One of the most common problems facing the SAS® user is the problem of numeric precision. The fact is, the SAS System stores numeric values in floating point representation, which can introduce problems of which the casual user may not be aware. This paper attempts to shed some light on the method of storing numerics in floating point representation in order to increase the awareness of the SAS user. Also, we discuss some caveats caused by the use of floating point representation.

It is appropriate first to discuss decimal numeric representation. This form of storing numbers seems natural to us because it was developed on the basis that we have ten fingers on our hands. We all know how to add, subtract, multiply, and divide based on algorithms we learned by rote when we were children. The decimal system had been adequate for our numeric needs up until the computer age, when binary number systems became a necessity due to the ones and zeros representative of current flow. Computers are notoriously fast with binary arithmetic, but surprisingly slow with the emulated decimal arithmetic. A simple exercise helps to illustrate the computer's difficulty with decimal computation: try adding a large column of hexadecimal figures. You are operating out of your native mode of decimal math, thereby reducing your capabilities to tally the numbers rapidly. The computer has the same problem with decimal figures.

The binary number system allows the computer to operate at its most efficient pace. We, as humans, may use symbols and even decimal numbers to communicate with the machine, but ultimately, binary representations are used. One of the ways that we can reduce the volume of all those ones and zeros is by using hexadecimal representation, which allows for four binary digits to be collapsed into one hex digit. All of the examples in this paper use hex representation.

All of the computers on which the SAS System currently runs have a maximum binary number of 7FFFFFFF or approximately 2.1 billion. In most cases, this number may be sufficiently large to handle the magnitude of your data values. However, this representation is only sufficient for integers. Certainly, your applications are going to need fractions! Therefore, to allow for fractional values and for values of much greater magnitude, all of the host systems employ a floating point representation, which is used to store all numeric values in the SAS System. The maximum number of decimal places and the minimum representable values are host-dependent, as you will see.

Before discussing the limitations and shortcomings of the floating point representation, let us first see the format in which the number is stored. I will present most of the examples in IBM mainframe format, that is used on MVS, CMS, VS1, and similar operating systems. A floating point number is represented by a number (called the mantissa) multiplied by a base raised to a power. In the case of the IBM mainframe format, the base is 16. So, for example, 1 is \(1 \times 16^0\), and 32 is \(2 \times 16^1\). Negative numbers have negative mantissas, and numbers that have an absolute value less than 1 have negative exponents. This is all very much the same as what you may remember being called "scientific notation" except that powers of 16 are used instead of powers of 10.

A further aspect of scientific notation and floating point representation is the use of fractions. For decimal scientific notation, a number is represented as a mantissa between 0 and 1 raised to a power of 10. So, for example, 142 would be represented as \(1.42 \times 10^{2}\). Floating point representation works the same way: numbers are stored with a mantissa less than 10 (that's 16 in decimal), raised to a power of 10 (again 16). For example, the number 30 in decimal is \(1E\) in hexadecimal. That would be represented as \(1E \times 16^{2}\).

Now how does this number actually get represented in the computer? As you may know, the default size of a numeric variable in a SAS data set is 8. This means that the native mode for storing a floating point number requires 8 bytes. The first byte of the number contains the exponent, and the remaining 7 bytes contains the mantissa. However, the first bit of the exponent byte contains the sign of the mantissa. The remaining 7 bits of the exponent byte contain the exponent. Then, the remaining bytes are all dedicated to the mantissa. If you use a length \(n\) where \(n\) is less than 8, then only the bytes up the the \(n\)th byte are used for the mantissa. You can see why you cannot have a length of 1 for numeric variables in the SAS System: at least one byte is needed for the exponent, and at least one byte is needed for the mantissa. Any additional bytes will simply allow there to be more precision available. There will be more discussion on precision later.

