PHUSE 2008

Paper CS08

Floating Point error – what, why and how to!!

Nigel Montgomery, Novartis Pharma AG, Basel, Switzerland

ABSTRACT

Sooner or later a programmer will discover certain peculiar situations that occur while programming, that seem to be unexplainable and cause many a furrowed brow and countless head scratching.

Floating point error is one such situation that has happened to me several times during my 10 years as a SAS™ programmer. This situation, I've encountered in working for a CRO, Insurance and now Large Pharma.

INTRODUCTION

The intention of this paper is to describe what floating point error is, why it occurs, how it was encountered and possible solutions. These solutions are simply suggested ways to approach the issue, other people may have different preferences.

DISCLAIMER

My research for this paper and indeed most of the content of this paper, has come from various papers and articles I've read into over the years on floating point error and similar. All of these papers and articles can be referenced in the appendix for your use.
Through my research into this topic, I've combined what I think are the best bits of all these papers and articles with a number of my own examples that myself and colleagues have encountered over the years, in order to make people more aware of this issue and to phrase in a clear and succinct way.

1.WHAT IS FLOATING POINT ERROR? 3

1.1 FLOATING POINT REPRESENTATION. 3

1.2 KINDS OF NUMBERS REPRESENTED BY FLOATING POINT REPRESENTATION 6

1.3 TYPES OF FLOATING POINT ERRORS. 7

1.4 ROUNDING FLOATING POINT NUMBERS 8

1.5 ACCURACY ISSUES 9

1.6 SAS EXAMPLES OF FLOATING POINT ERROR 10

1.6.1 Example 1 10

1.6.2 Example 2 11

1.6.3 Example 3 13

1.7 REAL LIFE EXAMPLES ENCOUNTERED 13

1.7.1 Example 1 – Deriving categorical values 13

1.7.2 Example 2 – different results when run via different methods / systems 14

1.7.3 Example 3 – different results effecting critical statistical output 15

1.8 PROGRAMMING CONSIDERATIONS – WAYS TO HANDLE THE PROBLEM 15

1.8.1 Using Integers 15

1.8.2 Rounding your numbers 16

1.8.3 'Fuzz' factor - 'Fuzzing' your comparisons 17

1.8.4 General considerations 18

1.9 APPENDICES 20

1.9.1 References 20

1.9.2 Thankyou to..... 20

1.9.3 Contact Information 20

1.WHAT IS FLOATING POINT ERROR?

Consider the real number system that we are familiar with. This decimal system ($0 \rightarrow \pm \infty$) is obviously infinite. Have you ever considered how a particular computer system copes with such an infinite number system? Personally, and I don't mind admitting this, but I had not, until I encountered floating point error, approximately 10 years ago.

Thus, to answer the above question, floating point error is a problem that affects computer applications because of hardware limitations, in that, a finite set of numbers must be used to represent the infinite real number system. In order for me to fully describe floating point error, I must firstly describe floating point representation, the method by which most computers store numbers, both real and integers.

1.1 FLOATING POINT REPRESENTATION.

Consider the following definitions, before we proceed. These will be referred to later in this section:

PRECISION: the accuracy with which a number can be represented.

SIGNIFICANT DIGITS: a measure of the correctness of the numeric value being used. Note that the upper limit on the number of significant digits available in a representation is determined by the precision of the representation.

LOSS OF SIGNIFICANCE: the affect of an operation which reduces the number of significant digits in a numeric value.

MAGNITUDE: a measure of quantity, and in this discussion, it is expressed as a multiple of a base number system. Number systems used are normally Base 2 (binary), Base 10 (decimal) and Base 16 (hexadecimal).

Floating point representation is one of many methods, but perhaps the most common, a computer system can use to store numbers. Indeed, SAS uses this method of representing numbers for reasons such as computational speed, it allows for numbers of very large magnitude and high degrees of precision.

The term floating point refers to the fact that the decimal point (in computers, binary point) can "float". The term float here, means it can be placed anywhere relative to the significant digits of the number. This position is indicated separately in the internal representation, and floating-point representation can thus be thought of as a computer realization of scientific notation, which I am sure you are more familiar with.

Consider the following decimal value of 987, represented in scientific notation.

987 (decimal) = 0.987×10^3 (scientific notation)

The scientific notation having the following parts:
the mantissa (or significand) is the number multiplied by the base, i.e 0.987 above
the base is the number raised to a power, i.e. 10 above
the exponent is the power to which the base is raised, i.e. 3 above.

Additional to the above, the scientific notation has 2 additional components, the sign for the mantissa (+ above) and the sign for the exponent (+ above).

The major difference between scientific notation and floating point notation is that on most operating systems the base is not 10, but is either 2 or 16 depending on the system. Indeed, almost all modern computer architectures use base 2 (binary), as per the IEEE (Institute of Electrical and Electronics Engineers) standard 754.

Consider the most commonly used representation for real numbers on computers today, the **IEEE standard 754 for floating point numbers**, as mentioned above.

