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: \( (sign) \times (base^{exponent} \times mantissa) \). 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: \( N = (\text{sign}) \times (16^{\text{exponent}-64} \times \frac{d_1}{16^1} + \frac{d_2}{16^2} + \cdots + \frac{d_{15}}{16^{15}}) \). The exponent range is from 0 to 127 which gives a range of \( 10^{-28} \) to \( 10^{28} \).

### IEEE 745 Binary Floating-Point Representation

With the exception of the mainframe, all of the systems that SAS runs on use the IEEE 745 binary floating-point representation. The IEEE 745 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 745, 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 745 double-precision floating-point standard.

The IEEE 745 double-precision binary floating-point format used by SAS on Windows, Linux, and UNIX platforms can be represented by the formula: 
\[
N = (-1 \times \text{sign}) \times (2^{\text{exponent}-1023} \times (1 + \frac{b_1}{2^1} + \frac{b_2}{2^2} + \cdots + \frac{b_{52}}{2^{52}}))
\]

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

![IEEE 745 Double-Precision Binary Floating-Point Format](image)

**Figure 1 IEEE 745 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 \( \pm 1.7976931348623157 \times 10^{-308} \). 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: 
\[
1 = (1) \times (2^{1023-1023} \times 1)
\]

The exponent is the binary representation of 1023 (011111111111). The hidden bit is (as always) 1 and the remainder of the mantissa is set to zeros. Here is the bit representation:

![IEEE 745 Double-Precision Representation of 1.0](image)

**Figure 2 IEEE Double-Precision Representation of 1.0**

Similarly, two is represented as 
\[
2 = (1) \times (2^{1024-1023} \times 1)
\]

The sign bit is set to 0, and the exponent is 1024 (10000000000). Again, the significand is set to all zeros.

![IEEE 745 Double-Precision Representation of 2.0](image)

**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.

![IEEE 745 Double-Precision Representation of -2.0](image)

**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.

---

**SAS Global Forum 2011**

**Programming: Foundations and Fundamentals**
A slightly more complicated number is 3. Three is represented as $3 = (1) \times \left(2^{\frac{1024-1023}{2}} \times (1 + \frac{1}{2})\right)$ or more simply: $2 \times (1.5)$.

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

**WHAT ABOUT ZERO?**

You may have noticed a problem with the formula that we have been using to represent floating-point numbers: $N = (\text{sign}) \times (2^{\text{exponent} - 1023} \times (1 + \frac{b_1}{2} + \frac{b_2}{2^2} + \frac{b_3}{2^3} + \frac{b_4}{2^4} + ...))$. Because the hidden bit always evaluates to 1, it is impossible to exactly represent zero. Fortunately for everyone, the designers of the floating-point format realized this limitation and added a way to represent zero.

An exponent value of ‘00000000000’ ($2^{-1023}$) with an all zero fraction represents zero. Zero may be represented as either positive or negative.

**PROGRAM – DECONSTRUCTING A FLOATING POINT NUMBER WITH THE DATA STEP**

At this point, we can write a program that demonstrates the structure of floating-point representation. This program uses the BINARY64 format to produce the bit representation of a SAS numeric. It is a simple matter to extract the sign, exponent, and mantissa bits.

```sas
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);
  format n best19. ;
cards;
0 1 -1 2 3 .5 .75 9007199254740991 9007199254740992
; ods rtf;
proc print noobs ;
  var n sign exponent mantissa;
run;
ods rtf close;
```

This program produces this table:
Table 1 Floating-Point Numbers Expressed as Exponents and Bit Fields

