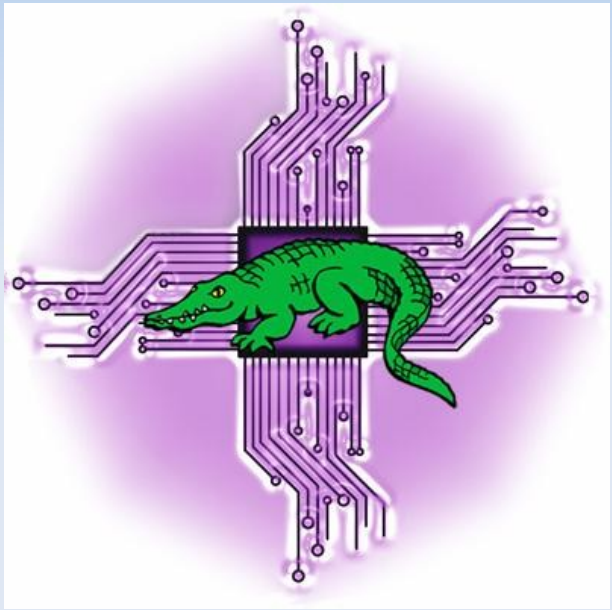
Universal Numbers



Linux Users of Victoria, May 3, 2016

lev@levlafayette.com

About John Gustafson and Unums

Claims to abolish computational error, whilst using fewer bits and less power consumption.

* Unums is an abbreviation of the Universal Number format, a superset of IEEE types (754 and 1788): integers -> floats -> unums.

* Reputation for finding elegant solutions to problems. Most famous for overcoming Amdahl's Law in 1988, which partially resolved the issue of limited performance improvement in multicore systems by increasing problem size. Has worked as Chief Graphics Product Architect at AMD, Director of Intel Labs-SC, CEO of Massively Parallel Technologies, Inc. and CTO at ClearSpeed Technology.

* The problem being addressed: Computers are imprecise (1/3 = 0.333 * 3 = 0.999) with dangerous rounding errors. The IEEE standard give different results on different machines! Calculations use too much power, we don't have enough bandwidth, and we have a lot of processing power.

"[Computers] lie all the time, and at incredibly high speeds... The phrase 'correctly rounded' is an oxymoron, since a rounded number is by definition the substitution of an incorrect number for the correct one."

* Unums are released under a MIT license, a free software license (also used with the Mono development platform class libraries, Ruby on Rails, Node.js, jQuery, and the X Window System). Very close to the 3-clause "modified" BSD license.

What's Wrong With Floating Point?

* Scientific notation was developed Leonardo Torres y Quevado in 1914. It expresses a number as the product of two parts; a mantissa and an exponent (e.g., 3*10^2). It started appearing around (Konrad Zuse's Z3), using base two. Incompatible expressions of binary notation led to the establishment of IEEE 754 in 1985).

* Programmers will adopt one of several IEEE standard levels of precision (e.g., 16 bit (half), 32 bit (single), 64 bit (double), 128 bit (quad), depending on the level of precision they think they need. Too little precision means that they accuracy may be too low. Too high a precision choice means the application will consume unnecessary storage space, bandwidth, energy, and power to run, and of course will also run slower.

* This can have some dramatic results; on June 4, 1996 the Ariane 5 rocket of the European Space Agency exploded just forty seconds after launch. The rocket was the result of a decade of development costing $7 billion; the destroyed rocket and its cargo were valued at $500 million. The cause was a 64 bit floating point number relating to the horizontal velocity of the rocket with respect to the platform was converted to a 16 bit signed integer.

Are We Using Too Much Power?

* Arithmetic on a modern computer uses very little energy; relatively a lot more is spent on moving data. Transistors are cheaper and faster, but the wiring connecting them has remained relatively slow, expensive, and energy inefficient.

Operation	**Approximate energy consumed**
64-bit floating multiply-add	64 picoJoules
Load or store register data	6 picoJoules
Read 64 bits from DRAM	4200 picoJoules

* Moving data around on a chip is relatively low-power because the wires are tiny, but moving data to and from external DRAM means driving wires that are big enough to see - it can take over 60 times as much energy to move a 64-bit float into a processor as it takes to perform a multiply and an add - and that ratio is increasing as transistors improve.

What Unums Promise

* Unums encompasses all IEEE floats, fixed point, and exact integer, but allows for dynamic variation in the precision and range to the optimum number of bits, and also records whether the number is exact or lies within a range instead of rounding the number.

