

Paper 275-2011

Why .1 + .1 Might Not Equal .2 and Other Pitfalls of Floating-Point Arithmetic

Clarke Thacher, SAS Institute Inc., Cary, NC

ABSTRACT

All numeric values in SAS® are represented as 64-bit floating point numbers. Floating-point representation can store a wide range of values. It is very important for every programmer using SAS to understand how floating point arithmetic works and to be aware of the limits of floating point arithmetic.

In this paper, we give a detailed explanation of floating point representation and demonstrate some of the common mistakes that can be made by SAS programmers. We also offer suggestions for avoiding these traps.

We show:

- Why .1 + .1 may not equal .2
- How the rules of algebra may not apply to floating point numbers
- Why it isn't a good idea to store a twenty digit identifier in a floating point number.
- How sorting may affect numeric results

We demonstrate these principles through a series of easy-to-understand SAS programs. This paper requires no previous knowledge, but it should be valuable to every SAS programmer.

The examples in this paper have been run on SAS 9.2 (TS2M3) running on Windows XP, unless otherwise noted.

INTRODUCTION

It is essential for any programmer working with SAS software to understand the strengths and weaknesses of floating-point representation. This paper discusses, in detail, how floating point numbers are represented. Next, this paper provides a description of how two numbers are added. Finally, this paper discusses some of the features of the SAS language that can be used to deal with the limits of floating-point representation. This paper includes several SAS programs that demonstrate the principles as they are discussed. Most of this paper deals with the numeric representation used on Microsoft Windows, Linux, and UNIX operating systems.

NUMERIC REPRESENTATION

There are only two fundamental data types in the SAS programming environment, character and numeric. SAS numbers are represented in the “native” double-precision floating point of the host computer. All mathematical operations are performed as floating point operations.

Floating-point representation is a very powerful method to store an extended range of values with finite precision in a relatively compact amount of memory. Floating-point representation is similar in concept to scientific or exponential representation. Current computer architectures support efficient and accurate hardware implementations of floating point that can provide high-speed computations to solve very large and complex numerical problems.

Floating-point systems represent values as the product of a base value raised to a power multiplied by a fractional value: $(sign) * (base^{exponent} * mantissa)$. The sign indicates whether the number is positive or negative. The base defines the number system that is used for the calculations. The most common floating-point bases are powers of two, which “fit” well into binary representation. The exponent is the power (positive, negative, or zero) that the base is raised to. The mantissa is a value, usually expressed as the sum of a series of fractions of the base.

While binary representations fit efficiently into computer memory, they do not match the way most people do arithmetic. Most people calculate with decimal numbers and may run into trouble when converting to and from floating-point binary (or hex) representation. To address this, a decimal floating-point standard was defined as part of IEEE 754-2008. While this standard can be implemented via software or hardware, only a few hardware implementations are available.

The mainframe uses a floating-point format that was introduced in 1964 with the release of the IBM System/360. Floating-point values are represented by a sign, exponent, and a 56 bit mantissa, which can be represented by the

formula: $N = (Sign) * (16^{exponent-64} * (\frac{d_1}{16^1} + \frac{d_2}{16^2} + \dots + \frac{d_{14}}{16^{14}}))$. The exponent range is from 0 to 127 which gives a range of 10^{-78} to 10^{75} .

IEEE 754 BINARY FLOATING-POINT REPRESENTATION

With the exception of the mainframe, all of the systems that SAS runs on use the IEEE 754 binary floating-point representation. The IEEE 754 floating-point standard was first defined in 1985 by a cross-industry committee that sought to improve the floating-point implementations that were available at the time. Before IEEE 754, there were over 40 different floating-point formats, and each format defined widely differing behaviors. This inconsistency made the writing of reliable, portable, efficient and accurate mathematical software extremely difficult and expensive.

The rest of this paper focuses on the IEEE 754 double-precision floating-point standard.

The IEEE 754 double-precision binary floating-point format used by SAS on Windows, Linux, and UNIX platforms can be represented by the formula: $N = (-1 * sign) * (2^{exponent-1023} * (1 + \frac{b^1}{2^1} + \frac{b^2}{2^2} \dots + \frac{b^{52}}{2^{52}}))$

This diagram shows how these values are stored in a double-precision number.

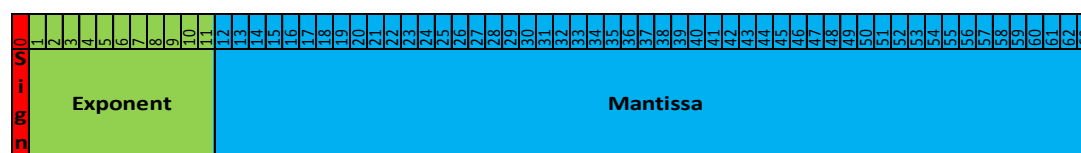


Figure 1 IEEE 754 Double-Precision Binary Floating-Point Format

The **sign** is stored as a single bit at the beginning of the number. Positive numbers have a zero sign bit and negative numbers have the sign bit set to one.

The **exponent** is represented as an unsigned 11-bit binary value. The exponent representation is called biased representation because, in order to evaluate it, you must subtract 1023. As a result, normal double-precision values have a range of $\approx \pm 1.7976931348623157 \times 10^{308}$. The values closest to zero are $\pm 2^{-1022} \approx \pm 2.2250738585072020 \times 10^{-308}$.

The **mantissa** is stored in the last fifty-two bits of the 64-bit floating point number. The *mantissa* may also be referred to as the *fraction* or *significand*. The first bit of the mantissa is always set to one. This bit is not stored and is called the *hidden bit*. Each successive bit in the mantissa represents a successively smaller fraction of two.

This format can be illustrated with few simple examples. The simplest number that we can show is one, which is represented by the formula: $1 = (1) * (2^{1023-1023} * 1)$. Because the number is positive, the sign bit is **0**. The exponent is the binary representation of 1023 (**0111111111**). The hidden bit is (as always) **1** and the remainder of the mantissa is set to zeros. Here is the bit representation:



Figure 2 IEEE Double-Precision Representation of 1.0

Similarly, two is represented as $2 = (1) * (2^{1024-1023} * 1)$. The sign bit is set to **0**, and the exponent is 1024 (**1000000000**). Again, the significand is set to all zeros.