<table>
<thead>
<tr>
<th>n</th>
<th>sign</th>
<th>exponent</th>
<th>Mantissa</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>-1023</td>
<td>00000000000000000000000000000000000000000000000000000</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>00000000000000000000000000000000000000000000000000000</td>
</tr>
<tr>
<td>-1</td>
<td>1</td>
<td>0</td>
<td>00000000000000000000000000000000000000000000000000000</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>00000000000000000000000000000000000000000000000000000</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>10000000000000000000000000000000000000000000000000000</td>
</tr>
<tr>
<td>0.5</td>
<td>0</td>
<td>-1</td>
<td>00000000000000000000000000000000000000000000000000000</td>
</tr>
<tr>
<td>0.75</td>
<td>0</td>
<td>-1</td>
<td>10000000000000000000000000000000000000000000000000000</td>
</tr>
<tr>
<td>1.5</td>
<td>0</td>
<td>0</td>
<td>10000000000000000000000000000000000000000000000000000</td>
</tr>
<tr>
<td>0.1</td>
<td>0</td>
<td>-4</td>
<td>1001000110011001100110011001100110011001100110011010</td>
</tr>
<tr>
<td>9007199254740991</td>
<td>0</td>
<td>52</td>
<td>111111111111111111111111111111111111111111111111111111</td>
</tr>
<tr>
<td>9007199254740992</td>
<td>0</td>
<td>53</td>
<td>000000000000000000000000000000000000000000000000000000</td>
</tr>
</tbody>
</table>

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

```plaintext
/* 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

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
0.1 = 1/2^4 + 1/2^5 + 1/2^8 + 1/2^9 + 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 \( \frac{10}{9} \) as a binary floating-point number. The polynomial approximation of the decimal value 0.1 can be expressed as:

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

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:

```sas
/*
This program demonstrates what happens when you read very large
integral values into SAS numeric variables.
*/
DATA big_ids;
KEEP ID ID_STRING DIGITS NDIF;
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;
111111111111111111111111111
22222222222222222222222222
33333333333333333333333333
44444444444444444444444444
55555555555555555555555555
66666666666666666666666666
77777777777777777777777777
88888888888888888888888888
99999999999999999999999999;
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

<table>
<thead>
<tr>
<th>Obs</th>
<th>ID_STRING</th>
<th>ID</th>
<th>DIGITS</th>
<th>ndiff</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11111111111111111111</td>
<td>11111111111111110656</td>
<td>20</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>22222222222222222222</td>
<td>22222222222222222222</td>
<td>20</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>33333333333333333333</td>
<td>33333333333333333333</td>
<td>20</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>44444444444444444444</td>
<td>44444444444444444444</td>
<td>20</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>55555555555555555555</td>
<td>55555555555555555555</td>
<td>20</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>66666666666666666666</td>
<td>66666666666666666666</td>
<td>20</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>77777777777777777777</td>
<td>77777777777777777777</td>
<td>20</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>88888888888888888888</td>
<td>88888888888888888888</td>
<td>20</td>
<td>3</td>
</tr>
<tr>
<td>9</td>
<td>99999999999999999999</td>
<td>100000000000000000000</td>
<td>20</td>
<td>20</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Length</th>
<th>Bits in Mantissa</th>
<th>EXACTINT</th>
<th>Bit Layout</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>12</td>
<td>8,192</td>
<td>![Exponent Mantissa]</td>
</tr>
<tr>
<td>4</td>
<td>20</td>
<td>2,097,152</td>
<td>![Exponent Mantissa]</td>
</tr>
<tr>
<td>5</td>
<td>28</td>
<td>536,870,912</td>
<td>![Exponent Mantissa]</td>
</tr>
<tr>
<td>6</td>
<td>36</td>
<td>137,438,953,472</td>
<td>![Exponent Mantissa]</td>
</tr>
<tr>
<td>7</td>
<td>44</td>
<td>35,184,372,088,832</td>
<td>![Exponent Mantissa]</td>
</tr>
<tr>
<td>8</td>
<td>52</td>
<td>9,007,199,254,740,992</td>
<td>![Exponent Mantissa]</td>
</tr>
</tbody>
</table>

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.

```sas
/* 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**

<table>
<thead>
<tr>
<th>Length</th>
<th>exactint</th>
<th>digits</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>8,192</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>2,097,152</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>536,870,912</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>137,438,953,472</td>
<td>11</td>
</tr>
<tr>
<td>7</td>
<td>35,184,372,088,832</td>
<td>13</td>
</tr>
<tr>
<td>8</td>
<td>9,007,199,254,740,992</td>
<td>15</td>
</tr>
</tbody>
</table>
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.

