Introduction to numeric precision and representation issues: why 4.8 minus 4.6 is not always equal to 0.2

Nicola Tambascia

Accovion GmbH, Eschborn, Germany

Numeric precision and representation issues are well-known topics in computer science. Non computer scientists sometimes are not aware of the problems that occur with these issues. This paper provides a brief introduction to problems based on numeric precision and the presentation of numbers on computerized systems. Numbers are stored as binary numbers in computerized systems. However, not all floating-point numbers can be represented properly in the binary system. When calculating with numbers that cannot be stored exactly, the result may not be as expected. In our case, 4.8 and 4.6 can not be stored exactly and therefore, the result of 4.8 minus 4.6 will not be exactly 0.2. The conversion of floating-point numbers to a storable IEEE-754 binary number is illustrated. Finally, some SAS functions and options are introduced that may help to deal with the issue.

Keywords: Numeric precision, SAS, Floating-point numbers, Conversion to binary numbers, IEEE-754

Introduction

As illustrated in Fig. 1, the value '0.200' in dataset WORK.A is not equal to the same value in dataset WORK.B. At this point, it is important to know that this issue is SAS-specific but rather a general issue in computer science.

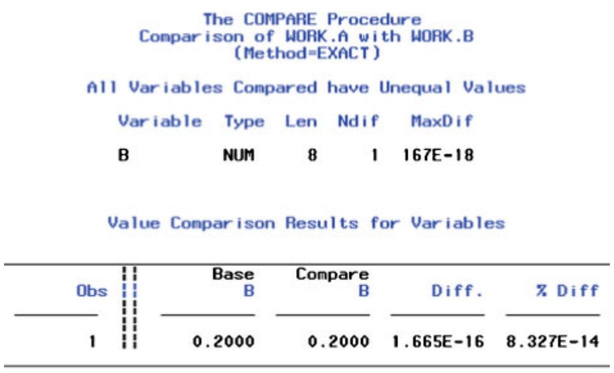
The problem is that decimal numbers are stored as binary numbers in computerized systems and thus have to be converted for storing. Often decimal numbers can only be represented with infinite binary numbers, but computerized systems are not able to store infinite numbers, which leads to loss of precision. When calculating with imprecise values, the result must not necessarily be correct. This can cause problems when comparing values.

Of course, we expect that a computerized system can handle a simple calculation like 4.8–4.6 and so when we create a dataset in SAS containing the result of this calculation, it seems to be correct (Fig. 2).

Now to be sure that the result is correct the value stored in dataset 'example1' will be compared with the actual value of 0.2.

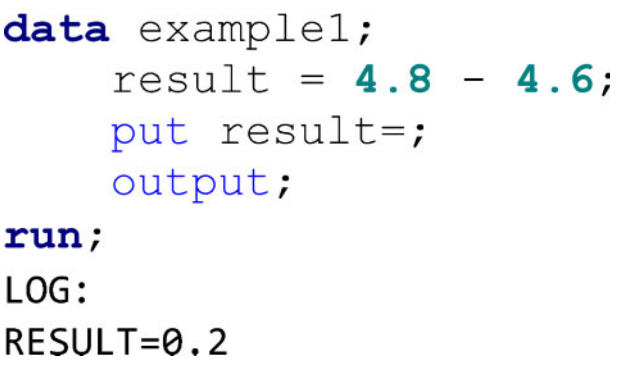
The comparison (Fig. 3) showed that the result of the calculation 4.8–4.6 is not equal to 0.2 on the system used for this test, even though SAS displays us the correct values in the dataset. It is important to note that SAS uses formatted or rounded values for representation.

To understand why a simple calculation like this goes wrong, it is important to understand how



```
                The COMPARE Procedure
           Comparison of WORK.A with WORK.B
                   (Method=EXACT)

        All Variables Compared have Unequal Values

        Variable  Type  Len  Ndif   MaxDif

        B         NUM     8     1  167E-18


        Value Comparison Results for Variables

       ||      Base     Compare
   Obs ||         B           B        Diff.      % Diff
 ______||  ________    ________     ________    ________
       ||
     1 ||    0.2000      0.2000    1.665E-16   8.327E-14
```

Figure 1 PROC COMPARE output



```
data example1;
    result = 4.8 - 4.6;
    put result=;
    output;
run;
LOG:
RESULT=0.2
```

Figure 2 Calculation of 4.8–4.6

Correspondecne to: Nicola Tambascia, Accovion GmbH, Helfmann-Park 10, Eschborn 65760, Germany. Email: nicola.tambascia@accovion.com

Published by Maney
DOI 10.1179/175709311X13166801334398

floating-point numbers are stored on computerized systems.

