The End of (Numeric) Error

An Interview with John L. Gustafson

by Walter Tichy

Editor's Introduction

Crunching numbers was the prime task of early computers. Wilhelm Schickard's machine of 1623 helped calculate astronomical tables; Charles Babbage's difference engine, built by Per Georg Scheutz in 1843, worked out tables of logarithms; the Atanasoff-Berry digital computer of 1942 solved systems of linear equations; and the ENIAC of 1946 computed artillery firing tables.

The common element of these early computers is they all used integer arithmetic. To compute fractions, you had to place an imaginary decimal (or binary) point at an appropriate, fixed position in the integers. (Hence the term fixed-point arithmetic.) For instance, to calculate with tenths of pennies, arithmetic must be done in multiples of 10^{-3} dollars. Combining tiny and large quantities is problematic for fixed-point arithmetic, since there are not enough digits.

Floating-point numbers overcome this limitation. They implement the "scientific notation," where a mantissa and an exponent represent a number. For example, 3.0×10^{8} m/s is the approximate speed of light and would be written as 3.0E8. The exponent indicates the position of the decimal point, and it can float left or right as needed. The prime advantage of floating-point is its vast range. A 32-bit floating-point number has a range of approximately 10^{-45} to 10^{+38}. A binary integer would require more than 260 bits to represent this range in its entirety. A 64-bit floating point has the uber-astronomical range of 10^{632}.

A precursor of floating point arithmetic is embodied in the slide rule, invented by William Oughtread in 1620. The slide rule multiplies and divides, but handles only mantissas. The user keeps track of the exponents. Computing machines with floating-point numbers (including exponents) were suggested by the Spanish inventor Leonardo Torres y Quevedo in 1914; they were first implemented in Konrad Zuse's computers, from the Z1 (1938) to Z4 (1945).

Despite their range, floats have a serious drawback: They are inaccurate. Anybody can try this out on a pocket calculator: Punch in ⅓ and you get 0.333333. You wonder, of course, how close an approximation this is. Now multiply by 3. Most likely you will see 0.999999 and not 1.0. If the result is 1.0, subtract 1.0 from it, which will probably give you something like −1E−10. This is a perfectly simple example—why can't computers get this right?

John Gustafson, one of the foremost experts in scientific computing, has proposed a new number format that provides more accurate answers than standard floats, yet saves space and energy. The new format might well revolutionize the way we do numerical calculations.

Walter Tichy
Associate Editor

The End of (Numeric) Error

An Interview with John L. Gustafson

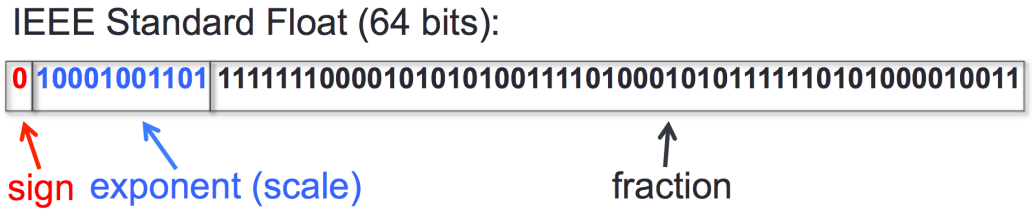
by Walter Tichy

Walter Tichy: You recently published a book with a new format for floating-point numbers. The format prevents rounding errors, overflow, and underflow; produces more accurate results than the current standard; saves storage space and energy; and is faster, to boot! This is hard to believe, but before we get into technical details, is this a topic for the hard-core numerical analyst, or is it of interest to any programmer or even any computer user?

John Gustafson: I'm glad you asked, because it is very much not specific to numerical analysts. It is for anyone who uses computers to perform calculations, whether it is on a spreadsheet, a pocket calculator, a data center doing big data analytics, or a workstation doing design optimization. We are so used to tolerating rounding error in all the computing we do that we don't stop to ask if there is a better alternative.

