Transcription of "The Great Debate": John Gustafson vs. William Kahan on Unum Arithmetic Held July 12, 2016 Moderated by Jim Demmel

DEMMEL: As you know, John Gustafson recently published a book, called *The End of* Error, Unum, in which he proposes a new floating-point format, called "unums," which is short for "universal number," and, as a replacement for conventional floating point. And as the book's title suggests, he makes some rather strong claims about what unum can do. So for example, that they are easier to use than floating point, *more* accurate, *less* demanding on energy, on memory and bandwidth, or power, and guaranteed to give bitwise identical results, and in most situations, faster.

So, in many places throughout the book, John acknowledges the contributions of William Kahan, of floating point arithmetic of course. And so he uses many of Kahan's examples, throughout the book about how floating point can go wrong, to illustrate the potential advantage of unum. So, for example, one section of Gustafson's book is entitled "The Wrath of Kahan" [*laughter*] and another is entitled "Another Kahan Booby Trap". So, The Great Debate is foreshadowed in Section 18.1, where Gustafson responds at length to a challenge proposed by Professor Kahan to an early draft of the book.

Now, this development has not stopped with this, and John has continued to develop these ideas, and most recently he's proposed something called SORNs, for Sets Of Real Numbers. And I believe, he will explain that new proposal here, and that will also be a subject of the debate.

So the way this is going to be organized is as follows: So each speaker, starting with John, will get thirty minutes, just to explain their position. And you can only ask explanatory questions, *clarifying* questions. Debate questions will come later. And then Professor Kahan will get his thirty minutes. And then we'll have thirty minutes of a free-for-all. But I'm the Moderator [*laughter*], and I will make sure everybody sticks to their thirty minutes and then we'll ask questions and the two gentlemen can speak on it.

So without further ado… John?

GUSTAFSON: (takes the podium) Thank you for this opportunity.

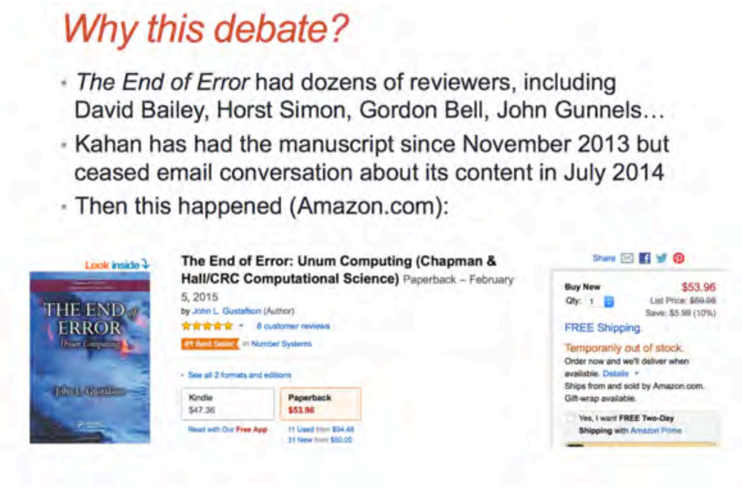
KAHAN: If people move closer, they'll find it easier to read the slides. [*commotion*]

GUSTAFSON: So I also have a take on why this debate is occurring. It's a little bit different from what you just heard.

The End of Error: Unum Computing had dozens of reviewers, which included David Bailey, Horst Simon (colleagues of Dr. Kahan at Berkeley), also Gordon Bell, John Gunnels, who has *won* three Gordon Bell awards, and about half a dozen other PhD numerical analysts. And the book was critiqued not just based on its mathematical content but also on its style, and its approach, since it was aimed at the, let's say, pre-college reader, who did not necessarily have to know calculus in order to understand a lot of what the book is about. [*audience requests adjustments to microphone*]

So I was purposely targeting pre-calculus people, people who had had a good high school education in mathematics. So, Professor Kahan has had the manuscript since November 2013, at least a preliminary version, Part 1, and we had email conversations about it, but all email *cut off* suddenly in July 2014 [*laughter*], going back and forth, but then I posed a counter-question to him, I said, "Can you solve *this* one?" And then the line went dead. I didn't know what I said, but I think it's possible health problems were involved, but I was really missing his contribution because I wanted to make sure I wanted to make sure he was on top of everything I was about to publish and would be able to give me his feedback.

Then this happened.



Why this debate?

- *The End of Error* had dozens of reviewers, including David Bailey, Horst Simon, Gordon Bell, John Gunnels…
- Kahan has had the manuscript since November 2013 but ceased email conversation about its content in July 2014
- Then this happened (Amazon.com):

Look inside

The End of Error: Unum Computing (Chapman & Hall/CRC Computational Science) Paperback – February 5, 2015
by John L. Gustafson (Author)
★★★★★ 8 customer reviews
#1 Best Seller in Number Systems

See all 2 formats and editions

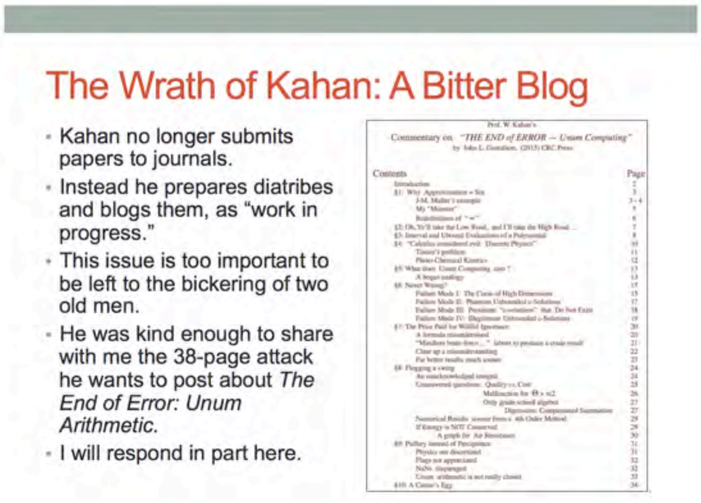
Kindle $47.36 | Paperback $53.96
Read with Our Free App | 11 Used from $54.48 31 New from $50.00

Share

Buy New $53.96
Qty: 1
List Price: $59.95
Save: $5.99 (10%)
FREE Shipping.
Temporarily out of stock.
Order now and we'll deliver when available. Details
Ships from and sold by Amazon.com.
Gift-wrap available.
Yes, I want FREE Two-Day Shipping with Amazon Prime

The book came out in February, and, fairly quickly became the number one best seller in Numerical Systems, admittedly a small category but there's four thousand books on Numerical Systems on Amazon, and it got… eight reviews of five out of five stars, and they kept running out of the thing. They couldn't keep it in stock. And I think this is probably what roused the attention of Dr. Kahan, is because people were actually *reading* this thing, so he should do something… to *caution* people about it [*chuckling sounds*].

So. “The Wrath of Kahan: A Bitter Blog”. As you may know, Kahan does not submit papers to journals, at least not very often. What he does instead is, he writes *blog* articles, and some of them are… very colorful, let’s say. You know, like, “How Java’s Floating Point Hurts Everyone Everywhere.” And things like that. And he prepares these diatribes, and he *loves* them, and usually labels them as “Work in progress.” I don’t know whether he later changes them later or not. But the issue we’re discussing today, the future of number representation on computers, I think is too important to be left to the bickering of two old men [*chuckling sounds*].



The Wrath of Kahan: A Bitter Blog

- Kahan no longer submits papers to journals.
- Instead he prepares diatribes and blogs them, as “work in progress.”
- This issue is too important to be left to the bickering of two old men.
- He was kind enough to share with me the 38-page attack he wants to post about *The End of Error: Unum Arithmetic.*
- I will respond in part here.

Prof. W. Kahan's

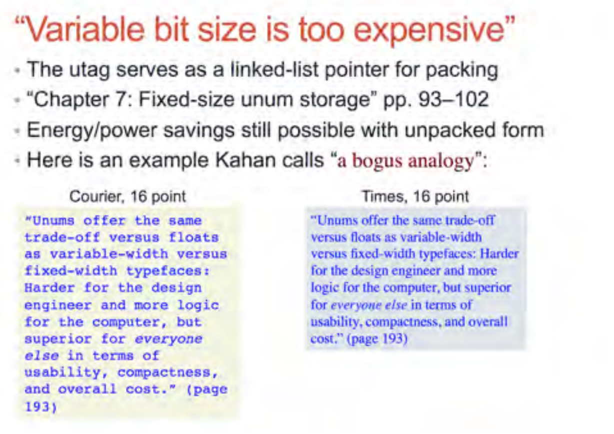
Commentary on *"THE END of ERROR — Unum Computing"*

by John L. Gustafson, (2015) CRC Press

So. He *was* kind enough to share with me the 38-page attack he wants to post about *The End of Error: Unum Arithmetic*. And so I was able to see some of the things he plans to say. I expect him, probably, to be covering some of these things today. Since this is the only thing that’s really finished, I won’t be talking about what are now called “unums type 1” or “unums 1.0.” I don’t have any material prepared about 2.0, because in fact I’m not finished with it yet, ready to show the world. It’s still in progress. So I will respond in part here to the “Bitter Blog” that I have seen. You see the “Table of Contents” there; I’m sure you can’t read those words.

First of all, I think everybody has this reaction: Variable bit size is just too darned expensive. To be able to change the widths and sizes of representations is going to be a headache for hardware designers, and for software, and storage, and so on. And I always sigh and say, well, it doesn’t *have* to be variable size; you can do these things fixed size. But if you *do* have to use variable size, you don’t need an additional set of pointers in order to pack and unpack these things. There’s these three bits—three fields in the *utag*. They serve as a pointer to the next unum, so you untangle it as a linked list; they already have the overhead necessary for packing and unpacking. No additional bits required. So, Chapter 7: the title of the Chapter is “Fixed Size Unum Storage.” And I don’t know how people can miss it, pages 93 to 102, talks about how you can make sure that you can fill matrix arrays with unums and still be able to have constant-time access to anything and do exactly what you would do with current floating-point arithmetic, no problems. There is still energy and power savings when you unpack things, leaving a few bits unused. Hardware designers know how to turn off bits if they know that certain fields within a 64-bit register are not going to be touched. So even within the chip you can still get energy-power savings in the unpacked form.

Now, this brings up an example that Kahan calls "A Bogus Analogy." Here's what I put, a version of it. In Courier, 16-point type, 'Unums offer the same tradeoff versus floats as variable width versus fixed width typefaces. Harder for the design engineer and more logic for the computer, but superior for *everyone* else in terms of usability, compactness, and overall cost.' Do that in Times, and notice how much space it takes up on the screen. Or how much space it would take up if printed.



"Variable bit size is too expensive"

- The utag serves as a linked-list pointer for packing
- "Chapter 7: Fixed-size unum storage" pp. 93–102
- Energy/power savings still possible with unpacked form
- Here is an example Kahan calls "a bogus analogy":

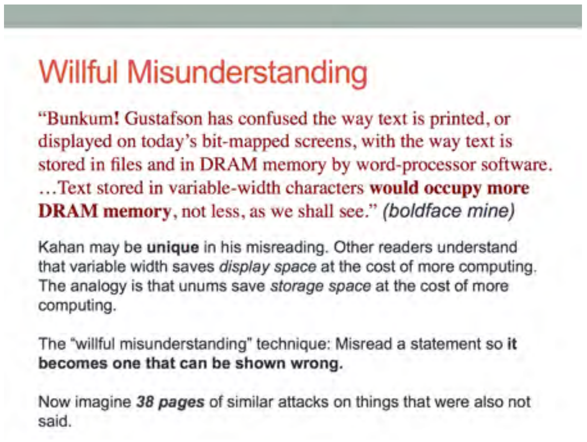
Courier, 16 point

```
"Unums offer the same
trade-off versus floats
as variable-width versus
fixed-width typefaces:
Harder for the design
engineer and more logic
for the computer, but
superior for everyone
else in terms of
usability, compactness,
and overall cost." (page
193)
```

Times, 16 point

"Unums offer the same trade-off versus floats as variable-width versus fixed-width typefaces: Harder for the design engineer and more logic for the computer, but superior for *everyone else* in terms of usability, compactness, and overall cost." (page 193)

Everybody I know who has read this, understood the analogy. Except… Professor Kahan. He said, 'BUNKUM! Gustafson has confused the way text is printed, or displayed. It would occupy **more** DRAM memory, not less!' …Uh, yeah, it takes more DRAM memory to take *less space on a piece of paper or on a screen*, which might be the most valuable resource these days. So, I was saying that if you have more memory, you can do, certainly, tighter fitting of things if they only have to take up the amount of space that they each require, instead of everything has to as big as the largest letter. So the same way, if you have variable size arithmetic, you can shrink and adjust and in general you will save space using only as many bits as are needed in memory for the numbers.



Willful Misunderstanding

"Bunkum! Gustafson has confused the way text is printed, or displayed on today's bit-mapped screens, with the way text is stored in files and in DRAM memory by word-processor software. …Text stored in variable-width characters **would occupy more DRAM memory**, not less, as we shall see." *(boldface mine)*

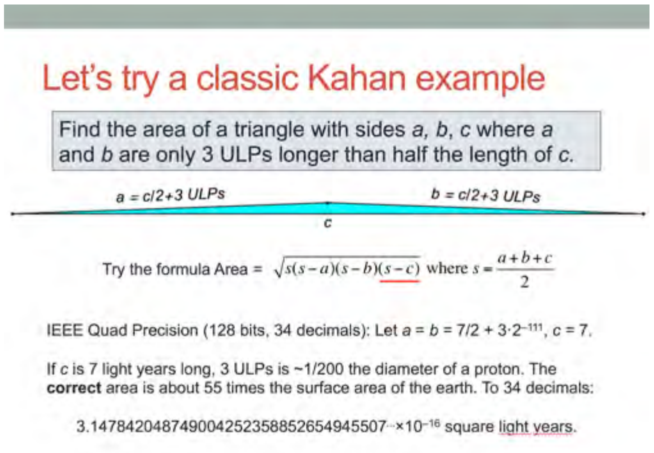
Kahan may be **unique** in his misreading. Other readers understand that variable width saves *display space* at the cost of more computing. The analogy is that unums save *storage space* at the cost of more computing.

The "willful misunderstanding" technique: Misread a statement so **it becomes one that can be shown wrong.**

Now imagine ***38 pages*** of similar attacks on things that were also not said.

I have not met any other reader who misunderstood this, and then attacked it. But here's the fallacy, and it's a 'willful misunderstanding' technique. You take something I'd written, or anybody had written, and misread it to be something that's clearly a false statement, and then you attack *that*, even though it's not what I said. So as I read through this 38-page document, imagine… 38 pages of attacks just like this one. I didn't say that! So he's attacking things that were not the position I was taking at all.

I want to go over a classic Kahan example, the area of a triangle that is very, very skinny. So let's take a triangle, sides *a*, *b*, and *c* where *c* is the long side, and *a* and *b* are just barely, *barely* more than half the length of *c*. So it pops up a little bit, like buckled concrete in the sun. And it's only 3 ULPs larger, and then you're going to try to use the formula, the classic generic formula, of the square root of *s*, *s* minus *a*, *s* minus *b*, *s* minus *c*, after finding the sum *a* plus *b* plus *c* over two. This is hazardous, because of *that*:



Let's try a classic Kahan example

Find the area of a triangle with sides *a*, *b*, *c* where *a* and *b* are only 3 ULPs longer than half the length of *c*.

$a = c/2+3$ ULPs

$b = c/2+3$ ULPs

c

Try the formula Area = $\sqrt{s(s-a)(s-b)\underline{(s-c)}}$ where $s = \frac{a+b+c}{2}$

IEEE Quad Precision (128 bits, 34 decimals): Let $a = b = 7/2 + 3 \cdot 2^{-111}$, $c = 7$.

If *c* is 7 light years long, 3 ULPs is ~1/200 the diameter of a proton. The **correct** area is about 55 times the surface area of the earth. To 34 decimals:

$3.147842048749004252358852654945507\cdots \times 10^{-16}$ square light years.

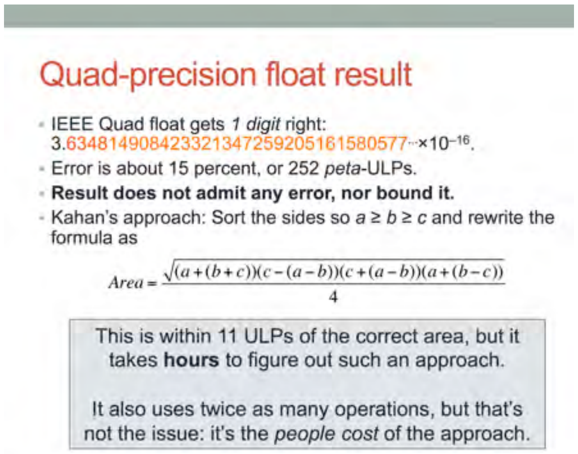
[*points to the (s – c) term*]

s and *c* are very close; in fact they're only six ULPs apart. So you're going to be taking two very similar numbers and subtracting, not necessarily a problem if it's *exact*. You don't lose any information if those two are properly represented. But if there's any problem that you rounded *s* when you calculated it, that's going to amplify the relative error in a massive way.

So in IEEE quad precision, let's pull out all the stops and use the best stuff you can get right now: 128 bits, 34 decimals. We'll let *a* equal *b*, seven halves, and add three ULPs where an ULP is two to the negative 111th power. That's a really small number. And *c* equals exactly seven. And just to make it a little bit more interesting, let's say that the units of length are *light-years*. So, seven light-years away, that's like, Alpha Centauri in each direction, our nearest star. Three ULPs is 1/200th the diameter of a proton. Amazingly, that manages to pop the triangle up by roughly the width of a standard door, 85 centimeters. So finally we have a number we can kind of comprehend. And that area, of that *extremely* long strip, is about 55 times the area of the earth. It's getting almost up to the surface area of one of the gas giants, like Neptune.

So here is the answer correct to 34 decimals, in square light-years. Just remember the first few decimals, please. It's easy to remember 3.14, times ten to the minus 16th, square light-years. That's what we *should* get.

The quad precision float result is, it gets *one digit* right. A lot of work for one digit! Three point *six*, times ten to the minus 16th. About fifteen percent too high. That's—in terms of ULPs, that's 250-some *peta*-ULPs. The result does not admit any error, nor bound it. It gives no warning to the user. You're expected to do error analysis to figure out that this happened to you because it's completely silent. Nothing protects you from this error.



