Nicht jede Zahl ist das, was sie zu sein scheint

Matthias Lehrkamp
Bayer Pharma AG
Müllerstr. 178
13353 Berlin
matthias.lehrkamp@bayer.com

Zusammenfassung

Der Computer hat nur einen begrenzten Speicherplatz, um numerische Werte darzustellen. Im Allgemeinen ist bekannt, dass es dadurch bei Berechnungen zu Ungenauigkeiten kommt. Aber wie sieht es mit einer einfachen Summe aus? Können wir uns beim Addieren auf die Genauigkeit des Computers verlassen? Die Antwort lautet: „Leider nein!“. Schon bei einfachen Berechnungen, die wir im Kopf lösen können, versagt der Computer. Dieser Bericht verrät Ihnen, wie der Rechner Zahlen abbildet und wie die Addition von Zahlen zu Fehlern führen kann.

Schlüsselwörter: Numerische Fehler, IEEE 754, Bitmuster, Dezimalzahl, addieren, Binärcode ausgeben, binary64, double floating point number, round, fuzz, trunc

1 Eine einfache Addition

Die vorliegende Arbeit ist eng an die SAS Hilfe [1] und der Wikipedia Seite zum IEEE 754 Standard [2] angelehnt. Das Fach Numerik beschäftigt sich mit Fehlern, die bei Berechnungen und Algorithmen mit Rechnern vorkommen, versucht diese zu optimieren und eine Fehlerabschätzung des Ergebnisses zu ermitteln. Die Fehlerabschätzung gibt dabei die größtmögliche Abweichung an. Jedoch kommt es schon bei der Speicherung von Kommazahlen, aufgrund der beschränkten Möglichkeiten der Hardware und der verwendeten Abbildung der Zahlen, zu Fehlern. Deswegen ist es nicht verwunderlich, wenn die folgende Gleichung als FALSCH (engl. FALSE) gewertet wird.

$0.3 = 0.1 + 0.2$

Mathematisch ist die Gleichung WAHR (engl. TRUE), sollte also als richtig gewertet werden. SAS interpretiert diese Gleichung hingegen als falsche Aussage, und deklariert die Zahl 0.3 echt kleiner als das Ergebnis von 0.1 + 0.2.

$0.3 < 0.1 + 0.2$

Dies ist kein SAS spezifisches Problem. Es liegt an dem Umgang mit Zahlen nach dem Standard IEEE 754. Dieser Standard wird hauptsächlich von Windows und UNIX Systemen verwendet. Einerseits können die rationalen Zahlen wie z.B. 1/3 nicht korrekt wiedergegeben werden, andererseits gibt es bei der Speicherung der Zahlen im Binärsystem neue unendliche, nichtdarstellbare Zahlen. Eine dieser Zahlen ist 0.1. Das Fatale

daran ist, dass der Unterschied zwischen dem Ergebnis 0.1 + 0.2 und der Zahl 0.3 in SAS nicht sichtbar wird, da die Ausgaben immer gerundet angezeigt werden.

```
2   DATA summation;
3     FORMAT x1 x2 32.31;
4     x1= 0.3;
5     x2= 0.1 + 0.2;
6     IF x1 = x2 THEN check= "x1 = x2";
7       ELSE IF x1 < x2 THEN check= "x1 < x2";
8       ELSE IF x1 > x2 THEN check= "x1 > x2";
9       ELSE check= "FEHLER";
10    PUT x1= / x2= / check=;
11  RUN;
```

```
x1=.3000000000000000000000000000000
x2=.3000000000000000000000000000000
check=x1 < x2
```

In der Ausgabe wird für beide Fälle eine 0.3 angezeigt, obwohl bereits das längste Format mit 31 Nachkommastellen gewählt wurde. Aus dem Ergebnis der Variable CHECK lässt sich jedoch erkennen, dass die Zahl 0.3 (gespeichert in x1) kleiner ist als das Ergebnis der Rechnung 0.1 + 0.2 (gespeichert in x2).

