Numeric Precision 101

This paper is intended as a basic introduction to the area of numeric precision. Maybe you're just beginning to program with SAS or maybe it's been a long time since you studied number theory in college. This paper gives a basic overview of numeric precision and representation issues within SAS applications. You can read more details in TS-230 which can be downloaded from our website, SUPPORT.SAS.COM.

One of the most common issues that face each SAS user is numeric precision and representation. Many users write programs that crunch numbers and when the results are output, they assume that "what they see is what they get". Of course we expect computers to be able to do math properly…..they're at least that smart, right? If computers weren't reliable, why would nearly all business functions rely on their integrity? These sound like reasonable arguments to support our belief that computers are able to output exactly what is stored internally and represent every number as we can on paper. However, this is not entirely true.

We have a difficult time understanding what happens inside of a computer system when we ask it to represent numbers. This is because the number system with which we're most familiar and the number system the computer uses to store numbers are different. As young children we were taught math with the decimal (base ten) number system by using our ten fingers for counting. When the numbers became too large to do our calculations, we did them on paper and always knew we could trust the results. After all, that was the only way we knew to add, subtract, multiply, and divide. This number system became the standard by which everything quantitative would be based. Years later when we began using a computer, many of us tried to apply the same rules that had become second nature and realized the same rules didn't apply. Why aren't the results the same? The reason is that due to hardware limitations and the way numbers are stored, a finite set of numbers must be used to represent the infinite real number system.

People are accustomed to decimal arithmetic. Computers use binary arithmetic with finite precision. Therefore, when dealing with numbers that do not have an exact binary representation, computers often produce results that differ slightly from what people expect.

For example, the decimal values 0.1 and 0.3 do not have exact binary representations. In decimal arithmetic 3*0.1 is exactly equal to 0.3, but this equality does not hold in binary arithmetic. If you print these two values in SAS, they appear the same, but by computing the difference, you can see they are really not the same:

```
data a;
 point_three=0.3;
 three_times_point_one=3 * 0.1;
 difference=point_three – three_times_point_one;
run;
proc print noobs; run;
```

point_three	three_times_point_one	difference
0.3	0.3	1.3878E-17

Floating Point Representation

Although there are various means to store numbers, SAS uses floating point, or real binary, representation to store all numeric values. One of the main reasons SAS chose floating point was for computational speed. Floating point also allows for numbers of very large magnitude (numbers such as 2 to the 30th power) and high degrees of precision (many digits to the right of the decimal place). You're probably more familiar with this method than you realize, because it's an implementation of scientific notation. Each number is represented as a mantissa between 0 and 1 raised to a power of 10. One major difference between scientific notation and floating point representation is that on most operating systems the base is not 10, but is either 2 or 16 depending on the system. The following diagram shows a number written in scientific notation:

decimal value 987 = .987 X 10 ^{3}

- the mantissa is the number multiplied by the base, in this example, .987
- the base is the number raised to a power, in this example, 10
- the exponent is the power to which the base is raised, in this example, 3

You'll see how floating point representation is calculated on an MVS system and a PC system a little later. In most situations, you will not be affected directly by the way SAS stores numbers, however, you may see what appear to be peculiar results from time to time due to precision and representation issues.

Precision Versus Magnitude

SAS stores all numeric values in eight bytes of storage unless you specify differently. This doesn't mean that a value is limited to eight digits, but rather that eight bytes are used to store the value. A numeric value is comprised of the mantissa, the sign for the mantissa, the exponent, and the sign for the exponent. Many times real numbers can't be stored with precision, which is the accuracy with which a number can be represented. For integers, precision isn't an issue but magnitude can be. There is a maximum integer that can be accurately represented depending on the number of bytes used to store the variable.

