Association for Computing Machinery
Advancing Computing as a Science & Profession

*Ubiquity,* an ACM publication
September 2016

# Unums 2.0

An Interview with John L. Gustafson

***by Walter Tichy***

## Editor's Introduction

*In an earlier interview (April 2016),* Ubiquity *spoke with John Gustafson about the unum, a new format for floating point numbers. The unique property of unums is that they always know how many digits of accuracy they have. Now Gustafson has come up with yet another format that, like the unum 1.0, always knows how accurate it is. But it also allows an almost arbitrary mapping of bit patterns to the reals. In doing so, it paves the way for custom number systems that squeeze the maximum accuracy out of a given number of bits. This new format could have prime applications in deep learning, big data, and exascale computing.*

*Walter Tichy*
*Associate Editor*

# Unums 2.0

An Interview with John L. Gustafson

***by Walter Tichy***

We recently interviewed John Gustafson about a new floating point format, the unum, or universal number. What's interesting about the unum is both the mantissa and the exponent may vary in the number of bits during runtime, depending on accuracy. There is a bit that indicates whether the unum is exact, and if it is not, then the number is only inaccurate in the last digit. So we always know how exact a result is. If it happens to be exact to only three digits, then there will be only three digits in the mantissa. If that's not precise enough, we may have to start with higher accuracy in the inputs

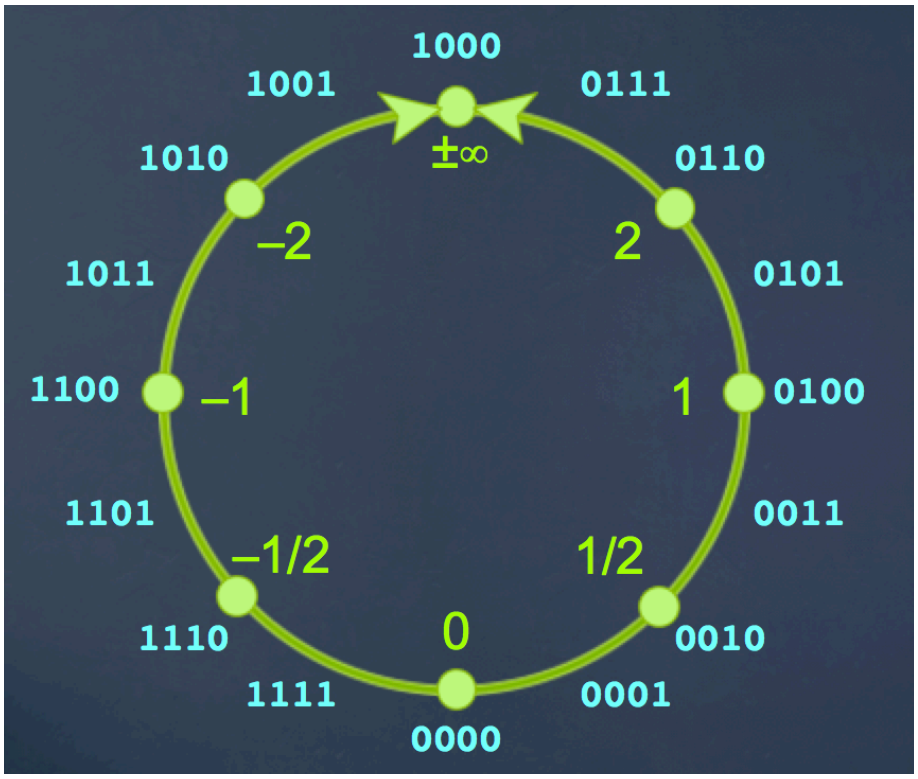
**Walter Tichy: Shortly after our interview, you came out with "unums 2.0." What is different about them? Apparently, you are doing away with exponents altogether, which would be a radical break with traditional floating-point numbers.**

**John Gustafson:** I started out calling them "unums 2.0," which seemed to be as good a name for the concept as any, but it is really not a "latest release" so much as it is an alternative. It might be more accurate to call them "Type 2 unums," and the original ones "Type 1." They share about 80 percent of the mathematical advantages that Type 1 unums have, such as the ability to avoid rounding error, overflow, and underflow. But they are crafted to a different esthetic: speed and simplicity, both for the hardware designer and the programmer who wants to use them. They also allow a user to design a custom number system for a particular workload, something the deep learning community is particularly interested in.