* Unums have more accurate answers than floating point arithmetic, yet use fewer bits, saving memory, bandwidth, energy, and power. Unlike floating point, unums make no rounding errors, and cannot overflow or underflow.

* The trick (in advance), unums gives bounds on answers when it can't give an exact number.

* Main drawback is that it requires more gates to implement in a chip, but gates are something we have in abundance.

Integer Representation

* Binary numbers require need more symbols to represent a number. (e.g., a decimal like 999,999 turns into 11110100001000111111 in binary). It takes more than 3.3 times as many digits (3 from 0-7, 4 for 8-9).

* A fixed-size representation for basic arithmetic is often not expressible in this format; "closure plots" show that there are significant overflow areas in addition, underflow in subtraction, overflow in multiplication, and inexpressible regions in division. (e.g., 11111 + 11111 has overflow, it requires more than 5 bits to represent the result etc)

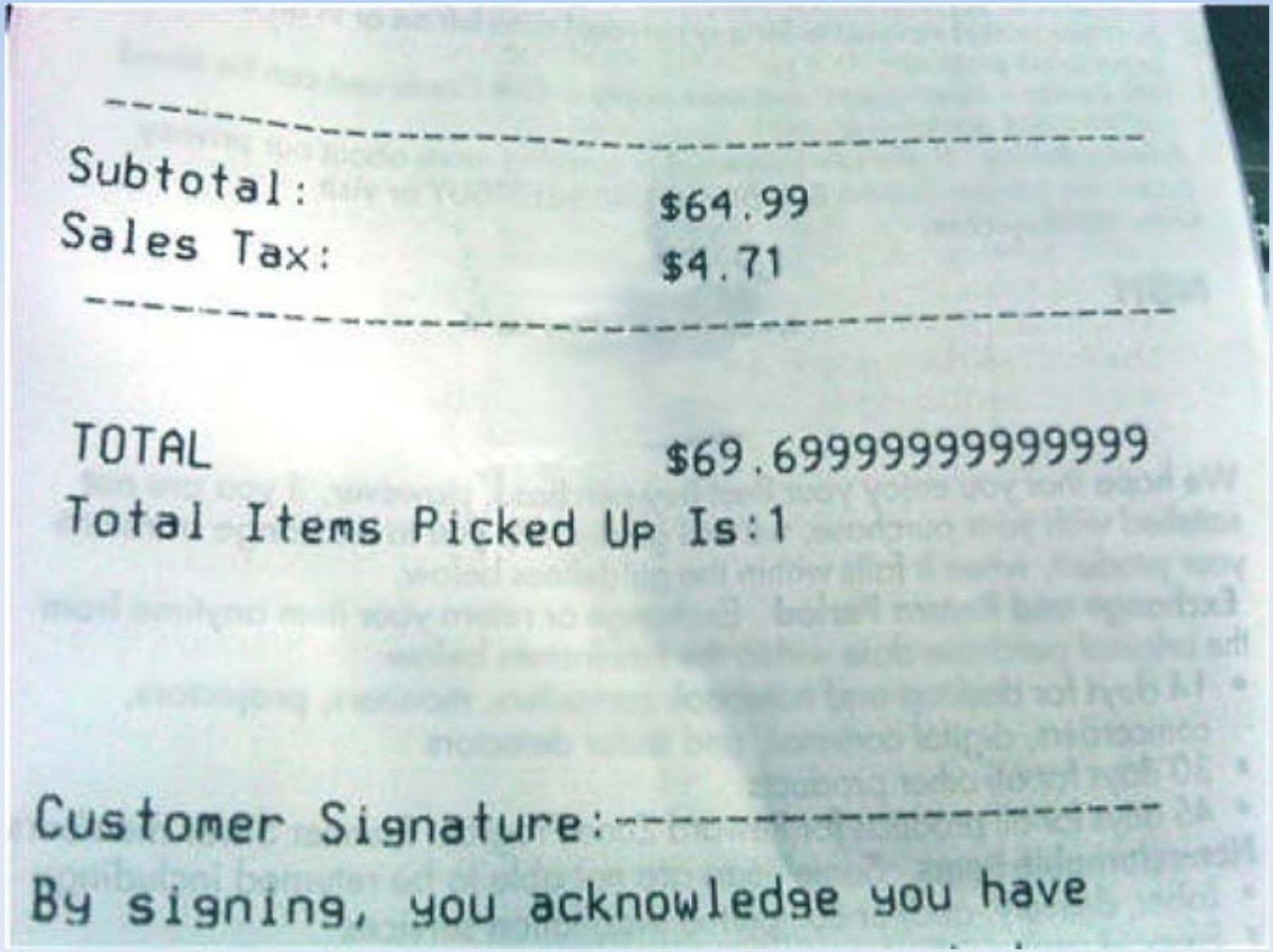
* Negative numbers can be expressed by offset-m or excess-m method (aka "biased representation"), subtracting a fixed number m from the usual meaning of the bits so that the first m numbers are negative.