The first byte, as indicated, is the exponent, except for the first bit, which is the mantissa's sign. This leaves 7 bits to represent the exponent including the sign. You can represent an unsigned integer up to a value of 127 with 7 bits. On the IBM system, the exponent 41 (decimal 65) actually corresponds to an
exponent of 1. This is called the "bias" or "excess." The bit representation of 41 is

\[
0 \ 1 \ 0 \ 0 \ 0 \ 0 \ 0 \ 1
\]

If we turn on the high-order bit (the first bit of the byte):

\[
1 \ 1 \ 0 \ 0 \ 0 \ 0 \ 0 \ 1
\]

we derive C1, and that indicates an exponent of 1 but with a negative mantissa.

Now for some examples. The value of 1 is represented as

41 10 00 00 00 00 00 00

or .1 \* 10^0 (all hexadecimal). The value of 16 (decimal) is represented as

42 10 00 00 00 00 00 00

or .1 \* 10^4 (all hexadecimal). The value -16 (decimal) is

C2 10 00 00 00 00 00 00

The trailing bytes of zeros are there because there are no more bits needed to fully represent the number. However, consider the value of .1 (decimal). This number can be represented exactly in terms of decimal numbers, that is, 1 \* 10^-1 (decimal). However, the binary representation is approximate. In floating point representation, it appears as

40 19 99 99 99 99 99 99

It is not difficult to see why we have the repeating 9s. First, consider that .1 (decimal) = 1/10 = 1.6/16. We obtain our first hexadecimal digit, 1, from 1.6, leaving .6 = 6/10 = 9.6/16. We obtain our next hexadecimal digit, 9, from 9.6. We will continue to repeat the .6 portion infinitely.

Here is where we reintroduce the subject of precision. Because bits up to the last byte of the representation are used (albeit to correspond to a repeating hex digit), we are dealing with an approximation. And whenever operations are performed on an approximation, the results may be a little unexpected. The best way to understand this is through example. In fact, I call this the "classic example" because I have used it so many times to illustrate the point of precision.

If you have the following SAS statements:

```sas
DATA NULL; DO I=-1 TO 1 BY .1; X+I; IF I = 0 THEN PUT 'AT ZERO'; END; PUT X=; RUN;
```

What do you expect to see displayed? What you will not see it the message "AT ZERO" printed anywhere. And why is that? It is easily understood by looking back at the representation of the value of .1 in hex. When you add this imprecise number to the precise value of -1, you will become less and less accurate with each incrementation. When you get to the point where the variable I should be zero, it is instead

00 00 00 00 00 00 00 00 (value of zero)

B3 60 00 00 00 00 00 00 (accumulated value).

If you compare the value of I with 0, they will not match. However, if instead you had said

IF ROUND(I,.0001)=0 THEN PUT 'AT ZERO';

you would have indeed seen the message. The number is very close to zero, but it is not exactly equal to zero. The ROUND function will round it to zero.

Furthermore, you should note that when the DATA step completes, the value of I is not exactly 1:

41 10 00 00 00 00 00 00 (value of 1)

40 FF FF FF FF FF FF F4 (actual value of I).

The problem of imprecise storage is exacerbated by the use of the LENGTH statement to truncate variables. The only purpose of the LENGTH statement is to reduce the amount of disk space needed to hold the SAS data set. As you may know, the DATA step expands all numeric values to a full 8 bytes in its internal program data vector before operating on the values. So if you use the LENGTH statement, remember that the DATA step will pad the short numeric value, operate on the full length value, then truncate it when outputting the observation. The PROC step works the same way. The main reason for this is that the operating system only deals with two kinds of floating point numbers: single precision (4 bytes) and double precision (8 bytes). It was a decision made many years ago in the early designs of the SAS System that double precision would be used whenever possible. Therefore, all the DATA step operations, and almost all the PROC step operations involve double precision floating point.

After that introduction, let's examine the ramifications of these facts. We will use the classic example again, with an added twist.

```sas
DATA TEMP; LENGTH I 4; KEEP I; DO I=-1 TO 1 BY .1; PUT I=I=HEX16.; OUTPUT; X+I; END; PUT X=X=HEX16.; DATA NULL; SET TEMP END=EOF; X+I; PUT I=I=HEX16.; DATA NULL;
```

The DATA step completes, the value of I is not exactly 1:

41 10 00 00 00 00 00 00 (value of 1)

40 FF FF FF FF FF FF F4 (actual value of I).

After that introduction, let's examine the ramifications of these facts. We will use the classic example again, with an added twist.