How floating-point numbers are stored

SAS offers the functionality to display the binary and hexadecimal values of numbers. For better readability, the hexadecimal values of our example are displayed (Fig. 4).

The output shows that the values are 'almost' equal. They have differences in the last two digits of the hexadecimal values. This differences result from the calculation. The result of the subtraction is not exact because 4.8 and 4.6 could not be stored exactly.

The following sections illustrate how the numbers are stored in computerized systems. Only a general overview on how numbers are stored, is given since there are a lot of factors (e.g. the hardware used, the operating system, the programming language used, etc.) that impact the precision of a system.

The IEEE Standard for Floating-Point Arithmetic (IEEE-754) is the most widely used standard for floating-point computation so that the following explanation will be based on this standard as far as possible.

Conversion to the binary system

In our daily life, we usually use the decimal system. In this system, the numbers are represented in digits from 0 to 9 and the set value is represented to the power of the base 10.

As example, we will use the number 225.75, representing a coefficient to the power of 10 (Fig. 5).

In computerized systems, numbers are stored as binary numbers. The base for the binary system is 2 and numbers are represented in the digits 0 and 1.

The number 225.75 from the previous example converted to binary is shown in Fig. 6.

But how does a computerized system get from the decimal system to the binary system?

To convert the number 225.75 from the decimal system to the binary system, the integer part and the decimal part are considered separately.

```
data example2;
    set example1;

    if result eq 0.2 then equal = "Y";
    else equal = "N";

    put equal=;
run;

LOG:
EQUAL=N
```

Figure 3 Comparison of the result with 0.2

```
data example3;
    set example2;

    resulthex = put(result,hex16.);
    zeropointtwo = put(0.2,hex16.);

    put resulthex=;
    put zeropointtwo=;
run;

LOG:
RESULTHEX    = 3FC99999999999A0
ZEROPOINTTWO = 3FC999999999999A
```

Figure 4 Hexadecimal output

Firstly, the integer part of the number, '225', is converted into the binary system.

To do this you have to take the following steps:

1. divide the number by the base (2) of the target number system. The remainder will represent a binary digit in the binary number. Use the integer part of the result for calculations in the following steps;
2. if the result of the division in step 1 is zero, then you are finished;
3. if not, use the integer from step 1 as the next number to calculate with and repeat steps 1 and 2 until the integer part of the result equals zero;
4. the first calculated binary digit represents the last digit in the binary number.

225 : 2=112	Remainder: 1	(Binary digit: 2^0)
112 : 2=56	Remainder: 0	(Binary digit: 2^1)
56 : 2=28	Remainder: 0	(Binary digit: 2^2)
28 : 2=14	Remainder: 0	(Binary digit: 2^3)
14 : 2=7	Remainder: 0	(Binary digit: 2^4)
7 : 2=3	Remainder: 1	(Binary digit: 2^5)
3 : 2=1	Remainder: 1	(Binary digit: 2^6)
1 : 2=0	Remainder: 1	(Binary digit: 2^7)

Result:	1	1	1	0	0	0	0	1
Binary digit:	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

Next, the decimal part of the number, '0.75', is converted into the binary system.

To do this you have to take the following steps:

1. multiply the number by the base (2) of the target number system. The number before the decimal point will be the binary digit (starting from left to right);

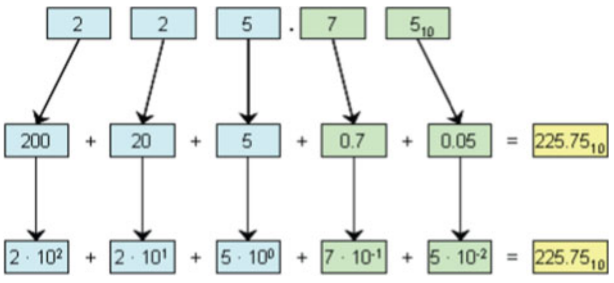


Figure 5 The number 225.75 in digits and in the set value to the power of the base 10

