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:

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

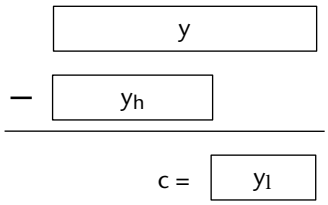
- Damit wird der relative Fehler des Algorithmus':

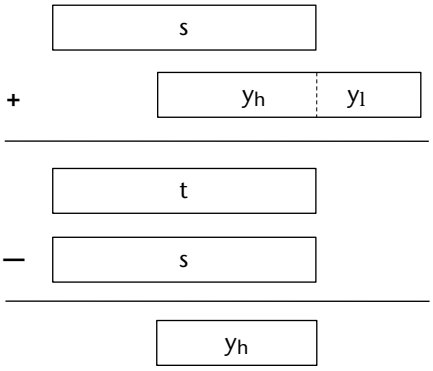
$$\frac{\left|s_n - \sum_{j=1}^{n} x_j\right|}{\left|\sum_{j=1}^{n} x_j\right|} \leq n\varepsilon_m$$

Die "*Kahan Summation Formula*"

- Der Kahan-Algorithmus heißt auch kompensierte Summation :

```
s = x[1]
c = 0.0
for j = 2,...,n:
   y = x[j] + c
   t = s + y
   c = y - (t - s)
   s = t
```



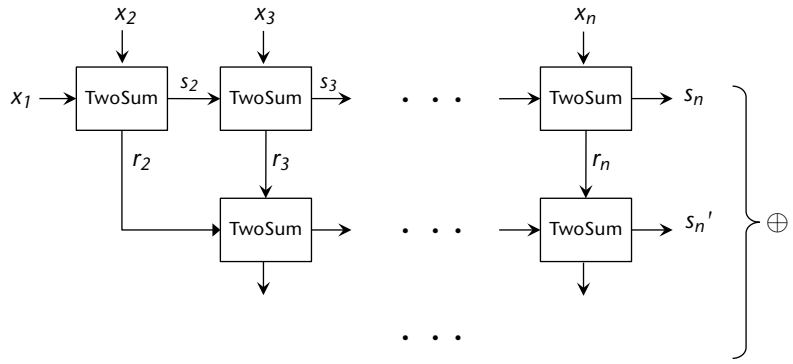


c = yl ← Korrekturterm in der nächsten Iteration

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

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

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

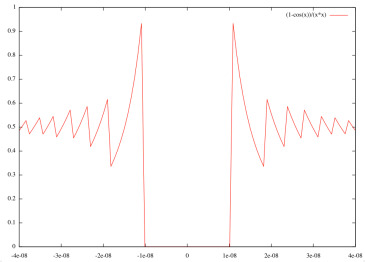


Cancellation

- Definition Cancellation (Auslöschung):
  Seien x und y zwei Floats, die N signifikante Stellen haben.
  Falls x und y auf den ersten k Stellen übereinstimmen, dann hat das Resultat von $x-y$ nur noch $N-k$ signifikante Stellen.

- Catastrophic Cancellation:
  Von *catastrophic cancellation* spricht man dann, wenn $N-k$ "klein" ist,
  z.B. $N-k = 0,\ldots,2$.

- Beispiel: die Funktion
  $$(1 - \cos(x))/x^2$$
  sollte im Nullpunkt = ½ sein.



Bsp.: Korrektes Lösen von quadratischen Gleichungen

- Aufgabe: bestimme die Nullstellen von

$$ax^2 + bx + c = 0$$

(im Folgenden oBdA die Annahme, daß $b > 0$)

- Schulmethode:

$$x_1 = \frac{-b + \sqrt{b^2-4ac}}{2a} \quad , \quad x_2 = \frac{-b - \sqrt{b^2-4ac}}{2a}$$

