Is .1+.2 equal to .3?

Shaoji Xu

Introduction

Is .1+.2 equal to .3? Let us run the following program on PC first.

```
data _null_;
if .1+.2=.3 then put 'equal';
else put 'not equal';
run;
```

We will see 'not equal' in the printout. Now let's run the following program.

```
data s;
do i=.1 to .9 by .1;
a=i;
output;
end;
data t;
do i=.1 ,.2,.3,.4,.5,.6,.7,.8,.9;
b=i;
output;
end;
data p;
merge s t;
by i;
proc print;
run;
```

Obs	i	a	b
1	0.1	0.1	0.1
2	0.2	0.2	0.2
3	0.3	.	0.3
4	0.3	0.3	.
5	0.4	0.4	0.4
6	0.5	0.5	0.5
7	0.6	0.6	0.6
8	0.7	0.7	0.7
9	0.8	0.8	.
10	0.8	.	0.8
11	0.9	0.9	.
12	0.9	.	0.9

You can see there are three more observations in data p than that in s and in t. SAS treats .3 (in s) ^=.3 (in t), .8 (in s)^=.8 (in t) and .9 (in s)^=.9 (in t). Furthermore you can see that .3 (in s)>.3 (in t), .8(in s)<.8(in t) and .9(in s) <.9(in t).

So what is wrong in this program? Nothing is wrong. You have to notice that SAS uses format BEST. to print out values. Values in the printout are not that stored in computer, not that SAS uses to make comparison. To see exact values stored in computer that SAS

uses to make comparison, we can use hex16. format or binary64. format. Now let's run the following program and see the results.

```
data s;
do i=.1 to .9 by .1;
a=i;a1=put(i,hex16.);
output;
end;
data t;
do i=.1 ,.2,.3,.4,.5,.6,.7,.8,.9;
b=i;b1=put(i,hex16.);
output;
end;
data p;
merge s t;
by i;
proc print;
var i a b a1 b1;
run;
```

Obs	i	a	b	a1	b1
1	0.1	0.1	0.1	3FB999999999999A	3FB999999999999A
2	0.2	0.2	0.2	3FC999999999999A	3FC999999999999A
3	0.3	.	0.3		3FD3333333333333
4	0.3	0.3	.	3FD3333333333334	
5	0.4	0.4	0.4	3FD999999999999A	3FD999999999999A
6	0.5	0.5	0.5	3FE0000000000000	3FE0000000000000
7	0.6	0.6	0.6	3FE3333333333333	3FE3333333333333
8	0.7	0.7	0.7	3FE6666666666666	3FE6666666666666
9	0.8	0.8	.	3FE9999999999999	
10	0.8	.	0.8		3FE999999999999A
11	0.9	0.9	.	3FECCCCCCCCCCCCC	
12	0.9	.	0.9		3FECCCCCCCCCCCCD

You can see .3^=.3 because one is `3FD3333333333333` and the other is `3FD3333333333334`. Although their printouts are both .3 in decimal, they are not equal. In this paper we talk about how SAS stores numeric values and does addition.

What is difference between .1+.2 and .3? Let's run the following program.

```
data s;
a=.1+.2-.3;
b=2**(-54);
put a= hex16. a= b=;
run;
```

We have the following output in the log file.

```
a=3C90000000000000 a=5.551115E-17 b=5.551115E-17
```

The exact difference is 2**(-54). Its decimal approximation is 5.55E-17. And its hex16 expression is `3C90000000000000`.

How does SAS store numeric values?
Now we talk about how SAS stores numbers and how SAS does operations. Readers can also find detailed information for storing numbers in [4].

As we know usually we use base 10 decimal to represent a number in our daily life. There is a decimal point, digits preceded decimal point represent digits with 10 powers, and digits behind decimal point represent digits with negative 10 powers. Therefore,

```
34.56=3*10**1+4*10**0+5*10**(-1)+6*10**(-2)
```

However in SAS numbers are expressed in base 2. To represent a number as binary decimal is similar to that express it as decimal.

```
34.56=2**5+2**1+2**(-1)+2**(-5)+...  (decimal)
      =10**101+10**1+10**(-1)+10**(-101)+...  (binary)
```

Therefore 34.56(base 10)=100010.10001….(base 2)

To find binary expression for 34.56, we can also use the following way to handle its integer portion and fraction portion, then put them together.