Figure 3 IEEE Double-Precision Representation of 2.0

Negative two is almost exactly the same as positive two with the only difference in the sign bit.



Figure 4 IEEE Double-Precision Representation of -2.0

These examples illustrate that all exact powers of two are stored with a mantissa ending with 52 zeros. The only difference will be in the sign bit and exponent.

[illegible]

This floating-point format enables us to exactly represent every integer from 1 to 2^{53} (9,007,199,254,740,992).

[illegible]

3

Table 1 Floating-Point Numbers Expressed as Exponents and Bit Fields

[illegible]

We can take the previous program and modify it to display our input values as polynomial expressions.

```

/*
  Parse a double precision number and express it as a polynomial.
*/
ods rtf;
data nums;
  keep n sign exponent mantissa;
  length bits $ 64 sign $ 1 mantissa $ 52;
  input n @@;
  bits = put(n,binary64.);
  sign = substr(bits,1,1);
  exponent=input(substr(bits,2,11),binary12.)-1023;
  mantissa = substr(bits,13,52);
  file print;
  put n '= ' @;
  e = -exponent;
  put '1/2^' e @;
  do i=1 to 52;
    e = -exponent + i;
    if (substr(mantissa,i,1)='1') then put '+ 1/2^' e @;
  end;
put;
  format n best19.;
cards;
1 2 3 .5 .75 1.5 .1
;
ods rtf close;

```

This program produces this output:

Table 2 Numeric Values Expressed as Polynomial Fractions

$$\begin{aligned} 1 &= 1/2^0 \\ 2 &= 1/2^{-1} \\ 3 &= 1/2^{-1} + 1/2^0 \\ 0.5 &= 1/2^1 \\ 0.75 &= 1/2^1 + 1/2^2 \\ 1.5 &= 1/2^0 + 1/2^1 \end{aligned}$$

$$0.1 = 1/2^4 + 1/2^5 + 1/2^8 + 1/2^9 + 1/2^{12} + 1/2^{13} + 1/2^{16} + 1/2^{17} + \\ 1/2^{20} + 1/2^{21} + 1/2^{24} + 1/2^{25} + 1/2^{28} + 1/2^{29} + 1/2^{32} + 1/2^{33} + \\ 1/2^{36} + 1/2^{37} + 1/2^{40} + 1/2^{41} + 1/2^{44} + 1/2^{45} + 1/2^{48} + 1/2^{49} + \\ 1/2^{52} + 1/2^{53} + 1/2^{55}$$

Looking at the results, it is interesting to note that the simple decimal value of **0.1** produces a complicated bit pattern in the mantissa. This complicated bit pattern points out one of the fundamental limitations of any floating-point representation. Most decimal fractional values cannot be exactly represented as a sum of a series of base 2 fractions. This limitation is the same problem that you encounter when you try to express the fraction $\frac{1}{3}$ as a decimal value. The decimal result repeats forever (0.33333...).

In the case of the binary representation of .1, we find that the mantissa bit pattern "0011" repeats. The last bit in the mantissa breaks the pattern because of floating-point rounding rules. In fact, we can never exactly store $\frac{1}{10}$ as a binary floating-point number. The polynomial approximation of the decimal value 0.1 can be expressed as:

$$0.1 = \frac{3}{2^5} + \frac{3}{2^9} + \frac{3}{2^{13}} + \frac{3}{2^{17}} + \frac{3}{2^{21}} + \frac{3}{2^{25}} + \frac{3}{2^{29}} + \frac{3}{2^{33}} + \frac{3}{2^{37}} + \frac{3}{2^{41}} + \frac{3}{2^{45}} + \frac{3}{2^{49}} + \frac{3}{2^{53}} + \frac{1}{2^{55}}$$

This simplifies to:

$$\frac{3602879701896397}{36028797018963968}$$

Or (in decimal): 0.1000000000000000055511151231257827021181583404541015625

PITFALL - YOU CAN'T FIT 20 DIGITS INTO A 15 DIGIT DATA TYPE

A SAS programmer was asked to read a data file that included a unique twenty-digit ID field. When she validated the data, she discovered that many records had incorrect ID fields. How did this happen?

This program demonstrates the problem:

```
/*
   This program demonstrates what happens when you read very large
   integral values into SAS numeric variables.
*/
DATA big_ids;
  KEEP   ID ID_STRING DIGITS NDIFF;
  LENGTH ID_STRING id2 $ 21 ;
  INPUT  ID 20. @1 ID_STRING $;
  DIGITS = length(ID_STRING);
  ndiff = 0;
  id2 = compress(put(id,21.));
  do i = 1 to length(id_string);
    if (substr(id_string,i,1) ^= substr(id2,i,1)) then ndiff = ndiff + 1;
  end;
CARDS;
11111111111111111111
22222222222222222222
33333333333333333333
44444444444444444444
55555555555555555555
66666666666666666666
77777777777777777777
88888888888888888888
99999999999999999999
;
ODS rtf;
PROC PRINT;
```

```

var ID_STRING ID DIGITS NDIFF;
format ID 21. ID_STRING $21.;
run;
ods rtf close;

```

The program reads our ID as a number using the 20. informat, and reads the same input field as a character field (ID_STRING). DIGITS is the number of digits in the ID number. Then the program counts the number of places where our ID string differs from our numeric representation.

Table 3 Twenty Character Numeric Identifiers Expressed as Decimal Numbers

Obs	ID_STRING	ID	DIGITS	ndiff
1	11111111111111111111	11111111111111110656	20	4
2	22222222222222222222	22222222222222221312	20	3
3	33333333333333333333	33333333333333331968	20	4
4	44444444444444444444	44444444444444442624	20	3
5	55555555555555555555	5555555555555557376	20	4
6	66666666666666666666	6666666666666663936	20	3
7	77777777777777777777	7777777777777770496	20	4
8	88888888888888888888	8888888888888885248	20	3
9	99999999999999999999	10000000000000000000	20	20







These results show that we have lost several identifying digits from our ID number. The differences may not be considered numerically significant, but the difference could be disastrous for the project. Our programmer would do much better if the ID was kept as an ID string.