Like scientific notation, IEEE floating point numbers have three basic components, the sign, the exponent and the mantissa. Table 1 below shows the layout for single (32-bit) and double (64-bit) precision floating point values. The number of bits for each field are shown.

Table 1

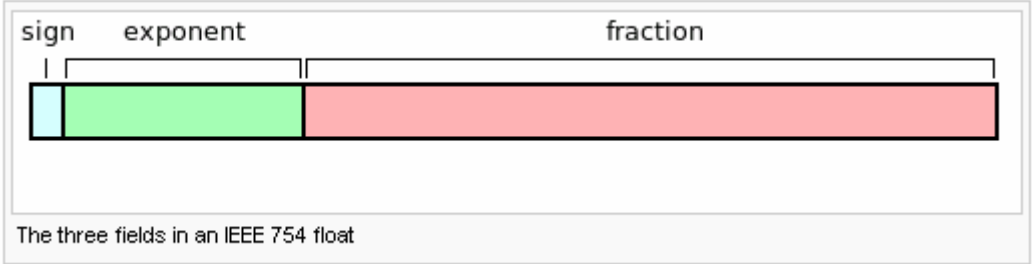
	Sign	Exponent	Fraction	Bias
Single Precision (32 bit)	1	8	23	127
Double Precision (64 bit)	1	11	52	1023

The **Sign** bit is either a 0 which denotes a positive number or a 1 which denotes a negative number.

The **Exponent** field indicates the actual magnitude of the number. It needs to represent both negative and positive exponents. To do this, a Bias (as in table 1 above) is added to the actual exponent in order to get the stored exponent (and subtracted from stored to get actual).

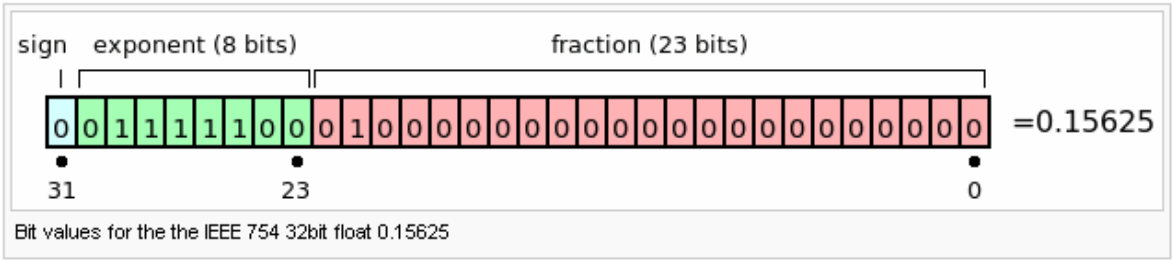
The **Mantissa**, also known as the significand, is composed of an implicit leading bit and the fraction bits. The length of the significand determines the precision to which numbers can be represented.
We assume a leading digit of 1 and thus don't need to represent it explicitly. As a result, the mantissa has 24 bits of resolution, by way of 23 fraction bits (52 and 51 fraction bits for double precision).

Now consider the following diagrammatical representation of the above floating point representation.



The three fields in an IEEE 754 float

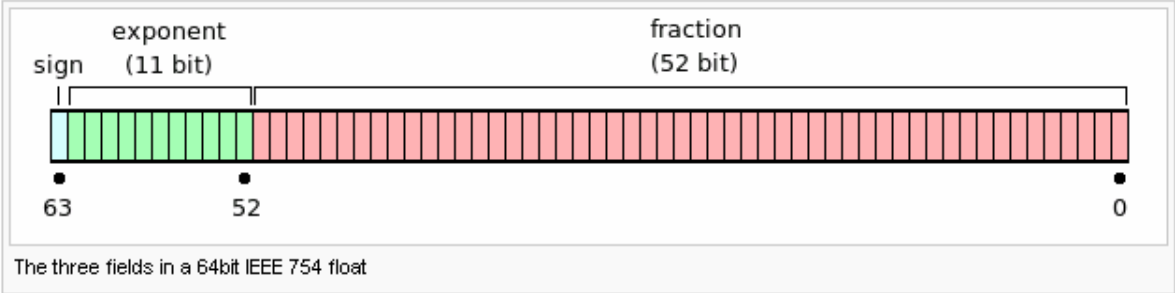
Now consider the following diagrammatical representation of the above floating point representation (32-bit / single precision) for the real number 0.15625.



Let's consider the above step by step.

- Sign bit = 0, so number is positive,
- Exponent is 01111100_2 which is 124_{10},
- Exponent bias is + 127 (as in table 1), so **actual** exponent represented here is -3,
- Fraction here is $01000000000000000000000_2$,
- Significand is 1.fraction, which is $1.01_2 = 1.25_{10}$, (where Binary fractions: $0.1_2=0.5_{10}$, $0.01_2=0.25_{10}$, $0.001_2=0.125_{10}$, $0.0001_2=0.0625_{10}$ etc)
- Decimal number represented is then: $1.25 \times 2^{-3} = 1.25 \times 0.125 = 0.15625$

Similarly, the following is a diagrammatical representation of the above floating point representation in 64-bit / double precision.



