Scientific Notation and Floating Point Representation

Douglas A. Kerr, P.E. (Ret.)

Issue 1
July 20, 2011

ABSTRACT AND INTRODUCTION

Scientific notation is a scheme for presenting numbers over a wide range of values that avoids the consumption of page space and the other inconveniences of long and unprofitable strings of leading or trailing zeros. A related convention provides for the convenient entry of numbers over a wide range into calculators or their introduction as constants into computer programs. A closely-related concept, *floating point representation*, provides for the compact representation of numerical values over a wide range inside computers or in data recording structures. In this article we will examine these concepts and then give details (some very tricky) of various standardized conventions for their use. The metrics *range* and *precision* are discussed. Background is given on pertinent basic number theory concepts.

SCIENTIFIC NOTATION

The situation

Especially if we wish to retain the basic SI units (that is, to not use multipliers such as *milli* and *mega*), we often have to write numbers that have a substantial number of leading or trailing zeros.

For example, the mass of an electron is about:

0.000000000000000000000000000091093822 kg

The Earth's mass is approximately:

597360000000000000000000 kg

These are not only tedious to write or keyboard, they also take up space on the page, and the reader, to grasp them, must do a lot of digit counting (perhaps using a pencil as a handy pointer, with the result that a treasured textbook gets defaced).

There is of course a convention for facilitating digit counting:

0.000 000 000 000 000 000 000 000 000 000 910 938 22 kg

5 973 600 000 000 000 000 000 000 kg

or (in the US in particular)

5,973,600,000,000,000,000,000,000 kg

but it doesn't help much.

Significant digits

In this example:

1.0230 kg

it is clear that we intend to assert a precision of 5 significant digits (else the trailing "0" would not be there; the value would be the same without it).

But in this case:

12,500 m

it cannot be made clear whether we are asserting a precision of 3, 4, or 5 significant digits.

Principle

In scientific notation, in its generalized form, we express a number in this form:

$\mathbf{s} \times 10^{e}$

where **s** (the *significand*[1]) is any real number and **e** (the *exponent*) is any integer, positive, negative or zero.

Thus the following would be legitimate examples of the use of scientific notation:

1.235×10^{-7}

0.027×10^{6}

7×10^{3}

We might even run into this:

0.0007×10^{6}, or

$100{,}000 \times 10^{-3}$

We might wonder what is the purpose of the latter two, since they seem to be burdened with "unprofitable" zeros. But this can make sense if we wish to compare two numbers:

---

[1] This is also called the *mantissa*. This is deprecated today out of concern over confusion with the use of "mantissa" to mean the fractional part of a logarithm, a related but different thing (as is discussed later in this article).

2000×10^{15}

0.001×10^{15}

making it fairly obvious by inspection that the first is 2 million times bigger than the second.

Another benefit of scientific notation is that it can also allow us to make an explicit assertion about the number of significant digits of a value. For example, if we take this value that we saw earlier:

12,500 m

and write it as:

1.250×10^{4}

then we make it clear that we intend 4, not 3 or 5, significant digits—1-2-5-0.

Normalized scientific notation

In many cases we adopt the convention of making the significand, **s**, always in the range:

$1 \leq \mathbf{s} < 10$

That is, it can have a value from 1.00000... through 9.99999... .The exponent, **e**, will then have to be chosen, in the face of that, so that s and e together represent the intended quantity.

This is often called *normalized scientific notation*.

Examples of values conforming to this convention are:

6.23×10^{23}

1.0001×10^{-40}

Zero

In basic (non-normalized) scientific notation we can represent zero (although we might ask what is the point). We can write:

0×10^{0}, or, if for some weird reason we prefer to,

0×10^{-108}

They are both 0.

In normalized scientific notation we cannot express 0: in that mode, the significand cannot be 0, as would be required to do that directly.

But this is rarely of any concern. We use scientific notation to "write" or "print" numbers. We do not have to use it for all numbers. We do not use it for page numbers, nor when we say "251 machines of this

type are already in service". So when we mean zero, we can write "0" (or "zero", depending on the editorial context).

