Paper DH08

"Do Not Trust Your Own Computer-generated Numbers"

Matthias Lehrkamp, Bayer AG, Berlin, Germany

ABSTRACT

Winston Churchill formulated the saying "Do not trust any statistics you did not fake yourself." but the full truth is you should not trust your own computer-generated numbers even if you did not want to fake them. The computer has limitations that lead to inaccuracies. In most cases these inaccuracies can be ignored; but in certain cases, they should be considered. I am not talking rocket science here: I am talking about simple statistics. This paper will show you how the computer works with numbers and why simple operations lead to errors. But be warned: once you have read this paper you will never trust a number again.

INTRODUCTION

In every number system there are numbers that cannot be represented correctly like infinite numbers or numbers that have not yet been calculated exactly (pi). Humans use the decimal number system because we have ten fingers to count things. Computer systems use the binary number system, because it represents the number system with a combination of the numbers 0 and 1. The representation and calculation of binary numbers can be easily done via switches (transistors), where 0 stands for off and 1 for on. Many floating-point numbers cannot be represented exactly in the binary number system and that leads to rounding errors. Therefore, it is not a SAS specific problem, but SAS is used in this paper because it is the most widely used software in the pharmaceutical industry. There are also some special ways in SAS to detect this kind of error. This article will explain the error in general and use SAS with its special limitations and functionalities for the presentation. At the end possible solutions are shown on the basis of examples.

THE NUMBER SYSTEM

Not all numbers can be represented exactly.

```
1/3 = 0.333333…
 pi = 3.141593…
```

In addition, computer software has to deal with hardware limitations. SAS always uses 64 bits to store a numerical value, also known as double-precision floating-point. Since the computer and the human use two different number systems, the numbers must be converted, which can lead to conversion errors. Two examples for both systems are shown below.

```
Binary      Decimal
1010.1001 = 10.5625
0101.1110 =  5.875
```

Most computers use the binary number system standard IEEE 754 for floating-point arithmetic. Another number system is the IBM hexadecimal floating-point format. The hexadecimal system is based on the binary system. Basically, the hexadecimal system reduces the possible combination of 4 bits to one character, 0-9 and A-F.

```
Binary      Hex   Decimal
1010.1001 = A.9 = 10.5625
0101.1110 = 5.E =  5.875
```

Due to some different rules defined in both systems, the results may differ slightly. This paper will focus only on the standard of the binary number system IEEE 754.

LOOKING AT THE ERROR

Even simple calculations can lead to errors, but how simple can it be. Is the following equation equal?

```
0.1 + 0.2 = 0.3
```

All school leavers will immediately confirm that the equation given is correct. However, the situation is different in the binary system.

```
DATA _NULL_;
   a= 0.1 + 0.2;
   b= 0.3;
   d= a - b;
   IF a=b THEN check= 'equal    ';
          ELSE check= 'not equal';
   PUT a= 32.31 / b= 32.31;
   PUT d= 32.31 / d= best.;
   PUT check=;
RUN;
```

Log output:

```
a=.30000000000000000000000000000000
b=.30000000000000000000000000000000
d=.00000000000000005551115123125 78
d=5.551115E-17
check=not equal
```

The binary system has a small difference (`d`) between the number `0.3` and the calculation of `0.2 + 0.1`. Even through the decimal numbers in the variables `a` and `b` look the same, the binary numbers are different. This is a wanted conversion issue in SAS because the rounding error is already known and therefore SAS rounds the number in the decimal representation even though 31 decimal places were requested for the output. One way to detect the error is to look at the numbers stored by the computer, which means the binary number. The binary numbers can be shown by using the format `binary64.` instead of `32.31`.

```
a=0011111111010011001100110011001100110011001100110011001100110100
b=0011111111010011001100110011001100110011001100110011001100110011
```

R offers the possibility to change the accuracy of the representation of the decimal numbers by using the option `options(digits=22)`. In this case, the differences can already be seen in the decimal system.

```
> 0.1 + 0.2
[1] 0.3000000000000000444089
> 0.3
[1] 0.29999999999999998889 78
```

However, this would be very confusing for humans, as the numbers displayed differ from the input. But it would be the truth and a way to find errors. Read the next section to understand why these numbers cannot be correctly represented in the binary number system.