```sas
/* 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)**

<table>
<thead>
<tr>
<th>d</th>
<th>n</th>
<th>n3</th>
<th>n4</th>
<th>n5</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>9999</td>
<td>9998</td>
<td>9999</td>
<td>9999</td>
</tr>
<tr>
<td>5</td>
<td>9999</td>
<td>9998</td>
<td>9999</td>
<td>9999</td>
</tr>
<tr>
<td>6</td>
<td>9999</td>
<td>9998</td>
<td>9999</td>
<td>9999</td>
</tr>
<tr>
<td>7</td>
<td>99999</td>
<td>99989</td>
<td>99999</td>
<td>99999</td>
</tr>
<tr>
<td>8</td>
<td>999999</td>
<td>999899</td>
<td>999999</td>
<td>999999</td>
</tr>
<tr>
<td>9</td>
<td>9999999</td>
<td>9998999</td>
<td>9999999</td>
<td>9999999</td>
</tr>
<tr>
<td>10</td>
<td>99999999</td>
<td>99999999</td>
<td>99999999</td>
<td>99999999</td>
</tr>
<tr>
<td>11</td>
<td>999999999</td>
<td>999999999</td>
<td>999999999</td>
<td>999999999</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>d</th>
<th>n</th>
<th>n6</th>
<th>n7</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>999999999999</td>
<td>999999999999</td>
<td>999999999999</td>
</tr>
<tr>
<td>13</td>
<td>999999999999999</td>
<td>999999999999999</td>
<td>999999999999999</td>
</tr>
<tr>
<td>14</td>
<td>9999999999999999999</td>
<td>9999999999999999999</td>
<td>9999999999999999999</td>
</tr>
<tr>
<td>15</td>
<td>999999999999999999999</td>
<td>999999999999999999999</td>
<td>999999999999999999999</td>
</tr>
</tbody>
</table>
LITTLE ENDIAN AND BIG ENDIAN REPRESENTATION

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.

Table 8 Little and Big Endian Byte Layout

<table>
<thead>
<tr>
<th>Byte 8</th>
<th>Byte 7</th>
<th>Byte 6</th>
<th>Byte 5</th>
<th>Byte 4</th>
<th>Byte 3</th>
<th>Byte 2</th>
<th>Byte 1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Exponent</th>
<th>Mantissa</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte 1</td>
<td>Byte 2</td>
</tr>
<tr>
<td>Byte 8</td>
<td>Byte 7</td>
</tr>
</tbody>
</table>

This program demonstrates the SAS numerical representation on endian machines.

```sas
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 @@;
edian = "&sysendian";
string = put(n,rb8.);
put n 5.1 @10 n binary64./
   Endian @10 string $binary64. ;
cards;
1 -1 3
;
ods rtf close;
```

We have been using the BINARY64 format to examine the bit patterns of our numbers. The BINARY64 format returns a representation that is consistent across all IEEE platforms and hides the effects of the “endianness”. The PUT function allows us to access the number as a character variable. We use the SAS RB8 format to output the number in “native” format. The SYSENDIAN macro variable indicates whether the SAS program is running on a big or little endian system.

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.

```
1.0 LITTLE 0000000000000000000000000000000000000000000000000000000000000000
-1.0 LITTLE 0000000000000000000000000000000000000000000000000000000000000000
 3.0 LITTLE 0000000000000000000000000000000000000000000000000000000000000000
1.0 BIG    0000000000000000000000000000000000000000000000000000000000000000
-1.0 BIG   0000000000000000000000000000000000000000000000000000000000000000
 3.0 BIG    0000000000000000000000000000000000000000000000000000000000000000
```

Little endian representation can be made to match big endian representation by swapping the bytes around. The next SAS program demonstrates this technique.
/* Demonstrates the conversion of little endian format to big endian format */

data swap;
file print;
   length s $8 bits $64 endian $1;
   endian = "$SYSENDIAN";
   input n @@;
   s = put(n,rb8.);
   put n @5 'Native' @15 s $binary64.;
if endian = 'L' then do;
   /* swap bytes */
   s = reverse(s);
   put @5 'Swapped' @15 s $binary64.
   bits = put(s,$binary64.
cards;
1 2 3 .5 .1

This program produces the following output.