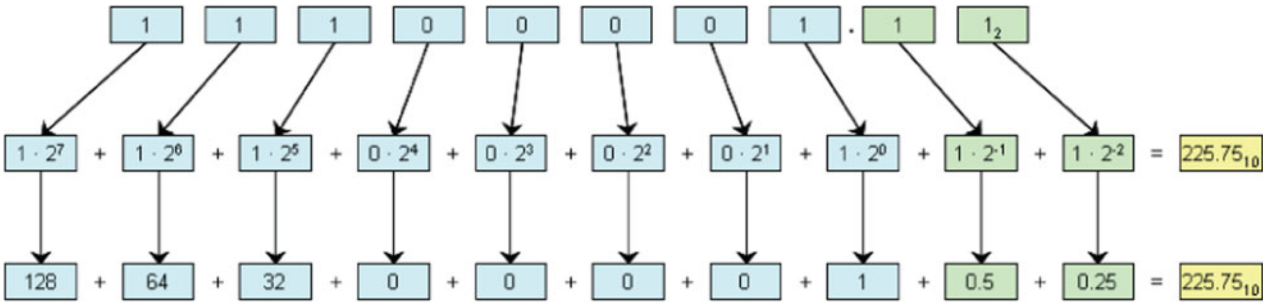


Figure 6 The number 225.75 as binary number

2. the number after the decimal point will be the new number to calculate with;
3. repeat steps 1 and 2 until you reach a number without a decimal point, a result that has already been produced, or the required precision.

2	·	0.75=1.5	digit: 1
2	·	0.5=1	digit: 1

Result:	0	·	1	1
Binary digit:	2^0	·	2^{-1}	2^{-2}

Together with the previous converted integer part, you will get complete binary number:

1	**1**	**1**	**0**	**0**	**0**	**0**	**1**	·	**1**	**1**
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	·	2^{-1}	2^{-2}

Normalized floating-point representation

Since computerized systems do not store decimal points the binary number ($11100001.11_2=225.75_{10}$) cannot be stored like this. That is why numbers have to be converted to the normalized floating-point representation form.

Normally floating-point numbers consist of the integer part and the fractional part separated by a decimal point. The normalized floating-point representation form consists of the mantissa, the base of the number system used, and an integer exponent. These three are illustrated in Fig. 7.

The term 'normalized' refers to the mantissa. It is always <1 but $\geqslant 0.1$. Thus the only correct normalized representation form for 225.75 is 0.22575×10^3 and not 0.022575×10^4 or 2.2575×10^2.

Example: $225.75_{10}=0.22575_{10}\times10^3$

To bring the binary number from the previous examples into the normalized floating-point representation form, we need to move the decimal point to the left and use the number of digits we moved the point as exponent to the basis.

$11100001.11_2=0.1110000111_2\times2^8$

Floating point numbers are basically stored like this in computerized systems. There are, however, several limitations. For example, due to the limited number of digits that can be stored, numbers in computerized systems are limited to the value range of the exponent and the mantissa (for further explanations, see the following sections).

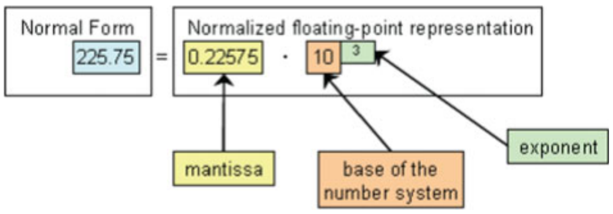

Normal Form
225.75
=
Normalized floating-point representation
0.22575
·
10
3
mantissa
base of the number system
exponent

Figure 7 Normalized floating-point representation

Floating-point Numbers According to IEEE-754 Coding

The following explanation will give only a short overview to IEEE-754 coding.

According to IEEE-754, numbers are stored as shown in Fig. 8. Depending on the precision you are working with, you can store larger exponents and larger mantissas. The following steps are necessary to create a storable binary number according to IEEE-754.

1-plus-form

The so-called 1-plus-form is generated by shifting the mantissa of the normalized floating-point representation form to the left by one digit. For our example this would look like:

$0.1110000111_2\rightarrow1.110000111_2$

The first digit must not be saved since it is always 1 due to the 1-plus-form. This fact is the advantage of the 1-plus-form in comparison to the normalized floating-point representation. One bit less is used to store the number, which means that you have one additional bit available for storage. The mantissa to be saved will be: 110000111_2. In comparison to the normalized form, the exponent is also different because the decimal point is moved one digit less in the 1-plus-form. In the 1-plus-form, the exponent for this number is 7, while it is 8 in the normalized floating-point representation.

