Maths meets Engineering

* The reason why computers lie is fundamentally a conflict between mathematics and engineering. Maths is largely a descriptive language (“quasi-empirical”), and implementing this language is an engineering problem. The implementation of maths into physical systems is imperfect.

* If it is imperfect it is also rational and therefore can also be improved. Multiple interpretations, that is mathematical and engineering, can assist in this process.

* The “survivor bias”, illustrated by a statistical analysis of WWII military aircraft was intuitively solved by Abraham Wald in Columbia University’s Statistical Research Group. A naive interpretation would be to review damage and apply additional armour where damage was identified. Wald pointed out that these planes had returned, and therefore it was those areas that had not been hit that needed more armour.

* An engineer might point out is that the cockpit and the engines had not been hit. Maybe these were somewhat important to the functioning of the plane?

* In the field of computer system management, the sysadmin is a profession somewhat notorious for constant concern with “what could possibly go wrong?”

* Much of this presentation will discuss issues of matters that have gone very wrong indeed, and suggests that there is an area where this engineering concern can meet with the mathematical concerns of efficiency, precision, and accuracy.
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 intractable problems. Most famous for overcoming Amdahl's Law (1967), which partially resolved the issue of limited performance improvement in multicore systems by increasing problem size (Gustafson’s Law, 1988).

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

* The Promise: Unums gets more accurate answers than floating point arithmetic, uses fewer bits in most cases, saves memory, bandwidth, energy, and power. Unlike floating point, unums make no rounding errors, and cannot overflow or underflow.


"[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."

piii, *The End of Error*
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., $3 \times 10^2$). 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.
* February 25, 1991, an American Patriot Missile battery in Dharian, Saudi Arabia, failed to track and intercept an incoming Iraqi Scud missile. The Scud struck an American Army barracks, killing 28 soldiers and injuring around 100 other people. The failure was because the time in 10ths of a second was system's internal clock was multiplied to produce seconds. The calculation, using a 24 bit fixed point register, was truncated at this point. This small error, multiplied over and over again, led to a significant error.

* August 23, 1991, the Sleipner A platform in the North Sea sprang a leak and sank, causing a seismic event registering 3.0 on the Richter scale, and left nothing but a pile of debris at 220m of depth. The failure involved a total economic loss of about $700 million. The collapse was caused by an inaccurate approximation of the tricell void in the finite element approximation which under-estimated the sheer stresses by almost 50%.

* At least Y2K was one that was mostly avoided, albeit at a cost of a trillion or so dollars.

* Exploiting buffer overflows is an extremely common issue in computer security exploitation, from many website and database hacks, the Morris Worm (1988), Code Red (2001), SQL Slammer (2003), Free McBoot (2008), etc and most recently… Notepad.exe (CVE-2019-1162)

* Still a problem: In 2018 in April 3, half of Europe's flights were grounded effecting some 500,000 passengers as the Enhanced Tactical Flow Management System crashed. In May, Fiat Chrysler Automobiles NV recalled more than 5.3 million vehicles in North America over a defect that could prevent drivers from deactivating cruise control. Plus many, many more!

* Numerical errors, errors in arithmetic, buffer overflow and underflow errors, sampling errors are all are to blame here - and "cargo cult" programming; "I just copied my my friend's script" (code re-use is good, but...).
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.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Approximate energy consumed</th>
</tr>
</thead>
<tbody>
<tr>
<td>64-bit floating multiply-add</td>
<td>64 picoJoules</td>
</tr>
<tr>
<td>Load or store register data</td>
<td>6 picoJoules</td>
</tr>
<tr>
<td>Read 64 bits from DRAM</td>
<td>4200 picoJoules</td>
</tr>
</tbody>
</table>

* 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.

* Scale this up to datacentre costs, exascale computing etc and even on personal devices (e.g., mobile device battery life). Heat intensity requires larger size for devices, larger devices increases latency, which reduces performance.

* The computation-memory bottleneck has lead many contemporary computer manufacturers to initiate development in non-volatile memory (NVM), close to the processor. Of course, the PDP-11 had this as well. It defers, rather than resolves, the problem as it is becoming more significant over time.

* And, on a tangent, there is much more hardware parallelism than we seem to know how to use (except for LINPACK to get a machine into the Top500 for marketing purposes).
Integer Representation

* Binary numbers require need more symbols to represent a number. (e.g., a decimal like 999,999 turns into 1111010001000111111 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., if using 5 bit numbers the calculation 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. Sign-magnitude will have two for zero (+0, -0). Major advantage is that turning a positive to a negative simply involves flipping the sign bit.

* Both these methods have improved closure plots. "Overflow" now means the magnitude is too large to express, and "underflow" means that the magnitude is too small to express.
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.