TRUNCATED NUMERIC REPRESENTATION USING THE LENGTH STATEMENT

The LENGTH statement allows you to save space on disk by storing a smaller portion of SAS numeric variables. By default, SAS numeric variables are eight bytes in length. The LENGTH statement enables you to set lengths from 3 to 7 bytes. When short numeric variables are read from disk, they are lengthened to eight bytes by padding the mantissa with zeros. All mathematical operations on numeric variables are conducted on 64-bit representations. When short numeric variables are stored on disk, the lower bits of the mantissa are lost.

Because the length of the exponent is the same for all lengths of SAS numbers, the range that can be represented is essentially the same. The shortened mantissa does reduce the range of values that can be exactly represented. For eight-byte floating point numbers, it is possible to exactly represent every integer from 0 to 2^{53} . As the storage length of the number is reduced, this range is reduced. The following table summarizes the range provided by each numeric length.

Table 4 Short Numeric Representation

Length	Bits in Mantissa	EXACTINT	Bit Layout							
			Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7	Byte 8
3	12	8,192								
4	20	2,097,152								
5	28	536,870,912								
6	36	137,438,953,472								
7	44	35,184,372,088,832								
8	52	9,007,199,254,740,992								

The column labeled **EXACTINT** shows the maximum value that can be represented as an integral value in a continuous range from 0. It is possible to exactly represent integral values greater than **EXACTINT**, but there will be “gaps” that cannot be exactly represented.

If you don't want to remember the value of **EXACTINT**, you can use the DATA step function **CONSTANT**. **CONSTANT("EXACTINT")** will return this value. The **CONSTANT** function will take an optional second argument, which specifies the length in bytes used to store the variable. The following SAS program demonstrates the use of the **CONSTANT** function.

```

/* This program demonstrates the use of CONSTANT("EXACTINT").
   In addition to calculating EXACTINT, this program also calculates the
   maximum number of decimal digits that can be reliably stored
   in a variable of that length. */
data EXACTINT;
  format exactint comma21. length 2.;
  do length=3 to 8;
    exactint = constant("EXACTINT",length);
    digits = floor(log10(exactint));
    output;
  end;
run;
ods rtf;
proc print noobs;
  var length exactint digits;
run;
ods rtf close;

```

This program produces this table:

Table 5 Values of **CONSTANT("EXACTINT") for Different Numeric Lengths**

length	exactint	digits
3	8,192	3
4	2,097,152	6
5	536,870,912	8
6	137,438,953,472	11
7	35,184,372,088,832	13
8	9,007,199,254,740,992	15

The next program demonstrates the effect of using shorter lengths for storing numeric variables. The program is similar to the previous program that demonstrated the limits of the default storage length. The new program adds five variables of length ranging from 3 to 7.

```
/* This program demonstrates the effect of using shorter lengths on decimal
precision. */
data shorts;
  keep d n n3 n4 n5 n6 n7 ;
  length N_STRING $ 16;
  length n3 3 n4 4 n5 5 n6 6 n7 7;
  do d=4 to 15;
    n = input(repeat('9',d-1),16.);
    n3 = n;      n4 = n;      n5 = n;      n6 = n;      n7 = n;
    output;
  end;
run;
ods rtf;
Title 'LENGTHs of 3, 4, 5';
proc print noobs data=shorts ;
  where d <= 11;
  format n n3 n4 n5 11.;
  var D n n3 n4 n5;
run;
Title 'LENGTHs of 6 and 7';
proc print noobs data=shorts ;
  where d > 11;
  format n n3 n4 n5 n6 n7 17.;
  var D n n6 n7;
run;
ods rtf close;
```

This program that specifies shorter lengths for storing numeric variables produces the following tables. Highlighted are the digits that lost precision.

Table 6 Precision Loss for Short LENGTH Numerics (3-5)

d	n	n3	n4	n5
4	9999	9998	9999	9999
5	99999	99984	99999	99999
6	999999	999936	999999	999999
7	9999999	9998336	9999992	9999999
8	99999999	99991552	99999936	99999999
9	999999999	999948288	999999488	999999998
10	9999999999	9999220736	9999998976	9999999968
11	99999999999	99992207360	99999940608	99999999744

Table 7 Precision Loss for Short LENGTH numerics (6-7)

d	n	n6	n7
12	999999999999	99999999992	999999999999
13	9999999999999	9999999999872	9999999999999
14	99999999999999	99999999998976	99999999999996
15	999999999999999	99999999991808	999999999999968

We have described 64-bit precision floating-point numbers as composed of 1 bit for the sign, 11 bits for the exponent, and 52 bits for the mantissa. These bit fields may differ in the way that they are stored in memory and on disk. Operating systems that use the Intel architecture (Windows and Linux) store numbers in “little endian” order. IBM AIX, HP-UX and Solaris on SPARC use “big endian” order. A big endian system stores the most significant bits at the lower memory locations. Little endian systems store in the reverse order. The following figures show how big and little endian systems store 64-bit floating point numbers.

[illegible]

```
ods rtf body="endian.rtf";
Title "SAS Numerical Representation on &SYSENDIAN endian machines";
/* demonstrate the way that numeric variables are encoded in memory
   The RB8. format writes the numeric variable in the "native"
   binary format */
data byteswap;
file print;
input n @@;
endian = "&sysendian";
string = put(n,rb8.);
put n 5.1 @10 n binary64./
    Endian @10 string $binary64. ;
cards;
1 -1 3
;
ods rtf close;
```

We ran this program on Windows (32bit) and AIX and got these results. The first three lines are from Windows and the last three are from AIX.