This is essentially the same as single precision, except that the fields are wider.

- The fraction part is much larger, 52 bits (instead of 23),
- The exponent is only slightly larger: 11 bits (instead of 8).

Exponent bias is +1023 (instead of +127).

The above examples are for IEEE standard 754 for floating point numbers with single and double precision, however different platforms have different specifications. Table 2 below summarizes other platform specifications. You'll notice that the IEEE standard specified here is the double precision one.

Table 2

Platform	IBM Mainframe	VAX/VMS	IEEE
Base	16	2	2
Bias	64	128	1023
Exponent bits	7	8	11
Number of digits in mantissa	14	55	52
Mantissa bits	56	55	52
Round or truncate	Truncate	Round	Round

1.2 Kinds of numbers represented by floating point representation

Let's consider the single-precision (32-bit) number below:

11110000 11001100 10101010 00001111 (32-bit integer)

Now convert this to single precision floating point representation. Essentially we 're-jig' the fields to cover a much broader range and therefore something will be 'lost'.

$= +1.1110000\ 11001100\ 10101010 \times 2^{31}$ (Single-Precision representation)

Single-precision floating-point, on the other hand, is unable to match this resolution with its 24 bit mantissa. It does, however, approximate this value by effectively truncating from the lower end giving us:

= 11110000 11001100 10101010 00000000 (Corresponding Value)

This approximates the 32-bit value, but doesn't yield an exact representation (this is loss of significance, as mentioned at the start of section 1.1). On the other hand, besides the ability to represent fractional components (which integers lack completely), the floating-point value can represent numbers around 2^{127}, compared to 32-bit integers maximum value around 2^{32}.

The range of positive floating point numbers can be split into **normalized numbers** (which preserve the full precision of the mantissa), and **de-normalized numbers** which use only a portion of the fraction's precision.

Perhaps, from the above example, now we can see where floating point error can occur?

There are actually 5 types of numbers that can be represented by floating point representation and they are as follows:

- **Zero:** Not directly representable in the straight format, due to the assumption of a leading 1. Zero is a special value denoted with an exponent field of zero and a fraction field of zero. Note that -0 and +0 are distinct values, though they both compare as equal.

  0 00000000 00000000000000000000000 = 0
  1 00000000 00000000000000000000000 = -0

- **Infinity:** The values +infinity and -infinity are denoted with an exponent of all 1s and a fraction of all 0s. The sign bit distinguishes between negative infinity and positive infinity. Generated by overflow (see later) and divisions by zero.

  0 11111111 00000000000000000000000 = Infinity
  1 11111111 00000000000000000000000 = -Infinity

- **NaNs:** The value NaN (*Not a Number*) is used to represent a value that does not represent a real number. NaN's are represented by a bit pattern with an exponent of all 1s and a non-zero fraction.

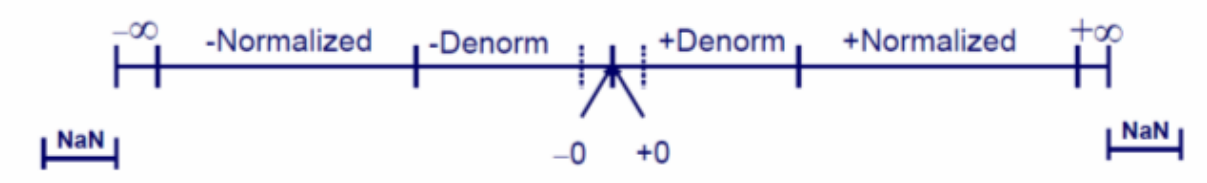
- **De-normalized:** If the exponent is all 0s, but the fraction is non-zero and does *not* have an assumed leading 1 before the binary point. Usually numbers very close to zero.

  0 00000000 10000000000000000000000 = 2^{-127}

- **Normalized:** non-zero representable real numbers.

  0 10000001 10100000000000000000000 = +1 * $2^{(129-127)}$* 1.101 = 6.5

Graphically, the above 5 types would look like the following:



1.3 Types of floating point errors.

IEEE Standards describe 5 types of floating point errors, which are as follows:

- **Inexact:** any real number that cannot be represented exactly.
  Consider the following example.

  1/3 ≠ 0.3333333333333333..... with ..... meaning recurring.

  Consider the following SAS code and resulting output:

  ```
  %let b = 1/3;
  b = &b.;
  format b 22.20;
  put "INEXACT:   Literal value is &b.   Represented value is " b;
  ```

  INEXACT: Literal value is 1/3 Represented value is 0.33333333333333000000

- **Overflow:** any number too large in magnitude to represent.
  Consider the following example:

  123456789012345**78** = 123456789012345**80**

  Consider the following SAS code and resulting output:

  ```
  %let a = 123456789012345678;
  a = &a.;
  format a 18.;
  put "OVERFLOW:   Literal integer value is &a.   Represented integer value is " a;
  ```

OVERFLOW: Literal integer value is 123456789012345678 Represented integer value is 123456789012345680

- **Underflow:** any number smaller in magnitude to that which can be represented. Underflow is actually negative overflow of the floating point quantity.