Die genauen abgespeicherten Dezimalwerte können in R ausgeben werden, indem eine höhere Ausgabegenauigkeit festgelegt wird.

```
> options( digits=22 )
> 0.1+0.2
[1] 0.3000000000000000444089
> 0.3
[1] 0.2999999999999999888978
```

Leider lässt sich eine ähnliche Funktionalität in SAS nicht finden. Dafür bietet SAS die Möglichkeit, mit dem Format binary64 das genaue Bitmuster der Zahl wiederzugeben.

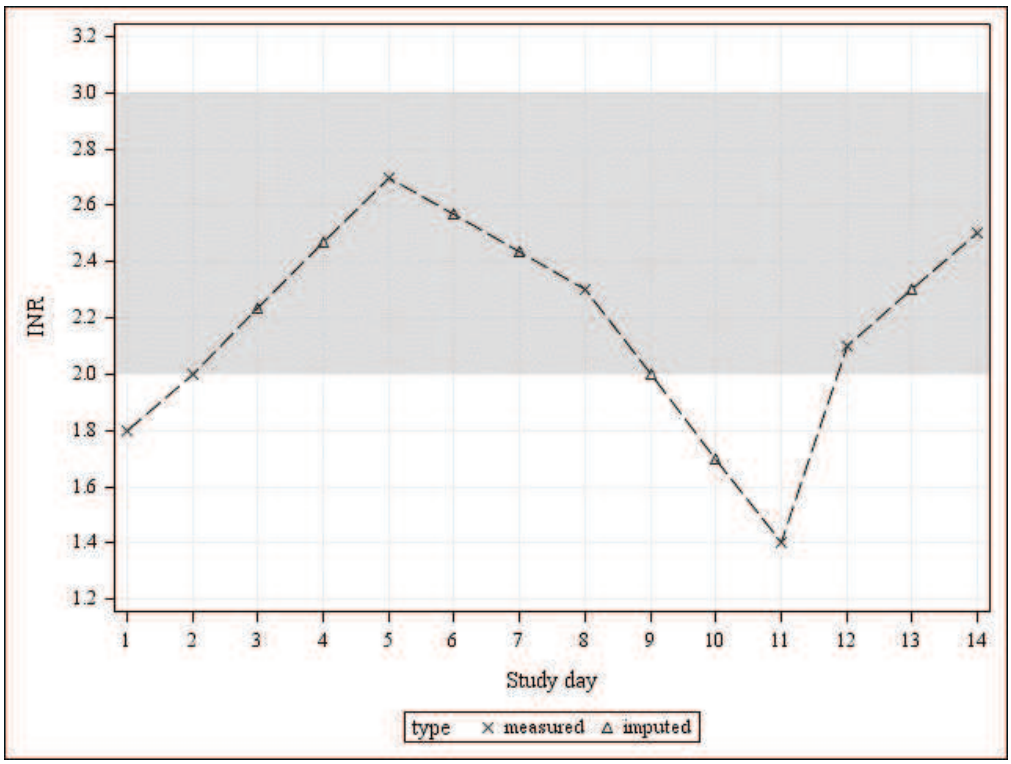
```
2  DATA _NULL_;
3    SET summation;
4    PUT x1= binary64.; /* x1= 0.3 */
5    PUT x2= binary64.; /* x2= 0.1 + 0.2 */
6    diff= x2-x1;
7    PUT diff= best.;
8  RUN;
```

```
x1=0011111111010011001100110011001100110011001100110011001100110011
x2=0011111111010011001100110011001100110011001100110011001100110100
diff=5.551115E-17
```

In der Logausgabe ist zu erkennen, dass die letzten 3 Bits unterschiedlich sind, was ein Unterschied im Bereich der 17ten Kommastelle ausmacht. Wodurch diese Unterschiede zu Stande kommen, wird in Kapitel 4 genau dargestellt. Als nächstes wird die Auswirkung des Fehlers aus einem aktuellen Beispiel einer klinischen Studie gezeigt. Im Kapitel 3 wird der IEEE 754 Standard genauer erklärt und im Kapitel 5 werden einige Hinweise gegeben, wie mit solchen Fehlern am besten umgegangen werden kann.

