DOI: 10.14529/jsfi160203

# A Radical Approach to Computation with Real Numbers

***John L. Gustafson***[1]

If we are willing to give up compatibility with IEEE 754 floats and design a number format with goals appropriate for 2016, we can achieve several goals simultaneously: Extremely high energy efficiency and information-per-bit, no penalty for decimal operations instead of binary, rigorous bounds on answers without the overly pessimistic bounds produced by interval methods, and unprecedented high speed up to some precision. This approach extends the ideas of unum arithmetic introduced two years ago by breaking completely from the IEEE float-type format, resulting in fixed bit size values, fixed execution time, no exception values or "gradual underflow" issues, no wasted bit patterns, and no redundant representations (like "negative zero"). As an example of the power of this format, a difficult 12-dimensional nonlinear robotic kinematics problem that has defied solvers to date is quickly solvable with absolute bounds. Also unlike interval methods, it becomes possible to operate on arbitrary disconnected subsets of the real number line with the same speed as operating on a simple bound.

*Keywords: Floating point, Unum computing, Computer arithmetic, Energy efficiency, Valid arithmetic.*

## 1. A Quick Overview of "Type 1 unums"

The ***unum*** (***u***niversal ***num***ber) arithmetic system was presented publicly in 2013, and a text describing the approach was published in 2015 [2]. As originally defined, a unum is a superset of IEEE 754 floating-point format [6] that tracks whether a number is an exact float or lies in the open interval one Unit of Least Precision (ULP) wide, between two exact floats. While the meaning of the sign, exponent, and fraction bit fields take their definition from the IEEE 754 standard, the bit lengths of the exponent and fraction are allowed to vary, from as small as a single bit up to some maximum that is set in the environment. The formats are contrasted in fig. 1.

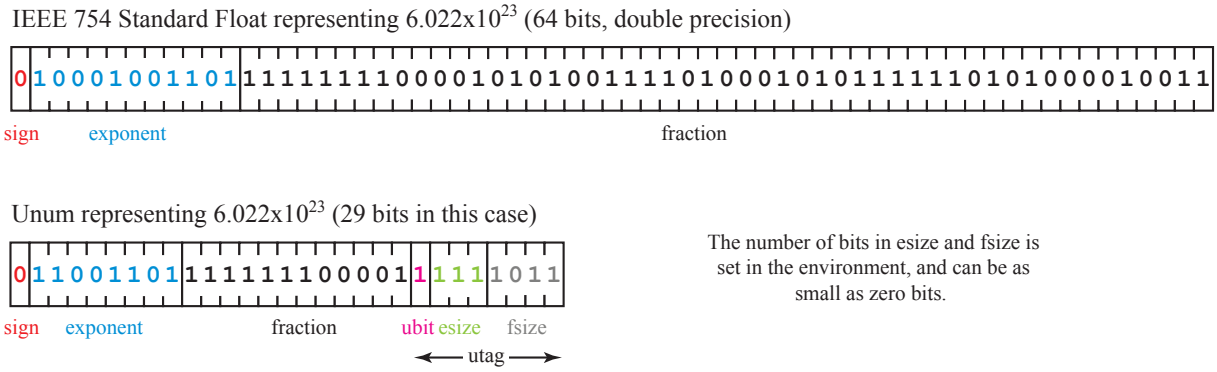



**Figure 1.** Comparison of IEEE 754 64-bit float and a Type 1 unum representing the same value

The inclusion of the "uncertainty bit" (ubit) at the end of the fraction eliminates rounding, overflow, and underflow by instead tracking when a result lands between representable floats. Instead of underflow, the ubit marks it as lying in the open interval between zero and the smallest nonzero number. Instead of overflow, the ubit marks it as lying in the open interval between the maximum finite float value and infinity.

Flexible dynamic range and precision eliminates the pressure for a programmer to choose a "one size fits all" data type. Programmers typically choose double precision (64-bit) as over-

[1] A*STAR Computational Resources Center and National University of Singapore, Singapore (joint appointment)

insurance against numerical problems; however, this typically wastes storage and bandwidth by about a factor of two or more since the maximum precision is only needed for some fraction of the calculations. Furthermore, high precision is no guarantee against catastrophic errors [5]. As fig. 1 shows, adding self-descriptive bits (the "utag") to the float format can actually *save* total bit count, much the same way that having an exponent field in a float saves bits compared to an integer or fixed-point representation.

The unum definition in [2] also fixes some problems with the IEEE 754 Standard, such as negative zero, the wasting of trillions of unique bit patterns for representing "Not-a-Number" (NaN), and the failure to guarantee bit-identical results from different computer systems. However, the original "Type 1 unums" have drawbacks, particularly for hardware implementation:

- They either use variable storage size or must be unpacked into a fixed storage size that includes some unused bits, much like managing variable file sizes in mass storage.
- The utag adds a level of indirection; it must be read first to reference the other fields.
- The logic involves more conditional branches than floats.
- Some values can be expressed in more than one way, and some bit patterns are not used.

## 2. The Ideal Format

The current IEEE 754 format evolved from historical formats and years of committee discussions selecting compromises between speed and correctness. What if we abandon ties to the past and ask what would be an *ideal* representation of real number values? Here are some goals, some of which are similar to mathematical goals defined in [3]

- All arithmetic operations would be equally fast.
- There would be no performance penalty for using decimal representation.
- It would be easy to build using current processor technology.
- There would be no exceptions like subnormal numbers, NaN, or "negative zero."
- Accuracy could be managed automatically by the computer.
- No real numbers would be overlooked; there might be limited accuracy, but no omissions.
- The system would be mathematically sound, with no rounding errors.

The last three goals are achieved by Type 1 unums, so we want to preserve those advantages. A value like $\pi$ can be represented honestly as $3.14\cdots$ where the "$\cdots$" indicates that the value lies in the open interval between two exact numbers, (3.14, 3.15). The error of non-unum number systems is in restricting representations to exact values *sampled* from the real number line; when an algorithm exposes the mathematical omission, the strategy to date has been to demand more exact points to fill in the gaps, a futile chase that ignores inherent properties of the real number line. Once a system embraces the idea that finite-state representations can represent exact values **and** the open ranges between exact values, there is hope of creating a bijection: that is, a mapping from bit strings to *sets* of real numbers that is one-to-one and onto. Furthermore, it may happen that a surprisingly low-precision bit string has more expressive power than a high-precision format that can represent only a set of exact values.

# 3. A Mapping to the Projective Real Line, and its Power Set

## 3.1. The projective reals resemble modern signed (two's complement) integers

One representation of the real line bends the two infinite extremes into a circle. The distinction between positive and negative infinity is lost, replaced by a single "point at infinity." We can label it $\pm\infty$ for now, and think of it as the *reciprocal of zero*. A mapping of the projective reals to two-bit integer strings is shown in fig. 2.