- **Division by zero:** IEEE floating point standard specifies that every floating point arithmetic operation, including division by zero, has a well-defined result. In IEEE 754 arithmetic, a ÷ 0 is positive infinity when *a* is positive, negative infinity when *a* is negative, and NaN (*not a number*) when *a* = 0. The infinity signs change when dividing by -0 instead. This is possible because in IEEE 754 there are two zero values, plus zero and minus zero, and thus no ambiguity.

- **Invalid operation:** any floating point operation which results in NaNs (Not a Number).

1.4 Rounding floating point numbers

You'll see from section 1.1, table 2 and example in section 1.2 that floating point representation and indeed floating point arithmetic are subject to 'rounding'.

All numbers expressed in floating point format are rational numbers (ratio of 2 integers) with a terminating expansion in the relevant base. Irrational numbers, such as π, or non-terminating rational numbers, must be approximated. The number of digits (or bits) of precision also limits the set of rational numbers that can be represented exactly. If the conversion of a number to floating point cannot be represented exactly, then the conversion needs to make a choice of which floating point number to use to represent the original value. The representation chosen will have a different value to the original and this is called the rounded value.

Whether or not a rational number has a terminating expansion depends on the base. Eg, in base 10 the number 1/2 has a terminating expansion (0.5) while the number 1/3 does not (0.333...). In base-2 only rationals with denominators that are powers of 2 (such as 1/2 or 3/16) are terminating. Any rational with a denominator that has a prime factor other than 2 will have an infinite binary expansion. This means that numbers which appear to be short and exact when written in decimal format may need to be approximated when converted to binary floating point representation.

Consider the decimal number, 0.1. This is expressed as a fraction in it's simplest form, as 1/10 and going by what I've said above, with 10 as the denominator, 10 is not a power of 2 and therefore this will have an infinite binary expansion (see Example 1 in section 1.6.1 for SAS example).

Again referring to the IEEE standard 754, this specifies four different rounding schemes They are as follows:

- Round to nearest, this is the default method and by far the most common,
- Round up (towards + ∞, negative results round toward zero),
- Round down (towards - ∞, negative results round away from zero),
- Round toward zero (directed rounding toward zero).

Let's concentrate on the default method above, round to nearest. This method is known by a number of names such as unbiased rounding, convergent rounding, Dutch rounding to name but a few. It is identical to the common method of rounding except when the digit(s) following the rounding digit starts with a five and has no non-zero digits after it. The new algorithm is as follows:

- Decide which is the last digit to keep.
- Increase it by 1, if the next digit is 6 or more, or a 5 followed by one or more non-zero digits.
- Leave it the same if the next digit is 4 or less
- Otherwise, if all that follows the last digit is a 5 and possibly trailing zeroes, then change the last digit to the nearest even digit. That is, increase the rounded digit if it is currently odd; leave it if it is already even.

Now we will use the above method to round to hundredths in the following examples:

- 3.016 rounded to hundredths is 3.02 (because the next digit (6) is 6 or more),
- 3.013 rounded to hundredths is 3.01 (because the next digit (3) is 4 or less),
- 3.015 rounded to hundredths is 3.02 (because the next digit is 5, and the hundredths digit (1) is odd),
- 3.045 rounded to hundredths is 3.04 (because the next digit is 5, and the hundredths digit (4) is even),
- 3.04501 rounded to hundredths is 3.05 (because the next digit is 5, but it is followed by non-zero digits)

1.5 Accuracy issues

Given the fact that floating-point numbers cannot faithfully mimic real numbers, and that floating-point operations cannot faithfully mimic true arithmetic operations, leads to many surprising situations.

While floating-point addition and multiplication are both commutative ($a + b = b + a$ and $a \times b = b \times a$), they are not necessarily associative. That is, $(a + b) + c$ is not necessarily equal to $a + (b + c)$.
Using 7-digit decimal arithmetic:

```
1234.567 + 45.67844 = 1280.245
                      1280.245 + 0.0004 = 1280.245
but
45.67840 + 0.0004 = 45.67844
                    45.67844 + 1234.567 = 1280.246
```

They are also not necessarily distributive. That is, $(a + b) \times c$ may not be the same as $a \times c + b \times c$:

```
1234.567 × 3.333333 = 4115.223
1.234567 × 3.333333 = 4.115223
                      4115.223 + 4.115223 = 4119.338
```

```
but
1234.567 + 1.234567 = 1235.802
                      1235.802 × 3.333333 = 4119.340
```

In addition to loss of significance, inability to represent numbers such as π and 0.1 exactly, and other slight inaccuracies, the following phenomena may occur:

- Cancellation: subtraction of nearly equal operands may cause extreme loss of accuracy. This is perhaps the most common and serious accuracy problem. See section 1.6.2 for example of this.
- Conversions to integer are unforgiving: converting (63.0/9.0) to integer yields 7, but converting (0.63/0.09) may yield 6. This is because conversions generally truncate rather than round.
- Limited exponent range: results might overflow yielding infinity, or underflow yielding a denormalised value or zero. If a denormalised number results, precision will be lost.
- Testing for safe division is problematic: Checking that the divisor is not zero does not guarantee that a division will not overflow and yield infinity.
- Testing for equality is problematic. Two computational sequences that are mathematically equal may well produce different floating-point values. Programmers often perform comparisons within some tolerance (often a decimal constant, itself not accurately represented). See section 1.7.1 for example of this.

1.6 SAS Examples of floating point error

1.6.1 Example 1

One of the more common SAS examples I've seen and perhaps the most common problems that programmers encounter is comparisons that do not seem correct. Consider the following mathematical expression:

$$0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 = 1$$

Coded in SAS this can be represented in SAS as the following:

```
data _null_;
   z=1;
   x=0;
   do i = 1 to 10;
      x+0.1;
   end;
   if x = z then put 'x is equal to z';
            else put 'what happened?';
run;
```

When run this writes the following to the log window, 'what happened?'. So what really did happen? Let's try outputing both x and z with a format of 32.30 using the following code:

```
data _null_;
   z=1;
   x=0;
   do i = 1 to 10;
      x+0.1;
      end;
   put x= 32.30 '32.30 format ' x= '(formatted dec)  ' z= 32.30 ' 32.30
format ';
   if x = z then put 'x is equal to z';
            else put 'what happened?';
run;
```

When run, this writes the following results to the log window:

```
X=1.000000000000000000000000000000  32.30 format
X=1 (formatted dec)
Z=1.000000000000000000000000000000  32.30 format
what happened?
```

As we can see from the above, the format displays many significant digits and there appears to be no difference between x and y, yet SAS says they are not equal. As we know from earlier, all SAS numeric variables are stored as floating point binary numbers. Usually numeric values are printed using a SAS format, which makes the value more readable, but is it exactly what is stored in the variable? In many cases it is not.

Now let's try using the same code as before but outputing both x and y with a format of hex16. (hexadecimal format).

```
data _null_;
   z=1;
   x=0;
   do i = 1 to 10;
      x+0.1;
      end;
   put x= hex16. '(hex) ' x= '(formatted dec)  ' z= hex16. ' (hex) ';
   if x = z then put 'x is equal to z';
            else put 'what happened?';
run;
```

When run, this writes the following results to the log window:

```
X=3FEFFFFFFFFFFFFF (hex) X=1 (formatted dec)
Z=3FF0000000000000 (hex)
what happened?
```

From the above output, we can see that indeed, x and y are different when represented in hexadecimal format. Indeed, HEX16. is the only SAS format that displays the exact value of the variable. The reason for this, as stated earlier in section 1.4, 0.1 cannot be represented exactly in binary floating point, no matter what the precision.

1.6.2 Example 2

Let's consider the following example, that involves an arithmetic operation.

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

When run, this writes 'not equal' results to the log window.
Now let's output the value to output window using proc print, giving us the following output:

```
    The SAS System

    Obs     X      Z
     1     3.8    3.8
```

The proc print indicates that the value is 3.8, but is it really?
Let's now try putting x and z out to the following formats, 10.8 and 18.16.

```
data a;
 x = 15.7 - 11.9;
 z = 3.8;
 if x = z then put 'equal';
 else put 'not equal';
 put x=10.8 '    ' z=10.8;
 put x=18.16 '    ' z=18.16;
run;
```

When run, this writes the following results to the log window:

```
not equal
X=3.80000000    Z=3.80000000
X=3.7999999999999900    Z=3.8000000000000000
```

From the above, we see that proc print seems to have applied some kind of fuzzing to the W.D format and outputs the value of x as 3.8, when really it is not.

As per our last example, we can also check this by using the HEX16. (hexadecimal) format.

```
data a;
 x = 15.7 - 11.9;
 z = 3.8;
 if x = z then put 'equal';
 else put 'not equal';
 put x=hex16. ' :x in hexadecimal format ' z=hex16.  ' :z in hexadecimal
format ';
run;
```

When run, this writes the following results to the log window:

```
not equal
X=400E666666666664  :x in hexadecimal format Z=400E666666666666  :z in hexadecimal format
```

From the hexadecimal format above, we can definitely see that the values are different.

The above 2 examples may make you question the integrity of the computer output, as indeed I did. However, regardless of how much precision is available, the issue remains that not all numbers can be represented exactly using floating point representation.
Consider the fraction of 1/3 (one third), in the decimal number system. This cannot be precisely represented as a finite decimal. This is similar to floating point representation.