1
Native 0000000000000000000000000000000000000000000000000000000000000000
  Swapped 0011111111111111111111111111111111111111111111111111111111111111
2
Native 0000000000000000000000000000000000000000000000000000000001000000
  Swapped 0100000000000000000000000000000000000000000000000000000000000000
3
Native 0000000000000000000000000000000000000000000000001000000000000000
  Swapped 0100000000000000000000000000000000000000000000000000000000000000
0.5
Native 0000000000000000000000000000000000000000000000001110000000111111
  Swapped 0011111111111111111111111111111111111111111111111111111111111111
0.1
Native 1001101010011001100110011001100110011001100110011011100100111111
  Swapped 0011111111111111111111111111111111111111111111111111111111111111

WHAT’S MISSING? NANS

The IEEE 754 defines a set of numbers that are not treated as normal numbers. In fact, they are called “Not A Number”s, or NaNs. SAS uses NaNs to represent missing values.

An exponent value of ‘11111111111’ is used to represent two special classes of values. If the fraction is zeros, then the value is infinity. Non-zero fractions result in “Not a Number”s (NaNs). NaNs are similar to SAS missing values. Operations on NaNs result in NaNs. SAS missing values are represented as NaNs.

Because most SAS numeric formats detect missing values, we can’t use the BINARY64 format to peek at the way that missing values are represented. Instead, we can use the same technique that we used in the last program.

Running this program with missing values as input produces the following output on Windows.

.    Native 0000000000000000000000000000000000000000110100011111111111111111
  Swapped 1111111111111111110100010000000000000000000000000000000000000000
A    Native 0000000000000000000000000000000000000000101111101111111111111111
  Swapped 1111111111111111101111100000000000000000000000000000000000000000
Z    Native 0000000000000000000000000000000000000000101001011111111111111111
  Swapped 1111111111111111101001010000000000000000000000000000000000000000
-    Native 0000000000000000000000000000000000000000110100101111111111111111
  Swapped 1111111111111111110100100000000000000000000000000000000000000000

FLOATING POINT ADDITION

By now, you should understand how floating point numbers are represented. Now we need to do something with those numbers. We will show how to add two floating point numbers. This is a little more complicated than the way that you first learned to add.

If we want to calculate \( A + B \) where \( A \) and \( B \) both positive, we start with \( A \) and \( B \) in normalized floating point form:

\[
(A + B) = 2^{\exp_A} \times \left(1 + \frac{a_1}{2^1} + \frac{a_2}{2^2} + \cdots + \frac{a_{52}}{2^{52}}\right) + 2^{\exp_B} \times \left(1 + \frac{b_1}{2^1} + \frac{b_2}{2^2} + \cdots + \frac{b_{52}}{2^{52}}\right)
\]
The first thing to do is to make the exponents of A and B agree. We restate the exponent of B as $2^{\text{expA} - \text{Xb}}$ where $\text{Xb}$ is the difference in the exponents of A and B. We then transform the second term by dividing by $2^{\text{xB}}$, where $\text{xB}$ is the difference between the exponent values A and B.

\[
(A + B) = 2^{\text{expA}} \left( 1 + \frac{a^1}{2^1} + \frac{a^2}{2^2} + \cdots + \frac{a^{52}}{2^{52}} \right) + 2^{\text{expA}} \left( \frac{1 + b^1}{2^1} + \frac{b^2}{2^2} + \cdots + \frac{b^{52}}{2^{52}} \right)
\]

This gives us:

\[
(A + B) = 2^{\text{expA}} \left( 1 + \frac{1}{2^1} + \frac{a^2}{2^2} + \frac{a^{52}}{2^{52}} + \frac{1}{2^{\text{xB}}} + \frac{b^1}{2^1 + \text{xB}} + \frac{b^2}{2^2 + \text{xB}} + \cdots + \frac{b^{52}}{2^{52 + \text{xB}}} \right)
\]

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

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: $(1 + 1) = 2^0 \times (1) + 2^0 \times (1)$. The exponents are the same, so we can simply add the mantissas. $(1 + 1) = 2^0 \times (1 + 1)$. 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 $2^1 \times (1)$.

