1. Fliesskommazahlen

Die Typen `float` und `double`; Fliesskommazahlensysteme; Löcher im Wertebereich; IEEE Standard; Grenzen der Fliesskommaarithmetik; Fliesskomma-Richtlinien; Harmonische Zahlen

„Richtig Rechnen”

```
// Program: fahrenheit.cpp
// Convert temperatures from Celsius to Fahrenheit.

#include <iostream>

int main()
{
    // Input
    std::cout << "Temperature in degrees Celsius =? ";
    int celsius;
    std::cin >> celsius;

    // Computation and output
    std::cout << celsius << " degrees Celsius are "
              << 9 * celsius / 5 + 32 << " degrees Fahrenheit.\n";
    return 0;
}
```

28 degrees Celsius are 82 degrees Fahrenheit.

„Richtig Rechnen”

```
// Program: fahrenheit.cpp
// Convert temperatures from Celsius to Fahrenheit.

#include <iostream>

int main()
{
    // Input
    std::cout << "Temperature in degrees Celsius =? ";
    int celsius;
    std::cin >> celsius;

    // Computation and output
    std::cout << celsius << " degrees Celsius are "
              << 9 * celsius / 5 + 32 << " degrees Fahrenheit.\n";
    return 0;
}
```

richtig wäre 82.4

28 degrees Celsius are 82 degrees Fahrenheit.

„Richtig Rechnen”

```
// Program: fahrenheit.cpp
// Convert temperatures from Celsius to Fahrenheit.

#include <iostream>

int main()
{
    // Input
    std::cout << "Temperature in degrees Celsius =? ";
    float celsius; // Fliesskommazahlentyp
    std::cin >> celsius;

    // Computation and output
    std::cout << celsius << " degrees Celsius are "
              << 9 * celsius / 5 + 32 << " degrees Fahrenheit.\n";
    return 0;
}
```

28 degrees Celsius are 82.4 degrees Fahrenheit.

Repräsentation von Dezimalzahlen (z.B. 82.4)

Fixkommazahlen (z.B. mit 10 Stellen)

- feste Anzahl Vorkommastellen (z.B. 7)
- feste Anzahl Nachkommastellen (z.B. 3)

Repräsentation von Dezimalzahlen (z.B. 82.4)

Fixkommazahlen (z.B. mit 10 Stellen)

- feste Anzahl Vorkommastellen (z.B. 7)
- feste Anzahl Nachkommastellen (z.B. 3)

```
82.4 = 0000082.400
```

Repräsentation von Dezimalzahlen (z.B. 82.4)

Fixkommazahlen (z.B. mit 10 Stellen)

- feste Anzahl Vorkommastellen (z.B. 7)
- feste Anzahl Nachkommastellen (z.B. 3)

```
82.4 = 0000082.400
```

- Nachteile
  - Wertebereich wird *noch* kleiner als bei ganzen Zahlen.

Repräsentation von Dezimalzahlen (z.B. 82.4)

Fixkommazahlen (z.B. mit 10 Stellen)

- feste Anzahl Vorkommastellen (z.B. 7)
- feste Anzahl Nachkommastellen (z.B. 3)

```
0.0824 = 0000000.082
```
← dritte Stelle abgeschnitten

- Nachteile
  - Wertebereich wird *noch* kleiner als bei ganzen Zahlen.
  - Repräsentierbarkeit hängt von der Stelle des Kommas ab.

Repräsentation von Dezimalzahlen (z.B. 82.4)

Fliesskommazahlen (z.B. mit 10 Stellen)

- feste Anzahl signifikante Stellen (10)
- plus Position des Kommas

Repräsentation von Dezimalzahlen (z.B. 82.4)

Fliesskommazahlen (z.B. mit 10 Stellen)

- feste Anzahl signifikante Stellen (10)
- plus Position des Kommas

`82.4` = $824 \cdot 10^{-1}$

`0.0824` = $824 \cdot 10^{-4}$

- Zahl ist $\mathit{Signifikand} \times 10^{\mathit{Exponent}}$

Typen `float` und `double`

- sind die fundamentalen C++ Typen für Fliesskommazahlen
- approximieren den Körper der reellen Zahlen $(\mathbb{R}, +, \times)$ in der Mathematik
- haben grossen Wertebereich, ausreichend für viele Anwendungen (**`double`** hat mehr Stellen als **`float`**)
- sind auf vielen Rechnern sehr schnell

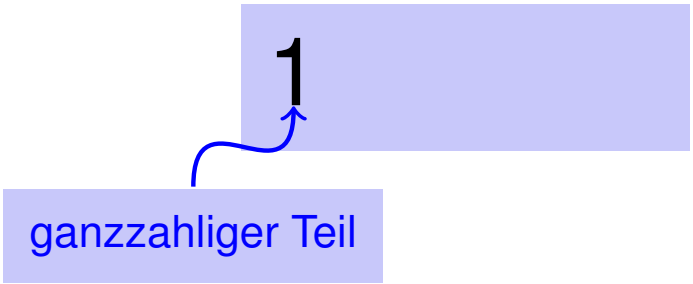
Arithmetische Operatoren

Wie bei `int`, aber ...

- Divisionsoperator `/` modelliert „echte” (reelle, nicht ganzzahlige) Division
- Keine Modulo-Operatoren `%` oder `%=`

Literale

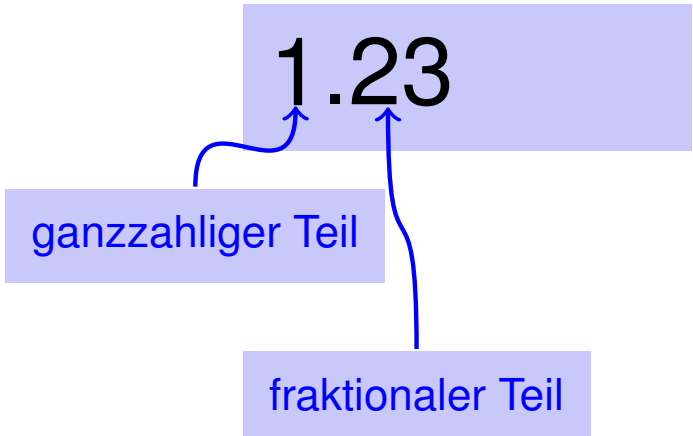
Unterscheiden sich von Ganzzahlliteralen



