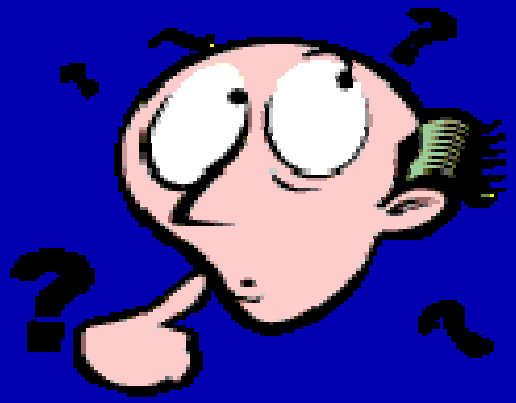
CS08: FLOATING POINT ERROR – What, why and how to!!



Nigel Montgomery, Senior Statistical Programmer at Novartis Pharma AG

PhUSE 2008 Manchester, 13th October 2008,

PhUSE 2009 Single Day Event in Basel, 18th March 2009.

NOVARTIS

Table of contents

1. Introduction

- How does a computer deal with our infinite real number system?
  - Floating point error affects computer applications because of hardware limitations

1. Introduction

- SAS examples ex1 & ex2

2. Floating point representation

Why floating point?

Decimal point (or binary point in computers) can *float*

The position of the point is represented seperately in the internal representation and thus, Floating Point representation can be thought of as a computer realization of scientific notation.

1 major difference:

- Most operating systems NOT base 10, but either 2 or 16

2. Floating point representation

Consider the decimal value 987 in scientific notation:

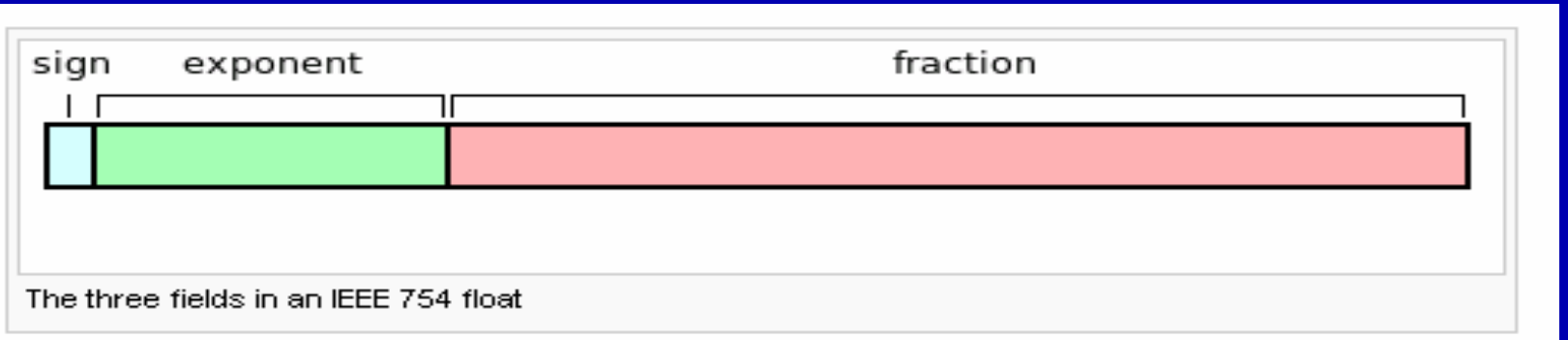
987 (decimal) = 0.987×10^3 (scientific notation)

Having the following parts:

- Mantissa (significand or fraction), is the number multiplied by the base, i.e. 0.987
- Base, is the number raised to a power i.e. 10
- Exponent, the power to which the base is raised i.e. 3
- Two additional parts are the signs for mantissa and exponent i.e. +

2. Floating point representation

IEEE standard 754 for floating point numbers



The three fields in an IEEE 754 float

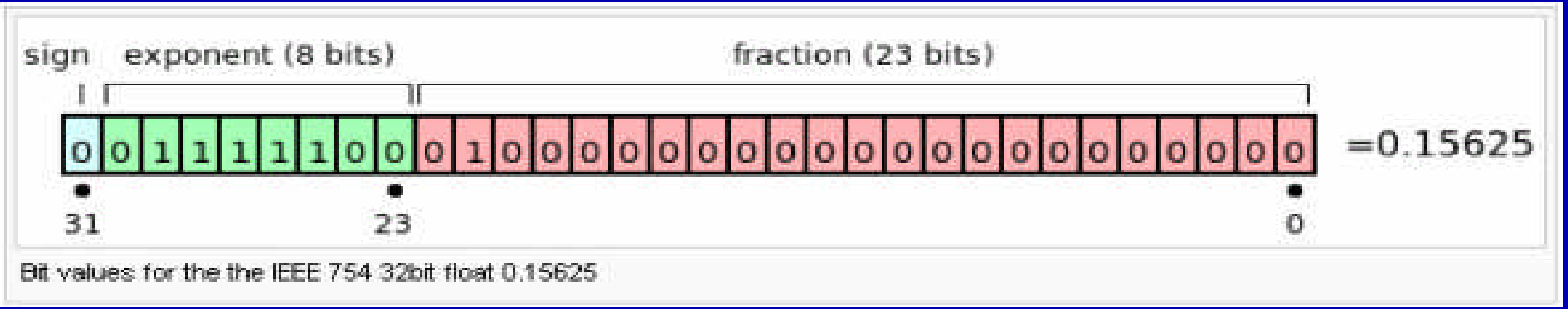
	Sign	Exponent	Fraction	Bias
Single Precision (32 bit)	1	8	23	127
Double Precision (64 bit)	1	11	52	1023

Bias: used to represent both –ve and +ve exponents.