### 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(quot('&sign','&mantissa'),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_frc B_frc S_frc bit_bucket $ 53 ;
  format a b sum best20.;
  input a b @@;
  expected = a + b;
  %crack_num(a,a_sign,a_exp,a_frc);
  %crack_num(b,b_sign,b_exp,b_frc);
  shift = a_exp - b_exp;
  if (shift < 0) then do;
    %shifter(-shift,a_frc,bit_bucket);
    r_exp = b_exp;
  end;
  else if (shift > 0) then do;
    %shifter(shift,b_frc,bit_bucket);
    r_exp = a_exp;
  end;
  else /* no shifting needed */
    r_exp = a_exp;
  %add_fractions(a_frc,b_frc,s_frc,carry);
  /* Do we need to shift one more time? */
  if (carry = 1) then do;
    r_exp = r_exp + 1;
    bit_bucket = catt(substr(s_frc,53,1),bit_bucket);
    s_frc = '1'||substr(s_frc,1,52);
  end;
  exp_bits = put(r_exp+1023,binary11.);
  sum = %make_num(0,r_exp,s_frc);
  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

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>sum</th>
<th>expected</th>
<th>diff</th>
<th>bit_bucket</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>.5</td>
<td>.5</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>.25</td>
<td>.75</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>1.01</td>
<td>.99</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>01</td>
</tr>
<tr>
<td>.001</td>
<td>.999</td>
<td>1</td>
<td>1</td>
<td>8.6736E-19</td>
<td>011111110</td>
</tr>
</tbody>
</table>
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. 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.

\[
(A + B) + C \neq A + (B + C)
\]

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 \((A+B)C=(AC)+(BC)\)

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.

```sas
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 '** ( ' b ' * ' c ' )';
cards;
  .1 .2 .3;
```

This program produces 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.

```sas
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,rnd); /* 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) ...

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 ±2^100, any variation whose absolute value is less than 200*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 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.
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, $\frac{1}{2^{23}}$. 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

```sas
data pennies2; set Pennies;
twoexp = 2**floor(log2(dollars));
diff_eps = (diff/CONSTANT("MACEPS"))/twoexp;
run;
```

This program produces this scatter plot.

---

**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.