34/2= 17 r 0
17/2= 8 r 1
8/2=4 r 0
4/2=2 r 0
2/2=1 r 0
½=0 r 1

Taking the reminder from bottom to top, we get integer 100010.

.56*2=1.12
.12*2=0.24
.24*2=0.48
.48*2=0.96
.96*2=1.92
.92*2=1.84

Taking the integer part from top to bottom, we get 100011… Therefore the whole number is 100010.100011… These expressions (34.56 in base 10 or 100010.100011… in base 2) are called *fixed-point representation*.

However SAS, as the same as most computer system, does not use fixed-point representation numeric values. Instead, SAS uses 64 bits Floating-Point representation (the IEEE_754 Standard, double precision) to store numeric values. Over the years several different floating-point representations have been used in computers; however, for the last ten years the most commonly encountered representation is that defined by the IEEE

754 Standard. The advantage of floating-point representation over fixed-point representation is that it can support a much wider range of values (magnitude), but it sacrifices some precision. Let's see how IEEE 754 standard saves numeric values. For the number 34.56, we first normalize it as the following.

100010.1000111…=10**101*1.000101000111

Where 10 is base (2 in base 10) 101 is power (5 in base 10) and the remainder has integer part 1 and other fraction portion which is called *mantissa*. Notice that a number can have sign, also the power can have sign.

IEEE 754 standard (double precision) uses 64 bits to save a numeric value. The first bit (bit 63, left most) is for sign (0 for positive and 1 for negative), bit 62-52 (eleven bits) are for exponent field. We can omit the saving of power sign by adding power with a constant, called *Bias*. The IEEE 754 Standard uses 1023 (in decimal, or 11 1111 1111, 10 bits, in binary) as Bias. Bit 51-0 (total 52 bits, or 6.5 bytes) are for mantissa. Notice that the integer portion of the reminder, 1, is not saved. It is called *hidden bit*. Let us consider number 1.

1=10**0*1.0000000000+

Therefore 1 in floating-point representation is
0011 1111 1111 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000

The hexadecimal representation of this number is
3FF0000000000000

Let's run the following program to validate.

```
data _null_;
x=1;
put x= hex16. '(hex)  ' x= '(decimal)';
put x= binary64.;
run;
```

You can see the following printout in the log file.

```
x=3FF0000000000000 (hex)  x=1 (decimal)
x=0011111111110000000000000000000000000000000000000000000000000000
```

You can guess the floating-point representation for -1 is BFF0000000000000.

Now we discuss fraction number, the number that is less than 1. As we see, a number with terminating expansion in base 10 may become recurring decimal in binary representation. Since computer only store limited number of bits for a number, we have to consider how to handle last digit, rounding or truncating. IEEE 754 uses rounding in floating-point representation. For number 34.56, i.e. 10**101*1.000101000111… in binary, its power in normalization is 101

```
101+11 1111 1111=100 0000 0100

Therefore it is saved as

0100 0000 0100 0001 0100 0111...
```

Or 404147AE147AE148 in hex value. Its true binary expressing is recurring. The last digit is rounding. Next we see how 0.1 is saved. It is easy to see the binary expression for .1 is

0.000110011001100….=10**(-100)*1.1001 1001 1001…

Therefore its floating-point representation is

0011 1111 1011 1001 1001 1001 …..1010

Its hex value is

```
3FB999999999999A
```

How does SAS do addition?

Next we talk about operation. Actually we only talk about addition, and for positive numbers. We imitate how SAS does, or IEEE 754 standard does, additions for floating point expressions. If two numbers have the same exponent portion, (bits 63-52) then it is easy. We only have to add up their mantissas. As they both have hidden bit 1, the sum will be 10. So for normalization, we have to add 1 to exponent portion. Also we have to move decimal point one bit to the left for the sum.

Suppose two mantissas of two numbers are 1101 and 1011. They represent 1.1101 and 1.1011.

```
    1.1101
+   1.1011
-------------
   11.1000
```

11.1000=10**1*1.11000

Therefore when doing addition, we can do as following: add 1 to exponent portion, add up two mantissas and move resulting mantissa one bit to the right, and do rounding.

If two numbers have different exponent portions, we have first to adjust decimal point.

Let us see our example .1+.2. First we show you how we do additions on fixed point expression of numbers.