Literale

Unterscheiden sich von Ganzzahlliteralen durch Angabe von

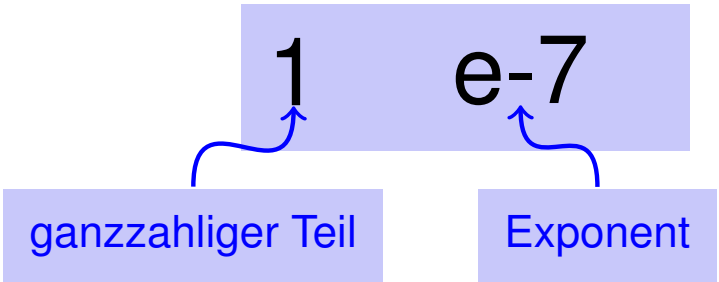
- Dezimalkomma



Literale

Unterscheiden sich von Ganzzahlliteralen durch Angabe von

- Dezimalkomma

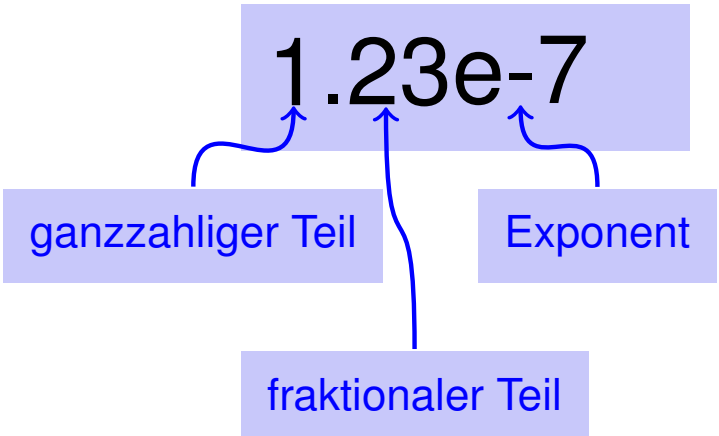


- oder Exponent.

Literale

Unterscheiden sich von Ganzzahlliteralen durch Angabe von

- Dezimalkomma

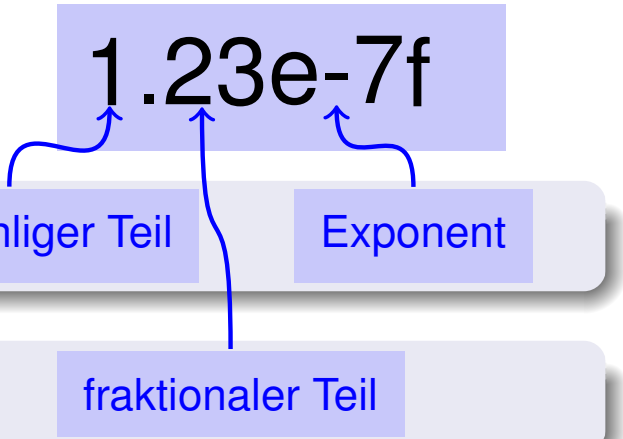


- und / oder Exponent.

Literale

Unterscheiden sich von Ganzzahlliteralen durch Angabe von

- Dezimalkomma



`1.0` : Typ **`double`**, Wert 1

`1.27f` : Typ **`float`**, Wert 1.27

- und / oder Exponent.

`1e3` : Typ **`double`**, Wert 1000

`1.23e-7` : Typ **`double`**, Wert $1.23 \cdot 10^{-7}$

`1.23e-7f` : Typ **`float`**, Wert $1.23 \cdot 10^{-7}$

Rechnen mit `float`: Beispiel

Approximation der Euler-Zahl

$$e = \sum_{i=0}^{\infty} \frac{1}{i!} \approx 2.71828\ldots$$

mittels der ersten 10 Terme.

Rechnen mit `float`: Beispiel

```
// Program: euler.cpp
// Approximate the Euler number e.

#include <iostream>

int main ()
{
    // values for term i, initialized for i = 0
    float t = 1.0f; // 1/i!
    float e = 1.0f; // i-th approximation of e

    std::cout << "Approximating the Euler number...\n";
    // steps 1,...,n
    for (unsigned int i = 1; i < 10; ++i) {
        e += t /= i; // compact form of t = t / i; e = e + t
        std::cout << "Value after term " << i << ": " << e << "\n";
    }
    return 0;
}
```

Rechnen mit `float`: Beispiel

```
// Program: euler.cpp
// Approximate the Euler number e.

#include <iostream>

int main ()
{
    // values for term i, initialized for i = 0
    float t = 1.0f; // 1/i!
    float e = 1.0f; // i-th approximation of e

    std::cout << "Approximating the Euler number...\n";
    // steps 1,...,n
    for (unsigned int i = 1; i < 10; ++i) {
        e += t /= i; // compact form of t = t / i; e = e + t
        std::cout << "Value after term " << i << ": " << e << "\n";
    }
    return 0;
}
```

Zuweisungen sind rechtsassoziativ: `(t/= i)` wird vor `e +=` ausgewertet.
D.h. erst Änderung von t : $\frac{1}{(i-1)!} \to \frac{1}{i!}$, dann $e \to e + \frac{1}{i!}$.

Rechnen mit `float`: Beispiel

Ausgabe:

```
Approximating the Euler number...
Value after term 1: 2
Value after term 2: 2.5
Value after term 3: 2.66667
Value after term 4: 2.70833
Value after term 5: 2.71667
Value after term 6: 2.71806
Value after term 7: 2.71825
Value after term 8: 2.71828
Value after term 9: 2.71828
```

Gemischte Ausdrücke, Konversion

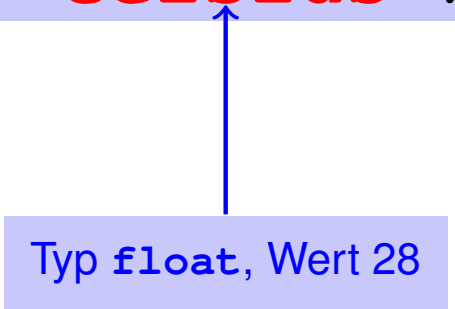
- Fliesskommazahlen sind allgemeiner als ganzzahlige Typen.
- In gemischten Ausdrücken werden ganze Zahlen zu Fliesskommazahlen konvertiert.

```
9 * celsius / 5  + 32
```

Gemischte Ausdrücke, Konversion

- Fliesskommazahlen sind allgemeiner als ganzzahlige Typen.
- In gemischten Ausdrücken werden ganze Zahlen zu Fliesskommazahlen konvertiert.

```
9 * celsius / 5   + 32
```