Quad-precision float result

- IEEE Quad float gets *1 digit* right: $3.634814908423321347259205161580577\cdots\times 10^{-16}$.
- Error is about 15 percent, or 252 *peta*-ULPs.
- **Result does not admit any error, nor bound it.**
- Kahan's approach: Sort the sides so $a \geq b \geq c$ and rewrite the formula as

$$Area = \frac{\sqrt{(a+(b+c))(c-(a-b))(c+(a-b))(a+(b-c))}}{4}$$

This is within 11 ULPs of the correct area, but it takes **hours** to figure out such an approach.

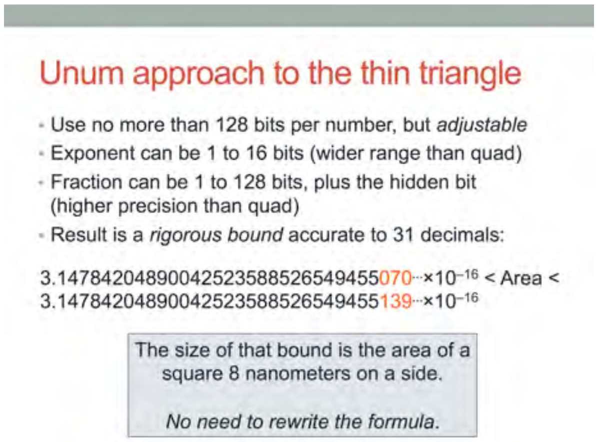
It also uses twice as many operations, but that's not the issue: it's the *people cost* of the approach.

Professor Kahan suggests that you *sort* the sides so that they're *a* greater than *b* greater than *c*, and then apply *this* formula which is algebraically equivalent. I would love to know how many hours it took him to find that formula. Or how many hours it would take *you* to figure out that formula.

KAHAN: Half an hour.

GUSTAFSON: Half an hour. [*laughter*] Half an hour of a Berkeley professor's time, conservatively let's suppose your time is worth three hundred dollars an hour, it's probably more. So it cost about one hundred and fifty dollars to get that formula out, in order to save the computer from having to do a little bit more work. That's now provably within eleven ULPs of the correct area. But it usually takes *hours* to figure out that kind of approach. It also uses twice as many operations. I have no objection to using twice as many operations; we've got operations to *burn*, but that's not the issue. It's the *people cost* of the approach. It's the wrong direction. We need to make things easier for users.

Let's try unum arithmetic. And let's make sure that we're not cheating, so I'm not going to use *any* more than 128 bits, period. We were limited on the quads, 128 bits per number, but it can adjust, it will sometimes use less than 128 but it will never go over. The exponent is more flexible. It can go from one to sixteen bits, so I've actually got a bigger dynamic range. The fraction can be one to 128 bits, plus the hidden bit. That's higher precision than quad.



Unum approach to the thin triangle

- Use no more than 128 bits per number, but *adjustable*
- Exponent can be 1 to 16 bits (wider range than quad)
- Fraction can be 1 to 128 bits, plus the hidden bit (higher precision than quad)
- Result is a *rigorous bound* accurate to 31 decimals:

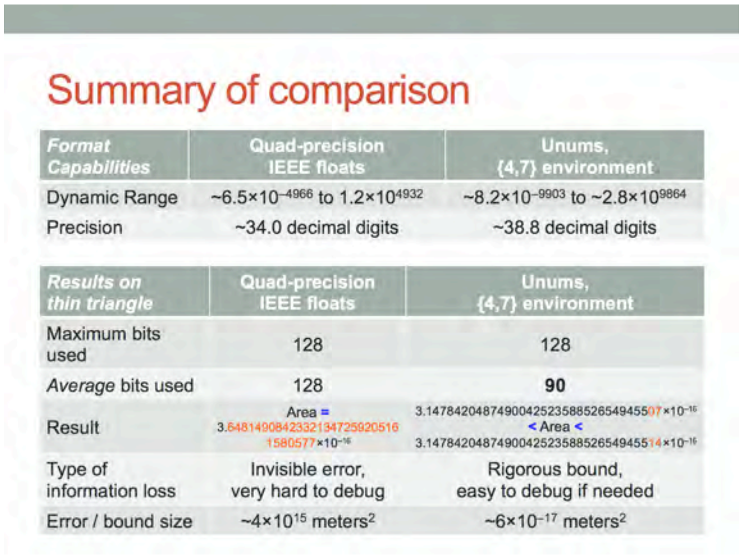
$$3.147842048900425235885265494550\,70\cdots\times 10^{-16} < \text{Area} < 3.147842048900425235885265494551\,39\cdots\times 10^{-16}$$

The size of that bound is the area of a square 8 nanometers on a side.

No need to rewrite the formula.

The result is a rigorous bound, accurate to 31 decimals. The area is strictly bounded by those two numbers. They don't start differing (you see the orange over there) until the 32nd decimal. The size of that bound is a square about eight nanometers on a side, so we're getting down to about the limits of present lithography, in order to find the error in finding this thing that has 55 times the area of the planet Earth.

OK, so in summary, let's take a look at quad-precision IEEE floats versus unums, also 128 bits. The dynamic range is almost the *square* of what you get with quad-precision floats. Instead of 34 decimals, I could have up to 38.8. And then, trying it on the triangle, I used an average of 90 bits; they shrink. And the result is, the float result says, area *exactly equals* this wrong number, versus unums that say the answer is definitely between these two numbers, provably. The *type* of information loss in the case of floats is invisible error, very hard to debug. When I actually did this example, I used unums to do the quad. I just set the exponent to the right size and the fraction to the right size, and simulated it with my environment. And at the point where you get the rounding error, where suddenly six ULPs becomes eight ULPs, it's instantly visible. It was *so* easy to spot. There was no problem. Whereas I think if it just keeps cranking away, always giving what looks like 34 decimals of precision, you're going to have to walk through it, one step at a time, very laboriously, so by having more information, it makes it easier to debug and figure out what's going on in your computation.

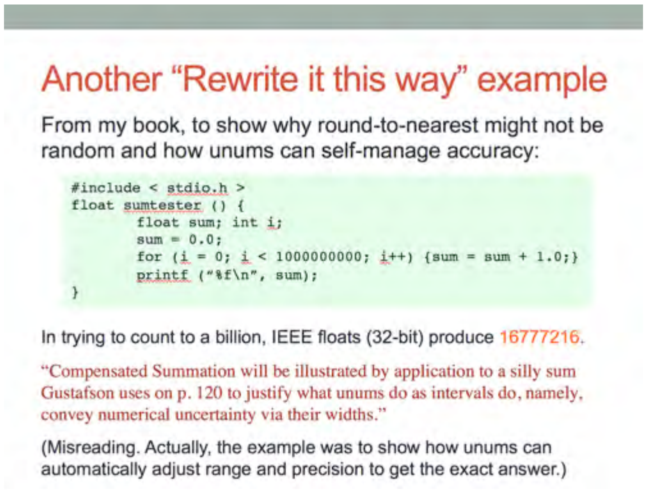


Summary of comparison

Format Capabilities	Quad-precision IEEE floats	Unums, {4,7} environment
Dynamic Range	~6.5×10^{-4966} to 1.2×10^{4932}	~8.2×10^{-9903} to ~2.8×10^{9864}
Precision	~34.0 decimal digits	~38.8 decimal digits

Results on thin triangle	Quad-precision IEEE floats	Unums, {4,7} environment
Maximum bits used	128	128
Average bits used	128	**90**
Result	Area = 3.6481490842332134725920516 1580577$\times10^{-16}$	3.14784204874900425235885265494550 7$\times10^{-16}$ < Area < 3.14784204874900425235885265494551 4$\times10^{-16}$
Type of information loss	Invisible error, very hard to debug	Rigorous bound, easy to debug if needed
Error / bound size	~4×10^{15} meters^2	~6×10^{-17} meters^2

Here's another example of, "Well, just re-write the code this way." Now I did an example just to show how the rounding error might not be random, and also how the system could automatically adjust to give you higher accuracy and self-manage accuracy. The simplest example I could think of was just, count to a billion. Just keep on adding one, adding one, adding one, a billion times, and see how you do with



Another "Rewrite it this way" example

From my book, to show why round-to-nearest might not be random and how unums can self-manage accuracy:

```
#include < stdio.h >
float sumtester () {
        float sum; int i;
        sum = 0.0;
        for (i = 0; i < 1000000000; i++) {sum = sum + 1.0;}
        printf ("%f\n", sum);
}
```

In trying to count to a billion, IEEE floats (32-bit) produce 16777216.

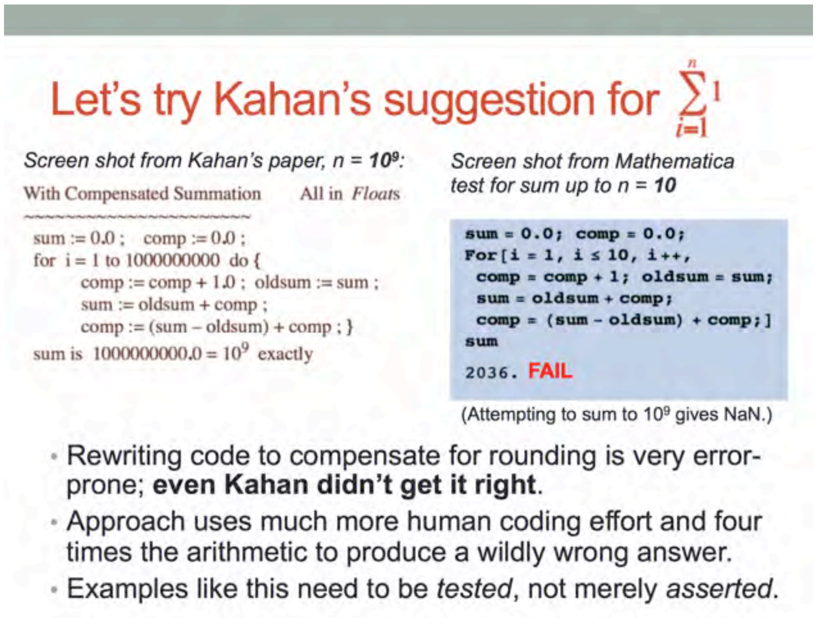
"Compensated Summation will be illustrated by application to a silly sum Gustafson uses on p. 120 to justify what unums do as intervals do, namely, convey numerical uncertainty via their widths."

(Misreading. Actually, the example was to show how unums can automatically adjust range and precision to get the exact answer.)

floating-point arithmetic. Floating point, floats, of course, can *represent* the number one billion, but they will get stuck, as we heard in an earlier talk. In trying to count to a billion, IEEE floats get stuck at two to the 24th, sixteen million some. They can't get any higher because as you add one, it keeps on rounding down. Ah, well, this is a well-known problem, and Dr. Kahan says in his big write-up:

"Compensated summation will be illustrated by application to a *silly sum* Gustafson uses on page 120 to justify what unums do as intervals do, namely, convey numerical uncertainty via their widths."

I don't think that's silly at all. I like to know when something has suddenly lost its ability to give me a precise answer. And that's actually not what this section was about. But let's try it. So. This is a screen shot from Kahan's paper. With *n* equal to ten to the ninth, compensated summation, all in floats. Set the sum. Now you've introduced two new variables: oldsum, and compensation, and you keep track of this. Your summation in a program often will be scattered all over the program, so for *every sum* you're accumulating you're going to have to keep two auxiliary variables like this and keep track of them. What do suppose this does for the maintainability of the code? And the portability? And the chance that you might have made a bug somewhere, maybe an error? I'd say it goes *sky high*.



Let's try Kahan's suggestion for $\sum_{i=1}^{n} 1$

Screen shot from Kahan's paper, n = ***10^9****:*

With Compensated Summation   All in *Floats*

```
sum := 0.0 ;    comp := 0.0 ;
for  i = 1 to 1000000000  do {
        comp := comp + 1.0 ;  oldsum := sum ;
        sum := oldsum + comp ;
        comp := (sum – oldsum) + comp ; }
sum is  1000000000.0 = 10^9  exactly
```

Screen shot from Mathematica test for sum up to n = ***10***

```
sum = 0.0; comp = 0.0;
For[i = 1, i ≤ 10, i++,
 comp = comp + 1; oldsum = sum;
 sum = oldsum + comp;
 comp = (sum - oldsum) + comp;]
sum
2036. FAIL
```

(Attempting to sum to 10^9 gives NaN.)

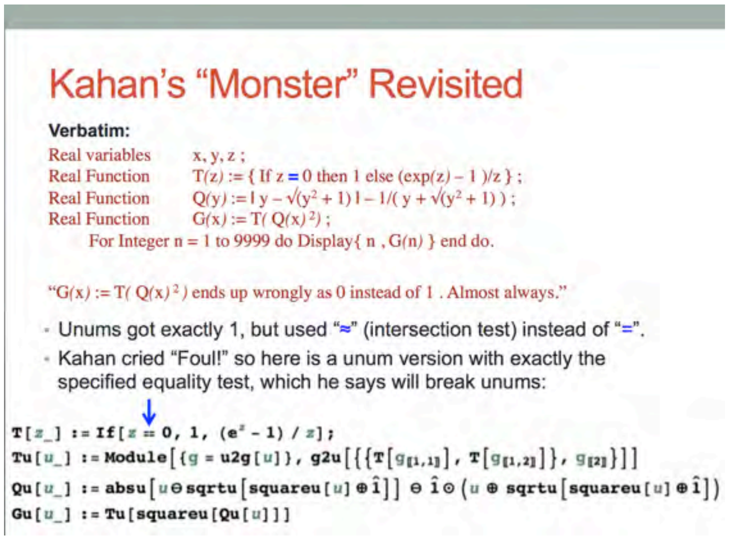
- Rewriting code to compensate for rounding is very error-prone; **even Kahan didn't get it right**.
- Approach uses much more human coding effort and four times the arithmetic to produce a wildly wrong answer.
- Examples like this need to be *tested*, not merely *asserted*.

So I coded it up in Mathematica, *exactly* as you see here; it's a slight translation from C, from whatever pidgin language that is to Mathematica, but I think it is exactly what you see. And I summed from one to ten. I got the number 2036. I tried adding one plus one. It gave me *four* [*pained expression*]. I tried going all the way to a billion. NaN. I think sum minus oldsum is infinity minus infinity, and it said, not a number.

What is going on? This is a FAIL. He got an error when he coded it up! And he put it right there, in the code, in the text. So even *Kahan* didn't get it right. In rewriting code to try to compensate for these things, and say, "Let's just rearrange all this stuff" is *very error-prone*. It also makes your code completely *cryptic* to anyone trying to understand "what are you doing?" They won't be able to see it. Plus, it gets huge, potentially huge mistakes. So an approach that uses more coding effort and three times as many bits to produce a wildly wrong answer does not impress me.

Examples like this need to be *tested*, not merely asserted. He says, "Sum equals ten to the ninth, exactly!" No, it's not. I wonder how many other things he says, "This is true, for this formula."

When I wrote the book, I did not use a word processor. I used Mathematica to generate the book as a computer program. All the examples, therefore, are *computed output*. So any mistake I would make would be, like, transcribing a result somewhere into the comment field. I might make a mistake doing that, but other than that, it's all pretty much bombproof.



Kahan's "Monster" Revisited

Verbatim:

```
Real variables     x, y, z ;
Real Function      T(z) := { If z = 0 then 1 else (exp(z) – 1 )/z } ;
Real Function      Q(y) := | y – √(y² + 1) | – 1/( y + √(y² + 1) ) ;
Real Function      G(x) := T( Q(x)² ) ;
      For Integer n = 1 to 9999 do Display{ n , G(n) } end do.
```

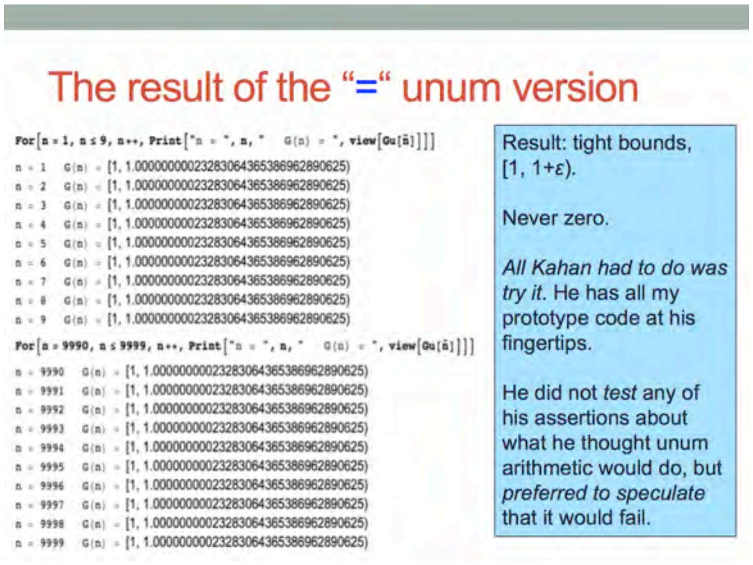
"$G(x) := T(\,Q(x)^2\,)$ ends up wrongly as 0 instead of 1 . Almost always."

- Unums got exactly 1, but used "≈" (intersection test) instead of "=".
- Kahan cried "Foul!" so here is a unum version with exactly the specified equality test, which he says will break unums:

```
T[z_] := If[z == 0, 1, (e^z - 1) / z];
Tu[u_] := Module[{g = u2g[u]}, g2u[{{T[g[[1,1]]], T[g[[1,2]]]}, g[[2]]}]]
Qu[u_] := absu[u ⊖ sqrtu[squareu[u] ⊕ 1̂]] ⊖ 1̂ ⊙ (u ⊕ sqrtu[squareu[u] ⊕ 1̂])
Gu[u_] := Tu[squareu[Qu[u]]]
```

Now I had a—one of my favorite examples from Kahan, is, I call it his "monster," and that's meant as a flattering term because I think it's a really clever example. But it's a very tough one for floats. There's three functions, *T*, *Q*, and *V*, I got the letters right this time, I think, and *Q* is, *should* be identically zero. But floats can't do the square roots and reciprocals exactly, so they tend to get a little slop. You square that number in the bottom function, *G*, and that means you're going to get something a little bit bigger than zero. And then you use the removable singularity function *e* to the *z* minus one all over *z*, and floats will tend to get zero most of the time. They might get lucky and actually get *one*, which is the correct answer.