2 Alles im Zielbereich?

Die Fehler beschränken sich im Allgemeinen auf die hinteren Kommastellen und haben für Berechnungen kaum bis gar keine Auswirkungen. Wenn es aber auf die Genauigkeit ankommt, wie beispielsweise bei Mondflügen oder Vergleichen, sieht die Sachlage schon anders aus. An einem Beispiel soll gezeigt werden, wie viel Einfluss diese kleinen Fehler haben können. Bei einem Medikament zur Beeinflussung des Blutgerinnungsfaktors zu Verhinderung einer Thrombose, soll der Laborparameter INR (Blutgerinnungsparameter) für mindestens 75% des Beobachtungszeitraumes (14 Tage) im Zielbereich 2.0 bis 3.0 liegen (Ränder miteingeschlossen). Alle 2-3 Tage wird eine Messung vorgenommen. Wenn der Wert außerhalb des Zielbereichs liegt, werden tägliche Messungen vorgenommen. Die Tage zwischen den gemessenen Werten sollen durch eine lineare Approximation bestimmt werden. Bei einem Patienten ergab sich folgendes Ergebnis (fiktive Werte).



Die gemessenen Werte sind als Kreuz dargestellt, die linear approximierten Werte als Dreiecke. Der INR ist in der Grafik bei 11 von 14 Tagen der Gesamtzeit im Zielbereich. Das entspricht 78,57% und somit liegt der Wert über den geforderten 75% im Zielbereich. SAS hat dagegen einen Anteil von 71,43% berechnet, was gerade unterhalb der geforderten Grenze liegt. Nach einer Überprüfung stellte sich heraus, dass es sich, um einen numerischen Fehler handelt. Der approximierte Wert am Tag 9, der über die Rechnung 2.3 – (2.3 – 1.4) * 1/3 = 2.3 – 0.3 = 2.0 ermittelt wurde, wurde als echt kleiner als 2.0 interpretiert. Auch hier erkennt man den Fehler beim Vergleichen der Bitmuster.

Inzwischen haben wir gesehen, dass die Genauigkeit der berechneten Werte und abgebildeten Zahlen sehr geringen Abweichungen unterliegt. Aber genau diese kleinen Ab-

weichungen können bei Vergleichen große Auswirkungen haben und so zu falschen Ergebnissen führen.

3 Der IEEE 754 Standard

3.1 Das Bitmuster

Zahlen im Computer werden über Schaltkreise realisiert, diese wiederum haben einzelne Ein- und Ausgänge. Jeder Ausgang kann dabei den Zustand an=1 oder aus=0 haben und belegt 1 Bit Speicherplatz. 8 Bits ergeben 1 Byte und damit können die Zahlen 0-255 exakt über 2er Potenzen dargestellt werden.

2er Potenz	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Zahlenwert	128	64	32	16	8	4	2	1
Bit (an/aus)	0	1	0	1	1	0	1	0

Jedes Bit (0 oder 1) wird mit seinem Zahlenwert multipliziert und die Summe aus all diesen Werten, ergibt die darzustellende Dezimalzahl. In diesem Fall ergibt das vorgegebene Bit-Muster die Zahl 90.

$0*2^7+1*2^6+0*2^5+1*2^4+1*2^3+0*2^2+1*2^1+0*2^0 = 64+16+8+2 = 90$

Genauso funktioniert es auch bei Kommazahlen, nur das hier negative Exponenten auftreten.

2er Potenz	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}
Zahlenwert	1/2	1/4	1/8	1/16	1/32	1/64	1/128	1/256
Bit (an/aus)	0	1	0	1	1	0	1	0

Das vorgegebene Bit-Muster ergibt somit die Zahl 0.3515625.

$0*2^{-1}+1*2^{-2}+0*2^{-3}+1*2^{-4}+1*2^{-5}+0*2^{-6}+1*2^{-7}+0*2^{-8} = 0.25+0.0625+0.03125+0.0078125 = 0.3515625$

Die beiden Bytes zusammengeschrieben ergeben dann die Dezimalzahl 90.3515625.