The following chart summarizes the largest integer by length for SAS variables:

Length In Bytes	Mainframe	Unix	PC
2	256	N/A	
3	65,536	8,192	8,192
4	16,777,216	2,097,152	2,097,152
5	4,294,967,296	536,870,912	536,870,912
6	1,099,511,627,776	137,438,953,472	137,438,953,472
7	281,474,946,710,656	35,184,372,088,832	35,184,372,088,832
8	72,057,594,037,927,936	9,007,199,254,740,992	9,007,199,254,740,992

One of the first things you probably noticed about the table is that the maximum integers for each additional byte represented precisely with various lengths is different across hosts. This is due to the specifications of the floating point representation being used by each operating system. The base (2 or 16), the number of bits used to represent the mantissa, and the number of bits used to represent the exponent affect the precision that is available. Whether an operating system truncates or rounds digits that cannot be stored also affects representation error as well. Due to all of these factors that can affect storage, it's imperative that real numbers are stored in the full eight bytes of storage. If you're interested in saving disk space and would like to shorten the lengths of numeric variables that contain integers, use a LENGTH statement in your DATA step to change the number of bytes used to store the value. Be sure the values will be less than or equal to the longest integer by the length specified in the above table.

The hardware differ from one another in a variety of ways. The following chart outlines them:

	Platforms		
Specifications	IBM Mainframe	VAX/VMS	IEEE**
Base	16	2	2
Exponent Bits	7	8	11
Mantissa bits	56	55	52
Round or Truncate	Truncate	Round	Round
Bias for Exponent	64	128	1023

**IEEE is used by OS/2, Windows, and UNIX

Let's discuss what the table means.

BASE - base number system used to compute numbers in floating point representation

* base 16 - uses numbers 0-9 and letters A-F (to represent the values 10-15)

268,435,456	. . .	65,536	4096	256	16	1
16^7		16^4	16^3	16^2	16^1	16^0

the value 3000 is represented as BB8

= B*(16**2) + B*(16**1) + 8*(16**0)
= 11*256 + 11*16 + 8*1
= 3000

* base 2 – uses digits 0 and 1

128	. . .	16	8	4	2	1
2^7		2^4	2^3	2^2	2^1	2^0

the value 184 is represented as 10111000

= 1*(2**7) + 0*(2**6) + 1*(2**5) + 1*(2**4) + 1*(2**3) + 0*(2**2) + 0*(2**1) + 0*(2**0)
= 128 + 0 + 32 + 16 + 8 + 0 + 0 + 0
= 184

EXPONENT BITS – the number of bits reserved for storing the exponent, which determines the magnitude of the number we can store.

You can see that the number of exponent bits varies between the operating systems. It stands to reason that IEEE systems yield numbers of greater magnitude since they use more bits for the exponent.

MANTISSA BITS – the number of bits reserved for storing the mantissa which determines the precision by which the number is stored. Since there are more bits reserved for the mantissa on mainframes, you can expect greater precision on a mainframe compared to a PC.

ROUND or TRUNCATE – Because there is room for only two hexadecimal digits in the mantissa, a convention must be adopted on how to handle more than two digits. One convention is to truncate the value at the length that can be stored. This convention is used by IBM Mainframe systems. An alternative is to round the value based on the digits that cannot be stored, which is done on VAX/VMS and IEEE systems. There is no right or wrong way to handle this dilemma since neither convention results in an exact representation of the value.

BIAS – The bias for the exponent makes it possible to store both positive and negative exponents without the need for an additional sign bit. This will make more sense when you see examples of computing floating point representation below.

Although the IEEE platforms use the same set of specifications for floating point representation, you may occasionally see varying results between the platforms due to hardware differences. The IEEE standard determines how they store numbers in floating point, but that doesn't mean they perform all calculations the same way. It only means that the same number entered into each operating system is stored in floating point representation exactly the same way.