OK. So when I coded this up, I used the approximately-equals sign, which actually means *intersects*, so if a unum interval intersects zero, then that's the wobbly "not nowhere equal" sign. I didn't use exact equality, there, shown in bright blue, just the way he showed it in bright blue because he wanted to emphasize that I was cheating. So he got bent out of shape by that, so I coded it up with an equals sign, and you can see it right there: If *z* equals zero, one; else *e* to the *z* minus 1 over *z*. I don't know how more direct I could code *exactly* what he wanted as his example problem. The other three lines are just putting it in unum form, as unum operations.



The result of the "=" unum version

```
For[n = 1, n ≤ 9, n++, Print["n = ", n, "   G(n) = ", view[Gu[n̂]]]]
n = 1   G(n) = [1, 1.000000000232830643653869628906250)
n = 2   G(n) = [1, 1.000000000232830643653869628906250)
n = 3   G(n) = [1, 1.000000000232830643653869628906250)
n = 4   G(n) = [1, 1.000000000232830643653869628906250)
n = 5   G(n) = [1, 1.000000000232830643653869628906250)
n = 6   G(n) = [1, 1.000000000232830643653869628906250)
n = 7   G(n) = [1, 1.000000000232830643653869628906250)
n = 8   G(n) = [1, 1.000000000232830643653869628906250)
n = 9   G(n) = [1, 1.000000000232830643653869628906250)
For[n = 9990, n ≤ 9999, n++, Print["n = ", n, "   G(n) = ", view[Gu[n̂]]]]
n = 9990   G(n) = [1, 1.000000000232830643653869628906250)
n = 9991   G(n) = [1, 1.000000000232830643653869628906250)
n = 9992   G(n) = [1, 1.000000000232830643653869628906250)
n = 9993   G(n) = [1, 1.000000000232830643653869628906250)
n = 9994   G(n) = [1, 1.000000000232830643653869628906250)
n = 9995   G(n) = [1, 1.000000000232830643653869628906250)
n = 9996   G(n) = [1, 1.000000000232830643653869628906250)
n = 9997   G(n) = [1, 1.000000000232830643653869628906250)
n = 9998   G(n) = [1, 1.000000000232830643653869628906250)
n = 9999   G(n) = [1, 1.000000000232830643653869628906250)
```

Result: tight bounds, $[1, 1+\varepsilon)$.

Never zero.

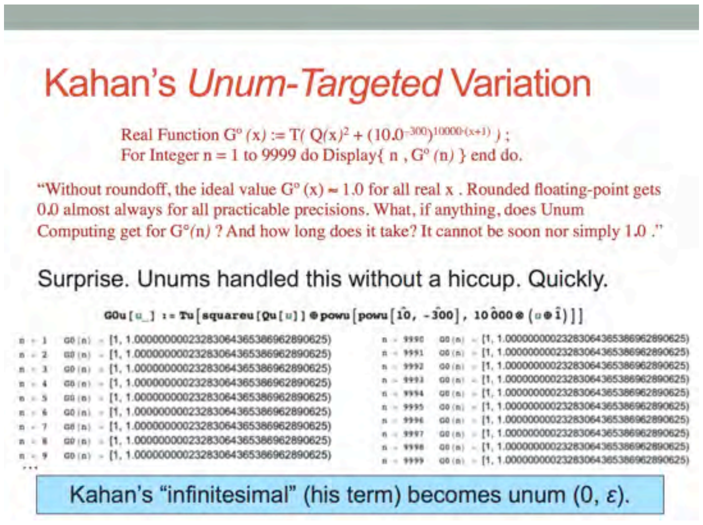
All Kahan had to do was try it. He has all my prototype code at his fingertips.

He did not *test* any of his assertions about what he thought unum arithmetic would do, but *preferred to speculate* that it would fail.

Here's the result. Instead of getting exactly 1, I got 1 plus an ULP. I'm showing you the first nine and the last nine, but it's all very boring. The list of all ten thousand numbers is all exactly one, and an ULP. And they're all of the form one plus epsilon, and this is a half-open interval. It includes the number one, and is a little big bigger, goes a little bit to the right, but it's never *zero*. It never makes that mistake.

Now he would have found that out if he'd *tried* it. He has access to all my code at his fingertips, as do all of you; it's a free download. Just go to the CRC publication site and use it. Or if you get it in Julia, or—there's about five versions of unum arithmetic now in Julia on the web, and one in Python that's a line-by-line transcription of the prototype, C versions are popping up everywhere, C++... it won't be long before, name your language, it's going to have some unum support from somebody. So I don't know why he didn't just *try* these things and he just said, "That's not gonna work." He *speculated* that it would fail.

So he made one up, and I feel so honored that I've earned my first, targeted, "This will BREAK unums!" from Dr. Kahan. What he did was, he added on a number, ten to the negative 300th power, all raised to the power of ten thousand, times *x* plus 1, which could be as large—or I should say as *small* as—ten to the negative thirty billion. So I'm very flattered that he thought it would take a number like ten to the negative thirty billion to break unum arithmetic. Or at least slow it down, so it would take an intractable amount of time. "What, if anything, does unum computing get for this? How long does it take? It cannot be soon nor simply 1.0."



Kahan's *Unum-Targeted* Variation

Real Function $G^{o}(x) := T(\, Q(x)^2 + (10.0^{-300})^{10000\cdot(x+1)}\,)$;
For Integer n = 1 to 9999 do Display{ n , $G^{o}(n)$ } end do.

"Without roundoff, the ideal value $G^{o}(x) = 1.0$ for all real x . Rounded floating-point gets 0.0 almost always for all practicable precisions. What, if anything, does Unum Computing get for $G^{o}(n)$? And how long does it take? It cannot be soon nor simply 1.0 ."

Surprise. Unums handled this without a hiccup. Quickly.

```
G0u[u_] := Tu[squareu[Qu[u]] ⊕ powu[powu[1̂0, -3̂00], 10 0̂00 ⊗ (u ⊕ 1̂)]]
n = 1   G0(n) = [1, 1.000000000232830643653869628906250)    n = 9990   G0(n) = [1, 1.000000000232830643653869628906250)
n = 2   G0(n) = [1, 1.000000000232830643653869628906250)    n = 9991   G0(n) = [1, 1.000000000232830643653869628906250)
n = 3   G0(n) = [1, 1.000000000232830643653869628906250)    n = 9992   G0(n) = [1, 1.000000000232830643653869628906250)
n = 4   G0(n) = [1, 1.000000000232830643653869628906250)    n = 9993   G0(n) = [1, 1.000000000232830643653869628906250)
n = 5   G0(n) = [1, 1.000000000232830643653869628906250)    n = 9994   G0(n) = [1, 1.000000000232830643653869628906250)
n = 6   G0(n) = [1, 1.000000000232830643653869628906250)    n = 9995   G0(n) = [1, 1.000000000232830643653869628906250)
n = 7   G0(n) = [1, 1.000000000232830643653869628906250)    n = 9996   G0(n) = [1, 1.000000000232830643653869628906250)
n = 8   G0(n) = [1, 1.000000000232830643653869628906250)    n = 9997   G0(n) = [1, 1.000000000232830643653869628906250)
n = 9   G0(n) = [1, 1.000000000232830643653869628906250)    n = 9998   G0(n) = [1, 1.000000000232830643653869628906250)
...                                                          n = 9999   G0(n) = [1, 1.000000000232830643653869628906250)
```

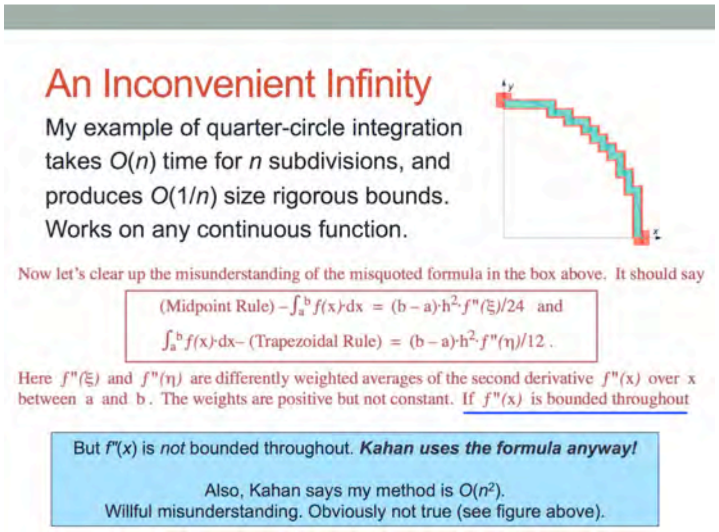
Kahan's "infinitesimal" (his term) becomes unum $(0, \varepsilon)$.

Wanna bet?

It had absolutely no trouble handling this, and it didn't slow it down one bit. For a very simple reason: What he called—that very small number—is immediately

turned into… the unum *zero to epsilon*. Open interval. It's not zero, it's too small to represent. This is the way unums handle what normally would become *underflow*. They don't underflow. They don't overflow, and they don't round. They just say, "It's in that interval, zero to epsilon, and I will keep track of it that way." You want to exponentiate it? Fine. Once again, it evaluates to one (*gestures the start of an interval*), to one plus epsilon (*gestures the end of an interval*). It didn't go any slower. It didn't require high precision. It—I guess you could say it just gave up. It didn't even attempt to represent this number. And Dr. Kahan referred to it as an "infinitesimal," which I think is a strange use of the word "infinitesimal" since it's clearly a number and not an infinitesimal. Anyway.

Here's what I call "An Inconvenient Infinity." And I just was finding a very simple example—as I say, this is a very broad audience book, so I said what's the area of a circle? Let's give it an arc of a circle, multiply by four and see how close to pi we can get with one-decimal precision, with very, very crude *limits*. So, if you divide—if you pick a point on the left, it will kind of like a burning fuse move over to the right and enclose the gap of what evaluates to *x* squared plus *y* squared equals one. All you have to know is how to multiply and add and test for equality with one. The error tends to be one over *n*, and it's a rigorous bound. This works on *any* continuous function. It doesn't care; in fact, with slight modification, it can even work on discontinuous functions. So I was looking for generality, *not* for "How tight can I make the bound."



An Inconvenient Infinity

My example of quarter-circle integration takes $O(n)$ time for n subdivisions, and produces $O(1/n)$ size rigorous bounds. Works on any continuous function.

Now let's clear up the misunderstanding of the misquoted formula in the box above. It should say

$$\boxed{\begin{array}{c}(\text{Midpoint Rule}) - \int_a^b f(x)\cdot dx = (b-a)\cdot h^2\cdot f''(\xi)/24 \quad \text{and} \\ \int_a^b f(x)\cdot dx - (\text{Trapezoidal Rule}) = (b-a)\cdot h^2\cdot f''(\eta)/12\ .\end{array}}$$

Here $f''(\xi)$ and $f''(\eta)$ are differently weighted averages of the second derivative $f''(x)$ over x between a and b. The weights are positive but not constant. If $f''(x)$ is bounded throughout

But $f''(x)$ is *not* bounded throughout. ***Kahan uses the formula anyway!***

Also, Kahan says my method is $O(n^2)$.
Willful misunderstanding. Obviously not true (see figure above).

So here is one of the things that Kahan wrote up in his attack on my book. He mentioned that the midpoint rule has an error term associated with it, the trapezoidal rule has an error associated with it, and if you use those two together, you can put a quote, *bound*, about the numerical evaluation of integrals. Pretty good! It does require that the second derivative exist and be bounded throughout. And as I read through his write-up, he mentioned several times that *f*-prime-prime is bounded throughout. And then he sort of noticed that… it's *not*. It's not bounded. For the quarter circle, it goes to infinity.

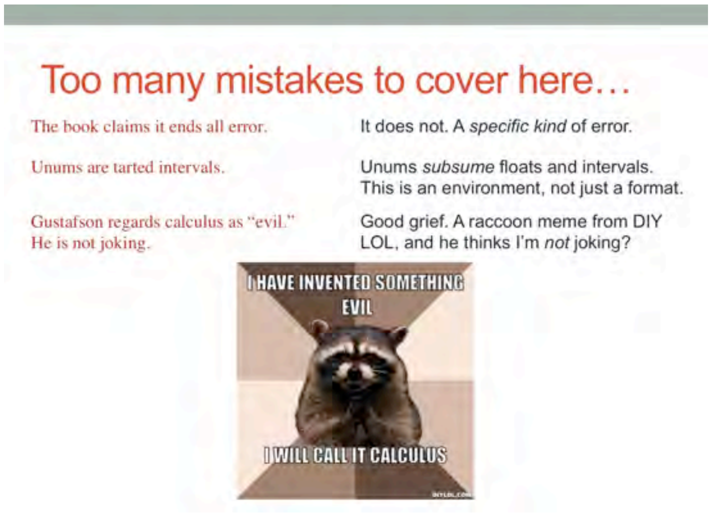
To my astonishment, *he used the formula anyway*. He used the bounds, ignoring the fact that his "this bound, plus infinity, is less than…" *You can't do that!* If my students do something like this, I take off at least half points on the test. There were very clear conditions for this theorem to work. And he ignored them because he wanted a nice, tight bound.

He also said that my method is order *n* squared, which I think, if you just *look* at the method, you can see that it's not order *n* squared. I do give an earlier method which populates all the squares underneath, and quickly say, "That's not the right way to do it; you do it this way" but he still stuck with "Oh, that's a really slow way!" It would be very, very slow just to fill in all the squares.

There's too many mistakes to really cover here, but perhaps the biggest one is he says, "The book claims it ends all error." It does nothing of the kind. It says it ends a *specific kind* of error. It's the error of rounding and sampling. Always substituting a specific exact number when you really wanted something different.

And he says, "Unums are tarted intervals." Well, unums subsume floats and intervals. You can use them to be floating point numbers; you can use them to be interval numbers… This is a whole environment. It's not just a number format. It's a set of rules for how to use them as well. What you can do in the scratch pad, for example; that's what gives you bitwise compatible answers. But you know, "tarted" means, like, too much lipstick, and makeup, and things like that? And when I look at these 34-digit numbers that claim to have all this precision, and only one of them is correct, I think *that's* the one that looks like it's wearing too much makeup. You see, unums *admit* what they don't know, whereas floats always *claim* all this precision, which might not be true!

"Gustafson regards calculus as evil. He is *not joking*!" I was amazed, because I used a raccoon meme that says DIYLOL on it. I don't know how *anyone* could know that I'm not joking. "I've invented something evil. I will call it calculus." I think most people would laugh when they see this, and that was my intent. I was trying to make the point that people tend to confuse when they are *calculating*, and

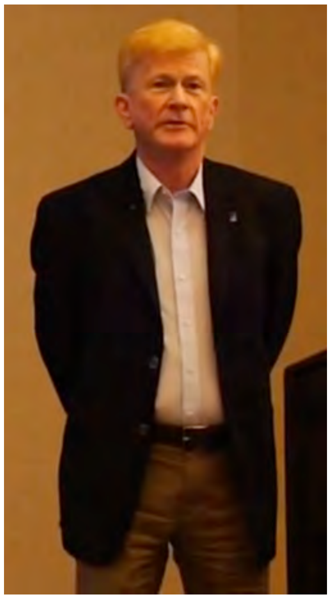


Too many mistakes to cover here…

The book claims it ends all error.	It does not. A *specific kind* of error.
Unums are tarted intervals.	Unums *subsume* floats and intervals. This is an environment, not just a format.
Gustafson regards calculus as "evil." He is not joking.	Good grief. A raccoon meme from DIY LOL, and he thinks I'm *not* joking?

I HAVE INVENTED SOMETHING EVIL
I WILL CALL IT CALCULUS

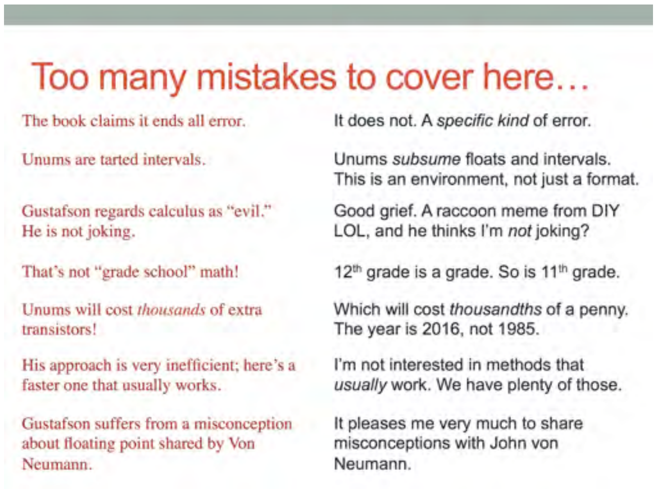
when they're using *calculus*, and they get in a lot of trouble doing that. There's a lot of problems that can be solved, without using calculus. I wanted to kind of push the limits: How far can we go without requiring calculus? …Anyway.



I would say in a few places that grade school math would suffice. And you say, "That's not grade school! That's trigonometry!" And, well, 12th grade is a grade. [*chuckling sounds*] 11th grade is a grade. That's why they're called… *grades*. If I meant elementary school, I would have said elementary school. This is not a book with elementary school math.

"Unums will cost thousands of extra transistors!" …Really? Which will cost *thousandths* of a penny? This is 2016; this is not 1985. A thousand transistors will cost you roughly one one-hundredth of a cent. Right now. And it's getting cheaper, still.

"His approach is very inefficient. Here's a *faster* one that usually works." How many times have I heard that. I am *so* tired of methods that "usually" work, and require an expert to tell you when they do work and when they don't work. I'm interested in very *robust* methods that you can hand to people who are not experts, that *work every time*, even if it's slowly.



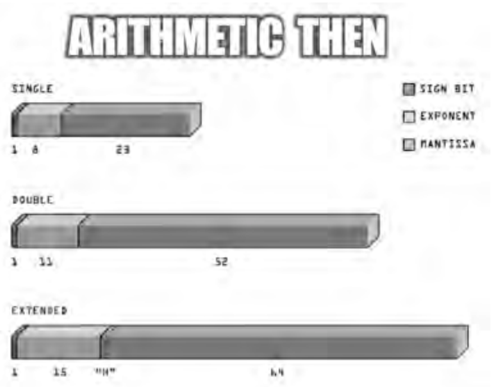
Too many mistakes to cover here…

The book claims it ends all error.	It does not. A *specific kind* of error.
Unums are tarted intervals.	Unums *subsume* floats and intervals. This is an environment, not just a format.
Gustafson regards calculus as "evil." He is not joking.	Good grief. A raccoon meme from DIY LOL, and he thinks I'm *not* joking?
That's not "grade school" math!	12th grade is a grade. So is 11th grade.
Unums will cost *thousands* of extra transistors!	Which will cost *thousandths* of a penny. The year is 2016, not 1985.
His approach is very inefficient; here's a faster one that usually works.	I'm not interested in methods that *usually* work. We have plenty of those.
Gustafson suffers from a misconception about floating point shared by Von Neumann.	It pleases me very much to share misconceptions with John von Neumann.