01011010.01011010

Diese Anordnung wird auch Mantisse genannt. Der Punkt unterteilt die Stellen vor und nach dem Komma, stellt also das Dezimaltrennzeichen dar. Die Mantisse bei einer „Double Floating Point Number“ hat 52 Bits zur Verfügung.

3.2 Der Verschiebungswert

Damit das System nicht so starr ist, gibt es zusätzlich einen Verschiebungswert. Dieser Wert besteht aus 11 Bits und gibt die Verschiebungen des Punktes in der Mantisse wieder. Der Basiswert für die Kommastelle wird mit dem Verschiebungswert 1023 angegeben.

Verschiebungswert 1023

2er Potenz	2^{10}	2^{9}	2^{8}	2^{7}	2^{6}	2^{5}	2^{4}	2^{3}	2^{2}	2^{1}	2^{0}
Zahlenwert	1024	512	256	128	64	32	16	8	4	2	1
Bit (an/aus)	0	1	1	1	1	1	1	1	1	1	1

Eine Verschiebung des Punktes um x Stellen nach links wird mit 1023+x angegeben, eine Verschiebung des Punktes nach rechts wird mit 1023-x. Im Beispiel von 90.3515625 ist im Bitmuster eine Verschiebung um 6 Stellen vorzunehmen.

01011010.01011010 → 01.01101001011010

Aus Optimierungsgründen werden alle führenden Nullen und die erste 1 in der Mantisse weggelassen. Somit sieht der Verschiebungswert mit 1029 (vor dem |) und die Mantisse (nach dem |) wie folgt aus.

10000000101|01101001011010

3.3 Geht es auch Negativ?

Dem finalen Bitmuster wird noch ein Bit vorgeschoben. Dieses Bit entscheidet, ob die Zahl positiv=0 oder negativ=1 ist. Das volle 64 Bitmuster („Double Floating Point Number“) besteht aus einem Vorzeichenbit, 11 Bits für den Verschiebungswert und 52 Bits für die Mantisse – der eigentlichen Zahlendarstellung. Für die Zahl 90.3515625 sieht das volle Bitmuster wie folgt aus.

0|10000000101|0110100101101000000000000000000000000000000000000000

3.4 Bitmuster in Dezimalzahl umwandeln

Das Bitmuster lässt sich auch als mathematische Formel umschreiben. Mit dieser Formel ist auch die Umrechnung in eine Dezimalzahl möglich. Dabei muss darauf geachtet werden, dass die aus Optimierungsgründen weggelassene 1 und der Punkt wieder hinzufügt werden.

d=Dezimalzahl
z=Vorzeichenbit
v=Verschiebungswert
m=Mantisse

$d = (-1)^{z} * 2^{v-1023} * m$

Für die Dezimalzahl 90.3515625 sieht die Formel also wie folgt aus.

$90.3515625 = (-1)^{0} * 2^{1029-1023} * 1.0110100101101000…0$
$90.3515625 = 1 * 2^{6} * 1.4117431640625$
$90.3515625 = 64 * 1.4117431640625$
$90.3515625 = 90.3515625$

Nachdem die Darstellung der 64 Bitmuster („Double Floating Point Number“) nach dem IEEE 754 Standard erläutert wurde, wird im nächsten Kapitel die Addition mit dem aus Kapitel 1 gezeigten Zahlen 0.1 und 0.2 durchgeführt.

4 Die Addition nach IEEE 754

4.1 Dezimalzahl in Bitmuster umwandeln

Vor der Addition muss zunächst das Bitmuster ermittelt werden. Als Beispiel soll die Zahl 90.3515625 umgewandelt werden. Die Vorkommazahl wird umgewandelt, indem der Wert so oft durch 2 mit Rest geteilt wird, bis die 0 erreicht ist. Der jeweilige Restbetrag bildet das Bitmuster in umgekehrter Reihenfolge.

```
90/2 = 45 Rest 0
45/2 = 22 Rest 1
22/2 = 11 Rest 0
11/2 =  5 Rest 1
 5/2 =  2 Rest 1
 2/2 =  1 Rest 0
 1/2 =  0 Rest 1 --> Bitmuster: 1011010.
```