WT: Please summarize IEEE 754, the current floating point standard.

JG: Floating-point format for computers is an idea about 100 years old, actually. The idea of all floating-point numbers is to store a scaling factor (the exponent) separate from the significant digits (the fraction, also sometimes called the "mantissa" or the "significand"). Also, a bit to indicate if the number is negative. The IEEE 754 Standard settled on the sizes of the bit fields for the exponent and fraction and where they would be located in the bit string. It also said how to represent exception values like infinity and Not-a-Number, or NaN for short. NaN is for calculations like zero divided by zero, or the square root of negative one, where the answer cannot be expressed as a real number. Here's what a double-precision IEEE float looks like, for example:



That particular bit pattern stores the physical constant known as Avogadro's number, about 6.022×10^{23}. If we say it has four decimals of accuracy, then we really can only justify using about 13 bits in the binary fraction of the float. But you can see we have 53 bits of fraction (including the sign bit), and the last 40 bits are spurious. Also, it only takes eight bits to express the scale factor of 10^{23}, not eleven bits as shown above. That's why IEEE floats waste so much memory and energy; they are "one size fits all," so they have to be big enough for the worst case instead of being the right size for each calculation.

The exponent can be positive or negative to represent very large or very small numbers, more economically than doing it with integers. Avogadro's number would take 80 bits to store as an integer instead of a 64-bit float.

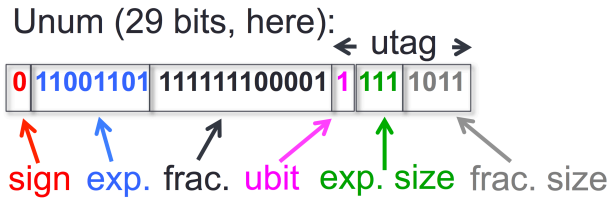
The exponent marks where the decimal point, actually the "binary point" is, and instead of being in one location all the time like it is for money calculations, it can float left and right as needed. That's why it's called floating-point. Notice that you can use floating-point to represent integers, so they are actually a superset of integers, not an alternative.

WT: Your new format, the "unum" number, allows varying sizes for exponent and fraction, plus a special bit that indicates whether the number is exact or inexact. Please explain the new format. Why is it called unum?

JG: The word "unum" is short for "universal number," the same way the word "bit" is short for "binary digit." Some people, like me, pronounce it "you-num" because that's how I pronounce the u in "universal". But a German speaker might prefer "oo-num" since that reflects how they would pronounce the word for universal. It's a lower-case word; there is no need to capitalize any of its letters!

A unum has three additional fields that make the number self-descriptive: The "ubit" that marks whether a number is exact or in between exact values, the size of the exponent in bits, and the size of the fraction in bits. So not only does the binary point float, the number of significant digits also floats. There is something called "significance arithmetic" that does a

crude version of what the unum representation does, but unum math overcomes some serious drawbacks of both significance arithmetic and interval arithmetic. Here is an example of a unum for that same Avogadro's number:



The added three fields are called the "utag" and the user can select how many bits are in that. The total number of bits varies, which is really no harder to deal with than all the other variable-size values we regularly handle in a computer, like strings and programs and font widths. That unum still expresses 6.022×10^{23}, but it does it with the appropriate number of exponent and fraction bits. So even though a unum has three additional bit fields, it can wind up requiring fewer total bits than the "one size fits all" IEEE float format.

WT: Let's look at the first problem of IEEE floating point numbers, digit cancellation. Suppose we subtract two numbers that are close to each other, say 1.000 007 and 1.000 006, which results (after normalization) in $1.000\ 000\times10^{-6}$. The problem here are the zeros that seem to be exact. If we add to this 1.0×10^{-12}, we get $1.000\ 001\times10^{-6}$. This result looks like it is accurate to six places after the decimal point, when in fact everything after the decimal point is noise. What would unums do differently here?

