PharmaSUG 2014 - Paper CC50

What do you mean 0.3 doesn't equal 0.3? Numeric Representation and Precision in SAS and Why it Matters

Paul Stutzman, Axio Research LLC, Seattle, Washington

ABSTRACT

The manner in which numeric values are stored in data sets or memory can cause SAS programs to produce seemingly surprising results if that manner is not well understood. Numeric representation and precision describes how these values are stored.

This paper shows how numeric representation and precision can affect program output and produce unintended results. It shows how numbers are actually represented, it identifies the magnitude of the difference between how values are represented and their absolute values, and it provides solutions to the problems that numeric representation and precision can introduce.

INTRODUCTION

SAS programs can produce unexpected results if numeric representation and precision are not considered. This is particularly important when computed fractional values are used for filtering, with IF, SELECT and WHERE statements, for comparisons with the COMPARE procedure or, potentially, in categorical analyses.

This paper provides an example program where numeric representation and precision create results that might seem surprising. It explains how numeric values are stored in SAS data sets and how this can introduce problems. It identifies how big these problems are and the conditions under which they are likely to manifest themselves. Finally, it describes techniques for mitigating these effects.

Note: The specifics described here are for a Windows-based SAS computing environment, but these concepts and techniques apply to all SAS computing environments.

THE PROBLEM

Consider the following pair of DATA steps:

```
data ds1;
     do A = .1 to .8 by .1;
        Index + 1;
        output;
     end;
     Index = 21;
     A = 0.000000002;
     output;
     Index = 22;
     A = 0.000000002;
     output;
run;
```

```
data ds2;
     do B = .1,.2,.3,.4,.5,.6,.7,.8;
        Index + 1;
        output;
     end;
     Index = 21;
     B = 0.000000002;
     output;
     Index = 22;
     B = 0.000000003;
     output;
run;
```

These DATA steps create variables (A and B, respectively) with values 0.1, 0.2, 0.3, etc. to 0.8, with two additional values (one pair equal; one pair unequal). The difference: A is created computationally, whereas B's values are specified explicitly. Output 1 shows the output from PRINT procedures for the resulting data sets.

```
Index          A
   1      0.100000000
   2      0.200000000
   3      0.300000000
   4      0.400000000
   5      0.500000000
   6      0.600000000
   7      0.700000000
   8      0.800000000
  21      0.000000002
  22      0.000000002
```

```
Index          B
   1      0.100000000
   2      0.200000000
   3      0.300000000
   4      0.400000000
   5      0.500000000
   6      0.600000000
   7      0.700000000
   8      0.800000000
  21      0.000000002
  22      0.000000003
```

Output 1. Output from PROC PRINTs for datasets ds1 and ds2; differences shown in red

Next, the previously created data sets (ds1 and ds2) are merged, and the variable A_eq_B is set to "Yes" if variables A and B are equal.

```
data comb1;
     merge ds1 ds2;
     by Index;
     if A = B then A_eq_B = "Yes";
run;
```

The output from a PROC PRINT of the resulting data set (Output 2) might seem surprising.

```
Index              A               B    A_eq_B

   1     0.100000000     0.100000000     Yes
   2     0.200000000     0.200000000     Yes
   3     0.300000000     0.300000000
   4     0.400000000     0.400000000     Yes
   5     0.500000000     0.500000000     Yes
   6     0.600000000     0.600000000     Yes
   7     0.700000000     0.700000000     Yes
   8     0.800000000     0.800000000
  21     0.000000002     0.000000002     Yes
  22     0.000000002     0.000000003
```

Output 2. Output from PROC PRINT of comb1

A and B are obviously not equal in the record where Index=22, however their values in records 3 and 8 sure look the same. In order to see what's going on, we can do another PROC PRINT (Output 3.). This time a FORMAT called HEX16 is used to display the values of A and B in hexadecimal notation, as they are actually stored in the data set.

```
Index                   A                   B    A_eq_B

   1    3FB999999999999A    3FB999999999999A     Yes
   2    3FC999999999999A    3FC999999999999A     Yes
   3    3FD3333333333334    3FD3333333333333
   4    3FD999999999999A    3FD999999999999A     Yes
   5    3FE0000000000000    3FE0000000000000     Yes
   6    3FE3333333333333    3FE3333333333333     Yes
   7    3FE6666666666666    3FE6666666666666     Yes
   8    3FE9999999999999    3FE999999999999A
  21    3E212E0BE826D695    3E212E0BE826D695     Yes
  22    3E212E0BE826D695    3E29C511DC3A41DF
```