Engineering notation

A variation of scientific notation, often called *engineering notation*, caters to the widespread use in engineering of units that are basic units affected by the standard SI multipliers, whose numerical values are always 10^{3n}, where **n** is a positive or negative integer (not zero).

In engineering notation we always:

- make the significand in the range $1 \leq \mathbf{s} < 1000$
- make the exponent an integer multiple of 3

Examples would be:

12.5×10^{6} V ("ah, yes, 12.5 megavolts")

850×10^{-9} s ("ah, yes, 850 nanoseconds")

E notation

Closely related to scientific notation, and in fact derived from it, is what is perhaps best called "E notation". It is often used to state numerical constant in computer programs, and is also often used as a "poor man's scientific notation" in informal papers, or when corresponding by e-mail when scientific notation cannot be properly presented owing to typographical limitations.

The principle can perhaps best be shown through examples. A value will first be stated in normalized scientific notation and then in (normalized) E notation.

1.5609×10^{16} 1.5609E16 or, preferably, 1.5609E+16

8.75×10^{-11} 8.75E−11

In some contexts it is acceptable to use a non-normalized form. In some contexts this scheme is adapted to parallel engineering notation.

The symbol "E" is said to be evocative of "exponential". The notation is sometimes called "exponential notation", although that term technically applies equally well to scientific notation as well as related forms using a base other than 10 (as we will encounter in our later work on binary floating point representation).

FLOATING POINT REPRESENTATION

Principle

Floating point representation adapts the concept of scientific notation, in a binary implementation context, to the representation of numbers in computer memory, or in stored data.

Basically, again we represent a number in the form:

$\mathbf{s} \times 10^{\mathbf{e}}$

where **s** and **e** are binary numbers, **s** being a binary integer, fraction, or mixed number and **e** being a binary integer.

The binary point

The *base* of a number system is also called its *radix*. It is 10 for decimal numbers, 16 for hexadecimal numbers, and 2 for binary numbers.

In the decimal number system, we are familiar with the use of the decimal point (shown in US practice as a period). This allows us to have "places" in the number whose value is less than one. The digit just to the left of the decimal point is the units digits, worth "1 each". The place just to the right is worth "1/10 each"; the place to the right of that "1/100" each; and so forth. As we go to the right, the value of each is the fraction 1/10 of the prior place value. That fraction is by definition 1/*r*, where *r* is the radix, 10 in this case.

Though less familiar to us, the same concept is valid for any number system. In the general case, this marker is called the *radix point*. In binary numbers, this is specifically called the *binary point*.

We see here a binary number with a binary point:

1111.11

As in the decimal case (and for any radix), the place just to the left of the binary point (the "units place") is worth "1 each". The place to the right of the point is worth (in decimal) "1/2 each" (that is, 1/r, or r^{-1}, where $r = 2$ for the binary system). The place to the right of that is worth 1/2 that, or "1/4 each" (r^{-2}). And so forth.

Thus the entire number is worth In decimal):

$8 + 4 + 2 + 1 + 1/2 + 1/4$, or 15-3/4 (15.75).

In general, in a computer or a data file, we don't put markers in mixed binary numbers (that is, those that have both an integer and a fractional part) to indicate the presence and location of a "binary

point". Thus, if in a certain situation in a computer the plan is to represent a number like this:

1011.0011 [its decimal value is 11-3/16 or 11.1875]

it is actually represented in memory just as:

10110011

The program or whatever "knows" that the binary point is between the 4th and 5th bits. It is always in that spot in the "field" for this particular data type. As a result, this class of scheme is often described as a *fixed point representation*.

The normal interpretation of a number (decimal or binary), if no radix point can be "seen", assumes the radix point to be just to the right of the rightmost (units) digit.

If the bit pattern above were interpreted (incorrectly) on that premise, it would seem to represent the decimal value 179. I sometimes call that the *manifest value* of a bit pattern.

We will work with binary mixed numbers here, aided by our supernatural ability to see the binary point.

The name

The name *floating point* was coined in distinction with *fixed point* (which was coined at the same time to be in distinction with floating point). "Fixed point" evokes a number lying in state in a box, with the radix point at a certain place. The spot had best be large enough to accommodate all the significant digits of the number over the range we wish to represent.

