SESUG Proceedings (c) SESUG, Inc (http://www.sesug.org) The papers contained in the SESUG proceedings are the property of their authors, unless otherwise stated. Do not reprint without permission.
SESUG papers are distributed freely as a courtesy of the Institute for Advanced Analytics (http://analytics.ncsu.edu).

PO–082

Rounding in SAS®: Preventing Numeric Representation Problems

Imelda C. Go, South Carolina Department of Education, Columbia, SC

ABSTRACT

As SAS programmers, we come from a variety of backgrounds. We range from having little or no formal computer science background to having academic degrees in computer science. Numeric precision and representation are standard topics in computer science education. Programmers eventually encounter numeric precision and representation issues. Their associated problems are particularly harmful when the programmer is not aware that they are present and hence, did not take programming steps to handle such problems. For example, we might notice that our numeric comparison results are not what we expected. Consider the SAS statement “if 0.3=3*0.1 then equal='Y'; else equal='N';” If you think the result is Y, then this paper is a *must-read*. Fortunately, there are steps you can take to prevent these types of problem.

Note: Due to differences in hardware limitations and operating systems, the PC SAS examples shown below may vary from the results on other computer systems. There is not a commonly used method of performing calculations across computer systems.

NOT ALL NUMBERS CAN BE REPRESENTED EXACTLY ON THE COMPUTER

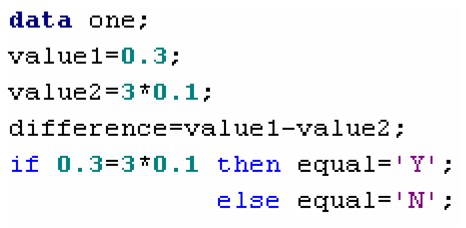
Numeric precision (i.e., the accuracy with which a number can be represented) and representation in computers are the roots of the problem. SAS uses floating-point (i.e., real binary) representation. The original decimal number and the binary-represented number may be very close, but very close is not the same as equal. There happens to be no exact binary representation for the decimal values of 0.1 and 0.3, which accounts for the difference in example #1 below. The advantage of floating-point representation is speed and its disadvantage is representation error.

Repeating decimals and irrational numbers are other obvious problems for exact storage on a computer. For example, 1/3 is equal to a decimal point followed by an infinite number of 3’s. Computers cannot store an infinite number of digits.

We need to make a distinction between the expected mathematical result (our decimal values) and what the computer can store (binary values) and program accordingly. Readers may refer to two SAS technical support references (TS-230 and TS-654) listed at the end of this paper for in-depth explanations and examples regarding floating-point representation.

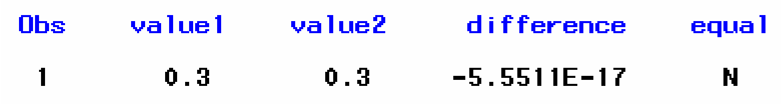
EXAMPLE #1

We know, as a mathematical fact, that 3 multiplied by 0.1 is 0.3. Therefore, when we examine the following code below, it would seem reasonable to expect that the *equal* variable will have a value of Y because both variables resolve to 0.3 (at least mathematically).



```
data one;
value1=0.3;
value2=3*0.1;
difference=value1-value2;
if 0.3=3*0.1 then equal='Y';
             else equal='N';
```

If you are thinking the only possible answer is Y, then you are in for surprise! Let’s look at the PROC PRINT output for the data set above.



```
Obs    value1    value2    difference    equal

 1       0.3       0.3    -5.5511E-17      N
```

If we use the following statement with PROC PRINT,

format value1 value2 32.31;

we get the following output.

```
Obs                     value1                          value2      difference    equal

 1    .300000000000000000000000000000    .300000000000000000000000000000    -5.5511E-17      N
```

The two values are both 0.3, but that is only as far as the PROC PRINT output goes. The two values are stored in the computer differently. In a later page, we note that SAS formats round and that is why we are not able to see the difference in the values.