People universally worry about the variable size of the original unums, and I fully understand their worry. The new unums are fixed size but crafted to squeeze as much information as possible into every bit. They have some very attractive properties, like decimal representation with no performance penalty, and the ability to take an exact reciprocal of a number just as quickly and easily as we can negate a number now. Addition, subtraction, multiplication, division, and even powers, that is, *x* to the *y* where both *x* and *y* are real, can all be done in a single clock cycle. There is only one way to represent a particular real value, whereas in Type 1 unums there are usually multiple ways, and that creates programming headaches when you want to check if two unums represent the same value. IEEE floats have that problem too, since there are two ways to represent zero.

What I realized four months ago was that I had put a great deal of effort into maintaining compatibility with the IEEE 754 floating-point standard, making unums a superset of that format and able to do the same things as a subset of their capabilities. I thought the best way to ease people into a new way to compute with real numbers would be to give them an upward-compatible format and let them convert codes to use the new capabilities in a gradual way. But then I noticed that a "clean slate" design of a number system crafted for 2016-era computers would not look much like the IEEE Standard from over thirty years ago. No surprise there, right? Computers are about a million times faster (or a million times cheaper, however you want to think about it) than they were when the IEEE 754 rules were crafted, so it seems unlikely that the engineering tradeoffs we have now would be decided anything like the way they were over 30 years ago!

Type 2 unums are a direct map of signed integers to the *projective real number line*. The projective reals map the reals onto a circle, so positive and negative infinity meet at the top. Here's a picture of what the new formulation looks like if you only had four bits but wanted to represent the entire extended real number line.



It's ultra-low accuracy, but mathematically clean, and the hardware designers are going to love this one. To negate a unum, you negate the integer associated with the bit string, as if that integer was a standard two's complement number. Flip the bits and add one, ignoring any overflow; that gives you the negative of an integer. It works with no exceptions. But get this: To *reciprocate* a unum, you ignore the first bit and negate what remains! Geometrically, negating is like revolving the circle about the vertical axis and reciprocating is revolving it about the horizontal axis. And yes, the reciprocal of zero is ±∞ and vice versa.

No more "negative zero" redundant representations to handle. No more "denormalized" (or "subnormal") values. Dividing becomes as simple and fast as multiplying, just as subtraction has the same cost as addition in current float arithmetic. And you don't have to make exceptions for dividing by zero. That's rather liberating, don't you think?

Notice that exact numbers have bit strings ending in zero and open intervals between exact numbers have bit strings ending in one. This is just like Type 1 unums. Also, the first bit still acts like a sign bit, so long as you remember that both zero and ±∞ are unsigned. Other than that, there is no exponent field, like you say. Instead we have a "*u*-lattice" of exact numbers between 1 and ∞ that defines the meaning of the bit strings. Here, that *u*-lattice is just the number 2, but I can show you some more interesting *u*-lattice choices later on.

**WT: That brings up a lot of questions. First, I noticed you also did away with time-honored radix or positional representation: The value of a bit in position *i* is no longer $2^i$. I also noticed that every number and interval is complemented with its inverse, which makes inversion easy. But how do you choose the numbers and assign the bit patterns so negation and inversion work the way you described?**

**JG:** Here's the thing: It's extremely flexible, and can be defined according to the needs of the computer user instead of being decided by a vendor or a standards committee. As I said, I call the selected set of exact real numbers between one and infinity the "*u*-lattice." You can optimize it for dynamic range, for digits of accuracy, or even to include numbers important to a particular application. Suppose you work with $\pi$ a lot. Well, $\pi$ is an exact number, just one that we cannot express with a finite string of decimals. Just define it as one of the points in the *u*-lattice, and now you can work with $\pi$ as an exact number.

I was just talking with some astrophysicists in Australia working on the Square Kilometer Array project, and they are doing discrete Fourier transforms on data that is ultra-low precision. Input data can be –1, 0, or 1, and that's it. I understand the oil and gas people doing seismic exploration are similarly happy to use very low precision inputs, so long as the Fourier transform is accurate. I noticed that if you do an eight-point FFT on every possible set of such data, there are only 57 possible answer values! Which means you could represent the answer *exactly* with only six-bit data. Do you see where I'm going with this? We can have *software-defined number systems*, and a compiler can generate them to be optimal for a particular application program, loading in the tables just as it loads in the binary instructions now.

The catch, and it's a big one, is that this works well for low precision but quickly becomes unwieldy for, say, 32-bit and 64-bit precision. But these are early days for this idea and I think we may discover ways it might be made practical for higher-accuracy computations.