"Floating point" evokes the notion of the significand lying in state in a more cozy box (just big enough for all its significant digits), while the binary point could be anyplace, even well off the left or right edge of the "page", as required by the actual size of the number being represented.

Examples

Here we have some examples of binary floating point numbers. I will show the significand, **s**, in binary, but the rest in decimal. After some of the examples I will then show its value in binary "fixed point" form, and then in all cases its decimal equivalent.

1.1101×2^{5}	111010	58
0.1001×2^{12}	10010000	144
11.0101×2^{-6}	0.0000110101	0.05175...
101×2^{35}		$1.717... \times 10^{11}$

Normalized significand

Paralleling the situation with the scientific notation, often (for consistency in data handling) we will use *normalized floating point notation*, in which the significand is forced to lie in a certain range and the exponent has to be chosen in light of that.

Two schemes are common.

Normalized fractional significand

Here the significand, **s**, must lie in this range (expressed in decimal):

$0.5 \leq \mathbf{s} < 1$

That is, its value in binary must be in the range:

0.10000... through 0.11111...

A typical value might be:

0.101001101×2^6

Normalized mixed significand

Now, we will make a slightly different arrangement.

Here the significand, **s**, must lie in this range (expressed in decimal):

$1 \leq \mathbf{s} < 2$

That is, its value in binary must be in the range:

1.00000... through 1.11111...

A typical number (the same value as in the example above) might be:

1.01001101×2^5

Note that here the units digit (just to the left of the binary point) is always "1". Thus we do not need to store it in the representation of the number—when that is "decoded" the program just adds 1 to the significand as read.

The advantage of doing this is that with, for example 8 bits of storage, we can store a significand with 9 bits of precision.

What about zero?

In the "normalized mixed significand" scheme, we cannot represent zero, a deficiency not so easily ignored as in the case of writing values in a textbook. The reason of course is that we cannot ever make the significand 0.

Thus actual floating point representation schemes of this type have special provisions for representing 0. We will see this when we look, shortly, at the details of an actual standard scheme.

RANGE AND PRECISION

We often wish to characterize number representation schemes in terms of two attributes, *range* and *precision*. We will work with those when I discuss some specific floating point representation systems. For now, let's review the concepts we will use so they will be familiar to us when we get to that work.

Range

The term range as applied to a number representation system can have several meanings, not always carefully distinguished.

The *range* of a number representation system (without further qualification) identifies the maximum and minimum values it can represent (making use of the algebraic sign if the system accommodates one).

For example, consider a decimal system comprising five digits with the decimal place (perhaps implicitly) at the far right and an algebraic sign, thus:

± ddddd.

Its range is:

−99999 through +99999

As a numerical value, the *range* is the difference between those two values. (We now have already seen two meanings of the term.)

For the system above, the numerical value of its range is 199998.

The *absolute value range* of a system is the same but only embracing its positive values (and zero, if it accommodates same). We assume that the system, if it does allow negative values, has a symmetrical range.

For the system example above, its *absolute value range* is:

0 through 99999

and the numerical value of its *absolute value range* is 99999.

Often, we are concerned with the *non-zero range* of a system. That is the range from the smallest positive non-zero value that can be represented to the largest positive value that can be represented.

For the system above that is:

1 through 99999

The *relative non-zero range* is the ratio of the larger of those to the smaller, in this case 99999.

Precision

By the *precision* of a numbering system we mean the "fineness" of its ability to distinctly represent different values. As with "range", the term comes in several flavors, often not carefully distinguished.

Basically, the *precision* of a number representation system is the size of the "step" in value between consecutive possible representations.

For example, in a decimal system of five digits (all of them "significant"), the precision is always 1 unit. Note that this is not "± 1 unit", as if we were stating an accuracy or a range of error.

In fact, in a system with "symmetrical rounding", the range of error due to "quantization" (the forcing of values to an allowable representation) is $\pm$ 1/2 the precision.

Often we are interested in the *relative precision*, which is the ratio of the precision to the actual value (and thus requires us to assume some value).

For example, in a decimal system with all digits significant, and thus a precision of 1, the *relative precision* at a value of 1500 is 1/1500.