[illegible]

9

1 2 3 .5 .1

[illegible][illegible]
$$(A + B) = 2^{expA} * \left(1 + \frac{a^1}{2^1} + \frac{a^2}{2^2} \dots + \frac{a^{52}}{2^{52}}\right) + 2^{expB} * \left(1 + \frac{b^1}{2^1} + \frac{b^2}{2^2} \dots + \frac{b^{52}}{2^{52}}\right)$$

The first thing to do is to make the exponents of A and B agree. We restate the exponent of B as $2^{\text{expA}-Xb}$ where Xb is the difference in the exponents of A and B. We then transform the second term by dividing by 2^{Xb} , where Xb is the difference between the exponent values A and B.

$$(A + B) = 2^{\text{expA}} * \left(1 + \frac{a^1}{2^1} + \frac{a^2}{2^2} \dots + \frac{a^{52}}{2^{52}}\right) + 2^{\text{expA}} * \frac{\left(1 + \frac{b^1}{2^1} + \frac{b^2}{2^2} \dots + \frac{b^{52}}{2^{52}}\right)}{2^{Xb}}$$

This gives us:

$$(A + B) = 2^{\text{expA}} * \left(1 + \frac{a^1}{2^1} + \frac{a^2}{2^2} \dots + \frac{a^{52}}{2^{52}} + \frac{1}{2^{Xb}} + \frac{b^1}{2^{1+Xb}} + \frac{b^2}{2^{2+Xb}} \dots + \frac{b^{52}}{2^{52+Xb}}\right)$$

We then simplify our mantissa expression by adding the individual terms and, if necessary, normalizing our results.

Here's how we add $2 + 1$. $2 + 1$ is equivalent to $2^1 * (1) + 2^0 * (1)$. When we adjust the exponent in the second term to match the first, we get $2^1 * (1) + 2^1 * \left(\frac{1}{2}\right)$ which reduces to a floating-point normalized form of $2^1 * \left(1 + \frac{1}{2}\right)$.

The result of our calculation matches the normalized floating-point representation of 3.

Interestingly, adding $1 + 1$ might be considered to be more complicated because we need to adjust the exponent to return to normalized form: $(1 + 1) = 2^0 * (1) + 2^0 * (1)$. The exponents are the same, so we can simply add the mantissas. $(1 + 1) = 2^0 * (1 + 1)$. The first term of the mantissa exceeds one, so we need to increase the exponent by one and divide the mantissa by two. The final floating-point normalized representation is $2: 2^1 * (1)$.

ADDING TWO NUMBERS IN SAS, THE HARD WAY

Our next SAS program demonstrates how we might add two numbers represented in floating-point normalized form. We start with some macros.

The first macro, **crack_num**, takes a numeric value and returns the sign, exponent, and mantissa. The sign is a character variable that is either '0' (positive) or '1' (negative). The exponent is returned as a numeric value in the range. The mantissa is returned as a 53-byte character string composed of zeros and ones.

```
%macro crack_num(n,sign,exponent,mantissa);
  &sign = substr(put(&n,binary64.),1,1);
  &exponent = input(substr(put(&n,binary64.),2,11),binary12.)-1023;
  &mantissa = '1' || substr(put(&n,binary64.),13,52);
%mend;
```

The next macro, **make_num**, is the inverse of **crack_num**. It takes a sign, exponent, and mantissa and returns a SAS numeric value.

```
%macro make_num(sign,exponent,mantissa);
  input(catt("&sign",substr(put(&exponent+1023,binary12.),2,11),
    substr(&mantissa,2,52)),binary64.)
%mend;
```

In this program, we will treat the mantissa as a bit string. Division by powers of two can be accomplished by adding zeros to the beginning of the string. The bits at the end are shifted. We will put these bits into the bit_bucket variable. The shifter macro will handle the shifting, along with catching the bits that are shifted off the end in the bit bucket.

```
%macro shifter(shift,fract,bit_bucket);
  &bit_bucket = substr(&fract,53-&shift,&shift);
  &fract = repeat('0',&shift-1) || substr(&fract,1,53-&shift);
%mend;
```

The last macro will add the mantissa bits from right to left. In addition to the input bit strings, we have an additional variable for when the sum of the individual bits is greater than two. Addition is very simple. For each bit, our inputs are either 1 or 0, and we get either a 0 or 1 with carry set to either 0 or 1.

```
%macro add_fractions(m_a,m_b,m_sum,carry);
  /* add the bits right to left */
  &carry = 0;
  do n = 53 to 1 by -1;
    sum = &carry + input(substr(&m_a,n,1),1.) + input(substr(&m_b,n,1),1.);
    substr(&m_sum,n,1) = put(mod(sum,2),1.);
  end;
```

```

        carry = (sum > 1);
    end;
%mend;

```

For the sake of brevity, we will not handle inputs of zero, missing values, or negative values. Adding those checks into the program should be fairly easy.

```

data addition;
    keep a b sum expected diff bit_bucket;
    length A_fract B_fract S_fract bit_bucket $ 53 ;
    format a b sum best20.;
    input a b @@;
    expected = a + b;
    %crack_num(a,a_sign,a_exp,a_fract);
    %crack_num(b,b_sign,b_exp,b_fract);
    shift = a_exp - b_exp;
    if (shift < 0) then do;
        %shifter(-shift,a_fract,bit_bucket);
        r_exp = b_exp;
    end;
    else if (shift > 0) then do;
        %shifter(shift,b_fract,bit_bucket);
        r_exp = a_exp;
    end;
    else /* no shifting needed */
        r_exp = a_exp;
    %add_fractions(a_fract,b_fract,s_fract,carry);
    /* Do we need to shift one more time? */
    if (carry = 1) then do;
        r_exp = r_exp + 1;
        bit_bucket = catt(substr(s_fract,53,1),bit_bucket);
        s_fract = '1' || substr(s_fract,1,52);
    end;
    exp_bits = put(r_exp+1023,binary11.);
    sum = %make_num(0,r_exp,s_fract);
    diff = expected - (a + b);
cards;
1 1 1 2 .5 .5 .25 .75 1.01 .99 .001 .999
;
ods rtf;
proc print noobs;
    var a b sum expected diff bit_bucket;
run;
ods rtf close;

```

The program results in the following table.