Gemischte Ausdrücke, Konversion

- Fliesskommazahlen sind allgemeiner als ganzzahlige Typen.
- In gemischten Ausdrücken werden ganze Zahlen zu Fliesskommazahlen konvertiert.

```
9 * 28.0f / 5  + 32
```

Gemischte Ausdrücke, Konversion

- Fliesskommazahlen sind allgemeiner als ganzzahlige Typen.
- In gemischten Ausdrücken werden ganze Zahlen zu Fliesskommazahlen konvertiert.

```
9 * 28.0f / 5  + 32
```

wird zu `float` konvertiert: 9.0f

Gemischte Ausdrücke, Konversion

- Fliesskommazahlen sind allgemeiner als ganzzahlige Typen.
- In gemischten Ausdrücken werden ganze Zahlen zu Fliesskommazahlen konvertiert.

```
252.0f / 5  + 32
```

wird zu **`float`** konvertiert: 5.0f

Gemischte Ausdrücke, Konversion

- Fliesskommazahlen sind allgemeiner als ganzzahlige Typen.
- In gemischten Ausdrücken werden ganze Zahlen zu Fliesskommazahlen konvertiert.

Gemischte Ausdrücke, Konversion

- Fliesskommazahlen sind allgemeiner als ganzzahlige Typen.
- In gemischten Ausdrücken werden ganze Zahlen zu Fliesskommazahlen konvertiert.

```
82.4
```

Wertebereich

Ganzzahlige Typen:

- Über- und Unterlauf häufig, aber ...
- Wertebereich ist zusammenhängend (keine „Löcher”): $\mathbb{Z}$ ist „diskret”.

Wertebereich

Ganzzahlige Typen:

- Über- und Unterlauf häufig, aber ...
- Wertebereich ist zusammenhängend (keine „Löcher”): $\mathbb{Z}$ ist „diskret”.

Fliesskommatypen:

- Über- und Unterlauf selten, aber ...
- es gibt Löcher: $\mathbb{R}$ ist „kontinuierlich”.

Löcher im Wertebereich

```
// Program: diff.cpp
// Check subtraction of two floating point numbers

#include <iostream>

int main()
{
    // Input
    float n1;
    std::cout << "First number     =? ";
    std::cin >> n1;
    float n2;
    std::cout << "Second number    =? ";
    std::cin >> n2;
    float d;
    std::cout << "Their difference =? ";
    std::cin >> d;

    // Computation and output
    std::cout << "Computed difference - input difference = "
    << n1 - n2 - d << ".\n";
    return 0;
}
```

Löcher im Wertebereich

```
// Program: diff.cpp
// Check subtraction of two floating point numbers

#include <iostream>

int main()
{
    // Input
    float n1;
    std::cout << "First number     =? ";
    std::cin >> n1;
    float n2;
    std::cout << "Second number    =? ";
    std::cin >> n2;
    float d;
    std::cout << "Their difference =? ";
    std::cin >> d;

    // Computation and output
    std::cout << "Computed difference - input difference = "
    << n1 - n2 - d << ".\n";
    return 0;
}
```

Eingabe 1.5

Eingabe 1.0

Eingabe 0.5

Löcher im Wertebereich

```
// Program: diff.cpp
// Check subtraction of two floating point numbers

#include <iostream>

int main()
{
    // Input
    float n1;
    std::cout << "First number     =? ";
    std::cin >> n1;
    float n2;
    std::cout << "Second number    =? ";
    std::cin >> n2;
    float d;
    std::cout << "Their difference =? ";
    std::cin >> d;

    // Computation and output
    std::cout << "Computed difference - input difference = "
    << n1 - n2 - d << ".\n";
    return 0;
}
```

Eingabe 1.5

Eingabe 1.0

Eingabe 0.5

Ausgabe 0

Löcher im Wertebereich

```
// Program: diff.cpp
// Check subtraction of two floating point numbers

#include <iostream>

int main()
{
    // Input
    float n1;
    std::cout << "First number     =? ";
    std::cin >> n1;
    float n2;
    std::cout << "Second number    =? ";
    std::cin >> n2;
    float d;
    std::cout << "Their difference =? ";
    std::cin >> d;

    // Computation and output
    std::cout << "Computed difference - input difference = "
    << n1 - n2 - d << ".\n";
    return 0;
}
```

Eingabe 1.1

Eingabe 1.0

Eingabe 0.1

Löcher im Wertebereich

```
// Program: diff.cpp
// Check subtraction of two floating point numbers

#include <iostream>

int main()
{
    // Input
    float n1;
    std::cout << "First number     =? ";
    std::cin >> n1;
    float n2;
    std::cout << "Second number    =? ";
    std::cin >> n2;
    float d;
    std::cout << "Their difference =? ";
    std::cin >> d;

    // Computation and output
    std::cout << "Computed difference - input difference = "
    << n1 - n2 - d << ".\n";
    return 0;
}
```

Eingabe 1.1

Eingabe 1.0

Eingabe 0.1

Ausgabe 2.23517e-8

Löcher im Wertebereich

```
// Program: diff.cpp
// Check subtraction of two floating point numbers

#include <iostream>

int main()
{
    // Input
    float n1;
    std::cout << "First number     =? ";
    std::cin >> n1;
    float n2;
    std::cout << "Second number    =? ";
    std::cin >> n2;
    float d;
    std::cout << "Their difference =? ";
    std::cin >> d;

    // Computation and output
    std::cout << "Computed difference - input difference = "
    << n1 - n2 - d << ".\n";
    return 0;
}
```

Eingabe 1.1

Eingabe 1.0

Eingabe 0.1

Ausgabe 2.23517e-8

Was ist hier los?

Fliesskommazahlensysteme

Ein Fliesskommazahlensystem ist durch vier natürliche Zahlen definiert:

- $\beta \geq 2$, die Basis,
- $p \geq 1$, die Präzision,
- $e_{\min}$, der kleinste Exponent,
- $e_{\max}$, der grösste Exponent.

