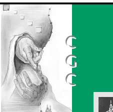
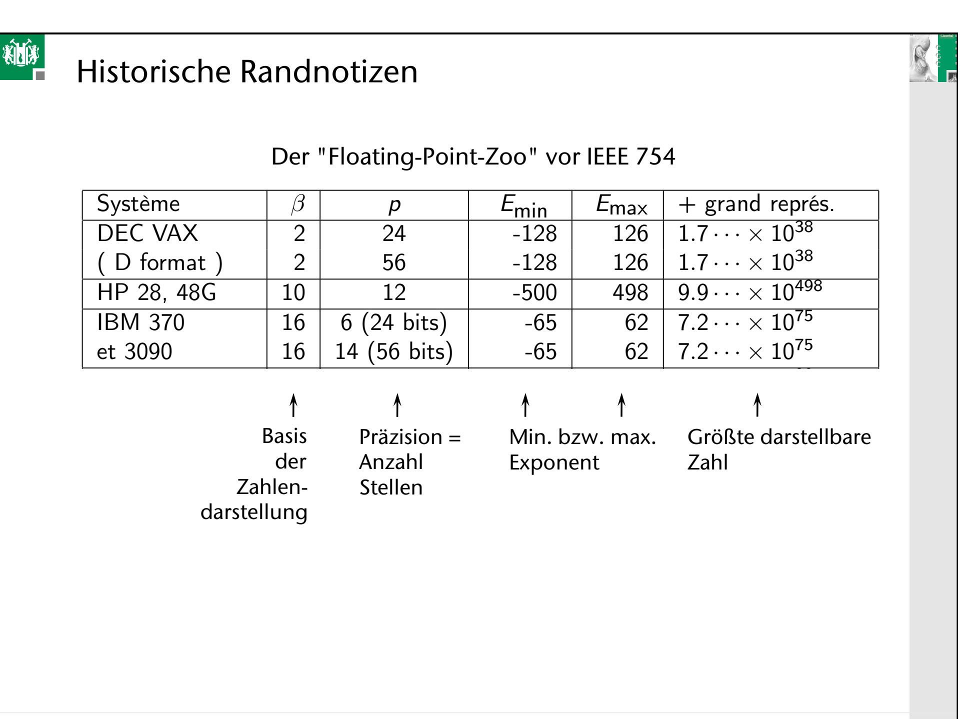


Informatik II
Numerische Robustheit

EPSILON is NOT 0.00001!

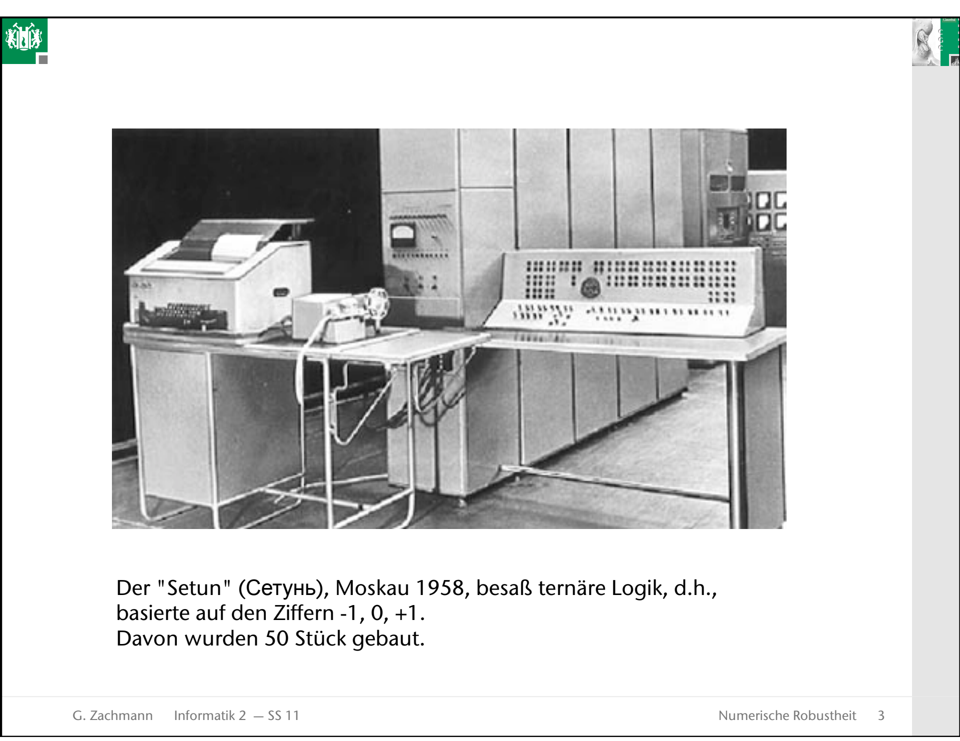
G. Zachmann
Clausthal University, Germany
cg.in.tu-clausthal.de