1.6.3 Example 3

Consider the following example SAS code that again uses arithmetic operations:

```
data _null_;
 x= -16 - 0.1 + 16 + 0.1; put x= hex16. x= ;
run;
```

When run, this writes the following results to the log window:

```
X=BCD9800000000000 X=-1.41553E-15
```

As we can see, the result is certainly not one that we would expect and obviously we would expect to see a value of zero for x.

Now let's break this down into separate expressions to see if we can see what has happened.

```
data _null_;
 x= -16;                  put x= hex16. x= ' Step 1';
 x= x - 0.1;              put x= hex16. x= ' Step 2';
 x= x + 16;               put x= hex16. x= ' Step 3';
 x= x + 0.1;              put x= hex16. x= ' Step 4';
run;
```

When run, this writes the following results to the log window:

```
X=C030000000000000 X=-16  Step 1
X=C03019999999999A X=-16.1  Step 2
X=BFB9999999999A00 X=-0.1  Step 3
X=BCD9800000000000 X=-1.41553E-15  Step 4
```

Looking at the hexadecimal format for each step of the calculation, we can see the following. At step 2 we have representation error (floating point error) and at step 3 we have loss of significance, thus giving us a value that is not equal to zero.

1.7 Real life examples encountered

The following are examples of floating point error/representation, that myself and colleagues have encountered in our daily work in recent years. Example 1 goes into specific detail on the issue, whereas examples 2 and 3 are only brief descriptions of the issues discovered.

1.7.1 Example 1 – Deriving categorical values

Consider the following data for 2 patients, whereby, MAPB is a mean calculated at baseline, MAPM1 is a mean calculated at endpoint, CMAPM1 is the change from baseline (MAPB – MAPM1) and DUM is simply a dummy variable (entered as DUM = -1 in SAS).

PATIENT	MAPB	MAPM1	CMAPM1	DUM
1	4.0714285714	3.0714285714	-1	-1
2	1.6428571429	0.6428571429	-1	-1

This all looks fine, but wait.....this data was displayed using default formats – using longer formats (32.30) the data look different.

PATIENT	MAPB	MAPM1	CMAPM1	DUM
1	4.07142857142857000000	3.07142857142857000000	-0.99999999999999000000	-1.00000000000000000000
2	1.64285714285714000000	0.64285714285714000000	-1.00000000000000000000	-1.00000000000000000000

As you can see, the decimal values of MAPB and MAPM1 look the same for both patients, yet only one patient (patient 2) seems to give CMAPM1 = -1.

Now, as learnt previously, the HEX16. format will display the data exactly and this gives the following:

PATIENT	MAPB	MAPM1	CMAPM1	DUM
1	4010492492492492	4008924924924925	BFEFFFFFFFFFFFFC	BFF0000000000000
2	3FFA492492492492	3FE4924924924925	BFEFFFFFFFFFFFFF	BFF0000000000000

From the above, we see that for patient 1, CMAPM1 is not equal to -1 exactly (DUM is the hexadecimal value of -1) which I'm sure we expected from the previous data, but also we see that patient 2 has a CMAPM1 that does not seem to equal to -1.

Hence, given what we have dicovered about this data, when we then create our categorical response variables (as per SAS code below) we get the following results for the response variables:

```
if (. < cmapm1 le -1) then r1cmapm1=1;
else r1cmapm1=0;
```

PATIENT	MAPB	MAPM1	CMAPM1	DUM	R1CMAPM1
1	4.07142857142857000000	3.07142857142857000000	-0.99999999999999000000	-1.00000000000000000000	0
2	1.64285714285714000000	0.64285714285714000000	-1.00000000000000000000	-1.00000000000000000000	0

Now, from the above explanation, we can explain what is going on here, but is this the correct interpretation of the data?

NO – we should have taken 'uncertainty' into account here i.e uncertainty due to the fact that we're dealing with floating point representation here and also uncertainty in the underlying measurement data.

In section 1.8, we will see a number of ways to deal with such a situation.

1.7.2 Example 2 – different results when run via different methods / systems

This issue occurred when a current colleague worked on the helpdesk for their previous company and they received a call from an employee within the statistics section of the company. The employee described the issue, that they were seeing a discrepancy in a mean value in an output, between when the same program was run on windows and when it was run on UNIX.Although the difference in mean value did not make a substantial difference in the output, the caller wanted to know why such a discrepancy existed.

Knowing what I know now, I feel this was an issue due to the difference in specifications for methods of running via windows to that of running via UNIX, in a similar way as we describe in table 1 and 2 in section 1.1.

1.7.3 Example 3 – different results effecting critical statistical output

This example is very similar to the one above, in that 2 slightly different results were produced by 2 different people (one person producing and one qcing) from 2 different companys. In this case, a more serious discrepancy was found, in that the discrepancy was the difference between a significant p-value and non-significant one.
Again, I feel this discrepancy can be explained due to the fact the 2 people were operating off two different platforms and thus had 2 separate specifications for floating point representation, as per tables 1 and 2 in section 1.1.