* Combining large and small quantities is difficult for fixed real representation because there is simply not enough digits. Floating point overcomes this limitation.
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 effective 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.

\[(a + b) + (c + d) \neq ((a + b) + c) + d\] (serial)

\[e.g., \text{A floating point representation with a 4-bit mantissa:}\]
\[1.0002\times20 + 1.0002\times20 + 1.0002\times24 = 1.0002\times21 + 1.0002\times24 = 1.0012\times24\]
\[1.0002\times20 + (1.0002\times20 + 1.0002\times24) = 1.0002\times20 + 1.0002\times24 = 1.0002\times24\]

* Problems have been known for a very long time: Goldberg, D., What Every Computer Scientist Should Know About Floating-Point Arithmetic, ACM Computing Surveys, Vol 23, No1, March 1991, pp5-48
Towards a Solution: ULPs and Ubits

* 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 f f f f f f u
± exponent  h. fraction  ubit
```

* For example, it is wrong to suggest that \( \pi = 3.14 \). It is true to say that \( \pi = 3.14.. \), i.e. \( 3.14 < \pi < 3.15 \)

* 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".
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)
* 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 \(a \leq x \leq b\). 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 very imprecise vague results.
  - 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 \([\text{Min}(x) - \text{Max}(x), \text{Max}(x) - \text{Min}(x)]\).
  - Axis-aligned problems (“wrapping problem” and has problems with internal dependences (e.g., \(F(x) = x - x\))
  - 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.

* With unums if you divide 1 by 3 it will return the following open interval \((0.3330078125, 0.333984375)\). A float would round to one of the endpoints and treat it as the exact answer going forward, which is clearly an error and which will result in problems.
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 – all for getting a wrong answer which is consistently wrong with everyone else!
The Whole Unum

* The complete Unum format has two more self-descriptive fields: The exponent size (es) field and the fraction size (fs) field; $esize$ 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.

$$\text{sign} + \text{exponent} + \text{fraction} + \text{ubit} + \text{exponent size (es)} + \text{fraction size (fs)}$$

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

* For the programmer they are alerted to when a number is exact or inexact. The precision is not silent, it is explicit, and as a result, error checking can occur. Explicit protections against issues like the Patriot missile and the Ariane 5 can be included as the programmer is alerted to the potential problems.

"[U]nums are to floating point numbers what floating point numbers are to integers."
The Wrath of Kahan

* Unums and SORNs (Sets of Real Numbers, unum v2) have been subject to criticism, most prominently by Professor Kahan, the primary architect of IEEE standard floats, who claims that (a) the promises of unums are extravagant, and “you can't Always know whether they have betrayed you”.

* Variable size is too expensive; unums will require thousands of extra transistors, which will also come with memory management costs etc. Gustafson points out that there is an entire chapter on fixed-size unum storage and besides, transistors are cheap!

* Argues that like interval arithmetic, SORN and unum computing can overestimate the uncertainty in computed results, where F(x) is too big. The program may fall into an infinite sequence of increasing precision. It can also cause intervals to grow too fast (wrapping problem).

* “Never wrong” != “Always right”. SORN arithmetic lacks algebraic integrity, instead of NaN results (for 0/0, inf - inf, 0.inf, inf/inf etc, SORN produces Ω, the set of extended reals, Ω^2 = [ 0, inf]. Unums uses a quiet NaN and a signalling NaN.

* The Great Debate!
https://www.youtube.com/watch?v=KEAKYDyUua4
The issue of implementing Unums into hardware has always been significant. Main drawback is that it requires more gates to implement in a chip, but gates are something we have in abundance. It is the architecture and production that has been challenging. Version 2 (SORNs) and Version 3 (Posits) takes some of these challenges into account.

“Version 2” of unums introduced notion of SORNs (sets of real numbers). Starts with projective reals using 2 bits, (00 = 0, 01 = + real, 11 = - real, 10 = inf). Infinity expressed as the reciprocal of zero (notation, /0). Sets become numeric quantities, 0 represents absence, 1 represents presence, plus signing. Increase size by a bit and create +1/-1. Easy negation (flip sign), easy reciprocation (reverse non-sign bits, add 1). Add ubit for uncertainty. ULP size variance become sets, and low-precision but rigorous easy to implement as very high-speed look-up tables.

“Version 3” of unums develops a more “hardware friendly” approach with posits, which add an additional “regime” bit after the sign (sign bit, regime, exponent, fraction). The regime bit is variable, continuing until encountering the flip of the sign. The exponent bits (es) determines the dynamic range, but also decreases precision. Like SORNs there is only one version of 0 and one version of Inf. There are no NaN bit representations with posits; the calculation is interrupted, and the interrupt handler reports an error.

Version 4? A valid is a pair of equal-size posits, each ending in a ubit. They are intended for use where applications need the rigour of interval-type bounds (e.g., such as when debugging a numerical algorithm).

Nevertheless, the future of unums really depends on the first hardware vendor who puts up the capital to fund the chips, from software implementations to a FPGA, then custom VLSCI processor.
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
  - $1.172603940053178$ in 128-bit precision
- Using IEEE double precision: $1.18059 \times 10^{21}$
- Correct answer: $-0.82739605994682136\ldots$
  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
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 $-\infty$ and $\infty$. **EPIC FAIL**
- Unums, 6-bit average size: (1, 1, 1, 1) **CORRECT**

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

More recently...

A C++ template library for posits has been developed
https://github.com/stillwater-sc/universal

SoftPosit, provided by the NGA research team based on Berkeley the latest addition to the available software in C
https://gitlab.com/cerlane/SoftPosit
References


Gustafson, J.L., Yonemoto I. *Beating Floating Point at its Own Game: Posit Arithmetic*. 2017, DOI: 10.14529/jsfi170206


Tichy, W., *UNUMS 2.0 : An Interview with John L. Gustafson*, Ubiquity, September 2016
THANKS FOR WATCHING
& LISTENING PATIENTLY