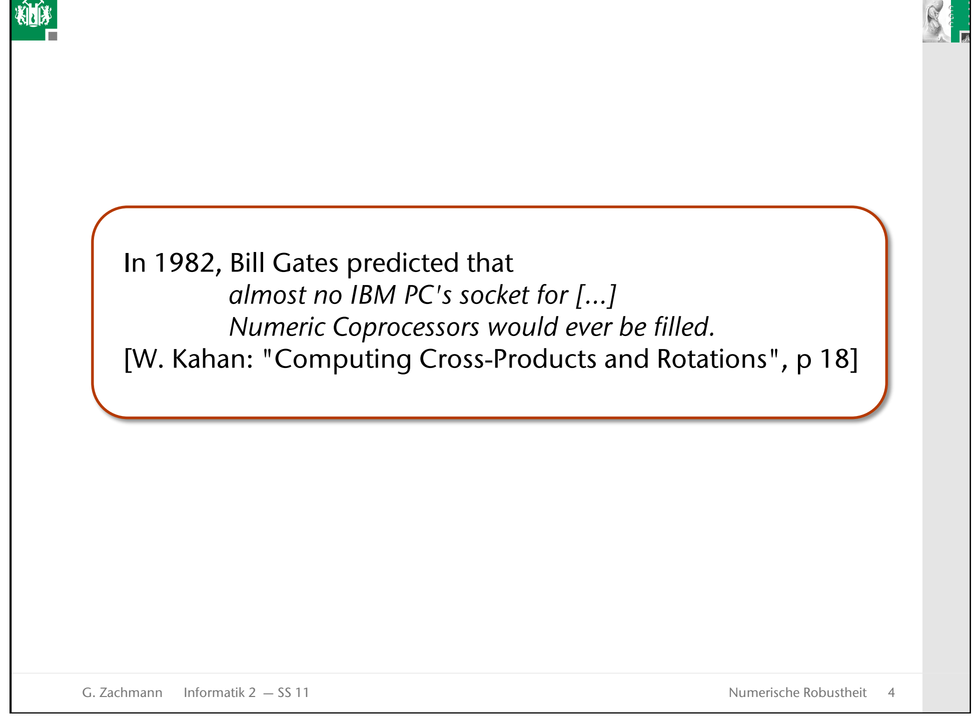


Historische Randnotizen

Der "Floating-Point-Zoo" vor IEEE 754

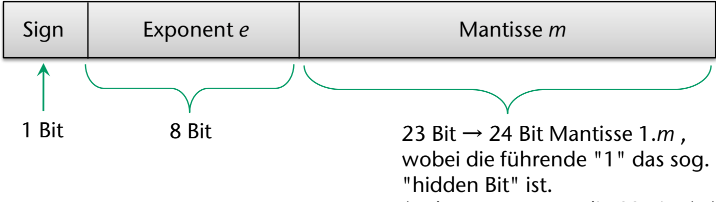
Système	β	p	E_{min}	E_{max}	+ grand représ.
DEC VAX	2	24	-128	126	$1.7\cdots\times 10^{38}$
(D format)	2	56	-128	126	$1.7\cdots\times 10^{38}$
HP 28, 48G	10	12	-500	498	$9.9\cdots\times 10^{498}$
IBM 370	16	6 (24 bits)	-65	62	$7.2\cdots\times 10^{75}$
et 3090	16	14 (56 bits)	-65	62	$7.2\cdots\times 10^{75}$


Der "Setun" (Сетунь), Moskau 1958, besaß ternäre Logik, d.h.,
basierte auf den Ziffern -1, 0, +1.
Davon wurden 50 Stück gebaut.
G. Zachmann Informatik 2 — SS 11
Numerische Robustheit 3


In 1982, Bill Gates predicted that
almost no IBM PC's socket for [...]
Numeric Coprocessors would ever be filled.
[W. Kahan: "Computing Cross-Products and Rotations", p 18]
G. Zachmann Informatik 2 — SS 11
Numerische Robustheit 4

Zur Erinnerung: das IEEE-754-Format

- **Der** Standard zur Darstellung von Floating-Point-Zahlen
- Das Format für single precision Floats:



23 Bit → 24 Bit Mantisse 1.*m* , wobei die führende "1" das sog. "hidden Bit" ist.
(Daher nennt man die 23 Bits (*m*) oft auch Signifikand.)

- Der Wert ist definiert als:

$$v = (-1)^s \cdot (1.m) \cdot 2^{e-127}$$

  - Aber es gibt (natürlich) Ausnahmen!

Alle darstellbaren Werte in single precision (`float`)

Exponent	Mantissa	Sign	Value
1...254	any	0|1	$v = (-1)^s \cdot (1.m) \cdot 2^{e-127}$
0	≠0	0|1	$v = (-1)^s \cdot (0.m) \cdot 2^{-126}$
0	0	0	$v = 0$
0	0	1	$v = -0$
255	0	0	$v \equiv +\infty$
255	0	1	$v \equiv -\infty$
255	≠0	0|1	$v \equiv NaN$

- Beispiel: $0.1_{10} = 0.000110011001100110011001100110011..._2$
  $= 1.100110011001100110011001100110011..._2 \times 2^{-4}$
  $\approx 1.\underbrace{10011001100110011001101}_{23\ \text{Bit}}{}_2 \times 2^{-4}$
  $= 0.10000000149..._{10}$

- Wertebereich für Floats $\approx [10^{-38} .. 10^{38}]$
  - Bekommt man in C:

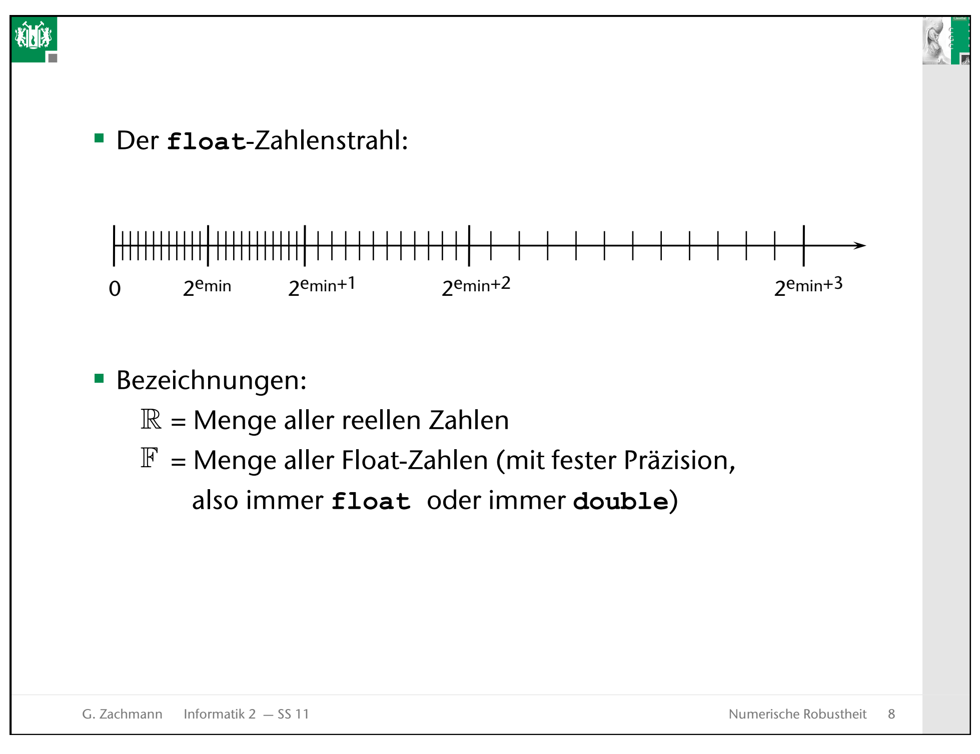
```
#include <float.h>
x = FLT_MAX;
```

  - Und in C++ mittels:

```
#include <numeric_limits>
x = numeric_limits<float>::min();
```