Biased exponent

The biased exponent makes it possible to store both positive and negative exponents without the need for an additional sign bit. Numbers smaller than the bias

Sign of the mantissa	Biased exponent	Mantissa in the 1-plus-Form	Total number of bits
1 Bit 0 for positive 1 for negative	Single precision: 8 Bits Double precision: 11 Bits Quadruple precision: 15 Bits	23 Bits 52 Bits 112 Bits	32 Bits 64 Bits 128 Bits

Figure 8 Layout of floating-point numbers according to IEEE-754.

Sign of the mantissa	Biased exponent	Mantissa in the 1-plus-Form
0	1 0 0 0 0 0 0 0 1 1 0	1100001110000000000000000000000000000000000000000000
1 Bit 0 for positive 1 for negative	Double precision: 11 Bits	52 Bits

Figure 9 The number 225.75 as storable binary number according to IEEE-754

are negative exponents and numbers greater than the bias are positive exponents.

The biased exponent is calculated as follows:

Bias + original exponent (from 1-plus-form) = biased exponent

The bias for the different number formats is:

- single precision: 127;
- double precision: 1023;
- quadruple precision: 16 383.

For our example of a double precision number, this would mean:

$1023_{10}+7_{10}=1030_{10}=10000000110_2$

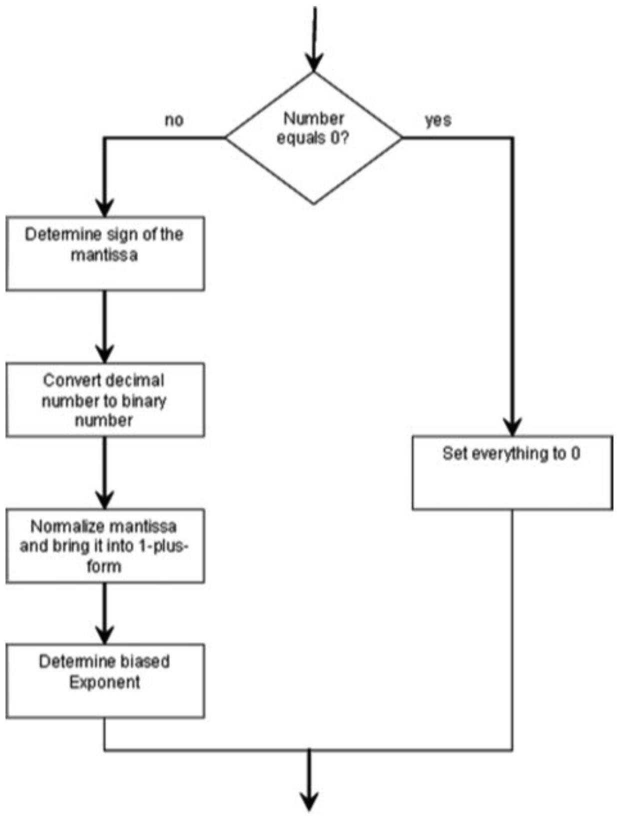
Finalize

Now all the components are available to store the number. The number 225.75 as a double precision binary number would be stored as shown in Fig. 9. In case that a mantissa does not need all the bits available, it will be filled with zeros at the end.

When storing negative numbers, the process is equal to positive number. The sign bit of the mantissa is set to 1 and the stored mantissa and biased exponent are equal to the positive number.

So the conversion of a decimal number to a storable binary number for computerized systems is processed according to the flowchart shown in Fig. 10.

The conversion from binary back to decimal is done by logically following the flowchart backwards.



Number equals 0?
no
yes
Detemine sign of the mantissa
Convert decimal number to binary number
Normalize mantissa and bring it into 1-plus-form
Detemine biased Exponent
Set everything to 0

Figure 10 The number 225.75 in digits and in the set value to the power of the base 10

Why 4.8–4.6 Is Not Always Equal to 0.2

After this example illustrating how numbers are converted and stored in computerized systems, it is still not clear why 4.8–4.6 is not equal to 0.2. In the following, the numbers 4.8 and 4.6 are converted to see what happens during the storing process.

The sign bit will be 0 for both numbers.