Of course here, a smaller number is "finer precision". We may want a metric that increases for "finer precision". For that we can use the reciprocal of the relative precision, which has no consistent name. I will call this metric here the *precision score*. For example, for a decimal system with all digits significant, and thus whose precision is 1, at a value of 1500 the precision score is 1500.

Note that the precision score increases for increasing value. For a five digit system, at its maximum value, 99999, the precision score is 99999. Note that this is almost exactly 10^5, where 5 is the number of decimal digits.

If we increase the number of digits to 6, the precision score at maximum becomes 999999, almost exactly 10^6.

Thus, it is convenient to refer to the precision of a system in terms of equivalent decimal digits (5 for the first system, 6 for the second). We recognize that (by definition) this metric is almost exactly the logarithm (base 10) of the precision score.

Now, following this line of thought, for a decimal system at a value of 1500, the relative precision can be said to be about 3.2 decimal digits.

In the binary realm, the same notions apply. To cut to the punch line, we can describe the relative precision of a binary number representation system, at some particular value, in terms of equivalent bits, where that metric is the logarithm (base 2) of the "precision score" at that value, and is stated in "bits". Not surprisingly, for a regular 7-bit binary number, at its maximum value, that metric is almost exactly 7 bits.

THE SIGNIFICAND IN ACTION

It may seem curious that in both formulations of a floating-point scheme, the fractional significand system, we only use about half the possible range (the upper half) of the significand. This seems "wasteful", but there is a very good reason for it, which will shortly become apparent.

For now, lets watch the action. We will do this with the fractional significand formulation, but the concept applies to either.

For compactness of the example, imagine an 8-bit significand field, whose manifest value (that is, as if the binary point is at the far right) can run from 0 to 255. But we have agreed to only use the values from 128 to 255.

In fact we consider a binary point to exist at the left of the leftmost place. Thus the actual value would run from 0/256 to 255/256[2]; we will only use the range of values from 128/256 to 255/256.

Lets start with the significand at 255/256 (.11111111 in binary), and the exponent at +2. We can write that value (all in decimal) as:

$255/256 \times 2^2$ [3.9843750]

Now we will count down by increments of 1 in the significand:

$254/256 \times 2^2$ [3.9687500]

$253/256 \times 2^2$ [3.9531250]

$252/256 \times 2^2$ [3.9375000]

This isn't too interesting, so we'll jump ahead a few steps:

---

[2] Many people seem to expect that divisor to be 255, based on the fact that (irrelevant here) the maximum value of an 8-bit number is 255. But each time we shift the binary point to the left one place (or shift the number to the right, if you prefer), the value drops to 1/2. Eight of those shifts means it drops to 1/256.

$130/256 \times 2^2$ [2.0312500]

$129/256 \times 2^2$ [2.0156250]

$128/256 \times 2^2$ [2.0000000]

Do we next go to:

$127/256 \times 2^2$ [1.9843750]

No, and I'll talk about why shortly.

Instead, we "downshift and rev up" (to use an auto metaphor); we move the exponent down one notch to 1, and push the significand up to its maximum:

$255/256 \times 2^1$ [1.9921875] [not quite the same; stay tuned]

We then continue as expected:

$254/256 \times 2^1$ [1.9843750]

$253/256 \times 2^1$ [1.9765625]

Lets use a new analogy with an electrical test voltmeter, not a car drive train. Assume that its ranges were each a factor of 2 from the previous one (not usual, I know—this is just a metaphor).

We are testing what turns out to be successively decreasing voltages. Following good lab practice, as soon as the needle gets down to "half scale", we switch the meter range to one 1/2 the size.

Why? Because the relative precision of the meter decreases as the meter goes down. Thus we want to keep the needle in as high a part of the scale as possible. With successive scales having a ratio of 2:1, as soon as we get to half scale we can switch to the next lower scale, which would now put the needle at the top, and so we should do that.

So it is with our floating point representation. The precision of the representation is the step size, Δn , where n is the number represented. Recall that we can consider the *relative precision* of the representation as:

$\Delta n/n$

where n is the current number and Δn is the change in the number between adjacent steps.[3]

---

[3] To be really rigorous, n should be the average value of n for the two steps between which the difference is Δn; I didn't do it that way in my calculations for simplicity.