  - In Python:

```
import sys
sys.float_info.max
```

Der relative Fehler und ULPs

- Definition relativer Fehler (*relative error*):
  Seien x = das exakte Ergebnis einer Rechnung und
  $\hat{x}$ = das Ergebnis derselben Rechnung mittels Float-Zahlen.
  Dann ist der relative Fehler

$$\frac{|x - \hat{x}|}{|x|}$$

- Definition ULP:
  ULP = "unit in the last place"
  1 ULP = $\underbrace{0.00 \ldots 01}_{\text{Anzahl Stellen der gewählten Darstellung}}$
  Achtung: der Wert eines ULP hängt von der Anzahl Stellen und der Basis ab! (d.h., von der Zahlendarstellung!)

Exkurs: Zugang zur internen Repräsentation

- Wie erhält man das Bit-Muster einer Float-Zahl, die im Speicher steht, mittels Programm?
- Der Standard-Trick ist hier: den Speicherbereich mit zwei Variablen verschiedenen Typs überlagern!
- Im Programm (in C):

```
typedef union
{
    float f;
    unsigned char c[4];
} FloatAndBytes;
...
```

```
...
int main()
{
    FloatAndBytes x;
    x.f = 0.1;
    for (int i = 0; i < 4; i ++ )
        printf( "%02X ", x.c[i] );
    putchar('\n');
    return 0;
}
```

Exkurs: "Infinity Arithmetic"

- Manche propagieren das Rechnen mit **`Inf`** und **`NaN`**
- Beispiel:
  - Die quadratischen Gleichung: $ax^2 + bx + c = 0$
  - Lösungen:

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

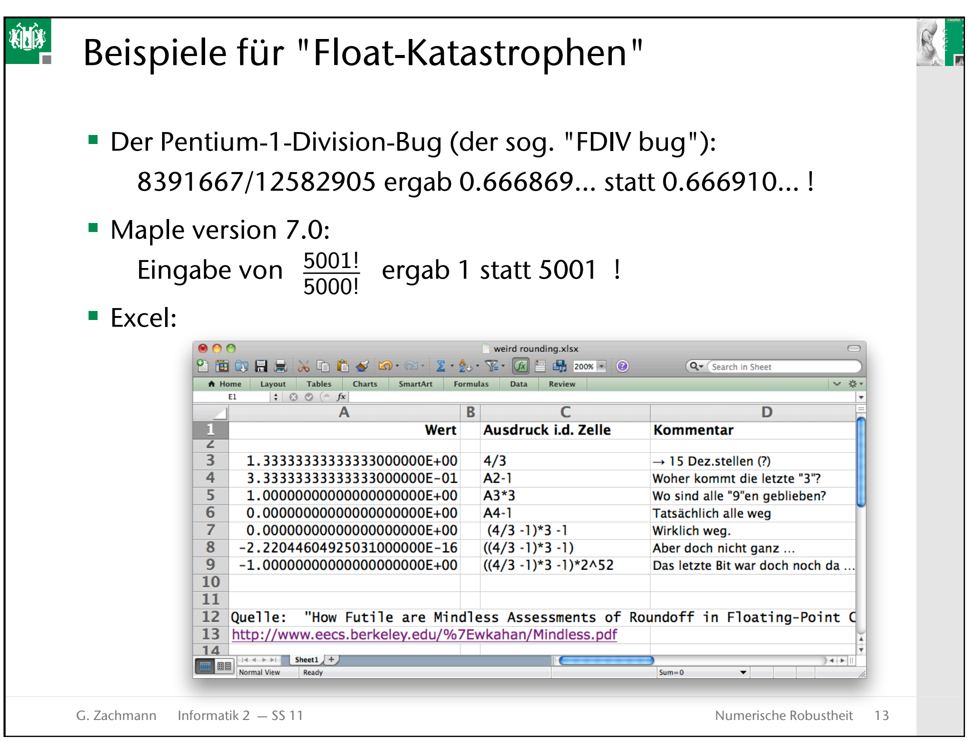
  - Als Programm:

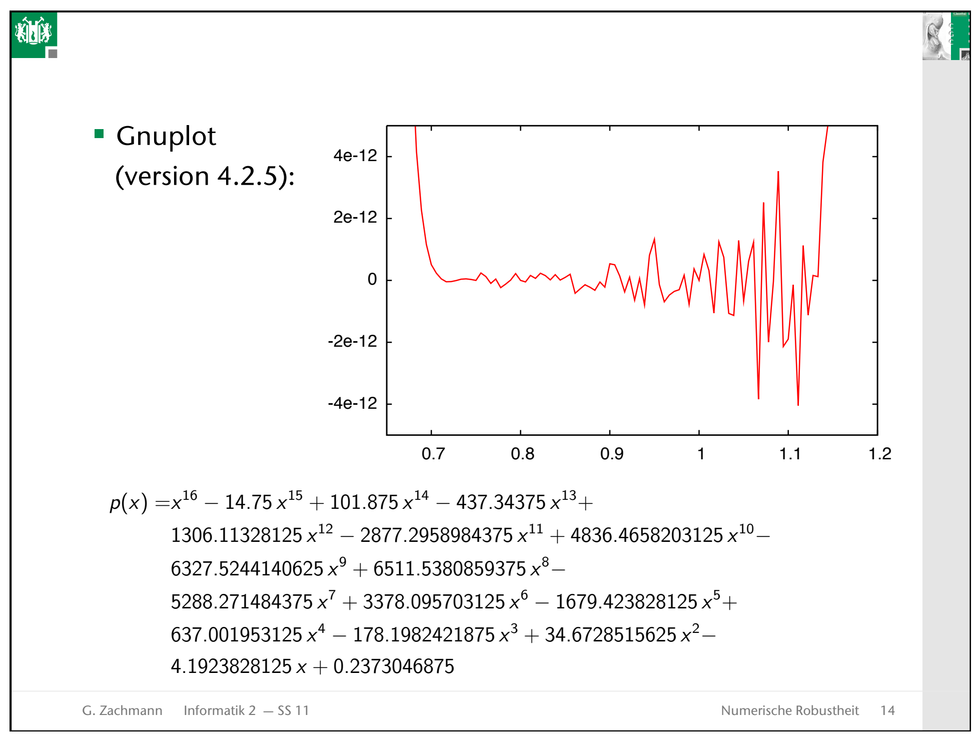
```
def solveQuadratic( a, b, c ):
    d = b*b - 4*a*c
    if d < 0.0:
        # Flag "no real solution"
        return (float('nan'), float('nan'))
    s = sqrt( d )
    x1 = (-b + s) / (2*a)
    x1 = (-b - s) / (2*a)
    return (x1, x2)
```