Table 9 Results of Addition

a	b	sum	expected	diff	bit_bucket
1	1	2	2	0	0
1	2	3	3	0	0
0.5	0.5	1	1	0	0
0.25	0.75	1	1	0	00
1.01	0.99	2	2	0	01
0.001	0.999	1	1	8.6736E-19	011111110

We see that our program does a pretty good job of simulating the way our machine's floating point arithmetic unit operates. The only place where we see a difference is in the last row. The IEEE 754 standard has a number of rules that deal with rounding when significant bits are shifted away. For more details on these rounding rules, consult the standard. We have included links to information about the standard at the end of this paper.

(A + B) + C DOES NOT NECESSARILY EQUAL A + (B + C)

Next, we will present a series of SAS programs that demonstrate some commonly encountered floating-point pitfalls. The next program shows that what we learned in algebra may not apply when we are working with floating-point arithmetic.

One of the first lessons that we learned in algebra was about the associative property. That property states that the addition or multiplication of a set of numbers is the same regardless of how the numbers are grouped in algebraic representation: **(A+B)+C=A+(B+C)**

At the same time, we learned about the distributive property which states that the sum of two numbers multiplied by a third number is equal to the first number multiplied by the third number added to the second number multiplied by the third. This property is represented by the equation **(A+B)*C=(A*C)+(B*C)**

If you are working with floating point numbers, you can't rely on these properties to be true. Consider the following program. We input three values, A B C. Next, we compute five variables that are the sums and products of our input values. Finally, we test for equality with our algebraic identities $(A_PLUS_B + c) = (A + B_PLUS_C)$, $((A_TIMES_B * c) = (A * B_TIMES_C))$ and $(A_PLUS_B * C) = (A_TIMES_C + B_TIMES_C)$. The **IFC** function will return "=" if our test is true and "!=" if our test is false. In the following program, our input values are **0.1, 0.2, and 0.3**.

```
data A;
  input A B C;
  A_PLUS_B = A + B;
  B_PLUS_C = B + C;
  A_TIMES_C = A * C;
  A_TIMES_B = A * B;
  B_TIMES_C = B * C;
  equals = ifc(((A_PLUS_B + c) = (A + B_PLUS_C)), ' = ', ' != ');
  put 'Associative' ' ' ( ' a ' + ' b ' ) + ' c equals a ' + ( ' b ' + ' c ' );
  equals = ifc(((A_TIMES_B * c) = (A * B_TIMES_C)), ' = ', ' != ');
  put 'Associative' ' ' ( ' a ' * ' b ' ) * ' c equals a ' * ( ' b ' * ' c ' );
  equals = ifc(((A_PLUS_B * C) = (A_TIMES_C + B_TIMES_C)), ' = ', ' != ');
  put 'Distributive' ' ' ( ' a ' + ' b ' ) * ' c equals
      ' ( ' a ' * ' c ' ) + ( ' b ' * ' c ' );

  cards;
  .1 .2 .3
;
```

This program products the following output:

```
Associative   ( 0.1 + 0.2 ) + 0.3 ^= 0.1 + ( 0.2 + 0.3 )
Associative   ( 0.1 * 0.2 ) * 0.3 ^= 0.1 * ( 0.2 * 0.3 )
Distributive  ( 0.1 + 0.2 ) * 0.3 ^= ( 0.1 * 0.3 ) + ( 0.2 * 0.3 )
```

The output illustrates how simple operations can lose some bits of precision. When we calculated A_PLUS_B, we had some bits shifted beyond the range of our floating-point representation, and they were lost when we stored into our 64-bit floating point number.

HOW SORTING MAY AFFECT YOUR RESULTS

Another algebraic identity that we learned was the commutative property: **A + B + C = C + B + A**. In other words, the sum of a series of numbers should be the same, no matter what order that you sort them. The commutative property is not necessarily true when you are dealing with floating point values.

First, we'll create a data set. The data set has 200 observations. For every positive value, we have a corresponding negative value, so we know that the sum of all of the values is zero.

```
data Pow2;
  pow2 = 1;
```

```

do i = 1 to 100;
  pow2 = pow2 * 2;
  val = pow2; rand = uniform(0);output;
  val = -pow2; rand = uniform(0);output;
end;
Put 'Range: ' val e16. ' to ' pow2 e16.;
run;

```

Next, we define a macro that will sort this data set by a variable and then run a DATA _NULL_ step to compute the sum of the variable **val** and print the value.

```

%macro sum_set(Set,val,key);
proc sort data=&set;by &key;run;
data _null_;
  set &set end=done;
  format sum best16.;
  retain sum;
  if (_n_ = 1) then sum = 0;
  sum = sum + &val;
  if (done) then put "Sorted by &key" @30 sum e16. ;
run;
%mend;

```

Then we run this macro against our data set, sorting our data in different ways.

```

%sum_set(pow2,val,rand);          /* Random order */
%sum_set(pow2,val,val);          /* order by val */
%sum_set(pow2,val,descending val); /* order by reverse val */
%sum_set(pow2,val,pow2);         /* order by absolute value */

```

This program produces these results:

```

Range: -1.267650600E+30 to  1.267650600E+30

Sorted by rand          -7.036874418E+13
Sorted by val           -2.814749767E+14
Sorted by descending val 2.814749767E+14
Sorted by pow2          0.000000000E+00

```

As you can see, the results are vastly different, ranging from -2.8E14 to +2.8E14. We got the expected answer only when we sorted by the absolute value of the value that we were summing.

What happened?