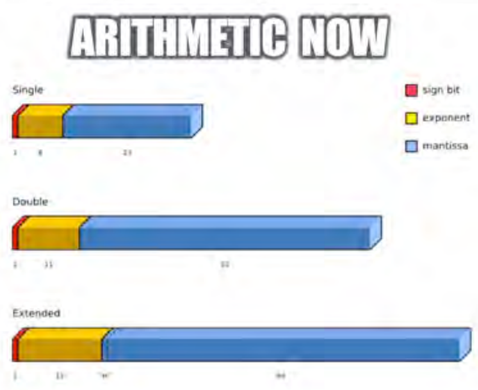
"Gustafson suffers from a misconception about floating point shared by von Neumann." It pleases me no end to share misconceptions with John von Neumann. One of the misconceptions he was talking about is actually, von Neumann was concerned that if you hand floating-point hardware to people, they would misuse it; they would just use it blindly, without doing proper error analysis, and they would get in trouble, because they'd get wrong answers and they wouldn't know it. That's my "misconception." I think it's perfectly accurate.

Let me just give you some memes. That was computers then. 1980s. Remember those days?

This is computers now. A thousand times cheaper. A thousand times faster.

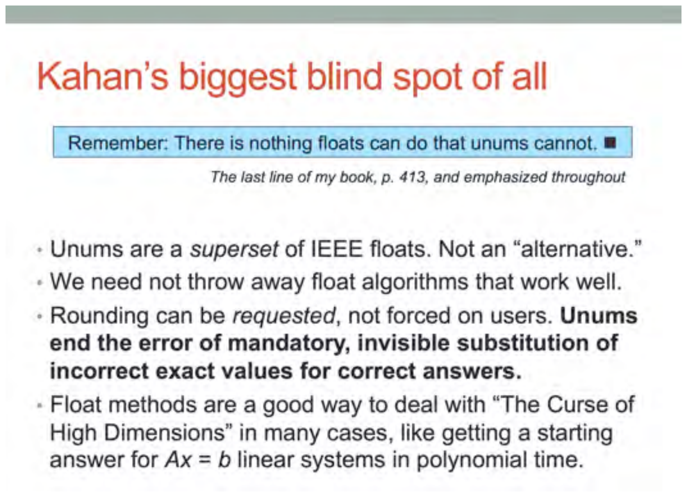


Here's what arithmetic looked like back then. Single, double, extended.



Here's what arithmetic looks like now.

Do you see my point? Isn't it time to do *something* about floating-point arithmetic? Maybe not what I did, but *something* to improve the quality? With a million times as much power at our fingertips, why can't we do something to make things a little less painful for people?



Kahan's biggest blind spot of all

Remember: There is nothing floats can do that unums cannot. ■

The last line of my book, p. 413, and emphasized throughout

- Unums are a *superset* of IEEE floats. Not an "alternative."
- We need not throw away float algorithms that work well.
- Rounding can be *requested*, not forced on users. **Unums end the error of mandatory, invisible substitution of incorrect exact values for correct answers.**
- Float methods are a good way to deal with "The Curse of High Dimensions" in many cases, like getting a starting answer for $Ax = b$ linear systems in polynomial time.

The biggest blind spot of all in the entire 38 pages of criticism he's levied—and watch for this when he speaks, because I hope he doesn't make it now. I say several times: "Remember: There is nothing floats can do that unums cannot." They are perfectly capable of mimicking every float that there is, all the IEEE floats, and all the sizes in between. That's just special cases of IEEE floats. It's the last line because I wanted to make sure people saw that. There's a chapter dedicated to this, called "Permission to Guess." Because what you do with unums is, you explicitly ask for a round operation. It doesn't happen every time.

So, you don't have to throw away all these great float algorithms that we've built up over the decades. I'm not saying we should. There are many things that, well, quite a few things you can find, where a floating-point algorithm is perfectly stable. It works, it's got a solid base of theory behind it, and there's absolutely no reason to use unums, in fact unums may complicate things to the point where you really wish you didn't use them. Then just use the unum as a float. It's a *superset*.

When you *request* rounding, it's very much more honest than when it happens every time without you being under control. Unums end the error of mandatory, invisible substitution of incorrect exact answers for correct answers. It gives you back the control over what's going on in the calculation, and makes it so visible, that you have a much better chance of knowing what went wrong, if something goes wrong with your calculation. It's easier to debug. *Far* easier.

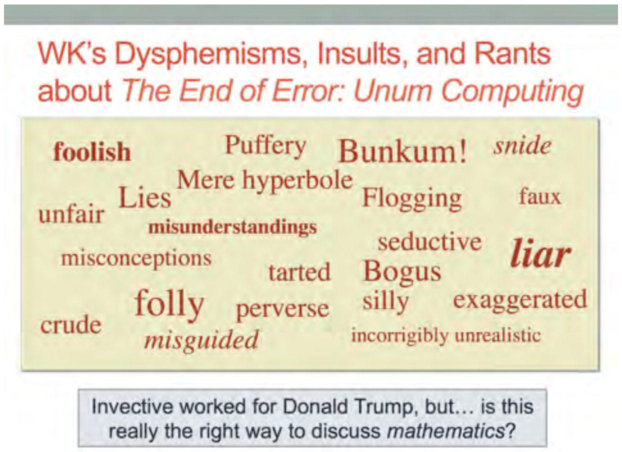
Float methods are an especially good way to deal with "the curse of high dimensions." If you aren't careful, you do get kind of a two-to-the-n^{th} effect when you're trying to solve something that has n degrees of freedom, whereas maybe with a float method, like Gaussian elimination, that solves the equations with order n cubed. You can use that to get a first guess, and if you *want* to do more with it using unums, you have that option. If not, you've got way more flexibility in the precision and the dynamic range than you've ever had with IEEE floats.

AUDIENCE MEMBER: Sorry, do you mean *float* or *unum* methods?

GUSTAFSON: Did I say something wrong? Float methods, in other words take a method that exists as a float method; use the unums to *do* the float method. Use that algorithm. Use that numerical method. Like, Gaussian elimination does *not* work

well with intervals. *Bad* idea, OK? It just *explodes*; you get a huge bound and it's not useful.

But I did mean float methods, here. So, use unums to perform float numerical methods.



WK's Dysphemisms, Insults, and Rants about *The End of Error: Unum Computing*

foolish Puffery Bunkum! *snide* unfair Lies Mere hyperbole Flogging faux misunderstandings seductive misconceptions tarted *liar* Bogus folly perverse silly exaggerated crude *misguided* incorrigibly unrealistic

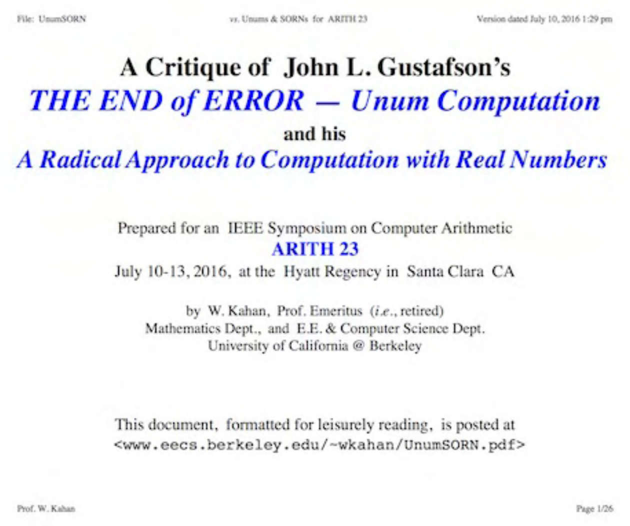
Invective worked for Donald Trump, but… is this really the right way to discuss *mathematics*?

As I look through these pages, I see things that you normally do not find in a math review. And I know invective worked for Donald Trump [*laughter*], but is this really the right way to discuss… mathematics?

Every time anyone poses, seriously, "Let's do something about floating point," there's this thing, that—in my head I envision it this way. "The Lord of the Reals… Does NOT Share Power." [*laughter*] Pretty soon the blogs are, you know, darting out saying "That's wrong!" "That will never work!"

I'll leave you with that and… Dr. Kahan? It's all yours. [*applause*]

KAHAN: Well, while you're up getting my slides up, what John has done here is to take examples, out of their context, starting with the triangle. The *flat* triangle is not the issue. The idea was to show that floating point will get results that are excessively wrong compared with the condition of the problem. And so if he had chosen to set up a different triangle, then we could have asked the question, well, how does your answer's error vary with the uncertainty in the data? Suppose the data's uncertain by a unit in the last place? Well, it turns out that for that flat triangle, the uncertainties are outright. For a *different* triangle, you get the same horrible uncertainty from Heron's Rule, but it's not deserved. In other words, you can't tell with ordinary floating-point arithmetic unless you do an error analysis, whether what you're getting is deserved, or not. And what John is saying, in effect, is "You don't have to *do* an error analysis! You just have to use unums!" Well, I'll answer the other issues that John raised on *our* time, but *my* time I'm going to use for something else. Let's observe— (*audience clamors for him to use microphone*)


File: UnumSORN vs. Unums & SORNs for ARITH 23 Version dated July 10, 2016 1:29 pm

A Critique of John L. Gustafson's
THE END of ERROR — Unum Computation
and his
A Radical Approach to Computation with Real Numbers

Prepared for an IEEE Symposium on Computer Arithmetic
ARITH 23
July 10-13, 2016, at the Hyatt Regency in Santa Clara CA

by W. Kahan, Prof. Emeritus (*i.e.*, retired)
Mathematics Dept., and E.E. & Computer Science Dept.
University of California @ Berkeley

This document, formatted for leisurely reading, is posted at
`<www.eecs.berkeley.edu/~wkahan/UnumSORN.pdf>`

Prof. W. Kahan Page 1/26

DEMMEL: Are you wearing a microphone?

KAHAN: Say again?

DEMMEL: You need to use the microphone.

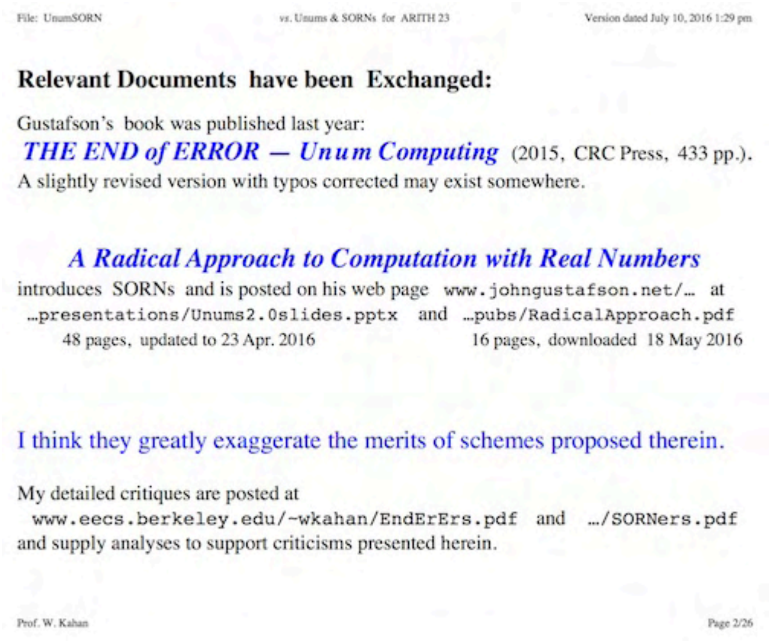
KAHAN: Do I need a microphone? [*audience shouts yes*] It's OK, I'll use this one. It's not a problem. If I can just get it turned on. [*audience chuckling*] Now. Is it working now?

DEMMEL: Hold it up.

KAHAN: Do I have to hold it here?

DEMMEL: Yeah, that's good.

KAHAN: That's fine. Let's, let's observe, and you can read this at your leisure after it's posted, on my web page. And if you'll go to the next page please.


File: UnumSORN vs. Unums & SORNs for ARITH 23 Version dated July 10, 2016 1:29 pm

Relevant Documents have been Exchanged:

Gustafson's book was published last year:
THE END of ERROR — Unum Computing (2015, CRC Press, 433 pp.).
A slightly revised version with typos corrected may exist somewhere.

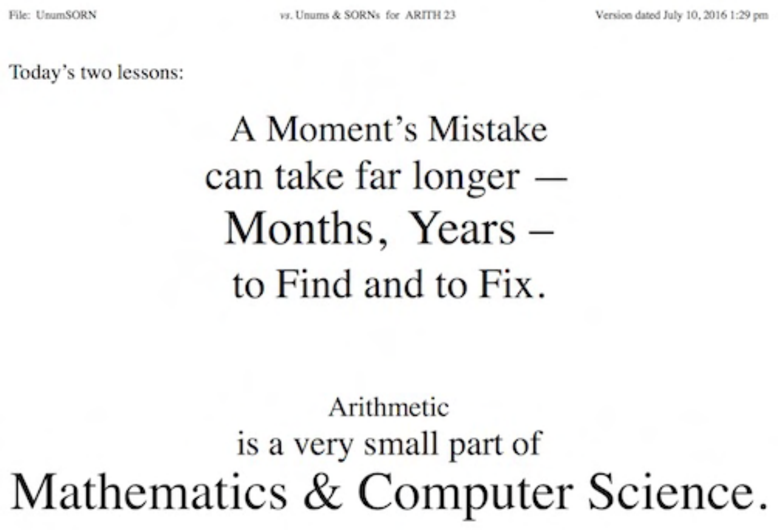
A Radical Approach to Computation with Real Numbers
introduces SORNs and is posted on his web page `www.johngustafson.net/…` at
`…presentations/Unums2.0slides.pptx` and `…pubs/RadicalApproach.pdf`
48 pages, updated to 23 Apr. 2016 16 pages, downloaded 18 May 2016

I think they greatly exaggerate the merits of schemes proposed therein.

My detailed critiques are posted at
`www.eecs.berkeley.edu/~wkahan/EndErErs.pdf` and `…/SORNers.pdf`
and supply analyses to support criticisms presented herein.

Prof. W. Kahan Page 2/26

The detailed critiques are here, so you can see whether John has represented, or misrepresented, the criticisms that I've levied. You can read it for yourself, but these are the documents that we exchanged. Those are John's documents, and then I explained we exchanged the first drafts of these documents. Let's go on to the next page now.


File: UnumSORN vs. Unums & SORNs for ARITH 23 Version dated July 10, 2016 1:29 pm

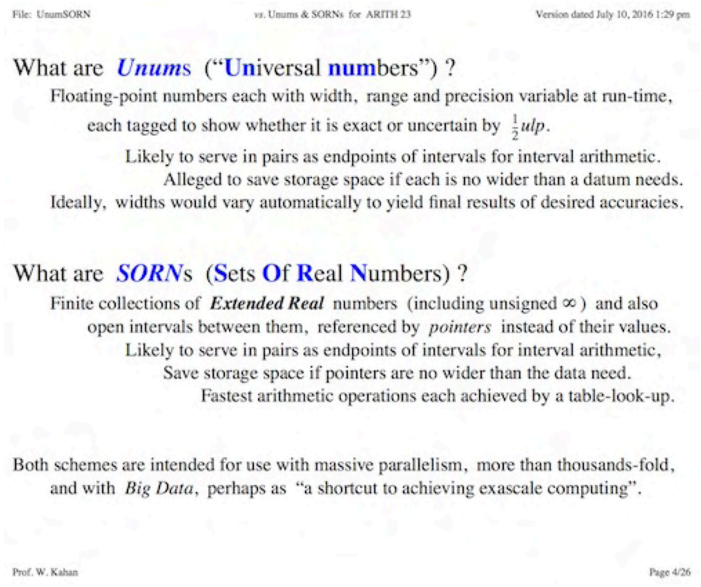
Today's two lessons:

A Moment's Mistake
can take far longer —
Months, Years –
to Find and to Fix.

Arithmetic
is a very small part of
Mathematics & Computer Science.

Here's the issue: It takes a long time to correct a mistake. And in any event, the arithmetic is a small part of the issues that we have to face. Let's go onto the next page, please.

First of all, what are unums? Well, I said they're *tarted intervals*. And so they are. Floating point numbers, each has its own width, range, and precision, you can vary them as much as you like. Each is tagged to show if it's uncertain or not. Usually you use them in pairs, because they're endpoints of intervals. So they make up and are included in the interval. And they're alleged to save storage space. Ah! That's going to be the critical issue. Do they really


File: UnumSORN vs. Unums & SORNs for ARITH 23 Version dated July 10, 2016 1:29 pm

What are *Unums* ("Universal numbers") ?
Floating-point numbers each with width, range and precision variable at run-time,
each tagged to show whether it is exact or uncertain by $\frac{1}{2}$*ulp*.
Likely to serve in pairs as endpoints of intervals for interval arithmetic.
Alleged to save storage space if each is no wider than a datum needs.
Ideally, widths would vary automatically to yield final results of desired accuracies.

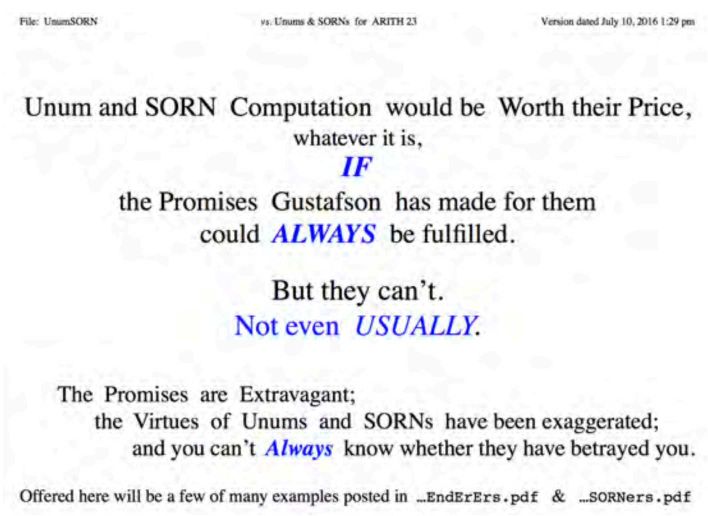
What are *SORNs* (Sets Of Real Numbers) ?
Finite collections of ***Extended Real*** numbers (including unsigned ∞) and also
open intervals between them, referenced by *pointers* instead of their values.
Likely to serve in pairs as endpoints of intervals for interval arithmetic,
Save storage space if pointers are no wider than the data need.
Fastest arithmetic operations each achieved by a table-look-up.