1.8 Programming considerations – ways to handle the problem

Having read this paper up to this section, you should now have a good understanding of floating point representation and also be aware of why floating point error occurs. Knowing this, puts us in a good position to knowing the way to handle the issue of floating point error. Indeed, there are many proposed solutions, but no one solution fits all. What is a good solution for one issue, may be far from ideal for another one.

In the following section, I will describe the suggested methods for solving floating point error and issues it causes. I will make reference to previous examples in sections 1.6 and 1.7 and provide my own personal preferred solution to each example. I'll also point out some programming considerations to take into account when you are doing your own programming.

1.8.1 Using Integers

Using integers is one way of dealing with floating point error, in fact, it avoids it. This method is to simply store the entire numeric value as an integer. Financial applications use this method, whereby instead of storing pounds and pence or dollars and cents, the values would be stored as pence or cents only and therefore the value would be stored as an integer.
Hence, once the values are stored as integers, expressions can be written to work with integer values. Integer arithmetic would be used provided the maximum continuous value was not exceeded.
One method of converting fractions to integers is to simply, multiply by a scale factor to create an integer from the fraction and then apply the INT function to remove any representation error that was introduced by the fraction.
Personally, this would not be my preferred option, but is definitely worth considering dependant on your use.

1.8.2 Rounding your numbers

Rounding your numbers at selected points during computation, is another way to deal with floating point error. It allows you to determine how many significand digits you require for your computation and get a consistent representation based on that number of significant digits. Note, rounding is still subject to representation error, if the value is rounded to a fractional value.

Consider Example 2 in section 1.6.1.

```
data a;
 x = 15.7 - 11.9;
 z = 3.8;
 if x = z then put 'equal';
 else put 'not equal';
 put x=10.8 '    ' z=10.8;
 put x=18.16 '    ' z=18.16;
run;
```

When run, this writes the following results to the log window:

```
not equal
X=3.80000000    Z=3.80000000
X=3.7999999999999900    Z=3.8000000000000000
```

Now applying appropriate rounding to the if condition would be one possible solution.

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

When run, this writes 'equal' to the log window. You'll notice that I've rounded both x and z, so that I am comparing like with like.
We have applied rounding here (to hundredths) due to the underlying uncertainty of floating point representation, but how do we know what is the appropriate degree of uncertainty?

My final solution for Example 1 in Section 1.7.1 was to use rounding, as this seemed the best way to deal with the data involved. You'll notice that I've created a macro variable called 'fuzz', but this is simply the round-off unit used in the round function.

Code used:

```
%let fuzz=1e-15;
data trial2;
 set trial;
 if (. < cmapm1 le -1) then r1cmapm1=1;
 else r1cmapm1=0;
 if (round((cmapm1),&fuzz) le -1 and round((cmapm1),&fuzz) gt .) then
r2cmapm1 = 1;
 else r2cmapm1 = 0;
run;
```

Giving me my new response variable R2CMAPM1 (of 1 instead of 0, as in R1CMAPM1)

PATIENT	MAPB	MAPM1	CMAPM1	DUM	R1CMAPM1	R2CMAPM1
1	4.0714285714	3.0714285714	-1	-1	0	1
2	1.6428571429	0.6428571429	-1	-1	0	1

1.8.3 'Fuzz' factor - 'Fuzzing' your comparisons

Another possibility to deal with floating point error is to use a 'fuzz' approach. There exists in SAS the function FUZZ that when used at the correct time and place, can be useful when dealing with floating point error. This function returns the nearest integer if the argument is within 1E -12, otherwise it returns the argument and has the following syntax:

FUZZ (argument)

This is fairly useful and should be considered when you are dealing with comparisons and boundary values.
The fuzz function could have been used in the previous example, to give the correct outcome, using the following code:

```
if ( . lt fuzz( cmapm1 ) le -1 ) then r2cmapm1 = 1;
else r2cmapm1 = 0;
```

Indeed, floating point error significantly affects comparisons. You have already seen an example of this with the creation of the response variables R1CMAPM1 and R2CMAPM1 as in section 1.8.2 (and 1.7.1), whereby originally, R1CMAPM1 was created with a response of zero (No), when it should really have been one (Yes), as we have done for R2CMAPM1.

When using the 'eq' (EQUAL) operator (includes 'ge' and 'le') in a comparison, the items being compared, must be exactly equal for the comparison to be true. Now as most systems use floating point representation and can introduce some form of floating point error, then when using the operator, we can see where unexpected results can occur.
One way of getting around the potential issues here is to round one or both of the operands in the comparison as in the first example in section 1.8.2. Alternatively, we can **fuzz** the comparison. This means that if the operands are close enough, then they should be considered equal. Again we could ask here, how do we define 'close enough'?

