

Warnung

Alles wird beliebig viel schwieriger wenn

- ▶ einige Variablen ganzzahlig sein müssen
- und / oder
- ▶ die Lösung gegebenen Ungleichungen genügen muss wie in der Optimierung üblich.

Binärdarstellung, d.h. Basis $b = 2$

ist die am häufigsten verwendete Basis von Gleitkommazahlen
 Auch $b = 10$ wird zuweilen in Hardware verwendet.

Arten von Gleitkommazahlen

- ▶ **normalisierte Gleitpunktzahl:**

$$m_1 > 0 \implies \frac{1}{b} \leq m \leq |x b^{-e}| < 1$$

$$x = \pm 0.m_1 m_2 m_3 \dots m_l \cdot b^e \text{ with } m_1 > 0 \implies \text{eindeutige Darstellung}$$

- ▶ **unnormalisiert:** $m_1 = 0$ zugelassen \implies keine Eindeutigkeit
- ▶ **denormalisiert:** $m_1 = 0, e = e_{\min}$

Vorsicht:

Rechnen mit denormalisierten Zahlen führt zu verstärkten Rundungseffekten.

D-2 Gleitkommaarstellung und -arithmetik

Ein System von Gleitkommazahlen wird definiert durch:

- ▶ Basis (oder Radix) \mathbf{b} (= üblicherweise 2)
- ▶ Mantissenlänge \mathbf{l}
- ▶ Minimaler Exponent \mathbf{e}_{\min}
- ▶ Maximaler Exponent \mathbf{e}_{\max}

Teilmenge der reellen Zahlen \mathbb{R} mit Darstellung

$$x = (-1)^s \underbrace{0.m_1 m_2 \dots m_l}_{\text{Mantisse } m} b^e \sim (-1)^s [m_1 b^{e-1} + m_2 b^{e-2} + m_3 b^{e-3} + \dots + m_l b^{e-l}]$$

Vorzeichenbit s , Mantisse m , Exponent e

$$s \in \{0, 1\} \quad m_i \in \{0, 1, \dots, b-1\} \quad e \in \{e_{\min}, e_{\min} + 1, \dots, e_{\max}\}$$

Betragsmässig kleinste normalisierte Zahl TINY

$$\text{TINY} = 0.1 \cdot b^{e_{\min}} = b^{e_{\min}-1}$$

Betragsmässig größte normalisierte Zahl HUGE

$$\text{HUGE} = 0.(b-1)(b-1)(b-1)\dots(b-1)\dots b^{e_{\max}} = b^{e_{\max}}(1 - b^{-l})$$

Epsilon (relative Maschinengenauigkeit) ε

ist die kleinste Zahl ε für die $1 + \varepsilon$ in Gleitkommaarithmetik nicht 1 ergibt, d.h. $\varepsilon \approx b^{-l}$

Merke:

- ▶ Mantissenlänge l bestimmt die Rechengenauigkeit.
- ▶ Exponentenbereich $e_{\max} - e_{\min}$ bestimmt den Wertebereich.

Beispiel D.1 (Gleitpunktzahlsystem mit Basis 2 und Mantissenlänge 3)

$x = 0.m_1 m_2 m_3 2^e$ Exponentenbereich $-1 \leq e \leq 1$
 Normalisierte positive Zahlen: $m_1 = 1, m_2 \in \{0, 1\} \ni m_3$
 Denormalisierte positive Zahlen: $m_1 = 0, e = -1, m_2 \in \{0, 1\} \ni m_3$

denormalisiert $TINY = \frac{1}{4}, HUGO = \frac{7}{4}, EPSILON = \frac{1}{8}$

e	-1	-1	-1	-1	-1	-1	-1	0	0	0	0	1	1	1	1
m_1	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1
m_2	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1
m_3	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0