- Problem: falls $ac \ll b^2$
  - Dann ist $\sqrt{b^2-4ac} \approx |b|$ → Cancellation bei x_1
  - Korrekte Lösung für x_1:

$$x_1 = \frac{2c}{-b - \sqrt{b^2-4ac}}$$

- Analog falls $b < 0$

Beispiel: Auswertung von Polynomen

- Gegeben:

$$p(x) = \sum_{i=0}^{n} p_i x^i$$

- "Falschest-mögliche" Art der Implementierung:

```
p = 0
for i = 0 ... n:
   p += p[i] * pow(x,i)
```

- Fast genauso falsch ist diese
  (hier am Beispiel eines kleinen, konstanten Grades):

```
p = p[0] + p[1]*x + p[2]*x*x + p[3]*x*x*x
```

- Die korrekte Methode: Horner-Schema

$$p(x) = (\cdots((p_n x + p_{n-1})x + p_{n-2})\cdots + p_0)$$

- Pseudo-Code:

```
p = p[n]
for i = n-1 ... 0:
    p = p*x + p[i]
```


- Example of failure:

$$p(x) = (x-2)^4 \quad \text{mit} \quad x = 2.0 + 1E - 6$$

- Wahrer Wert = 10^{-24}
- Auswertung der expandierten Form

$$p(x) = x^4 - 8x^3 + 24x^2 - 32x + 16$$

mittels Horner-Schema (implementiert mit **`double`**)
liefert 0.0 !

- Grund: catastrophic cancellation
  - In jedem Schleifendurchlauf werden Zahlen der Größenordnung 10..100 addiert
  - Aber: unterschiedliche Vorzeichen
  - Um auf 10^{-24} zu kommen, müsste bis zum Schluß 10^{-24} in Zahlen der Größenordnung ~10 erhalten bleiben
  - Aber: das Maschinen-Epsilon für Doubles ist $2^{-53} \approx 10^{-16}$
- Fazit (u.a.): Nullstellensuche geht schief!

Overflow-Errors

- Overflow-Error:
  - Entsteht immer dann, wenn das Ergebnis einer Rechenoperation außerhalb des darstellbaren Bereich liegt (bzw. liegen würde)
  - Z.B. bei $10^{20} / 10^{-20}$ in single-precision (`float`)
  - Zeigt sich meistens daran, daß (einige) Ergebnisse den Wert `Inf` oder `NaN` haben

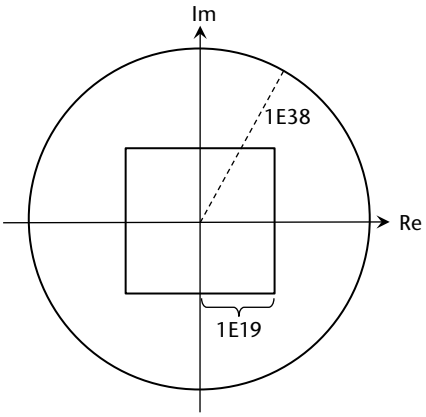
Beispiel: eine robuste Betragsoperation für komplexe Zahlen

- Betrag einer komplexen Zahl:

$$|a+ib| = \sqrt{a^2+b^2}$$

- Problem: Overflow ab

$$|a|, |b| \gtrsim 10^{19}$$



- Bessere Rechnung:

$$|a+ib| = \begin{cases} |a| \cdot \sqrt{1+\left(\frac{b}{a}\right)^2} & , |a| \geq |b| \\ |b| \cdot \sqrt{1+\left(\frac{a}{b}\right)^2} & , |a| < |b| \end{cases}$$

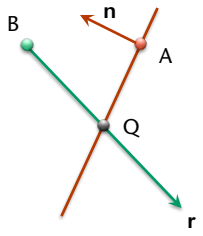
Beispiel: Berechnung des Schnitts zwischen Gerade und Ebene

- Ebene E ist gegeben durch

$$(X - A)\mathbf{n} = 0$$

- Gerade L ist gegeben durch

$$X = B + t\mathbf{r}$$



- Lösung für den Schnittpunkt Q ist

$$t^* = \frac{(A-B)\mathbf{n}}{\mathbf{r}\cdot\mathbf{n}} \qquad Q = B + t^*\mathbf{r}$$