Both schemes are intended for use with massive parallelism, more than thousands-fold,
and with *Big Data*, perhaps as "a shortcut to achieving exascale computing".

Prof. W. Kahan Page 4/26

save storage space? And ideally, the widths would vary automatically. Of course, that would happen with interval arithmetic if we could get it, with varying precision. But the people who are trying to standardize interval arithmetic say they have enough trouble grandfathering in old schemes so they aren't interested in that.

But about SORNs. SORNs don't have variable precision; you've got to commit yourself. They are finite collections of extended real numbers; that means they include infinity. They allow open intervals. They're also serve in pairs. And they typically save storage space because what you're manipulating are not the numbers, but *pointers* to the values. And so, it's *possible* to run this arithmetic very, very fast. Now, both schemes are typically produced with massive parallelism. And when we look at this we'll see how massively parallelism they have to be. Next, please.


File: UnumSORN vs. Unums & SORNs for ARITH 23 Version dated July 10, 2016 1:29 pm

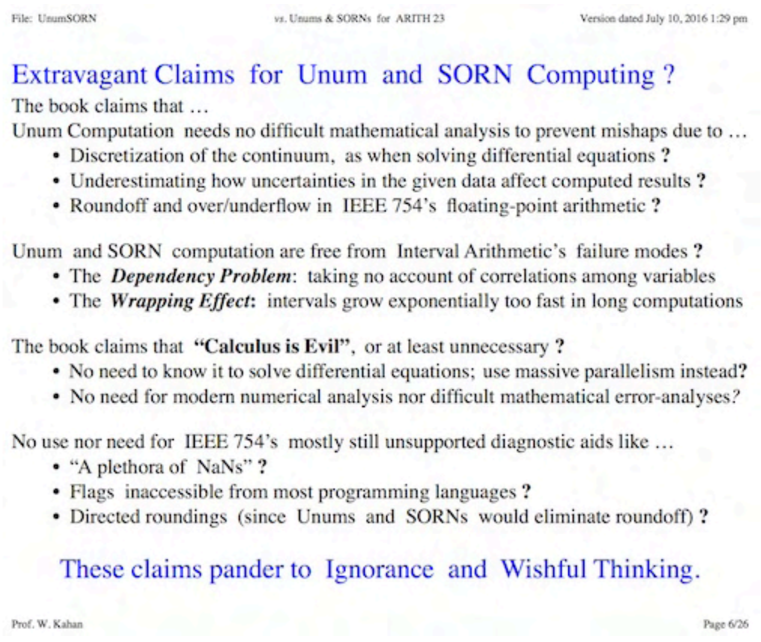
Unum and SORN Computation would be Worth their Price,
whatever it is,
IF
the Promises Gustafson has made for them
could ***ALWAYS*** be fulfilled.

But they can't.
Not even *USUALLY.*

The Promises are Extravagant;
the Virtues of Unums and SORNs have been exaggerated;
and you can't ***Always*** know whether they have betrayed you.

Offered here will be a few of many examples posted in `…EndErErs.pdf` & `…SORNers.pdf`

The issue here, is… not so much whether they would be worthwhile, if what John says were true. They *would* be! It would be *lovely* to be able to get away from error analysis. I would rather do something interesting. [*laughter*] But they *can't*. And it's possible to prove it from fundamentals. However I'd rather not bore you with fundamentals; I'll try to give you counterexamples. Let's go to the next page.


File: UnumSORN vs. Unums & SORNs for ARITH 23 Version dated July 10, 2016 1:29 pm

Extravagant Claims for Unum and SORN Computing ?

The book claims that …
Unum Computation needs no difficult mathematical analysis to prevent mishaps due to …
- Discretization of the continuum, as when solving differential equations ?
- Underestimating how uncertainties in the given data affect computed results ?
- Roundoff and over/underflow in IEEE 754's floating-point arithmetic ?

Unum and SORN computation are free from Interval Arithmetic's failure modes ?
- The ***Dependency Problem***: taking no account of correlations among variables
- The ***Wrapping Effect***: intervals grow exponentially too fast in long computations

The book claims that **"Calculus is Evil"**, or at least unnecessary ?
- No need to know it to solve differential equations; use massive parallelism instead?
- No need for modern numerical analysis nor difficult mathematical error-analyses*?*

No use nor need for IEEE 754's mostly still unsupported diagnostic aids like …
- "A plethora of NaNs" ?
- Flags inaccessible from most programming languages ?
- Directed roundings (since Unums and SORNs would eliminate roundoff) ?

These claims pander to Ignorance and Wishful Thinking.

Prof. W. Kahan Page 6/26

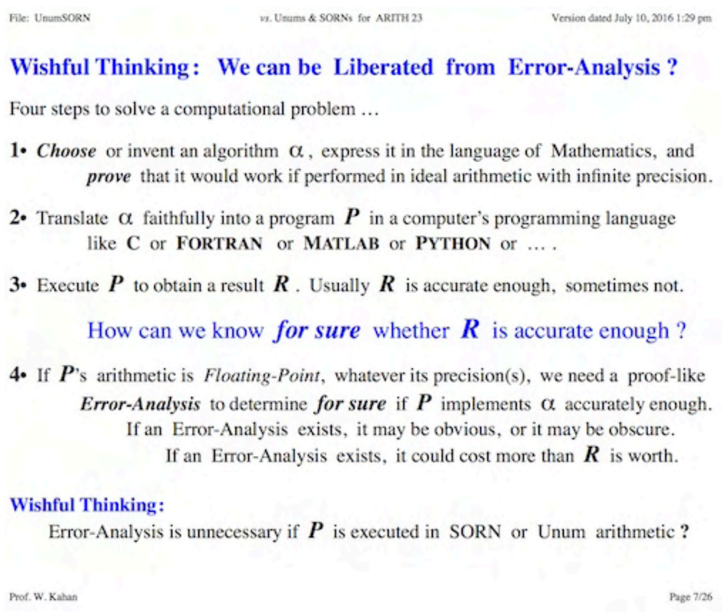
What are the extravagant claims? Well, he says you don't have to have an understanding of difficult mathematical analysis; in particular, the discretization of the continuum, when you solve a differential equation. You don't really have to understand what's involved here, you don't have to worry about underestimating the uncertainty in your data, and you don't have to worry about roundoff. Well, now that's interesting, but is it true. None of that is true. Now, SORN computations, they have a different system. And, unums, and SORNs, are not independent of the dependency problem of interval arithmetic, nor of the wrapping effect, although John does *claim* they are. He claims all the error goes away and that's not quite true. As you'll see.

He claims that calculus is evil, OK, so he's got a funny cartoon. But he really *does* disparage it! He says you don't have to know it to solve differential equations. He says you don’t have to know modern numerical analysis in order to do error analysis. That’s what he says in his book. You can read it for yourself!

And here we’ve got a problem here with diagnostics. “A plethora of NaNs” he says, for instance. We’ve got too many NaNs. He doesn’t know what they’re for, I guess. And, “Flags: Inaccessible from most programming languages and they just annoy people.” Although if they’re inaccessible, and you can’t use them, why would they be annoying?

GUSTAFSON: Because they’re slow.

KAHAN: Well, there’s nothing slow about them. But directed roundings, well, you’re going to have to worry about those because unums and SORNs don’t have directed roundings. There’s the trouble. And that’s what we have to combat. Let’s look at the next page.



File: UnumSORN   vs. Unums & SORNs for ARITH 23   Version dated July 10, 2016 1:29 pm

Wishful Thinking: We can be Liberated from Error-Analysis ?

Four steps to solve a computational problem …

1• *Choose* or invent an algorithm α, express it in the language of Mathematics, and *prove* that it would work if performed in ideal arithmetic with infinite precision.

2• Translate α faithfully into a program P in a computer’s programming language like **C** or **FORTRAN** or **MATLAB** or **PYTHON** or … .

3• Execute P to obtain a result R. Usually R is accurate enough, sometimes not.

How can we know ***for sure*** whether R is accurate enough ?

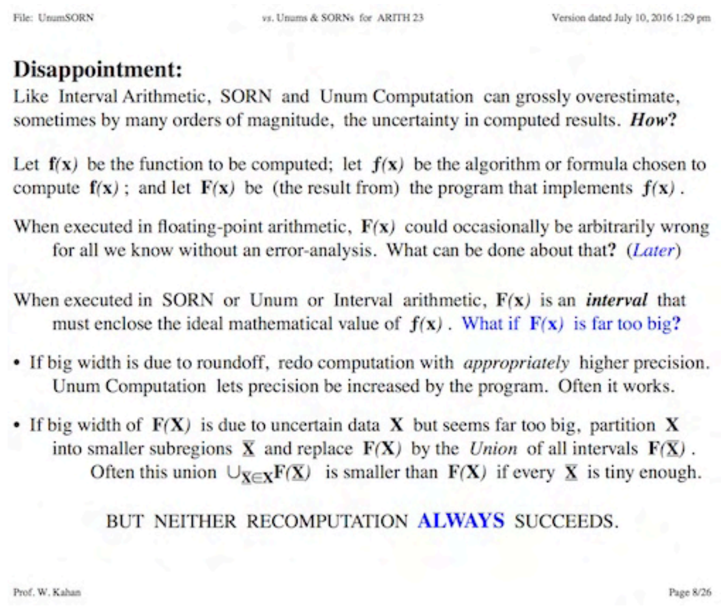
4• If P’s arithmetic is *Floating-Point*, whatever its precision(s), we need a proof-like ***Error-Analysis*** to determine ***for sure*** if P implements α accurately enough.
If an Error-Analysis exists, it may be obvious, or it may be obscure.
If an Error-Analysis exists, it could cost more than R is worth.

Wishful Thinking:
Error-Analysis is unnecessary if P is executed in SORN or Unum arithmetic ?

Prof. W. Kahan   Page 7/26

“Can we be Liberated from Error-Analysis?” Well, in order to understand the issues, we have to look at the process. First you choose an algorithm. You express it in a mathematical language, and you prove that it would work if you did it in ideal arithmetic or arithmetic with unlimited precision. Then you translate it faithfully into a programming, some programming language. Then you execute the program and you get a result. Now, these three things are *different*: The result, *R*, might not be accurate enough. You have to cope with that. And *usually* it is. But, often it isn’t. So there’s this question: If *P*’s arithmetic is floating point but of a precision, do we have an error analysis, to determine for sure, if the program *P* implements the algorithm, accurately enough, and the answer is, maybe you do, and maybe you don’t. And, if you do, it may cost you a fortune! First, to *find* it… you have to pay some PhD guy for quite a while to look for it, and then if he finds it, it might cost you a fortune to compute that error bound.

So that’s why people are unhappy about floating-point arithmetic. And if *your* arithmetic did away with that, wouldn’t that be lovely? But it doesn’t. Next page please.



File: UnumSORN vs. Unums & SORNs for ARITH 23 Version dated July 10, 2016 1:29 pm

Disappointment:

Like Interval Arithmetic, SORN and Unum Computation can grossly overestimate, sometimes by many orders of magnitude, the uncertainty in computed results. ***How?***

Let $\mathbf{f}(\mathbf{x})$ be the function to be computed; let $f(\mathbf{x})$ be the algorithm or formula chosen to compute $\mathbf{f}(\mathbf{x})$; and let $\mathbf{F}(\mathbf{x})$ be (the result from) the program that implements $f(\mathbf{x})$.

When executed in floating-point arithmetic, $\mathbf{F}(\mathbf{x})$ could occasionally be arbitrarily wrong for all we know without an error-analysis. What can be done about that? (*Later*)

When executed in SORN or Unum or Interval arithmetic, $\mathbf{F}(\mathbf{x})$ is an ***interval*** that must enclose the ideal mathematical value of $f(\mathbf{x})$. What if $\mathbf{F}(\mathbf{x})$ is far too big?

- If big width is due to roundoff, redo computation with *appropriately* higher precision. Unum Computation lets precision be increased by the program. Often it works.
- If big width of $\mathbf{F}(\mathbf{X})$ is due to uncertain data $\mathbf{X}$ but seems far too big, partition $\mathbf{X}$ into smaller subregions $\overline{\mathbf{X}}$ and replace $\mathbf{F}(\mathbf{X})$ by the *Union* of all intervals $\mathbf{F}(\overline{\mathbf{X}})$. Often this union $\cup_{\overline{\mathbf{X}}\in\mathbf{X}}\mathbf{F}(\overline{\mathbf{X}})$ is smaller than $\mathbf{F}(\mathbf{X})$ if every $\overline{\mathbf{X}}$ is tiny enough.

BUT NEITHER RECOMPUTATION **ALWAYS** SUCCEEDS.

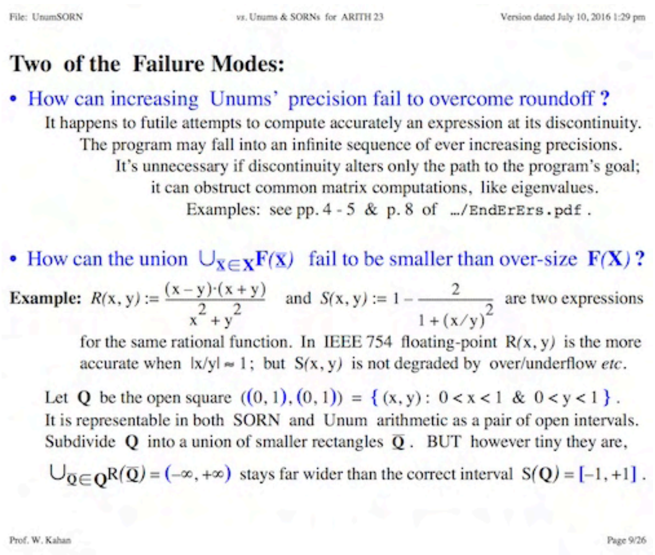
Prof. W. Kahan Page 8/26

We're going to be disappointed. We're going to be disappointed, because, when we affect the discrimination upon the function we want to compute, straight F, and the algorithm we hope to compute it, it could be a formula, curly F, and then we have big F which is the program, they're all… different. When you execute this program in ordinary floating point, it could be arbitrarily wrong sometimes for all we know. But I'm in error analysis, and of course what we'd like to do is something about that. And what John has in mind, is to use unums. But there are other ways to deal with this problem, which I'll get to later. In fact, they were even mentioned at this conference yesterday.

Now what about SORN and unum interval arithmetic? Well, the problem there is that F can be an interval, that although *encloses* the ideal mathematical value, it can be awfully big. And if it is terribly big, too often, then you won't do it. If the big width is due to roundoff, there's something you can do about it; you re-do the computation with appropriately higher precision. Of course that means you have to have interval arithmetic with variable precision. Well, good luck finding *that*!

Unum computation lets the precision be increased by the programmer. And often, it works. If the big width is due to uncertain *data*, now you have a different problem. And what you have to do is break up the data into small subregions, and replace F evaluated once over a block of data by the *union* of all the intervals, F with little x in here. And often the union is smaller. That often works. The trouble is, they don't *always* work. *That's* the issue. If they *always* worked, we could celebrate. But they don't. Next please.

So here are a couple of failure modes. How can increasing the precision of unums fail to overcome roundoff? Well, it can. There are examples; the trouble is, they're so subtle, and I'll explain them only in the question period. It takes too much time, now. What about this "union" issue? How can the union of F as small intervals fail to be smaller than the overside F? Well, here's a simple example. This is a rational function, which has two expressions. This expression and that expression are the


File: UnumSORN   vs. Unums & SORNs for ARITH 23   Version dated July 10, 2016 1:29 pm

Two of the Failure Modes:

- How can increasing Unums' precision fail to overcome roundoff ?

  It happens to futile attempts to compute accurately an expression at its discontinuity.
  The program may fall into an infinite sequence of ever increasing precisions.
  It's unnecessary if discontinuity alters only the path to the program's goal;
  it can obstruct common matrix computations, like eigenvalues.
  Examples: see pp. 4 - 5 & p. 8 of `…/EndErErs.pdf` .

- How can the union $\cup_{\bar{X}\in\mathbf{X}}\mathbf{F}(\bar{X})$ fail to be smaller than over-size $\mathbf{F}(\mathbf{X})$?

Example: $R(x, y) := \frac{(x-y)\cdot(x+y)}{x^2+y^2}$ and $S(x, y) := 1 - \frac{2}{1+(x/y)^2}$ are two expressions for the same rational function. In IEEE 754 floating-point $R(x, y)$ is the more accurate when $|x/y| \approx 1$; but $S(x, y)$ is not degraded by over/underflow *etc*.

Let **Q** be the open square $((0, 1), (0, 1)) = \{(x, y) : 0 < x < 1 \ \& \ 0 < y < 1\}$.
It is representable in both SORN and Unum arithmetic as a pair of open intervals.
Subdivide **Q** into a union of smaller rectangles $\bar{\mathbf{Q}}$. BUT however tiny they are,

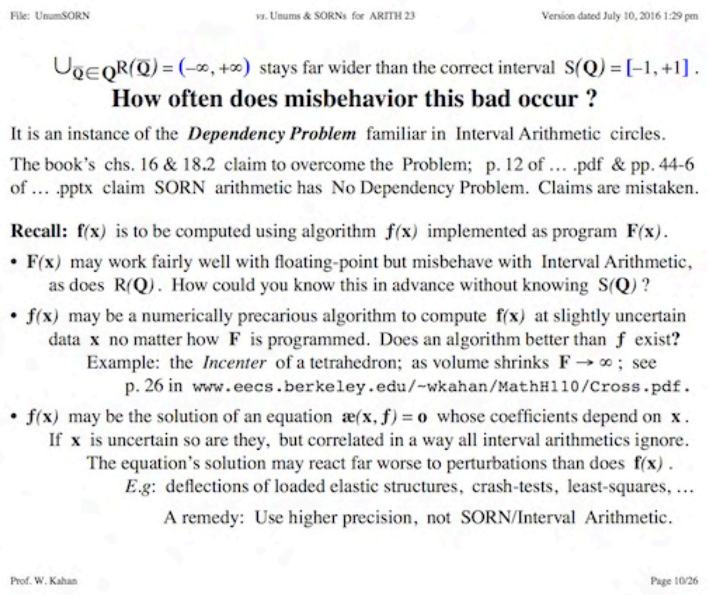
$\cup_{\bar{Q}\in\mathbf{Q}}R(\bar{\mathbf{Q}}) = (-\infty, +\infty)$ stays far wider than the correct interval $S(\mathbf{Q}) = [-1, +1]$.

