



Floating-Point Error

DH08: Do Not Trust Your Own Computer- generated Numbers



PHUSE EU Connect 2019

12Nov2019 / Matthias Lehrkamp / 1.0





Not all numbers can be represented exactly

$$1/3 = 0.333333...$$

$$\pi = 3.141593...$$

SAS using 64 bits to store numbers

→ Lead to rounding errors

Computer and humans using different number systems



Humans



Decimal number system

$$\begin{array}{rcl} 10.5625 & = & 1010.1001 \\ 5.875 & = & 0101.1110 \end{array}$$



Computer

Binary number system

 On  Off

→ Can lead to conversion errors

Even simple calculations can lead to rounding and conversion errors!

0.1 + 0.2 = 0.3 ?

```
DATA _NULL_;  
a= 0.1 + 0.2;  
b= 0.3;  
d= a - b;  
IF a=b THEN check= 'equal';  
      ELSE check= 'not equal';  
PUT a= 32.31 / b= 32.31;  
PUT d= 32.31 / d= best.;  
PUT check=;  
RUN;
```

Numbers are rounded for display representation (conversion error)

Log:

[illegible]

The small difference is based on a rounding error



Even simple calculations can lead to rounding and conversion errors!

0.1 + 0.2 = 0.3 ?

```
DATA _NULL_;
  a= 0.1 + 0.2;
  b= 0.3;
  d= a - b;
  IF a=b THEN check= 'equal';
  ELSE check= 'not equal';
  PUT a= binary64. / b= binary64.;
  PUT d= 32.31 / d= best.;
  PUT check=;
RUN;
```

Log:

```
a=0011111111101001100110011001100110011
001100110011001100110011001100110100
b=0011111111101001100110011001100110011
001100110011001100110011001100110011
d=.00000000000000000000555111512312578
d=5.551115E-17
check=not equal
```





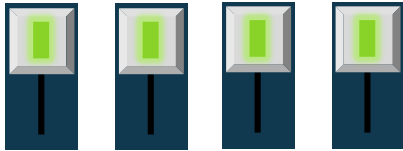
Understanding the issue



Understanding the issue

How numbers are stored in the standard IEEE 754

The 4-bit pattern for integers



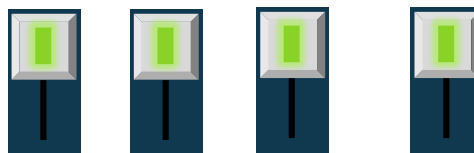
2^3	2^2	2^1	2^0	
8	4	2	1	← decimal value of the bit
0	0	0	1	$= 1 = 0*2^3 + 0*2^2 + 0*2^1 + 1*2^0$
0	0	1	0	$= 2 = 0*2^3 + 0*2^2 + 1*2^1 + 0*2^0$
0	0	1	1	$= 3 = 0*2^3 + 0*2^2 + 1*2^1 + 1*2^0$
0	1	0	0	$= 4 = 0*2^3 + 1*2^2 + 0*2^1 + 0*2^0$
0	1	0	1	$= 5 = 0*2^3 + 1*2^2 + 1*2^1 + 0*2^0$
0	1	1	0	$= 6 = 0*2^3 + 1*2^2 + 1*2^1 + 0*2^0$
0	1	1	1	$= 7 = 0*2^3 + 1*2^2 + 1*2^1 + 1*2^0$
1	0	0	0	$= 8 = 1*2^3 + 0*2^2 + 0*2^1 + 0*2^0$

How does it work
with decimal places?

Understanding the issue

How numbers are stored in the standard IEEE 754

4-bit pattern for decimal numbers



	2^{-1}	2^{-2}	2^{-3}	2^{-4}
	.5	.25	.125	.0625
$0 \cdot 2^{-1} + 0 \cdot 2^{-2} + 0 \cdot 2^{-3} + 1 \cdot 2^{-4} = 0.0625$	0	0	0	1
$0 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} + 0 \cdot 2^{-4} = 0.125$	0	0	1	0
$0 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} + 1 \cdot 2^{-4} = 0.1875$	0	0	1	1
$0 \cdot 2^{-1} + 1 \cdot 2^{-2} + 0 \cdot 2^{-3} + 0 \cdot 2^{-4} = 0.25$	0	1	0	0
$0 \cdot 2^{-1} + 1 \cdot 2^{-2} + 0 \cdot 2^{-3} + 1 \cdot 2^{-4} = 0.3125$	0	1	0	1
$0 \cdot 2^{-1} + 1 \cdot 2^{-2} + 1 \cdot 2^{-3} + 0 \cdot 2^{-4} = 0.375$	0	1	1	0
$0 \cdot 2^{-1} + 1 \cdot 2^{-2} + 1 \cdot 2^{-3} + 1 \cdot 2^{-4} = 0.4375$	0	1	1	1
$1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 0 \cdot 2^{-3} + 0 \cdot 2^{-4} = 0.5$	1	0	0	0