- Was ist, wenn L und E (fast) parallel sind?
- Dann wird der Nenner in

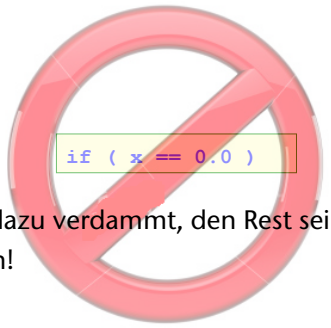
$$t^* = \frac{(A - B)\mathbf{n}}{\mathbf{r} \cdot \mathbf{n}}$$

ungefähr 0, und damit t^* = **`Inf`**!

- Fazit: im Programm bei Division den Nenner immer auf 0 testen:

```
Vector3d A, B, n, r;
float denom = dotprod( n, r );
if ( denom ist ungefähr 0.0 )
    # Ebene und Gerade sind (ungefähr) parallel
    Spezialbehandlung ...
else
    Vector3d ab = vecsub( A, B );
    t = dotprod( ab, n ) / denom;
    ...
```


- Achtung: wer



schreibt, wird dazu verdammt, den Rest seines Lebens in Lisp zu programmieren!

- Ein Lisp-Beispiel:

```
(define (dismiss lst x k)
  (cond ((null? lst) ())
    ((= (remainder x k) 0) (dismiss (cdr lst) (+ 1 x) k))
    ((cons (car lst) (dismiss (cdr lst) (+ 1 x) k)))))
```

  - Um fair zu bleiben: Lisp ist eine sehr interessante Sprache ...

Ein robuster Test auf "Gleichheit" mit Epsilon-Guard

- Aufgabe: teste, ob zwei Floats x und y "fast gleich" sind, unter Berücksichtigung der Darstellungspräzision
- Erster Ansatz:

```
if ( abs(x - y) <= epsilon )
```

  - Welchen Wert soll **epsilon** haben?
- Nächster Ansatz:

```
if ( (abs(x - y) / y) <= epsilon )
```

  - Was ist, wenn y ≈ 0? Außerdem ist Division relativ teuer

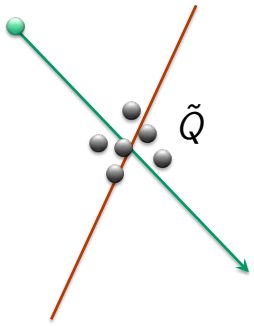
- Beste Lösung:

```
if ( abs( x-y ) <= max(abs(x), abs(y)) * epsilon )
```

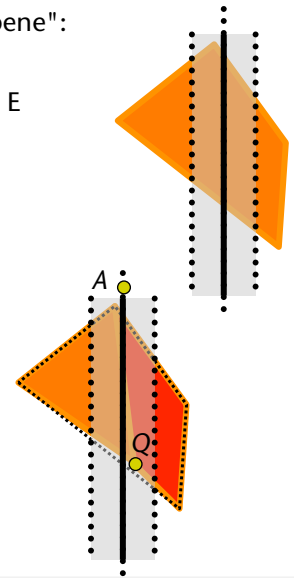
  - Achtung: wichtig ist **<=** , nicht **<** !
- Allerbeste Lösung: Aus dem Wissen über die Anwendung (Simulation von Sonnensystem oder Simulation von Molekülen?) den Skalierungsfaktor für **epsilon** ableiten
  - Oder, falls Daten verarbeitet werden: einmal vorab alle Daten scannen und daraus **epsilon** ableiten
- Bemerkung: "Epsilon-Gleichheit" ist nicht transitiv!