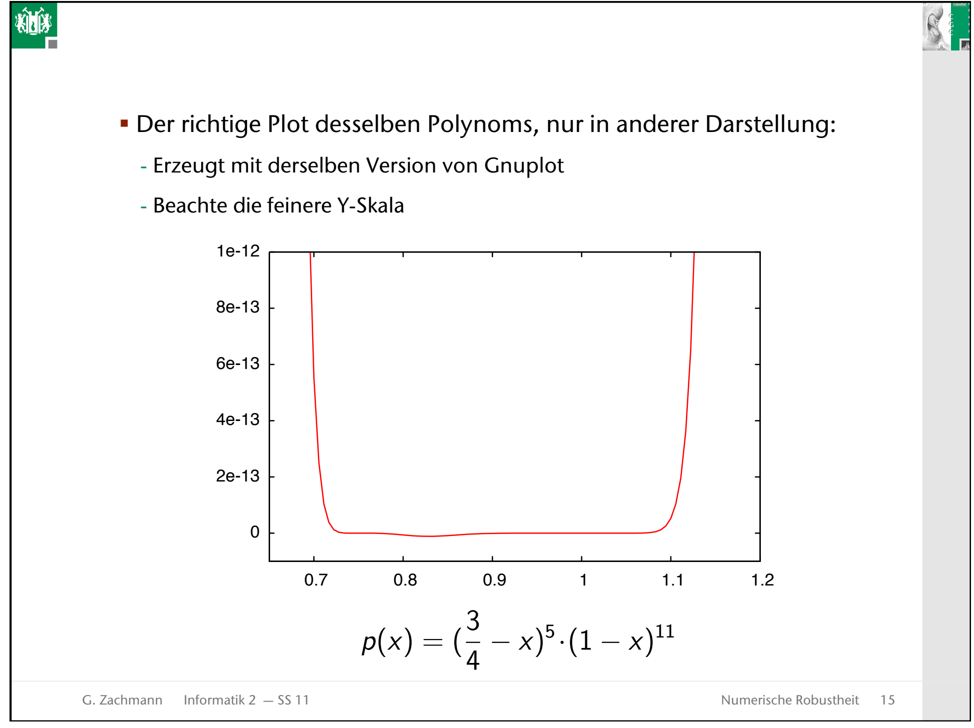
- Branches kosten sehr viel Zeit (die komplette CPU-Pipeline muß geleert werden ("*flush*"))
- Falls solveQuadratic() sehr oft aufgerufen wird (1 Mio mal), kann es sinnvoll sein, die NaN's von der FPU erzeugen zu lassen:

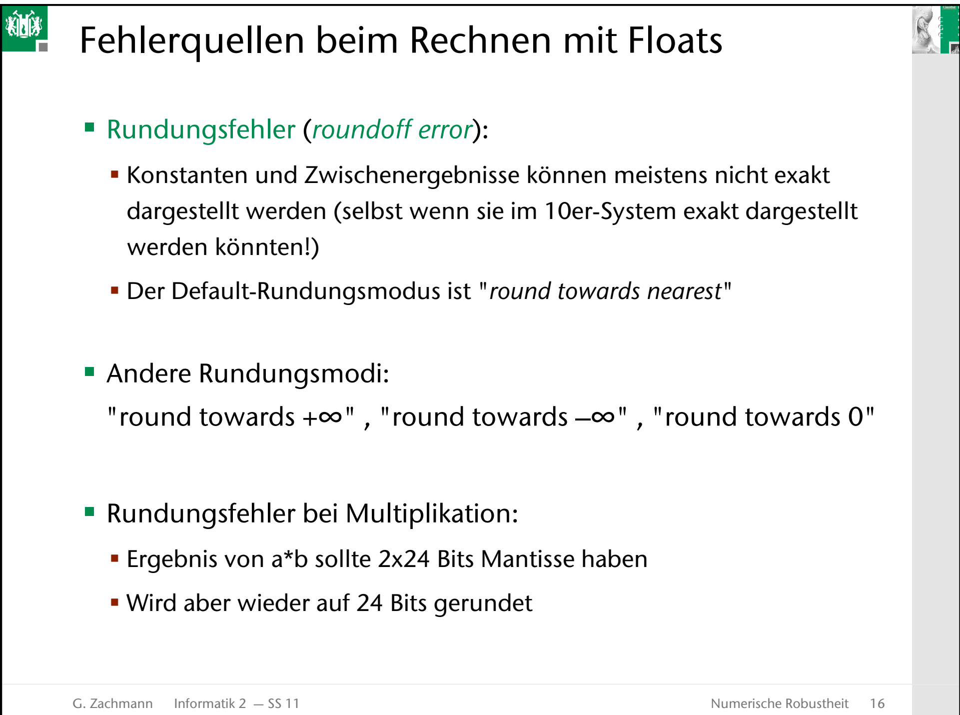
```
def solveQuadratic( a, b, c ):
    s = sqrt( b*b - 4*a*c ) # = evtl. NaN!
    x1 = (-b + s) / (2*a)
    x1 = (-b - s) / (2*a)
    return (x1, x2)
```

- Aber:
  - Früher oder später muß man *doch* auf NaN testen ...
  - Ich will in Ihrem Code sehen, daß Sie sich des Falls d<0 bewußt sind!
  - Mein Rat: verwenden Sie diese Art von "NaN/Inf-Arithmetik" erst, wenn der Profiler hier einen echten Hotspot zeigt!