On the Windows platforms, the processor does computations in extended real precision. This means that instead of the normal 64 bits that are used to store numeric values (52 bits for the mantissa and 11 bits for the exponent), there are 16 additional bits, 12 for the mantissa and 4 for the exponent. Numeric values aren't stored in 80 bits (10 bytes) since the maximum width for a numeric variable in SAS is 8 bytes. This simply means that the processor uses 80 bits to represent numeric values before it's passed back to memory, which is 64 bits. On Windows, this allows storage of larger numbers than the standard IEEE floating point format used by operating systems such as UNIX. This is one reason why you may see slightly different values from operating systems that use the IEEE standard.

It is not uncommon to get slightly different results between operating systems whose floating point representation components differ (i.e. MVS and PC, MVS and UNIX). Some problems with numeric precision arise because the underlying instructions that each operating system uses to do addition, multiplication, division, etc. are slightly different. There is no standard method for doing computations since all operating systems attempt to compute numbers as accurately as possible.

It would be helpful at this point to show how floating point representation is calculated on Windows and MVS platforms. This further illustrates how each operating system represents numeric values as best it can, but with some limitations.

FLOATING POINT REPRESENTATION on WINDOWS

The byte layout for a 64 bit double precision number is as follows:

SEEEEEEE	EEEEMMMM	MMMMMMMMM	MMMMMMMMM
byte 1	byte 2	byte 3	byte 4
MMMMMMMMM	MMMMMMMMM	MMMMMMMMM	MMMMMMMMM
byte 5	byte 6	byte 7	byte 8

S=sign E=exponent M=mantissa

This example shows the conversion process for the decimal value 255.75 to floating point representation.

1. Write out the value in binary which involves the base 2 number system.

	128	64	32	16	8	4	2	1	½	¼	1/8
	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}
255	1	1	1	1	1	1	1	1			
.75									1	1	(.5 + .25=.75)

255.75 is represented in binary as 1111 1111.11

2. Move the decimal over until there's only one digit to the left of it. This process is called normalizing the value. In this case we'll move it 7 places so our exponent is 7. (1.111 1111 11)

3. The bias is 1023 so we add 7 to this number to get 1030.

4. Convert 1030 to hexadecimal representation (using base 16 number system) which is 406 and this will be placed in the exponent portion of our final result.

256	16	1
16^2	16^1	16^0
4	0	6

5. Convert 406 to binary

0100	0000	0110
4	0	6

If the value you're converting is negative, change the first bit to 1 so you have

1100 0000 0110

which in hex translates to

C 0 6

6. Go back to #2 above … throw away the first digit and decimal point (called implied 1 bit) so that you now have 11111111. Break these up into nibbles (half bytes) so that you have 1111 1111 1. In order to have a complete nibble at the end, add enough zeros to complete 4 bits: 1111 1111 1000.

7. Convert these to their hex equivalent 1111 1111 1000 and this is the mantissa portion.
F F 8

8. The final floating point representation for 255.75 is 406FF80000000000

   The final floating point representation for –255.75 is C06FF80000000000

This example was fairly straight forward in that the value was represented precisely. After seeing how the decimal portion of the value is computed in floating point, it's easy to see that many values can't be done so easily. Here's an example of how floating point representation is computed on MVS, and I've purposely chosen a value that isn't represented exactly.

FLOATING POINT REPRESENTATION on MVS

IBM mainframe operating environments (OS/390 and CMS) use the same representation made up of 8 bytes as follows:

SEEEEEEE	MMMMMMMM	MMMMMMMM	MMMMMMMM
byte 1	byte 2	byte 3	byte 4

MMMMMMMM	MMMMMMMM	MMMMMMMM	MMMMMMMM
byte 5	byte 6	byte 7	byte 8

S=sign E=exponent M=mantissa

This example shows the conversion process for the decimal value 512.1 to floating point representation. This illustrates how values that we can represent in our decimal number system can't be represented exactly in floating point, which results in numeric precision issues.

1. Because the base is 16, you must first convert the value to hexadecimal notation.

2. Convert the integer portion

256	16	1
16^2	16^1	16^0

512(decimal) = 2 0 0
= 200 * 16 ** 0

Move the decimal point all the way to the left, count the number of positions that you moved, and this is the exponent.