**WT: Could there be unums to the base of 10? Would arithmetic still be exact?**?

**JG:** Yes, and that's one of the best things about Type 2 unums. There is no performance penalty for using base 10! If you have, say, a six-bit unum, you might choose a lattice like this to cover a decent-sized dynamic range:

$$1 < \{2, 5, 10, 20, 50, 100, 200\} < \infty$$

That sets the upper right quadrant of the circle. Looks like the values used for money, doesn't it? The reciprocals are all easy to write in decimal, notice:

$$0 < \{0.005, 0.01, 0.02, 0.05, 0.10, 0.20, 0.5\} < 1$$

This gives the exact numbers in the lower right quadrant. Combine those with their negatives to get the left half of the circle, and finally, include the open intervals *between* the exact points, so that no real numbers are left out. That creates a total of 64 numbers or number ranges that perfectly cover the projective real numbers. Not that I'm recommending weird data sizes like six bits, but I'm trying to keep things simple here, and that means using very low precision.

Here's just a small excerpt of how these numbers map to the unum bit strings, where I use some color-coding of the binary values:

| | | |
|---|---|---|
| 1 | **010000** | The unum for exact 1 always begins with "**01**", then all **0**s. |
| (1, 2) | **010001** | Open intervals between exact values always end in "**1**". |
| 2 | **010010** | Positive numbers and zero start with "**0**". |
| ⋮ | | |
| 200 | **011110** | This is the largest exact positive real value. |
| (200, ∞) | **011111** | Instead of overflowing to infinity, too-large numbers fall here. |
| ±∞ | **100000** | Negative numbers and ±∞ start with "**1**". |
| (–200, –∞) | **100001** | Keep counting up to wrap to the left half of the circle. |
| ⋮ | | |
| –1 | **110000** | The unum for exact –1 always begins with "**11**", then all **0**s. |
| ⋮ | | |
| (–0.005, 0) | **111111** | The unum for the smallest negative range is always all **1**s. |
| 0 | **000000** | The unum bit string for zero is always… zero! |
| (0, 0.005) | **000001** | Instead of underflowing to zero, too-small numbers fall here. Now we're climbing back up the right half of the circle. |

The arithmetic operations are not *exact*, though in a remarkable number of cases, you can perfectly describe the result using these values. If you ask for 2 + 2 at this very low precision, there is no exact representation for four, so you have to use the open interval (2, 5), for example.

**WT: There is also the concept of sets of numbers. What is this for?**

**JG:** There are a lot of mathematical problems for which the answer is not a single number. Even something a simple as "What number, when squared, equals nine?" has two answers: positive three and negative three. Or "What numbers are odd integers?" Or "What real values are strictly less than ten?" You may have heard of interval arithmetic, where you represent numbers as being between *a* and *b*, where *a* and *b* are values that have an exact representation in your number system. Being able to work with any subset of the real number line as if it were *itself* a numerical quantity is a powerful thing.

I hate showing people large bit strings, but, well, I'm going to show you a large bit string, 64 bits long. I'll use some color-coding and spaces to help the readability. You know that table I just showed you for the "currency" type of unum? There are 64 possible quantities represented, right? I can represent a *set* of them by using a **1** if the quantity is present, and a **0** if it is absent in the set. I call them SORNs, for Sets Of Real Numbers. Here is how I would represent a SORN that has the numbers 0, 2, and 50, and also the open interval (2, 5):

**0000 0000 0000 0000 0000 0000 0000 0000 1000 0110 0000 0100 0000 0000 0000 0000**

Negative numbers are red, positive numbers are blue, but zero and ±∞ are black. See how there is a black **1** in the center? That means the number 0 is present in the set. A few bits to the right, there are two more: **11**, which indicate the presence of the number 2 and the open interval (2, 5). The presence of the number 50 in the set is that rightmost **1** bit. You can compute with SORNs as if they were numbers, because to take the union of two of them, you OR the bits. To find the intersection, you AND the bits. It's very close to native computer hardware logic, instead of requiring symbolic storage and symbolic manipulation of sets. And it can be done in parallel. It's not like adding and multiplying bit strings, where you have to propagate carries from right to left, which means SORN operations can be very fast.

**WT: There is no NaN, or Not-a-Number, in this format. What happened to that?**

**JG:** The NaN is replaced here with something I think is much more informative, and it is made possible by the SORN idea.

Suppose I try to take the square root of negative one. What real values, when squared, are equal to negative one? There aren't any, so in floating point you would get a NaN. Instead, the SORN result is *the empty set*:

**0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000**

All zero bits. There are *no* quantities in the set.