JG: Good question, because this is where unums are different from significance arithmetic. If the inputs are marked as inexact, then the subtraction will result in one significant digit and the fraction bits will automatically shrink to take up less space. But if the inputs are exact, which is possible, then the answer will be expressed as the binary value closest to $1.000\ 001\times10^{-6}$ since there really are that many significant figures.

WT: Next problem, overflow. In 1996, the Ariane 5 rocket crashed during launch because of an overflow. The overflow was detected by hardware but, unfortunately, not caught by software. There was no exception handler for this case. Suppose we choose a unum format too small to store the numbers occurring during launch, what would happen? We still would need a handler for "too large a number," wouldn't we?

JG: Had the Ariane 5 system been designed with unum arithmetic, the scale factors and precision would have automatically adjusted between the high-precision (64-bit) speed measurement and the low-precision (16-bit) guidance controller, instead of having to blindly assume that each fixed format had sufficient dynamic range. Unums would have prevented the disaster. Unums do not overflow to infinity or underflow to zero like floats do. They increase the number of exponent bits automatically to accommodate the dynamic range needed, and even if they completely run out of available space, an overflow is stored as the range between the largest real and infinity, which means you can still do arithmetic with it.

WT: So in a case like this, where we shoehorn a large unum into a small one, the result is a bit pattern that indicates the number is between the largest one representable by the small unum, and infinity. We can continue calculating with that pattern, but the result will remain "too large a number." Somewhere the guidance controller would have to have a special handler for that. Which was what was missing from the Ariane software in the first place.

Which brings us to the question about the advantage of distinguishing between "too large a number" and infinity. Traditional floats map any number that is too large to infinity. With unums, will programmers have to handle two special cases rather than one?

JG: Programmers do not have to handle it, and that is the point. Right now, an infinite value on a computer could be the result of overflow or it could be the correct answer (like the logarithm of zero is exactly negative infinity). With floats, you cannot tell the difference. More importantly, it is mathematically dishonest. It is easy to find examples where substituting infinity for a too-large-to-represent number gives an answer that is infinitely wrong. The problem with Ariane is programmers specified the dynamic range, and chose poorly. Computers should figure out what dynamic range is needed automatically to avoid that kind of mistake.

The prototype unum environment can notice when a result is too large or too small to express and automatically increase the number of exponent bits. That way, information about the answer is not destroyed the way it is with floats.

WT: Now let's look at the failure of the Patriot surface-to-air missile system. The control software counts tens of seconds since initialization. After the system runs for several days uninterrupted, there comes a time when there are more digits than there is room in the

fraction of the floating point number, so the least significant digit is dropped in favor of an additional leading digit (with a simultaneous adjustment of the exponent). This scaling operation causes a loss of precision (but not an overflow), which is the reason why an attacking Scud missile could not be intercepted. The Scud killed 28 soldiers. How would unums help us prevent loss of precision?

JG: That was probably the worst disaster ever caused by floating point being so tricky to use properly.

This is the "flip side" of the Ariane disaster: programmers selecting the number of significant bits of storage, and choosing poorly when the computer should be doing that automatically. Within limits, unums will promote the precision as needed. The maximum allowed relative error can be maintained as an environment variable, and the computer can detect when too much accuracy has been lost to fire the missile. Then it can either fix the problem by increasing the number of fraction bits or, if that is not possible for some reason, alert the user. I know it sounds too good to be true, but I have tested it on case after case. It may sound like one of these multiple precision libraries, but it really is quite different and far more economical for storage.

WT: How would the programmer tell the unum-arithmetic when too much accuracy has been lost? Even if the programmer is willing to do the required analysis, the problem could be fixed as it always has been fixed: Pick a suitable format with enough digits from the start.