Bezeichnung

$$F(\beta, p, e_{\min}, e_{\max})$$

Fliesskommazahlensysteme

$F(\beta, p, e_{\min}, e_{\max})$ enthält die Zahlen

$$\pm \sum_{i=0}^{p-1} d_i \beta^{-i} \cdot \beta^e,$$

$d_i \in \{0, \ldots, \beta - 1\}, \quad e \in \{e_{\min}, \ldots, e_{\max}\}.$

Fliesskommazahlensysteme

$F(\beta, p, e_{\min}, e_{\max})$ enthält die Zahlen

$$\pm \sum_{i=0}^{p-1} d_i \beta^{-i} \cdot \beta^e,$$

$d_i \in \{0, \ldots, \beta - 1\}, \quad e \in \{e_{\min}, \ldots, e_{\max}\}.$

In Basis-β-Darstellung:

$$\pm\, d_0.d_1 \ldots d_{p-1} \cdot \beta^e,$$

Fliesskommazahlensysteme

Beispiel

- $\beta = 10$

Darstellungen der Dezimalzahl 0.1

$$1.0 \cdot 10^{-1}, \quad 0.1 \cdot 10^{0}, \quad 0.01 \cdot 10^{1}, \quad \ldots$$

Normalisierte Darstellung

Normalisierte Zahl:

$$\pm\, d_0.d_1 \ldots d_{p-1} \cdot \beta^e, \qquad d_0 \neq 0$$

Bemerkung 1

Die normalisierte Darstellung ist eindeutig und deshalb zu bevorzugen.

Normalisierte Darstellung

Normalisierte Zahl:

$$\pm\, d_0.d_1 \ldots d_{p-1} \cdot \beta^e, \qquad d_0 \neq 0$$

Bemerkung 2

Die Zahl 0 (und alle Zahlen kleiner als $\beta^{e_{\min}}$) haben keine normalisierte Darstellung (werden wir später beheben)!

Normalisierte Darstellung

Normalisierte Zahl:

$$\pm\, d_0.d_1 \ldots d_{p-1} \cdot \beta^e, \qquad d_0 \neq 0$$

Die Menge der normalisierten Zahlen bezeichnen wir mit

$$F^*(\beta, p, e_{\min}, e_{\max})$$

Normalisierte Darstellung

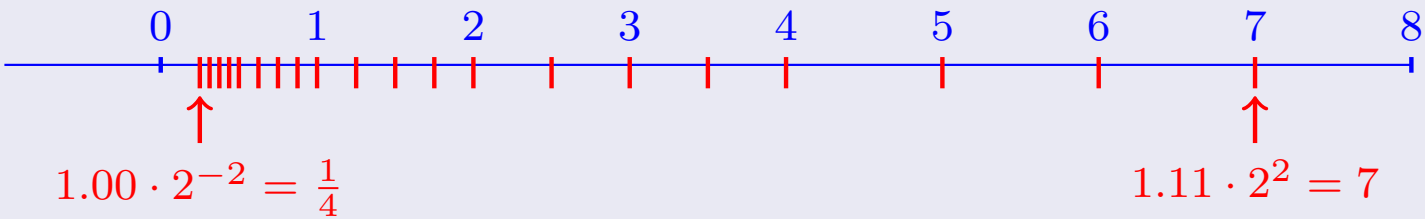
Normalisierte Zahl:

$$\pm\, d_0.d_1 \ldots d_{p-1} \cdot \beta^e, \qquad d_0 \neq 0$$

Beispiel $F^*(2, 3, -2, 2)$

Abgedeckter Wertebereich (nur positive Zahlen)

$d_0.d_1d_2$	$e=-2$	$e=-1$	$e=0$	$e=1$	$e=2$
1.00_2	0.25	0.5	1	2	4
1.01_2	0.3125	0.625	1.25	2.5	5
1.10_2	0.375	0.75	1.5	3	6
1.11_2	0.4375	0.875	1.75	3.5	7



Binäre und dezimale Systeme

- Intern rechnet der Computer meistens mit $\beta = 2$ (binäres Fliesskommazahlensystem)

Binäre und dezimale Systeme

- Intern rechnet der Computer meistens mit $\beta = 2$ (binäres Fliesskommazahlensystem)
- Literale und Eingaben haben $\beta = 10$ (dezimales Fliesskommazahlensystem)

Binäre und dezimale Systeme

- Intern rechnet der Computer meistens mit $\beta = 2$ (binäres Fliesskommazahlensystem)
- Literale und Eingaben haben $\beta = 10$ (dezimales Fliesskommazahlensystem)
- Eingaben müssen umgerechnet werden!

Umrechnung dezimal $\rightarrow$ binär

Angenommen, $0 < x < 2$.

Umrechnung dezimal $\rightarrow$ binär

Angenommen, $0 < x < 2$.

Binärexpansion:

$$x = \sum_{i=-\infty}^{0} b_i 2^i$$

Umrechnung dezimal $\rightarrow$ binär

Angenommen, $0 < x < 2$.

Binärexpansion:

$$x = \sum_{i=-\infty}^{0} b_i 2^i = b_0.b_{-1}b_{-2}b_{-3}\ldots$$

Umrechnung dezimal $\rightarrow$ binär

Angenommen, $0 < x < 2$.

Binärexpansion:

$$x = \sum_{i=-\infty}^{0} b_i 2^i = b_0.b_{-1}b_{-2}b_{-3}\ldots$$

$$= b_0 + \sum_{i=-\infty}^{-1} b_i 2^i$$

Umrechnung dezimal $\rightarrow$ binär

Angenommen, $0 < x < 2$.

Binärexpansion:

$$x = \sum_{i=-\infty}^{0} b_i 2^i = b_0.b_{-1}b_{-2}b_{-3}\ldots$$

$$= b_0 + \sum_{i=-\infty}^{-1} b_i 2^i = b_0 + \sum_{i=-\infty}^{0} b_{i-1} 2^{i-1}$$

Umrechnung dezimal $\rightarrow$ binär

Angenommen, $0 < x < 2$.

Binärexpansion:

$$x = \sum_{i=-\infty}^{0} b_i 2^i = b_0.b_{-1}b_{-2}b_{-3}\ldots$$

$$= b_0 + \sum_{i=-\infty}^{-1} b_i 2^i = b_0 + \sum_{i=-\infty}^{0} b_{i-1} 2^{i-1}$$

$$= b_0 + \underbrace{\left(\sum_{i=-\infty}^{0} b_{i-1} 2^i\right)}_{x'=b_{-1}.b_{-2}b_{-3}b_{-4}\ldots} /2 = b_0 + x'/2$$

Umrechnung dezimal $\rightarrow$ binär

Angenommen, $0 < x < 2$.

- Also: $x' = b_{-1}.b_{-2}b_{-3}b_{-4}\ldots = 2 \cdot (x - b_0)$

Umrechnung dezimal $\rightarrow$ binär

Angenommen, $0 < x < 2$.