What if I try to compute zero divided by zero, a well-known "indeterminate form"? If you take the limit of *x* divided by *y* as both *x* and *y* approach zero, you get *every possible number* as well as ±∞. Well, there's a SORN for that, too:

**1111 1111 1111 1111 1111 1111 1111 1111  1111 1111 1111 1111 1111 1111 1111 1111**

If you tried to compute $1^{\infty}$, again a float environment would return a NaN, Not-a-Number. There's not a lot of information there. What if instead you could at least say that the value is *any non-negative value*? Because that's what the limit of *x* to the *y* is, as *x* approaches 1 and *y* approaches infinity. The SORN can return:

**1000 0000 0000 0000 0000 0000 0000 0000  1111 1111 1111 1111 1111 1111 1111 1111**

because now we have the vocabulary to describe *sets* of real numbers, instead of having to pick a single rational value (a floating point number) as the answer to every mathematical question.

**WT: Does it matter that there is no distinction between plus and minus infinity?**

**JG:** A little, yes. I take some comfort in the fact that I still have the open intervals (*maxreal*, ∞) and (–∞, –*maxreal*). So as *limits*, I can still make the distinction. For example, if I take the logarithm of the open interval (0, 1), I get the open interval (–∞, 0), and there's a SORN for that. But I lose the ability to compute the logarithm of exact zero as negative infinity. I think that's a small price to pay for the elegance of the formulation, and for eliminating "negative zero," one of the most confusing and dubious entities in the IEEE 754 definition of floating-point arithmetic.

**WT: Now for the hard stuff: How does arithmetic work? Plus, minus, multiply, divide, all in one cycle?**

**JG:** Remember, I said I only have an answer for low precision, and it leans heavily on table look-up. Let me show you what the addition table looks like if you only have two bits for the unums. With two-bit unums, you have only four quantities you can represent:

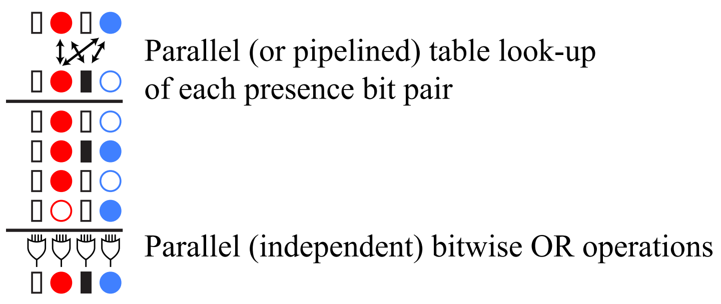
| 00 | 01 | 10 | 11 |
|---|---|---|---|
| Zero | Positive reals | ± Infinity | Negative reals |

The addition table shows what SORN results from adding all 16 possible combinations of these four input unum values. To make it a little less confusing whether a binary string describes a unum or a SORN, I'll switch to a different notation for the SORN that makes use of shapes. Rectangles are for exact values and circles are for the open intervals between exact values. They are filled if the unum is present and hollow if the unum is absent. I color them as before, with zero and ±∞ black, negative unums red, and positive unums blue.

| $+$ | $x = \pm\infty$ unum 11 SORN ▮○▯○ | $-\infty < x < 0$ unum 10 SORN ▯●▯○ | $x = 0$ unum 00 SORN ▯○▮○ | $0 < x < \infty$ unum 01 SORN ▯○▯● |
|---|---|---|---|---|
| $y = \pm\infty$ unum 11 SORN ▮○▯○ | $[-\infty, \infty]$ SORN ▮●▮● | $\pm\infty$ SORN ▮○▯○ | $\pm\infty$ SORN ▮○▯○ | $\pm\infty$ SORN ▮○▯○ |
| $-\infty < y < 0$ unum 10, SORN ▯●▯○ | $\pm\infty$ SORN ▮○▯○ | $(-\infty, 0)$ SORN ▯●▯○ | $(-\infty, 0)$ SORN ▯●▯○ | $(-\infty, \infty)$ SORN ▯●▮● |
| $y = 0$, unum 00 SORN ▯○▮○ | $\pm\infty$ SORN ▮○▯○ | $(-\infty, 0)$ SORN ▯●▯○ | 0 SORN ▯○▮○ | $(0, \infty)$ SORN ▯○▯● |
| $0 < y < \infty$ unum 01, SORN ▯○▯● | $\pm\infty$ SORN ▮○▯○ | $(-\infty, \infty)$ SORN ▯●▮● | $(0, \infty)$ SORN ▯○▯● | $(0, \infty)$ SORN ▯○▯● |