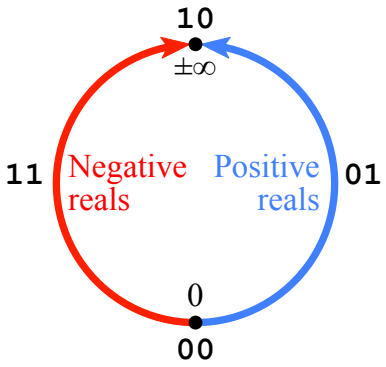



**Figure 2.** Projective real number line and two-bit two's complement integers

As signed integers, the four strings around the circle **00, 01, 10** and **11** correspond to two's complement integers $0, 1, -2, -1$. They "wrap" from positive to negative at exactly the point where the projective reals wrap from positive to negative. We call these two-bit strings *unum* values because they represent either exact quantities or the open interval between exact quantities. For our purposes, we choose to treat $\pm\infty$ as if it were an exact quantity.

## 3.2. SORNs and the elimination of "indeterminate forms"

Imagine another bit string now, one that expresses the absence (**0**) or presence (**1**) of each of the sets shown in fig. 2. That is, the power set of the four subsets of the projective real numbers, $\pm\infty$, $(-\infty, 0)$, 0 and $(0, \infty)$. Such a bit string allows us to operate on subsets of the reals using bit-level arithmetic, as opposed to symbol manipulation. For example, the bit vector **0011** means the absence of $\pm\infty$ and $(-\infty, 0)$ and the presence of 0 and $(0, \infty)$ in the set, which we could write in more conventional notation as the interval $[0, \infty)$. Call this bit string a *SORN*, for Sets Of Real Numbers. To make them easier to distinguish from the binary numbers that use positional notation to represent integers, we can use the following shapes: ● ○ ▮ ▯. A circle represents an open interval and a thin rectangle represents an exact value. They are filled if present and hollow if absent. If we have access to color display, then it further helps read SORNs if negative values are red and positive values are blue; the ▮ or ▯ shapes for values 0 and $\pm\infty$ remain black since they have no sign. Here is a table of the 16 possible SORN values for the unum set shown in fig. 2:

**Table 1.** Four-bit SORN values

| SORN notation | English description | Math notation |
|---|---|---|
| ▯○▯○ | The empty set | $\{\}$ or $\varnothing$ |
| ▯○▯● | All positive reals | $(0, \infty)$ |
| ▯○▮○ | Zero | $0$ |
| ▯○▮● | All nonnegative reals | $[0, \infty)$ |
| ▯●▯○ | All negative reals | $(-\infty, 0)$ |
| ▯●▯● | All nonzero reals | $(-\infty, 0) \cup (0, \infty)$ |
| ▯●▮○ | All nonpositive reals | $(-\infty, 0]$ |
| ▯●▮● | All reals | $(-\infty, \infty)$ or $\mathbb{R}$ |
| ▮○▯○ | The point at infinity | $\pm\infty$ |
| ▮○▯● | The extended positive reals | $(0, \infty]$ |
| ▮○▮○ | The unsigned values | $0 \cup \pm\infty$ |
| ▮○▮● | The extended nonnegative reals | $[0, \infty]$ |
| ▮●▯○ | The extended negative reals | $[-\infty, 0)$ |
| ▮●▯● | All nonzero extended reals | $[-\infty, 0) \cup (0, \infty]$ |
| ▮●▮○ | The extended nonpositive reals | $[-\infty, 0]$ |
| ▮●▮● | All extended reals | $[-\infty, \infty]$ or $\mathbb{R}^+$ |

Because we have the reals on a circle instead of a line, it is possible to notate the nonzero extended reals, for example, as the interval "$(0, 0)$" instead of having to write $[-\infty, 0) \cup (0, \infty]$. Usually, an interval is written as a closed interval $[x, y]$ where $x \leq y$, or one with open endpoints $(x, y]$, $[x, y)$, $(x, y)$ where $x < y$. Any contiguous set can be written as a simple interval, where numbers increase from the left endpoint until they reach the right endpoint, even if it means passing through $\pm\infty$. This is similar to the idea of "outer intervals" [4]. Also, for the reader who is already missing the values $-\infty$ and $+\infty$ that IEEE floats provide, notice that **we still have them**, in the form of unbounded intervals flanking $\pm\infty$. For example, we could take the logarithm of $(0, \infty)$ and obtain $(-\infty, \infty)$. The square of that result would be $[0, \infty)$, and so on. If a SORN shows the presence of $\pm\infty$ and one of the two unbounded unums $(1, \infty)$ or $(-\infty, -1)$, we treat it as a closed endpoint, "$\infty]$" on the right, or "$[-\infty$" on the left.

The arithmetic tables for $+ \; - \; \times \; \div$ on these SORN values look at first like a hellish collection of **all the things you are never supposed to do**: zero divided by zero, infinity minus infinity, and other so-called *indeterminate forms*. They are called indeterminate because they do not produce *single numbers*. There is usually some wringing of hands about dividing nonzero numbers by zero as well; is the answer $+\infty$ or $-\infty$? We have no such difficulties here. If we take the limit of, say, $x - y$ as $x \to \infty$ and $y \to \infty$, we find it can be any value, and there is a SORN for that: ▮●▮●. Similarly, divide any positive number by $x$ as $x \to 0$ and the result is $\infty$ or $-\infty$ depending on whether the limit is from the left or the right. However, we have just the thing for that situation: $\pm\infty$, represented by ▮○▯○. There is no reason to wish for a NaN representation.

Looking ahead a bit, we can imagine taking the square root of the negative reals, ▯●▯○. With conventional floats we would certainly have to throw up our hands and return a NaN. With SORNs, the answer is the empty set, ▯○▯○. We can even take care of indeterminate forms like $1^\infty$, using limits of $x^y$ as $x \to 1$ (from above or below) and $x \to \infty$. It is simply the nonnegative extended reals, ▮○▮●. There is nothing wrong with the result of a calculation being

a set, including the empty set. We do not need to admit defeat by declaring something NaN, and in fact can continue calculating. Even if it appears that all information has been lost and the answer can be anything, that is, ▮●▮●, if the next operation were to square the SORN it would result in the nonnegative extended reals ▮○▮●, which has some information about the answer.

### 3.3. Fast calculation of SORNs with bitwise OR operations

Imagine that we have filled out the addition table for the four unum values $\pm\infty$, $(-\infty, 0)$, 0 and $(0, \infty)$. We can express each unum as a SORN with just one of the four shapes filled in. As tab. 2 shows, addition sometimes produces a SORN with more than one shape filled in.