In all but the last example, we lost precision through addition. In many additions, the difference in the exponents was greater than the number of bits in our mantissa, and significant bits were shifted away. We got the expected answer only when the observations were ordered in a way that every positive observation was followed by a negative observation, resulting in an intermediate value of 0: $2^2 + (-2^2) + 2^3 + (-2^3) \dots$

Strange as it might seem, all of the results that we got should be considered within the range of acceptable answers. Because we had 200 observations in the range of $\pm 2^{100}$, any variation whose absolute value is less than $200 \cdot 2^{100-52}$ ($5.6E16$) should not be considered significant.

DOLLARS AND CENTS

If you are developing applications that perform financial calculations and these calculations depend on getting exact results to the last penny, you should be very careful if you use floating-point arithmetic. Remember that it is impossible to precisely represent .01 as a binary floating point value. The value is very close, but precision loss could lead to problems.

The next program illustrates the “gaps” in precisely representing decimals. We will create a data set that contains dollars and cents values from \$0.00 to \$100.00. We will have one variable, DOLLARS, that represents our value as a dollars and cents represented as fractions of a dollar. A second variable, CENTS, is integer value that represents the number of cents (one dollar is 100 cents). We then calculate the precision loss by taking the difference between CENTS and DOLLAR*100.

```

data pennies;
keep dollars cents diff;
format diff e15. dollars dollar5.2;
do d = 0 to 99;
  do c = 1 to 100;
    dollars = d+c/100;
    cents = d*100+c;
    diff = cents - dollars*100;
    output;
  end;
end;
run;
ods graphics / noANTIALIAS;
ods rtf;
Title 'Distribution of decimal conversion error';
Title2 'Range $0.01 to $100.00';
proc sgplot data=pennies;
  scatter y=diff x=dollars /MARKERATTRS=(size=1pt symbol=circlefilled);
run;
ods rtf close;

```

Here is the graph that shows the distribution of the precision loss when we use fractional values to represent cents.

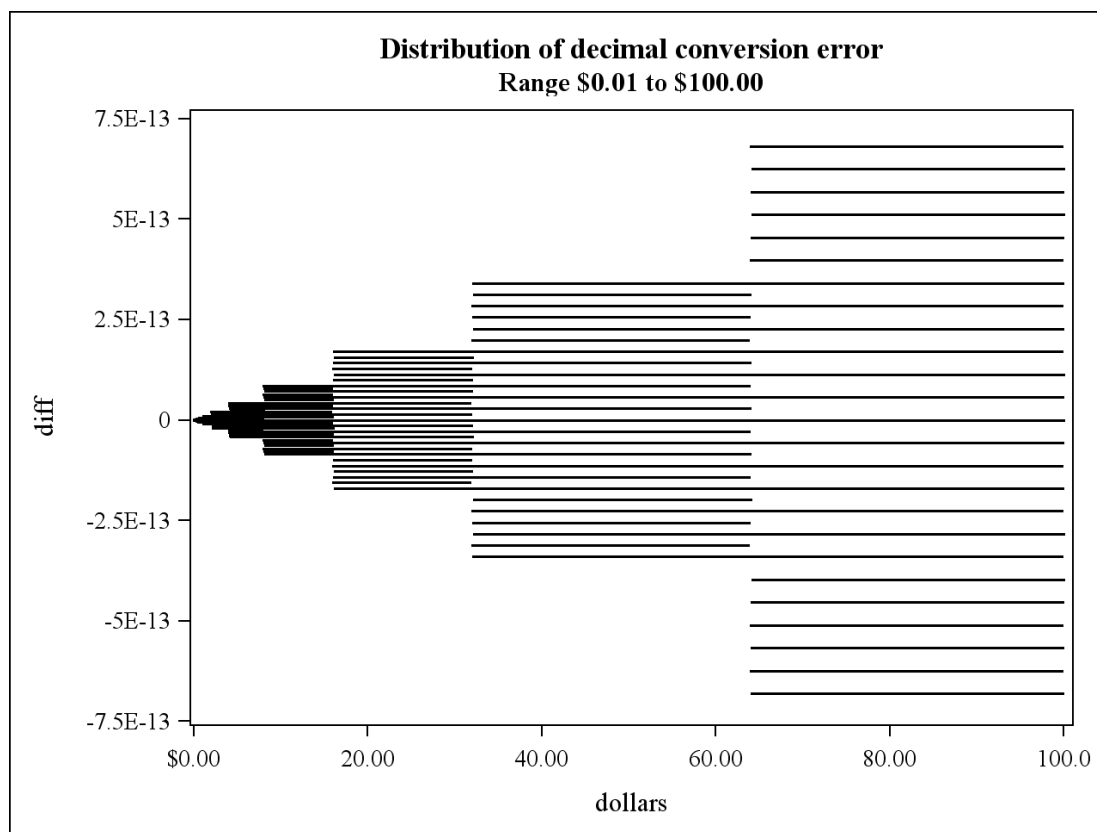


Figure 8 Distribution of Decimal Conversion Error

This graph shows that our rounding errors are not randomly distributed. They follow a definite and predictable pattern. The first thing that we can observe is that there are jumps in the range of DIFF when DOLLARS is equal to a power of two (2, 4, 8, 16, 32, 64). These jumps are because the exponent in our floating number represents a power of two. The round-off errors represent the portion of the mantissa that was shifted away.

MACEPS is a machine constant that is defined as the smallest positive value that can be added to one and results in a value greater than one. This constant is represented by the least significant bit in our mantissa, $\frac{1}{2^{52}}$. You can

access this value by using the `CONSTANT("MACEPS")` function. We can use this value when we examine loss of precision. Our next program takes the data set that we produced in the previous example, scales the data by MACEPS, and raises the exponent value of DOLLARS to 2

```
data pennies2;set Pennies;
  twoexp = 2**floor(log2(dollars));
  diff_eps = (diff/CONSTANT("MACEPS"))/twoexp;
run;
Title3 'Scaled by MACEPS and the exponent';
proc sgplot data=pennies2;
  scatter y=diff_eps x=dollars /MARKERATTRS=(size=1pt symbol=circlefilled);
run;
```

This program produces this scatter plot.