0.1(dec)= 0.000110011001100….
0.2(dec)= 0.00110011001100….

```
        0.000|110011001100
+       0.001|10011001100
-------------------------------------------
=       0.010|0110011001100
=10**(-10)*1.001100110011…
```

As we know its floating-point representation is 3FD3333333333333. We can see relations between IEEE standard and real numbers.

.1 in IEEE `3FB999999999999A > .1 in real number 3FB9999999999999999...`
.2 in IEEE `3FC999999999999A > .2 in real number 3FC9999999999999999...`
.3 in IEEE `3FD3333333333333 < .3 in real number 3FD3333333333333333...`

It looks .1+.2>.3, but we still have to do real addition. Now let us see how to do addition for floating-point representation. The following are floating point expressions for .2 and .1.

.2	0011 1111 1100|100110011001……1001 1010	`3FC999999999999A`
.1	0011 1111 1011|100110011001……1001 1010	`3FB999999999999A`

Since they have different power field, in order to do addition, we first have to adjust decimal point. Since .2>.1, we keep representation of .2 not changed, but change .1. We add one to power of .1 and move mantissa of .1 one bit to the right. At this time we have to get back hidden bit of .1, as the following.

```
        .2      0011 1111 1100|1001 1001 1001 1001 1001 ... 1001 1001 1001 1001 1001 1010
+       .1      0011 1111 1100|1100 1100 1100 1100 1100 ... 1100 1100 1100 1100 1100 1101
-------------------------------------------------------------------------------------------
                              1 0110 0110 0110 0110 0110 ... 0110 0110 0110 0110 0110 0111
```

Notice that for this expressions, the hidden bit for .2 (bigger number) is still 1, but that for .1 (smaller number) is 0.

Now we add up their mantissas. In general there are two possibilities: either there is 1 at bit 53, or not. In our example, the bit 53 is 1. It must be added to hidden bit, which becomes 10. For normalization, 1 becomes new hidden bit, 0 goes to mantissa, (move mantissa one bit to the right and add 0 at beginning), and raise exponent field by 1. Therefore the sum becomes

```
0011 1111 1101|0011 0011 0011 0011 0011 0011 0011 0011 0011 0011 0011 0011 0011|1
```

At this time we have to do rounding. Finally the floating-point representation of .1+.2 will be 3FD333333333334. Therefore .1+.2^=.3. Further .1+.2>.3. Moreover the difference between .1+.2 and .3 is 2**(-54), or approximately `5.551115E-17`.

Let us see the example .1+.7^=.8.

```
.7      3FE6666666666666        0011 1111 1110 0110 0110 ... 0110
```

```
.1      3FB999999999999A        0011 1111 1011 1001 1001 ... 1010
.8      3FE999999999999A        0011 1111 1110 1001 1001 ... 1010

        .7      0011 1111 1110| 0110 0110 ... 0110
+       .1      0011 1111 1110| 0011 0011 ... 0011| 010
                                1001 1001 ... 1001
```

Since there is no 1 goes to hidden bit, we don't have to do anything.
Therefore .1+.7=3FE9999999999999 and .1+.7<.8

How to avoid problems

When doing analysis, for example, calculating mean or other statistic values, floating point expression has almost no effect. Comparing to .1, 5.55E-17 is really a small number. But for categorical values, it really matters. Also for comparison, it really matters. The results of comparison only take two values: true or false. AS we showed in the above, SAS cannot even do MERGE.

Now we talk about how to avoid problems. The main point is: avoiding using fraction operations.

1. Use integers to be levels of categorical variable, instead of fractions.
2. When using fraction, make sure fractions are the same as they are stored in computer. If you use DO loop, use hard code fraction values, rather accumulate adding, as the following code.

```
data  s;
 do i=.1 ,.2,.3,.4,.5,.6,.7,.8,.9;
output;
end;
run;
```

3. use integer division,

```
data  s;
do j = 1 to 9;
i=j/10; output;
end;
run;
```

4. You can use accumulate adding, but use function ROUND to adjust them.

```
data  s;
do j = .1 to .9 by .1;
i=round(j,.1);
 output;
end;
```

In order to make sure the above approaches work, you can use the following simple program to check.

```
data t;
```

```
set s;
if i in (.1 .2 .3 .4 .5 .6 .7 .8 .9);
proc print;
run;
```

You can also modify comparison.