But our *precision score*, which we want to be higher for "better precision", is the inverse of that, or:

$n/\Delta n$

But since Δn in this case is always 1, that becomes just n. (Well, **that**'s handy!)

Now let's restate our countdown, this time showing the precision score, $n/\Delta n$, after each step:

$255/256 \times 2^2$ [3.9843750] [starting step]

$254/256 \times 2^2$ [3.9687500] $n/\Delta n = 254$ [I said it was handy]

$253/256 \times 2^2$ [3.9531250] $n/\Delta n = 253$

$252/256 \times 2^2$ [3.9375000] $n/\Delta n = 252$

-
-
-

$130/256 \times 2^2$ [2.0312500] $n/\Delta n = 130$

$129/256 \times 2^2$ [2.0156250] $n/\Delta n = 129$

$128/256 \times 2^2$ [2.0000000] $n/\Delta n = 129$

<shift range>

$255/256 \times 2^1$ [1.9921875] $n/\Delta n = 255$

$254/256 \times 2^1$ [1.9843750] $n/\Delta n = 254$

$253/256 \times 2^1$ [1.9765625] $n/\Delta n = 253$

Note that at the beginning, as we move down, the relative precision decreases (just as for our declining meter needle).

We "shift range" (that is, change to the next lower exponent value) as soon as we can (just at "half scale"). Just before that, the precision has declined from its original value of 254 (we missed 255 because of where we started) to 128.

But after the "range shift" (with the "meter needle at the top of the scale again"), the precision is back up to 255.

Had we allowed the significand to drop below 128/255 (and not "shifted range"), we would have suffered a further decline in precision. But we got our mojo back as soon as we could.

So that's why we only use the top half of the meter scale in our meter metaphor—and only the top half of the possible range of the significand in our floating point system.

The story is similar for the mixed number significand scheme.

An inconsistency?

Note that in the first part of this scenario, we discussed two possibilities for the step below this one:

$127/256 \times 2^2$ [1.9843750]

They were:

$127/256 \times 2^2$ [1.9843750], and

$255/256 \times 2^1$ [1.9921875]

Note that these don't have the same value (they differ by about 0.5%).

This is because, as we saw from the part of the exercise on precision, the step size stays constant for a whole "cycle" (for a given value of the exponent), but drops (we might say, "suddenly") to half its size for the next lower cycle (with an exponent of one less).

Thus, which "cycle" a representation is in affects its precise value. This is the nature of a system with a "step size jump".

Logarithmic schemes, an alternative to floating point schemes, have a step size that decreases linearly with value; there are no "cycles". In these, by the way, the precision score is constant throughout the entire range.

THE IEEE-754 FLOATING POINT REPRESENTATION SYSTEMS

Several highly-specified binary floating point systems, intended for use in computers or data storage systems, are defined by IEEE standard IEEE 754-1985.

These principally differ in terms of the total number of bits used to store the presentation, and thus in the precision and range of the representation.

The "baseline" scheme uses 32 bits to store the number. We will use it to illustrate the principles shared by all the schemes.

The 32-bit scheme

The IEEE-754 32-bit scheme uses the "normalized mixed significand" approach. Here the significand, **s**, must lie in this range (expressed in decimal):

$1 \leq \mathbf{s} < 2$

That is, its value in binary must be in the range 1.0 through 1.11111...

The significand is conceptually 24 bits in length, but because the first bit (the units digit) is always 1, we need not store it, and thus use only a 23-bit field to carry the "fractional part" of the significand.

A separate sign bit is used. There is no notion of "twos-complement" or anything like that. O means positive, 1 means negative, and the magnitude is given by the fraction field just as it seems.

In this scheme, the range of the exponent is -126 through +127. Rather than using a sign bit, we use an 8-bit field (range 0-255), which carries a value of 127 greater than the exponent (an "exponent offset", or “exponent bias”, of +127).