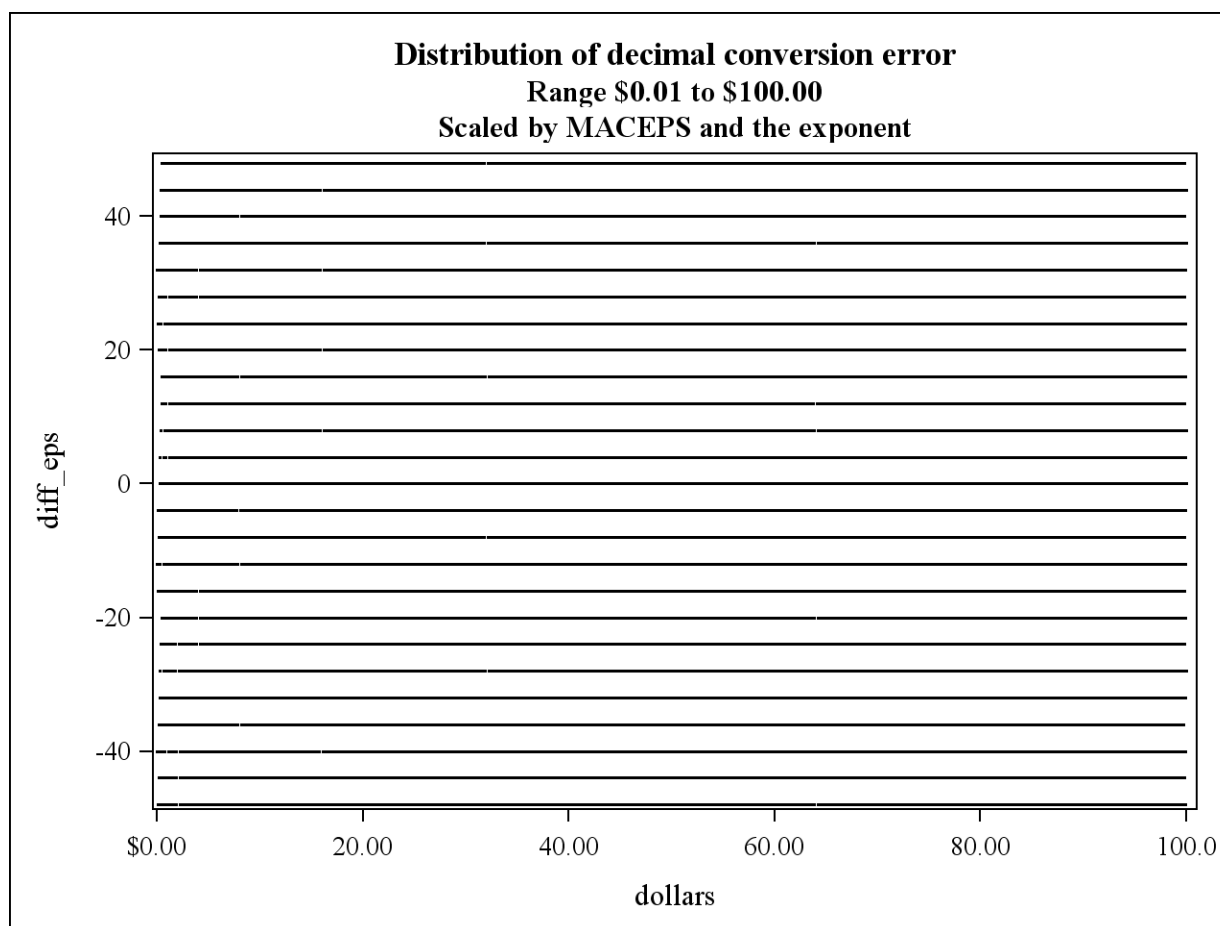


Figure 9 Distribution of Conversion Error After Scaling

We can see from this graph that the round-off errors from our calculations are regular and follow a predictable pattern. The round-off errors occurred when we evaluated " $d+c/100$ ". When d was 1 and c was 1, we got the $1 + 1/100$. The exponent of the first term is 0, and the exponent of the second term is 7, which results in seven bits of the mantissa being shifted away. IEEE rounding rules kept the result close to our expected result.

SAS FORMATS, INFORMATS

SAS provides a number of tools that enable you to deal with the limitations of floating-point representation.

SAS formats and informats handle the heavy lifting of converting numeric values to and from binary floating point representation. Almost every SAS format does a binary to decimal conversion operation. The challenge for the developers at SAS Institute who support this code has been to make these conversions as accurate as possible.

The downside of numeric formats is that they can give the illusion of precision or equality for values that are actually numerically distinct. If you print out a value with a format of **1.0**, the value may be exactly 1, or it may be anywhere between .5 and 1.499999999. This very simple program illustrates this issue.

```
data _null_;
  format n 1.;
  input n @@;
  put n= +1 @;
  cards;
1 .75 1.25 1.4999999999999999 .5
;
```

This program produces the following output:

```
n=1 n=1 n=1 n=1 n=1
```

FUNCTIONS FOR ROUNDING AND TRUNCATION

In our last program, we found that we can see small differences when dealing with decimal values, such as dollars and cents. One way to get more predictable results is to round or truncate calculated values. You might want to round a calculated price to two decimal places.

The following table lists the functions that are available in SAS 9.2 that can be used to truncate or round numbers to integer values. Most of the functions listed will truncate or round numerical results to integer boundaries. SAS programmers can use these functions to take the results of calculations that may have suffered from precision loss and convert the results to integer values using a variety of calculation options.

Table 10 Rounding and Truncation Functions

Function	Description
CEIL	Returns the smallest integer that is greater than or equal to the argument, fuzzed to avoid unexpected floating-point results.
CEILZ	Returns the smallest integer that is greater than or equal to the argument, using zero fuzzing.
FLOOR	Returns the largest integer that is less than or equal to the argument, fuzzed to avoid unexpected floating-point results.
FLOORZ	Returns the largest integer that is less than or equal to the argument, using zero fuzzing.
FUZZ	Returns the nearest integer if the argument is within 1E-12 of that integer.
INT	Returns the integer value, fuzzed to avoid unexpected floating-point results.
INTZ	Returns the integer portion of the argument, using zero fuzzing.
ROUND	Rounds the first argument to the nearest multiple of the second argument or to the nearest integer when the second argument is omitted.
ROUNDE	Rounds the first argument to the nearest multiple of the second argument and returns an even multiple when the first argument is halfway between the two nearest multiples.
ROUNDZ	Rounds the first argument to the nearest multiple of the second argument, using zero fuzzing.
TRUNC	Truncates a numeric value to a specified number of bytes.