* Another method is sign-magnitude, which uses a bit to represent positive or negative. Both these methods have improved closure plots.

Fixed Real Representation

* The next step up from signed integers is to represent non-integer fractional values.
One method is to store the numerator and denominator separately, however these have poor closure. Another method is to use a "binary point" in the bit string in the bit string.

* A simple example is fixed point format which works well for financial calculations, where most of the arithmetic is addition and subtraction, but for scientific calculations is a lot of work, because it requires constantly having to watch out for running out of digits to the right or the left of the binary point when multiplying or dividing.



Floating Point Real Representation

* Instead of a fixed location, floating point creates a new field within the bit string that indicates where the binary point. That is, we use bits to represent the sign and magnitude but also to store a variable scale factor as part of each number.

* The IEEE Standard tests if all exponent bits are 1, and uses that case for all of the exceptions. If the fraction bits are all 0 , the value is inf or -inf. If the fraction bits are anything else, then the string represents Not-a-Number. In single precision, there are over sixteen million ways to say that a result is indeterminate, and in double precision, there are over nine quadrillion (9*10^15).

* Floats can express very large and very small numbers, with a wide range of precisions. But they cannot express all real numbers. They can only represent rational numbers where the denominator is some power of 2.

* In some cases they provide they prevent the use of parallelism! There is no associative property for floats. In mathematics, addition and multiplication of real numbers is associative. But the addition and multiplication of floating point numbers is not associative, as rounding errors are introduced when dissimilar-sized values are joined together.

(a + b) + (c + d) (parallel) != ((a + b) + c) + d (serial)

e.g., A floating point representation with a 4-bit mantissa:
(1.0002×20 + 1.0002×20) + 1.0002×24 = 1.0002×21 + 1.0002×24 = 1.0012×24
1.0002×20 + (1.0002×20 + 1.0002×24) = 1.0002×20 + 1.0002×24 = 1.0002×24

Ubits and ULPs

* Append a single bit (the ubit) after the last fraction bit. If it is 0 , the number is exact. If it is 1 then there are more bits after the last one shown, which are not all 0 and they also are not an infinite number of 1 bits (which would make it equal to the next higher number, e.g., 0.999... = 1).

* The last bit in a fraction is called the Unit in the Last Place, or ULP (coined by William Kahan, the same person who drove the original IEEE 754 Standard).

* A number now has a sign, an exponent, a hidden bit for the float, the fraction, and the ubit.
s e e e e e e f f f f f f f u
± exponent h.+ fraction ubit

* With a Ubit, +inf and -inf are "exact" numbers - because it means "a finite number that is too big to express". Likewise "almost (positive or negative) zero" can be represented; the smallest representable non-zero number has a 1 in the last bit of the fraction, and all other bits are 0. Also, the Ubit can be used to represent NaNs, as "beyond infinity".

Example Unums

Consider the small set of numbers for clarity delimited with sign, exponent, fraction, ubit:

0 00 0 0 (sign positive, exponent and fraction are 0, ubit is exact, value is +0)
1 00 0 0 (sign negative, exponent and fraction are 0, ubit is exact, value is -0)

0 00 0 1 (sign positive, exponent and fraction are 0, ubit is inexact, interval value [tiny, +0])
1 00 0 1 (sign negative, exponent and fraction are 0, ubit is inexact, interval value [-0, -tiny])

0 11 1 0 (sign positive, all exponent and fraction bits are 1, ubit is exact, value is +Inf)
1 11 1 0 (sign negative, all exponent and fraction bits are 1, ubit is exact, value is -Inf)

0 11 0 1 (sign positive, all exponents 1, fraction 0, ubit inexact, interval value [4, +Inf])
1 11 0 1 (sign negative, all exponents 1, fraction 0, ubit inexact, interval value [-Inf, -4])

0 11 1 1 (sign positive, all exponent and fraction bits are 1, ubit is inexact, value is quiet NaN)
1 11 1 1 (sign positive, all exponent and fraction bits are 1, ubit is inexact, signalling NaN)

Overflow, Underflow, Wombling Free

* The IEEE Standard says that when a calculation overflows, the value +inf should be used for further calculations. If a number is too small, the Standard says to use 0 instead. *Both substitutions are potentially catastrophic things to do to a calculation*.