To see how the values differ, let us use the HEX16. format with PROC PRINT.

```
format value1 value2 hex16.;
```

We get the following output that shows the difference between the two values.

```
Obs              value1              value2    difference    equal

 1     3FD3333333333333    3FD3333333333334    -5.5511E-17     N
```

EXAMPLE #2

Here is still another example. The difference of both pairs of numbers is mathematically 3.8, but the comparisons fail.

```
data one;
input value1 value2;
difference=value1-value2;
if difference=3.8
   then equalto3point8='Y';
   else equalto3point8='N';
cards;
16.3 12.5
15.7 11.9
;
```

Without specifying a format, we get the following results.

```
Obs    value1    value2    difference    equalto3point8

 1      16.3      12.5        3.8              N
 2      15.7      11.9        3.8              N
```

If we use the following statement with PROC PRINT,

```
format difference 32.31;
```

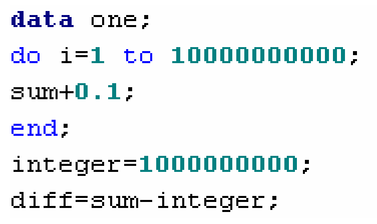
we get the following output.

```
Obs    value1    value2                          difference    equalto3point8

 1      16.3      12.5    3.800000000000000000000000000000           N
 2      15.7      11.9    3.799999999999990000000000000000           N
```

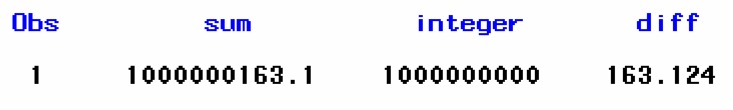
EXAMPLE #3

Representation error can become a serious problem when one is unaware it could even happen and takes no precautions against it. Unaccounted for, the size of the errors or discrepancies could accumulate over multiple operations. Let's take the simple example of adding 0.1 ten trillion times. We know the result should be one trillion.



```
data one;
do i=1 to 10000000000;
sum+0.1;
end;
integer=1000000000;
diff=sum-integer;
```

After adding all those numbers, SAS produces the following.



```
Obs          sum          integer       diff

 1      1000000163.1    1000000000    163.124
```

Over so many, many calculations, the difference accumulated to 163.124. How serious is that? It all depends on your data. This might still be tolerable for some and totally unacceptable for others. Something else to think about is what happens to other results when the tainted sum is used in other calculations.

COPING WITH THE PROBLEM

We are responsible for our data, programs, and results. The first step in solving the problem is identifying the problem and being aware of the conditions under which the problem might create undesirable results. As far as the computer science field is concerned, this is a known problem. "Most of the published algorithms for numerical analysis are designed to account for and minimize the effect of representation error."(TS-230)

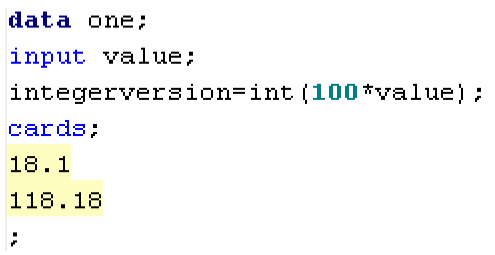
"Unfortunately there is no one method that best handles the problems caused by numeric representation error. What is a perfectly good solution for one application may drastically affect the performance and results of another application." (TS-230) Hence, this paper focuses on the simplest examples of this problem.

Coping Strategy #1: Keep It Whole

The safest way is to just deal with integers or whole numbers. If on a computer, the results of operations on integers are always integers, then there is no problem because an integer can be stored exactly in computers as long as the largest integer value the computer can represent has not been exceeded.

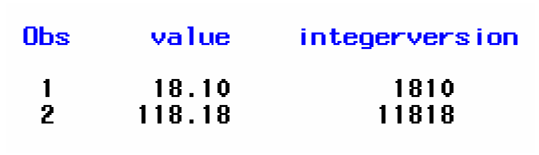
Whether you can stay within the realm of integers depends on what data are involved and what needs to be done to the data. Unless you're just adding, subtracting, and multiplying integers with integers, you could encounter a noninteger when it's time to divide an integer with another integer.

Consider the following example, which could be monetary amounts, such as dollars and cents.



```
data one;
input value;
integerversion=int(100*value);
cards;
18.1
118.18
;
```

The input values were multiplied by the scale factor of 100 (to "transfer" the digits after the two decimal places to the integer side of the number). The INT function, which returns the integer value of the argument, is then applied to remove the representation error that might have been introduced by the decimal or fractional portion of the input.



```
Obs      value     integerversion

 1       18.10          1810
 2      118.18         11818
```

You can proceed to apply integer arithmetic to the integer values. When you reach the last integer arithmetic result, you can divide it by 100 to regain the decimal portion. You can also apply a similar strategy to percentages. Percentages, such as 18%, can be multiplied by 100 and stored as 18.