Beispiele für "Float-Katastrophen"
Der Pentium-1-Division-Bug (der sog. "FDIV bug"):
8391667/12582905 ergab 0.666869... statt 0.666910... !
Maple version 7.0:
Eingabe von 5001!/5000! ergab 1 statt 5001 !
Excel:
weird rounding.xlsx
Wert
Ausdruck i.d. Zelle
Kommentar
1.33333333333333000000E+00 4/3 → 15 Dez.stellen (?)
3.33333333333330000000E-01 A2-1 Woher kommt die letzte "3"?
1.00000000000000000000E+00 A3*3 Wo sind alle "9"en geblieben?
0.00000000000000000000E+00 A4-1 Tatsächlich alle weg
0.00000000000000000000E+00 (4/3 -1)*3 -1 Wirklich weg.
-2.22044604925031000000E-16 ((4/3 -1)*3 -1) Aber doch nicht ganz ...
-1.00000000000000000000E+00 ((4/3 -1)*3 -1)*2^52 Das letzte Bit war doch noch da ...
Quelle: "How Futile are Mindless Assessments of Roundoff in Floating-Point C
http://www.eecs.berkeley.edu/%7Ewkahan/Mindless.pdf
G. Zachmann Informatik 2 — SS 11 Numerische Robustheit 13


Gnuplot (version 4.2.5):
4e-12
2e-12
0
-2e-12
-4e-12
0.7
0.8
0.9
1
1.1
1.2
$p(x) = x^{16} - 14.75\,x^{15} + 101.875\,x^{14} - 437.34375\,x^{13} + 1306.11328125\,x^{12} - 2877.2958984375\,x^{11} + 4836.4658203125\,x^{10} - 6327.5244140625\,x^{9} + 6511.5380859375\,x^{8} - 5288.271484375\,x^{7} + 3378.095703125\,x^{6} - 1679.423828125\,x^{5} + 637.001953125\,x^{4} - 178.1982421875\,x^{3} + 34.6728515625\,x^{2} - 4.1923828125\,x + 0.2373046875$
G. Zachmann Informatik 2 — SS 11 Numerische Robustheit 14


Der richtige Plot desselben Polynoms, nur in anderer Darstellung:
Erzeugt mit derselben Version von Gnuplot
Beachte die feinere Y-Skala
1e-12
8e-13
6e-13
4e-13
2e-13
0
0.7
0.8
0.9
1
1.1
1.2
$p(x) = (\frac{3}{4} - x)^5 \cdot (1 - x)^{11}$
G. Zachmann
Informatik 2 — SS 11
Numerische Robustheit
15


Fehlerquellen beim Rechnen mit Floats
Rundungsfehler (roundoff error):
Konstanten und Zwischenergebnisse können meistens nicht exakt dargestellt werden (selbst wenn sie im 10er-System exakt dargestellt werden könnten!)
Der Default-Rundungsmodus ist "round towards nearest"
Andere Rundungsmodi:
"round towards +∞" , "round towards –∞" , "round towards 0"
Rundungsfehler bei Multiplikation:
Ergebnis von a*b sollte 2x24 Bits Mantisse haben
Wird aber wieder auf 24 Bits gerundet
G. Zachmann
Informatik 2 — SS 11
Numerische Robustheit
16

- Entstehung des Rundungsfehlers bei Addition:
  - Algorithmus zur Addition zweier Floats a, b :

```
if exponent(a) = exponent(b):
    addiere Mantissen von a und b
    ....
else:
    sei oBdA a die kleinere Zahl
    shifte Mantisse von a nach rechts (≡ Shift des
       Binär-Kommas nach links ≡ Div durch 2)
       und erhöhe Exponent um 1
    bis Exponenten von a und b gleich sind
    weiter wie oben ...
```

  - Allermeistens gehen dabei Bits verloren!

- Konsequenz: die Addition auf Floats ist nicht assoziativ!

$$(a+b)+c \neq a+(b+c)$$

- Beispiel:

$$-10^{20} + (10^{20} + 1) = 0$$

$$(-10^{20} + 10^{20}) + 1 = 1$$

- Auch das Distributivgesetz gilt nicht!
- Zum Glück gilt aber Kommutativität

Das Maschinen-Epsilon

- Definition Maschinen-Epsilon:
  ε_m ist die kleinste Zahl, so daß
  $$1.0 + \varepsilon_m \neq 1.0$$
- Wenn $x \in \mathbb{R}$ und $\tilde{x} \in \mathbb{F}$ durch Rundung aus x hervorgeht, dann gilt also
  $$|x|(1.0 - \varepsilon_m) \leq |\tilde{x}| \leq |x|(1.0 + \varepsilon_m)$$
- Das Maschinen-Epsilon kann man in den meisten Sprachen abfragen:
  - In C: `FLT_ESILON` in `float.h`
  - In C++: `numeric_limits<float>::epsilon()`
  - In Python: `sys.float_info.epsilon`

- Die Darstellungsgenauigkeit von Floats:

Decimal x	Hex	Next representable number
10.0	0x41200000	x + 0.000001
100.0	0x42C80000	x + 0.000008
1,000.0	0x447A0000	x + 0.000061
10,000.0	0x461C4000	x + 0.000977
100,000.0	0x47C35000	x + 0.007813
1,000,000.0	0x49742400	x + 0.0625
10,000,000.0	0x4B189680	x + 1.0

- Fazit: es macht überhaupt keinen Sinn, Floats mit mehr als 8 Stellen auszugeben!