Data Temp;
LENGTH I 4;
KEEP I;
DO I=-1 TO 1 BY .1;
PUT I=I=HEX16.;
OUTPUT;
X+I;
END;
PUT X=X=HEX16.;
DATA NULL;
SET TEMP END=EOF;
X+I;
PUT I=I=HEX16.;
IF EOF THEN PUT X=X=HEX16.;
A portion of the output from this example is as follows:

(first DATA step)
I=-1 I=3100000000000000
I=-0.9 I=COE6666666666667
I=.9 I=40E66666666666666

X=2.08167E-16 X=33F0000000000000

(second DATA step)
I=-1 I=C1100000
I=-.9 I=COE66666
I=.9 I=40E66666
I=0.9999999 I=40FFFFFF
X=-1.19209E-07 X=B200000000000000

The value of X, if decimal precision were not a problem, would be zero. Notice the difference in X between the first and second DATA steps. At first glance, you'd think that X should be exactly the same. The reason that this isn't true is that in the first DATA step, X is being accumulated on the full 8-byte representation of I, which we know uses all 8 bytes for the most precise representation. In the second data step, the value of I is being read from a 4-byte truncated location, into the 8-byte location in the program data vector, and this value is what's being added to X. Therefore, there is some loss of precision. So keep this in mind when using the LENGTH statement.

There is another consideration to remember when using the LENGTH statement. I think that most SAS programmers have encountered this at least once in their SAS programming careers. If you look at the LENGTH table in User's Guide: Basics, Version 5 Edition, you'll see the following information for the IBM mainframe system:

<table>
<thead>
<tr>
<th>length</th>
<th>maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>25</td>
</tr>
<tr>
<td>3</td>
<td>65535</td>
</tr>
<tr>
<td>4</td>
<td>16777215</td>
</tr>
<tr>
<td>&gt;4 billion</td>
<td></td>
</tr>
<tr>
<td>&gt;1 trillion</td>
<td></td>
</tr>
<tr>
<td>&gt;281 trillion</td>
<td></td>
</tr>
<tr>
<td>&gt;72 quadrillion</td>
<td></td>
</tr>
</tbody>
</table>

What this table means is that if you try to store a number in an area that can't fully accommodate it, you won't simply get a loss of precision; you'll get an entirely different number. Consider for example:

DATA TEMP;
LENGTH X 2;
X=1000;
PUT X= X=HEX16.;
RUN;
DATA NULL;
SET TEMP;
PUT X= X=HEX16.;
RUN;
X=1000 X=433E800000000000
X=992 X=433E000000000000

As you recall from our discussion of what makes up a floating point number, you can see why it takes more than 2 bytes to properly store the value 1000: the number 1000 in hexadecimal is 3E8 or .3E8 * 10**2 (hexadecimal). The 3E8 requires 3 "nibbles" (half-bytes), but we must use 2 full bytes to store these three nibbles, plus one byte for the exponent, for a total of three bytes. What confuses some people, however, is that larger numbers may not need as many bytes. The decimal number 1024, for example, is 400 in hexadecimal, or 4 * 10**2 (hexadecimal). In this case, a total of 2 bytes is needed.

In the first DATA step of our example just above, there appears to be no problem because the program data vector location is 8 bytes. When we store it away in the TEMP data set, we only allow 2 bytes. So in the second DATA step, when we read the value back in, we come up with 992, which is 3E8 * 10**2 (hexadecimal).