5. Use round function in comparison. Use IF round(a)=.1; instead of IF a=.1;
6. For greater than or less than comparison, we can use round function too. We can also use tail number method. In stead of IF a<.4; write IF a<.4-.000001; Instead of IF a<=.4; write IF a<=.4+.000001;
7. for equal comparison, we can also use function ABS with small tail value: Instead of IF a=b; write IF ABS(a-b)<.00001;

Magnitude and precision

Now we talk about magnitude and precision. The magnitude is that the biggest number that SAS can store. The precision the biggest number (integer) that all integers less than it can be saved exactly. Theoretically the biggest number that SAS can store is the following.

7fff ffff ffff ffff

or

(2-2**(-52))*2**2**10

For IEEE 754 standard, it should be OK. But for SAS system, in reality, the biggest number is a little smaller than this. If you run the following program, you will get error.

```
32   data s;
33   a=2**1024;
NOTE: Invalid argument(s) to the exponential operator "**".
34   put a hex16.;
35   run;
```

But if you run the following program, it is OK.

```
36   data t;
37   a=2**1023*(2-2**(-52));
38   put a= hex16. @20 a;
39   run;

a=7FEFFFFFFFFFFFFF 1.797693E308
```

From the above analysis we can see, the magnitude is `2**1023*(2-2**(-52))`, and for precision, it is 2**53. Let us run the following program.

```
data s;
a0=2**53-2;
a1=2**53-1;
a2=2**53;
a3=2**53+1;
```

```
b=a1-a0;c=a3-a2;
proc print;
run;
```

Obs	a0	a1	a2	a3	b	c
1	9.0072E15	9.0072E15	9.0072E15	9.0072E15	1	0

Length attribute

The range of LENGTH attribute for numeric values is 3-8, counted in bytes. So if you define length of a numeric variable to be 2 or 10, SAS will complain. The default length attribute for a numeric variable is 8. It is not related with assignment or INPUT statement. The only way to set length attribute for a numeric variable is to use LENGTH statement. By setting length smaller than 8, we can save computer resource. So what does the length mean? What do different length settings affect operations and results?

As we know, IEEE 754 expression uses one bit for sign, 11 bits for exponent field and 52 bits for mantissa. Different length setting will not change bits for sign and exponent field, but will reduce bits for mantissa. The following is correspondence.

Length setting	3	4	5	6	7	8
Space for mantissa (bits)	12	20	28	36	44	52

Different length attribute settings have similar magnitude (same number of bits for power field), but different accuracy (different number of bits for mantissa). You can also use function CONSTANT() to find the largest integer that SAS can save as exact integers for all integers less than it in absolute value. For example CONSTANT('exactint',3)=8192 (=2**13), which tells us that using length=3, all integers less than 8192 have an exact representation. But if the integer is more than 8192(=2**13), there will possible some accuracy problem. For example, SAS saves 8193 as 8192, 8195 as 8194, etc. Let us run the following program.

```
DATA s;
LENGTH a b 3;
a=8192; b=8193;
DATA t;
SET s;
PUT a= b=;
IF a=b THEN PUT "They are equal.";
RUN;
```

We can see the following printout in the log file.

```
a=8192 b=8192
They are equal.
```

Notice that length setting for numeric variable is in effect only for saved data set, but not in PDV (Program Data Vector) that SAS is working on. So if we put two PUT statements in the first DATA step, we will see different results.

Now we know, if an integer is big, small length setting will affect the results. Different length setting will also affect the results on fraction operations. To see how different length settings affect the operation, let us first run the following program.

```
data s;
length a 3;
a=.1;
put a hex16.;
data t;
set s;
put a hex16.;
run;
```

We see the following output in the log file.

```
3FB999999999999A

3FB9990000000000
```

This example tells us:

1. In PDV, SAS always use 64 bits to handle numeric values, no matter what length setting is. When SAS writes data set, the length will be effect.
2. When writing to data set, SAS truncates the value according to length setting, rather than doing rounding.

