Paper CT05
Floating-Points are as Easy as 01, 10, 11!

Chloe Floodgate, PHASTAR, Macclesfield, United Kingdom

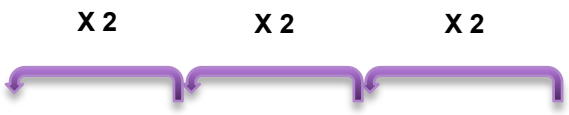
ABSTRACT
Why can't our computer do basic maths? Sooner or later as SAS programmer we will come across the dreaded floating-point error and what we see is not always what we get. Sometimes our SAS programs can produce some very peculiar results, and the intent of this paper is to give a basic overview as to why this could occur. We aim to understand what numeric representation and precision mean, or simply how SAS sees numbers, and describe what a floating-point error is.

INTRODUCTION
Floating-point error is an inherent consequence of representing real numbers with finite precision in digital computers. While floating-point arithmetic enables the efficient handling of a vast range of values, spanning from very large to very small, this comes at the cost of numerical accuracy. Because not all real numbers can be represented exactly in binary or decimal floating-point formats, small discrepancies arise during storage, arithmetic operations, and data conversion. These discrepancies, known as floating-point errors, can accumulate and propagate through computations, potentially leading to significant deviations from expected results. The aim of this paper is to understand the fundamentals of what floating-point errors are, discuss why they occur, show an example of a floating-point error, and provide you with some viable solutions on how you can deal with them.

NUMERIC REPRESENTATION AND PRECISION
Numeric representation and precision are the fundamentals to understanding floating-point errors. Numerical representation means how we express numbers using a specific format or system. For example, most of us will be familiar with the decimal number system, where the fraction 1/10 has a decimal representation of 0.1. Numerical precision means the accuracy with which these numbers are approximated or represented to.

FLOATING-POINT REPRESENTATION
Floating Point Representation is how SAS, and most other programming languages, chooses to represent real numbers. On an operating system such as Windows, SAS uses IEEE 754 double-precision floating point representation. This means that numbers are stored in binary, not decimal, with a fixed amount of storage. Depending on the operating system, it will use binary base 2 or 16, and the fixed number of bits is usually 64 (double precision). Although Floating-point representation is a very powerful method to store an extended range of values, this representation can't exactly store all decimal numbers, which therefore leads to small inaccuracies.

BINARY NUMBERS
As discussed, floating point representation, chooses to store numbers in binary – but what exactly are binary numbers? Binary is a number system that uses only two digits: 0 and 1. Each digit in a binary number is called a bit (short for "binary digit") and it is the foundation of how computers store and process data. Binary is called a base-2 numeral system, unlike the decimal system (base-10) which uses digits from 0 to 9. How it works is that each position in a binary number represents a power of 2, starting from the right.
For example, what the binary number 100111 looks like in our decimal system:



Binary Place Value	32 (2^5)	16 (2^4)	8 (2^3)	4 (2^2)	2 (2^1)	1 (2^0)
Binary	1	0	0	1	1	1

By creating a table in which the header is multiplied by 2 each time, you can then place your binary numbers into this table and add the numbers in the binary place value row that have a 1 in the binary row.

In this instance:

- **1** in the **32** column means there is a **32**

- **0** in the **16** columns means there are no **16**'s
- **0** in the **8** columns means there are no **8**'s
- **1** in the **4** column means there is a **4**
- **1** in the **2** column means there is a **2**
- **1** in the **1** column means there is a **1**

We can then add these together:

(1x32) + (0x16) + (0x8) + (1x4) + (1x2) + (1x1) = 32 + 4 + 2 + 1 = 39

Hence the number 100111 in binary is equivalent to 39 in our decimal system.

FLOATING-POINT ERRORS

For those of you who are lucky enough to not have encountered a floating-point error already, it is essentially a result of inaccuracies due to hardware limitations. Computers are finite machines, with a finite memory, and because of this, they cannot store an infinite set of numbers with perfect precision. As discussed, our computers store numbers in binary, and similar to decimal representation, there are some binary values that have infinitely repeating representations, which are ultimately rounded or truncated in order to be stored in our computers' limited storage.

For example, the same way we can't decimally represent 2/3's as the 6s repeated indefinitely, the fractional value 1/10 has an infinitely repeating representation in binary.