=.200 * 16 **3

3. Convert the fraction portion

.1 = 1/10 = 1.6/16

The numerator can't be a fraction so we'll keep the 1 and convert the .6 portion again

.6 = 6/10 = 9.6/16

Again, can't have fractions for the numerator so keep the 9 and reconvert .6 portion. The .6 continues to repeat as 9.6 which means you keep the 9 and reconvert. The closest that .1 can be represented is .1999999… * 16 ** 0 (hexadecimal)

4. The exponent for the value is 3 (step 2 above). To determine the actual exponent that will be stored, take the exponent value and add the bias to it:

true exponent + bias = 3 + 40 = 43(hexadecimal) = stored exponent

The final portion to be determined is the sign of the mantissa. By convention, the sign bit for positive mantissas is 0, and the sign for negative mantissas is 1. This information is stored in the first bit of the first byte. From the hexadecimal value in step 4, compute the decimal equivalent and write it in binary. Add the sign bit to the first position. The stored value now looks like this:

43 hexadecimal = 4*16**1 + 3*16**0 = 67 decimal = **0**1000003 binary

Had we been computing the floating point representation for -512.1, the binary representation for the exponent with sign would be:

11000003 = 195 in decimal = C3 in hex

5. The final step is to put it all together:

4320019999999999 – floating point representation for 512.1

C320019999999999 – floating point representation for –512.1

This example illustrates perfectly how values that can be represented exactly in our decimal number system can't necessarily be represented as precisely in floating point. When you notice a floating point value has a repeating pattern of numbers (like the above value has repeating '9s'), it's a good sign that this value can't be represented exactly. This is analogous to trying to represent 1/3 in base 10…..it's not exact. The closest we can come is .33333333 (repeating '3s' forever).

Now that you've learned how floating point representation is determined, let's look at an example of an arithmetic operation involving real numbers. We know that 15.7 – 11.9 should be equal to 3.8 if the calculation is done in decimal arithmetic.

```
data a;
x=15.7-11.9;
run;
```

So we should be able to compare it to the literal value of 3.8 and find that they are equal right?

```
data a;
x=15.7-11.9;
if x=3.8 then put 'equal';
else put 'not equal';
run;

not equal
```

The above output to the SAS log indicates that they are not equal. The next thing you'd probably do is output the data set with PROC PRINT to verify the value is indeed 3.8.

```
                The SAS System

          Obs                 x
           1                3.8
```

PROC PRINT indicates the value is 3.8. That's because a slight amount of fuzzing is done in the W.D format used by PROC PRINT whereby some values may appear to be different than they are stored. Now let's look at the value using a format.

```
279  data a;
280  x=15.7-11.9;
281  if x=3.8 then put 'equal';
282  else put 'not equal';
283  put x=10.8;
283  put x=18.16;
284  run;

not equal
x=3.80000000
x=3.7999999999999900
```

The first format used was 10.8 which still shows the value as 3.8; however, displaying the value with a larger width indicates that the value is slightly less than 3.8. Another way to verify that X is not 3.8 is by outputting the assigned value of 3.8 and the stored value with the HEX16. format. This is a special format that can be used to show floating point representation.

```
322  data a;
323  x=15.7-11.9;
324  literal=3.8;
325  put x=hex16. literal=hex16.;
326  run;

x=400E666666666664 literal=400E666666666666
```

You can see that the values are definitely different.

This example may be alarming to you or make you question the integrity of computer output. Regardless of how much precision is available, the problem remains that not all numbers can be represented exactly. In the decimal number system, the fraction 1/3 cannot be precisely represented. Likewise, many fractions (for example, .1) cannot be exactly represented in base 2 or base 16 numbering systems. What happens when you add 1/3 three times? Is the value exactly one? No, it's .99999. . . rather than 1. It stands to reason that when values can't be represented exactly in floating point, performing mathematical operations with other non-exact values will further compound the impreciseness of the result.