THE BINARY SYSTEM ACCORDING TO THE STANDARD IEEE 754

This section explains how numbers are represented in the binary system following the standard IEEE 754 and why this rounding problem occurs.

The full 64 bits of a numerical value in SAS, is divided into three parts.

- 1 bit for the sign
- 11 bits for the offset
- 52 bits for the mantissa

Each bit can have a value of 0 (switch is off) or 1 (switch is on).

THE MANTISSA

The mantissa stores the absolute value of the decimal number as a combination of 52 bits. Each bit has a value (0|1) and a decimal value. The value is the multiplier for the decimal value, where the decimal value is given by a value of

2 to the power of n. The sum of all values builds the final decimal number. The decimal number 5 is equal with the binary number 101.

Decimal value	$2^2 = 4$	$2^1 = 2$	$2^0 = 1$
Bit value	1	0	1

→ $101 = 1*2^2 + 0*2^1 + 1*2^0 = 5$

Floating point numbers can be separated into an integer part and a decimal part. The following table shows how it works for the integer numbers from 1 to 8 using 4 bits.

Table 1: Integer numbers with 4 bits

2^3	2^2	2^1	2^0	
8	**4**	**2**	**1**	← value of the bit
0	0	0	1	= **1** = $0*2^3 + 0*2^2 + 0*2^1 + 1*2^0$
0	0	1	0	= **2** = $0*2^3 + 0*2^2 + 1*2^1 + 0*2^0$
0	0	1	1	= **3** = $0*2^3 + 0*2^2 + 1*2^1 + 1*2^0$
0	1	0	0	= **4** = $0*2^3 + 1*2^2 + 0*2^1 + 0*2^0$
0	1	0	1	= **5** = $0*2^3 + 1*2^2 + 0*2^1 + 1*2^0$
0	1	1	0	= **6** = $0*2^3 + 1*2^2 + 1*2^1 + 0*2^0$
0	1	1	1	= **7** = $0*2^3 + 1*2^2 + 1*2^1 + 1*2^0$
1	0	0	0	= **8** = $1*2^3 + 0*2^2 + 0*2^1 + 0*2^0$

This is how it works with integer values. There are no gaps between any two integers. That means all integers can be represented exactly in the binary system, as long as there are enough bits to represent the number.

With the decimal places the principle remains the same, only negative numbers are used as power.

Table 2: fractional numbers with 4 bits

	2^{-1}	2^{-2}	2^{-3}	2^{-4}
value of the bit →	.5	.25	.125	.0625
$0*2^{-1}+0*2^{-2}+0*2^{-3}+1*2^{-4} = 0.0625$	0	0	0	1
$0*2^{-1}+0*2^{-2}+1*2^{-3}+0*2^{-4} = 0.125$	0	0	1	0
$0*2^{-1}+0*2^{-2}+1*2^{-3}+1*2^{-4} = 0.1875$	0	0	1	1
$0*2^{-1}+1*2^{-2}+0*2^{-3}+0*2^{-4} = 0.25$	0	1	0	0
$0*2^{-1}+1*2^{-2}+0*2^{-3}+1*2^{-4} = 0.3125$	0	1	0	1
$0*2^{-1}+1*2^{-2}+1*2^{-3}+0*2^{-4} = 0.375$	0	1	1	0
$0*2^{-1}+1*2^{-2}+1*2^{-3}+1*2^{-4} = 0.4375$	0	1	1	1
$1*2^{-1}+0*2^{-2}+0*2^{-3}+0*2^{-4} = 0.5$	1	0	0	0

The bit patterns are the same as used for the integer values but now there are big gaps between the numbers. That is because of the decimal values. The decimal value from one bit to the next bit is always the half value. As the last number of 2^{-1} is 5 and the half of 5 is 2.5, two important conclusions can be made. On the one hand the value of each bit always ends with the number 5. On the other hand, there is always exactly one decimal place more per bit. This leads to the conclusion that only a few numbers can be displayed correctly in the decimal part. From all floating-point numbers with 1 decimal place, only 1 can be stored exactly.