```sas
data _null_
    format n 1.
    input n @;
    put n= +1 @;
cards;
  1.75 1.25 1.499999999999999 .5
```

This program produces the following output:

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

**FUNCTIONS FOR ROUNDDING 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>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CEIL</td>
<td>Returns the smallest integer that is greater than or equal to the argument, fuzzed to avoid unexpected floating-point results.</td>
</tr>
<tr>
<td>CEILZ</td>
<td>Returns the smallest integer that is greater than or equal to the argument, using zero fuzzing.</td>
</tr>
<tr>
<td>FLOOR</td>
<td>Returns the largest integer that is less than or equal to the argument, fuzzed to avoid unexpected floating-point results.</td>
</tr>
<tr>
<td>FLOORZ</td>
<td>Returns the largest integer that is less than or equal to the argument, using zero fuzzing.</td>
</tr>
<tr>
<td>FUZZ</td>
<td>Returns the nearest integer if the argument is within 1E-12 of that integer.</td>
</tr>
<tr>
<td>INT</td>
<td>Returns the integer value, fuzzed to avoid unexpected floating-point results.</td>
</tr>
<tr>
<td>INTZ</td>
<td>Returns the integer portion of the argument, using zero fuzzing.</td>
</tr>
<tr>
<td>ROUND</td>
<td>Rounds the first argument to the nearest multiple of the second argument or to the nearest integer when the second argument is omitted.</td>
</tr>
<tr>
<td>ROUNDE</td>
<td>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.</td>
</tr>
<tr>
<td>ROUNDZ</td>
<td>Rounds the first argument to the nearest multiple of the second argument, using zero fuzzing.</td>
</tr>
<tr>
<td>TRUNC</td>
<td>Truncates a numeric value to a specified number of bytes.</td>
</tr>
</tbody>
</table>

**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

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘EXACTINT’ &lt;, nbytes&gt;</td>
<td>Exact integer</td>
</tr>
<tr>
<td>‘BIG’</td>
<td>The largest double-precision number</td>
</tr>
<tr>
<td>‘LOGBIG’ &lt;, base&gt;</td>
<td>The log with respect to base of BIG</td>
</tr>
<tr>
<td>‘SQRTBIG’</td>
<td>The square root of BIG</td>
</tr>
<tr>
<td>‘SMALL’</td>
<td>The smallest double-precision number</td>
</tr>
<tr>
<td>‘LOGSMALL’ &lt;, base&gt;</td>
<td>The log with respect to base of SMALL</td>
</tr>
<tr>
<td>‘SQRTSMALL’</td>
<td>The square root of SMALL</td>
</tr>
<tr>
<td>‘MACEPS’</td>
<td>Machine precision constant</td>
</tr>
<tr>
<td>‘LOGMACEPS’ &lt;, base&gt;</td>
<td>The log with respect to base of MACEPS</td>
</tr>
<tr>
<td>‘SQRTMACEPS’</td>
<td>The square root of MACEPS</td>
</tr>
<tr>
<td>‘E’</td>
<td>The natural base</td>
</tr>
<tr>
<td>‘EULER’</td>
<td>Euler constant</td>
</tr>
<tr>
<td>‘PI’</td>
<td>Pi</td>
</tr>
</tbody>
</table>

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 COMPUFUZZ FUNCTION

The COMPUFUZZ 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 COMPUFUZZ. COMPUFUZZ is not documented in the SAS 9.2 documentation, but it is discussed in SAS note 15553.

Table 12 COMPUFUZZ Function

<table>
<thead>
<tr>
<th>COMPUFUZZ</th>
<th>This function is used for numerical error analysis and performs a fuzzy comparison of two numeric values.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax</td>
<td>COMPUFUZZ(value1, value2 &lt;, fuzz &lt;, scale&gt;&gt;)</td>
</tr>
<tr>
<td>Arguments</td>
<td>value1 specifies the first of two numeric values to be compared.</td>
</tr>
<tr>
<td></td>
<td>value2 specifies the second numeric value to be compared.</td>
</tr>
<tr>
<td></td>
<td>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</td>
</tr>
<tr>
<td></td>
<td>scale specifies the scale factor used in computing the threshold. Default: MAX (ABS(value1), ABS(value2))</td>
</tr>
<tr>
<td>Returns</td>
<td>-1 if value1 &lt; value2 – threshold</td>
</tr>
<tr>
<td></td>
<td>0 if ABS(value1 - value2) &lt;= threshold</td>
</tr>
<tr>
<td></td>
<td>1 if value1 &gt; value2 + threshold</td>
</tr>
<tr>
<td></td>
<td>where threshold = fuzz * ABS(scale) /* if 0 &lt;= fuzz &lt; 1 */</td>
</tr>
<tr>
<td></td>
<td>threshold = fuzz * ABS(scale) * CONSTANT('MACEPS') /* if 1 &lt;= fuzz &lt; 1 / CONSTANT('MACEPS') */</td>
</tr>
</tbody>
</table>

Our next program uses COMPUFUZZ 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 COMPUFUZZ to compare our sums. SCALE is the sum of the absolute
values of \texttt{val}. We set \texttt{fuzz} to the 2 times the number of observations, because that is the total number of terms in our sums.

```sas
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 \texttt{2.8147498E14} and \texttt{-2.81475E14}, they should be considered to be equal, within the context of our input values.

\section*{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.

\section*{RESOURCES}

The IEEE 754 standard for binary floating-point arithmetic can be found at \url{http://grouper.ieee.org/groups/754/}.


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

SAS Note 15553 describes the \texttt{COMPFUZZ} function. \url{http://support.sas.com/kb/15/553.html}

\section*{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. \textregistered{} indicates USA registration.

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