Decimal: **2/3** = **0.66666…** with the **6**s repeated indefinitely.
Binary: **1/10** = **0.0001100110011001100110011**… where the pattern **0011** is repeated indefinitely.

In binary, you can see that 1/10 has a pattern where 0011 is repeated indefinitely. Therefore, to be able to store this in a computer, the value must be rounded or truncated. When we repeatedly perform calculations and comparisons on these imprecise numbers in SAS, we can end up with some unexpected results.

We can visualize the issues that may arise when performing calculations on these repeating binary values, by using this decimal representation as an example:

2/3 rounded to **3dp** = **0.667**
As a result of performing repeated calculations, we get:
2/3 + 2/3 + 2/3 = 0.667+ 0.667+ 0.667 = 2.01

If we round 2/3 to 3 decimal places, we get 0.667. We would expect 3 lots of 2 thirds to be 6/3 or 2. However, when we add 3 times the amount of 0.667 we get 2.01 not 2. Imagine this issue applied to infinitely recurring binary values, that have had to be rounded or truncated in order to be stored in the computer, albeit they are stored with more precision than this, the issue persists.

HOW FLOATING-POINT ERRORS CAN AFFECT US

Although these differences seem tiny in nature, there are certain occasions where these tiny differences will make a difference to our SAS procedures. Some examples of our day-to-day programming affected by this error include:

- IF, SELECT and WHERE statements may not produce desired results when using calculated values that may have been rounded or truncated.
- Using PROC COMPARE on SAS data sets might show differences in variables that look as though they should be equal.
- Statistical models may behave unpredictably when comparing numeric values with floating point rounding involved.

FLOATING-POINT ERRORS IN ACTION

Below is a worked example of a floating-point error in SAS:

```
data PhuseEU2025;
  *Create a computed variable which add 10
   amounts of 1/10;
   Computed = (1/10)+(1/10)+(1/10)+(1/10)
             +(1/10)+(1/10)+(1/10)+(1/10)
             +(1/10)+(1/10);

  *Create a specified variable which assigns
   value 1;
   specified = 1;

  *Compare computed variable and specified
   variable and flag if equal;
   If computed = specified then help = "Y";
run;
```

rows: 1 Total columns: 3

COMPUTED	SPECIFIED	HELP
1	1	

Above is a worked example of a floating-point error in SAS. A variable called computed is created- which is the addition of 10 amounts of 1/10. In the resulting data set, the value of the computed variable is 1, which seems logical. However, we are aware that 0.1 has an infinitely repeating representation in binary. A new variable called specified is created which explicitly assigns value 1. Recalling that both of these variables have output a value of 1, an IF statement is produced to flag the record if computed= specified. Logically 1 = 1, so we expect our record to be flagged with variable help = Y. However, when run in SAS, 1 does not in fact equal 1, and the variable help is not assigned.

HOW TO HANDLE FLOATING-POINT ERRORS

A few examples of best programming practices we should be using to help account for floating-point errors are:

- We can use the **ROUND** function to ensure that the computed numeric values are represented consistently, with as much precision as applicable. This allows us to have full control of how SAS is storing our numbers for future IF/WHERE statements for example, and ensure our values are as close as possible to their exact values.

An example of when we may want to implement the round function is given below:

```
data PhuseEU2025;
  *Calculated percentage - option 1;
   a=(5/9997654)*100

  *Calculated percentage - option 2;
   b=(5*100)/9997654;

  *Compare options 1 and option 2;
   if a=b then c="Y"
run;
```

Total rows: 1 Total columns: 3

	A	B	C
1	0.0000500117	0.0000500117	

In theory, calculating the same percentage in two different ways should give the same result; however, we can see that variable 'C' is not subsequently flagged as 'Y' despite them looking as though they are equal. In this instance, performing IF/WHERE statements will give incorrect results, and proc compares won't match (unless CRITERION is specified – which is discussed in more detail below). If we use the ROUND function to round both results, to a level of precision that is statistically suitable – for example 10dp, we can ensure the values stored in SAS are equal and we can have more control of our results, for example:

```
data PhuseEU2025;
*Calculated percentage - option 1;
a_rnd= round ( (5/9997654)*100 , 0.00000000001);

*Calculated percentage - option 2;
b_rnd= round ( (5*100)/9997654 , 0.00000000001);

*Compare options 1 and option 2;
if a_rnd=b_rnd then c_rnd="Y";
run;
```

Total rows: 1 Total columns: 3

	A_RND	B_RND	C_RND
1	0.0000500117	0.0000500117	Y

We can see here that once the values have been rounded to 10dp, variable C is succesfully assgined as 'Y'.