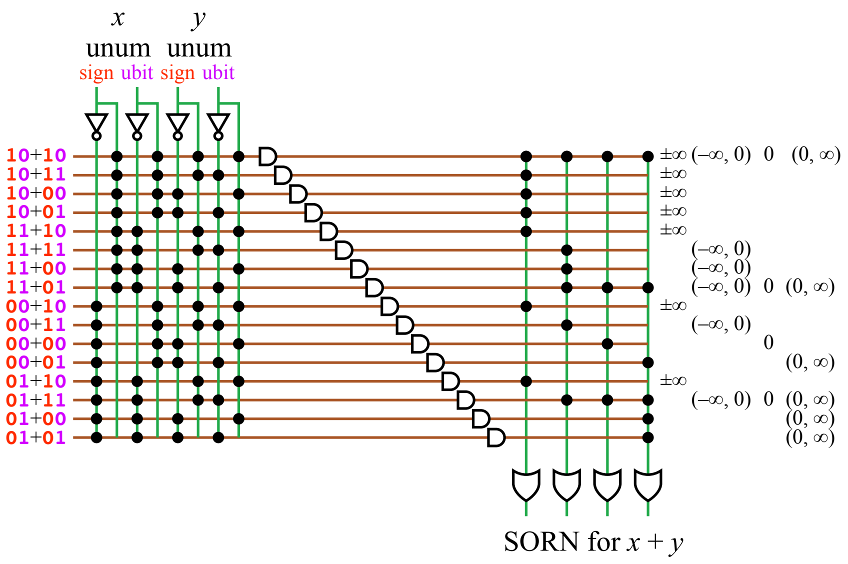
Notice the three entries that are highlighted in light green. Those indicate where the SORN has more than a single unum present in the sum.

Now remember how you learned to do multi-digit multiplication in elementary school. You do it with pairs of digits in each number, using your memory of multiplication tables for zero to nine, and then sum them all up. If we want to do an operation on two SORNs, you treat each present unum as a digit, look up in the table what the operation is on every unum pair, and then OR all the table entries together. Here's a diagram that show how to add, say, the SORN for the set of nonzero reals ▯●▯● to the SORN for the set of nonpositive reals ▯●▮○:


Parallel (or pipelined) table look-up
of each presence bit pair
Parallel (independent) bitwise OR operations

The higher the precision, the more you have to use algorithmic methods that look like float algorithms, and then I need more than one clock cycle. But I think you can see how fast these operations can be, since they consist of independent OR gates running in parallel.

The prototype environment I am developing allows flexible definition of the *u*-lattice exact values, and then it automatically populates the tables. Table look-up in hardware can be extremely fast, like three gate delays. Here's what the function generator would look like for two-bit unum inputs and four-bit SORN addition:



I'm assuming each black dot in the circuit is simply a wired connection, not a switch. If it were a programmable table, it would require far more transistors, of course, and it would take a lot more electrical power. One of the earliest commercial computers, the IBM 1620 Model 1, used table look-up for all of its arithmetic. That came out in 1959; IBM used the internal code name "CADET" for the computer, and when customers found out that it used tables instead of logic to do arithmetic, they joked that CADET stood for "Can't Add, Doesn't Even Try."

That can all be done in parallel, up to some size. There are shortcuts you can use as the size of the operands gets larger, and finding the right balance of algorithmic methods and look-up methods is what I'm working on now.

**WT: I'm confused—are we doing arithmetic in unums or in SORNs? The above table goes from unums to SORNs. Once we have SORNs, do we continue with SORNs?**

**JG:** The result of doing an arithmetic operation on a unum might be another unum, but in general it is a contiguous block of unums, like an interval. The answer spreads out in general, though there are things that can shrink the interval, too, like dividing by two. So you do not get a closed system unless you go all the way up to SORN representation. The unum-unum operations are what a computer uses to figure out a SORN, just as knowing the multiplication tables from zero times zero to nine times nine lets you multiply multiple-digit numbers together, one pair of digits at a time. Think of unums as digits, but digits that are not restricted to being counting numbers.

**WT: If we work with larger unums, the SORNs are going to explode. For example, if we have 16 bit unums, then we'll need $2^{16}$ bits for a single SORN. That's eight Kbytes. Is there a compression method for the cases when there are few intervals or they are contiguous?**

**JG:** If you have a single interval, that's a contiguous set of unums. It could be a single unum. Under plus-minus-times-divide, contiguous SORNs *stay contiguous*. That holds even if you divide by zero,  because of the way the projective reals wrap around the top. To store a contiguous SORN with $n$-bit unums, you can always do it with just $2n$ bits, not $2^n$ bits. One way is to store the first unum of the contiguous block, and the number of consecutive 1s. Another is to store the first and last unum in the contiguous block. No matter how you do it, there are enough bit patterns in $2n$ bits to store all the possible contiguous SORNs from empty set to the entire set of extended real numbers. By the way, you can write an interval like (3, 2] where the left endpoint is *greater* than the right endpoint. It means the complement of (2, 3]. You start just to the right of the number 3, go clockwise up to ±∞ and back down through the negative reals to zero, then up to two.