Prof. W. Kahan   Page 9/26

same rational functions in the field of rational functions. Well, why would you use this one? It's more accurate when x and y have approximately equal magnitudes. Why would you use that one? It's a better choice when x and y are sensitive to over or underflow that might threaten you, so you might you one here and another one there in your program.

Next look at the open square. It turns out, with both the SORN and unum arithmetic, you can represent open intervals. So, this square is representable. And now, if you subdivide it into little bits, no matter how *tiny* they are, you'll always find that the union of the regions with little bits is always this interval. Whereas the correct way to size it except for a misprint; those should be round parentheses, it should be open interval, not closed. That's what this thing reminds me of.

So, the trouble is, that if you don't use the right expression, you may end up with terribly wide intervals. And that can happen to unums, and to intervals, and to SORNs. Next, please.


File: UnumSORN   vs. Unums & SORNs for ARITH 23   Version dated July 10, 2016 1:29 pm

$\cup_{\bar{Q}\in\mathbf{Q}}R(\bar{\mathbf{Q}}) = (-\infty, +\infty)$ stays far wider than the correct interval $S(\mathbf{Q}) = [-1, +1]$.

How often does misbehavior this bad occur ?

It is an instance of the ***Dependency Problem*** familiar in Interval Arithmetic circles.

The book's chs. 16 & 18.2 claim to overcome the Problem; p. 12 of … .pdf & pp. 44-6 of … .pptx claim SORN arithmetic has No Dependency Problem. Claims are mistaken.

Recall: $\mathbf{f}(\mathbf{x})$ is to be computed using algorithm $f(\mathbf{x})$ implemented as program $\mathbf{F}(\mathbf{x})$.

- $\mathbf{F}(\mathbf{x})$ may work fairly well with floating-point but misbehave with Interval Arithmetic, as does $R(\mathbf{Q})$. How could you know this in advance without knowing $S(\mathbf{Q})$?
- $f(\mathbf{x})$ may be a numerically precarious algorithm to compute $\mathbf{f}(\mathbf{x})$ at slightly uncertain data **x** no matter how **F** is programmed. Does an algorithm better than f exist?
  Example: the *Incenter* of a tetrahedron; as volume shrinks $\mathbf{F} \to \infty$; see
  p. 26 in `www.eecs.berkeley.edu/~wkahan/MathH110/Cross.pdf.`
- $f(\mathbf{x})$ may be the solution of an equation $æ(\mathbf{x}, f) = \mathbf{o}$ whose coefficients depend on **x** .
  If **x** is uncertain so are they, but correlated in a way all interval arithmetics ignore.
  The equation's solution may react far worse to perturbations than does $\mathbf{f}(\mathbf{x})$.
  E.g: deflections of loaded elastic structures, crash-tests, least-squares, …

A remedy: Use higher precision, not SORN/Interval Arithmetic.

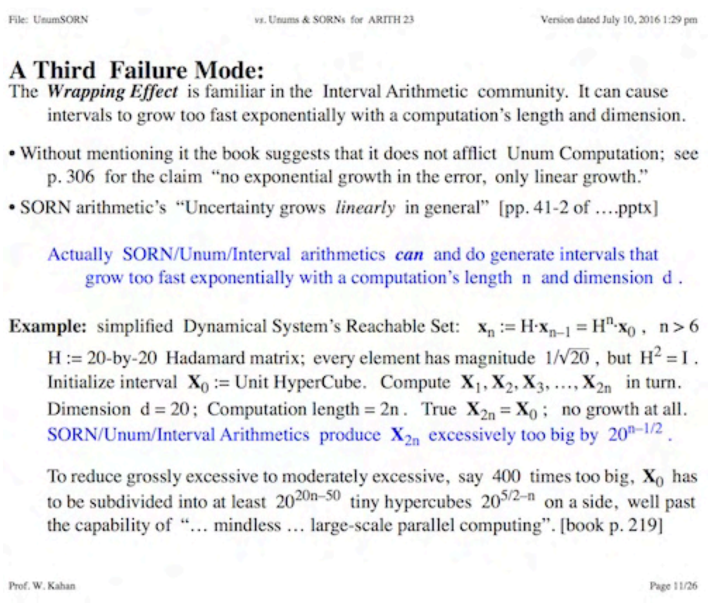
Prof. W. Kahan   Page 10/26

Well, how often does something this bad happen? Well, it's an instance of the dependency problem. The interval arithmetic people know about this. The book's chapters claim that they overcome this problem. And, and the documents about SORNs claim they have no dependency problem. Those are their pages. Well, it turns out that this isn't quite true.

Now remember, you want *straight* F, that's what you want to compute, but you're going to use an algorithm curly f, and you're going to write a program, called big F, and the trouble is that big F may be perfectly happy with floating-point numbers. It just misbehaves when you use intervals. That's what happened, to our— and on the previous page. I don't know how you ever figure this out if you didn't have an alternative way of doing the computation.

But there's *another* theorem or mode. You see, it's possible that the algorithm you chose, although it would, in infinite precision, compute the right function, straight F, the algorithm you chose gave you a really bad way to compute what you want, if you have rounding errors. And that really could no matter how this is programmed, interval arithmetic notwithstanding. And there's an example. The incenter of a tetrahedron is the center of the biggest sphere that fits in the tetrahedron. And no matter *how* you program the function, unums, intervals, whatever you like, as the volume of a tetrahedron shrinks, the value you compute goes off to infinity. And this example is in some of my notes, and it's also in one of the pages put up of the documents that I told you about.

Another place things going wrong is that curly f may be the solution of an equation. Now, this equation may be satisfied by the ideal function, that's what you want, and what you've got is an equation manufactured with curly f, that's your algorithm, solve this equation. The trouble is, that the coefficients could depend on your data. And if your data is a little uncertain, then the coefficients will be uncertain, but they're *correlated*, and they're correlated in a way that all interval arithmetic schemes ignore. So that means the equation solution may react far worse to perturbations than it does to the true function. And this happens often: load on elastic structures, crash tests, least squares; it used to happen to eigenvalue problems until we got better algorithms. So, there's a remedy. Use higher precision. Don't use SORN, or ordinary interval arithmetic. You've got to do something else. Next page, please.



File: UnumSORN vs. Unums & SORNs for ARITH 23 Version dated July 10, 2016 1:29 pm

A Third Failure Mode:
The ***Wrapping Effect*** is familiar in the Interval Arithmetic community. It can cause intervals to grow too fast exponentially with a computation's length and dimension.

- Without mentioning it the book suggests that it does not afflict Unum Computation; see p. 306 for the claim "no exponential growth in the error, only linear growth."
- SORN arithmetic's "Uncertainty grows *linearly* in general" [pp. 41-2 of ….pptx]

Actually SORN/Unum/Interval arithmetics ***can*** and do generate intervals that grow too fast exponentially with a computation's length n and dimension d .

Example: simplified Dynamical System's Reachable Set: $\mathbf{x}_n := H\cdot\mathbf{x}_{n-1} = H^n\cdot\mathbf{x}_0$, $n > 6$

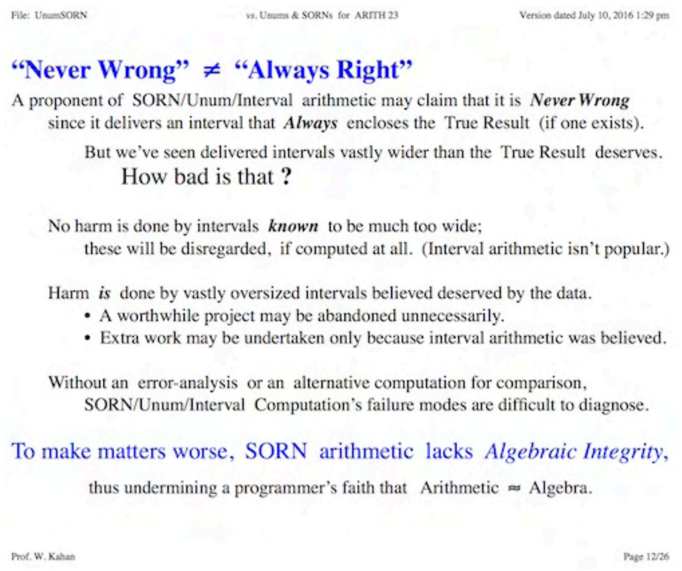
$H :=$ 20-by-20 Hadamard matrix; every element has magnitude $1/\sqrt{20}$, but $H^2 = I$.
Initialize interval $\mathbf{X}_0 :=$ Unit HyperCube. Compute $\mathbf{X}_1, \mathbf{X}_2, \mathbf{X}_3, \dots, \mathbf{X}_{2n}$ in turn.
Dimension $d = 20$; Computation length $= 2n$. True $\mathbf{X}_{2n} = \mathbf{X}_0$; no growth at all.
SORN/Unum/Interval Arithmetics produce $\mathbf{X}_{2n}$ excessively too big by $20^{n-1/2}$.

To reduce grossly excessive to moderately excessive, say 400 times too big, $\mathbf{X}_0$ has to be subdivided into at least 20^{20n-50} tiny hypercubes $20^{5/2-n}$ on a side, well past the capability of "… mindless … large-scale parallel computing". [book p. 219]

Prof. W. Kahan Page 11/26

A third failure mode is called the "wrapping effect." Now, we're calling— the word wrapping effect doesn't appear in the book. And it's suggested it doesn't happen because there's no exponential growth in the error. And SORN is the same thing, on the same page, "Uncertainty grows linearly in general." It ain't true! If you have a long computation, and especially if you have loops, you run a risk of having exponential growth. And here is a simplified example. All I want is, multiply a vector by a matrix. This matrix is a twenty by twenty matrix. You can get it from, well, Matlab, it's a Hadamard matrix. Every element is one over the square root of twenty. But the square of the matrix is the identity matrix. So, when you compute X1, X2, in order, in turn, what you find is, you've got a big dimension that can— the propagation is a fair length. But X2N is X0; there'd be no growth. But in any interval arithmetic scheme that we have nowadays, your computed X2N would be too big by this factor: Twenty to the power n minus a half. And n is supposed to be pretty big.

Well, let's see. What should we do to cut that down? Well, what we do is, we say, let's suppose I'll tolerate an error bound that's 400 times bigger than it ought to be. So I have to break up the initial data, into little bits! This is how big— how many little bits you'd have to have, and that's how big they'd have to be, but when you compute this number, this exceeds the number of parallel processors that anybody has, including the Chinese. [*chuckles*] So it isn't gonna work. "Mindless, large-scale parallel computing," that's the theme of the book. It doesn't—always—work! And that's the problem. It doesn't *always* work. Next page, please.


File: UnumSORN   vs. Unums & SORNs for ARITH 23   Version dated July 10, 2016 1:29 pm

"Never Wrong" ≠ "Always Right"

A proponent of SORN/Unum/Interval arithmetic may claim that it is ***Never Wrong***
since it delivers an interval that ***Always*** encloses the True Result (if one exists).

But we've seen delivered intervals vastly wider than the True Result deserves.

How bad is that **?**

No harm is done by intervals ***known*** to be much too wide;
these will be disregarded, if computed at all. (Interval arithmetic isn't popular.)

Harm *is* done by vastly oversized intervals believed deserved by the data.

- A worthwhile project may be abandoned unnecessarily.
- Extra work may be undertaken only because interval arithmetic was believed.

Without an error-analysis or an alternative computation for comparison,
SORN/Unum/Interval Computation's failure modes are difficult to diagnose.

To make matters worse, SORN arithmetic lacks *Algebraic Integrity*,

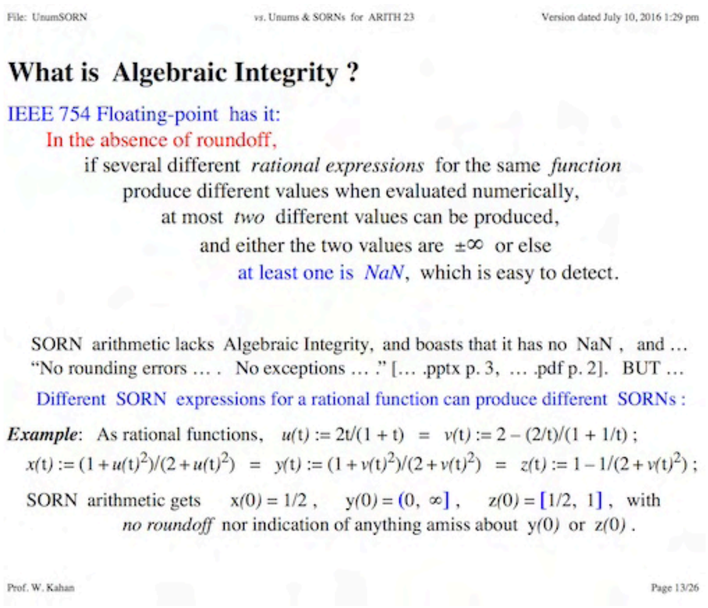
thus undermining a programmer's faith that Arithmetic ≈ Algebra.

Prof. W. Kahan   Page 12/26

In general, the interval arithmetic community, and John, figured out that if your answer's never wrong, then it must be always right. But that isn't quite true. Because you can get intervals that are extremely big, *much* too big… Now, how, how *bad* is that, is that really a bad thing? *No*, not if we know the intervals too big because if we know they're too big you'll disregard them or perhaps not compute them at all. But if you *don't* know they're too big, then you've got a problem. Should you discard a worthwhile project because it appears, that it just doesn't work? Intervals are just enormous? So, you aren't gonna do it? Or, maybe what you want to do is some extra work you better undertake extra work, or maybe because the intervals are too big; you *believe* them. This is the horror that's done by oversize intervals.

Without that error analysis, or, another way of doing the computation, there isn't a way to *diagnose* these kinds of failures. And then just to make matters worse, the proposal for SORN arithmetic lacks something called *algebraic integrity*, which undermines a programmer's faith, that arithmetic is at least an approximation to algebra. Let's see how that is.

Algebraic integrity is at least something that the IEEE Standard has. If there's no rounding errors; if it has several different rational expressions for the same function… then they produce different values, without breaking it! But if they do, you could get at most two different values no matter how many expressions you got, you have plus or minus infinity, that's a possibility, or else you may have at least one NaN, and you can *detect* that. You can detect it because NaNs are obvious, or you can detect it— detect it because the Invalid Operation flag is not raised. But SORN doesn't have that. You see? It says "No rounding errors… No exceptions…" and those are the pages in which you find those boasts.



File: UnumSORN   vs. Unums & SORNs for ARITH 23   Version dated July 10, 2016 1:29 pm

What is Algebraic Integrity ?

IEEE 754 Floating-point has it:
In the absence of roundoff,
if several different *rational expressions* for the same *function*
produce different values when evaluated numerically,
at most *two* different values can be produced,
and either the two values are $\pm\infty$ or else
at least one is *NaN*, which is easy to detect.

SORN arithmetic lacks Algebraic Integrity, and boasts that it has no NaN , and …
"No rounding errors … . No exceptions … ." [… .pptx p. 3, … .pdf p. 2]. BUT …

Different SORN expressions for a rational function can produce different SORNs :

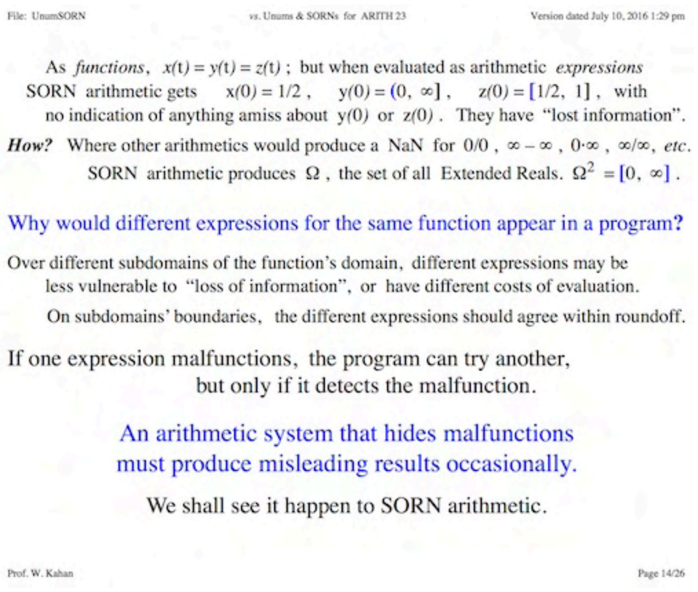
Example: As rational functions, $u(t) := 2t/(1 + t) = v(t) := 2 - (2/t)/(1 + 1/t)$;
$x(t) := (1 + u(t)^2)/(2 + u(t)^2) = y(t) := (1 + v(t)^2)/(2 + v(t)^2) = z(t) := 1 - 1/(2 + v(t)^2)$;

SORN arithmetic gets $x(0) = 1/2$, $y(0) = (0, \infty]$, $z(0) = [1/2, 1]$, with
no roundoff nor indication of anything amiss about $y(0)$ or $z(0)$.

Prof. W. Kahan   Page 13/26

So different SORN expressions for rational functions will produce different SORNs, and there it is, now, u and v happen to be the same rational function… x, y, z, they're the same rational function. As functions in the *field* of rational functions. But SORN arithmetic, that's three different values. And there's no rounding error, and there's no indication that there's anything wrong with these guys. Let's see why that happens.

You see, it happens because: SORN arithmetic doesn't produce a NaN for these exceptional cases. It produces the whole real number set, extended real number set including infinity. And of course if you square that, what you get is all the non-negative numbers including infinity, and now unless you've got an interval that isn't everywhere, you can squish it in any number of ways, and that's what happens for these examples. But, of course you could ask, why would get all these different expressions for the same function? Well, it's because, as numerical analysts, we know that there are occasions where *one* formula… doesn't work very well. It'll work in this domain but not in that domain.



File: UnumSORN   vs. Unums & SORNs for ARITH 23   Version dated July 10, 2016 1:29 pm

As *functions*, $x(t) = y(t) = z(t)$; but when evaluated as arithmetic *expressions*
SORN arithmetic gets $x(0) = 1/2$, $y(0) = (0, \infty]$, $z(0) = [1/2, 1]$, with
no indication of anything amiss about $y(0)$ or $z(0)$. They have "lost information".

How? Where other arithmetics would produce a NaN for $0/0$, $\infty - \infty$, $0{\cdot}\infty$, ∞/∞, *etc.*
SORN arithmetic produces Ω , the set of all Extended Reals. $\Omega^2 = [0, \infty]$.

Why would different expressions for the same function appear in a program?