Thus an exponent field value of 0 means an exponent of -127; a field value of 127 means an exponent of 0; and a field value of 255 means an exponent of +128. Looks likes we have wasted some values: 0, meaning an exponent of -127, and 255, meaning an exponent +128. Those exponent values are not used. Why? Well, we have a couple of "funny things" to do, and these values are reserved as ingredients in doing them.

The total number of bits per value is 32: one *sign bit* for the significand, 23 bits for the *significand fraction*, and 8 bits for the *exponent*.

Representing zero[4]

We have to do something special to represent the value 0, since the significand cannot become 0 as would be needed to represent 0 in the usual way.

For the value 0, we code all 0s in the significand fraction field (which would ordinarily mean a significand proper of 1.0000) and 0 in the exponent field (which would ordinarily mean an exponent of -127). This whole thing is recognized as meaning a value of 0. (The significand sign bit still works, so we can have +0 and -0 if that does anything for us.)

What would that set of field encodings ordinarily mean? It would mean:

$1.0000000000\ldots \times 2^{-127}$.

This is outside the legitimate exponent range of the scheme, so this encoding would never be used with that meaning. It can thus be unambiguously recognized as the secret code for 0.

Another complication

The wrinkle

There is another important wrinkle.

---

[4] Sounds like a perfect title for a Monty Python skit—think “Summarizing Proust”.

The four smallest non-zero positive numbers than can be represented are:

$+1.00000011 \times 2^{-126}$

$+1.00000010 \times 2^{-126}$

$+1.00000001 \times 2^{-126}$

$+1.00000000 \times 2^{-126}$

The next lower value that can be represented is:

+ 0 (via a special code)

So the values just above zero are each separated by:

$0.00000001 \times 2^{-126}$

but the lowest of them is separated from the next lower value (zero) by:

$1.00000001 \times 2^{-126}$

an increment 257 times as large. So it is as if there is a really big "dead zone" near zero.

This can cause a problem in calculations. If we subtract two quite distinct small non-zero values, for example:

$(+1.01000010 \times 2^{-126}) - (+1.00000010 \times 2^{-126})$

we get $+0.01000000 \times 2^{-126}$ (to write it in an easily compared form).

But there is no representation for that: −126 is the smallest possible exponent, so we would need to use the significand shown, but we can't: we cannot have a significand that does not have "1" in the units place. So that result would have to be "rounded down" to zero (and be represented by the special code for that)."Discarded" would be a more candid description of its destiny.

Then, even though the subtraction produces a non-zero answer, that is lost. This can result in divide-by-zero faults when there should not be such (any small non-zero value would avoid that), and so forth.

The solution

This dilemma would not exist if, once we got into this region, the significand could lose its "enforced 1" in the units place. The significand could then just keep dropping to whatever was needed—in the subtraction example above, to 0.01000000 (while the exponent stayed at -126).

But the units "1" is not stored, so we can't just leave it off.

Rather, some special coding is needed that would say to the "interpreter", "**do not** add 1 to the decoded significand fraction to get the significand—use the fraction as the significand".

For these values, the exponent field carries all 0s (0) (which would ordinarily mean an exponent of -127), while the significand fraction field carries the actual significand. It is of course never 0, so this situation can be distinguished from the special coding for 0 (where the significand fraction field carries 0).

Recognizing this tells the "interpreter":

- Do not add 1 to the decoded significand fraction to get the significand—use the fraction, as it appears, as the significand.
- Consider the exponent to be -126.

The resulting specially-represented values have been called in the past *denormal values*, or sometimes *denormalized values*.) Neither term is really apt, and in the 2008 version of IEEE-754, they are called "subnormal values".

Infinity

There are a few code combinations not yet used, and rather than wasting them, they have been assigned to some other useful special things.

It can sometimes be useful to be able to represent infinity (with a choice of algebraic sign) in place of an actual numeric value.

For either sign of infinity we put all 1s (255) in the exponent field (a value never used otherwise) and all 0s ("0") in the significand fraction field. The significand sign bit still works so we can have both positive and negative infinity.

Not a Number

Finally, we wish to represent the pseudo-value "not a number" (NaN), used to record a failed or meaningless result of a calculation, or as a filler in synchronous data transmission to mean "no data here yet".