A second point to make about this length table is that it refers to integers. It is highly recommended that if the variable is to contain noninteger values, that you don't use a truncated length at all. Remember the storage of .1 in our "classic example." Here is the table of values when lengths of 2 to 4 are used to store .1:

<table>
<thead>
<tr>
<th>length</th>
<th>decimal</th>
<th>hex representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>.09765625</td>
<td>40 19</td>
</tr>
<tr>
<td>3</td>
<td>.09999084</td>
<td>40 19 99</td>
</tr>
<tr>
<td>4</td>
<td>.09999996</td>
<td>40 19 99 99</td>
</tr>
</tbody>
</table>

As always, some fractions can be comfortably stored in truncated numbers with no problem. For example, the value .5 becomes

40 80 00 00 00 00 00 00

so it could be stored in a length of 2 with no loss of precision. But unless you know that your variable will only hold a given set of discrete fractions all storable without data loss, don't use a LENGTH statement for those variables. It isn't worth it to lose data for the gain of a small amount of disk space. If disk space is a great premium, you may want to use tape.

Up to this point, we have been focusing on the representation of floating point numbers on the IBM mainframes, using the MVS, CMS, V51, or DOS/VSE operating systems. This has been shee
for the sake of example. All of the problems we’ve seen can be found on the other operating systems on which the SAS System currently runs: VMS®, AOS/VS, PRIMOS®, and PC DOS. On each system, the floating point representation has its own peculiarities, and each merits a little discussion.

Let us consider VMS. On this system, the exponent represents a power of 2, and there is no bias. The sign of the mantissa takes the first bit. Therefore, the values of 1 and 2 would be

00 00 00 00 00 00 00 00
01 00 00 00 00 00 00 00

Because the limitation is $2^{61}$ (decimal), the magnitude extends approximately $2.9538$ to $1.7E36$. The VMS system has an extension feature available called the "floating point accelerator." When you compile programs, you use the GFLOAT option, and it generates different object code for floating point operations. This object code will only run on hardware that has the floating point accelerator boards added. The accelerated floating point allows for 16-byte floating point numbers, therefore increasing precision and magnitude. The Institute decided against incorporating the floating point accelerator object into the SAS system, because it would require a separately maintained set of objects, which would be terribly cumbersome.

The AOS/VS system uses the same floating point representation as the IBM mainframe system. As far as SAS users are concerned, the only difference is in the storage of missing values, which is discussed later in this paper.

Floating point numbers on PRIMOS are stored in a split fashion, with the mantissa sign bit at the beginning of the number and the exponent at the end of the number. The exponent is 16 bits long, with a bias of 128 (decimal). Because of the splitting and the 2-byte exponent, all numeric variables must have a minimum length of 3 on PRIMOS.

Finally, there’s the newest addition to our host operating system family, PC DOS. The confusing thing about PC DOS is that there are two accepted floating point representations, the Microsoft® standard and the Intel® standard. The latter is the one used and recognized by the Lattice® C compiler, which the Institute used in developing the SAS System on PC DOS. In the Intel standard, the sign of the mantissa is stored in the first bit of the first exponent byte. The exponent itself uses 11 bits (7 in the first exponent byte, 4 in the next byte). The exponent bias is 1023 (hex 3FF). The exponent refers to powers of 2, not 16. The remaining bits are used for the mantissa, with an implied 1 at the beginning of the mantissa. Moreover, the entire 8-byte representation is stored reversed from the IBM mainframe. To simplify our examples, we show the representations as if they were not reversed:

3F 00 00 00 00 00 00

This is the representation of 1. It is derived by subtracting 3FF from the exponent of 3FF, leaving an exponent of 0. With an implied 1, this becomes

$1 \times 2^{0}$

which is 1. The value 2 is represented as

40 00 00 00 00 00 00 00

which is $1 \times 2^{1}$.

In actuality, the floating point numbers listed above are stored as

00 00 00 00 00 00 00 00 FO 3F
00 00 00 00 00 00 00 00 00 40

so don’t be surprised to see these representations when you use the HEX16 format.

The Microsoft format is a little different and is not used in the SAS System except via the MRB informat and format, which converts to and from that representation. The Microsoft format is used in BASIC. Also not discussed here is the short form of the Intel floating point. It has fewer bits for the exponent.