The following program shows how length setting affects results.

```
data s;
length i 3;
do i=.1 to 1.6 by .1;
if i in (.1 .2 .3 .4 .5 .6 .7 .8 .9 1 1.1 1.2 1.3 1.4 1.5 1.6)
      then put 'in s  ' i=;
output;
end;
data t;
set s;
if i in (.1 .2 .3 .4 .5 .6 .7 .8 .9 1 1.1 1.2 1.3 1.4 1.5 1.6)
      then put 'in t  ' i=;
proc print;
run;
```

We can see the following output in the log file.

```
in s  i=0.1
in s  i=0.2
in s  i=0.4
in s  i=0.5
in s  i=0.6
in s  i=0.7
in s  i=1.2
in s  i=1.3
```

```
in t  i=0.5
in t  i=1.5
```

Here in the first data step, it is the same as we discussed. However in the second data set it is totally different. If a number is recurring, then in its length=3 representation, bits 44-0 are all zero. But in length=8 representation, for a recurring number usually they are not zero. Therefore, they are not equal We only have to check terminating decimals .5 (3FE0000000000000),1 (3FF0000000000000) and 1.5 (3FE8000000000000). The following are corresponding values of i.

	Before saving	after saving (come back to PDV)
.5	3FE0000000000000	3FE0000000000000
1	3FEFFFFFFFFFFFFF	3FEFFF0000000000
1.5	3FF8000000000001	3FF8000000000000

Therefore only two values are equal.

In this example, we compare value with length=3 with values with length=8. Next we compare values with the same length, say length=3. See the following example.

```
data s;
length i a 3;
do i=.1 to 1 by .1;
a=i;
output;
end;
data t;
length i b 3;
do i=.1 ,.2,.3,.4,.5,.6,.7,.8,.9,1;
b=i;
output;
end;
data p;
merge s t;
by i;
a1=put(a,hex16.);
b1=put(b,hex16.);
proc print;
var i a b a1 b1;
run;
```

Obs	i	a	b	a1	b1
1	0.09999	0.09999	0.09999	3FB9990000000000	3FB9990000000000
2	0.19998	0.19998	0.19998	3FC9990000000000	3FC9990000000000
3	0.29999	0.29999	0.29999	3FD3330000000000	3FD3330000000000
4	0.39996	0.39996	0.39996	3FD9990000000000	3FD9990000000000
5	0.50000	0.50000	0.50000	3FE0000000000000	3FE0000000000000
6	0.59998	0.59998	0.59998	3FE3330000000000	3FE3330000000000
7	0.69995	0.69995	0.69995	3FE6660000000000	3FE6660000000000
8	0.79993	0.79993	0.79993	3FE9990000000000	3FE9990000000000
9	0.89990	0.89990	0.89990	3FECCC0000000000	3FECCC0000000000

```
 10    0.99988    0.99988      .          3FEFFF0000000000                    .
 11    1.00000      .        1.00000                    .     3FF0000000000000
```

We can see, the results are totally different from that with length 8.

The above discussion is based on PC window. Since UNIX also use IEEE standard, the results are the same. Also VAX/VMS used different representation, different bit assignments for exponent, bias and mantissas, but it also use rounding, rather truncating, and the results should not be affected. However, IBM uses different method: truncating. The result is different from this paper. The following is output form IBM mainframe for the second program.

Obs	I	A	B	A1	B1
1	0.1	0.1	0.1	4019999999999999	4019999999999999
2	0.2	0.2	.	4033333333333332	
3	0.2	.	0.2		4033333333333333
4	0.3	0.3	.	404CCCCCCCCCCCB	
5	0.3	.	0.3		404CCCCCCCCCCCC
6	0.4	0.4	.	4066666666666664	
7	0.4	.	0.4		4066666666666666
8	0.5	0.5	.	407FFFFFFFFFFFD	
9	0.5	.	0.5		4080000000000000
10	0.6	0.6	.	4099999999999996	
11	0.6	.	0.6		4099999999999999
12	0.7	0.7	.	40B3333333333332F	
13	0.7	.	0.7		40B3333333333333
14	0.8	0.8	.	40CCCCCCCCCCCCC8	
15	0.8	.	0.8		40CCCCCCCCCCCCC

Reference

[1] Dealing with Numeric Representation Error in SAS Applications, http://support.sas.com/techsup/technote/ts230.html
[2] Floating point, http://en.wikipedia.org/wiki/Floating_point
[3] IEEE-754 Floating-Point Conversion, http://babbage.cs.qc.edu/IEEE-754/Decimal.html
[4] TS-DOC: TS-654 - Numeric Precision 101, http://support.sas.com/techsup/technote/ts654.pdf.

Contact Information

The author welcomes feedback via email at: **shaojixu@yahoo.com**