**Table 2.** The addition table for two-bit unum inputs and SORN outputs

| + | ▮○▯○ | ▯●▯○ | ▯○▮○ | ▯○▯● |
|---|---|---|---|---|
| ▮○▯○ | ▮●▮● | ▮○▯○ | ▮○▯○ | ▮○▯○ |
| ▯●▯○ | ▮○▯○ | ▯●▯○ | ▯●▯○ | ▯●▮● |
| ▯○▮○ | ▮○▯○ | ▯●▯○ | ▯○▮○ | ▯○▯● |
| ▯○▯● | ▮○▯○ | ▯●▮● | ▯○▯● | ▯○▯● |

The highlighted parts of the table indicate *information loss*; three of the entries "blur" in that they do not produce a single unum output from two unum inputs. In some systems, this would entail recording a variable amount of data for table entries. With SORNs, all entries are the same number of bits, facilitating table look-up in a computer.

Furthermore, they lend themselves to very fast and simple evaluation of SORN operations with logic OR gates and a bit of parallelism, as shown in fig. 3.

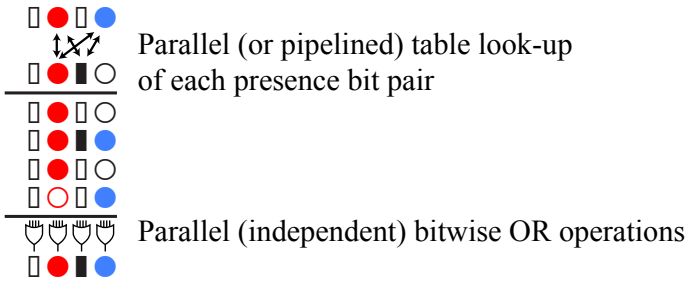



**Figure 3.** Fast SORN operation using parallel table look-up followed by parallel bitwise OR

The logic gate delay time at current 14 nm technology is about 10 picoseconds. A table lookup involves three gate delays and the parallel OR operation adds another 10 picoseconds, depending on the fan-in design rules. This suggests that scalar operations on the real number line at this ultra-low accuracy level can be done at around 25 GHz. Fig. 4 shows how simple a ROM is for table look-up; the black dots are wired connections requiring no transistors. If the table were stored in DRAM, it would require 3.2 times as many transistors (not counting refresh circuitry) with one transistor per bit. SRAM requires six transistors per bit, which would take 14 times as many transistors.

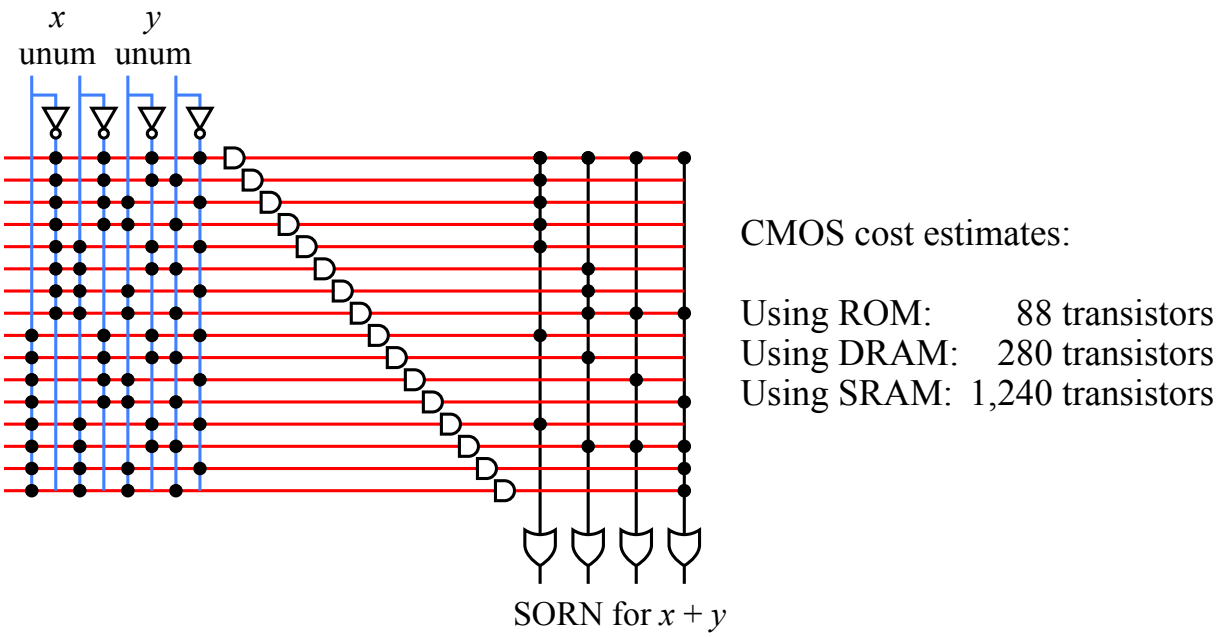



**Figure 4.** ROM circuit example for the table look-up of the preceding SORN addition

The next step is to start ramping up the accuracy of the unum lattice and the SORNs that go with that lattice.

### 3.4. The start of a useful number system: a kinematics application

Append another bit to the unum so that we can represent +1 and −1, and the open intervals surrounding those exact numbers. The annotated circle of real values and some examples of SORN representations are shown in fig. 5. When assigned to binary strings, the first bit resembles a sign bit like that of IEEE floats, though we ignore it for values 0 and ±∞. The *last* bit serves as the "uncertainty bit" or *ubit*, exactly as it did with the IEEE-compatible Type 1 unums definition. The ubit is **0** for exact unums, **1** for open ranges between exact numbers ("inexact" unums, for short). Hence, we color-code those two bits the same way as the original unums [2], with the sign bit in red and the ubit in magenta.