* The Standard also says that different flag bits should be set in a processor register to indicate that an overflow, underflow, or rounding (!) occurred. These are usually ignored by programmers and sysadmins - if they can find them, and are often disabled.

* Alerting to inexactness in processor flags is the wrong place. The right place is in the number itself. Putting that one bit in the number eliminates the need for overflow, underflow, and rounding flags.

* There's no need for overflow because unums have +/- posbig and Inf. There's no need for underflow because they have +/- supersmall, 0. At this point when a closure plot is conducted, overflow and underflow cases have been eliminated, and the system is almost closed, where the results of arithmetic are always representable.

* A computation need never erroneously tell you, say, that 10^-100000 is equal to zero but with an underflow error. Instead, the result is marked strictly greater than zero but strictly less than the smallest representable number. Similarly, if you try to compute something like the factorial of one billion, there is no need to incorrectly substitute infinity with overflow.

Interval Horrors

* Interval arithmetic dates back to the 1950s (Raymond Moore is usually credited with its introduction), and is a partially successful attempt to deal with the rounding errors of floating point numbers. A traditional closed interval is all reals between two floating point numbers [a, b] where a <= x <= b. When the result of a calculation is inexact, the a and b are rounded in the direction that makes the interval larger, to insure the interval contains the correct answer.

* However people still tend to use floats because:
- They require significantly more expertise and numerical analysis to use than floating point numbers.
- They produce pessimistic and expanding bounds to the point that it is relatively easy to get a result like [Inf, -Inf].
- They treat numbers as ranges rather than an unknown point within the range; e.g., x - x should be identically zero, but the rules for interval math will instead return [Min(x) – Max(x), Max(x) – Min(x)].
- Intervals take twice as many bits to store a float of the same precision.

* Interval arithmetic will also generate the same sorts of errors as floats; iterating over the square root of a number will eventually display a constant 1.000000; which is wrong. A ubit will show that number but with a value equal to "greater than 1 but by an amount too small to express".

.

Gints and Unums

* A generalized interval (pron. "jint") is like a traditional interval, but the endpoints can be independently closed or open. Where a is less than or equal to b, and a and b can be any representable number, including -inf and +inf. If either endpoint is NaN, no matter whether endpoints are open or closed, the interval is treated as NaN. If the lower and upper bound are the same number, then the interval must be closed: [a, a], sometimes called a degenerate interval, i.e,. an exact value.

* Interval arithmetic has some good ideas about restoring rigour to computer arithmetic. When interval ideas are combined the expressiveness of the ubit, perhaps we can finally create a closed arithmetic system that has a limited number of bits for all operands and all results.

.

Fixed Problems

* Humans, when doing arithmetic, increase and decrease the number of digits as needed. Computer designers prefer fixed sizes to a power of two as a type (e.g., 'bit', 'byte', 'half', 'single', 'double', etc). But there are already many libraries for extended-precision arithmetic that use software to build operations out of lists of fixed-precision numbers. Why not go all the way down to a single bit as the unit building block for numerical storage?

* Currently Half-precision (16 bit) can represents all integers from -/+ 2048. Single precision (32-bit), with an 8-bit exponents, covers around +/- 10^38. Double (64 bits), has some 15 decimals of accuracy and a range from +/- 10^308. Double precision and single precision are the two most common sizes to be built into processor chips as fast, hard-wired data types. A quad precision number (128 bits) has 34 decimals in its fraction and a dynamic range of almost +/- 10^+5000. Quad precision arithmetic is typically executed with software routines that make it about twenty times slower than double precision.

* It's common practise to make every float double-precision to avoid rounding errors. Which means that programs consume more memory, run (slightly) slower, and consume a lot more power.

.

The Whole Unum

* The complete Unum format has two more self-descriptive fields: The exponent size (es) field and the fraction size (fs) field; *esizesize* bits to hold an integer one less than the number of bits in the exponent and *fsizesize* bits to hold an integer one less than the number of bits in the fraction.

sign + exponent + fraction + ubit + exponent size + fraction size

* With fixed-size floats, you have to load or store all bits at once. With unums, you can vary the size of bits accordingto the problem and even change the values as a program is running.

"[U]nums are to floating point numbers what floating point numbers are to integers."