Output 3. Output from PROC PRINT of comb1 using the HEX16 Format for A and B; differences shown in red

The differences in records 3 and 8 are quite small (one bit difference in the last bytes), but there they are. This can lead to unexpected results if these values are used in filtering, IF, SELECT or WHERE statements, in the output from a COMPARE procedure or, potentially, in categorical analyses. Subtracting A from B where Index=3 could result in something like "-0.00" (instead of "0.00") being displaying in a REPORT procedure's output. Consequently, it is important to understand how numeric variables are represented in SAS and what techniques are available to ensure these issues will not introduce unintended results.

NUMERIC REPRESENTATION AND PRECISOIN

The key to making sure these issues do not introduce erroneous results lies in understanding how numeric values are actually stored in data sets and in memory. This is referred to as numeric representation and precision.

As mentioned in the introduction to this paper, the specifics described here are for a Windows-based SAS computing environment, but these concepts and techniques apply to all SAS computing environments.

HOW NUMBERS ARE STORED

SAS uses eight bytes to store each numeric value. The method of representation is called floating point notation.

Floating point numbers are made up of four fundamental components: Sign, Base, Exponent and Mantissa. There is a fifth component, Bias, whose function is beyond the scope of this paper. (See, "Numeric Precision 101" in the References section below for more information on Bias.)

The four basic components provide the following functions:

- Sign: indicates whether the number is positive or negative
- Base: here the base is 2 (binary). Base is constant within each computing environment, so it is not explicitly specified in each stored numeric value.
- Exponent: how many times the base is to be multiplied (i.e. raised to the power of)
- Mantissa: the digits that define the number's magnitude (expressed as a value is between 0 and 1)

In other words: [**Sign**] * ([**Mantissa**] * [Base] $^{\text{[Exponent]}}$)

SAS divides the eight bytes that make up numbers into floating point components as shown in Figure 1.

1	2	3	…	8
SEEE EEEE	EEEE MMMM	MMMM MMMM	MMMM MMMM	MMMM MMMM

Figure 1. Floating point notation for numeric variables (Windows-based systems)

The first bit of the first byte indicates the Sign (0 = positive; 1 = negative). The remainder of the first byte and the first half of the second byte (11 bits total) make up the Exponent. The remaining half of the second byte and the entirety of bytes three through eight make up the Mantissa.

To see how the components of floating point notation work, consider the following example – first in the more familiar base 10, then in the computer's base 2 (binary). Given:

Sign = **1**, **Exponent** = **3** and **Mantissa** = **1492**

In base 10 this representation yields:

$(-1) * (0.1492 * 10^{3}) = -149.2$

In base 2 (binary) this representation yields:

$(-1) * (0.1492 * 2^{3}) = -1.1936$

WHERE PROBLEMS CAN ARISE

Integers can be represented exactly with this method in almost all practical applications (up to 9×10^{15}, or 9 quadrillion). Problems can arise with fractional values, however.

Some non-integer values simply cannot be represented exactly. For example, the decimal number 0.1 is stored as 3FB999999999999A, as shown in Output 3 above. However, the exact representation of 0.1 would be:

0.1 = 3FB99999999999999999999999… with the 9s going on to infinity (this can be expressed as $3FB\overline{9}$)

This issue is not unique to binary (base 2) systems. It is an issue for base 10 and other bases as well. In base 10, for example, two thirds is represented as:

⅔ = 0.66666666666666666666666666… with the 6s going on to infinity (or $0.\overline{6}$)

When faced with this issue, there are two choices:

The value can be truncated at the last digit.	⅔ = 0.666666	(for our base 10 example)
The value can be rounded.	⅔ = 0.666667	(for our base 10 example)

Neither of these solutions is perfect.