0 $\frac{1}{16}$ $\frac{1}{8}$ $\frac{3}{16}$ $\frac{1}{4}$ $\frac{5}{16}$ $\frac{3}{8}$ $\frac{7}{16}$ $\frac{1}{2}$ $\frac{5}{8}$ $\frac{3}{4}$ $\frac{7}{8}$ 1 $\frac{5}{4}$ $\frac{3}{2}$ $\frac{7}{4}$

Gleitpunktoperationen

Bemerkenswert

$$(1.0 / 8.0) * 8.0 = 1.0$$

$$(1.0 / 5.0) * 5.0 \neq 1.0$$

Konsequenz

Gleitpunktoperationen stören normale algebraische Rechenregeln, insbesondere Distributivität:

Im Allgemeinen gilt $(a + b) * c \neq a * c + b * c$.

Man muss sich also über die Reihenfolge der Anwendung von Operationen Gedanken machen.

Beispiel D.2 (Einfache genaue Gleitkommazahlen im Salford Fortran 95 Compiler)

$b = 2, l = 24, e_{min} = -125, e_{max} = 128$

$$HUGO \approx 2^{128} = (2^{10})^{12.8} \approx (10^3)^{12.8} \approx 10^{38}$$

$$TINY \approx 2^{-125-1} = (2^{10})^{-12.6} \approx (10^3)^{-12.6} \approx 10^{-38}$$

$$Epsilon \approx 2^{-24} = (2^{10})^{-2.4} \approx (10^3)^{-2.4} \approx 10^{-7}$$

Folgerung D.3

Bei Verwendung der Gleitkommazahlen des Salford Fortran 95 Compilers in Standardgenauigkeit wird mit etwa **sieben signifikanten Dezimalstellen** gerechnet.

Allgemein gültiger Standard: ANSI - IEEE 754

(ANSI → American National Standards Institute und IEEE → Institute of Electrical and Electronics Engineering.)

Grundideen:

- (i) Alle Zwischenergebnisse werden zur nächsten Gleitpunktzahl gerundet.
- (ii) *The show must go on.* Auch bei Fehlern wird weiter gerechnet.

Zu Grundidee (i) – Rundung von Zwischenergebnissen

Auch wenn x und y im Gleitpunktbereich liegen, gilt dies im Allgemeinen nicht für das Ergebnis $x \circ y$, wobei $\circ \in \{-, +, \cdot, /\}$. Dann wird $x \circ y$ zunächst mit erhöhter Genauigkeit berechnet und anschließend zur nächstliegenden Gleitpunktzahl gerundet.

Rundungsarten

- $\nabla(x \circ y)$ nach unten gerundet
(größte untere Schranke im Gleitpunktbereich)
- $\Delta(x \circ y)$ nach oben gerundet
(kleinste obere Schranke im Gleitpunktbereich)

Verhältnis der Rundung nach oben und unten

Falls e gemeinsamer Exponent von $\Delta(x \circ y)$ und $\nabla(x \circ y)$ ist, dann gilt

$$\begin{array}{ccc} \Delta(x \circ y) & - & \nabla(x \circ y) \leq 2^{-l} 2^e \leq 2^{-l} 2 \cdot |x \circ y|, \quad \text{da } |x \circ y| \geq \frac{1}{2} 2^e \\ \parallel & & \parallel \\ 0.m \cdot 2^e & & 0.\tilde{m} \cdot 2^e \end{array}$$

Warnung:

Rundungsfehler entstehen in (fast) jeder einzelnen Operation und pflanzen sich fort.

Algorithmen (z.B. zur Matrixfaktorisierung) müssen deswegen auf ihre Stabilität, d.h. die Verstärkung oder Abdämpfung von Rundungsfehlern, untersucht werden.

Beispiel D.4

Gaussche Elimination ohne Pivotierung ist extrem instabil.

Gauss mit Pivotierung ist dagegen recht stabil.