Over different subdomains of the function's domain, different expressions may be
less vulnerable to "loss of information", or have different costs of evaluation.
On subdomains' boundaries, the different expressions should agree within roundoff.

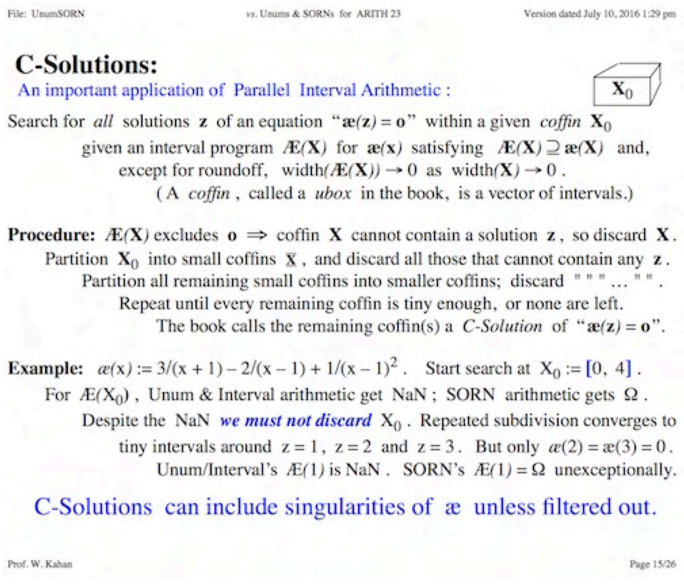
If one expression malfunctions, the program can try another,
but only if it detects the malfunction.

An arithmetic system that hides malfunctions
must produce misleading results occasionally.

We shall see it happen to SORN arithmetic.

Prof. W. Kahan   Page 14/26

And so we develop different formulas for different subdomains, and if need be, if the formula malfunctions, as long as you can detect it, you can decide OK, I guess I should use the other formula which may cost more, that's why I used the first one first.



File: UnumSORN vs. Unums & SORNs for ARITH 23 Version dated July 10, 2016 1:29 pm

C-Solutions:

An important application of Parallel Interval Arithmetic :

X_0

Search for *all* solutions $\mathbf{z}$ of an equation "$\text{æ}(\mathbf{z}) = \mathbf{o}$" within a given *coffin* $\mathbf{X}_0$
given an interval program $Æ(\mathbf{X})$ for $\text{æ}(\mathbf{x})$ satisfying $Æ(\mathbf{X}) \supseteq \text{æ}(\mathbf{X})$ and,
except for roundoff, $\text{width}(Æ(\mathbf{X})) \to 0$ as $\text{width}(\mathbf{X}) \to 0$.
(A *coffin* , called a *ubox* in the book, is a vector of intervals.)

Procedure: $Æ(\mathbf{X})$ excludes $\mathbf{o}$ $\Rightarrow$ coffin $\mathbf{X}$ cannot contain a solution $\mathbf{z}$, so discard $\mathbf{X}$.
Partition $\mathbf{X}_0$ into small coffins $\mathbf{X}$, and discard all those that cannot contain any $\mathbf{z}$.
Partition all remaining small coffins into smaller coffins; discard " " " ... " " .
Repeat until every remaining coffin is tiny enough, or none are left.
The book calls the remaining coffin(s) a *C-Solution* of "$\text{æ}(\mathbf{z}) = \mathbf{o}$".

Example: $\text{æ}(x) := 3/(x+1) - 2/(x-1) + 1/(x-1)^2$. Start search at $X_0 := [0, 4]$.
For $Æ(X_0)$, Unum & Interval arithmetic get NaN ; SORN arithmetic gets Ω .
Despite the NaN ***we must not discard*** X_0 . Repeated subdivision converges to
tiny intervals around $z = 1$, $z = 2$ and $z = 3$. But only $\text{æ}(2) = \text{æ}(3) = 0$.
Unum/Interval's $Æ(1)$ is NaN . SORN's $Æ(1) = \Omega$ unexceptionally.

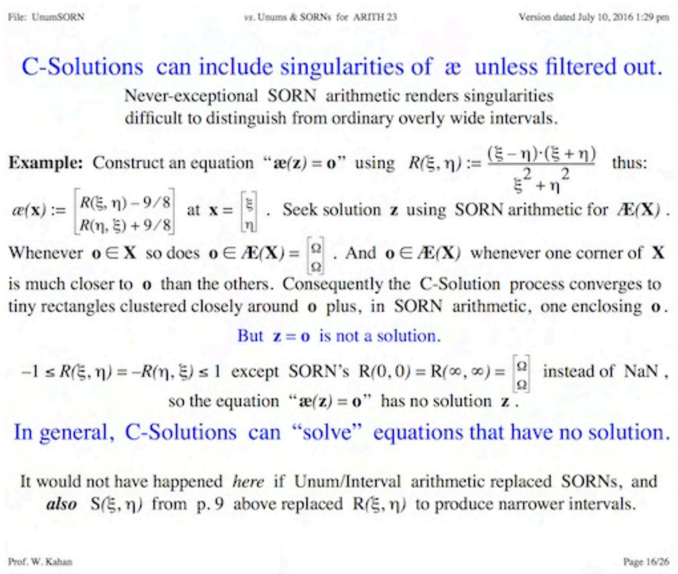
C-Solutions can include singularities of æ unless filtered out.

Prof. W. Kahan Page 15/26

If an arithmetic system hides malfunctions, you're going to get into trouble. And it's going to happen to SORN arithmetic. Let's see how that happens. Well, it has something to do with what are called C-solutions. That's John's name for an important application of parallel interval arithmetic. You see, what we'd like to find are all the solutions of an equation within a given region. I call these boxes "coffins" [*laughter*], it can be called a ubox; or it's just a vector of intervals. But it looks like a *coffin*. And so, given an interval program that has very moderate requirements, you have to be able to write a program that majorizes the true equation, and ideally, the interval version should take the zero as the width of the interval tends to zero, except for rounding errors, so ultimately there's a limit. It's how small you can make it.

Well, here's the procedure, and this is a procedure for interval arithmetic, usable by SORNs, and usable by unums. If, when you compute this expression, at a coffin, if you know that it excludes zero then you know the coffin doesn't contain the solution so you throw that particular interval away. You partition the one you start with into small coffins; you discard all the ones that can't contain the solution, and then you partition all the remaining ones into *smaller* coffins, and do it again, just throw away the ones that can't contain the solution and you repeat it, until it isn't worth further subdivision. And this is— the book calls these C-solutions.

Here's an example. I'd like to find the solutions of this equation. I'll start my search on the interval that goes from zero to four, well, that's a bad choice, because when you compute this, you get a NaN! Or in SORN arithmetic you get everything. So despite that, we mustn't discard this interval. Instead, you subdivide it. You see, you can't say that it *excludes* zero. So you've got to subdivide it. And when you do that, finally the intervals get small enough, you'll find some that exclude zero, you'll throw them away, and you'll converge to these three values of z. Ah, but two and three are the only solutions; z equals one is not a solution of this equation. In unums intervals, you would get a NaN. That would tip you off that there's something wrong. In SORNs, you'd get everything; there's nothing wrong with everything. You wouldn't know that there's something wrong. So that says that C-solutions can include singularities of an equation unless you filter them out. Let me go to the next page, please.



File: UnumSORN   *vs.* Unums & SORNs for ARITH 23   Version dated July 10, 2016 1:29 pm

C-Solutions can include singularities of æ unless filtered out.

Never-exceptional SORN arithmetic renders singularities difficult to distinguish from ordinary overly wide intervals.

Example: Construct an equation "$\mathbf{æ}(\mathbf{z}) = \mathbf{o}$" using $R(\xi, \eta) := \frac{(\xi - \eta)\cdot(\xi + \eta)}{\xi^2 + \eta^2}$ thus:

$æ(\mathbf{x}) := \begin{bmatrix} R(\xi, \eta) - 9/8 \\ R(\eta, \xi) + 9/8 \end{bmatrix}$ at $\mathbf{x} = \begin{bmatrix} \xi \\ \eta \end{bmatrix}$. Seek solution $\mathbf{z}$ using SORN arithmetic for $Æ(\mathbf{X})$.

Whenever $\mathbf{o} \in \mathbf{X}$ so does $\mathbf{o} \in Æ(\mathbf{X}) = \begin{bmatrix} \Omega \\ \Omega \end{bmatrix}$. And $\mathbf{o} \in Æ(\mathbf{X})$ whenever one corner of $\mathbf{X}$ is much closer to $\mathbf{o}$ than the others. Consequently the C-Solution process converges to tiny rectangles clustered closely around $\mathbf{o}$ plus, in SORN arithmetic, one enclosing $\mathbf{o}$.

But $\mathbf{z} = \mathbf{o}$ is not a solution.

$-1 \le R(\xi, \eta) = -R(\eta, \xi) \le 1$ except SORN's $R(0, 0) = R(\infty, \infty) = \begin{bmatrix} \Omega \\ \Omega \end{bmatrix}$ instead of NaN , so the equation "$\mathbf{æ}(\mathbf{z}) = \mathbf{o}$" has no solution $\mathbf{z}$.

In general, C-Solutions can "solve" equations that have no solution.

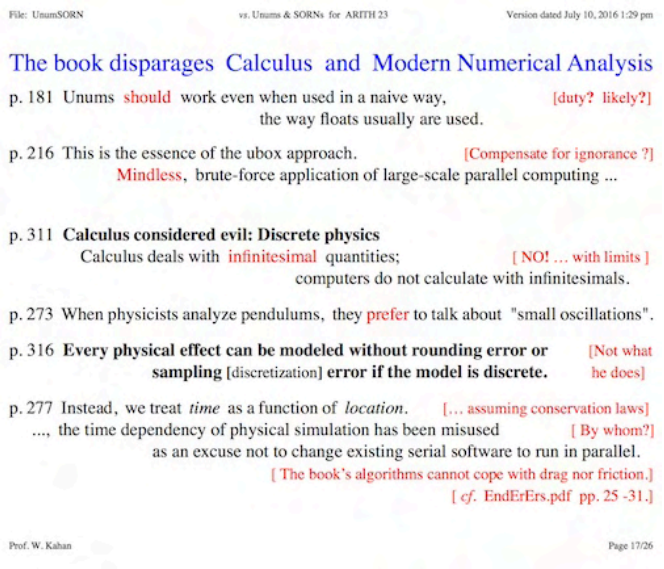
It would not have happened *here* if Unum/Interval arithmetic replaced SORNs, and ***also*** $S(\xi, \eta)$ from p. 9 above replaced $R(\xi, \eta)$ to produce narrower intervals.

Prof. W. Kahan   Page 16/26

Well, the trouble is that if SORNs are never exceptional, it's going to be very difficult to know whether you should filter something out. Here a *better* example. It uses that function R. And it's two equations in two unknowns. And it turns out that whenever zero belongs to a box, it's— it'll be a rectangle, so there's zero… in SORN arithmetic. And zero belongs to a box whenever one corner is a lot closer to zero than the others. So it turns out the C-solution process will converge to *tiny* rectangles clustered around zero, plus in SORN arithmetic you get one that encloses zero. That's not a solution, and you can see it's not a solution by looking at the fact that, "Oh! R can't be any bigger than one, and I want it to be nine-eighths." Can't do that. There *are* no solutions to the equation. C-solutions can solve equations that don't have a solution. And there are other ways for this to happen.

So, what, how much should we have done here instead? Well at this point, if the unum interval arithmetic had replaced SORNs, doesn't matter if it's unums or intervals, and if we'd used S instead of R, we would have gotten rid of bad intervals and then we would have gotten the C-solution process to converge to, no interval at all. But, you choose the wrong formula… too bad. OK, let's go on to the next problem.

Now look: He says he doesn't disparage calculus, well, look, here's what he's saying: They should work when used the naïve way, the way floats are usually used. Well, that's true. Floats are usually used by ignorant people. After all, you can get a degree in computer science without *ever* taking a course in numerical analysis, at least you can in this country. And, here's



File: UnumSORN   *vs.* Unums & SORNs for ARITH 23   Version dated July 10, 2016 1:29 pm

The book disparages Calculus and Modern Numerical Analysis

p. 181 Unums should work even when used in a naive way, the way floats usually are used.   [duty? likely?]

p. 216 This is the essence of the ubox approach.   [Compensate for ignorance ?]
Mindless, brute-force application of large-scale parallel computing ...

p. 311 **Calculus considered evil: Discrete physics**
Calculus deals with infinitesimal quantities;   [NO! … with limits]
computers do not calculate with infinitesimals.

p. 273 When physicists analyze pendulums, they prefer to talk about "small oscillations".

p. 316 **Every physical effect can be modeled without rounding error or sampling** [discretization] **error if the model is discrete.**   [Not what he does]

p. 277 Instead, we treat *time* as a function of *location*.   [… assuming conservation laws]
..., the time dependency of physical simulation has been misused   [By whom?]
as an excuse not to change existing serial software to run in parallel.
[The book's algorithms cannot cope with drag nor friction.]
[*cf.* EndErErs.pdf pp. 25 -31.]

Prof. W. Kahan   Page 17/26

the essence of the ubox approach: *Mindless*, brute-force application. You think that mindless brute-force application will compensate for ignorance. I don't think so. Calculus considered evil, discrete— hey, look! That's the *title* of the chapter! And then he says, calculus deals with infinitesimal quantities. Computers don't. They don't calculate with infinitesimals. It's not true! Calculus does not deal with infinitesimal quantities; it deals with *limits*! And the infinitesimal quantities are just a *shorthand*… for describing what happens with limits. They should— Bishop Berkeley, in the 16th— in the 17th century, had the same problem, he used to worry about the "ghosts of vanished quantities"… but calculus is about *limits*, not infinitesimals. If you write with infinitesimals, you're writing a *shorthand* for certain kinds of limits. We can go into it later if you want.

And then here: When physicists analyze pendulums, they prefer to talk about small oscillations—why do physicists prefer to talk about small oscillations? They do that in order to get a limiting case, to show you what happens with oscillations of small amplitude. We can even estimate the error in the period, if you want to use bigger amplitude, *that's* why grandfather clocks have such long pendulums. Because that way, the period of the pendulum doesn't vary much. And that's the whole point of a grandfather clock.

What about this, every physical effect, now this is his boldface, that he uses the word sampling, where I use discretization error. This is straight out of his book on that page! Well, he doesn't do that! He doesn't solve physical problems with a discrete model. He says he does, but he doesn't. If you want, we can discuss why he doesn't.

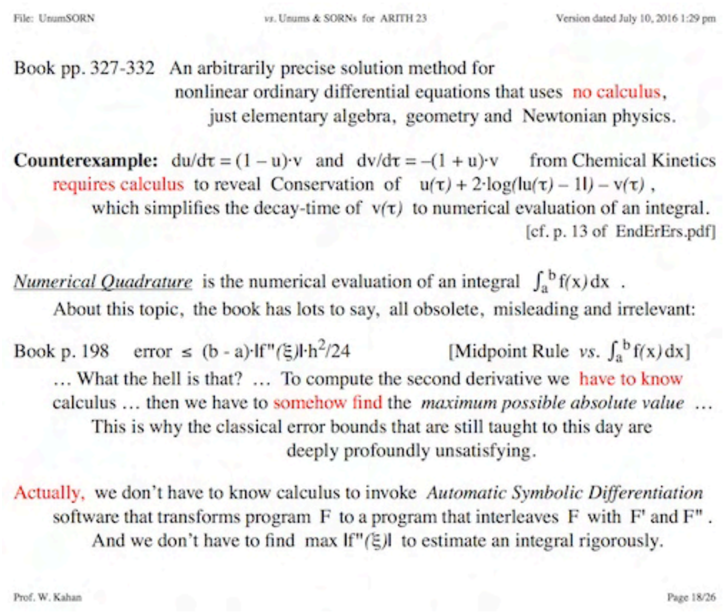
So here's what he really does. He treats time as a function of location. But that works only if you know what is conserved, and there are enough things conserved. But how do you know that energy and momentum are conserved? Well, the total energy, kinetic and potential, didn't exist as compu– as concepts, before the calculus!

And… Well! It's just an unreasonable way of thinking of the time dependency of physical simulations. He's saying, instead of saying the physical properties depend on time, what he's going to do is say, we're going to treat the time as a function of location. That means you have to know where this thing is gonna go. But we don't always know where a system is gonna go. Even the three-body problem, we don't know where it's gonna go. So you can't do that! And in any event, you can't cope with drag or friction. Because with drag or friction, you can't predict in advance where the mechanical system is gonna go. Next page please.

OK, …yes?

DEMMEL: Five minutes.

KAHAN: Well, I will probably finish in five minutes. So hear me. [*chuckling*]



File: UnumSORN   vs. Unums & SORNs for ARITH 23   Version dated July 10, 2016 1:29 pm

Book pp. 327-332   An arbitrarily precise solution method for nonlinear ordinary differential equations that uses no calculus, just elementary algebra, geometry and Newtonian physics.

Counterexample: $du/d\tau = (1-u)\cdot v$ and $dv/d\tau = -(1+u)\cdot v$ from Chemical Kinetics requires calculus to reveal Conservation of $u(\tau) + 2\cdot\log(|u(\tau)-1|) - v(\tau)$, which simplifies the decay-time of $v(\tau)$ to numerical evaluation of an integral. [cf. p. 13 of EndErErs.pdf]

Numerical Quadrature is the numerical evaluation of an integral $\int_a^b f(x)\,dx$.
About this topic, the book has lots to say, all obsolete, misleading and irrelevant:

Book p. 198   error $\le (b-a)\cdot|f''(\xi)|\cdot h^2/24$ [Midpoint Rule *vs.* $\int_a^b f(x)\,dx$]
… What the hell is that? … To compute the second derivative we have to know calculus … then we have to somehow find the *maximum possible absolute value* …
This is why the classical error bounds that are still taught to this day are deeply profoundly unsatisfying.

Actually, we don't have to know calculus to invoke *Automatic Symbolic Differentiation* software that transforms program F to a program that interleaves F with F' and F" .
And we don't have to find max $|f''(\xi)|$ to estimate an integral rigorously.