Conclusions

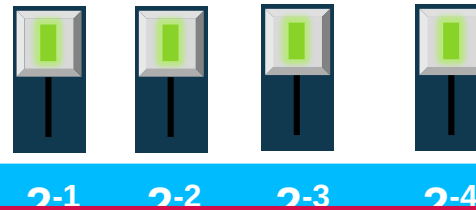
- the value from one bit to the next bit is always the half
- the value of each bit always ends with the number 5
- one decimal place more per bit



Understanding the issue

How numbers are stored in the standard IEEE 754

4-bit pattern for decimal numbers



Only a few floating-point numbers can be stored exactly in the binary system!

Conclusions

- the value from one bit to the next bit is always the half
- the value of each bit always ends with the number 5
- one decimal place more per bit

$$0 \cdot 2^{-1} + 1 \cdot 2^{-2} + 1 \cdot 2^{-3} + 1 \cdot 2^{-4} = 0.4375$$

0 1 1 1

$$1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 0 \cdot 2^{-3} + 0 \cdot 2^{-4} = 0.5$$

1 0 0 0



Understanding the issue

How numbers are stored in the standard IEEE 754

1/9 floating point numbers with 1 decimal place can be stored exactly

.1	.2	.3	.4	.5	.6	.7	.8	.9
----	----	----	----	----	----	----	----	----



Understanding the issue

How numbers are stored in the standard IEEE 754

3/99 floating point numbers with 2 decimal place can be stored exactly

.01	.02	.03	.04	.05	.06	.07	.08	.09	.10
.11	.12	.13	.14	.15	.16	.17	.18	.19	.20
.21	.22	.23	.24	.25	.26	.27	.28	.29	.30
.31	.32	.33	.34	.35	.36	.37	.38	.39	.40
.41	.42	.43	.44	.45	.46	.47	.48	.49	.50
.51	.52	.53	.54	.55	.56	.57	.58	.59	.60
.61	.62	.63	.64	.65	.66	.67	.68	.69	.70
.71	.72	.73	.74	.75	.76	.77	.78	.79	.80
.81	.82	.83	.84	.85	.86	.87	.88	.89	.90
.91	.92	.93	.94	.95	.96	.97	.98	.99	



Understanding the issue

How numbers are stored in the standard IEEE 754

Most numbers have an accuracy of around 16 digits.