Bezeichnet man also mit $\square(x \circ y) \in \{\nabla(x \circ y), \Delta(x \circ y)\}$ die Gleitpunktzahl, die am nächsten zu $x \circ y$ liegt, so gilt

$$|\square(x \circ y) - x \circ y| \leq \frac{1}{2} |\Delta(x \circ y) - \nabla(x \circ y)| \leq 2^{-l} |x \circ y| \leq \text{eps} \cdot |x \circ y|$$

wobei $\text{eps} = 2^{-l}$ die relative Maschinengenauigkeit ist.

Alternative Schreibweise:

$$\boxed{fl(x \circ y) = (x \circ y) * (1 + \varepsilon), \quad \text{wobei } |\varepsilon| \leq \text{eps}.}$$

$fl(x \circ y)$ bezeichnet das in Gleitpunktarithmetik erzielte Ergebnis für $x \circ y$.

Konsequenz für relativen Fehler:

$$\left| \frac{fl(x \circ y) - (x \circ y)}{x \circ y} \right| \leq |\varepsilon| \leq \text{eps}$$

Frage

Was passiert, wenn $x \circ y$ außerhalb des Wertebereichs $[-\text{HUGE}, \text{HUGE}]$ liegt, d.h. entweder $\nabla(x \circ y)$ oder $\Delta(x \circ y)$ nicht existiert?

Beispiel D.5 (Programm)

```
REAL u,s,t
s = TINY(u)**2    ! ergibt 0
t = HUGE(u)*8    ! ergibt INF, signalisiert OVERFLOW
```

Zu Grundidee (ii) – Fortsetzung der Berechnung trotz Fehlers

Mit INF und -INF kann (soweit es geht) *normal* weiter gerechnet werden, ohne dass sich je wieder normale Zahlen ergeben.

(Einige) Rechenregeln

$x + \text{INF} == \text{INF}$ für alle $x \neq -\text{INF}$
 $x * \text{INF} == \text{sign}(x) * \text{INF}$ für $x \neq 0$
 $x / 0 == \text{sign}(x) * \text{INF}$ für $x \neq 0$

wobei $\text{sign}(x)$ das Vorzeichen von x liefert.

Undefinierte Operationen wie $0/0$, INF/INF , $\text{INF}-\text{INF}$ und $0*\text{INF}$ ergeben den sehr speziellen Wert $\text{NaN} \approx \text{Not a Number}$.

Da ein NaN nicht mit sich selbst oder etwas anderem verglichen werden kann, gilt

$x \neq x$.EQUIV. .TRUE.

genau dann wenn x ein NaN ist.

- 21 -

D-3 Summation numerischer Reihen

Fehlerfortpflanzung

Erinnerung:

$f(x \circ y) = x \circ y * (1 + \varepsilon)$ mit $-\text{eps} \leq \varepsilon \leq \text{eps}$ wobei $\circ \in \{+, -, *, /\}$

Prinzip Hoffnung für komplexe Berechnungen

Da Auf- oder Abrunden mehr oder minder zufällig auftreten hebt sich deren Wirkung (hoffentlich) im Großen und Ganzen auf.

- 23 -

Infektionsprinzip:

Wenn immer ein NaN als Argument oder Operator einer Operation auftritt sind die Ergebnisse wiederum NaN s.

Auf diese Weise wird der gesamte Berechnungszweig als ungültig ausgewiesen.

- 22 -

Positives Beispiel: Geometrische Reihe:

$$s = \sum_{i=0}^n x^i = \frac{1 - x^{n+1}}{1 - x} \quad \text{falls } x \neq 1$$

Einfach genaues Auswertungsprogramm in Fortran 95

```
INTEGER i,n
REAL(KIND=1) x,y,s
REAL(KIND=2) check
s = 0
y = 1
DO i = 0, n
    s = s+y ; y = y*x
END DO
check = x ; eps = EPSILON(x)
check = (1-check**(n+1))/(1-check)
WRITE(*,*) s,check,s/check-1,n*eps
```