The integer part 4 is converted into the binary system (for both numbers):

4 : 2=2	Remainder: 0	(Binary digit: 2^0)
2 : 2=1	Remainder: 0	(Binary digit: 2^1)
1 : 2=0	Remainder: 1	(Binary digit: 2^2)

Result:	1	0	0
Binary digit:	2^2	2^1	2^0

The decimal part 0.8 is converted into the binary system:

2	·	0.8=1.6	digit: 1
2	·	0.6=1.2	digit: 1
2	·	0.2=0.4	digit: 0
2	·	0.4=0.8	digit: 0
2	·	0.8=1.6	digit: 1→the result 1.6 already appeared once and the calculation will repeat periodically until all available digits are used

Result:	0	·	1	1	0	0	…
Binary digit:	2^0	·	2^{-1}	2^{-2}	2^{-3}	2^{-4}	…

Decimal value:	4	2	1	.	1/2	1/4	1/8	1/16	1/32	1/64	1/128	1/256	…	…	…
Set value to the power of 2:	2^2	2^1	2^0		2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}	…	…	…
Binary digits:	1	0	0	.	1	1	0	0	1	1	0	0	1	1	…

Figure 11 Number line of the binary system.

Sign of the mantissa	Biased exponent	Mantissa in the 1-plus-Form
0	1 0 0 0 0 0 0 0 0 0 1	0011001100110011001100110011001100110011001100110011
1 Bit 0 for positive 1 for negative	Double precision: 11 Bits	52 Bits

Figure 12 The number 4.8 as storable binary number according to IEEE-754

The result 1.6 already appeared in the first step. From this point, the digits will repeat periodically. This is the root of the problem: numbers are stored with a finite length in computerized systems, but we would need an infinite length to represent the decimal number 4.8 as a binary number.

Figure 11 shows that the further you go to the right of decimal point the smaller the number will get.

Only fractions with a power of two in the denominator are finite in the binary system. All other fractions are infinite.

Since computerized systems do not save infinite numbers, the value gets truncated which means a loss of precision. In the end, the number is not stored as 4.8; it is stored as nearly as possible to 4.8. In double precision systems, the mantissa gets truncated after the fifty-second digit. Be aware that the mantissa according to the 1-plus-form also includes the values before the decimal point. That means that there are less than 52 bits available for the fractional. For 4.8, this means:

100.11001100110011001100110011001…=
1.00110011001100110011001…$\times 2^2$

The biased exponent is:
$1023+2=1025_{10}=10000000001_2$

When composing everything to a double precision number, the result is shown in Fig. 12.

The actual stored number is 4.7999997… and not 4.8.

The number 4.6 is stored as follows:

The sign bit is 0. The integer part has already been converted to binary (see above).

The decimal part 0.6 is converted into the binary system:

2	·	0.6=1.2	digit: 1
2	·	0.2=0.4	digit: 0
2	·	0.4=0.8	digit: 0
2	·	0.8=1.6	digit: 1
2	·	0.6=1.2	digit: 1$\rightarrow$the result 1.2 already appeared once

Result:	0	·	1	0	0	1	…
Binary digit:	2^0	·	2^{-1}	2^{-2}	2^{-3}	2^{-4}	…

Again the number is infinite in the binary system.

Double precision numbers have 52 bits for the mantissa in the 1-plus-form. So the 1-plus-form for the number 4.6 looks as follows:

100.1001100110011001100110011…=
1. 0010011001100110011001100110011…$\times 2^2$

The biased exponent is:
$1023+2=1025_{10}=10000000001_2$

When we compose everything to a double precision number, the result is as shown in Fig. 13.

The actual stored number is 4.5999999… and not 4.6.

SAS displays these values as 4.8 and 4.6 because it uses rounding according to IEEE-754. This is one of the reasons why programmers often are not aware of this problem. However, when calculating with these values, SAS and other systems use the binary numbers without rounding them. The system is therefore actually calculating ‘almost’ 4.8 minus ‘almost’ 4.6 and therefore, the result will not be

Sign of the mantissa	Biased exponent	Mantissa in the 1-plus-Form
0	1 0 0 0 0 0 0 0 0 0 1	0010011001100110011001100110011001100110011001100110
1 Bit 0 for positive 1 for negative	Double precision: 11 Bits	52 Bits