0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9

From all floating-point numbers with 2 decimal places, only 3 can be stored exactly.

0.01	0.02	0.03	0.04	0.05	0.06	0.07	0.08	0.09	0.10
0.11	0.12	0.13	0.14	0.15	0.16	0.17	0.18	0.19	0.20
0.21	0.22	0.23	0.24	0.25	0.26	0.27	0.28	0.29	0.30
0.31	0.32	0.33	0.34	0.35	0.36	0.37	0.38	0.39	0.40
0.41	0.42	0.43	0.44	0.45	0.46	0.47	0.48	0.49	0.50
0.51	0.52	0.53	0.54	0.55	0.56	0.57	0.58	0.59	0.60
0.61	0.62	0.63	0.64	0.65	0.66	0.67	0.68	0.69	0.70
0.71	0.72	0.73	0.74	0.75	0.76	0.77	0.78	0.79	0.80
0.81	0.82	0.83	0.84	0.85	0.86	0.87	0.88	0.89	0.90
0.91	0.92	0.93	0.94	0.95	0.96	0.97	0.98	0.99	

All numbers with a white background cannot be exactly represented by the binary number system. That is an incredible amount of numbers. SAS uses 52 bits for the mantissa, which leads to small errors. Larger numbers get larger rounding errors. Most numbers have an accuracy of around 16 digits.

```
  0.1 =   0.10000000000000001
  0.2 =   0.20000000000000001
  0.3 =   0.29999999999999999
  0.4 =   0.40000000000000002
  0.5 =   0.50000000000000000
  0.6 =   0.59999999999999998
  0.7 =   0.69999999999999996
  0.8 =   0.80000000000000004
  8.8 =   8.8000000000000007
 16.8 =  16.8000000000000007
256.8 = 256.8000000000000114
```

This should be sufficient for most of the calculations. But it is important to be careful when high accuracy is required. On average, the error will remain small, but that does not always have to be the case. Especially in the area of big data or numerical approximation methods. The error can easily add up, because many numbers are included in the calculation, or numbers are used repeatedly.

THE OFFSET

The mantissa has 52 bits. Those bits are not evenly distributed between the integer part and the decimal part. Because if a part of the number is not used at all, the split would be suboptimal. The offset value gives the binary number more flexibility, so that the decimal point can be shifted. Let us consider the following example.

`10.4375 = 1010.0111`

Because all numbers start with a 1, there is no reason to store the first 1 into the mantissa. Rather, all digits after the first 1 are stored.

`1010.0111` → `010.0111`

The offset has 11 bits and specifies the number of digits from the decimal point (between 2^0 and 2^{-1}) to the position after the first 1. The decimal point has the offset value 1023. An offset of x digits to the left will change the offset to 1023 + x, an offset of x digits to the right will change the offset to 1023 - x. In this example the position after the first one is 3 digits to the left. That means the offset is 1023 + 3 = 1026.

`1026 = 10000000010`

THE SIGN BIT

The sign bit marks a number as positive=0 or as negative=1.

The full binary number for `10.4375` according to the IEEE 754 looks as follow.

```
       0|10000000010|0100111000000000000000000000000000000000000000000000000000
sign bit|  offset    |  mantissa
```

This is exactly how the numbers are stored according to the IEEE 754 standard. The next chapter goes back to the initial example and shows exactly when the rounding error occurs.

UNDERSTANDING THE ISSUE

All three numbers 0.1, 0.2 and 0.3 cannot be exactly represented in the binary number system. The IEEE 754 representation for these numbers in SAS (64 bit) is shown below.

```
0.1: 0|01111111011|1001100110011001100110011001100110011001100110011001100110011010
0.2: 0|01111111100|1001100110011001100110011001100110011001100110011001100110011010
0.3: 0|01111111101|0011001100110011001100110011001100110011001100110011001100110011
```

To add up two numbers, both numbers must have the same offset. It is specified that the largest offset is used. From two numbers always the larger number has the larger offset. Therefore, the offset of 0.1 must be adjusted to the offset of 0.2, means from 1019 to 1020. With the increasing of the offset by one, the mantissa must also be shifted one time to the right.

```
0.1: 0|01111111100|1100110011001100110011001100110011001100110011001100110011001101 0
0.2: 0|01111111100|1001100110011001100110011001100110011001100110011001100110011010
```

The offset is now the same for both numbers. The hidden 1 is visible in the mantissa for 0.1. In addition, one 0 is now outside the storage area. This is possible during the calculation process because SAS uses a memory of 80 bits for calculations. That makes 12 additional bits for the mantissa and 4 additional bits for the offset value. The calculation works similar to the written calculation in the decimal system, but there are only the numbers 0 and 1.

