darstellung GKZ – C++

- Beispiel – C++
  - $0.1 + 0.1 + 0.1 = 0.3$ in C++?

```cpp
#include <iostream>
#include <cmath>

using namespace std;

int main() {
    cout << 0.1 << endl;
```

$\rightarrow$ `0.1`

```cpp
    cout << 0.1 + 0.1 + 0.1 << endl;
```

$\rightarrow$ `0.3`

rechnet C++ besser als Python?

```cpp
    cout << (0.1 + 0.1 + 0.1 == 0.3) << endl;
```

$\rightarrow$ `0`

Darstellung GKZ – C++

- Beispiel – C++
  - $0.1 + 0.1 + 0.1 = 0.3$ in C++?

```
#include <iostream>
#include <cmath>

using namespace std;

int main() {
    cout << 0.1 << endl;
```

$\rightarrow$ `0.1`

```
    cout << 0.1 + 0.1 + 0.1 << endl;
```

$\rightarrow$ `0.3`

rechnet C++ besser als Python?

```
    cout << (0.1 + 0.1 + 0.1 == 0.3) << endl;
```

$\rightarrow$ `0`

Was ist der Unterschied?

Darstellung GKZ – C++

- Beispiel – C++
  - $0.1 + 0.1 + 0.1 = 0.3$ in C++?

```
#include <iostream>
#include <cmath>

using namespace std;

int main() {
    cout << 0.1 << endl;
```

$\rightarrow$ `0.1`

```
    cout << 0.1 + 0.1 + 0.1 << endl;
```

$\rightarrow$ `0.3`

rechnet C++ besser als Python?

```
    cout << (0.1 + 0.1 + 0.1 == 0.3) << endl;
```

$\rightarrow$ `0`

Was ist der Unterschied? die Ausgabe!!!

Fehler beim Rechnen – Python

$f(n) = (1 + \frac{1}{n})^n$ $\lim_{n\to\infty} f(n) = e$ $e = 2.718281828459$

- $n = 10^3, f(n) = 2.7169239322355936$

Fehler beim Rechnen – Python

$f(n) = (1 + \frac{1}{n})^n$ $\lim_{n\to\infty} f(n) = e$ $e = 2.718281828459$

- $n = 10^3, f(n) = 2.7169239322355936$
- $n = 10^6, f(n) = 2.7182804690957534$

Fehler beim Rechnen – Python

$f(n) = (1 + \frac{1}{n})^n$ $\lim_{n\to\infty} f(n) = e$ $e = 2.718281828459$

- $n = 10^3, f(n) = 2.7169239322355936$
- $n = 10^6, f(n) = 2.7182804690957534$
- $n = 10^9, f(n) = 2.7182820520115603$

Fehler beim Rechnen – Python

$f(n) = (1 + \frac{1}{n})^n$ $\lim_{n\to\infty} f(n) = e$ $e = 2.718281828459$

- $n = 10^3, f(n) = 2.7169239322355936$
- $n = 10^6, f(n) = 2.7182804690957534$
- $n = 10^9, f(n) = 2.7182820520115603$
- $n = 10^{12}, f(n) = 2.7185234960372378$

Fehler beim Rechnen – Python

$f(n) = (1 + \frac{1}{n})^n$ $\lim_{n\to\infty} f(n) = e$ $e = 2.718281828459$

- $n = 10^3, f(n) = 2.7169239322355936$
- $n = 10^6, f(n) = 2.7182804690957534$
- $n = 10^9, f(n) = 2.7182820520115603$
- $n = 10^{12}, f(n) = 2.7185234960372378$
- $n = 10^{15}, f(n) = 3.035035206549262$

Fehler beim Rechnen – Python

$f(n) = (1 + \frac{1}{n})^n$ $\lim_{n\to\infty} f(n) = e$ $e = 2.718281828459$

- $n = 10^3, f(n) = 2.7169239322355936$
- $n = 10^6, f(n) = 2.7182804690957534$
- $n = 10^9, f(n) = 2.7182820520115603$
- $n = 10^{12}, f(n) = 2.7185234960372378$
- $n = 10^{15}, f(n) = 3.035035206549262$
- $n = 10^{18}, f(n) = 1.0$