- Another option to control floating point errors, briefly mentioned above, is using the **CRITERION** option on the PROC COMPARE to set the tolerance for numeric comparisons. This essentially controls how SAS decides whether two numeric values are "equal," even if they're not exactly the same due to floating point error. The ABSOLUTE method specific here means that the compare will use the absolute difference to the value specified by the CRITERION option. There are a variety of methods that can be used here.

An example of how the PROC CRITERION option works is shown below:

```
data base;
  input id value;
  datalines;
  1 10.0000
  2 20.0000
  3 30.0000
  ;
run;
```

Total rows: 3 Total columns: 2

	ID	VALUE
1	1	10
2	2	20
3	3	30

```
data compare;
  input id value;
  datalines;
  1 10.0001
  2 20.0002
  3 30.0000
  ;
run;
```

Total rows: 3 Total columns: 2

	ID	VALUE
1	1	10.0001
2	2	20.0002
3	3	30

ID Var 1 will have a difference of 0.0001

ID Var 2 will have a difference of 0.0002

ID Var 3 has no difference

Total rows: 3 Total columns: 2

	ID	VALUE
1	1	10
2	2	20
3	3	30

Total rows: 3 Total columns: 2

	ID	VALUE
1	1	10.0001
2	2	20.0002
3	3	30

```
proc compare base=base compare=compare
             listall error warning;
run;
```

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

Value Comparison Results for Variables

                    ||      Base    Compare
Obs                 ||     VALUE      VALUE      Diff.     % Diff
____________________||_____________________________________________
                    ||
  1                 ||   10.0000    10.0001  0.0001000   0.001000
  2                 ||   20.0000    20.0002   0.000200   0.001000
```

By creating 2 datasets called base and compare, we can see from looking at them side by side that ID var 1 has a difference of 0.0001, ID var 2 has a difference of 0.0002, and ID var 3 has no difference. By running a proc compare

on these datasets with no PROC CRITERION option we get observation 1 and 2 flagged as unequal, as these quite clearly have a difference of 0.0001 and 0.0002 respectively. Observation 3 is not flagged as unequal.

Total rows: 3 Total columns: 2

	ID	VALUE
1	1	10
2	2	20
3	3	30

Total rows: 3 Total columns: 2

	ID	VALUE
1	1	10.0001
2	2	20.0002
3	3	30

```
proc compare base=base compare=compare
             criterion=0.0001 method = absolute
             listall error warning;
run;
```

The COMPARE Procedure
Comparison of WORK.BASE with WORK.COMPARE
(Method=ABSOLUTE, Criterion=0.0001)

Value Comparison Results for Variables

Obs	Base VALUE	Compare VALUE	Diff.	% Diff
2	20.0000	20.0002	0.000200	0.001000

However, when running the proc compare with criterion specified as 0.0001, or in other words, asking SAS to ignore any differences less than or equal to this decimal, we get only observation 2 flagged, as this difference of 0.0002 is in fact larger than 0.0001.

CONCLUSION

Floating-point errors are a byproduct of how computers store numbers. They're not bugs-but you need to understand them, especially as a SAS programmer in clinical research, where calculations and manipulation of data is crucial.

REFERENCES

- https://pharmasug.org/proceedings/2014/CC/PharmaSUG-2014-CC50.pdf
- https://support.sas.com/resources/papers/proceedings11/275-2011.pdf
- https://documentation.sas.com/doc/en/pgmsascdc/9.4_3.5/lepg/p0dv87zb3bnse6n1mqo360be70qr.htm
- https://www.youtube.com/watch?v=PZRI1IfStY0
- https://library.fiveable.me/numerical-analysis-i/unit-2

CONTACT INFORMATION

Chloe Floodgate
Phastar
UK HQ:
2D Bollo Ln,
Chiswick,
London,
W4 5LE
Email:chloe.floodgate@phastar.com