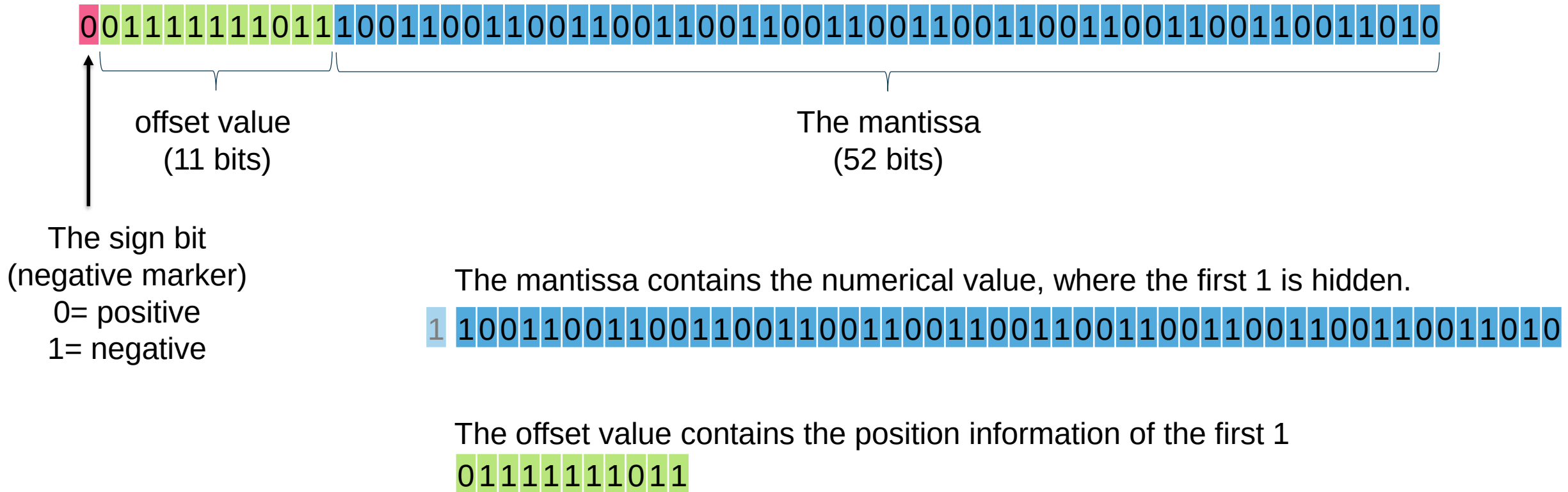
0.1	=	0.100000000000000001
0.2	=	0.200000000000000001
0.3	=	0.29999999999999999
0.4	=	0.400000000000000002
0.5	=	0.5
0.6	=	0.59999999999999998
0.7	=	0.69999999999999996
0.8	=	0.800000000000000004
8.8	=	8.80000000000000007
16.8	=	16.80000000000000007
256.8	=	256.80000000000000114



Understanding the issue

How numbers are stored in the standard IEEE 754

Double floating-point number 0.1





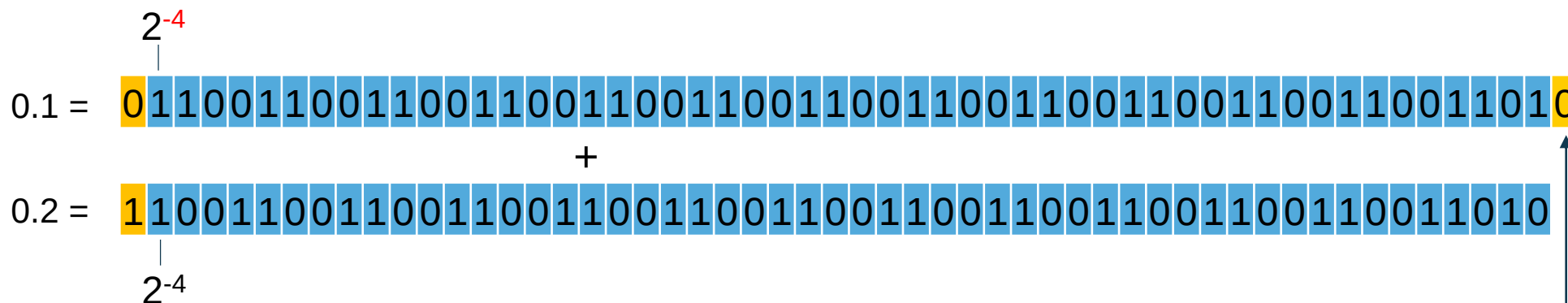
Understanding the issue

Step 1: All numbers must shift to the highest bit value

$$\begin{array}{r} 0.1 = \begin{array}{c} 2^{-5} \\ | \\ 1 \end{array} 1001100110011001100110011001100110011001100110011001100110011010 \\ + \\ 0.2 = \begin{array}{c} 2^{-4} \\ | \\ 1 \end{array} 100110011001100110011001100110011001100110011001100110011010 \end{array}$$

Understanding the issue

Step 1: All numbers must shift to the highest bit value



Adding up bit numbers	
0 + 0 =	0
0 + 1 =	1
1 + 0 =	1
1 + 1 =	0 remember 1 for the next bit

Overhanging bits are kept during the calculation as the cache provides 80 bits.
12 additional bits for the mantissa
4 additional bits for the offset

Understanding the issue

Step 2: Add up the numbers

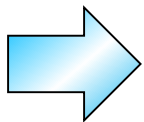
$$\begin{array}{rcl}
 0.1 = & 011001100110011001100110011001100110011001100110011001100110011010 & \\
 & + & \\
 0.2 = & 11001100110011001100110011001100110011001100110011001100110011010 & \\
 & = & \\
 0.1 + 0.2 = & 100110011001100110011001100110011001100110011001100110011001110 & \\
 & \underbrace{\hspace{1.5cm}}_{2 \text{ overhanging bits}} \rightarrow \text{shift the mantissa one position to the right} & \\
 \Rightarrow & 100110011001100110011001100110011001100110011001100110011001110 &
 \end{array}$$