- Missing values are always interpreted as 0.
- `0 & 0` remains `0`
- `0 & 1` or `1 & 0` gives `1`
- `1 & 1` results in `0 and` a `1` is reserved for the next line

In the calculation only the mantissa will be considered.

```
      0.1:  0 1100110011001100110011001100110011001100110011001101 0
        +
      0.2:  1 1001100110011001100110011001100110011001100110011010
        =
0.1 + 0.2: 10 0110011001100110011001100110011001100110011001100111 0
```

The overhanging bits are shown on the left and right side, separated by a blank character. These values cannot be stored into the mantissa. Since all values behind the first 1 are stored in the mantissa, the mantissa and the offset must be adjusted for the result. The mantissa needs to be moved one time to the right and the offset must be adjusted from 1020 to 1021.

Offset: `01111111100` → `01111111101`

```
0.1 + 0.2: 1 0011001100110011001100110011001100110011001100110011001100110011
```

Two bits are hanging over on the right side. The number is rounded to reduce the binary pattern to 52 bits. In the IEEE 754 the mathematical rounding is used. In the decimal number system the number is rounded to the next even number. In the binary number system the number is rounded to the next 0. See the table below.

Table 3: Mathematical rounding for decimal and binary numbers

Decimal system	Binary system
2.49 ≈ 2.0	0 01 ≈ 00 00
2.50 ≈ 2.0	0 10 ≈ 00 00
2.51 ≈ 3.0	0 11 ≈ 01 00
3.49 ≈ 3.0	1 01 ≈ 01 00
3.50 ≈ 4.0	1 10 ≈ 10 00
3.51 ≈ 4.0	1 11 ≈ 10 00

As the 52th bit is 1 and the overhanging bits are 1 and 0, the result needs to be round up to the next binary number.

```
 0.1 + 0.2: 1 0011001100110011001100110011001100110011001100110011 10
rounded to: 1 0011001100110011001100110011001100110011001100110100
```

The comparison of the calculated 0.3 and the entered 0.3 leads to the difference, since the calculated 0.3 is rounded up.

```
0.1 + 0.2: 0|01111111100|0011001100110011001100110011001100110011001100110011
      0.3: 0|01111111100|0011001100110011001100110011001100110011001100110100
```

For the calculation itself SAS uses 80 bits, but after calculation the number needs to be reduced to 64 bits. That is the timepoint where this issue occurs as possible overhanging bits needs to be rounded. The next section shows some examples and how the error can be kept small or even prevented completely.

EXAMPLES AND SOLUTIONS

Now it is time to look into some examples and how the error can be prevented, as far as possible. The examples are ordered from the highest to the lowest priority. So, the first example is the first preventing action to avoid rounding issues. If this is not possible use the second example, and so on.

EXAMPLE 1: WHENEVER POSSIBLE USE INTEGER NUMBER

This example shows how you can get money from nothing. It is also a good example to show that small errors can add up to an error in the final result. It has already been shown that integer numbers can be stored exactly. Imagine a bank transfer of 0.10 €, again and again. The same is done with cent amounts. Instead of the 0.10 € fee, 10 cents will be charged each time. This is done until a difference of 1 cent is detected.

```
DATA cent_from_nothing;
   euro= 0;
   cent= 0;
   step= 0;
   DO WHILE( diff<1 );
      euro= euro + 0.1;
      cent= cent + 10;
      diff= euro*100 - cent;
      step= step + 1;
   END;
   PUT _ALL_;
RUN;
```

Interesting is the result that is displayed in the log. Because here a real difference is represented between the results calculated in euro and cent. After 204,089,899 steps (money transfers) the small issue adds up to 1 cent difference.