Coping Strategy #2: Dare to Compare with Rounded Numbers

In examples #1 – #3 above, representation error manifested itself in the comparison of values. TS-654 recommends that you keep the following in mind when working with nonintegers or real numbers in general,

- ✓ Know your data.
- ✓ Decide on the level of significance you need.
- ✓ Remember that all numeric values are stored in floating-point representation.
- ✓ Use comparison functions, such as ROUND.

You can apply the ROUND function at strategic points in the calculation process (e.g., at the end of a series of calculations, after each calculation). What you do depends on the nature of the data, what you have to do with the data, and when representation error might become an issue. Before making an equality comparison, you can round one or both of the operands. An alternative to rounding is specifying to what degree two values are close enough so that they can be considered good as equal as far as your SAS programming is concerned. This process is called fuzzing the comparison. Refer to TS-230 for examples.

The ROUND function has the following syntax:

ROUND (*argument* <,*rounding-unit*>)

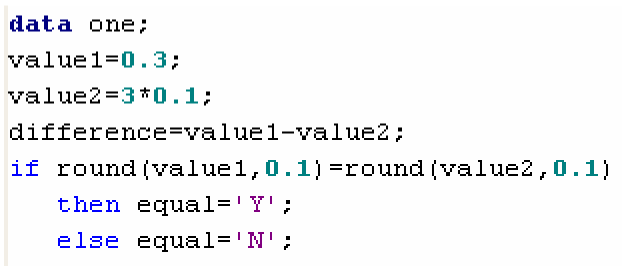
It rounds the first argument to the nearest multiple of the second argument. When the rounding unit is unspecified, it rounds to the nearest integer.

The *SAS® 9 Language Reference Dictionary* reassures us that we can expect to produce decimal arithmetic results if the result has no more than nine significant digits and either of the following conditions are true:

- ✓ The rounding unit is an integer or is a power of 10 greater than or equal to 1E-15.
- ✓ The expected decimal arithmetic result has no more than four decimal places.

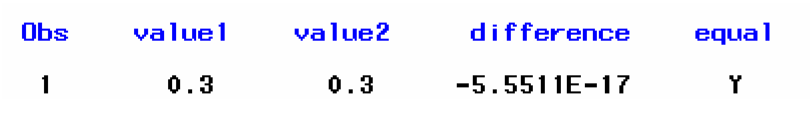
Refer to the *SAS® 9 Language Reference Dictionary* for more details regarding the ROUND function. Should the ROUND function fail to meet your needs, you may specify your own fuzz factor to use with the ROUND function. TS-230 provides examples of how to do this.

Let us modify EXAMPLE #1 to include the ROUND function for both values.



```
data one;
value1=0.3;
value2=3*0.1;
difference=value1-value2;
if round(value1,0.1)=round(value2,0.1)
   then equal='Y';
   else equal='N';
```

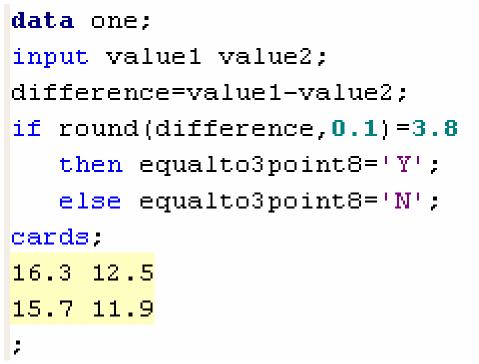
This time we can expect the correct mathematical results because the ROUND function returns the value that is based on decimal arithmetic by rounding the values to the first decimal place.



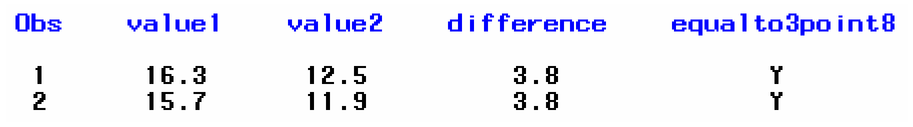
```
Obs    value1    value2    difference    equal

 1       0.3       0.3     -5.5511E-17     Y
```

Let us modify EXAMPLE #2 to include the ROUND function at the point of comparison.