Understanding the issue

Step 3: Store the result

$$\begin{array}{rcl}
 0.1 = & 0 & 11001100110011001100110011001100110011001100110011001100110011010 \\
 & + & \\
 0.2 = & 1 & 1001100110011001100110011001100110011001100110011001100110011010 \\
 & = & \\
 0.1 + 0.2 = & 10 & 01100110011001100110011001100110011001100110011001100110011001110
 \end{array}$$

2 overhanging bits → shift the mantissa one position to the right



$$100110011001100110011001100110011001100110011001100110011001110$$

mathematical rounding

2.49 ≈ 2.0	0 01 ≈ 00 00
2.50 ≈ 2.0	0 10 ≈ 00 00
2.51 ≈ 3.0	0 11 ≈ 01 00
3.49 ≈ 3.0	1 01 ≈ 01 00
3.50 ≈ 4.0	1 10 ≈ 10 00
3.51 ≈ 4.0	1 11 ≈ 10 00

Overhanging bits are rounded mathematically, always towards the next 0.

$$10000$$

Understanding the issue

0.1 = 011001100110011001100110011001100110011001100110011001100110011010

+

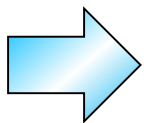
0.2 = 11001100110011001100110011001100110011001100110011001100110011010

=

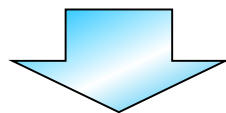
0.1 + 0.2 = 100110011001100110011001100110011001100110011001100110011001110



2 overhanging bits → shift the mantissa one position to the right.



100110011001100110011001100110011001100110011001100110011001110



Rounding

0.1 + 0.2 = 0011111111010011001100110011001100110011001100110011001100110100

≠

0.3 = 001111111101001100110011001100110011001100110011001100110011011



Examples and solutions



Examples and solutions

Example 1: Whenever possible use integer number

```
DATA cent_from_nothing;
euro= 0;
cent= 0;
step= 0;
DO WHILE( diff<1 );
euro= euro + 0.1;
cent= cent + 10;
diff= euro*100 - cent;
step= step + 1;
END;
PUT euro= cent= step= diff=;
RUN;
```

LOG: euro=20408989.91 cent=2040898990 step=204089899 diff=0.0100000016

Receive 1 cent after 204,089,899 money transfers!