So contiguous SORNs not only take a small number of bits to store, it's always the same number. Remember that was part of the idea of the alternative design of Type 2 unums: Keep the storage format a constant size.

If you really want a fully general SORN, then you have to either use run-length encoding and have variable size storage for the SORNs, or bite the bullet and, as you say, use 8 Kbytes for each SORN, in the case of 16-bit unums. I would argue that eight Kbytes really isn't very expensive these days, and you could fit thousands of such SORNs in on-chip level one cache. Maybe you use run-length encoding or something even more efficient to pack up the SORNs when you write them to memory or mass storage, but unpack them into the larger fixed size for purposes of computing with them. These are the types of tradeoffs we need to work out. When you think about it, however, it's pretty amazing to be able to perform computations on *subsets of the entire extended real number line* as your input and output variables, with only eight Kbytes for each variable.

**WT: Can you also give us an idea about exponentiation (with real numbers as exponents)? Not that I need this every day, but exponentiation in a single cycle sounds absolutely amazing.**

**JG:** Taking $x$ to the $y$ power is simply another function of two input values, so the same look-up table idea works. Also with a SORN, you have the option of representing multiple output values, like $x^{1/2}$ can return both $\sqrt{x}$ and $-\sqrt{x}$ . The power function for floats is littered with exception cases that library routines have to watch out for, so conditional branches slow down evaluation. The beauty of table look-up is that it pre-computes everything so you have no conditional branches at run time. It also eliminates "The Table-Maker's Dilemma" that certain table entries take much longer than others to determine correctly. Once they are determined, looking them up should only take one cycle. But remember, this only works up to a small number of decimals of precision at this point.

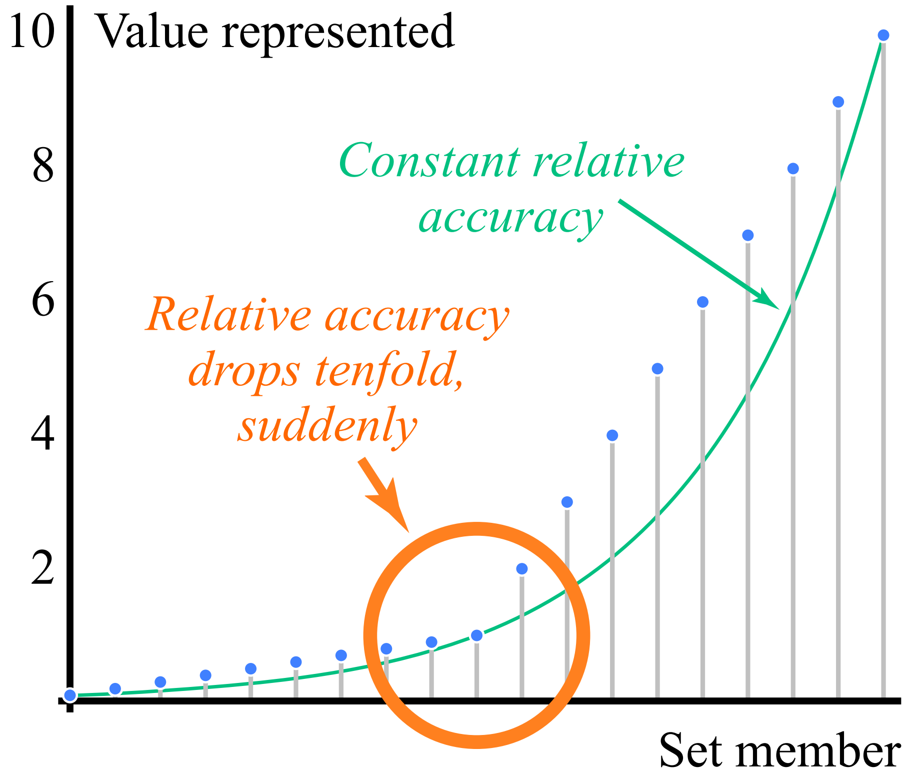
The really intriguing thing is how fast and easy it is to use the table look-up approach for single-argument functions like logarithms and trig functions, or even much more complicated stuff like Bessel functions. The Deep Learning programmers spend a large amount of time computing this sigmoid function:

$$S(t) = \frac{1}{1 + e^{-t}}$$

and that can easily take a hundred clock cycles. Many Deep Learning tasks only need about 16-bit precision for the training of the neural net, which is why vendors are starting to support half-precision floats in hardware. But with type 2 unums, in 16 bits, I can compute that sigmoid in a single clock with no exception handling and no "Table Makers Dilemma," and only spend a few kilobytes for the table. The tables are tiny compared to the ones for arithmetic, because they're one-dimensional, just a list, instead of a two-dimensional table. I envision a standard set built into a chip, and some RAM for user-definable functions. Imagine that you have some very complicated function of one variable that you use over and over in a program, and it takes a hundred floating-point operations to evaluate. A low-precision *float* table won't work well because the rounding error will kill you. But if you can guarantee containment of the answer the way unums and SORNs do, then for some situations you might get a very acceptable answer, a hundred times faster than using floats. Which sounds to me like a shortcut to exascale levels of computing.