JG: If only it were that easy! Take one of the oldest applications of automatic computers: Find a ballistic trajectory. Imagine that you start out knowing the velocity of a projectile and the angle it fires at and the gravitational pull and all the effects of air resistance, exactly. You compute the path by taking time steps of a chosen size. What is the accuracy of the computer prediction of the path it will take? The classic textbooks say "It is on the order of…" and then a complicated formula based on how big the time step is. But that is not a bound on the error, so you really do not know. The formula says the error will be less if you take smaller time steps, which means a longer time for the computer to run, so you probably take as many time steps as you are willing to wait but then you have more cumulative rounding errors. This is why numerical analysis as it stands today is more like art than science.

There are plenty of examples where all accuracy in input values can be destroyed with a single calculation, too. It does not have to be cumulative. Imagine computing the tangent of an angle very close to pi over two radians. Is the angle value rounded up, or rounded down? If it is

rounded down, the tangent function will produce a huge positive number. If it is rounded up, the result will be a huge negative number. Which is it? Floating point is treacherous, even in the hands of people with plenty of numerical experience and training.

In the prototype environment, you can specify a desired relative accuracy. For example, if you say that the uncertainty in a result relative to its size can be no more than 0.005, then that is like asking for three decimals of accuracy in the result. I do it in software presently, but hardware can do this at full speed by operating in parallel with the arithmetic. If a result is larger than the largest representable value, or smaller in magnitude than the smallest representable value, the exponent size is increased and the calculation done again; if the relative uncertainty becomes too large, the fraction size is increased and the calculation done again. It resembles the checkpoint-restart technique used to increase the reliability of supercomputer jobs that must run for very long durations. Except that with unums, once a programmer learns which parts of an algorithm demanded more fraction or more exponent, it is easy to request the right size in the first place and not have to recalculate in general.

WT: Returning to the problem from the introduction: If we compute ⅓ with unums, what do we get? And then multiply by 3 again? How many bits are needed for an accurate answer?

JG: Let me first explain what a "fused" operation is. All computers do more arithmetic than you actually see, in what I call "the hidden scratchpad." For example, when you multiply double-precision floats that have 53 bits of fraction each, the computer uses a scratchpad that can hold the full 106 bits of the product before it is then rounded to 53 bits. IBM pushed for something they called the "fused multiply add" in which you follow the scratchpad multiply immediately by a scratchpad add, and only then round the result. Which is a great idea, so long as you recognize that it's a different computer operation than a multiply followed by an add and will give different results in general. One of the most troublesome things about using IEEE floats is that there are no rules about how much can happen in the hidden scratchpad before a result is returned as a float, which is why the IEEE Standard does not guarantee consistency across computer systems!

The unum environment has a set of fused operations that can be computed exactly in the scratchpad, and use a limited amount of extra precision. And instead of rounding the result to the nearest float, the results are either exact or expressed as lying in the open interval between nearest-possible exact values, using the ubit. So now we can go back to your example.

If you know you are going to do a divide followed by a multiply, then the best approach is to use the "fused" unum function that computes this quantity in the scratch area before returning it to unum representation:

$$x = \frac{m_1 \times m_2 \times \ldots m_j}{n_1 \times n_2 \times \ldots n_k}$$

Think of the *m* and *n* values as IEEE floats. The numerator can be done as an exact calculation, and so can the denominator. The last step is to do the one divide operation and see if there is a remainder. If there is, the result will be inexact and set the ubit. If it is exact, then great! No loss of information. So when you ask for 1 divided by 3 multiplied by 3, the result will be identically 1, no matter how low a precision the unum environment is.

You could put fused operations like this into a floating-point standard, but you would still wind up in trouble because of rounding, and inaccurate results would be returned as if they were exact answers. People have been doing tricks to improve accuracy for a long time, but they always miss the key point: You have to admit what the inaccuracy is at some point. Every other method out there just substitutes a representable number for the correct one, and the instant you do that, no matter how precise you are or how clever your representation is, you have parted ways with mathematics and can no longer demand error-free calculations from the computer.