Examples and solutions

Example 2: Use the fuzzy function to compare integer numbers

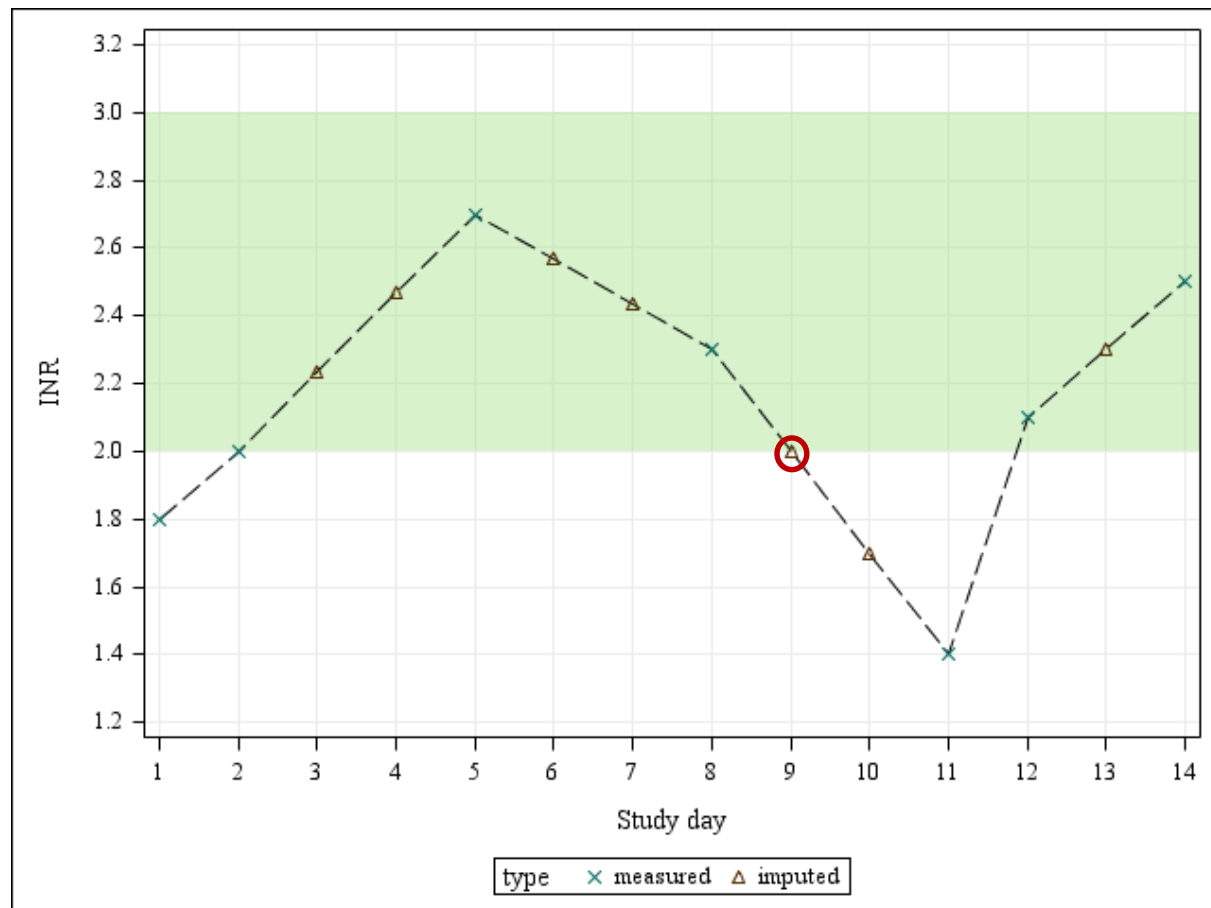
```
DATA inr01_checkRange;
  SET inr;
  LENGTH INRANGEFL $1.;
  IF 2.0 <= inr <= 3.0 THEN inrangefl= 'Y';
  ELSE inrangefl= 'N';
```

RUN;

Per figure:
11/14 days in range
78.57%

Per program compare:
10/14
71.43%

	LBDY	INR	INRANGEFL
1	1	1.8	N
2	2	2	Y
3	3	2.2667	Y
4	4	2.5333	Y
5	5	2.7	Y
6	6	2.5667	Y
7	7	2.4333	Y
8	8	2.3	Y
9	9	2	N
10	10	1.7	N
11	11	1.4	N
12	12	2.1	Y
13	13	2.3	Y
14	14	2.5	Y



Examples and solutions

Example 2: Use the fuzzy function to compare integer numbers

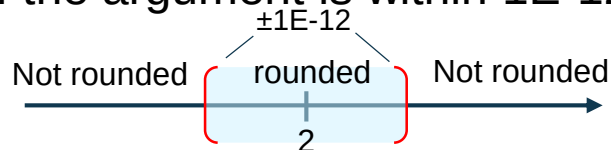
DATA in
SET
LENG
IF 2

RUN;