Akkumulation von Rundungsfehlern

- Sei ein Algorithmus mit N Rechenoperationen gegeben
- Wenn Rundungsfehler sowohl nach oben als auch nach unten zufällig und gleichverteilt sind, dann ist der Rundungsfehler am Ende des Algorithmus

$$\sqrt{N}\,\varepsilon_m$$

- In der Praxis ist es sehr häufig so, daß die Rundungsfehler nicht gleichverteilt sind! Dann gilt als Gesamtrundungsfehler eher

$$N\varepsilon_m$$

Beispiel: Berechnung der Standardabweichung

- Bezeichne mit
  σ = Standardabweichung
  μ = Mittelwert
- Seien n Datenwerte $x_1, \ldots, x_n$ gegeben
- Die Standardabweichung ist definiert als

$$\sigma_n^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \mu_n)^2$$

wobei

$$\mu_n = \frac{1}{n}\sum_{i=1}^{n} x_i$$

- Der naive Algorithmus:

```
mu = 0
for x in data:
    mu += x
mu /= len(data)
sigma2 = 0
for x in data:
    sigma2 += (x - mu) * (x - mu)
sigma2 /= len(data)
```

$$\sigma_n^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \mu_n)^2 \qquad \mu_n = \frac{1}{n}\sum_{i=1}^{n} x_i$$

Nachteile des naiven Algorithmus

- Benötigt 2 Durchläufe
  - Kann unpraktikabel oder unmöglich sein, z.B., wenn die Daten über viele Monate von einem Sensor kommen (z.B. in einem Satelliten)
  - Kann langsam sein, da man 2x durch den Speicher muß, und Zugriff auf den Speicher kostet viel Zeit
- Ergibt große Rechenfehler bei großen Datensätzen!
  - Grund: nach einer Weile werden Zahlen sehr verschiedener Größenordnung addiert

```
   1.xxxxxxxxxxxxxxxxxxxxxxxx          ← aktuelles sigma2
+             1.yyyyyyyyyyyyyyyyyyyyyyyy  ← aktuelles (x-mu)²
  ---------------------------------------
  zz.zzzzzzzzzzzzzzzzzzzzzz
```

- Definiere der Einfachheit halber

$$S_n = \sum_{i=1}^{n} (x_i - \mu_n)^2$$

- Gesucht ist also eine Methode, um S_n inkrementell zu berechnen, wobei in den Formeln nur Zahlen vorkommen sollten, die ungefähr in einer ähnlichen Größenordnung sind
  - D.h.: $S_n = f(S_{n-1} , x_n)$

- Dazu brauchen wir eine Rekurrenzrelation für μ_n und S_n
- Für μ_n gilt:

$$\mu_n = \frac{(n-1)\mu_{n-1} + x_n}{n} \qquad (1)$$

$$\mu_n = \mu_{n-1} + \frac{1}{n}(x_n - \mu_{n-1}) \qquad (2)$$

- Weiter gilt:

$$x_n - \mu_n = x_n - \mu_{n-1} - \frac{1}{n}(x_n - \mu_{n-1}) \qquad \text{[2 eingesetzt]}$$

$$= \frac{n-1}{n}(x_n - \mu_{n-1}) \qquad (3)$$

- Nun noch S_n umformen:

$$S_n = \sum_{i=1}^{n} (x_i - \mu_n)^2$$

$$= (x_n - \mu_n)^2 + \sum_{i=1}^{n-1} \left(x_i - \mu_{n-1} - \frac{1}{n}(x_n - \mu_{n-1})\right)^2$$ [2 eingesetzt]

$$= (\frac{n-1}{n})^2 (x_n - \mu_{n-1})^2 + \sum_{i=1}^{n-1} (x_i - \mu_{n-1})^2$$ [ausmultipliziert und für den ersten Term (3) eingesetzt]

$$- \sum_{i=1}^{n-1} \frac{2}{n}(x_i - \mu_{n-1})(x_n - \mu_{n-1}) + \sum_{i=1}^{n-1} \frac{1}{n^2}(x_n - \mu_{n-1})^2$$

- Zwischenrechnung:

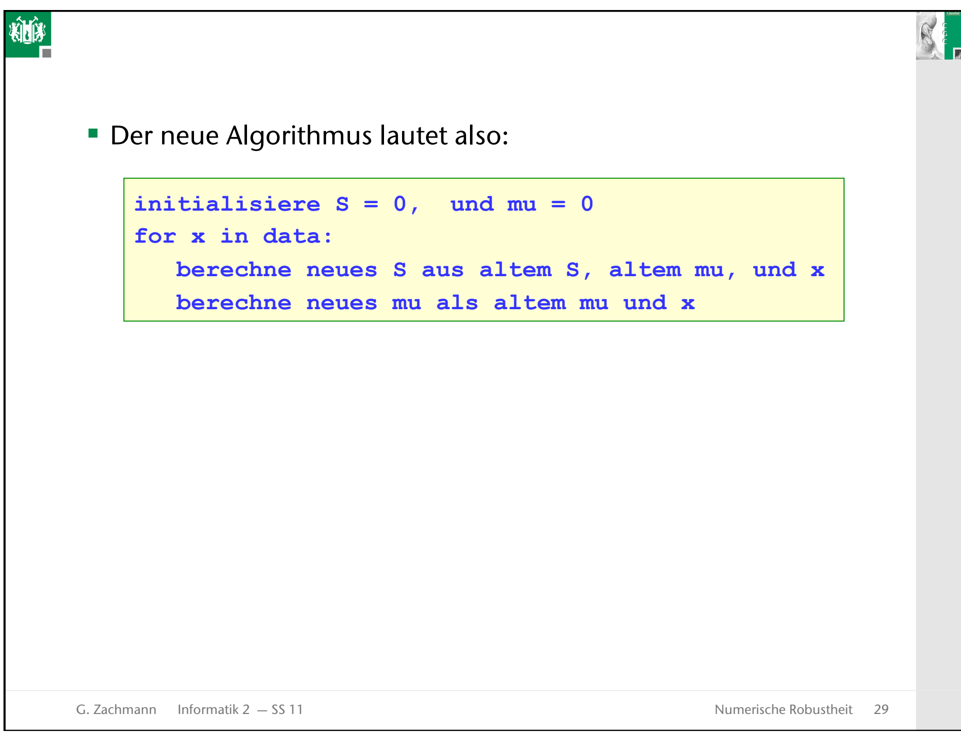
$$\sum_{i=1}^{n-1} \frac{2}{n}(x_i - \mu_{n-1})(x_n - \mu_{n-1}) =$$

$$\frac{2}{n}(x_n - \mu_{n-1}) \left(\underbrace{\sum_{i=1}^{n-1} x_i}_{(n-1)\mu_{n-1}} - \underbrace{\sum_{i=1}^{n-1} \mu_{n-1}}_{(n-1)\mu_{n-1}} \right) = 0$$