! Partialsumme
! jeweils Potenz von x

- 24 -

Programm ergibt für $n = 100$ und $x = 2.0/3.0$

s	check	s/check - 1	n * eps
3.00000002	3.00000019	$2 \cdot 10^{-8}$	$1.2 \cdot 10^{-5}$

Beobachtungen

- ▶ Gleitpunktwert von x ist offenbar größer als $\frac{2}{3}$ (durch Rundung), da beide Summen größer als

$$1 + \frac{2}{3} + \left(\frac{2}{3}\right)^2 + \dots + \left(\frac{2}{3}\right)^n = 3 \underbrace{\left(1 - \left(\frac{2}{3}\right)^{n+1}\right)}_{\leq 1} \leq 3$$

- ▶ Der beobachtete relative Fehler zwischen einfach und doppelt genauer Lösung ist lediglich $2 \cdot 10^{-8}$, d.h. von der Größenordnung der Maschinengenauigkeit, obwohl wir 100 Operationen durchgeführt haben. Die Rundungen scheinen sich partiell aufgehoben zu haben.
- ▶ Eine exakte Abschätzung für den *worst case* (d.h. schlimmster Fall) ergibt den Wert $(1 + \text{eps})^{100} \approx 100 \cdot \text{eps}$ als relativen Fehler. Das lässt sich wie folgt herleiten.

Entsprechend erhält man für die Partialsummen $s_{i+1} = fl(s_i + y_i)$ als berechnete Werte von $1 + x \dots + x^{i+1}$

$$s_1 = fl(y_0 + y_1) = fl(1 + x) = (1 + x)(1 + \varepsilon_{n+1})$$

$$\begin{aligned} s_2 &= fl(s_1 + y_2) = fl(s_1 + y_2)(1 + \varepsilon_{n+2}) \\ &= ((1 + x)(1 + \varepsilon_{n+1}) + x^2(1 + \varepsilon_2))(1 + \varepsilon_{n+2}) \\ &= (1 + x + x^2)(1 + \tilde{\varepsilon}_{n+2})^2 \quad \text{für } |\tilde{\varepsilon}_{n+2}| \leq \text{eps} \end{aligned}$$

$$s_n = (1 + x + x^2 + \dots + x^n)(1 + \tilde{\varepsilon}_{2n})^n \leq s(1 + \varepsilon)^n$$

so dass falls $\text{eps} \ll \frac{1}{n} \iff n \cdot \text{eps} \ll 1$

$$|(s_n/s - 1)| = |(1 + \varepsilon)^n| - 1 = 1 + n \cdot \varepsilon + \frac{n \cdot (n-1)}{2} \varepsilon^2 \dots - 1 \approx n \cdot |\varepsilon| \leq n \cdot \text{eps}$$

Ergebnis: *Worst case error* - Abschätzung:

$$|s_n/s - 1| \approx n \cdot \text{eps}$$

Theoretische Schranke des Fehlers im obigen Programm

Für $y_{i+1} = fl(y_i * x)$ als berechneter Wert von y im i -ten Schritt gilt:

$$\begin{aligned} y_0 &= 1 \\ y_1 &= x \\ y_2 &= fl(y_1 \cdot x) = x^2(1 + \varepsilon_2) \\ y_3 &= fl(y_2 \cdot x) = x^3(1 + \varepsilon_2)(1 + \varepsilon_3) = x^3(1 + \tilde{\varepsilon}_3)^2 \\ y_4 &= fl(y_3 \cdot x) = x^4(1 + \tilde{\varepsilon}_2)^2(1 + \varepsilon_4) = x^4(1 + \tilde{\varepsilon}_4)^3 \\ &\vdots \\ y_i &= x^i(1 + \tilde{\varepsilon}_i)^{i-1} \\ &\vdots \\ y_n &= x^n(1 + \tilde{\varepsilon}_n)^{n-1} \end{aligned} \quad \text{wobei } |\tilde{\varepsilon}_3| \leq \text{eps}$$