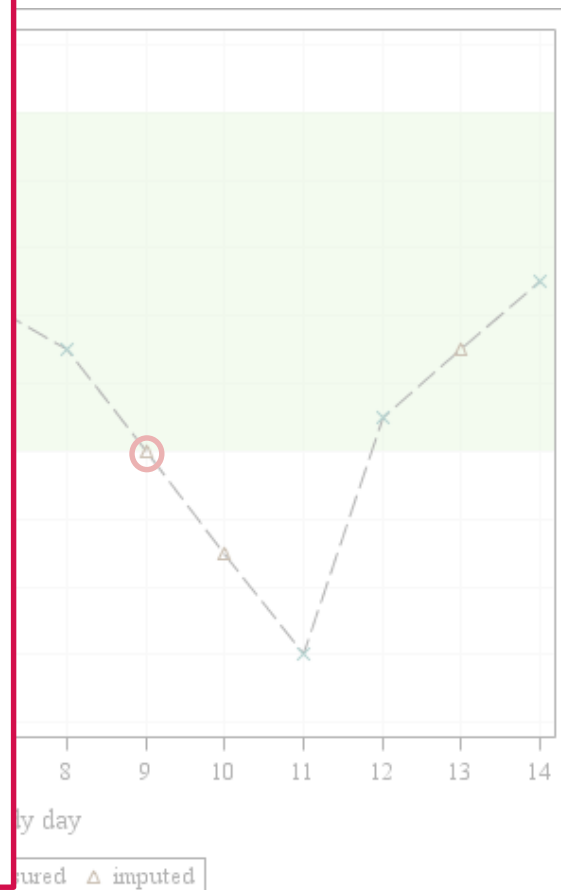
Fuzz function `fuzz(argument)`

Returns the nearest integer if the argument is within 1E-12 of that integer.

`f= fuzz(x);`



<code>x=1.9999999999988</code>	<code>f=1.9999999999988</code>	not rounded
<code>x=1.9999999999989</code>	<code>f=1.9999999999989</code>	not rounded
<code>x=1.9999999999990</code>	<code>f=2.0000000000000</code>	rounded 2 - 1E-12
<code>x=1.9999999999991</code>	<code>f=2.0000000000000</code>	rounded
...		
<code>x=2.0000000000000</code>	<code>f=2.0000000000000</code>	rounded
...		
<code>x=2.00000000000008</code>	<code>f=2.0000000000000</code>	rounded
<code>x=2.00000000000009</code>	<code>f=2.0000000000000</code>	rounded
<code>x=2.00000000000010</code>	<code>f=2.00000000000010</code>	not rounded 2 + 1E-12
<code>x=2.00000000000011</code>	<code>f=2.00000000000011</code>	not rounded



Examples and solutions

Example 2: Use the fuzzy function to compare integer numbers

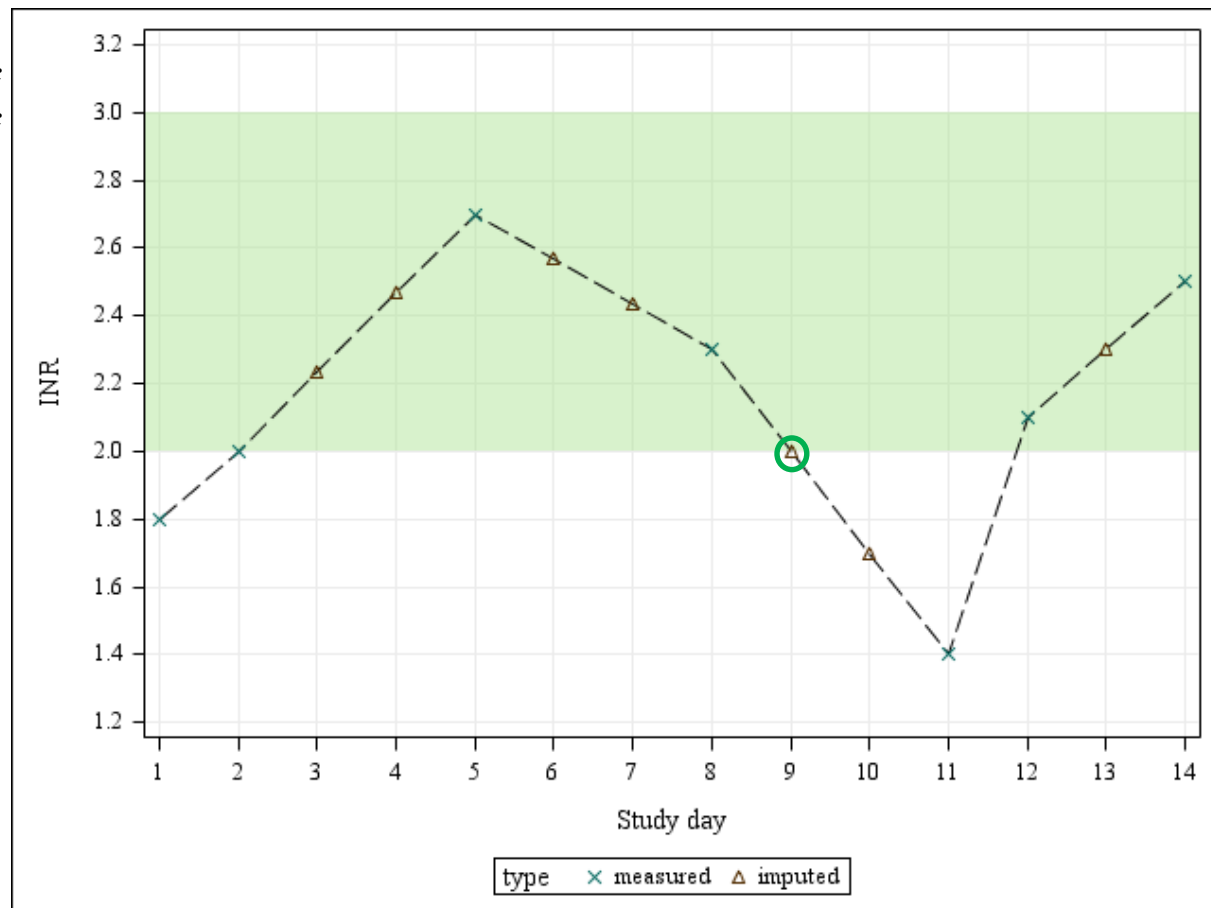
```
DATA inr01_checkRange;
  SET inr;
  LENGTH INRANGEFL $1.;
  IF 2.0 <= fuzz(inr) <= 3.0 THEN inrangefl= 'Y';
  ELSE inrangefl= 'N';
```