- Also: $x' = b_{-1}.b_{-2}b_{-3}b_{-4}\ldots = 2 \cdot (x - b_0)$
- Schritt 1 (für x): Berechnen von b_0:

$$b_0 = \begin{cases} 1, & \text{falls } x \geq 1 \\ 0, & \text{sonst} \end{cases}$$

Umrechnung dezimal $\rightarrow$ binär

Angenommen, $0 < x < 2$.

- Also: $x' = b_{-1}.b_{-2}b_{-3}b_{-4}\ldots = 2 \cdot (x - b_0)$
- Schritt 1 (für x): Berechnen von b_0:

$$b_0 = \begin{cases} 1, & \text{falls } x \geq 1 \\ 0, & \text{sonst} \end{cases}$$

- Schritt 2 (für x): Berechnen von $b_{-1}, b_{-2}, \ldots$:

Umrechnung dezimal $\rightarrow$ binär

Angenommen, $0 < x < 2$.

- Also: $x' = b_{-1}.b_{-2}b_{-3}b_{-4}\ldots = 2 \cdot (x - b_0)$
- Schritt 1 (für x): Berechnen von b_0:

$$b_0 = \begin{cases} 1, & \text{falls } x \geq 1 \\ 0, & \text{sonst} \end{cases}$$

- Schritt 2 (für x): Berechnen von $b_{-1}, b_{-2}, \ldots$:

  Gehe zu Schritt 1 (für $x' = 2 \cdot (x - b_0)$)

Binärdarstellung von 1.1

x	b_i	$x - b_i$	$x' = 2(x - b_i)$
1.1	$b_0 = 1$		

Binärdarstellung von 1.1

x	b_i	$x - b_i$	$x' = 2(x - b_i)$
1.1	$b_0 = 1$	0.1	0.2

Binärdarstellung von 1.1

x	b_i	$x - b_i$	$x' = 2(x - b_i)$
1.1	$b_0 = 1$	0.1	0.2
0.2	$b_{-1} = 0$		

Binärdarstellung von 1.1

x	b_i	$x - b_i$	$x' = 2(x - b_i)$
1.1	$b_0 = 1$	0.1	0.2
0.2	$b_{-1} = 0$	0.2	0.4

Binärdarstellung von 1.1

x	b_i	$x - b_i$	$x' = 2(x - b_i)$
1.1	$b_0 = 1$	0.1	0.2
0.2	$b_{-1} = 0$	0.2	0.4
0.4	$b_{-2} = 0$		

Binärdarstellung von 1.1

x	b_i	$x - b_i$	$x' = 2(x - b_i)$
1.1	$b_0 = 1$	0.1	0.2
0.2	$b_{-1} = 0$	0.2	0.4
0.4	$b_{-2} = 0$	0.4	0.8

Binärdarstellung von 1.1

x	b_i	$x - b_i$	$x' = 2(x - b_i)$
1.1	$b_0 = 1$	0.1	0.2
0.2	$b_{-1} = 0$	0.2	0.4
0.4	$b_{-2} = 0$	0.4	0.8
0.8	$b_{-3} = 0$		

Binärdarstellung von 1.1

x	b_i	$x - b_i$	$x' = 2(x - b_i)$
1.1	$b_0 = 1$	0.1	0.2
0.2	$b_{-1} = 0$	0.2	0.4
0.4	$b_{-2} = 0$	0.4	0.8
0.8	$b_{-3} = 0$	0.8	1.6

Binärdarstellung von 1.1

x	b_i	$x - b_i$	$x' = 2(x - b_i)$
1.1	$b_0 = 1$	0.1	0.2
0.2	$b_{-1} = 0$	0.2	0.4
0.4	$b_{-2} = 0$	0.4	0.8
0.8	$b_{-3} = 0$	0.8	1.6
1.6	$b_{-4} = 1$		

Binärdarstellung von 1.1

x	b_i	$x - b_i$	$x' = 2(x - b_i)$
1.1	$b_0 = 1$	0.1	0.2
0.2	$b_{-1} = 0$	0.2	0.4
0.4	$b_{-2} = 0$	0.4	0.8
0.8	$b_{-3} = 0$	0.8	1.6
1.6	$b_{-4} = 1$	0.6	1.2

Binärdarstellung von 1.1

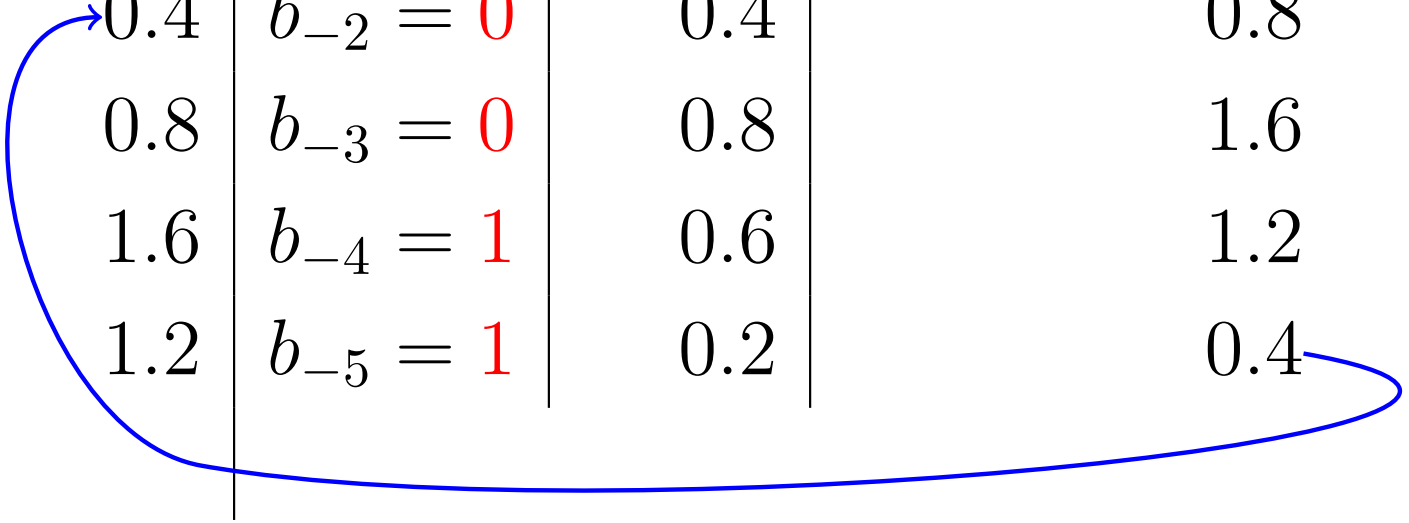
x	b_i	$x - b_i$	$x' = 2(x - b_i)$
1.1	$b_0 = 1$	0.1	0.2
0.2	$b_{-1} = 0$	0.2	0.4
0.4	$b_{-2} = 0$	0.4	0.8
0.8	$b_{-3} = 0$	0.8	1.6
1.6	$b_{-4} = 1$	0.6	1.2
1.2	$b_{-5} = 1$		

