Technische Universität München
Institut für Informatik
Prof. Dr. Hans-Joachim Bungartz
Dipl.-Inf. Christoph Riesinger
Dipl.-Inf. Dipl.-Math. Jürgen Bräckle

WS 2012/2013

Numerisches Programmieren, Übungen

Musterlösung 1. Übungsblatt: Zahlendarstellung, Rundungsfehler

1) Umrechnung von Zahlen

i) Zahldarstellung in allgemeinem System: $\sum_{i=0}^{N} r_i \text{Basis}^i$ mit $r_i \in \{0, 1, \ldots, \text{Basis} - 1\}$

	dezimal	binär	trinär	hexadezimal
Basis	10	2	3	16
darzust. Zahl	19	10011	201	13
darzust. Zahl	47	101111	1202	2f
darzust. Zahl	511	111111111	200221	1ff

ii) Schriftliches Dividieren der Brüche im binären System analog zum dezimalen:

$-\frac{1}{7} = -1_2 : 111_2$ (binär) :

$$
\begin{array}{rccccccccc}
- & 1 & 0 & 0 & 0 & : & 1 & 1 & 1 & = -0.00\overline{100} \quad \text{bzw.} -0.\overline{001} \\
-) & & 1 & 1 & 1 & & & & & \\
\hline
& & & & 1 & 0 & 0 & 0 & &
\end{array}
$$

$\frac{1}{10} = 1_2 : 1010_2$ (binär) :

$$
\begin{array}{rcccccccccc}
& 1 & 0 & 0 & 0 & 0 & : & 1 & 0 & 1 & 0 = 0.000\overline{1100} \quad \text{bzw. } 0.0\overline{0011} \\
-) & & 1 & 0 & 1 & 0 & & & & & \\
\hline
& & & 1 & 1 & 0 & 0 & & & & \\
-) & & & 1 & 0 & 1 & 0 & & & & \\
\hline
& & & & & 1 & 0 & 0 & 0 & 0 &
\end{array}
$$

2) Binärdarstellung von ganzen Zahlen

Beachte: n Bytes $= 8n$ Bits!

Wert 1. Bit	-2^{8n-1}
Wert 2. Bit	2^{8n-2}
kleinste Zahl x_{min}	-2^{8n-1}
größte Zahl x_{max}	$2^{8n-1} - 1$

Byte	Bits	x_{min}	x_{max}
1	8	-2^7	$2^7 - 1$
2	16	-2^{15}	$2^{15} - 1$
3	24	-2^{23}	$2^{23} - 1$
4	32	-2^{31}	$2^{31} - 1$
8	64	-2^{63}	$2^{63} - 1$

3) Gleitkomma-Zahlen

i) Schriftliches Dividieren (wie in Aufg. 1): $-\frac{11}{10} = -1011_2 : 1010_2$ (binär) :

$$
\begin{array}{cccccccccccccl}
 & 1 & 0 & 1 & 1 & : & 1 & 0 & 1 & 0 & & & & = \quad -1.000\overline{1100} \quad \text{bzw. } -1.0\overline{0011} \\
\text{-)} & 1 & 0 & 1 & 0 & & & & & & & & & \\
\hline
 & & & & 1 & 0 & 0 & 0 & 0 & & & & & \\
\text{-)} & & & & & 1 & 0 & 1 & 0 & & & & & \\
\hline
 & & & & & & 1 & 1 & 0 & 0 & & & & \\
\text{-)} & & & & & & 1 & 0 & 1 & 0 & & & & \\
\hline
 & & & & & & & & 1 & 0 & 0 & 0 & 0 &
\end{array}
$$

Damit erhält man insgesamt: $-1.0\overline{0011} \cdot 2^0$

Alternative:

$\frac{11}{10}$	$\cdot$	1	$=$	$\frac{11}{10}$	≥ 1	1
						.
$\frac{1}{10}$	$\cdot$	2	$=$	$\frac{2}{10}$	< 1	0
$\frac{2}{10}$	$\cdot$	2	$=$	$\frac{4}{10}$	< 1	0
$\frac{4}{10}$	$\cdot$	2	$=$	$\frac{8}{10}$	< 1	0
$\frac{8}{10}$	$\cdot$	2	$=$	$\frac{16}{10}$	≥ 1	1
$\frac{6}{10}$	$\cdot$	2	$=$	$\frac{12}{10}$	≥ 1	1
$\frac{2}{10}$	$\cdot$	2	$=$	$\frac{4}{10}$	< 1	0
$\frac{4}{10}$	$\cdot$	2	$=$	$\frac{8}{10}$	< 1	0
$\frac{8}{10}$	$\cdot$	2	$=$	$\frac{16}{10}$	≥ 1	1
$\frac{6}{10}$	$\cdot$	2	$=$	$\frac{12}{10}$	≥ 1	1
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

ii) Rundungsregel (korrektes Runden) für eine Zahl $x = (-1)^{\nu} \cdot 2^e \cdot 1.x_1 x_2 \ldots x_{t-1} | x_t x_{t+1} x_{t+2} \ldots$:

Nr.	Fallbed.	Rundungsvorschrift
Einfacher Fall: Abrunden		
1)	$x_t = 0$	$rd(x) = (-1)^\nu \cdot 2^e \cdot 1.x_1x_2\dots x_{t-1}$
Einfacher Fall: Aufrunden		
2)	$x_t = 1 \wedge$ $\lvert x_tx_{t+1}x_{t+2}\dots \neq \lvert 1000000000\dots$	$rd(x) = (-1)^\nu \cdot 2^e \cdot (1.x_1x_2\dots x_{t-1} + 2^{-(t-1)})$
Runden zur näheren geraden Mantissen-Zahl:		
3)	$x_{t-1}\lvert x_tx_{t+1}\dots = 0\lvert 100000000\dots$	$rd(x) = (-1)^\nu \cdot 2^e \cdot 1.x_1x_2\dots x_{t-1}$
4)	$x_{t-1}\lvert x_tx_{t+1}\dots = 1\lvert 100000000\dots$	$rd(x) = (-1)^\nu \cdot 2^e \cdot (1.x_1x_2\dots x_{t-1} + 2^{-(t-1)})$

Erläuterungen zu den verschiedenen Fällen:
Fall 1) und 2) stellen den normalen Fall dar, dass eine reelle Zahl nicht genau in der Mitte zwischen den beiden nächsten Maschinenzahlen liegt; es wird zum näheren Nachbarn gerundet. Fall 3) und 4) bewirken ein Runden mit Mantissenende $x_{t-1} = 0$ für reelle Zahlen, die genau zwischen zwei Maschinenzahlen liegen. In Fall 2) und 4) wird mit $+2^{-(t-1)}$ das letzte Bit um eins erhöht (und eventuell ein Übertrag durchgeführt).

Frage: Wie kann eine Floating Point Einheit auf der CPU unendlich viele Stellen zum Runden ausrechnen (z. B. bei periodischen Nachkommastellen)?
Das ist nicht nötig. Z. B. muss bei der Division lediglich bekannt sein, ob ein x_{t+1} ungleich 0 ist. Diese Information ist über den Rest erhältlich, der ab dem Berechnen von der Stelle x_{t+1} übrig bleibt.

Beispiel:
Es stehen zwei Mantissenbits zur Verfügung, also: $1,x_1x_{t-1}|x_tx_{t+1}x_{t-2}$...
Damit sind die folgenden Zahlen exakt darstellbar:

$$
\begin{aligned}
1,00_2 &= 1_{10} \\
1,01_2 &= 1,25_{10} \\
1,10_2 &= 1,5_{10} \\
1,11_2 &= 1,75_{10}
\end{aligned}
$$

Beispiele für alle vier Fälle aus der obigen Tabelle:

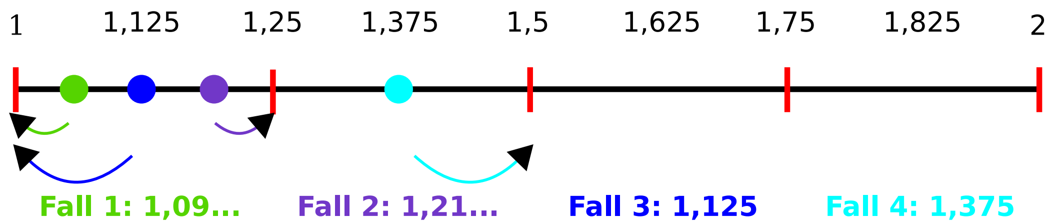


Abbildung 1: Veranschaulichung von Rundungen auf dem Zahlenstrahl

Nr.	Binärzahl	Dezimalzahl	ab-/aufrunden
1)	$1,\underline{00}\lvert 011_2$	$1,09375_{10}$	abrunden
2)	$1,00\lvert 111_2$	$1,21875_{10}$	aufrunden
3)	$1,0\underline{0}\lvert 100_2$	$1,125_{10}$	abrunden
4)	$1,0\underline{1}\lvert 100_2$	$1,375_{10}$	aufrunden

Abbildung 1 veranschaulicht die vier Fälle auf dem Zahlenstrahl.
Für unsere Zahl $-\frac{11}{10} = -1.0\overline{0011} \cdot 2^0$ aus Teilaufgabe i) bedeutet das:

Bit-Verwendungszweck	gesetztes Bit für $-1.0\overline{0011} \cdot 2^0$
Vorzeichen (1 Bit) ('-' wird zu '1')	1
Exponent (8 Bits) (0 + 127 = 127)	0 1 1 1 1 1 1 1
Mantisse (23 Bits)	0 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 1

Das letzte Bit der Mantisse ergibt sich nach der Rundungsregel (s.o.).

iii) Die Zahl $x = 1 + 2^{-30}$ ist nicht exakt darstellbar (zuwenig Stellen).
Die Zahl wird als rd(x) = 1 abgespeichert (vgl. Rundungsregeln).