But let's suppose you don't notice that it can be done as a fused operation, and you actually do the division first followed by the multiplication. And let's try it with a very low-precision unum environment, to exaggerate the effect. The "{2,3}" environment allows up to $2^2 = 4$ bits of exponent and $2^3 = 8$ bits of fraction, for example. So 1 divided by 3 will produce the following unum interval (I call them ubounds to distinguish the from traditional interval arithmetic that only has closed intervals):

$$(0.3330078125, 0.333984375)$$

That is, the ratio ⅓ is somewhere in the open interval, and not equal to either endpoint. This is the biggest difference between unums and floats; a float would round to one of the endpoints and treat it as the exact answer going forward, which is clearly an error.

So then you multiply by 3, and unum math returns the following open interval:[1]

$$(0.998046875, 1.00390625)$$

This is still mathematically honest, because the correct answer 1 is contained in the open interval. If that range is too approximate for the purpose of the calculation, the computer can *automatically* increase the maximum number of fraction bits to tighten the bound. The user simply expresses how much accuracy is needed in the answer and the computer can make the decisions about what kind of floating-point precision to use. And by the way, unums can express that open interval above using only 35 bits, which is far more economical than previously proposed ways of storing numerical bounds.

WT: Let's take the point of view of a compiler writer. The variability in size of unums causes problems for storage management. Suppose we have an array of unums, all of different sizes. How can we index into such an array? Or consider the local variables of a function. These variables are allocated in the activation record of that function one after the other on the call stack. How can they be located if they shift around due to changes in size? I suppose it will be necessary to select a suitable, fixed-size format and pad the numbers with zeros. Alternatively, one could use pointers into the heap, but that would add 64 bits for the pointer and slow down access. In both cases, the advantage of having to move fewer bits, and thereby saving space, bandwidth, and energy, evaporates. Is there a way around this?

JG: The brief answer is that a unum can be "unpacked" into a fixed-size field that still preserves the energy-saving properties of unum arithmetic, but allows constant-time indexing into an array. If you allocate 64 bits to each unum, like you do now for double-precision floats, you can represent unums ranging from 13 to 59 bits with each subfield right-justified within a fixed field within those 64 bits. Those unums can represent numbers ranging over 600 orders of magnitude with ten decimals of accuracy, and you build arrays with them and index into them exactly the same way you would with integers and floats like you do now. The main place you save space, bandwidth, and energy is when you pack the array up and write it to main memory or mass storage. Think of it as lossless compression.

If your algorithm does not require much random access but mainly processes numbers as lists, then you can keep the unums in packed form without using any pointers to them. This is a classic computer science problem, like managing sectors on a disk that have fixed size even

[1] That's not the same as multiplying each endpoint by three, because that would require more than eight bits of fraction for the resulting endpoints.

though the data they contain has variable length. I envision the work of managing unum storage being done by the processor, either with the usual indexing of unpacked unums, or as pointers to objects in the heap.

WT: Interval arithmetic is a method that provides intervals as answers to calculations, not just single values. This idea is attractive, because both inputs and calculations are not exact. Given, for instance, an input value such as 6.5 ± 0.05, we would get an interval as an answer for all values in the input range. However, this idea does not work well with traditional floating-point numbers, because after a few arithmetic operations, the intervals just get too large and inaccurate. Can unums rescue interval arithmetic?

JG: They not only solve this problem, they also do it in a way that requires no expertise. Instead of letting the interval bounds get bigger and bigger until they become meaningless, a unum calculation always starts with values that are either exact, or accurate to the Unit in the Last Place (also called an ULP). If any calculation increases the width of the bound, it splits into multiple unums that tile the region perfectly. The next step of the calculation again begins with values that are either exact or accurate to one ULP. That way, instead of the bounds growing exponentially like they do with interval arithmetic, they tend to grow linearly as error is accumulated. And linear increase in error is quite acceptable if you start with, say, seven decimals of accuracy and end up with five decimals after a hundred calculations. The computer is always dealing with intervals that have been automatically split into unums, and it creates work that is easy to do in parallel so that it does not have to run slower until you run out of processor cores.