LOG: `euro=20408989.91 cent=2040898990 step=204089899 diff=1 _ERROR_=0 _N_=1`

If we could do one money transfer per millisecond then this will happen after 2 days, 8 hours and 42 minutes. At this point I would like to personally point out that this possible source of error was only used for presentation purposes and the real implementation may be punishable by law.

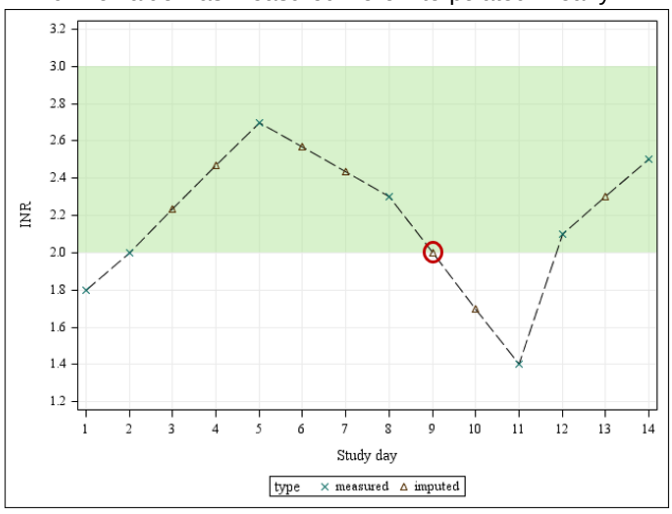
Summarized, this example is easy to use for money calculations. It could be used for other measurement values as well, but it makes not much sense if other operations lead to floating point values.

EXAMPLE 2: USE THE FUZZY FUNCTION TO COMPARE INTEGER NUMBERS

This example was from my own study a few years ago. A laboratory value called INR should be in the interval from 2 to 3, for as many days as possible. Days on which no value was measured were interpolated linearly.

VIEWTABLE: Work.Inr01_checkrange

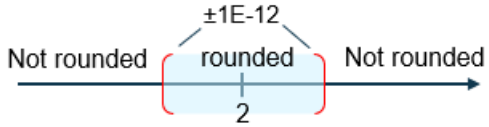
	LBDY	INR	INRANGEFL
1	1	1.8	N
2	2	2	Y
3	3	2.2667	Y
4	4	2.5333	Y
5	5	2.7	Y
6	6	2.5667	Y
7	7	2.4333	Y
8	8	2.3	Y
9	9	2	N
10	10	1.7	N
11	11	1.4	N
12	12	2.1	Y
13	13	2.3	Y
14	14	2.5	Y



In the review I noticed a difference between the table and the graph. The misinterpreted value was found quickly, but to find out what the reason was behind this issue took me a bit more time. The differences between the figure (78.57% of the time in the interval) and the table (71.43% of the time in the interval) is more than 7%. This can already cost the approval for a study. At that time, I had been able to solve the error by using the rounding function. But the following solution works even better.

```
DATA inr01_checkRange;
   SET inr;
   LENGTH INRANGEFL $1.;
   IF 2.0 <= fuzz(inr) <= 3.0 THEN inrangefl= 'Y';
                              ELSE inrangefl= 'N';
RUN;
```

The fuzzy function rounds only values close to an integer number.




```
f= fuzz(x);
x=1.9999999999988 f=1.9999999999988 not rounded
x=1.9999999999989 f=1.9999999999989 not rounded
x=1.9999999999990 f=2.0000000000000 rounded       2 - 1E-12
x=1.9999999999991 f=2.0000000000000 rounded
...
x=2.0000000000000 f=2.0000000000000 rounded
...
x=2.0000000000008 f=2.0000000000000 rounded
x=2.0000000000009 f=2.0000000000000 rounded
x=2.0000000000010 f=2.0000000000010 not rounded 2 + 1E-12
x=2.0000000000011 f=2.0000000000011 not rounded
```

If integer values are expected or a comparison to integer values must be made, it is always good to follow this example. Here it is also a good idea to convert to the next smaller unit so that you get integer numbers.