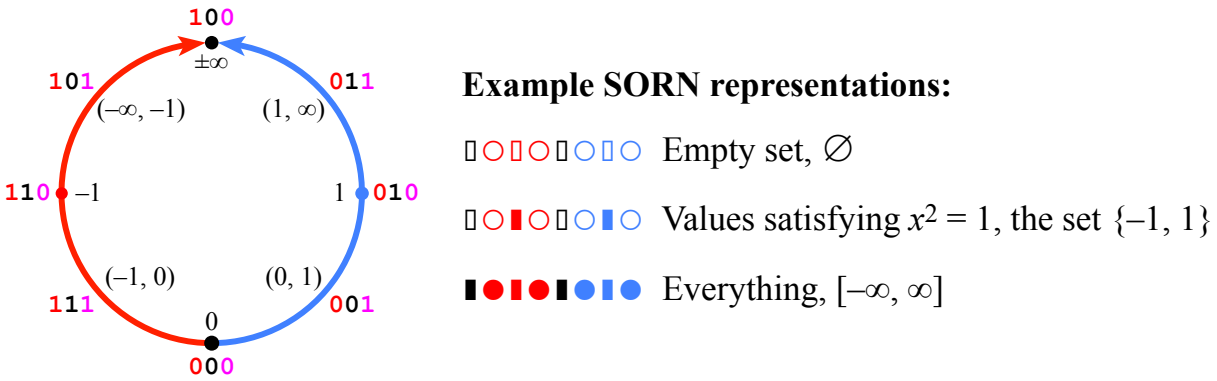
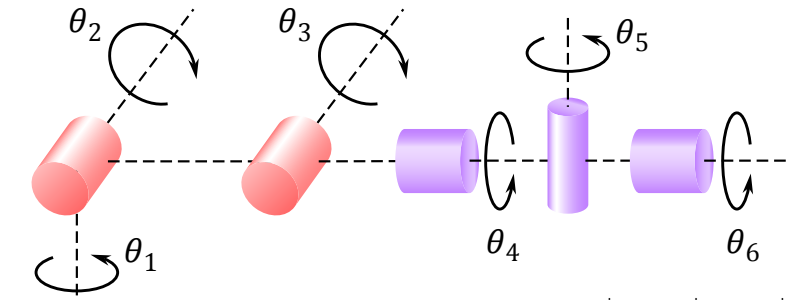



**Figure 5.** Three-bit unum representation with −1 and +1, and examples of SORNs

Such low precision can be surprisingly useful, since it is often helpful in the early stages of solving a problem to know at least a little bit about where to look for a solution. Are the solutions of bounded magnitude? Are they known to be positive? For example, a classic problem

in robotics is to solve the inverse kinematics of an elbow manipulator [1]. Such a problem and the twelve nonlinear equations in twelve unknowns that it gives rise to are shown in fig. 6.




$$s_2c_5s_6 - s_3c_5s_6 - s_4c_5s_6 + c_2c_6 + c_3c_6 + c_4c_6 = 0.4077$$
$$c_1c_2s_5 + c_1c_3s_5 - c_1c_4s_5 + s_1c_5 = 1.9115$$
$$s_2s_5 + s_3s_5 + s_4s_5 = 1.9791$$
$$c_1c_2 + c_1c_3 + c_1c_4 + c_1c_2 + c_1c_3 + c_1c_2 = 4.0616$$
$$s_1c_2 + s_1c_3 + s_1c_4 + s_1c_2 + s_1c_3 + s_1c_2 = 1.7172$$
$$s_2 + s_3 + s_4 + s_2 + s_3 + s_2 = 3.9701$$
$$c_i^2 + s_i^2 = 1 \; (1 \le i \le 6)$$

**Figure 6.** Inverse kinematics problem: a constrained elbow manipulator

The classic approach to such a set of equations is to guess a starting value for the twelve unknowns and iterate toward a solution, if a solution even exists. There might be multiple solutions, but such an approach will find at most one of them. With our ultra-low precision SORNs in fig. 5, it becomes feasible to test *the entire 12-dimensional space* for regions where solutions can or cannot exist. Unums of magnitude greater than 1 are ruled out by the $c_i^2 + s_i^2 = 1$ equations; the $c$ and $s$ variables are cosines and sines of the six angles, though we do not need that knowledge for the unum approach to converge quickly. If we split all twelve dimensions into two possible open unums $(-1, 0)$ or $(0, 1)$, there are $2^{12} = 4096$ regions of the space of solutions, which can be examined in parallel in a few nanoseconds using the SORN set shown in fig. 5. The result is the exclusion of 4000 of the spaces as *infeasible* solution regions, leaving only 96 possibilities for further examination. While this sort of approach has been used with interval arithmetic in the past, those computing environments involve 128 bits per variable (two double-precision endpoints), and very slow, energy-intensive arithmetic compared to the fast table lookup of 3-bit unums to populate 8-bit SORNs. With dedicated hardware for the low precision approach, the unum approach should reduce energy use and execution time by over a hundredfold, based on the number of exercised gates and the logic delay times.

If it is important to minimize the total number of constraint function evaluations, another approach is to split each of the twelve dimensions at a time, moving to the next dimension only when a split does not create a new excludable region. After six million low-precision calculations (requiring milliseconds to evaluate), the set of $c_i$-$s_i$ pairs form arcs specific enough that a robotic control system would be able to make a decision, as shown in fig. 7.

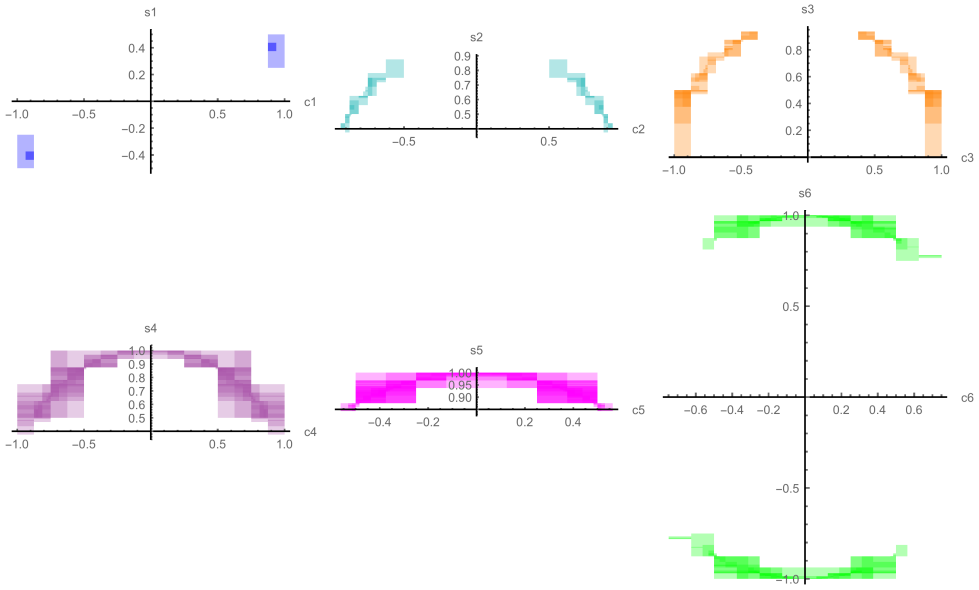



**Figure 7.** Proved feasible set for robotics inverse kinematics problem

# 4. Selecting the u-lattice and populating the number system

## 4.1. The u-lattice

Define the *u-lattice* as an ordered set of 1 followed by exact positive real numbers in the interval $(1, \infty)$. This set is then made closed under both negation and reciprocation, by including negatives and reciprocals of the u-lattice in the set of exact unums. For a fixed-size unum of length $m$ bits, the u-lattice should have $2^{m-3}$ values, that is, 1/8 of $2^m$, since $2^m$ is the total number of bit patterns possible with $m$ bits. The reason is that combining the u-lattice with its reciprocal almost doubles the number of exact values (1 is already in the set), and combining with the negative of that set doubles it again; finally, representing the open intervals between exact values doubles the number of exact values a third time. Including the values 0 and $\pm\infty$ brings the total count up to exactly $2^m$, so no bit patterns are wasted and no bit patterns are redundant.