It works in multiple dimensions as well. The solution to many computing problems is a shape in n-dimensional space. It could be any shape, but interval arithmetic always describes the shape by putting a box around the whole thing. Unums create an array of “uboxes” that are exact or one ULP wide in each dimension that describe the actual shape as perfectly as possible within the working accuracy. That has the benefit that when a solution is very sensitive to changes in the inputs, you can easily see it in the answer set.

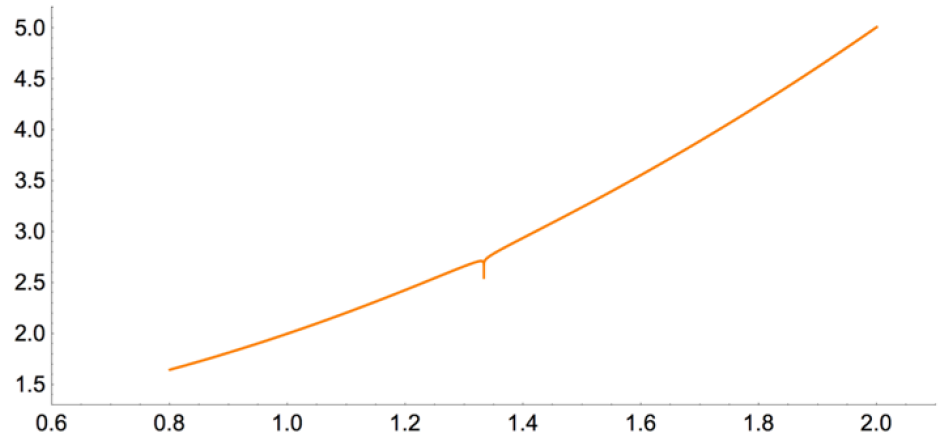
I know many people think unums will suffer the same rapid expansion of bounds that typically happens with interval arithmetic, but I have done quite a few experiments where interval arithmetic fails and unums succeed, like n-body problems. Of course the whole computing community will have to try them on many applications to test this assertion.

WT: Please give us an example of what you mean with tiling and "uboxes."

JG: There is an example that William Kahan calls "The Smooth Surprise," a simple optimization problem that floats cannot get right no matter how much precision they use. Suppose you want to know the minimum value of this function, for *x* between 0.8 and 2:

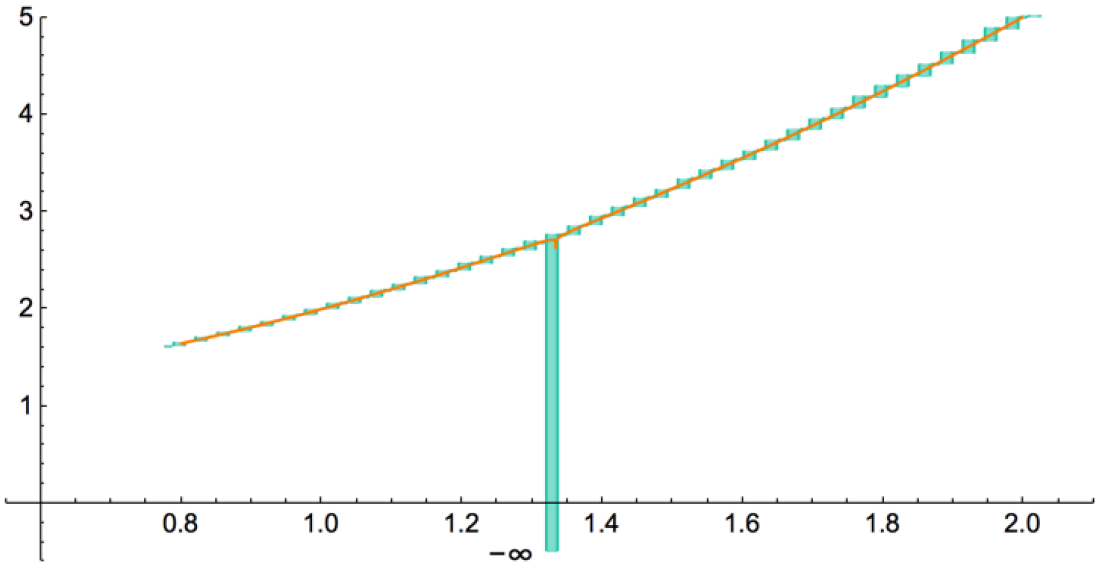
$$\frac{1}{80}\log(|3(1-x)+1|)+x^2+1$$

Here is a plot of the function, using half a million sample points with double precision floats:



It looks like the minimum value is over on the left, where *x* is 0.8. There is that strange little dimple in the function around 1.33, but even if you home in on it using floating-point that is spaced one ULP apart, it still looks like nothing to worry about. But it is something to worry about, because the function actually has a singularity. It plummets to negative infinity when *x* is four-thirds. Since there is no floating-point number equal to four-thirds, it completely skips over the singularity. That's sampling error, the counterpart to rounding error.

Unums, in contrast, do not sample. They can take into account an entire range of real numbers without omission. They "tile" the real number line, alternating between exact values and the open interval between exact values. A "ubox" is an n-dimensional tile. So if I just use a handful of low-precision unums (one-dimensional uboxes), here is how the function evaluates:



So the one-dimensional set of ubox values found the singularity without any skill required at all, and did it with almost a million times less work than floats! Just think what this means for global optimization problems. Interval arithmetic also has some ability to expose "smooth surprises" like this, but cannot distinguish between exact and inexact interval endpoints the way unums can. Unums can *prove* the existence or absence of extrema, not just poke around in the real number line with floating-point samples.

I really think that once people try this approach out, they will not want to go back to using just floats. Unums are to floats what floats are to integers. They are the next step in the evolution of computer arithmetic.

WT: Are unums going to appear in programming languages, and if so, as libraries or as language extensions?

JG: I was just at the Supercomputing conference in Austin, TX, and I was amazed at the groundswell of interest and software efforts surrounding unum math. The Python port is already done, and several groups are working on a C port. Because the Julia language is particularly good at adding new data types without sacrificing speed, several people jumped on making a port to Julia, including the MIT founders of that language. I learned of an effort to put unum math into Java and into D, the proposed successor to C++. Depending on the language, unums can be a new data type with overloaded operators, or handled with library calls. And of

course, Mathematica is a programming language and the prototype environment in Mathematica is free for the downloading from the CRC Press web site. I'd really like to see a version for Matlab.

The C port is going to be as "close to the metal" as we can make it, so that it guides us in building hardware for unums. There is at least one FPGA project underway, and REX Computing expects to build native unum arithmetic into its next-generation processor. It is all happening much faster than I expected. I thought it would take 10 years before there was wide support for unums, but there are all these projects underway just a few months after the idea was introduced in March 2015.

WT: Where can we find out more about unums?

JG: From my book, of course, *The End of Error: Unum Computing*. It's written for anyone who does numerical computing. The math is high school level, at most. There's even a chapter that shows how calculus is not needed for a lot of simulation-type computing, and it can even get in the way of correct computation. The writing style of the book is light even though the math behind it is solid. Lots of color illustrations everywhere. It's a tough subject to make fun to read, but I did everything I could! Several reviewers called it a "page-turner," which makes me smile. A book on computer arithmetic can be a page-turner?

Suggested Readings

John L. Gustafson. *The End of Error: Unum Computing*. CRC Press, 2015.

Rich Brueckner. Slidecast: John Gustafson Explains Energy Efficient Unum Computing. insideHPC. Blog. March 2, 2015.

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: 10.1145/2913029