Truncation: ⅔ + ⅔ + ⅔ = 0.666666 + 0.666666 + 0.666666 = 1.999998

Rounding: ⅔ + ⅔ + ⅔ = 0.666667 + 0.666667 + 0.666667 = 2.000001

And, imprecision can grow as these values are used repeatedly in calculations.

THE MAGNITUTDE OF THE PROBLEM

Fortunately, the differences in values that can be introduced by these representation and precision issues are usually extremely small. For example, the differences between the two values that represent decimal 0.3 in the example above is 5.55×10^{-17}. Or:

0.0000000000000000555

Differences of this size are almost always inconsequential, particularly when SAS procedures are used to calculate various statistics. There are certain circumstances, however, under which these tiny variances become extremely important. The following are descriptions of some of these circumstances.

- IF, SELECT and WHERE statements might produce unexpected results when calculated fractional values are compared to other values that are explicitly specified or calculated differently.
- Unexpected records might be included (or excluded) when sub-setting data sets for DATA steps or procedures based on calculated fractional values. This is basically a variant of the previous point, but it can be particularly problematic and hard to diagnose.
- Comparisons of SAS data sets using PROC COMPARE might indicate differences in variables' values, even though the values look identical.
- Categorical analyses of variables, where the categories are based on computed fractional values, might be problematic. Values might not always end up in the same categories, depending on how computations are performed.
- Calculated fractional values might display oddly (ex. "-0.00") in PROC REPORT and other procedures' output.

It is helpful to remember that these issues do NOT come into play with integers; integers can be represented exactly for values up to 9 quadrillion. Only fractional values are subject to potential issues.

SOLUTIONS THAT ENSURE EXPECTED RESULTS

The techniques required to address these potential issues are fairly straightforward, once these issues are known. The most common techniques are:

- Making use of the ROUND function
- Creating character versions of numeric variables
- Taking advantage of options in procedures that address these issues

These techniques apply in different situations, as discussed below.

THE ROUND FUNCTION

The ROUND function is, perhaps, the most powerful and universally applicable tool for dealing with issues related to numeric representation and precision. When the number of significant digits is known, the ROUND function ensures that computed fractional numeric values are represented consistently, and as closely as possible to their exact values. If the variable A is expected to be accurate to the hundredths decimal place, the following statement makes sure that happens as accurately as possible:

```
A = round(A, 0.01);
```

The ROUND function is also useful in cases where the level of precision is not known in advance. This is accomplished by picking a rounding factor that is smaller than any value that is likely to be encountered. For example:

```
A = round(A, 1E-12);
```

In this case, the variable A is round to 10^{-12}. That is almost certainly smaller than any meaningful value of A.

CHARACTER VERSIONS OF NUMERIC VARIABLES

In instances where fractional numeric values are used for categorical analyses or in displays, it is sometimes useful to convert a numeric variable into a character variable. This process takes advantage of PUT statements and formats to make character values that are literally what-you-see-is-what-you-get. If the variable A is expected to be accurate to the hundredths decimal place, the following statement creates a character version of A with two decimal places:

```
CharA = put(A, 6.2);
```

If A contained the numeric representation of the decimal number 123.45, CharA would contain the characters "1", "2", "3", ".", "4" and "5". There is no room for ambiguity. The trade off is: CharA is not computationally numeric.

The BEST format can be employed in instances where the level of precision is not precisely known. The statement below creates a character version of the variable A, based on the data itself.

```
CharA = put(A, best.);
```

Character versions of numeric variables can be used in categorical analyses, PROC REPORT output or other displays, titles, footnotes, labels, etc.

The effectiveness of character conversions and rounding can be demonstrated by adding a few statements to the final DATA step of the example provided in The Problem section above. First, character versions of variables A and B are created along with a comparison variable (ChA_eq_ChB).

```
ChA = put(A, best.);
ChB = put(B, best.);
if ChA = ChB then ChA_eq_ChB = "Yes";
```

The BEST format is used in the PUT statements, because even though the values of A and B are consistently significant to the tenths place in the first eight records, A and B are significant to nine decimal places in the last two records.