Negatives Beispiel (d.h. Prinzip Hoffnung versagt) : Harmonische Reihe

$$\sum_{i=1}^{\infty} \frac{1}{i} = \begin{cases} \infty & \text{(mathematisch, in exakter Arithmetik)} \\ 15.403 & \text{auf Griewank's Laptop, in einfacher Genauigkeit} \\ & \text{(für alle hinreichend großen Summations-Schranken} \\ & \text{= Zahl der Terme)} \end{cases}$$

Frage:

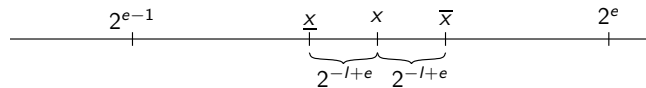
Was passiert?

Antwort:

Die Summation bleibt irgendwann *liegen*, da die zusätzlichen Terme im Vergleich zur berechneten Teilsumme zu klein werden.

Erklärung:

Betrachte *kleinen* Summanden y und *großen* Summanden $x = 0.m_1m_2 \dots m_l \cdot 2^e$ so dass $\bar{x} = x + 2^{-l+e}$ die nächst größere Gleitpunktzahl zu x ist und $\underline{x} = x - 2^{-l+e}$ ist die nächst kleinere Gleitpunktzahl zu x .



Konsequenz:

Falls $|y| < \frac{1}{2} 2^{-l+e} = 2^{-l-1+e}$ gilt immer $fl(x + y) = x$.

Eine hinreichende Bedingung ist: $|y| \leq |x| \cdot \text{eps}$.

Beispiel D.6 (Programm, das die harmonische Reihe summiert, bis die Partialsummen konstant bleiben:)

```
REAL(KIND=1) salt,sneu,one
salt = -1 ; sneu = 0 ; one = 1.0 ; n = 1
DO WHILE (sneu ≠ salt)
    salt = sneu
    sneu = sneu+one/n
    n = n+1
END DO
WRITE(*,*) sneu,n
```

Ergebnis auf Griewank's Laptop

sneu = 15.403...
 n = 2097152 $\approx 2 \cdot 10^6$
 Laufzeit $\approx \frac{1}{6}$ Sekunde

D.h. obiger Schleifenkörper wird in etwa 10^7 mal pro Sekunden ausgeführt (entspricht ca. 10 Megaflops, d.h. 10 Millionen Operationen/Sekunde.)

Am Beispiel der *harmonischen Reihe* gilt nach $(n - 1)$ Termen:

$$x = \sum_{i=1}^{n-1} \frac{1}{i} \approx \int_1^n \frac{1}{z} dz = \ln(n).$$

Also bleibt die Summation *liegen* (d.h. die Partialsummen wachsen nicht mehr weiter) wenn

$$|y| = \frac{1}{n} \approx \ln(n) \cdot \text{eps}$$

was auf jeden Fall gilt wenn

$$n \gtrsim \frac{1}{\text{eps} \cdot \ln(n)}$$

Vergleich zur theoretischen Herleitung

$$n = 2097152 \text{ ergibt } \ln(n) * n * \text{EPSILON}(x) = 3.6$$

Frage:

Was passiert bei Ausführung des obigen Programms, wenn statt mit einfacher Genauigkeit (d.h. KIND=1) nun mit doppelt genauen Gleitkommazahlen (d.h. KIND=2) gerechnet wird?

Antwort:

Das Programm läuft *ewig*, da eps^{-1} und damit dann auch n um Faktor $2^{53}/2^{24} \approx 2^{29} \approx \frac{1}{2} 10^9$ gewachsen ist.

In Sekunden:

$$\frac{1}{6} \cdot \frac{1}{2} \cdot 10^9 \text{ s} = \frac{10^8}{36 \cdot 10^3} \text{ h} = 25 \cdot 10^4 \text{ h} = 25.000 \text{ Stunden} \approx 1000 \text{ Tage.}$$