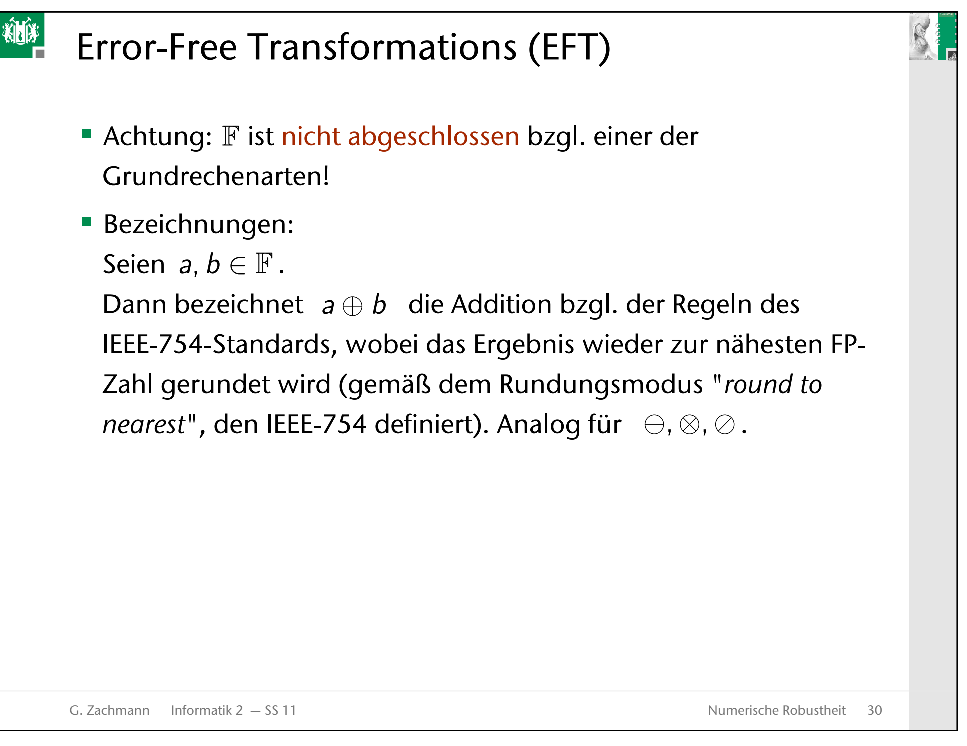
- Damit wird

$$S_n = (\frac{n-1}{n})^2 (x_n - \mu_{n-1})^2 + \sum_{i=1}^{n-1} (x_i - \mu_{n-1})^2 + \frac{n-1}{n^2}(x_n - \mu_{n-1})^2$$

$$= S_{n-1} + \frac{n-1}{n}(x_n - \mu_{n-1})^2$$



Der neue Algorithmus lautet also:
initialisiere S = 0, und mu = 0
for x in data:
berechne neues S aus altem S, altem mu, und x
berechne neues mu als altem mu und x
G. Zachmann Informatik 2 — SS 11
Numerische Robustheit 29



Error-Free Transformations (EFT)
Achtung: 𝔽 ist nicht abgeschlossen bzgl. einer der Grundrechenarten!
Bezeichnungen:
Seien a, b ∈ 𝔽.
Dann bezeichnet a ⊕ b die Addition bzgl. der Regeln des IEEE-754-Standards, wobei das Ergebnis wieder zur nähesten FP-Zahl gerundet wird (gemäß dem Rundungsmodus "round to nearest", den IEEE-754 definiert). Analog für ⊖, ⊗, ⊘.
G. Zachmann Informatik 2 — SS 11
Numerische Robustheit 30

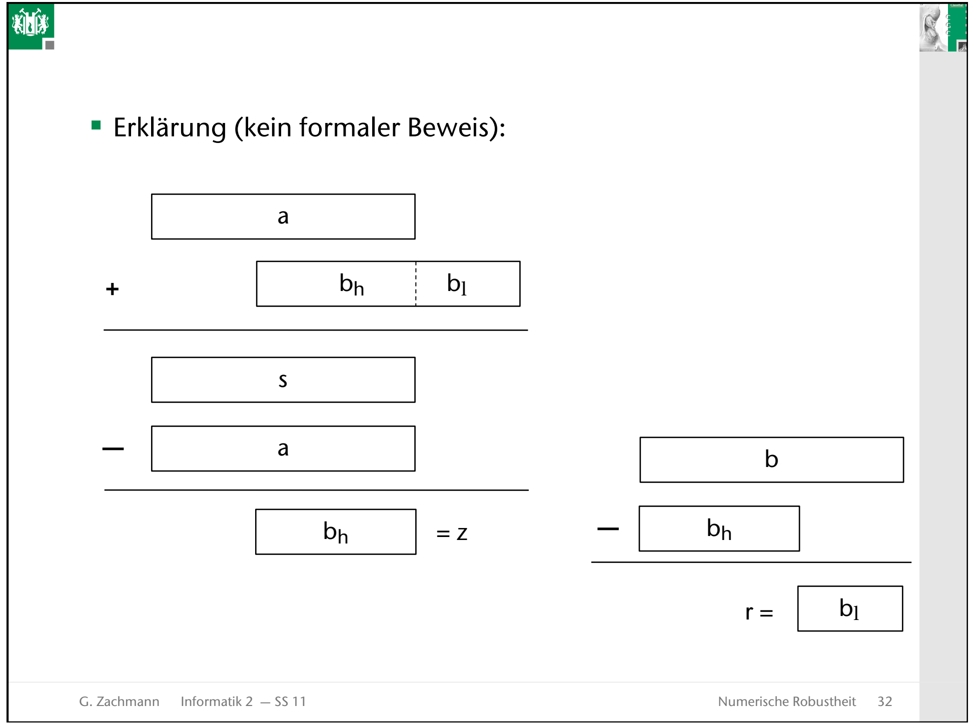
- Satz:
  Seien $a, b \in \mathbb{F}$, mit $|a| \geq |b|$.
  Dann gibt es Zahlen $s, r \in \mathbb{F}$, so daß
  1. $s + r = a + b$ exakt gilt; und
  2. s die FP-Zahl ist, die am nähesten an $a + b$ liegt.