## 4.2. Example for 4-bit unums

Four bits for each unum means $2^4 = 16$ bit patterns, and $2^{4-3} = 2$ values in the u-lattice. A simple example is to select $\{1,\ 2\}$ as the u-lattice. There is nothing special about the number 2; we could have used $\{1,\ 10\}$, or even $\{1,\ \pi\}$ as the exact numbers on which to base the number system. Some people are surprised that $\pi$ can be represented as an exact *number*, but of course it can, which is one reason for the term "universal number."

The set $\{1,\ 2\}$ united with its reciprocals $\{1/2,\ 1\}$ becomes $\{1/2,\ 1,\ 2\}$. Uniting with negatives of the set and the set $\{0,\ \pm\infty\}$ gives the eight possible exact unums $\{\pm\infty,\ -2,\ -1,\ -1/2,\ 0,\ 1/2,\ 1,\ 2\}$. The last step is to include the open intervals between each of these, such as $(-1/2,\ 0)$, so there are also eight possible inexact unums, as shown in fig. 8.

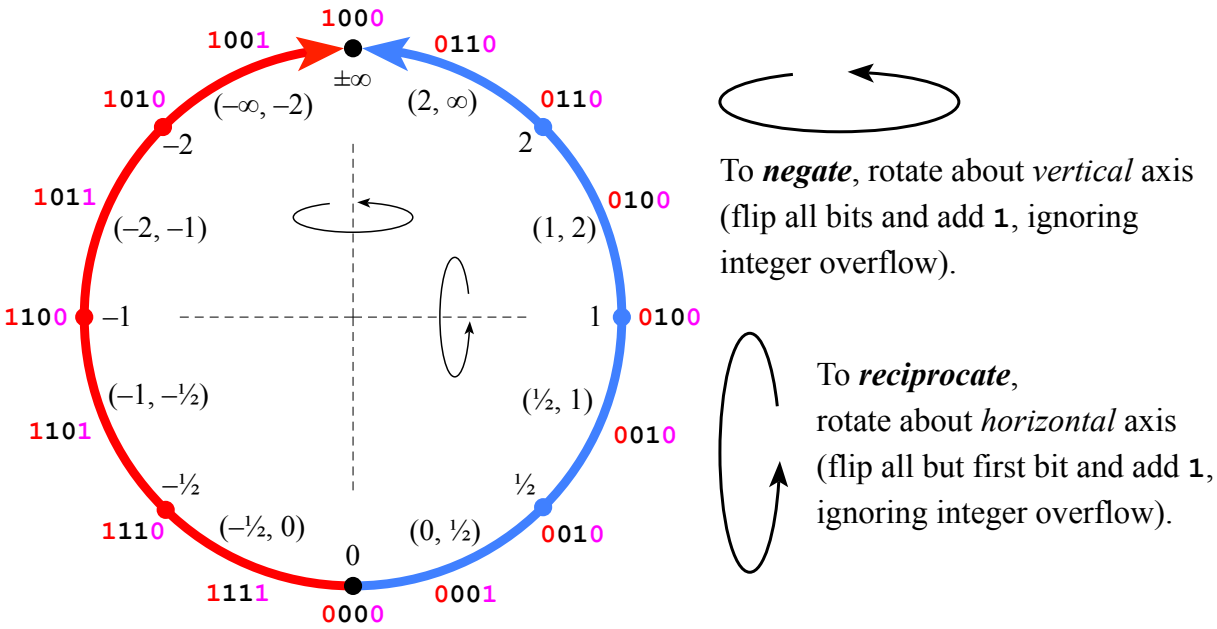



**Figure 8.** Four-bit unums and geometrical analogies for negating and reciprocating

This system places the arithmetic operations + − × ÷ on *equal footing*. With floats, it is dangerous to replace the operation $x \div y$ with $x \times (1/y)$ because there are two rounding operations; float math seldom calculates the reciprocal 1/y without rounding error, so for example, $3 \div 3$ may evaluate to 1 exactly, but $3 \times (1/3)$ will result in something like 0.9999···. On the other hand, it has long been safe to treat $x - y$ as identical to $x + (-y)$. Addition and subtraction share hardware. With the system described here, multiplication and division can share hardware as well.

It may be time to revive an old idea: "/" as a unary prefix operator. Just as unary "–" can be put before $x$ to mean $0 - x$, unary "/" can be put before $x$ to mean $1/x$. Pronounce it "over," so $/x$ would be pronounced "over $x$." Just as $-(-x) = x, //x = x$. Compiler writers and language designers certainly should be up to the task of parsing the unary "/" operator as they have with unary "–".

### 4.3. Freedom from division-by-zero hazards

What is $f(x) = 1/(1/x + 1/2)$ for $x = 0$? Most number systems balk at this and throw an exception because the $1/x$ step divides by zero. With the projective real approach used here, $1/0 = \pm\infty$; adding $1/2$ to $\pm\infty$ leaves $\pm\infty$ unchanged, and then the final reciprocal operation turns $\pm\infty$ back into zero. The expression can be rewritten as $f(x) = 2x/(2 + x)$, revealing that the singularity at $x = 0$ is perfectly removable, but that requires someone to do the algebra.

Suppose the input were a SORN, such as one representing the half-open interval $-1 < x \leq 2$. With the 4-bit unums defined in the previous section, the computation can be performed without any loss of information. The SORN sets remain contiguous sets through every operation, and provide the correct result, $-1/2 < f(x) \leq 1$, as shown in fig. 9. The figure saves space by using the unary "/" notation, and we will use that notation from now on.