- Added to actual exponent to get stored exponent.
- And vice versa

Significand is an implicit leading bit (assumed as 1) and the fraction bits

2. Floating point representation



Let‘s consider this step by step:

- sign bit=0, so number is positive
- exponent is 01111100_2 which is 124_{10}
- exponent bias is 127, so actual exponent here is -3
- fraction here is $01000000000000000000000_2$
- significand is 1.fraction, which is $1.01_2=1.25_{10}$ (where binary fractions $0.1_2=0.5_{10}$, $0.01_2=0.25_{10}$ etc)
- decimal number is then $1.25 \times 2^{-3} = 1.25 \times 0.125 = 0.15625$

3. Floating point error

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:

= +1.1110000 11001100 10101010 x 2^{31} (single precision rep)

We see that single precision floating point is unable to match this resolution with it's 24 bit mantissa, and therefore has to approximate (by truncating)

= 1110000 11001100 10101010 00000000

3. Floating point error

All numbers expressed in floating point format are rational numbers with a terminating expression in the relevant base.

Irrational numbers must be approximated

Whether or not a rational number has a terminating expansion depends on the base.

- 1/2 = 0.5 (terminating)
- 1/3 = 0.3333..... (non-terminating)

In base 2:

- Rationals with denominators that are powers of 2 (1/2, 3/16) are terminating,
- Any rational with denominator that has a prime factor other than 2 will have an infinite binary expansion

3. Floating point error

If the conversion of a number to floating point cannot be represented exactly, then conversion needs to make a choice of which floating point number to use to represent the original value.

This choice is called rounding (of which the IEEE standard 754 specifies 4):

- **Round to nearest (unbiased, convergent, Dutch rounding)**
- Round up
- Round down
- Round towards zero

3. Floating point error

5 types of floating point error:

- **Inexact (See ex3)**
- **Overflow (See ex4)**
- **Underflow - opposite of overflow**
- Division by zero - IEEE floating point standard specifies that every floating point arithmetic operation, including division by zero, has a well defined result.
- Invalid operation - any floating point operation which results in NaNs (Not a Number).

3. Floating point error

Accuracy issues:

- Associative law, (a + b) + c = a + (b + c) is not always true
- Distributive law, (a + b) x c = a x c + b x c is not always true
- Loss of significance
- Testing for equality is problematic (see working example later).

3. Floating point error

Floating point error example (ex5, ex5a):

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

NOTE: HEX16. is the only SAS format that displays the exact value of the variable

3. Floating point error

Floating point error daily work example:

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

Where:

- MAPB is mean calculated at baseline
- MAPM1 is mean calculated at endpoint
- CMAPM1 is change from baseline (MAPB - MAPM1)
- DUM is a dummy variable (entered as dum = -1 in sas)

This all looks fine, but wait.....

3. Floating point error

PATIENT	MAPB	MAPM1	CMAPM1	DUM
1	4.071428571428570000000	3.071428571428570000000	-0.999999999999990000000	-1.000000000000000000000
2	1.642857142857140000000	0.642857142857140000000	-1.000000000000000000000	-1.000000000000000000000

When we apply a format of 32.30 we get the above.

Now using the hex16. format that we know displays the exact value we get:

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

Thus using the following code for the response variable R1CMAPM1:

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

3. Floating point error

Gives us:

PATIENT	MAPB	MAPM1	CMAPM1	DUM	R1CMAPM1
1	4.071428571428570000000	3.071428571428570000000	-0.999999999999990000000	-1.000000000000000000000	0
2	1.642857142857140000000	0.642857142857140000000	-1.000000000000000000000	-1.000000000000000000000	0

Is this correct?

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.

I will provide a solution for this in the next section - possible solutions.

4. Possible Solutions

In my opinion, no one solution fits all.

- Using Integers: store the entire numeric value as an integer.
  - Uses: Financial applications use this method, whereby pounds and pence is stored as pence only and dollars and cents are stored as cents only
- Rounding your numbers: at selected points during computation
  - See ex2a
  - How do we know what to round to?
- Fuzz factor: SAS function FUZZ returns the nearest integer if the argument is within 1E-12, otherwise returns the argument.
  - Syntax: FUZZ(argument)

4. Possible Solutions

- Fuzz factor continued
  - This could have been used in the daily work example from earlier to give the correct outcome using the following code:

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

- Fuzzing your comparisons: if operands are close enough, then they should be considered equal.
  - Again, we could ask here, what is *close enough*?

The following code is suggested code for fuzzing your comparisons (based on relative error):

4. Possible Solutions

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

See ex1a for possible usage.

NOTE: some procedures trap floating errors using the TRAP / NOTRAP option (Default is not activated).

5. Summary

Pay particular attention to our coding and the values used when performing the following:

- repeating a slightly inaccurate computation many times
- adding 2 quantities of very different magnitude
- calculating the difference of 2 very similar values and using the result in further calculations

REMEMBER:

- what you see is often NOT what you‘ve got
- use rounding appropriately
- if you expect an integer, make sure it is one
- floating point values can be inexact representations of integers.

Questions?