Now let us clear up some very confusing points concerning precision and magnitude. The Technical Support staff at the Institute has received a lot of calls from users questioning the precision capabilities of the IBM Personal Computer. The thought is that if a computer is that small, it’s not capable of handling large numbers. On the contrary. The SAS System under PC DOS can handle numbers up to $1E308$, which is much larger than the IBM mainframe can handle. This is because the representation chosen by Intel allows for numbers up to $2^{1024}$ (decimal), which is around $1E308$. The IBM mainframe floating point number is limited to $16^{64}$ (decimal), which is around $1E75$. However, since the Intel representation uses 11 bits for the exponent, this means there are less bits to store the mantissa. So a less precise number can be stored, although a larger magnitude is available.

But there’s a further point on precision. The floating point operations in PC DOS may be in hardware or software. If you have a PC with an 8086 or 8088 chip (which most PC/XT machines have), your floating point operations will be performed by software. That is, these chips do not have built-in hardware floating point operations. The 8087 math processor does. The precision of hardware versus software operations is the same, but hardware operations are much faster. That’s why the Institute recommends an 8087 chip for anyone who will be doing large
amounts of statistical calculations. When the DATA or PROC step determines that you have such a chip present, it will use the chip’s functionality. Otherwise, software emulation takes place. When emulation occurs, the process is slower, but it does produce the same result.

All of this discussion about different representations becomes more significant when we consider data transfer. If you do this transfer via our micro-to-mainframe link with SAS data sets, simply keep in mind the precision differences, especially in uploading to the mainframe. You won’t be able to store the number 1E200 on the mainframe like you will on the PC. However, if you want to transfer a "flat" file of data stored in internal floating point representation, you’ll have a problem. This is because the RB informat is designed to operate only on native mode floating point numbers. So if you present the following 8 bytes to a DATA step on the PC (representing the number 1 on the IBM mainframe)

41 10 00 00 00 00 00 00

they’ll be read in on the PC via the RB informat as 1.11254E-308.

Now we turn to missing values. Although missing values do not relate to precision per se, they are certainly pertinent to any discussion on floating point numbers.

There are 28 possible missing values in the SAS System: _, ., and .A - .Z. On the IBM mainframe systems, these values are stored as numbers with zero mantissas but with nonzero exponents. The exponent corresponds to the hex representation of the character after the dot.

The character '_' is C1 in hex, so . would be represented as C1 00 00 00 00 00 00 00. The hex for the standard missing value is 80 00 00 00 00 00 00 00. Internally, the SAS System does not perform any numeric comparison or arithmetic operations on these numbers, because the operating system would normalize both operands to zero. Checking is always done before computations are performed.

In other words, missing values are a contrivance. In the various operating systems under which the SAS System runs, this is the case. Missing values have been "invented," and are not operated upon arithmetically. Under VMS, the missing values are the 28 smallest possible numbers:

_. FF FF FF FF FF FF FF . FF FF FF FF FF FF FF .A FF FF FF FF FF FF FF FF .Z FF FF FF FF FF FF etc.

Under AOS/VS, the missing values are stored as they are under IBM mainframe systems, with a slight difference. Since AOS/VS uses ASCII instead of EBCDIC for storage of characters, the ASCII character corresponding to '_', , (5F), or A-Z (41-5A) is used as the exponent, with a mantissa of zero.

Under PRIMOS, all bytes are zero except the sixth byte, which uses values starting with 01 for the lowest missing value, _. The values are

_. 00 00 00 00 00 01 00 00 . 00 00 00 00 00 02 00 00 .A 00 00 00 00 00 03 00 00 etc.

Z 00 00 00 00 01 00 00 00

Under PC DOS, the missing values are

_. 00 00 00 00 00 01 FF FF . 00 00 00 00 00 02 FF FF .A 00 00 00 00 00 03 FF FF .B 00 00 00 00 00 0D FF FF etc.

Z 00 00 00 00 0A FF FF

In order to use the SAS System on many different machines, you must keep in mind the disparity in precision and magnitude. Some operating systems (such as PRIMOS and PC DOS) give you very high magnitude but with loss of precision. Others (such as IBM mainframe operating systems and AOS/VS) give you more precision but less magnitude.