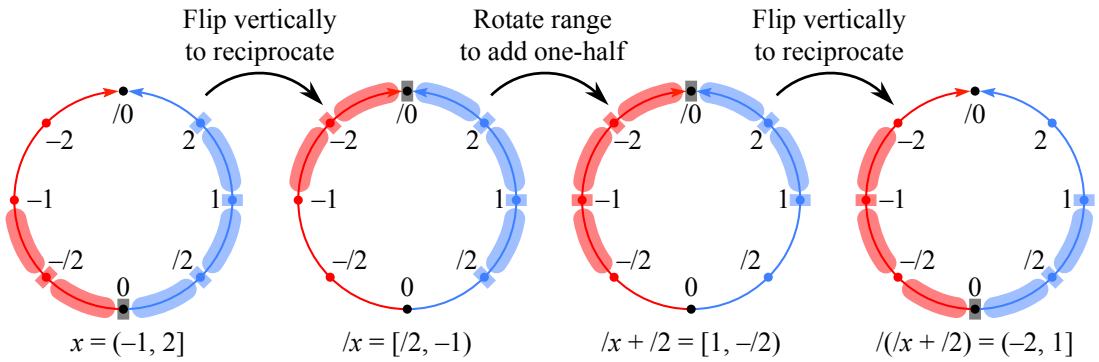



**Figure 9.** Tight bounding of $1/(1/x + 1/2)$ despite intermediate division by zero

Notice that if we had we rewritten the expression as $2x/(2+x)$ and attempted traditional interval arithmetic with $x = [-1, 2]$ (since we have no way to express open endpoints, we have to use a closed one at the $-1$ endpoint), $f(x)$ would evaluate to the loose bound $[-2, 4]$ because of the *dependency problem*: $x$ now appears twice in the expression, and the calculation ignores the dependency between them. The interval arithmetic result thus unnecessarily includes the value $-2$ and the range $(1,\ 4]$, and is twice as wide a bound as the tight bound shown in fig. 9.

### 4.4. Strategies for an 8-bit unum set

Since IEEE 754 specifies *decimal* floats only for 32-, 64-, and 128-bit sizes, it will be interesting to see if we can create a useful decimal system with as few as 8 bits. The choice of u-lattice depends on the application. If a large dynamic range is important, we could use this u-lattice: $\{1, 2, 5, 10, 20, 50, \ldots, 10^9, 2\times10^9\}$. The reciprocals of that set are also expressible with a single decimal times a power of 10, and that u-lattice provides over 18 orders of magnitude (from $5\times10^{-10}$ to $2\times10^9$) of dynamic range, but less than one decimal digit of accuracy.

If we prefer to have every counting number from 1 to 10 represented, then we could start with the set $\{0.1, 0.2, \ldots, 0.9, 1, 2, \ldots, 9\}$ as "must have" values. This is what IEEE decimal floats do, and one of the drawbacks is "wobbling accuracy" when the slope changes suddenly. Deviation from a true exponential curve means that the relative error is too low in some places and too high in others, indicating information-inefficient use of bit patterns to represent real numbers. The left graph in fig. 10 shows this effect, where the slope suddenly increases.

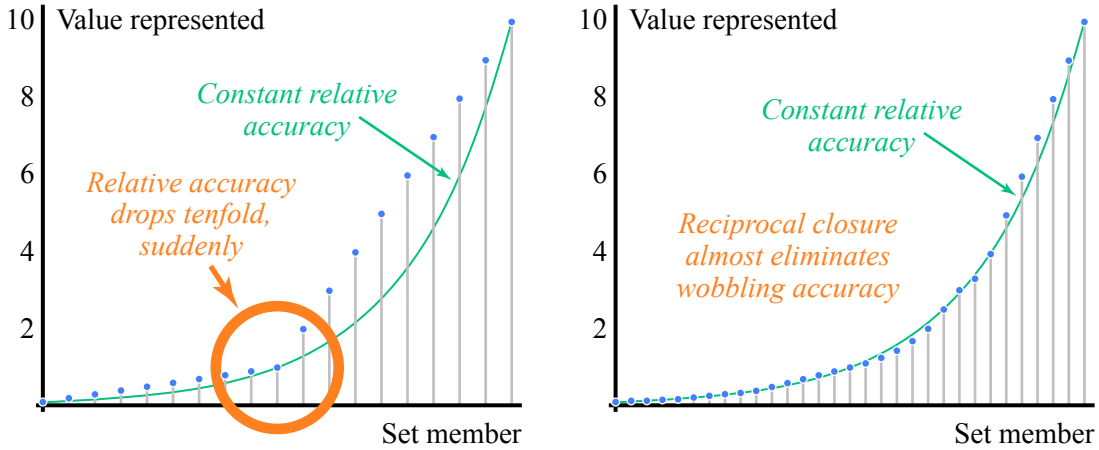



**Figure 10.** IEEE-style decimals versus decimal unums with reciprocal closure

Creating closure under reciprocation has the added benefit of dramatically reducing the wobbling accuracy in the selection of exact values. The width of uncertainty, divided by the magnitude of the value, is almost flat. The set of numbers may look peculiar since we are probably not yet used to reading the unary "/" operator, but from smallest to largest it sorts as follows:

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

Wherever a reciprocal has an equivalent traditional finite decimal, we show the finite decimal; instead of writing "/8" for one-eighth, we write the familiar "0.125" for now, even though it takes five characters to express instead of two. This u-lattice has 15 exact values per decade of magnitude. That means an 8-bit unum lattice could range from 0.009 to /0.009, slightly more than four orders of magnitude. Compared to the {1, 2, 5, 10, ...} u-lattice, we have less dynamic range but a solid decimal of accuracy while preserving closure under reciprocation.

There is an intermediate between these u-lattices that merits attention, even if it has no obvious way to scale to more decimals of accuracy: Powers of 2 ranging from −4 to +4, scaled to fit into the 1 to 10 range. That is, start with the following set:

$$\{0.0625, 0.125, 0.25, 0.5, 1, 2, 4, 8, 16\},$$

then scale each entry to fit into the 1 to 10 range:

$$\{1, 1.25, 1.6, 2, 2.5, 4, 5, 6.25, 8, 10\}.$$

This is a wonderful basis for a u-lattice, for those concerned with closure under multiplication and division. It looks like a nearly exponential spacing of points from 1 to 10, except for the relatively large gap between 2.5 and 4. Plug that gap with $\sqrt{10}$ and an amazing thing happens: the u-lattice produces a very close match to ten exponentially spaced points, as shown in tab. 3. Engineers will recognize the bottom row as the definition of *decibel* ratios, from 0 dB to 10 dB.