RUN;

Per figure:
11/14 days in range
78.57%

Per program compare:
10/14
78.57%

	LBDY	INR	INRANGEFL
1	1	1.8	N
2	2	2	Y
3	3	2.2667	Y
4	4	2.5333	Y
5	5	2.7	Y
6	6	2.5667	Y
7	7	2.4333	Y
8	8	2.3	Y
9	9	2	N
10	10	1.7	N
11	11	1.4	N
12	12	2.1	Y
13	13	2.3	Y
14	14	2.5	Y





Examples and solutions

Example 3: Round floating-point numbers before every compare with the round function

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

```
DATA _NULL_;  
  DO i=0.1 TO 0.5 BY 0.1;  
    IF i=0.3 THEN PUT i= '-> 0.3 reached';  
    ELSE PUT i= '-> 0.3 not reached';  
  END;  
RUN;
```

Log:

```
i=0.1 -> 0.3 not reached  
i=0.2 -> 0.3 not reached  
i=0.3 -> 0.3 not reached  
i=0.4 -> 0.3 not reached  
i=0.5 -> 0.3 not reached
```

0.3 is never met!!!



Examples and solutions

Example 3: Round floating-point numbers before every compare with the round function

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

```
DATA _NULL_;  
  DO i=0.1 TO 0.5 BY 0.1;  
    IF round(i,0.1)=0.3 THEN PUT i= '-> 0.3 reached';  
    ELSE PUT i= '-> 0.3 not reached';  
  END;  
RUN;
```

Log:

```
i=0.1 -> 0.3 not reached  
i=0.2 -> 0.3 not reached  
i=0.3 -> 0.3 reached  
i=0.4 -> 0.3 not reached  
i=0.5 -> 0.3 not reached
```



Examples and solutions

Example 4: Compare the decimal values (text comparison)

```
DATA _NULL_;  
  a= 0.1 + 0.2;  
  b= 0.3;  
  IF put(a,best.)=put(b,best.) THEN check= 'equal';  
    ELSE check= 'not equal';  
  PUT a= best. / b= best. / check=;  
RUN;
```

Log:

```
a=0.3  
b=0.3  
check=equal
```

In case the dimension of the numbers is unclear or it is difficult to find the right rounding unit, compare the decimal values (text comparison).

$0.3 = 0.3$



Summary

- rounding errors are small (~16 precise digits)
- if possible avoided decimal places
- Use SAS functions (FUZZ, ROUND) or text values to compare floating point numbers
- Be careful with Big Data analysis
- Be careful with statistical methods doing many calculations



Any Questions?

Thank you!



matthias.lehrkamp@bayer.com



Tip in

R offers the possibility to increase the display accuracy

```
> options( digits=22 )
```

```
> 0.1 + 0.2  
[1] 0.30000000000000000000444089
```

```
> 0.3  
[1] 0.2999999999999999999888978
```

```
> (0.1 + 0.2) - 0.3  
[1] 5.551115123125782702118e-17
```

It can be very confusing to see other numbers as entered,
but it would be a way to find errors