I would like to discuss another area where precision affects the output generated by the SAS System. It involves the use of the picture formats in PROC FORMAT. Consider this simple picture format:

PROC FORMAT;
    PICTURE OTHER='0000009.99';
and consider the value 1234567.12. If we format this value using the picture format defined above, we get '1234567.11'. However, if we use 10.2, we get '1234567.12'. Why the difference? It is due to the fact that the picture logic does not round the numbers; it truncates them. In the example above, the picture logic will determine the multiplier to be 100, because there are two places to the right of the decimal, and that’s 10^2, or 100. It will multiply 1234567.12 by 100 and truncate the result. The internal representation of 1234567.12 is

46 12 06 87 1E 88 51 EB

and multiplying it by 100 gives us an internal representation of

47 75 BC CC 7F FF FF FF

instead of 123456712, which is

47 75 BC CC 80 00 00 00
Part of the reason for the precision loss is that 1234567.12 is represented by 4612D687 followed by 1EB85 repeated, and we lose significant bits by the fortuitous placement of the bytes in the mantissa. In any event, the multiplication by 100 gives us a number like 123456711.99999999, but since the picture logic truncates, it becomes 123456711. In the case of the 12.2 format, a fuzz value of 1E-12 is added before the truncation in formatting, causing the number to exceed 123456712 appropriately.

Some users may think the FUZZ= option helps in picture formats. Unfortunately, it is only useful in finding a match with possible values, not for formatting the result.

Finally, some suggestions for improving the precision problems are warranted. First, I suggest the need for a global fuzzing option. If an arithmetic operation generates a result that is within a fuzz value's distance of a certain decimal position, this option would indicate that rounding is performed, to be as close as possible to that position. Our "classic example" would have worked fine with a fuzz value that was very small, say 1E-20. You could control the fuzz as needed. However, many caveats abound here. A DATA step could produce very different results with different fuzz values. The arithmetic operations would be slowed, because a check would be in order. And would the fuzzing occur with every intermediate operation? For example,

\[ X = \log(\sqrt{Z} + Q/\text{SUM(OF A B C)}) \]

Would we fuzz after summing \( A \), \( B \), and \( C \), then after dividing, and so on? If so, the results could become very different than if we only fuzzed before assigning \( X \).

I present this proposal partially to demonstrate the difficulty in implementing it. Users consider the ramifications of the "classic example" to be serious, but as demonstrated, there are equally serious ramifications of this global fuzzing operation.

Now on to a less insidious option, one that would really be more of a warning, without causing any internal changes. Recall the examples with the LENGTH statement in which data loss quietly occurred. I suggest a DATALOSSWARN option that would inform the user if there were nonzero bits in the program data vector that were being lost due to truncated lengths. This operation could be performed once when the observation was being written, so it wouldn't be too expensive. Perhaps it could become a standard warning, and one would use the NODATALOSSWARN option to omit it. Note that this warning would not have any bearing on the inputting of a truncated variable, because the data loss has already occurred by the time the observation is written.

For PROC FORMAT, I suggest an additional FUZZ option for PICTURE formats. Because the PICTURE process truncates, we have seen in previous examples that we may get unexpected results due to imprecision. If there were a fuzzing option that rounded to the nearest specified decimal place, many of these problems could be avoided. There is already a FUZZ option for considering if an input value matches with a given range, so that problem is resolved.

In conclusion, the SAS programmer should be aware of the ramifications introduced by the representation of numbers in floating point. The user should employ the LENGTH statement with caution. If accumulation of non-integral decimal numbers is to occur, a little advance testing of this accumulation will result in fewer surprises later. The SAS System can certainly store numbers with great magnitude and precision, but the programmer must remain knowledgeable about the problems discussed.

SAS is a registered trademark of SAS Institute Inc., Cary, NC, USA.

VMS is a trademark of Digital Equipment Corp., Maynard, MA, USA.

PRIMOS is a registered trademark of Prime Computer, Inc., Framingham, MA, USA.

Lattice is a registered trademark of Lattice, Inc., Glen Ellyn, IL, USA.

Microsoft is a registered trademark of Microsoft Corp.

Intel is a registered trademark of Intel Corp.