Figure 13 The number 4.6 as storable binary number according to IEEE-754

Sign of the mantissa	Biased exponent	Mantissa in the 1-plus-Form
0	0 1 1 1 1 1 1 1 1 0 0	1001100110011001100110011001100110011001100110100000
1 Bit 0 for positive 1 for negative	Double precision: 11 Bits	52 Bits

Figure 14 Binary result of the calculation of 4.8–4.6

Sign of the mantissa	Biased exponent	Mantissa in the 1-plus-Form
0	0 1 1 1 1 1 1 1 1 0 0	1001100110011001100110011001100110011001100110011010
1 Bit 0 for positive 1 for negative	Double precision: 11 Bits	52 Bits

Figure 15 The 0.2 as storable binary number according to IEEE-754

```
data example4;
    if 4.8 - 4.6 eq 0.2 then equal = "Y";
    else equal = "N";

    put equal=;
run;

LOG:
EQUAL=N
```

Figure 16 Comparison of the result with 0.2

exactly 0.2. There will be a loss of precision when calculating with this kind of numbers.

The result of the subtraction above stored as binary is shown in Fig. 14.

While the actual stored value for 0.2 (the number as it is, not as a result of a calculation) is shown in Fig. 15.

Thus, when comparing the mantissa, it becomes apparent that the last 6 bits are not equal.

Mantissa of the result of calculation: 1001100-110011001100110011001100110011001100110100000

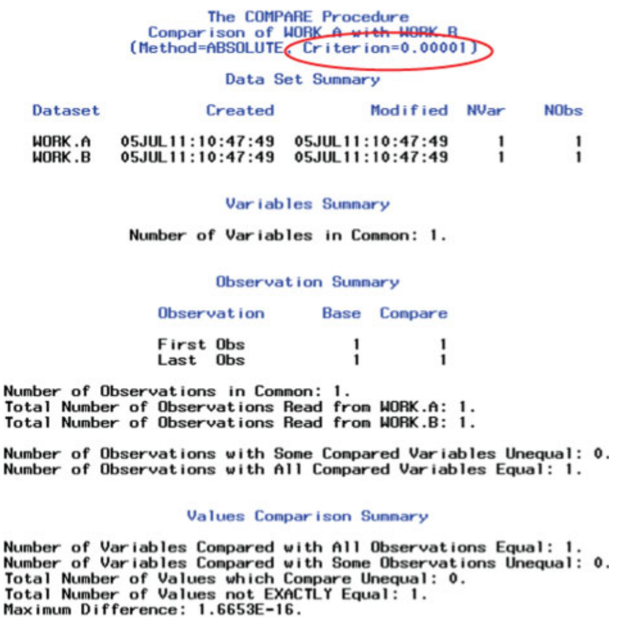
Mantissa of the actual value 0.2: 1001100-110011001100110011001100110011001100110011010

This is the reason why, in the output of the following SAS code (Fig. 16), the numbers are not equal.

It may happen that this example is equal on your system. This may be because your system is more precise. It may use more bits to store the mantissa and the exponent or it may round the values by default before comparing or representing them. The precision of a system depends on multiple factors like the hardware, the operating system, or the programming language used.

```
data example5;
    if round(4.8 - 4.6, 0.1) eq 0.2 then equal = "Y";
    else equal = "N";

    put equal=;
run;

LOG:
EQUAL=Y
```

Figure 17 Comparison of the result with 0.2 using the round function



```
                    The COMPARE Procedure
              Comparison of WORK.A with WORK.B
           (Method=ABSOLUTE, Criterion=0.00001)

                      Data Set Summary

Dataset           Created            Modified  NVar    NObs

WORK.A    05JUL11:10:47:49  05JUL11:10:47:49     1       1
WORK.B    05JUL11:10:47:49  05JUL11:10:47:49     1       1


                      Variables Summary

           Number of Variables in Common: 1.


                    Observation Summary

                Observation      Base  Compare

                First Obs           1        1
                Last  Obs           1        1

Number of Observations in Common: 1.
Total Number of Observations Read from WORK.A: 1.
Total Number of Observations Read from WORK.B: 1.

Number of Observations with Some Compared Variables Unequal: 0.
Number of Observations with All Compared Variables Equal: 1.


                  Values Comparison Summary

Number of Variables Compared with All Observations Equal: 1.
Number of Variables Compared with Some Observations Unequal: 0.
Total Number of Values which Compare Unequal: 0.
Total Number of Values not EXACTLY Equal: 1.
Maximum Difference: 1.6653E-16.
```

Figure 18 PROC COMPARE output with the option ‘method=ABSOLUTE’

Conclusion

It is possible to avoid this issue by simply using the round function whenever possible. The use of the round function in our SAS example is shown in Fig. 17.