Next, numeric versions of A and B are created using the ROUND function along with a comparison variable (RndA_eq_RndB).

```
RndA = round(A, 1E-12);
RndB = round(B, 1E-12);
if RndA = RndB then RndA_eq_RndB = "Yes";
```

The output from a PROC PRINT of the resulting data set is shown in Output 4.

```
                                                 ChA_                  RndA_eq_
Index       A             B         A_eq_B  ChA    ChB   eq_ChB  RndA   RndB     RndB

   1   0.100000000  0.100000000   Yes      0.1    0.1   Yes      0.1    0.1     Yes
   2   0.200000000  0.200000000   Yes      0.2    0.2   Yes      0.2    0.2     Yes
   3   0.300000000  0.300000000            0.3    0.3   Yes      0.3    0.3     Yes
   4   0.400000000  0.400000000   Yes      0.4    0.4   Yes      0.4    0.4     Yes
   5   0.500000000  0.500000000   Yes      0.5    0.5   Yes      0.5    0.5     Yes
   6   0.600000000  0.600000000   Yes      0.6    0.6   Yes      0.6    0.6     Yes
   7   0.700000000  0.700000000   Yes      0.7    0.7   Yes      0.7    0.7     Yes
   8   0.800000000  0.800000000            0.8    0.8   Yes      0.8    0.8     Yes
  21   0.000000002  0.000000002   Yes     2E-9   2E-9   Yes      0.0    0.0     Yes
  22   0.000000002  0.000000003           2E-9   3E-9            0.0    0.0
```

Output 4. Output from PROC PRINT of the new comb1; differences shown in red

Both the new character variable and the rounded variable pairs now compare equally in all records, except for the final record, as expected. RndA and RndB appear equal in the last record. This is due to the default format that was applied by PROC PRINT. The underlying values remain different, as they should, and as evidenced by RndA_eq_RndB not equaling "Yes".

OPTIONS IN PROCEDURES

A few SAS procedures include options for dealing with values that are almost, but not exactly equal. One of the most useful of these is the CRITERION option of the COMPARE procedure. While the particulars vary based on the METHOD option selected, the CRITERION option facilitates the specification of a threshold below which, values are considered equal (or, close enough). Consider the following:

```
proc compare base=BaseDS compare=CompDS criterion=0.0000000001;
```

CRITERION=0.0000000001(in conjunction with PROC COMPARE's default METHOD) establishes a threshold, below which, values are considered identical for the purposes of this PROC COMPARE step.

CONCLUSION

SAS' numeric representation and precision (i.e. the manner in which SAS stores numeric data) is an important concept to understand in order to avoid seemingly unexpected results from SAS programs. This is especially relevant when computed non-integer values are used for filtering, in IF, WHERE or SELECT statements, in comparisons, or in categorical analyses. Fortunately, tools and techniques, such as those described above, are available to ensure SAS programs perform as desired.

REFERENCES

- www.sas.com. SAS Institute. "SAS® 9.3 Language Reference: Concepts, Second Edition: Numeric Precision in SAS Software". Available at: https://support.sas.com/documentation/cdl/en/lrcon/65287/HTML/default/viewer.htm#p0ji1unv6thm0dn1gp4t01a1u0g6.htm
- www.sas.com. SAS Institute. "SAS® 9.3 Procedures Guide, Second Edition: Concepts: COMPARE Procedure". Available at: http://support.sas.com/documentation/cdl/en/proc/65145/HTML/default/viewer.htm#n04714wiqe78lln1dib6b2h0efoq.htm#n1qxqymug4990kn19wjpm7y98dah
- www.sas.com. SAS Institute. "Numeric Precision 101". Available at: http://support.sas.com/techsup/technote/ts654.pdf
- www.sas.com. SAS Institute. "Dealing with Numeric Representation Error in SAS® Applications". Available at: http://support.sas.com/techsup/technote/ts230.pdf

CONTACT INFORMATION

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

Paul Stutzman
Infrastructure Lead/Senior Statistical Programmer
Axio Research, LLC
2601 4th Avenue, Suite 200
Seattle, WA 98121
Work Phone: (206) 577-0233
E-mail: pauls@axioresearch.com