Die Nachkommazahl wird ermittelt, indem der Wert immer mit 2 multipliziert wird. Jedes Mal, wenn der Wert die 1 überschreitet, wird eine 1 subtrahiert. Das Verfahren wird solange fortgeführt, bis genau die 1 erreicht wird oder 52 Schritte durchgeführt wurden. Das Bitmuster entsteht über die Differenzen -0 und -1.

```
0.3515625*2 = 0.703125 -0
0.703125 *2 = 1.40625  -1
0.40625  *2 = 0.8125   -0
0.8125   *2 = 1.625    -1
0.625    *2 = 1.25     -1
0.25     *2 = 0.5      -0
0.5      *2 = 1        -1 --> Bitmuster: .0101101
```

Als nächstes werden die beiden Bitmuster mit einem Punkt verbunden und die Verschiebung des Punktes ermittelt.

```
1011010.0101101 --> (1.)0110100101101
```

Der Punkt muss um 6 Stellen nach links verschoben werden, das heißt, dass zu dem Basiswert 1023 6 hinzugezählt werden.

```
Verschiebungswert = 1023 + 6 = 1029 = 10000000101
```

Zum Abschluss wird das Vorzeichenbit gesetzt, 0 für positive Zahl.

```
0|10000000101|0110100101101000000000000000000000000000000000000000
```

4.2 Zurück zum Eingangsbeispiel

Weiter geht's mit der 0.1, hier gibt es keine Vorkommastellen, somit wird direkt mit den Nachkommastellen begonnen.

```
0.1*2 = 0.2 -0
0.2*2 = 0.4 -0
0.4*2 = 0.8 -0
0.8*2 = 1.6 -1
0.6*2 = 1.2 -1
0.2*2 = 0.4 -0 --> hier wiederholt sich der Vorgang
```

Aus der Umwandlung lässt sich sehr gut erkennen, dass die 0.1 ein unendliches Bitmuster hat. Dies gilt im Übrigen auch für die restlichen Zahlen, die in der Rechnung enthalten sind und auf der linken Seite stehen. Fehlt noch der Verschiebungswert.

```
0.0001100110011001100110… --> (00001.)1001100110011001100110…
```

Dieses Mal wird der Dezimalpunkt um 4 Stellen nach rechts verschoben, demzufolge ist der Verschiebungswert 1023-4 = 1019.

```
0|01111111011|1001100110011001100110011001100110011001100110011001
```

Da die Mantisse für 0.1 unendlich fortgeführt werden kann, der IEEE 754 Standard aber nur 52 Bits für die Mantisse vorsieht, wird die hintere Stelle gerundet. Der IEEE 754 Standard rundet wie beim mathematischen Runden immer zur geraden Zahl hin, im Binärsystem also stets zur „0". Zur besseren Anschauung sind im Folgenden einige Rundungen im Dezimal und Binärsystem gegenübergestellt. Die Zahl hinter dem | fällt aus dem Bitmuster wegen der technischen Begrenzung raus.

Übersicht mathematische Rundung

2.49 ≈ 2.0	00|01 ≈ 00|00
2.50 ≈ 2.0	00|10 ≈ 0**0**|00 --> Grenzfall, Abrundung zur Null
2.51 ≈ 3.0	00|11 ≈ 01|00
3.49 ≈ 3.0	01|01 ≈ 01|00
3.50 ≈ 4.0	01|10 ≈ 1**0**|00 --> Grenzfall, Aufrundung zur Null
3.51 ≈ 4.0	01|11 ≈ 10|00

Der vorletzte Fall trifft genau auf das Bitmuster der Dezimalzahl 0.1 zu. Deshalb muss entsprechend das Bitmuster mit einer Aufrundung angepasst werden.

```
0|01111111011|1001100110011001100110011001100110011001100110011010
```

Das Bitmuster der 0.2 kann aus dem Bitmuster der 0.1 abgeleitet werden, indem die Mantisse um eine Stelle nach links verschoben wird, also der Verschiebungswert um eins erhöht wird.

```
0|01111111100|1001100110011001100110011001100110011001100110011010
```

Übrigens verwendet der Prozessor beim Rechnen 80 Bits (12 zusätzliche Bits für die Mantisse und 4 zusätzliche Bits für den Verschiebungswert). Die Ergebnisse werden dann jeweils auf die 64 Bits, mit der gezeigten Rundung, zurücktransformiert.