Binärdarstellung von 1.1

x	b_i	$x - b_i$	$x' = 2(x - b_i)$
1.1	$b_0 = 1$	0.1	0.2
0.2	$b_{-1} = 0$	0.2	0.4
0.4	$b_{-2} = 0$	0.4	0.8
0.8	$b_{-3} = 0$	0.8	1.6
1.6	$b_{-4} = 1$	0.6	1.2
1.2	$b_{-5} = 1$	0.2	0.4

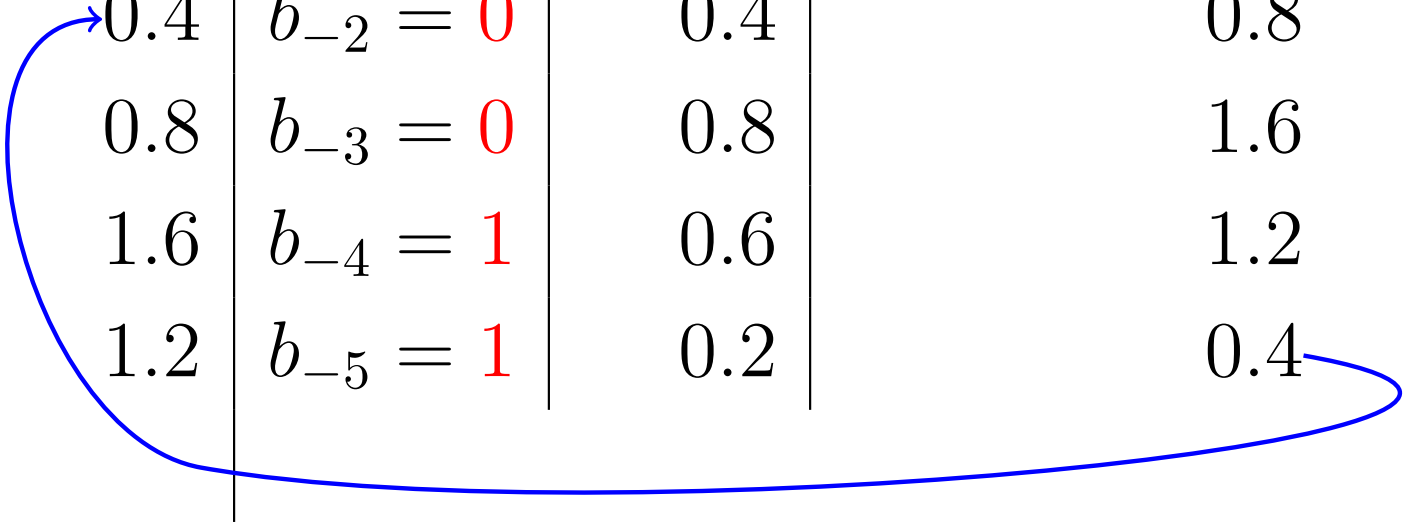
Binärdarstellung von 1.1

x	b_i	$x - b_i$	$x' = 2(x - b_i)$
1.1	$b_0 = 1$	0.1	0.2
0.2	$b_{-1} = 0$	0.2	0.4
0.4	$b_{-2} = 0$	0.4	0.8
0.8	$b_{-3} = 0$	0.8	1.6
1.6	$b_{-4} = 1$	0.6	1.2
1.2	$b_{-5} = 1$	0.2	0.4



Binärdarstellung von 1.1

x	b_i	$x - b_i$	$x' = 2(x - b_i)$
1.1	$b_0 = 1$	0.1	0.2
0.2	$b_{-1} = 0$	0.2	0.4
0.4	$b_{-2} = 0$	0.4	0.8
0.8	$b_{-3} = 0$	0.8	1.6
1.6	$b_{-4} = 1$	0.6	1.2
1.2	$b_{-5} = 1$	0.2	0.4



$\Rightarrow$ Binärdarstellung ist $1.0\overline{0011}$, periodisch, *nicht* endlich

Binärdarstellungen von 1.1 und 0.1

- sind nicht endlich, also gibt es Fehler bei der Konversion in ein (endliches) binäres Fliesskommazahlensystem.

Binärdarstellungen von 1.1 und 0.1

- sind nicht endlich, also gibt es Fehler bei der Konversion in ein (endliches) binäres Fliesskommazahlensystem.
- `1.1f` und `0.1f` sind für den Computer nicht 1.1 und 0.1, sondern geringfügig fehlerhafte Approximationen dieser Zahlen.

Binärdarstellungen von 1.1 und 0.1

- sind nicht endlich, also gibt es Fehler bei der Konversion in ein (endliches) binäres Fliesskommazahlensystem.
- `1.1f` und `0.1f` sind für den Computer nicht 1.1 und 0.1, sondern geringfügig fehlerhafte Approximationen dieser Zahlen.
- In `diff.cpp`: $1.1 - 1.0 \neq 0.1$

Der Excel-2007-Bug

- Umrechnungsfehler sind klein, können aber grosse Auswirkungen haben!

Der Excel-2007-Bug

- Umrechnungsfehler sind klein, können aber grosse Auswirkungen haben!

Microsoft Excel 2007: $77.1 \cdot 850 = 100,000$
(anstatt korrekt $65,535$)

Der Excel-2007-Bug

- Umrechnungsfehler sind klein, können aber grosse Auswirkungen haben!

  Microsoft Excel 2007: $77.1 \cdot 850 = 100,000$
  (anstatt korrekt $65,535$)

- Microsoft: Resultat wird korrekt berechnet, “nur” falsch angezeigt.

Der Excel-2007-Bug

- Umrechnungsfehler sind klein, können aber grosse Auswirkungen haben!

  Microsoft Excel 2007: $77.1 \cdot 850 = 100,000$
  (anstatt korrekt $65,535$)

- Microsoft: Resultat wird korrekt berechnet, “nur” falsch angezeigt.
- Das stimmt nicht ganz: 77.1 hat keine endliche Binärexpansion, berechnet wird $\lambda \approx 65,535$.

Der Excel-2007-Bug

- Umrechnungsfehler sind klein, können aber grosse Auswirkungen haben!