Consider the following small macro, as an example of a 'fuzzed comparison', that I've seen a number of times in different papers (See 'Dealing with Numeric Representation Error in SAS Applications' in references for example)

```
%macro eqfuzz(var1, var2, fuzz=1e-12);
 abs(&var1 - &var2) < &fuzz;
%mend eqfuzz;
```

This macro returns a true value if the two operands are equal within a certain tolerance.
We will use this code for the example in section 1.6.1.

```
data _null_;
   z=1;
   x=0;
   do i = 1 to 10;
      x+0.1;
      end;
   if x = z then put 'x is exactly equal to z';
   else if %eqfuzz(x,z) then put 'x close enough to z';
            else put 'x not even close to z';
run;
```

When run, this writes 'x close enough to z' to the log window.
You may notice that the definition of EQFUZZ is similar to that of absolute error. We know that any such operator based on relative error is more consistent across magnitudes. Thus the EQFUZZ operator, would look like the following when based on relative error.

```
%macro eqfuzz(var1, var2, fuzz=1e-12);
 abs((&var1 - &var2) / &var1) < &fuzz
%mend eqfuzz;
```

Notice, no adjusting of fuzz value is needed in changing from absolute error to relative error. Thus by defining this new operator, EQFUZZ, we are 'fuzzing' our comparisons to have a relative error of less than a pre-defined tolerance.

1.8.4 General considerations

The PROC COMPARE procedure judges numeric values to be equal / unequal if the magnitude of the difference is greater than the value specified in the CRITERION option. When using the compare procedure, we must think about what type of values we are comparing and if those are numeric types, then this may be affected by floating point error. The methods available are as follows:

- EXACT – tests for exact equality (default if no criterion specified),
- ABSOLUTE – compares absolute difference to that specified in the criterion,
- RELATIVE – compares the absolute relative difference to that specified by the criterion (default if criterion specified),
- PERCENT – compares the absolute percent difference to that specified by the criterion.

Note that this procedure uses unformatted values as the basis for all comparison methods.

The 'numeric' data type in SAS is the only numeric data type a SAS user can define. It represents all numbers as floating point values. The length statement in SAS is used to assign an amount of memory to the representation and thus would recommend that you always declare numeric variables with a length of 8. When numeric variables with different lengths specified, you may run into issues here, so another reason why to always declare numeric variables with a length of 8.

From the numerous examples we've seen in this paper, perhaps it is obvious that formatting can sometimes 'blind' us from what really is going on. Formats do not alter the underlying data values, but are simply used to control the output of the data. Thus, be aware, what you see, is NOT what you've got (unless in HEX16. format).

Some procedures can trap floating point errors using the TRAP / NOTRAP option, but by default this option is not activated. I guess it's good to know when and where we have a

floating point error, but then we have to make a decision of what to do with it, hopefully the correct one.

Thus to summarise, we should pay particularly attention to our coding and the values used when performing the following:

- Repeating a slightly inaccurate computation many times,
- Adding two quantities of very different magnitude,
- When calculating the difference of two very similar values and using result in further calculations.

Remember that what you see is often NOT what you've got and remember to use rounding appropriately. If you expect an integer, make sure it is one – floating point values can be inexact representations of integers.

1.9 APPENDICES

1.9.1 References

- IEEE Standard for Binary Floating-Point Arithmetic (ANSI/IEEE Std 754-1985) http://standards.ieee.org/
- SAS Technical support document TS-230 'Dealing with Numeric Representation Error in SAS Applications'
- 'Lets get to the (Floating) Point' by Chris Hecker
- SAS Online Doc
- 'Survey of floating point formats', (http://www.mrob.com/pub/math/floatformats.html)
- Comp.soft-sys.sas
- 'What every computer scientist should know about floating point arithmetic' by David Goldberg
- 'Handling Floating-Point Exceptions in Numeric Programs' by John R Hauser
- 'The pitfalls of verifying floating point computations' by David Monnieux
- 'The perils of floating point' by Bruce M Bush, http://www.lahey.com/float.htm
- Chapter 7.2 of the Intel Architecture Software Developers Manual Volume 1: Basic Architecture
- Fundamentals of Computer Science – IEEE floating point representations of real numbers, http://www.math.grin.edu/~stone/courses/fundamentals/IEEE-reals.html
- Floating Point formats, http://www.quadibloc.com/comp/cp0201.htm
- The ever reliable wikipedia website for definitions

1.9.2 Thankyou to.....

- Steve Pike (previously Novartis, now Roche) for triggering the initial discussions on floating point error and for giving a basis to work off for this paper,
- Benjamin Szilagyi and Laurence Designe (both Novartis) for reviewing this paper and giving some valuable feedback,
- Laurence Designe for allowing me to be 'flexible' with my working hours and location to enable me to prepare this paper,
- Roy Ward, Verena Walter and Colin Brown (all Novartis) for sharing with me, examples of where they had discovered floating point error in their daily work.

1.9.3 Contact Information

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

Author Name: Nigel Montgomery
Company: Novartis Pharma AG
Address: Novartis Pharma AG, Postfach
City / Postcode: Basel, CH 4002, Basel-Stadt, Switzerland
Work Phone: (0041) 61 32 46180
Fax: (0041) 61 32 43039
Email: nigel.montgomery@novartis.com