4.3 Addition der Bitmuster 0.1 und 0.2

Die Addition erfolgt nach den üblichen Regeln, nur erstreckt sich der Zahlenbereich nicht von 0 bis 9, sondern nur von 0 bis 1. Bevor es mit der Addition losgeht, müssen beide Zahlen auf den jeweils höchsten Wert angepasst werden. In diesem Fall wird die 0.1 auf die 0.2 angepasst, indem die Mantisse um eine Stelle nach rechts verschoben wird.

0.1: Verschiebungswert wird von 01111111**011** auf 01111111**100** geändert
→ die Mantisse wird um eine Stelle nach rechts verschoben

```
1.10011001100110011001100110011001100110011001100110011010
0.11001100110011001100110011001100110011001100110011001101|0
```

Da der IEEE 754 Standard mit 80 Bit rechnet, bleibt das überstehende Bit bestehen. Die beiden Mantissen können jetzt, wie bei der schriftlichen Addition, zusammenaddiert werden.

```
 0.1:  0.11001100110011001100110011001100110011001100110011001101|0
+0.2:  1.10011001100110011001100110011001100110011001100110011010
=0.3: 10.01100110011001100110011001100110011001100110011001100111|0
```

Das Ergebnis muss um eine Stelle nach rechts verschoben werden, damit der Dezimalpunkt wieder hinter der ersten 1 steht.

Neuer Verschiebungsvektor: `01111111101`

```
1.00110011001100110011001100110011001100110011001100110011|10
```

Da das Ergebnis als nächstes gespeichert werden muss, wird das Ergebnis auf 64 Bit gerundet. Wegen der mathematischen Rundung kommt es zu einer Aufrundung.

```
1.00110011001100110011001100110011001100110011001100110100
```

Genau diese Aufrundung führt letztendlich zu dem Unterschied zur umgewandelten Zahl 0.3. Es handelt sich also um einen Rundungsfehler, resultierend durch die technische Begrenzung des Rechners selbst und der Abbildungsmethode der Zahlen nach dem IEEE 754 Standard.

```
0.1+0.2: 1.00110011001100110011001100110011001100110011001100110100
    0.3: 1.00110011001100110011001100110011001100110011001100110011
```

Somit ist 0.3 < 0.1 + 0.2.

5 Der richtige Umgang mit Zahlen am Rechner

5.1 Vor Vergleichen passend runden

Leider gibt es nicht die goldene Regel für den Umgang mit dem Rundungsfehler. In der SAS Hilfe [1] wird empfohlen, die Werte mit Hilfe der Round-Funktion zu runden. Hierbei sollte man gezielt überlegen, welcher Rundungsfaktor eingesetzt wird.

```
2   DATA test;
3       FORMAT x1 x2 32.31;
4       x1= 0.1 + 0.2;
5       x2= 0.3;
6       IF x1 = x2 THEN comp1= "Y";
7                  ELSE comp1= "N";
8       IF ROUND(x1,0.01) = ROUND(x2,0.01) THEN comp2= "Y";
9                                          ELSE comp2= "N";
10      x1= 0.295;
11      IF ROUND(x1,0.01) = ROUND(x2,0.01) THEN comp3= "Y";
12                                         ELSE comp3= "N";
13      PUT ">>> " comp1= comp2= comp3=;
14  RUN;

>>> comp1=N comp2=Y comp3=Y
```

Wie sich an diesem Beispiel gut erkennen lässt ist die Auswahl eines Rundungsfaktors schwierig. Ist er zu groß, werden auch andere Werte als gleich gewertet, obwohl diese verschieden sind. Wird dagegen der Rundungsfaktor zu klein gewählt, werden aufsummierte Rundungsfehler nicht mehr eliminiert. Hinzu kommt, dass der Rundungsfaktor mit neuen, anderen Daten nicht mehr valide ist.