Robuste geometrische Prädikate

- Bemerkung: unter Rundungsfehlern (d.h. Rechnen mit Floats / Doubles) entsteht in geometrischen Algorithmen oft folgendes Bild

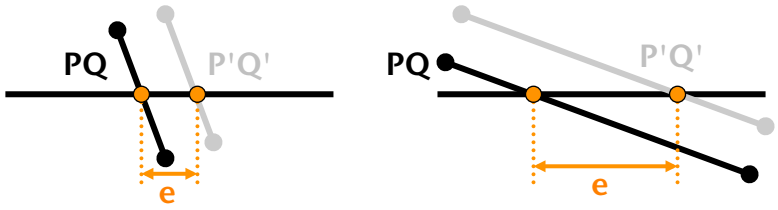


- Beispiel für ein Prädikat "Punkt ist auf Ebene":
  OnPlane(X;E) = true ⇔
  Punkt X befindet sich in der Ebene E
- Lösung mit Epsilon-Guard bedeutet:
  teste
  $$(Q - A)\mathbf{n} \leq \varepsilon_m \max(\|Q\|, \|A\|)$$
- Geometrische Interpretation:
  "fette Ebene"



Numerische Instabilität

- Stabilitätsfehler: ein Algorithmus ist numerisch instabil, wenn kleine Fehler am Anfang später große Fehler verursachen
- Am Beispiel "Schnitt zwischen Gerade und Ebene":



Exakte Arithmetik

- Idee:
  - Implementiere eine "Bignum"-Klasse, die Integer-Zahlen mit beliebig vielen Stellen repräsentieren kann
  - Implementiere darüber eine Klasse, die die rationalen Zahlen $\mathbb{Q}$ repräsentiert
  - Implementiere die üblichen Infix-Operatoren (+, -, *, /) und transzendenten Funktionen (sqrt, sin, cos, ...) durch Operator- und Funktionen-Overloading
  - Rechne nur noch mit Zahlen in $\mathbb{Q}$
- Vorteil: Algorithmen können trivial umgeschrieben werden
- Nachteile:
  - Zähler und Nenner werden schnell riesig
  - Ständiges Kürzen kostet sehr viel Zeit

Interval arithmetic

- Idee: rechne mit Intervallen, die die gesuchte / berechnete Zahl garantiert einschließt:
  - Variablen = Intervalle
  - Z.B.: x = [1,3] = { x ∈ R | 1 ≤ x ≤ 3 }
- Rechenregeln für die arithmetischen Operatoren:

$$[a, b] + [c, d] = [a + c, b + d]$$

$$[a, b] - [c, d] = [a - d, b - c]$$

$$[a, b] \times [c, d] = [\min(ac, ad, bc, bd), \max(ac, ad, bc, bd)]$$

$$[a, b]/[c, d] = [a, b] \times [1/d, 1/c] \quad \text{for } 0 \notin [c, d]$$

  - Dabei muß für die Berechnung der oberen bzw. unteren Schranke die Rundung der FPU auf "*always round down*" bzw. "*always round up*" gestellt werden!

- Beispiel: [100,101] + [10,12] = [110,113]

Das Minimum an *Take-Home Messages*

- Achtung bei Division (teste auf "ungefähr Null" !) ...
- ... und bei Subtraktion (Cancellation!)
- Bei Akkumulationsschleifen: wie groß kann die Summe werden? wie groß sind im Verhältnis dazu die Summanden?
- Speichere Floats als **`float`**, rechne zur Laufzeit mit **`double`** (d.h., deklariere lokale Variablen immer als **`double`**)
  - (In Python ist **`float`** = **`double`**)
- Extra-Vorsicht bei Casts (insbesondere von Float nach Integer!)

Literatur

- David Goldberg: *What Every Computer Scientist Should Know About Floating-Point Arithmetic*. March, 1991, Computing Surveys.
- Kahan & Darcy: *How Java's Floating-Point Hurts Everyone Everywhere*. March 1998, ACM Workshop on Java for High-Performance Computing, Stanford.
- W. Kahan: How Futile are Mindless Assessments of Roundoff in Floating-Point Computation? Jan 2006.
- Press, Teukolsky, Vetterling, Flannery: *Numerical Recipes*.