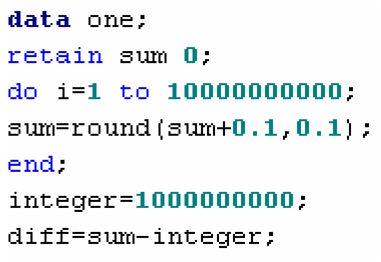
```
data one;
input value1 value2;
difference=value1-value2;
if round(difference,0.1)=3.8
   then equalto3point8='Y';
   else equalto3point8='N';
cards;
16.3 12.5
15.7 11.9
;
```



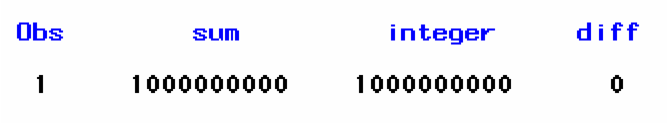
```
Obs     value1     value2     difference     equalto3point8

 1       16.3       12.5         3.8               Y
 2       15.7       11.9         3.8               Y
```

Let us modify EXAMPLE #3 to include the ROUND function each time addition occurs.



```
data one;
retain sum 0;
do i=1 to 10000000000;
sum=round(sum+0.1,0.1);
end;
integer=1000000000;
diff=sum-integer;
```



```
Obs          sum             integer        diff

 1       1000000000       1000000000         0
```

A Word on SAS Formats

Let us suppose you have numbers that have been stored with numeric representation error and all you want to do is print them out with their mathematically correct values. According to TS-230, numeric formats round. The HEX16. format is an exception (i.e., it displays the exact value of a variable in exact hexadecimal representation of the 8-byte floating-point number). Another exception is the user-defined PICTURE format, which truncates by default, in PROC FORMAT. Formats affect how numbers are displayed and do not affect how they are stored or represented internally.

Functions You May Find Useful

The ROUND and INT functions are not the only defenses against numeric representation problems. Which function you use depends on the nature of your data and computations. Here is a list of functions described in the *SAS® 9 Language Reference: Dictionary* that you may find helpful or insightful as you develop your strategies against these types of issues. The functions that have **z**ero fuzzing (i.e., CEILZ, FLOORZ, INTZ, MODZ, ROUNDZ) do not try to make their results match the values to be expected with decimal arithmetic. Hence, they may produce unexpected results. In other situations, you may have to create your own function to suit your particular needs.

FUNCTION	SYNTAX	DESCRIPTION
CEIL	CEIL (argument)	Returns the smallest integer that is greater than or equal to the argument, fuzzed to avoid unexpected floating-point results (If the argument is within 1E-12 of an integer, the function returns that integer.)
*CEIL**Z***	CEILZ (argument)	Returns the smallest integer that is greater than or equal to the argument, using zero fuzzing
FLOOR	FLOOR (argument)	Returns the largest integer that is less than or equal to the argument, fuzzed to avoid unexpected floating-point results. (If the argument is within 1E-12 of an integer, the function returns that integer.)
*FLOOR**Z***	FLOORZ (argument)	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, fuzzed to avoid unexpected floating-point results (If the argument is within 1E-12 of an integer, the function returns that integer.)
*INT**Z***	INTZ (argument)	Returns the integer portion of the argument, using zero fuzzing
MOD	MOD (*argument-1, argument-2*)	Returns the remainder from the division of the first argument by the second argument, fuzzed to avoid unexpected floating-point results
*MOD**Z***	MOD (*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, or to the nearest integer when the second argument is omitted
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
*ROUND**Z***	ROUNDZ (*argument* <,*rounding-unit*>)	Rounds the first argument to the nearest multiple of the second argument, with zero fuzzing

REFERENCES

SAS Institute Inc. 2002. *SAS® 9 Language Reference: Concepts.* Cary, NC: SAS Institute Inc.

SAS Institute Inc. 2002. *SAS® 9 Language Reference: Dictionary, Volumes 1 and 2*. Cary, NC: SAS Institute Inc.

TS-230: Dealing with numeric representation error in SAS applications. Retrieved July 1, 2008, from the SAS Web site: http://support.sas.com/techsup/technote/ts230.html

TS-654: Numeric precision 101. Retrieved July 1, 2008, from the SAS Web site: http://support.sas.com/techsup/technote/ts654.pdf

TRADEMARK NOTICE

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.