Here's another example of a numeric precision issue that occurs on MVS but not on the PC.

```
data a;
input gender $ height;
cards;
m 60
m 58
m 59
m 70
m 60
m 58
;
run;

proc freq;
tables gender/out=new;
run;
```

Output from PROC PRINT of data set NEW

Gender	COUNT	PERCENT
m	5	100

```
data final;
set new;
if percent=100 then put 'equal';
else put 'not equal';
run;
```

```
not equal
NOTE:  There were 1 observations read from the data set WORK.NEW.
```

PROC FREQ created an output data set with a PERCENT variable. Since all of the values for GENDER are the same, we would expect PERCENT to have a value of 100. When the value of PERCENT is tested, the log indicates that PERCENT is not 100. The algorithm used by PROC FREQ to produce the variable PERCENT involves mathematical computations. The result was very close to 100 but not exactly. Using the ROUND function on the IF statement resolves the issue.

```
data final;
set new;
if round(percent)=100 then put 'equal';
else put 'not equal';
run;
```

```
equal
NOTE: There  were 1 observations read from the data set WORK.NEW.
```

There are several things to keep in mind when working with real numbers.

1. Know your data
2. Decide the level of significance you need
3. Remember that all numeric values are stored in floating point representation
4. Use comparison functions such as ROUND

The safest way to avoid numeric precision issues is to use whole numbers. You may say this is impossible if your data coming into SAS contains real numbers rather than integers only. You can create integers using several functions. For example, in working with money amounts, you would be concerned with two decimal places. You could use the ROUND function to the thousands place(always use one additional decimal place than you need at the end), multiply by that amount (in this case 1000) and then use the INT function to ensure stray decimals are removed. Then you could do your calculations with the whole numbers and divide by 1000 at the end to get the decimal values back. A format with two decimal places (for money amounts) can be used.

In this example, the variable X values are stored in the SAS data set as real numbers. In order to convert them to integers, use the ROUND function to the level of significance, multiply by that level and use the INT function to return integers. To sum all values of NEW, use the SUM statement and on the last observation (detected with the END= option) the sum is divided by 1000.

```
data a;
  set b end=last;
  new=int(round(x,.001)*1000);
  sum+new;
  if last then sum=sum/1000;
run;
```

In the mathematical equation that I used to illustrate numeric precision issues, the ROUND function would have caused the values to compare equally. This happens because the computed value is very close to 3.8. Very close isn't good enough when comparing values to one another.

```
48   data a;
49     x=15.7-11.9;
50     if round(x,.01)=3.8 then put 'equal';
51     else put 'not equal';
52     run;

equal
NOTE: The data set WORK.A has 1 observations and 1 variables.
NOTE: DATA statement used:
      real time           0.20 seconds
      cpu time            0.06 seconds
```

TRANSFERRING DATA BETWEEN OPERATING SYSTEMS

The safest way to avoid numeric precision issues is to use whole numbers. You may say this is impossible if your data coming into SAS contains real numbers rather than integers only. You can create integers with several steps. For example, in working with money amounts, you would be concerned with two decimal places. You could multiply by 100 and then use the ROUND function to the nearest integer. Then you could do your calculations with the whole numbers and divide by 100 at the end to return the decimal values. A format with two decimal places (for money amounts) can be used.

In this example, the variable X values are stored in the SAS data set as real numbers. In order to convert them to integers, multiply by the level of significance (100) and then use the ROUND function to the nearest integer. To sum all values of NEW, use the SUM statement and on the last observation (detected with the END= option) the sum is divided by 100.

```
data a;
  set b end=last;
  new=round(x*100);
  sum+new;
  if last then sum=sum/100;
run;
```

For more information on moving data between operating systems, refer to the following SAS books:
Moving and Accessing SAS Files across Operating Environments, Version 8
Moving and Accessing SAS 9.1 Files