Absoluter Fehler: $f_{abs} := |x - rd(x)| = 2^{-30}$

Relativer Fehler: $f_{rel} := |f_{abs}/x| < 2^{-30}$

iv) Definition: Maschinengenauigkeit = Die größte positive Zahl ε_{Ma}, so dass $1 \oplus \varepsilon_{Ma} = 1$.

Die kleinste darstellbare Zahl größer als 1 ist $1 + 2^{-23}$ (23 Bits echt für Mantissen-Nachkommastellen frei, da Normierung '1.' nicht extra gespeichert werden muss!). Folglich liegt ε in einer Größenordnung von 2^{-24}.

4) Fehlerabschätzung von Zeitschrittweiten

Floating Point Darstellung von 1/60:

Wir starten mit einfacher Division:

	1	0	0	0	0	0	0	:	1	1	1	1	0	0	=	$0.000001\overline{0001}$ bzw. $0.00\overline{0001}$
-)		1	1	1	1	0	0									
					1	0	0	0	0	0	0					
-)						1	1	1	1	0	0					
									1	0	0					

Für die Mantisse stehen 23 Bit zur Verfügung. Die erste '1' an der 6-ten Nachkommastelle wird allerdings nicht explizit abgespeichert. D. h. erst ab der 30-ten Nachkommastelle können die Ziffern nicht mehr abgespeichert werden:

	0,000001	0001	0001	0001	0001	0001	000[1
Nr. der Nachkommastelle:	7	11	15	19	23	27	

Mit der Normalisierung und dem nicht-Abspeichern der führenden '1' kann die Zahl deshalb bis auf die 29. Stelle genau abgespeichert werden.

Wie wird die Zahl dann als floating-point Zahl abgespeichert?
Sign: 0 für positive Zahlen
Exponent: -6+127 = 0b1111001
Mantisse: 0b00010001000100010001001 mit nach oben gerundeter letzten Stelle!

0	01111001	00010001000100010001001
Sign	Exponent	Mantissa

Floating point number: $(1) \cdot 2^{-6-23} \cdot 8947849 = 0.016666667535901069641113 2...$

Kleines Beispielprogramm für Zweifler:

```
#include <iostream>

int main()
{
        float l = (1.0/60.0);
        std::cout << l << std::endl;
        std::cout << (void*)*(unsigned long*)&l << std::endl;
}
/*
Output:
0.0166667
0x3c888889
*/
```

Wie kann man das Problem umgehen?

- Positionsänderung relativ zur Startposition berechnen.

- Als Zeitschrittweite eine 2er-Potenz wählen (z.B. 1/64).
- Rechengenauigkeit erhöhen.
- Fehler ignorieren, wenn die Simulationsdauer klein genug ist.

5) Ermittlung von π nach Archimedes

i) Quadrat:

$$
\begin{aligned}
s &= \sqrt{2} \\
U &= 4\sqrt{2} \approx 5.6568
\end{aligned}
$$

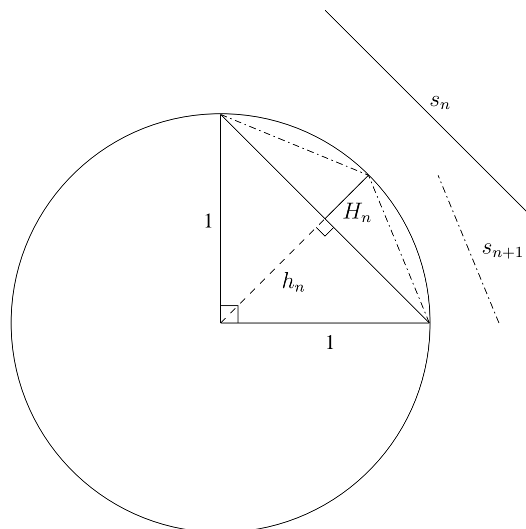


Abbildung 2: Definition der rekursiven Größen.