5.2 Abschneiden im Bitmuster

Mit Hilfe der Funktion TRUNC(*wert*, *byte-länge*) lässt sich der hintere Teil der Mantisse beschneiden. In fast allen Fällen wird sich der Rundungsfehler auf die letzten 16 Bits beschränken. Deshalb empfiehlt es sich, den Wert vor dem Vergleich auf 6 Byte zu reduzieren, wodurch die letzten 16 Bits auf 0 gesetzt werden.

```
2   DATA test;
3       FORMAT x1 x2 32.31;
4       x1= 0.1 + 0.2;
5       x2= 0.3;
6       IF x1 = x2 THEN comp1= "Y";
7                  ELSE comp1= "N";
8       t1= TRUNC(x1,6);
9       t2= TRUNC(x2,6);
10      IF t1 = t2 THEN comp2= "Y";
11                 ELSE comp2= "N";
12      PUT x1= binary64.;
13      PUT t1= binary64.;
14      PUT x2= binary64.;
```

```
15      PUT t2= binary64.;
16      t1= TRUNC(0.299999999999,6);
17      IF t1 = t2 THEN comp3= "Y";
18                 ELSE comp3= "N";
19      PUT ">>> " comp1= comp2= comp3=;
20  RUN;
```

```
x1=0011111111010011001100110011001100110011001100110011001100110100
t1=0011111111010011001100110011001100110011001100110011000000000000000
x2=0011111111010011001100110011001100110011001100110011001100110011
t2=0011111111010011001100110011001100110011001100110011000000000000000
>>> comp1=N comp2=Y comp3=N
```

Alle Rundungsfehler die in diesem Bereich fallen, werden dadurch abgeschnitten und die Werte entsprechend angeglichen. Weiterhin werden selbst kleine Abweichungen schon als Unterschied erkannt (im Beispiel wird die 0.299999999999 nicht gleich 0.3 erkannt). Das Abschneiden des Bitmusters sollte immer der Rundung vorgezogen werden, da hier eine gewisse Dynamik inbegriffen ist.

5.3 Bitte etwas Unschärfe reinbringen

Wenn immer eine ganze Zahl erwartet wird, sollte die Funktion FUZZ(*wert*) benutzt werden. Diese Funktion rundet den Wert zu der nächsten ganzen Zahl hin, aber nur dann, wenn die Zahl nicht weiter als 10^{-12} von einer ganzen Zahl entfernt liegt.

```
2  DATA _NULL_;
3      x1= FUZZ(5.9999999999999);
4      x2= FUZZ(5.99999999);
5      PUT x1= 16.14;
6      PUT x2= 16.14;
7  RUN;
```

```
x1=6.00000000000000
x2=5.99999999000000
```

Wie wertvoll diese Funktion sein kann, sieht man an dem folgenden Beispiel.

```
2  DATA _NULL_;
3      DO i=0 TO 2 BY .1;
4          f = FUZZ(i);
5          IF i=1 THEN PUT 'i is ZERO';
6          IF f=1 THEN PUT 'f is ZERO';
7      END;
8  RUN;
```

```
f is ZERO
```

Obwohl die 1 eine ganze Zahl ist, wird sie nicht genau erreicht, da die Schrittweite bei 0.1 liegt. Der Vorteil gegenüber der TRUNC oder ROUND Funktion ist, dass nur gerundet wird, wenn der Wert nah genug an einer ganzen Zahl liegt. Alle anderen Werte werden nicht gerundet.