  Microsoft Excel 2007: $77.1 \cdot 850 = 100,000$ (anstatt korrekt $65,535$)

- Microsoft: Resultat wird korrekt berechnet, “nur” falsch angezeigt.
- Das stimmt nicht ganz: 77.1 hat keine endliche Binärexpansion, berechnet wird $\lambda \approx 65,535$.
- λ ist laut Microsoft eine von nur zwölf Zahlen, für die die Umwandlung ins Dezimalsystem (Ausgabe!) in der ersten Version fehlerhaft war.

Rechnen mit Fliesskommazahlen

ist fast so einfach wie mit ganzen Zahlen

Beispiel ($\beta = 2$, $p = 4$):

$$\begin{array}{rl} & 1.111 \cdot 2^{-2} \\ + & 1.011 \cdot 2^{-1} \end{array}$$

Rechnen mit Fliesskommazahlen

ist fast so einfach wie mit ganzen Zahlen

Beispiel ($\beta = 2$, $p = 4$):

$$\begin{aligned} & 1.111 \cdot 2^{-2} \\ + \quad & 1.011 \cdot 2^{-1} \end{aligned}$$

Schritt 1: Exponenten anpassen durch Denormalisieren einer Zahl

Rechnen mit Fliesskommazahlen

ist fast so einfach wie mit ganzen Zahlen

Beispiel ($\beta = 2$, $p = 4$):

$$\begin{aligned} & 1.111 \cdot 2^{-2} \\ + \quad & 10.110 \cdot 2^{-2} \checkmark \end{aligned}$$

Schritt 1: Exponenten anpassen durch Denormalisieren einer Zahl

Rechnen mit Fliesskommazahlen

ist fast so einfach wie mit ganzen Zahlen

Beispiel ($\beta = 2$, $p = 4$):

$$\begin{array}{rr} & 1.111 \cdot 2^{-2} \\ + & 10.110 \cdot 2^{-2} \\ \hline \end{array}$$

Schritt 2: Binäre Addition der Signifikanden

Rechnen mit Fliesskommazahlen

ist fast so einfach wie mit ganzen Zahlen

Beispiel ($\beta = 2$, $p = 4$):

$$\begin{array}{rr} & 1.111 \cdot 2^{-2} \\ + & 10.110 \cdot 2^{-2} \\ \hline & = 100.101 \cdot 2^{-2} \checkmark \end{array}$$

Schritt 2: Binäre Addition der Signifikanden

Rechnen mit Fliesskommazahlen

ist fast so einfach wie mit ganzen Zahlen

Beispiel ($\beta = 2$, $p = 4$):

$$\begin{aligned} & 1.111 \cdot 2^{-2} \\ + \quad & 10.110 \cdot 2^{-2} \\ \hline & = 100.101 \cdot 2^{-2} \end{aligned}$$

Schritt 3: Renormalisierung

Rechnen mit Fliesskommazahlen

ist fast so einfach wie mit ganzen Zahlen

Beispiel ($\beta = 2$, $p = 4$):

$$\begin{array}{rr} & 1.111 \cdot 2^{-2} \\ + & 10.110 \cdot 2^{-2} \\ \hline & = 1.00101 \cdot 2^{0} \checkmark \end{array}$$

Schritt 3: Renormalisierung

Rechnen mit Fliesskommazahlen

ist fast so einfach wie mit ganzen Zahlen

Beispiel ($\beta = 2$, $p = 4$):

$$\begin{aligned} & 1.111 \cdot 2^{-2} \\ + \quad & 10.110 \cdot 2^{-2} \\ \hline & = 1.00101 \cdot 2^{0} \end{aligned}$$

Schritt 4: Runden auf p signifikante Stellen, falls nötig

Rechnen mit Fliesskommazahlen

ist fast so einfach wie mit ganzen Zahlen

Beispiel ($\beta = 2$, $p = 4$):

$$\begin{aligned} & 1.111 \cdot 2^{-2} \\ + \quad & 10.110 \cdot 2^{-2} \\ \hline & = 1.001 \cdot 2^{0} \checkmark \end{aligned}$$

Schritt 4: Runden auf p signifikante Stellen, falls nötig

Der IEEE Standard 754

- legt Fliesskommazahlensysteme und deren Rundungsverhalten fest
- wird von vielen Plattformen unterstützt.

Der IEEE Standard 754

- legt Fliesskommazahlensysteme und deren Rundungsverhalten fest
- wird von vielen Plattformen unterstützt.
- Single precision (`float`) Zahlen:

$$F^*(2, 24, -126, 127)$$

- Double precision (`double`) Zahlen:

$$F^*(2, 53, -1022, 1023)$$

Der IEEE Standard 754

- legt Fliesskommazahlensysteme und deren Rundungsverhalten fest
- wird von vielen Plattformen unterstützt.
- Single precision (`float`) Zahlen:

$F^*(2, 24, -126, 127)$ plus 0, ∞, ...

- Double precision (`double`) Zahlen:

$F^*(2, 53, -1022, 1023)$ plus 0, ∞, ...

Der IEEE Standard 754

- legt Fliesskommazahlensysteme und deren Rundungsverhalten fest
- wird von vielen Plattformen unterstützt.
- Single precision (`float`) Zahlen:

$F^*(2, 24, -126, 127)$ plus 0, ∞, ...

- Double precision (`double`) Zahlen:

$F^*(2, 53, -1022, 1023)$ plus 0, ∞, ...

- Alle arithmetischen Operationen runden das *exakte* Ergebnis auf die nächste darstellbare Zahl

Der IEEE Standard 754

Warum

$$F^*(2, 24, -126, 127)?$$

Der IEEE Standard 754

Warum

$$F^*(2, 24, -126, 127)?$$

- 1 Bit für das Vorzeichen
- 23 Bit für den Signifikanden (führendes Bit ist 1 und wird nicht gespeichert)
- 8 Bit für den Exponenten (256 mögliche Werte)

$\Rightarrow$ insgesamt 32 Bit.

Der IEEE Standard 754

Warum

$$F^*(2, 24, -126, 127)?$$

- 1 Bit für das Vorzeichen
- 23 Bit für den Signifikanden (führendes Bit ist 1 und wird nicht gespeichert)
- 8 Bit für den Exponenten (254 mögliche Exponenten, 2 Spezialwerte: 0, ∞,...)

Der IEEE Standard 754

Warum

$$F^*(2, 53, -1022, 1023)?$$

- 1 Bit für das Vorzeichen
- 52 Bit für den Signifikanden (führendes Bit ist 1 und wird nicht gespeichert)
- 11 Bit für den Exponenten (2046 mögliche Exponenten, 2 Spezialwerte: 0, ∞,...)