ii) Es gilt mit den Definitionen aus Abbildung 2 (+Satz von Pythagoras!):

$$
\begin{aligned}
h_n^2 + \left(\frac{s_n}{2}\right)^2 &= 1 \\
H_n &= 1 - h_n \\
H_n^2 + \left(\frac{s_n}{2}\right)^2 &= s_{n+1}^2
\end{aligned}
$$

und damit

$$
\begin{aligned}
s_{n+1} &= \sqrt{H_n^2 + \left(\frac{s_n}{2}\right)^2} = \sqrt{(1-h_n)^2 + \left(\frac{s_n}{2}\right)^2} \\
&= \sqrt{\left(1 - \sqrt{1 - \left(\frac{s_n}{2}\right)^2}\right)^2 + \left(\frac{s_n}{2}\right)^2} \\
&= \sqrt{\left(1 - 2\sqrt{1 - \left(\frac{s_n}{2}\right)^2} + 1 - \left(\frac{s_n}{2}\right)^2\right) + \left(\frac{s_n}{2}\right)^2} \\
&= \sqrt{2 - 2\sqrt{1 - \left(\frac{s_n}{2}\right)^2}} = \sqrt{2 - \sqrt{4 - s_n^2}}.
\end{aligned}
$$

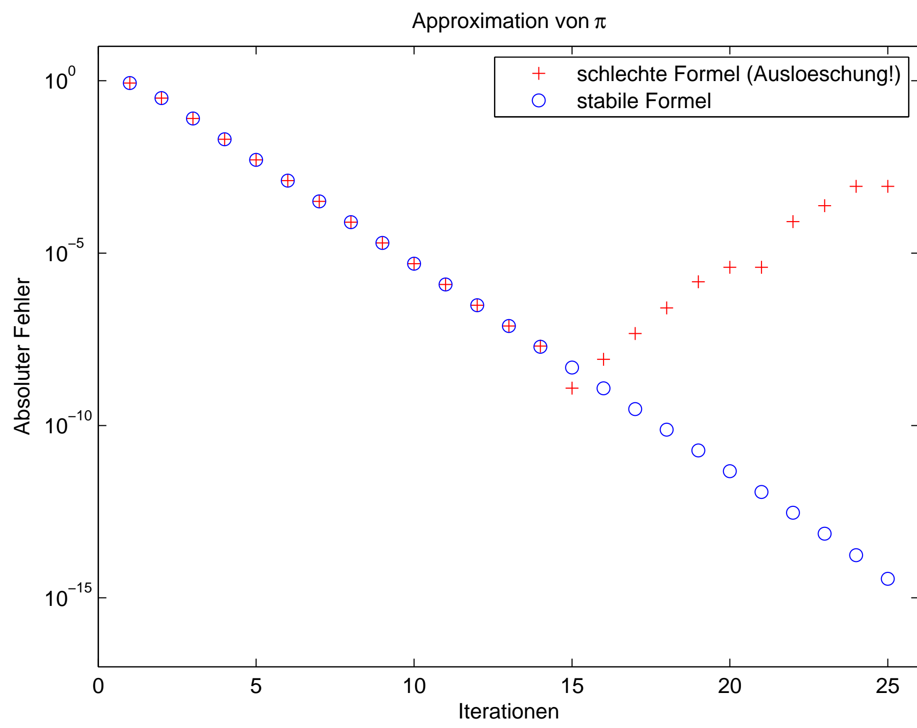
Der gesamte Umfang des 2^n-Ecks ergibt sich somit zu $U_n = 2^n \cdot s_n$ und damit also $\pi_n = U_n/2 = 2^{n-1} \cdot s_n$.

iii) Effekt wie in Tabelle auf Angabe: Vermeintlich höhere Genauigkeit durch mehr Rekursionen verbessert das Ergebnis nicht sondern zerstört den gesamten Wert!
Problem: Auslöschung!

Algebraische Umformung zur Vermeidung der Auslöschung:

$$
\begin{aligned}
s_{n+1} &= \sqrt{2 - \sqrt{4 - s_n^2}} = \sqrt{2 - \sqrt{4 - s_n^2}} \cdot \frac{\sqrt{2 + \sqrt{4 - s_n^2}}}{\sqrt{2 + \sqrt{4 - s_n^2}}} \\
&= \frac{|s_n|}{\sqrt{2 + \sqrt{4 - s_n^2}}}
\end{aligned}
$$

Vergleich der absoluten Fehler der beiden Formeln in Bezug auf echte Lösung π in semilogarithmischer Skala (erstellt mit matlab-Programm archimedes.m aus www):



Approximation von π
schlechte Formel (Ausloeschung!)
stabile Formel
Absoluter Fehler
Iterationen
10^0
10^−5
10^−10
10^−15
0
5
10
15
20
25