MORE CONSTANTS

We have already used the `CONSTANT` function to return `MAXINT` and `MACEPS`. `CONSTANT` can be used to return a number of useful numeric values, as shown in the table below.

Table 11 Arguments for the CONSTANT Function with Descriptions of the Return Values

Constant	Description
'EXACTINT' <,nbytes>	Exact integer
'BIG'	The largest double-precision number
'LOGBIG' <,base>	The log with respect to base of BIG
'SQRTBIG'	The square root of BIG
'SMALL'	The smallest double-precision number
'LOGSMALL' <,base>	The log with respect to base of SMALL
'SQRTSMALL'	The square root of SMALL
'MACEPS'	Machine precision constant
'LOGMACEPS' <,base>	The log with respect to base of MACEPS
'SQRTMACEPS'	The square root of MACEPS
'E'	The natural base
'EULER'	Euler constant
'PI'	Pi

BIG, SMALL, LOGBIG, LOGSMALL and SQRTSMALL can be used to guard against range errors (overflow, underflow) in calculations. E, EULER and PI return the best machine representation of those mathematical constants. Using these constants is a better programming practice than coding these special values by “hand.”

THE COMPFUZZ FUNCTION

The COMPFUZZ function can be used to do “fuzzy” comparisons. Fuzzy comparisons will compare two values and return equality if the first two arguments are within a defined range. The function allows the careful programmer to account for precision issues. The following table gives a synopsis of COMPFUZZ. COMPFUZZ is not documented in the SAS 9.2 documentation, but it is discussed in SAS note [15553](#).

Table 12 COMPFUZZ Function

COMPFUZZ	This function is used for numerical error analysis and performs a fuzzy comparison of two numeric values.
Syntax	COMPFUZZ(value1, value2 <, fuzz <, scale>>)
Arguments	
value1	specifies the first of two numeric values to be compared.
value2	specifies the second numeric value to be compared.
fuzz	is a nonnegative numeric value that specifies the relative threshold for comparisons. Values greater than or equal to one are treated as multiples of the machine precision. To determine machine precision, see the CONSTANT('MACEPS') function. Default: 1024
scale	specifies the scale factor used in computing the threshold. Default: MAX (ABS (value1), ABS (value2))
Returns	
-1	if value1 < value2 – threshold
0	if ABS(value1 - value2) <= threshold
1	if value1 > value2 + threshold
	where threshold = fuzz * ABS(scale) /* if 0 <= fuzz < 1 */ threshold = fuzz * ABS(scale) * CONSTANT('MACEPS') /* if 1 <= fuzz < 1 / CONSTANT('MACEPS') */

Our next program uses COMPFUZZ to compare two sums and determine whether they are relatively equal. We start with the data set POW2 that we created in a previous example. POW2 has a variable **val**, which ranges in value from -1.267650600E+30 to 1.267650600E+30. We then create two new data sets that sort POW2 in two different ways. The last DATA step computes the sum of **val** when it is sorted in two different orders. When we have added all of our observations, we use COMPFUZZ to compare our sums. SCALE is the sum of the absolute

values of **val**. We set **fuzz** to the 2 times the number of observations, because that is the total number of terms in our sums.

```
proc sort data=pow2 out=pow2_s1 (rename=(val=val1));by descending val;run;
proc sort data=pow2 out=pow2_s2 (rename=(val=val2));by val; run;
data powsum; merge pow2_s1(keep=val1) pow2_s2(keep=val2) end=done;
retain sum1 0 sum2 0 nobs 0 scale 0 ;
scale = scale + abs(val1) + abs(val2);
nobs = nobs + 1;
sum1 = sum1 + val1;
sum2 = sum2 + val2;
if done then do;
    compare = compfuzz(sum1, sum2,nobs*2,scale);
    put sum1= sum2= nobs= scale= compare=;
end;
run;
```

The program produces this output:

```
sum1=2.8147498E14 sum2=-2.81475E14 nobs=200 scale=1.0141204E31 compare=0
```

Even though there is a wide range between **2.8147498E14** and **-2.81475E14**, they should be considered to be equal, within the context of our input values.

CONCLUSION

This paper has shown several important aspects of IEEE floating-point numbers. Some of the important things to remember are:

- They can represent a wide range of values, $\pm 1.7976931348623157 \times 10^{308}$.
- They are represented as binary fractions, so many decimal values cannot be exactly represented.
- Integer values can be exactly represented in the range $\pm 9,007,199,254,740,992$.
- Double-precision floating-point numbers have approximately 15 decimal digits precision.
- Simple mathematical operations may result in loss of significance.

RESOURCES

The IEEE 754 standard for binary floating-point arithmetic can be found at <http://grouper.ieee.org/groups/754/>.

Wikipedia has a fairly readable discussion of IEEE 754 at http://en.wikipedia.org/wiki/IEEE_754-1985.

Goldberg, David. "What Every Computer Scientist Should Know About Floating-Point Arithmetic", *ACM Computing Surveys*, March 1991: Available at <http://www.validlab.com/goldberg/paper.pdf>

SAS Note 230 has an excellent discussion of dealing with representation error in SAS programs: <http://support.sas.com/techsup/technote/ts230.html>

SAS Note 15553 describes the COMPFUZZ function. <http://support.sas.com/kb/15/553.html>

CONTACT INFORMATION

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

Clarke Thacher
SAS Campus Drive
SAS Institute Inc.
Clarke.Thacher@sas.com

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are trademarks of their respective companies.