EXAMPLE 3: ROUND FLOATING-POINT NUMBERS BEFORE EVERY COMPARE WITH THE ROUND FUNCTION

Small rounding issues can lead to big differences as soon as values are compared. If a comparison is to be made between two floating point numbers, both numbers should always be rounded before the comparison. In the next example a loop goes from 0.1 to 0.5 and a message should be printed when 0.3 is reached. But the message will never occur in the log.

```
DATA _NULL_;
   DO i=0.1 TO 0.5 BY 0.1;
      IF i=0.3 THEN PUT i= '-> 0.3 reached';
               ELSE PUT i= '-> 0.3 not reached';
   END;
RUN;
```

The log shows that 0.3 is never reached.

```
i=0.1 -> 0.3 not reached
i=0.2 -> 0.3 not reached
i=0.3 -> 0.3 not reached
i=0.4 -> 0.3 not reached
i=0.5 -> 0.3 not reached
```

This is also related to small rounding issues as soon as values are compared. The rounding function should always be used for comparisons.

```
DATA _NULL_;
   DO i=0.1 TO 0.5 BY 0.1;
      IF round(i,0.01)=0.3 THEN PUT i= '-> 0.3 reached';
                           ELSE PUT i= '-> 0.3 not reached';
   END;
RUN;
```

Now the expected result is displayed in the log.

```
i=0.1 -> 0.3 not reached
i=0.2 -> 0.3 not reached
i=0.3 -> 0.3 reached
i=0.4 -> 0.3 not reached
i=0.5 -> 0.3 not reached
```

It is a little bit difficult to choose the right rounding unit. If the rounding unit is too large, unequal numbers are considered equal.

```
round(0.26,0.1) = 0.3
```

If the rounding unit is too small, binary rounding issues will not be fixed. Summarized, rounding is needed to compare floating point numbers but should always be carefully used.

EXAMPLE 4: COMPARE THE DECIMAL VALUES

This example is only suitable for the comparison of two floating point numbers. In case the dimension of the numbers is unclear or it is difficult to find the right rounding unit, why not just comparing the decimal values itself? The big advantage is that numbers will be compared as displayed or as present in the report. Below is how it works in our initial example 0.1 + 0.2 = 0.3.

```
DATA _NULL_;
   a= 0.1 + 0.2;
   b= 0.3;
   IF put(a,best.)=put(b,best.) THEN check= 'equal    ';
          ELSE check= 'not equal';
   PUT a= best. / b= best. / check=;
RUN;
```

The log now shows that we have no difference between our numbers.

```
a=0.3
b=0.3
check=equal
```

The put function converts both numbers to character. The format best. is the default format used in SAS when no format is specified. This format displays 12 characters including the dot and minus sign. To use the standard format would be the easiest way. Another way is to use the format as used in the report.

This solution does not bring reliable results, since larger rounding errors are not fixed. It is also limited to the comparisons of two numbers only. But this solution is easy to use and fixes most comparing issues.

CONCLUSION

The rounding error is irrelevant for most statistical results, as the results are rounded for the report and therefore the error is fixed. But the rounding error could add up and give false results, especially with regard to big data or numerical approximation methods, in which many calculations are necessary. Everyone should be a little more skeptical about their results and it is particularly important to take rounding errors into account when making comparisons. This article has presented some pitfalls and possible solutions. Hopefully these solutions will be used more often in the future to avoid some rounding issues.

REFERENCES

Numerical Accuracy in SAS Software, SAS Institute Inc. 2016. SAS® 9.4 Language Reference: Concepts, Sixth Edition, Cary, NC: SAS Institute Inc., 25Mar2019

Binary number, https://en.wikipedia.org/wiki/Binary_number, 15Sep2019

Decimal, https://en.wikipedia.org/wiki/Decimal, 15Sep2019

IEEE 754, https://en.wikipedia.org/wiki/IEEE_754, 15Sep2019

IBM hexadecimal floating point, https://en.wikipedia.org/wiki/IBM_hexadecimal_floating_point, 15Sep2019

Decimal-to-binary-converter, http://decimal-to-binary.com/decimal-to-binary-converter-online.html, 04Oct2019

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Matthias Lehrkamp
Bayer AG
Email: matthias.lehrkamp(at)bayer.com