**WT: Could unums also handle good-old decimal numbers?**

**JG:** What if I told you that decimal unums are not only as fast as binary unums, but *superior* in preserving accuracy? This is a breakthrough, because as you know, IEEE Standard decimal floats are about half the speed of binary floats, at best, and they have a problem with "wobble," meaning that the accuracy is uneven from one number to another. For instance, if you have three-decimal floats, then near the number 100 you have 99.6, 99.7, 99.8, 99.9 just below 100, but then 101, 102, 103 just above 100. The spacing jumps from 0.1 to 1.0, a sudden accuracy "wobble" of a factor of ten. Unless you know where on the real number line your application generates numbers, the best thing is to space numbers so the *relative accuracy* is constant, which means they approximate an exponential curve. I like to define *relative decimal inaccuracy* as the log base ten of the ratio of adjacent *u*-lattice values. For example, the relative decimal inaccuracy going from 99.9 to 100 is $\log_{10}(100/99.9)$, which is about 0.00043. The relative decimal inaccuracy going from 100 to 101, though, is $\log_{10}(101/100)$, which is about 0.0043, ten times more inaccuracy. You can also define the number of decimals of accuracy at that point, as the negative log base ten of the relative decimal inaccuracy. For the same example, $-\log_{10}(0.00043)$ is about 3.4, so there are about 3.4 decimals of accuracy going from 99.9 to 100. But that drops to $-\log_{10}(0.0043)$ which is about 2.4 decimals, going from 100 to 101. Here's a graph of single digit decimals from 0.1 to 0.9 and 1 to 10 so you can see the sharp "kink" in the range:

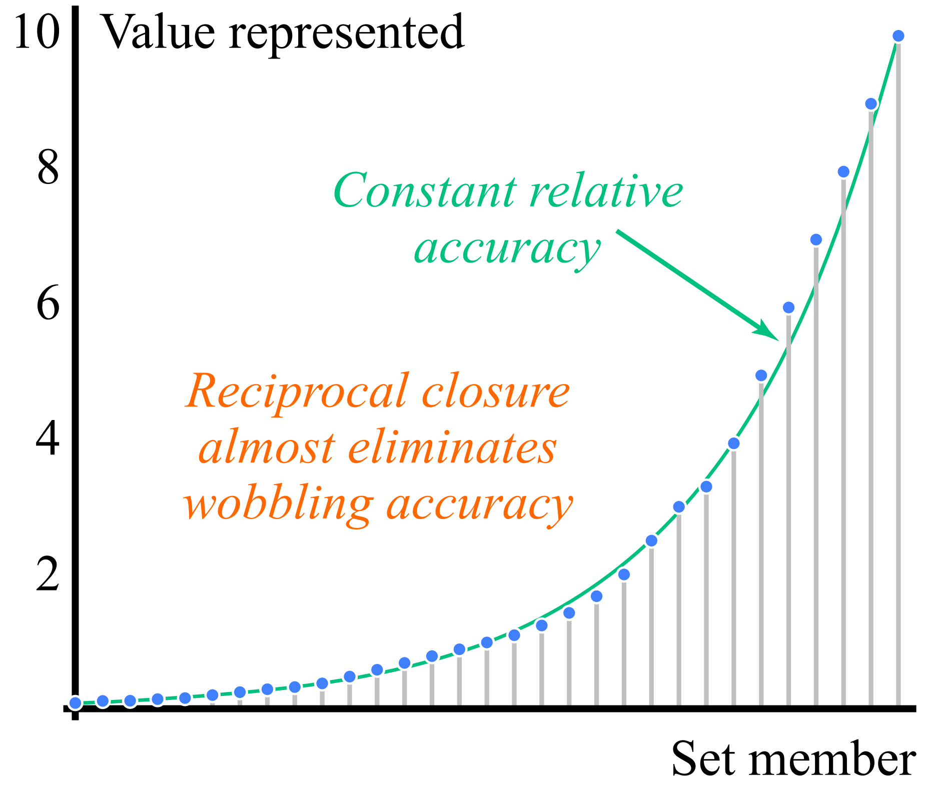


Remember how Type 2 unums have exact reciprocals? In going from 0.1 to 10, single-decimal unums have some exact values that floats do not. Since the number 3 is in the *u*-lattice, so is the number 1/3, exactly. That's what I call "reciprocal closure." If you take the numbers 0.1 to

0.9 and 1 to 10 and unite them with their reciprocals, and sort them into increasing order, you get this *u*-lattice:

0.1, 1/9, 0.125, 1/7, 1/6, 0.2, 0.25, 0.3, 1/3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9,
1, 10/9, 1.25, 10/7, 10/6, 2, 2.5, 3, 10/3, 4, 5, 6, 7, 8, 9, 10

Guess what happens when you plot that set of values? As a side effect of fixing division, we also fix most of the "wobble"!