**Table 3.** A new way to count from 1 to 10

| Exact unum | 1 | 1.25 | 1.6 | 2 | 2.5 | $\sqrt{10}$ | 4 | 5 | 6.25 | 8 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $10^{k/10}$, k = 0 to 10 | 1 | 1.25··· | 1.58··· | 1.99··· | 2.51··· | $\sqrt{10}$ | 3.98··· | 5.01··· | 6.30··· | 7.94··· | 10 |

By crafting the u-lattice this way, we obtain even less wobbling accuracy than binary floats, for which relative accuracy wobbles by a factor of 2. Some may balk at $\sqrt{10}$ being treated as a counting number, if for no other reason than it being difficult to type, but if written as "r10" then it should present no problem for computer input as a character string. The set of positive exact unum values gives over six orders of magnitude:

$$\{0.0008, 0.001, 0.00125, 0.002, 0.0025, 0.001r10, 0.004, \ldots, 100r10, 400, 500, 625, 800, 1250\}.$$

It is worth looking at the multiplication and addition tables for a decade's worth of values, to see how often a result is expressible as another exact unum (cyan) versus lying between two exact unums (red). The table leaves out input arguments 1 and 10 as trivial, and we write "···" after

an exact unum as the shorthand for "in the open interval beyond this exact unum," indicating information loss.

**Table 4.** Multiplication table within a decade

| × | 1.25 | 1.6 | 2 | 2.5 | r10 | 4 | 5 | 6.25 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| 1.25 | 1.25··· | 2 | 2.5 | 2.5··· | r10··· | 5 | 6.25 | 6.25··· | 10 |
| 1.6 | | 2.5··· | r10··· | 4 | 5··· | 6.25··· | 8 | 10 | 12.5··· |
| 2 | | | 4 | 5 | 6.25··· | 8 | 10 | 12.5 | 16 |
| 2.5 | | | | 6.25 | 6.25··· | 10 | 12.5 | 12.5··· | 20 |
| r10 | | | | | 10 | 12.5··· | 12.5··· | 16··· | 25··· |
| 4 | | | | | | 16 | 20 | 25 | 10r10··· |
| 5 | | | | | | | 25 | 25··· | 40 |
| 6.25 | | | | | | | | 10r10··· | 50 |
| 8 | | | | | | | | | 62.5··· |

A remarkable 25 of the 45 entries shown are exact. A desirable property of a u-lattice is that there not be too much "blurring" of results from the basic operations. The product of an exact unum and an inexact one should not require more than three contiguous unums: an inexact, an exact, and an inexact. The product of two inexact unums should not spread out to more than five contiguous unums. Having a nearly exponential spacing of u-lattice values helps achieve this goal.

There is obvious symmetry about the diagonal from upper left to lower right, since multiplication is commutative. There is a less-obvious symmetry about the orange cells shown from the top right to the center of the table; numbers reflected over those cells become the reciprocal (times 10) of the value, since we have closure under reciprocation. We could omit those, just as the table omits negative values. While a naïve table of all possible 128 by 128 multiplications (16 384 entries) could be maintained if memory is cheap and logic expensive, a little bit of logic can go a long way to reduce the number of necessary table entries to as few as 45.

## 4.5. Other strategies worth considering

*Rational* arithmetic has some advocates, where the numbers are of the form $\pm p/q$ where $p$ and $q$ are positive integers up to some limit, together with some accommodation for zero and infinity cases. (Without ubit support, rational arithmetic suffers the same rounding problems as floats since most operation results will not be exactly expressible.) The u-lattice provides excellent scaffolding for the creation of a rational number system. For example, we could find all $p/q$ such that $1 \le q \le p \le 10$ to populate one decade of a u-lattice:

$$1, 10/9, 9/8, 8/7, 7/6, 6/5, 5/4, 9/7, 4/3, 7/5, 10/7, 3/2, 8/5, 5/3,$$
$$7/4, 9/5, 2, 9/4, 7/3, 5/2, 8/3, 3, 10/3, 7/2, 4, 9/2, 5, 6, 7, 8, 9, 10$$

This could be simplified by requiring that $p + q \le 11$:

$$1, 6/5, 5/4, 4/3, 3/2, 5/3, 7/4, 2, 7/3, 5/2, 3, 7/2, 4, 9/2, 5, 6, 7, 8, 9, 10$$

In either case, it is easy to obtain closure under reciprocation, and a fair number of multiplications and divisions land on exact values. The percentage of the time an exact result is

produced, however, is not as high as it is for the "decibel" number set, and because there are twice as many exact values per decade, the dynamic range will be half as large. However, if an application demands many ratios of small integers, this approach may have its uses.

Finally, we have been showing *flat accuracy* systems where the spacing of the logarithm of values is approximately constant from smallest to largest positive lattice point. Another approach is to use *tapered accuracy* where there is more accuracy near 1 but less accuracy for very large and very small values. For example, with 8-bit unums we could still populate the decade between 1 and 10 as shown in the right-hand graph of fig. 10:

$$1, /0.9, 1.25, /0.7, /0.6, 2, 2.5, 3, /0.3, 4, 5, 6, 7, 8, 9,$$

but then increase the spacing for the next decade:

$$10, 12.5, 20, 25, 40, 50, 80,$$

still wider spacing in the next decade,

$$100, 200, 500,$$

and finally allow the exponent to grow so rapidly that it becomes very unlikely for a product to land in the (maxreal, /0) open interval:

$$1000, 10\,000, 10^6, 10^{10}, 10^{20}, 10^{50}, 10^{100}.$$

These 32 exact values, united with their reciprocals, negatives, 0 and /0, and the open intervals between those values, form a byte-sized unum that is decimal-based with slightly more than one digit of accuracy near unity and a dynamic range of 200 orders of magnitude. The main drawback to tapered accuracy is that it is harder to compress the look-up tables by exploiting patterns that repeat for every decade.

## 5. Why unums are *not* like interval arithmetic

Perhaps the most succinct form of the interval arithmetic "dependency problem" is this: Assign $x$ to an interval, and compute $x - x$. Obviously the correct answer is zero, but that's not what interval arithmetic gives you. Suppose $x$ is the interval [2, 4]. If we assign $x \leftarrow x - x$ repeatedly you will get the following interval ranges after a few iterations:

$x = [-2, 2]$
$x = [-4, 4]$
$x = [-8, 8]$
$x = [-16, 16]$
$x = [-32, 32]$

The bounds grow exponentially. The interval method takes $(\max x) - (\min x)$ for the upper bound and $(\min x) - (\max x)$ for the lower bound. The uncertainty feeds on itself. In contrast, a SORN with the 8-bit unums shown in tab. 3 produces the following sequence:

$x = (-1, 1)$ $\{(-1, -0.8), -0.8, (-0.8, -0.625), \ldots, (0.625, 0.8), 0.8, (0.8, 1)\}$
$x = (-0.2, 0.2)$ $\{(-0.2, -0.16), -0.16, \ldots, 0.16, (0.16, 0.2)\}$
...
$x = (-0.0008, 0.0008)$ $\{(-0.0008, 0), 0, (0, 0.0008)\}$

The sequence actually *decreases* in width and is stable. Similarly, $x \leftarrow x/x$ blows up very rapidly if iterated similarly using interval arithmetic:

$x = [1/2, 2]$
$x = [1/4, 4]$
$x = [1/16, 16]$
$x = [1/256, 256]$
$x = [1/65\,536, 65\,536]$

whereas the SORN set converges to a stable interval containing 1, with some loss of information caused by the limited (single-digit) accuracy. Because arguments to arithmetic operators are never more than one ULP wide, the expansion of the bounds cannot feed on itself. An $n$-body simulation written using with unums shows only linear growth in the bounds of the positions and velocities, but interval arithmetic quickly produces bounds of meaningless large size [2].