For that we put all 0s (255) in the exponent field and anything except all 0s (0) in the significand fraction field. The significand sign bit is allowed to have either value, but that rarely has any significance.

Summary of the properties

Range

The largest magnitude numbers (excluding infinity) that can be represented in this scheme are about $\pm 3.4 \times 10^{38}$.

The smallest magnitude non-zero numbers (not using the subnormal range) that can be represented are about $\pm 1.18 \times 10^{-38}$.

The smallest magnitude numbers (using the subnormal range) that can be represented are about $\pm 1.4 \times 10^{-45}$. The corresponding relative range is about 2.43×10^{83}.

Precision

The precision score at the top of each "cycle" is about 16.8 million. At the bottom of each cycle, it is about 8.39 million.

The 16-bit version

In the 16-bit version ("half-precision"), the significand fraction field is 10 bits in size (so the significand itself has 11 bits), and there is a sign bit for the significand. The exponent field has five bits. The range of actual exponent values is -14 through +15 (the exponent offset is -15). The zero, subnormal, and infinity values work just as in the 32-bit form.

The largest (absolute) representable values are ± 65504.

"MANTISSA" VS. "SIGNIFICAND"

Relationship to logarithms

The common (or base 10) logarithm of a number is the power to which 10 must be raised to equal the number.

Thus the common logarithm of 4500 is (to 4 decimal, places):

3.6532

The portion ".6532" is called the *mantissa*[5] of the logarithm, and the portion "3" is called its *characteristic*.

---

[5] The term was introduced by Briggs, who introduced the concept of the common (base 10) logarithm. The word, in Latin, means "something added on"; in Briggs' view the *mantissa* was added on to the major part of the logarithm, the *characteristic*. In fact the full connotation of the word mantissa was "something added on that adds no real value; a makeweight." Briggs should have better stuck to his mathematics, and left the adaption of Latin words to those qualified to do such.

That same number, 4500, could be represented in scientific notation as:

4.500×10^3

Here, the quantity 4.500 is the *significand* of the scientific notation.

But we could legitimately write that, with only a small error, as:

$10^{0.6532} \times 10^3$

This still follows the structural principle of scientific notation (of course not in the way we usually use it).

In this form, the quantity $10^{0.6532}$ is the *significand* of the scientific notation. Calling that the *mantissa* essentially says it is equivalent to the quantity 0.6532, which **is** the mantissa of the equivalent logarithmic form. But these are of course quite different quantities.

Thus the shift away from the use of "mantissa" to identify this component of scientific notation and the adoption instead of "significand".

But where does that term come from?

About "significand"

The term *significand* for a certain part of both scientific notation and floating point representation is called that because it describes the *significant digits* of the number being represented.

Here is an example in a decimal context. These numbers:

1546

1.546

0.1546

0.000000001546

all have the same significant digits: 1-5-4-6. (15460000 would as well, if we declared it to have four significant digits.)

Thus, in scientific notation (of the most common normalized significand type), for all those numbers the significand would be:

1.546

IEEE DECIMAL FLOATING POINT REPRESENTATIONS

The floating point representations of IEEE-754-1985 fully reflect their binary nature. Among other things, this means that many numbers often encountered in such fields as accounting (such as 149.99) do

not have a precise binary representation in a finite number of digits. That number in binary is (the underlined part repeating infinitely):

10010101.11<u>1111101011100001010 00</u>...

Thus, such "ordinary" rational numbers nevertheless may often suffer from rounding error when represented in binary form (including in floating point schemes). Of course, the amount of error is often miniscule, but its accumulation can result in "failure to close" in accounting work.

The later version of the standard, IEEE-754-2008, includes three "decimal-friendly" floating point representations. Although they still assume a binary representation in the computer or data record, they are organized so as to be "decimal" in their outlook. In particular, they allow numbers such as 149.99 to be represented without rounding error, and in fact essentially all unavoidable rounding honors decimal rounding precepts.

The details of these three schemes, with two flavors of each, are complicated, and are beyond the scope of this article.

ACKNOWLEDGMENT

Thanks to Carla Kerr for her insightful copyediting and proofreading of this difficult manuscript.

#