I was amazed when I saw that, and I thought, "There has to be a catch. What's the catch?" Whenever anyone thinks they've found a better way to represent numbers, there always seems to be a downside. I figured it was reduced dynamic range. So I tried comparing 16-bit decimal unums with 16-bit binary IEEE floats, the "half precision" floats that Nvidia has pioneered.

The binary floats have the equivalent of a little more than three decimals of precision. The normalized floats have a dynamic range of nine orders of magnitude, from about $6 \times 10^{-5}$ to 6 to $10^{4}$, which is not exactly symmetrical about one. To compare, I created a *u*-lattice of three-decimal values from 1.00 to 9.99, made them closed under reciprocation and included the ubit to distinguish between exact values and the open intervals between them, so I figured Type 2 unums were already at a disadvantage. But I was amazed again when I found that they had a dynamic range from $1/0.389 \times 10^{-5}$ to $0.389 \times 10^{5}$, which is slightly *larger* than the dynamic range for half-precision floats. Part of the reason is that unums do not use over a thousand bit patterns to represent various types of NaN, and the other part is that unums are more information-efficient in being closer to exponential spacing like the graph shows.

I'm really looking forward to experiments with 16-bit Type 2 unums that are decimal, can represent the real number line with proper mathematics, never lose accuracy under reciprocation, and *fast*, like Ludicrous Mode on a Tesla automobile.

**WT: The Type 1 unums have attracted quite a following already. What do you think will happen to Type 2 unums, seeing as they are a radical departure from the traditional floating-point format?**

**JG:** Right now, it looks tough to extend Type 2 unums to high precision, which means we still need Type 1. The "killer app" for Type 2 unums looks to me like Deep Learning. Deep Learning applications are the main reason Nvidia has put native half-precision float operations into its latest generation of GPU accelerators. Not only do Type 2 make better use of 16 bits than floats do for the training of neural networks, they also can go even smaller and even faster. Also, 16-bit floats have more dynamic range than Deep Learning applications require, so you could get better accuracy over a smaller dynamic range with Type 2 unums. I have a suspicion that even 8-bit unums could accomplish much of the training tasks, especially in the beginning of each training. After all, that's even more accuracy than our own brain neurons have, isn't it? But I'm no expert in deep learning, so I defer to those who have been working in that area.

**WT: Lookup tables certainly make arithmetic faster, but we could use tables for half-precision IEEE formats as well. How big are the tables for 16-bit, Type 2 unums? Are they significantly smaller than for IEEE formats?**

**JG:** That's complicated. And it's an interesting question! With unums, you only need entries for the exact values, so the two-argument function tables are one-fourth the number of entries as if you had an entry for every bit pattern. Floats are designed to be easy to work with by algorithmic methods, where you have separate exponent and fraction bits and know how much to shift and how to apply integer operations to the pieces; it *might* be faster to perform 16-bit float operations with a table, but the table might take more space on chip than conventional floating-point hardware. If I were still directing research at Intel, I'd get a team to try the experiment and find out!

**WT: What do you see as the future of Type 2 unums?**

**JG:** I think the Type 2 unum approach is a shortcut to the exascale performance levels we need for some applications. They can help train neural networks for artificial intelligence, and they might save millions of dollars in the storage needs of big data problems. But what really excites me is the idea of number systems customized to applications, systems that maximize the information from every bit of precision instead of the inefficient "one size fits all" fixed precision and dynamic range of a standard float type.

## Suggested Readings

J. Gustafson. A Radical Approach to Computation with Real Numbers. *Supercomputing Frontiers and Innovations* 3, 2 (August 2016).

W. Tichy. The End of (Numeric) Error: An Interview with John Gustafson. *Ubiquity* April 2016.

## About the Author

Walter Tichy has been professor of Computer Science at Karlsruhe Institute of Technology (formerly University Karlsruhe), Germany, since 1986. His major interests are software engineering and parallel computing. You can read more about him at www.ipd.uka.de/Tichy.

**DOI:** 10.1145/3001758