# 6. Higher precision, and table look-up issues

## 6.1. 16-bit unums and a comparison with half-precision floats

As we increase the unum size to 16 bits and larger, we start to notice the need for techniques to reduce the size of SORN representation and the size of look-up tables. If we start with a contiguous set of unums and only use $+ \; - \; \times \; \div$ operations, the unums *remain contiguous*. This property means that for an $n$-bit unum, the SORN can be represented with two $n$-bit integers, the first indicating the position of the first **1** bit and the second indicating the length of the string of **1** values. The pair of integers 0, 0 is reserved to represent the empty set, and any other identical pair is reserved to represent a SORN with all presence bits set, that is, $[-/0, /0]$ or $\mathbb{R}^+$.

IEEE half-precision binary floats have slightly more than 3-decimal accuracy, and the normalized numbers range from about $6 \times 10^{-5}$ to $6 \times 10^4$, or nine orders of magnitude dynamic range. Many bit patterns are wasted on redundant NaN representations and negative zero. Can a 16-bit unum do as well, and actually *express three-digit decimals exactly*?

The surprising answer is that 16-bit unums cover *more* than nine orders of magnitude, from $/0.389 \times 10^{-5}$ to $3.89 \times 10^4$, despite the cost of reciprocal closure and the ubit to track inexact results. Sometimes, an answer known accurate to three decimals is preferable to a (64-bit) 15-decimal answer of completely unknown accuracy. It is also possible to represent all the 2-decimal values with 16-bit unums, which allows a dynamic range of more than *93* orders of magnitude.

In general, to store decimals from $1.00\ldots0$ ($k$ digits) to $9.99\ldots9$ ($k$ digits) requires almost $3.6 \times 10^k$ distinct unum values, including the exact reciprocals and the values between exact unums. For example, a decimal unum equivalent to an IEEE 32-bit binary float might have 7 decimal digits of accuracy, and over 59 orders of magnitude dynamic range. If representation of physical constants like Avogadro's number and Planck's constant are important, we could settle for 6 decimals of accuracy and then be able to represent almost 600 orders of magnitude.

### 6.2. Table look-up issues and future directions

Arithmetic tables are rich in symmetries that can be used to reduce their size, at the cost of some conditional tests and some integer divides. For example, do we need both positive and negative entries in the multiplication table, or should we test the sign of each argument, make them positive if necessary, and then set the sign as appropriate? That tiny bit of logic cuts the table size by a factor of four, yet for extremely low-precision unums it may not be worth it! As the unum precision increases, we find ourselves wanting more logic and smaller tables.

Suppose we did a very naïve set of tables for $+ \; - \; \times \; \div$ for 16-bit unums, by fully populating all four tables and ignoring symmetry and repeating patterns. That would require 32 gigabytes. But a small amount of obvious logic easily reduces that to a few megabytes. A more sophisticated approach notices that the smoothly exponential spacing of a flat-accuracy u-lattice means that the unum bit strings can be added or subtracted to perform a close approximation to multiplication or division, and the table need only hold small integer correction factors.

The next research direction is to find a practical set of techniques for fast table look-up and for table size minimization, and determine tradeoffs between speed and table size. While the table sizes for higher precision unums may look daunting, it is possible to put many billions of bits of ROM storage in an integrated circuit without ruining the power budget or the area demands. It may turn out that the new formulation of unums is primarily practical for low-accuracy but high-validity applications, thereby complementing float arithmetic instead of replacing it.

## 7. Summary: Software-Defined Arithmetic

The approach described here could be called *software-defined arithmetic* since it can adjust to the needs of a particular application. In fact, an application could alter its number system from one stage to the next by pointing to alternative look-up tables; perhaps the early stages of a calculation must test a large dynamic range, but once the right range is found, bits in the representation are put to better use improving accuracy. Little is needed in the way of specialized hardware for "Type 2 unums" since all processors are well equipped with the machine instructions look up values in tables. Perhaps a processor could dedicate ROM to a standard set of u-lattice values, but allow the user to define other u-lattices in RAM, at the cost of slightly more energy and execution time. The processor instructions for processing SORNs would not need to change.

The *compiler* could select the u-lattice, especially if the compiler has information (provided by the user or by historical data from the application runs) regarding the maximum dynamic range needed, say, or the amount of accuracy needed. Perhaps the greatest savings in time would be for the compiler to populate look-up tables for *unary* functions needed in any particular program. Imagine a program that repeatedly computes, say, an expression like $\cos\left(x + e^{-3x^2}\right)$. The compiler could discover the common sub-expression, then provide a table that looks up the value in only one clock cycle and is mathematically perfect to the unum accuracy level. (The "Table Maker's Dilemma" disappears since there is no requirement of constant-time evaluation at run time.)

The energy savings and speed of such customized arithmetic might well be one or two orders of magnitude better than IEEE floats, while providing automatic control of accuracy loss in a way that resists the shortcomings of interval arithmetic. Type 2 unums may well provide a shortcut to achieving exascale computing.

## References

1. Yousif Ismaill Al-Mashhadany. Inverse kinematics problem (IKP) of 6-DOF Manipulator by Locally Recurrent Neural Networks (LRNNs). In *Management and Service Science (MASS), 2010 International Conference on*, pages 1–5. IEEE, 2010.

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

3. Timothy Hickey, Qun Ju, and Maarten H. Van Emden. Interval arithmetic: From principles to implementation. *Journal of the ACM (JACM)*, 48(5):1038–1068, 2001.

4. William M. Kahan. A more complete interval arithmetic. *Lecture notes for a summer course at the University of Michigan*, 1968.

5. Jean-Michel Muller, Nicolas Brisebarre, Florent De Dinechin, Claude-Pierre Jeannerod, Vincent Lefevre, Guillaume Melquiond, Nathalie Revol, Damien Stehlé, and Serge Torres. *Handbook of floating-point arithmetic.* Springer Science & Business Media, 2009.

6. Dan Zuras, Mike Cowlishaw, Alex Aiken, Matthew Applegate, David Bailey, Steve Bass, Dileep Bhandarkar, Mahesh Bhat, David Bindel, Sylvie Boldo, et al. IEEE standard for floating-point arithmetic. *IEEE Std 754-2008*, pages 1–70, 2008.