  Solch eine Äquivalenz heißt *error-free transformation* (EFT).

- Der Algorithmus zur Konstruktion der Zahlen s und r (d.h., die Transformation von a, b in s, r) nach Dekker:

```
TwoSum(a, b ) → s, r:
if |a| < |b|:
   swap( a, b )
s = a + b
z = s − a
r = b - z
```


- Erklärung (kein formaler Beweis):



- Der Algorithmus von Møller-Knuth:
  - Das **if** ist sehr kostspielig, vor allem, wenn dieser Algorithmus sehr oft aufgerufen wird (als Ersatz für das eingebaute "+").
  - Die Idee: führe einfach vorigen Algorithmus "in beide Richtungen" aus, d.h., unter der Annahme, daß $|a| \geq |b|$, als auch unter der Annahme, daß $|a| \leq |b|$
  - Damit erhalten wir die folgende EFT [Møller-Knuth]:

```
TwoSum(a, b ) → s, r:
s  = a + b
a' = s - b
b' = s - a
δa = a - a'
δb = b - b'
r  = δa + δb
```

6 FP-Operationen sind schneller als 3 FLOPS und ein **if**

Der Fehler der naiven Summationsformel

- Sei eine Folge von Zahlen $x_1, \ldots, x_n \in \mathbb{F}$ gegeben
- Betrachte zunächst den "kanonischen" Algorithmus zur Summation:

```
s = 0
for i = 1,...,n:
    s += x_i
```

- Der Fehler im Endergebnis:
  - Setze $s_1 = x_1$, $s_2 = s_1 \oplus x_2$, ... , $s_i = s_{i-1} \oplus x_i$
  - Der Fehler in s_n ist somit

$$\begin{aligned} s_n &= (s_{n-1} + x_n)(1 + \delta_n) \\ &= (s_{n-2} + x_{n-1})(1 + \delta_{n-1})(1 + \delta_n) + x_n(1 + \delta_n) \\ &\approx \sum_{j=1}^{n} x_j \left(1 + \sum_{k=j}^{n} \delta_k\right) = \sum_{j=1}^{n} x_j + \sum_{j=1}^{n} x_j \left(\sum_{k=j}^{n} \delta_k\right) \end{aligned}$$

wobei die $|\delta_i| \leq \varepsilon_m$

- Indem man alle Delta's nach oben abschätzt, kann man s_n nach oben abschätzen:

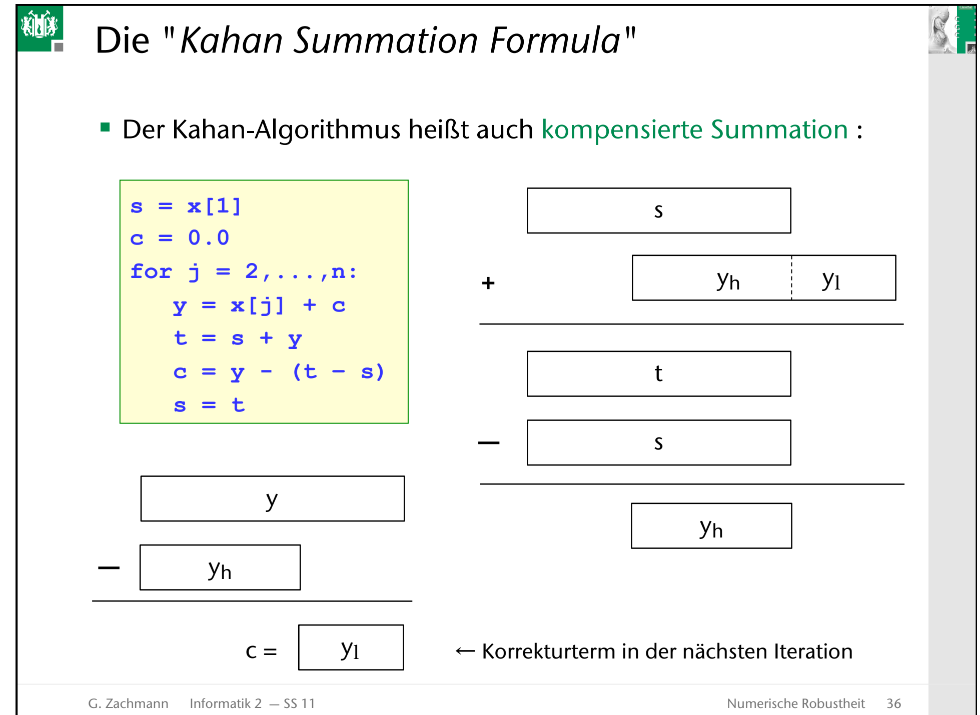
$$s_n \leq \sum_{j=1}^{n} x_j + n\varepsilon_m \sum_{j=1}^{n} x_j$$

- Damit wird der relative Fehler des Algorithmus':

$$\frac{|s_n - \sum_{j=1}^{n} x_j|}{|\sum_{j=1}^{n} x_j|} \leq n\varepsilon_m$$

Die "*Kahan Summation Formula*"

- Der Kahan-Algorithmus heißt auch kompensierte Summation :



- Der relative Fehler von Kahan's Summation Algorithm ist:

$$\frac{|s_n - \sum_{j=1}^{n} x_j|}{|\sum_{j=1}^{n} x_j|} \leq 2\varepsilon_m + O\left(n\varepsilon_m^2\right)$$

- Ein alternativer Ansatz: verwende rekursiv eine Kaskade von TwoSum's