5.4 Kommazahlen vermeiden

Ein relativ schockierendes Beispiel aus einem SESUG Paper [3] hat mich auf eine weitere schöne Idee gebracht. Schon mal eine Milliarde Dollar mit 10 Cent Sparrate je Millisekunde zusammengespart? Nein? Sollten Sie aber. Denn wenn Ihre Bank falsch rechnet, können Sie etwa alle 116 Tage ein paar Dollar hinzuverdienen.

```
2   DATA oneBillion;
3       sum= 0;
4       DO i=1 TO 10000000000;
5         sum= sum + 0.1;
6       END;
7       diff= sum - 1000000000;
8       difft= TRUNC(sum,6) - 1000000000;
9       PUT diff= dollar12.2 difft= dollar12.2;
10  RUN;

diff=$163.12 difft=$163.12
```

Selbst das Abschneiden der letzten 16 Bits hilft in diesem Fall nicht weiter, und so lassen sich schnell mal $163.13 verdienen. Besondere Vorsicht sollten Sie aber beim Umtausch walten lassen, denn der Umrechnungsfaktor ist meistens mit vielen Kommastellen versehen.

Der beste Tipp, um diese Ungenauigkeit zu vermeiden: rechne soweit es geht mit ganzen Zahlen und vermeide Kommazahlen. Bei Geldwerten kann ein Umstieg auf Cent-Beträge die volle Genauigkeit bringen, denn ganze Zahlen können stets exakt dargestellt werden, sofern sie nicht zu groß werden (>18.446.744.073.709.551.615).

```
2   DATA oneBillion;
3       sum= 0;
4       DO i=1 TO 10000000000;
5         sum= sum + 10;
6       END;
7       diff= sum/100 - 1000000000;
8       difft= TRUNC(sum/100,6) - 1000000000;
9       PUT diff= dollar12.2 difft= dollar12.2;
10  RUN;

diff=$0.00 difft=$0.00
```

Es entstehen keine Rundungsfehler und deshalb ist die TRUNC Funktion auch völlig überflüssig.

6 Zusammenfassung

Bei genauerer Betrachtung der Darstellung für die Nachkommastellen stellt sich heraus, dass nur wenige Zahlen zwischen 0 und 1 dargestellt werden können.

2er Potenz	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}
Zahlenwert	0.5	0.25	0.125	0.0625	0.03125	0.015625	0.0078125	0.00390625

Mit dem ersten Bit kann nur die 0.5 exakt dargestellt werden. Mit dem zweiten Bit erweitert sich die Darstellung auf 2 Nachkommastellen. Damit können die zahlen 0.25, 0.50 und 0.75 exakt abgebildet werden. Im Allgemeinen erweitert sich die Anzahl möglicher Kommastellen mit jedem Bit um eine weitere Kommastelle, also 10^b-1 wobei b die vorhandenen Bits für die Nachkommastellen sind. Die exakt darstellbaren Zahlen wachsen nur mit der Basis 2 an, genauer mit 2^b-1. Von der unendlichen Menge an Zahlen, die zwischen 0 und 1 liegen, kann nur ein minimaler Bruchteil exakt dargestellt werden. Wenn die Zahl nicht auf 5 endet, ist eine exakte Darstellung nicht möglich. Es ist also wahrscheinlich, dass es zu Ungenauigkeiten kommt, sofern Kommastellen benutzt werden.

Die Rundungsfehler sind letztendlich gering und können, sofern nicht gerade ein Flug zum Mond geplant werden muss, vernachlässigt werden. Aber wenn möglich, sollten Kommastellen vermieden werden und bei Vergleichen muss mit entsprechender Vorsicht vorgegangen werden. Zumindest sollten sofort die Alarmglocken läuten, wenn eine offensichtliche Rechnung nicht aufgeht.

Literatur

[1] SAS Institute Inc. 2015: Numerical Accuracy in SAS Software - SAS® 9.4 Language Reference: Concepts, Fifth Edition. Cary, NC: SAS Institute Inc.

[2] IEEE 754: https://de.wikipedia.org/wiki/IEEE_754#Berechnung_Dezimalzahl_.E2.86.92_IEEE754-Gleitkommazahl. 15. November 2015.

[3] Imelda C. Go, Rounding in SAS®: Preventing Numeric Representation Problems, http://analytics.ncsu.edu/sesug/2008/PO-082.pdf