$\Rightarrow$ insgesamt 64 Bit.

Richtlinien für das Rechnen mit Fliesskommazahlen

Regel 1

Teste keine zwei Fliesskommazahlen auf Gleichheit, wenn mindestens eine das Ergebnis einer Rundungsoperation ist!

Richtlinien für das Rechnen mit Fliesskommazahlen

Regel 1

Teste keine zwei Fliesskommazahlen auf Gleichheit, wenn mindestens eine das Ergebnis einer Rundungsoperation ist!

```
for (float i = 0.1; i != 1.0; i += 0.1)
    std::cout << i << "\n";
```

Richtlinien für das Rechnen mit Fliesskommazahlen

Regel 1

Teste keine zwei Fliesskommazahlen auf Gleichheit, wenn mindestens eine das Ergebnis einer Rundungsoperation ist!

```
for (float i = 0.1; i != 1.0; i += 0.1)
    std::cout << i << "\n";
```

In der Praxis ist das eine Endlosschleife, weil i niemals exakt 1 ist!

Richtlinien für das Rechnen mit Fliesskommazahlen

Regel 2

Vermeide die Addition von Zahlen sehr unterschiedlicher Grösse!

Richtlinien für das Rechnen mit Fliesskommazahlen

Regel 2

Vermeide die Addition von Zahlen sehr unterschiedlicher Grösse!

Beispiel: ($\beta = 2$, $p = 4$)

$$\begin{aligned} & 1.000 \cdot 2^4 \\ +& 1.000 \cdot 2^0 \end{aligned}$$

Richtlinien für das Rechnen mit Fliesskommazahlen

Regel 2

Vermeide die Addition von Zahlen sehr unterschiedlicher Grösse!

Beispiel: ($\beta = 2$, $p = 4$)

$$\begin{aligned} & 1.000 \cdot 2^4 \\ +& 1.000 \cdot 2^0 \\ =& \ 1.0001 \cdot 2^4 \end{aligned}$$

Richtlinien für das Rechnen mit Fliesskommazahlen

Regel 2

Vermeide die Addition von Zahlen sehr unterschiedlicher Grösse!

Beispiel: ($\beta = 2$, $p = 4$)

$$
\begin{aligned}
& 1.000 \cdot 2^4 \\
+ & 1.000 \cdot 2^0 \\
= & \ 1.0001 \cdot 2^4 \\
\text{”} = \text{”} & 1.000 \cdot 2^4 \quad \text{(nach Rundung)}
\end{aligned}
$$

Richtlinien für das Rechnen mit Fliesskommazahlen

Regel 2

Vermeide die Addition von Zahlen sehr unterschiedlicher Grösse!

Beispiel: ($\beta = 2$, $p = 4$)

$$
\begin{aligned}
& 1.000 \cdot 2^4 \\
+ & 1.000 \cdot 2^0 \\
= & \ 1.0001 \cdot 2^4 \\
\text{”} = \text{”} & 1.000 \cdot 2^4 \quad \text{(nach Rundung)}
\end{aligned}
$$

Addition von 1 hat keinen Effekt!

Beispiel Regel 2: Harmonische Zahlen

- Die n-te Harmonische Zahl ist

$$H_n = \sum_{i=1}^{n} \frac{1}{i} \approx \ln n.$$

Beispiel Regel 2: Harmonische Zahlen

- Die n-te Harmonische Zahl ist

$$H_n = \sum_{i=1}^{n} \frac{1}{i} \approx \ln n.$$

- Diese Summe kann vorwärts oder rückwärts berechnet werden, was mathematisch gesehen natürlich äquivalent ist.

Beispiel Regel 2: Harmonische Zahlen

```
// Program: harmonic.cpp
// Compute the n-th harmonic number in two ways.

#include <iostream>

int main()
{
  // Input
  std::cout << "Compute H_n for n =? ";
  unsigned int n;
  std::cin >> n;

  // Forward sum
  float fs = 0;
  for (unsigned int i = 1; i <= n; ++i)
    fs += 1.0f / i;

  // Backward sum
  float bs = 0;
  for (unsigned int i = n; i >= 1; --i)
    bs += 1.0f / i;

  // Output
  std::cout << "Forward sum  = " << fs << "\n"
            << "Backward sum = " << bs << "\n";
  return 0;
}
```

Beispiel Regel 2: Harmonische Zahlen

Ergebnisse:

- ```
  Compute H_n for n =?  10000000
  Forward sum = 15.4037
  Backward sum = 16.686
  ```
  ```

Beispiel Regel 2: Harmonische Zahlen

Ergebnisse:

- ```
  Compute H_n for n =?  10000000
  Forward sum = 15.4037
  Backward sum = 16.686
  ```
- ```
  Compute H_n for n =?  100000000
  Forward sum = 15.4037
  Backward sum = 18.8079
  ```

Beispiel Regel 2: Harmonische Zahlen

Beobachtung:

- Die Vorwärtssumme wächst irgendwann nicht mehr und ist “richtig” falsch.

Beispiel Regel 2: Harmonische Zahlen

Beobachtung:

- Die Vorwärtssumme wächst irgendwann nicht mehr und ist “richtig” falsch.
- Die Rückwärtsssume ist eine gute Approximation von H_n

Beispiel Regel 2: Harmonische Zahlen

Beobachtung:

- Die Vorwärtssumme wächst irgendwann nicht mehr und ist “richtig” falsch.
- Die Rückwärtsssume ist eine gute Approximation von H_n

Erklärung:

- Bei $1 + 1/2 + 1/3 + \cdots$ sind späte Terme zu klein, um noch beizutragen.

Beispiel Regel 2: Harmonische Zahlen

Beobachtung:

- Die Vorwärtssumme wächst irgendwann nicht mehr und ist “richtig” falsch.
- Die Rückwärtsssume ist eine gute Approximation von H_n

Erklärung:

- Bei $1 + 1/2 + 1/3 + \cdots$ sind späte Terme zu klein, um noch beizutragen.
- Problematik wie bei $2^4 + 1$ “=” 2^4

Richtlinien für das Rechnen mit Fliesskommazahlen

Regel 3

Vermeide die Subtraktion von Zahlen sehr ähnlicher Grösse!

Auslöschungsproblematik, siehe Skript.

Literatur

David Goldberg: What Every Computer Scientist Should Know About Floating-Point Arithmetic (1991)

Randy Glasbergen, 1996