There are a lot of other helpful functions that may help to handle the problem. The following table shows a list of these functions.

Table 1 Helpful functions

Function	Syntax	Description
CEIL	CEIL (argument)	The CEIL function returns the smallest integer that is greater than or equal to the argument
CEILZ	CEILZ (argument)	The CEIL function returns the smallest integer that is greater than or equal to the argument, using zero fuzzing
FLOOR	FLOOR (argument)	The FLOOR function takes one numeric argument and returns the largest integer that is less than or equal to the argument
FLOORZ	FLOORZ (argument)	The FLOOR function takes one numeric argument and returns the largest integer that is less than or equal to the argument, using zero fuzzing
FUZZ	FUZZ (argument)	Returns the nearest integer if the argument is within 1E-12.
INT	INT (argument)	Returns the integer value
INTZ	INTZ (argument)	Returns the integer value, using zero fuzzing
MOD	MOD (*argument-1, argument-2*)	Returns the remainder from the division of the first argument by the second argument
MODZ	MODZ (*argument-1, argument-2*)	Returns the remainder from the division of the first argument by the second argument, using zero fuzzing
ROUND	ROUND (*argument <,rounding-unit>*)	Rounds the first argument to the nearest multiple of the second argument
ROUNDE	ROUNDE (*argument <,rounding-unit>*)	Rounds the first argument to the nearest multiple of the second argument, and returns an even multiple when the first argument is halfway between the two nearest multiples
ROUNDZ	ROUNDZ (*argument <,rounding-unit>*)	Rounds the first argument to the nearest multiple of the second argument, with zero fuzzing

By default the option 'method=EXACT' is used and values are compared exactly in PROC COMPARE.

The option 'method=ABSOLUTE' is helpful when comparing datasets from different systems with PROC COMPARE. This option specifies the method for judging the equality of numeric values. When using this method, values are unequal if ABS($y-x$)>CRITERION. This means that the two compared values are subtracted and if the magnitude is greater than the specified value in the 'CRITERION=' option, these values are unequal. 'CRITERION=' is an option of PROC COMPARE and specifies the border of equality for the option 'method=ABSOLUTE'. By the default the option 'CRITERION=' has the value 0.00001, as marked in Fig. 18.

The output for the example in the introduction would then look like in Fig. 18.

The last three lines of the PROC COMPARE output show that there are no unequal values. However, PROC COMPARE also shows that there are values that were not exactly equal and that the maximum difference is to be found 15 digits after decimal point.

Numeric precision and representation issues are well known and often forgotten (especially while programming). These issues do not only occur in SAS.

Numeric precision problems should be taken into account during programming to avoid lengthy searches for errors.

References

1 Numeric Precision 101 [document on the Internet]. Cary, NC: SAS Institute Inc. [cited 2011 Jul 22]. Available from: http://support.sas.com/techsup/technote/ts654.pdf.
2 Dealing with numeric representation error in SAS applications [document on the Internet]. Cary, NC: SAS Institute Inc. [cited 2011 Jul 22]. Available from: http://support.sas.com/techsup/technote/ts230.html.
3 IEEE 754-2008 [document on the Internet] [cited 2011 Jul 22]. Available from: http://en.wikipedia.org/wiki/IEEE_floating-point_standard.
4 Kruker G. Zahlendarstellung in Rechnern (in German) [document on the Internet]. Bern: Hochschule für Technik und Archtektur Bern [cited 2011 Jul 22]. Available from: http://www.krucker.ch/Skripten-Uebungen/IAMSkript/IAMKap2.pdf.
5 Go I. Rounding in SAS®: preventing numeric representation problems. In: SESUG 2008: Proceedings of the South Eastern SAS User Group; 2008 Oct 19–22; St Pete Beach, FL, USA: SESUG [cited 2011 Jul 22]. Available from: http://analytics.ncsu.edu/sesug/2008/PO-082.pdf.
6 Gorrell P. Numeric length in SAS®: a case study in decision making. Pharm Program. 2008;**1**:56–64.