Implementations

John's book contains Mathematica unum prototype

The Unums.jl package provides an implementation of the unum format and ubound arithmetic in Julia.

https://github.com/JuliaComputing/Unums.jl

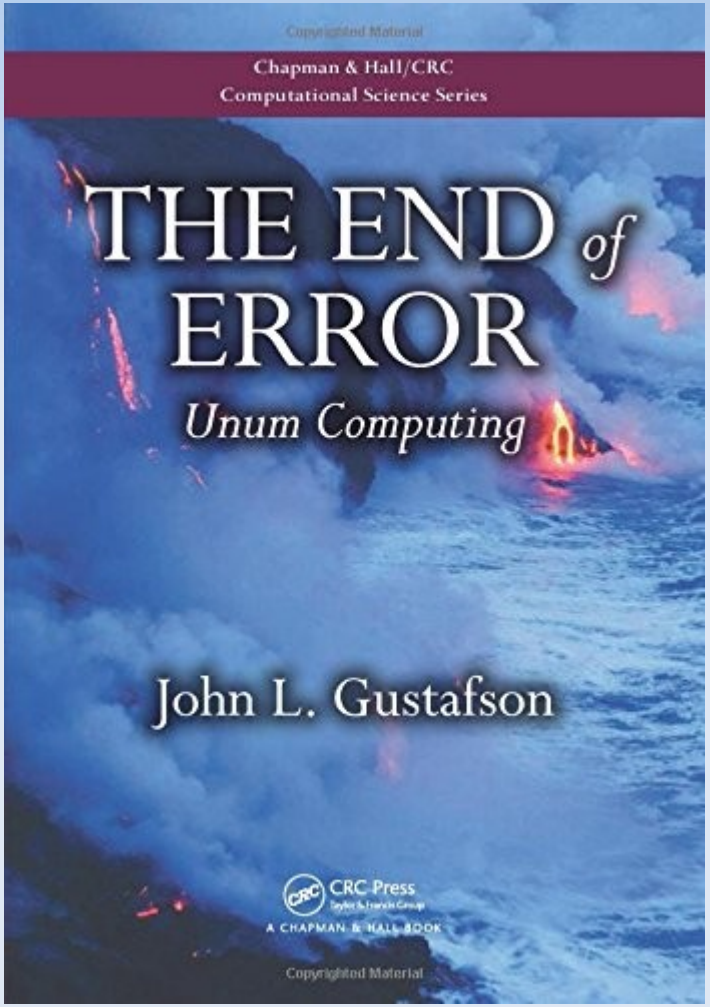
A pure Julia implementation of the unum prototype

https://github.com/REX-Computing/unumjl

A python port of the Mathematica unum prototype

https://github.com/jrmuizel/pyunum

The future of Unums really depends on the first hardware vendor who puts up the capital to fund the chips.



A Typical Kahan Challenge

"Define functions with: $E(0) = 1$, $E(z) = \frac{e^z - 1}{z}$. $Q[x] = \left| x - \sqrt{x^2 + 1} \right| - \frac{1}{x + \sqrt{x^2 + 1}}$. $H(x) = E(Q(x))^2$).

Compute $H(x)$ for x = 15.0, 16.0, 17.0, 9999.0. Repeat with more precision, say using BigDecimal."

- Correct answer: (1, 1, 1, 1).
- IEEE 32-bit: (0, 0, 0, 0) **FAIL**
- IEEE 64-bit: (0, 0, 0, 0) **FAIL**
- Myth: "Getting the same answer with increased precision means the answer is correct."
- IEEE 128-bit: (0, 0, 0, 0) **FAIL**
- Extended precision math packages: (0, 0, 0, 0) **FAIL**
- Interval arithmetic: Um, somewhere between –∞ and ∞. **EPIC FAIL**
- Unums, ***6-bit*** average size: (1, 1, 1, 1) **CORRECT**

I have been unable to find a problem that "breaks" unum math.

Copyright © 2014, 2015 John L. Gustafson

Rump's Royal Pain

Compute $333.75y^6 + x^2(11x^2y^2 - y^6 - 121y^4 - 2) + 5.5y^8 + x/(2y)$
where $x = 77617$, $y = 33096$.

- Using IBM (pre-IEEE Standard) floats, Rump got
  - 1.172603 in 32-bit precision
  - 1.1726039400531 in 64-bit precision
  - 1.17260394005317**8** in 128-bit precision
- Using IEEE double precision: 1.18059×10^{21}
- **Correct answer: –0.827396059946821**36…**!**

  Didn't even get *sign* right

Unums: **Correct answer** to 23 decimals using an average of only **75** bits per number. Not even IEEE 128-bit precision can do that. Precision, range adjust *automatically*.

Copyright © 2014, 2015 John L. Gustafson


THANKS FOR WATCHING
& LISTENING PATIENTLY
makeameme.org