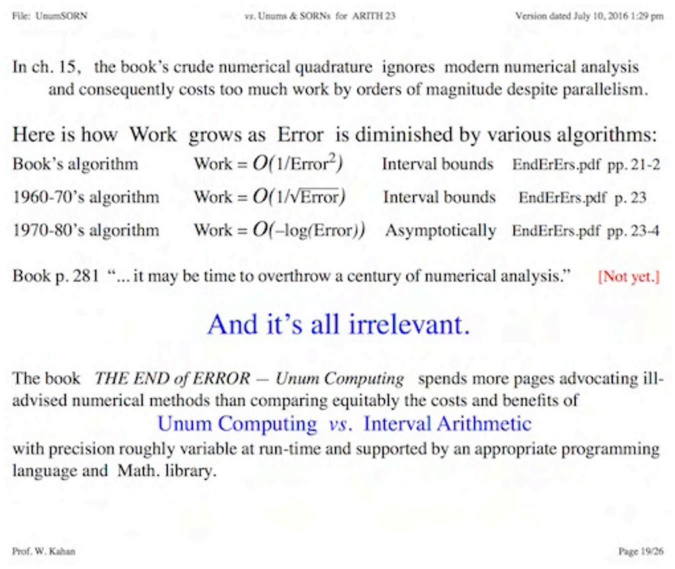
Prof. W. Kahan   Page 18/26

The book says, here, I will offer an arbitrarily precise solution method, no calculus! Just elementary algebra, well that's *wrong*. Here's a counterexample. You see, here's a differential equation. It looks *really easy*, doesn't it, I mean, there's no… higher mathematics here, that, gravity that's conserved, is transcendental. You can't get from here… to there, without the calculus. And if you don't get *there*, then you have a more complicated calculation than you'd like.

Well, you can look at the notes, and you'll see how that works out. Now, you have to evaluate an integral, if you've got, once you've got these conserved quantities, you, what you need is to reduce the solution to the evaluation of an integral, sometimes in closed form, as for, the Keplerian orbits, they're in closed form, but for the swing, you have to actually evaluate an integral, so, let's see what he says about that. He says, "Here is an error estimate." It's an error *bound*, actually. And it's for the midpoint rule but he doesn't tell us that. And this, this is a literal quote from his book! "What the hell is that?"

It turns out, that you *don't* have to know calculus to differentiate, because there are programs that will do it for you! If you write a program to compute a function, you can submit it to another program that will differentiate yours and produce a new program that interleaves the computation of your function, and its first, and if you want it, second, derivative. So you don't have to know how to differentiate, you just have to know that you need one. And you *don't* have to find the maximum value of the second derivative in order to get an error bound. Next page, please.

You see, there are algorithms— now, the book's algorithm, the work is of order one over error squared… for reasonable functions. Now he says I applied a case when, oh, the second derivative was infinite. And I *mentioned* that, because the second



File: UnumSORN vs. Unums & SORNs for ARITH 23 Version dated July 10, 2016 1:29 pm

In ch. 15, the book's crude numerical quadrature ignores modern numerical analysis and consequently costs too much work by orders of magnitude despite parallelism.

Here is how Work grows as Error is diminished by various algorithms:

Book's algorithm	Work = $O(1/\text{Error}^2)$	Interval bounds	EndErErs.pdf pp. 21-2
1960-70's algorithm	Work = $O(1/\sqrt{\text{Error}})$	Interval bounds	EndErErs.pdf p. 23
1970-80's algorithm	Work = $O(-\log(\text{Error}))$	Asymptotically	EndErErs.pdf pp. 23-4

Book p. 281 "... it may be time to overthrow a century of numerical analysis." [Not yet.]

And it's all irrelevant.

The book *THE END of ERROR — Unum Computing* spends more pages advocating ill-advised numerical methods than comparing equitably the costs and benefits of

Unum Computing *vs.* Interval Arithmetic

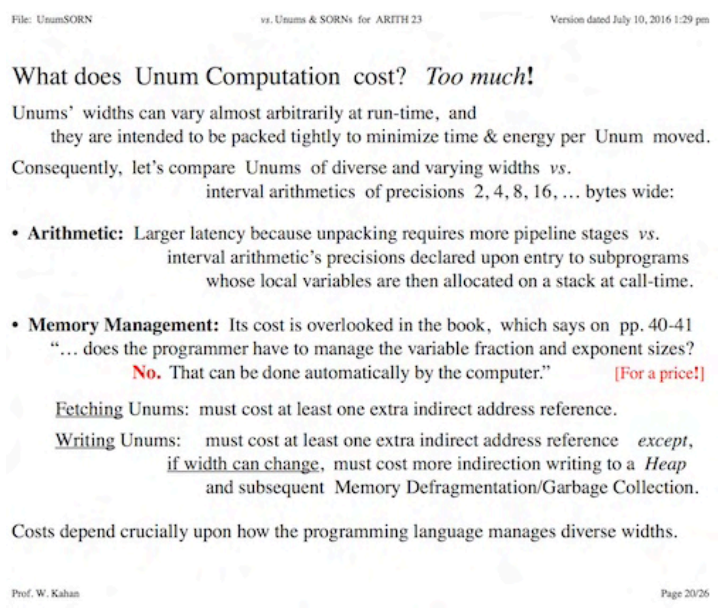
with precision roughly variable at run-time and supported by an appropriate programming language and Math. library.

Prof. W. Kahan Page 19/26

derivative is infinite, instead of getting work to go down like one over the square root of error, it goes down a little bit slower because the second derivative is unbounded. But that doesn't matter; it's still faster than this by an order of magnitude. And in any event, the fastest algorithms they go this way, and this is faster by *many* orders of magnitude than the other stuff. *But*: these schemes give you interval bounds, and that one doesn't. You have to know something about asymptotics. You have to follow estimates, in order to see, asymptotically, how the error is diminishing in order to get any error estimate. But, you get it so much sooner! And so, you now have a tradeoff: Would you like an answer real soon, or would you like it a lot later, but absolutely guaranteed?

Well, sometimes you prefer one to the other, and in any event: It's *not* time to overthrow a century of numerical analysis, not yet, and it's all irrelevant. *All this stuff* about, you don't like calculus, and you should use this numerical method or that, that has *nothing to do* with comparing unum computing with what you *should* compare it with, which is interval arithmetic, but interval arithmetic with precision variable at run time and supported by an appropriate programming language. *That* is the fair comparison. Let's look at that.

What does unum computing cost? It's too much! Because the widths can vary almost arbitrarily. They're intended to be packed together tightly. So let's see what happens. Well, the arithmetic is going to have large latency because the unpacking will require more pipeline stages. Oh, it's not just that we have more transistors; it's the wires! When you have a lot more transistors then you have complicated stuff. You're going to have to lay out wires, that's what's killing you.



File: UnumSORN vs. Unums & SORNs for ARITH 23 Version dated July 10, 2016 1:29 pm

What does Unum Computation cost? *Too much*!

Unums' widths can vary almost arbitrarily at run-time, and they are intended to be packed tightly to minimize time & energy per Unum moved.

Consequently, let's compare Unums of diverse and varying widths *vs.* interval arithmetics of precisions 2, 4, 8, 16, … bytes wide:

- **Arithmetic:** Larger latency because unpacking requires more pipeline stages *vs.* interval arithmetic's precisions declared upon entry to subprograms whose local variables are then allocated on a stack at call-time.

- **Memory Management:** Its cost is overlooked in the book, which says on pp. 40-41 "… does the programmer have to manage the variable fraction and exponent sizes? **No.** That can be done automatically by the computer." [For a price!]

  Fetching Unums: must cost at least one extra indirect address reference.

  Writing Unums: must cost at least one extra indirect address reference *except*, if width can change, must cost more indirection writing to a *Heap* and subsequent Memory Defragmentation/Garbage Collection.

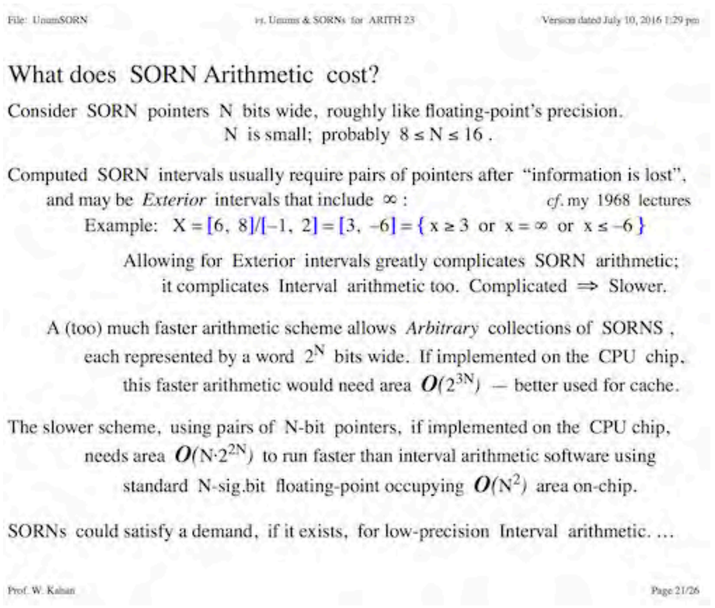
Costs depend crucially upon how the programming language manages diverse widths.

Prof. W. Kahan Page 20/26

Memory management. The book overlooks the cost— it says "does the programmer have to manage it, no, no, no, the computer can do it", yeah, for a price. The *price* comes when you want to write unums; fetching is not too bad. I don't think you're going to want to skip through the unums as a linked list. You would probably prepare a table of addresses in advance, to go with any set of unums you're going to read, if all you're ever gonna ever do is read them. That would speed up the process, and would be tolerable. But if you're gonna write unums, that can change their *width*, oh, that's a *much* more difficult problem, because you can't put the wider unum back where you got it. So that means you've got to use a heap. And if you use a heap, then you're going to have memory defragmentation and garbage collection problems, *oh come on*. You've just ignored the cost of those things, and in any event the cost depends crucially on how your programming language manages diverse widths. Next page, please.

DEMMEL: So, so, you've run out of your thirty minutes, so—

KAHAN: Well, just one more minute, I think I'm almost on the last page. [*laughter*]



File: UnumSORN vs. Unums & SORNs for ARITH 23 Version dated July 10, 2016 1:29 pm

What does SORN Arithmetic cost?

Consider SORN pointers N bits wide, roughly like floating-point's precision.
N is small; probably $8 \le N \le 16$.

Computed SORN intervals usually require pairs of pointers after "information is lost", and may be *Exterior* intervals that include ∞ : *cf.* my 1968 lectures
Example: $X = [6,\ 8]/[-1,\ 2] = [3,\ -6] = \{\ x \ge 3 \text{ or } x = \infty \text{ or } x \le -6\ \}$

Allowing for Exterior intervals greatly complicates SORN arithmetic; it complicates Interval arithmetic too. Complicated $\Rightarrow$ Slower.

A (too) much faster arithmetic scheme allows *Arbitrary* collections of SORNS , each represented by a word 2^N bits wide. If implemented on the CPU chip, this faster arithmetic would need area $O(2^{3N})$ — better used for cache.

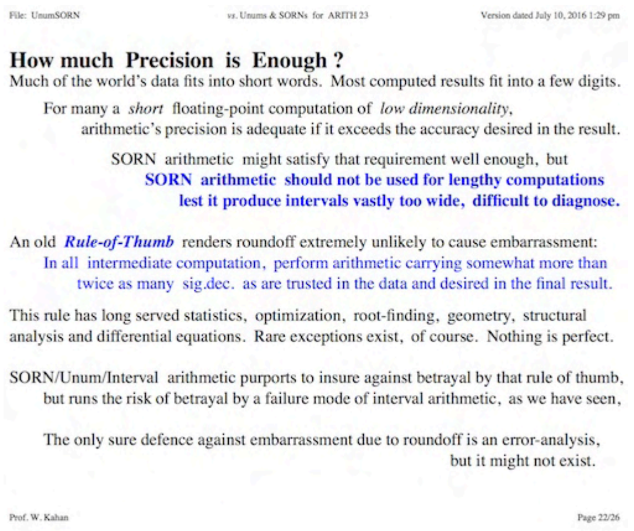
The slower scheme, using pairs of N-bit pointers, if implemented on the CPU chip, needs area $O(N \cdot 2^{2N})$ to run faster than interval arithmetic software using standard N-sig.bit floating-point occupying $O(N^2)$ area on-chip.

SORNs could satisfy a demand, if it exists, for low-precision Interval arithmetic. …

Prof. W. Kahan Page 21/26

What does SORN arithmetic cost, it turns out, SORN arithmetic could work, if you have sufficiently low precision. And it wouldn't be an unreasonable thing. You wouldn't do it in the fastest possible way, but you do it with a slower way. Next page.

How much precision is enough? You see that little rule of thumb? Well, that rule of thumb has worked for a very long time. And of course, it doesn't *always* work. But it works *so often*, that what you have to ask yourself is this: Would you like a result *soon*, or would you like to wait for it and be *absolutely sure*? For many results it isn't *worth* being absolutely sure, because it'll *cost* too much. I mean after all, for computer games, who cares? Suppose you have a transient rounding error problem in your computer game. It'll be gone in a moment.


File: UnumSORN vs. Unums & SORNs for ARITH 23 Version dated July 10, 2016 1:29 pm

How much Precision is Enough ?

Much of the world's data fits into short words. Most computed results fit into a few digits.

For many a *short* floating-point computation of *low dimensionality*, arithmetic's precision is adequate if it exceeds the accuracy desired in the result.

SORN arithmetic might satisfy that requirement well enough, but **SORN arithmetic should not be used for lengthy computations lest it produce intervals vastly too wide, difficult to diagnose.**

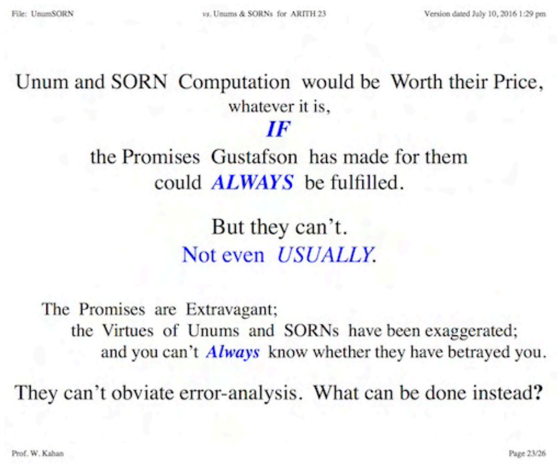
An old ***Rule-of-Thumb*** renders roundoff extremely unlikely to cause embarrassment: In all intermediate computation, perform arithmetic carrying somewhat more than twice as many sig.dec. as are trusted in the data and desired in the final result.

This rule has long served statistics, optimization, root-finding, geometry, structural analysis and differential equations. Rare exceptions exist, of course. Nothing is perfect.

SORN/Unum/Interval arithmetic purports to insure against betrayal by that rule of thumb, but runs the risk of betrayal by a failure mode of interval arithmetic, as we have seen,

The only sure defence against embarrassment due to roundoff is an error-analysis, but it might not exist.

Prof. W. Kahan Page 22/26


File: UnumSORN vs. Unums & SORNs for ARITH 23 Version dated July 10, 2016 1:29 pm

Unum and SORN Computation would be Worth their Price, whatever it is,

IF

the Promises Gustafson has made for them could ***ALWAYS*** be fulfilled.

But they can't.

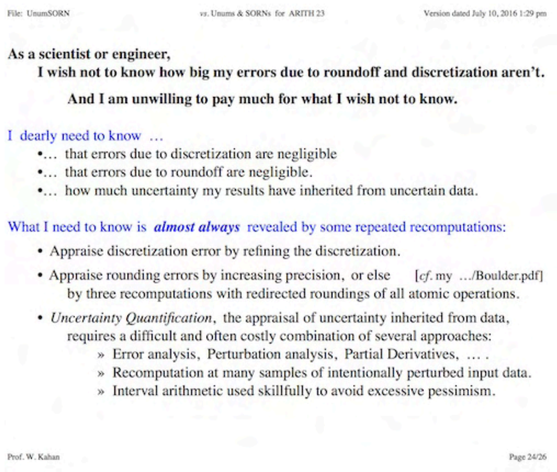
Not even *USUALLY*.

The Promises are Extravagant; the Virtues of Unums and SORNs have been exaggerated; and you can't ***Always*** know whether they have betrayed you.

They can't obviate error-analysis. What can be done instead**?**

Prof. W. Kahan Page 23/26

The only sure way to do deal with it, is, error analysis. And what can be done instead? Well, we'll have to discuss that later. There is something to do.


File: UnumSORN vs. Unums & SORNs for ARITH 23 Version dated July 10, 2016 1:29 pm

As a scientist or engineer,
I wish not to know how big my errors due to roundoff and discretization aren't.

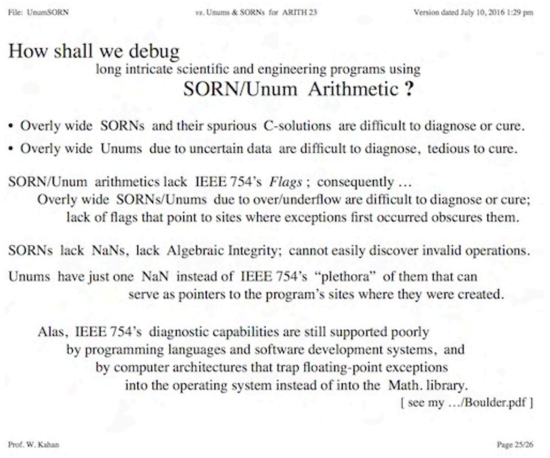
And I am unwilling to pay much for what I wish not to know.

I dearly need to know …
- … that errors due to discretization are negligible
- … that errors due to roundoff are negligible.
- … how much uncertainty my results have inherited from uncertain data.

What I need to know is ***almost always*** revealed by some repeated recomputations:
- Appraise discretization error by refining the discretization.
- Appraise rounding errors by increasing precision, or else [*cf.* my …/Boulder.pdf] by three recomputations with redirected roundings of all atomic operations.
- *Uncertainty Quantification*, the appraisal of uncertainty inherited from data, requires a difficult and often costly combination of several approaches:
  - » Error analysis, Perturbation analysis, Partial Derivatives, … .
  - » Recomputation at many samples of intentionally perturbed input data.
  - » Interval arithmetic used skillfully to avoid excessive pessimism.

Prof. W. Kahan Page 24/26

But *here's* the important thing. There are some things that I don't want to know. And what I don't want to know, I don't want to pay much for. Here's what I *do* want to know, but it turns out that this kind of thing, uncertainty quantification, this is what is *not* handled well by interval, SORN, or unum arithmetic. It's a bleeding sore.


File: UnumSORN vs. Unums & SORNs for ARITH 23 Version dated July 10, 2016 1:29 pm

How shall we debug
long intricate scientific and engineering programs using
SORN/Unum Arithmetic ?

- Overly wide SORNs and their spurious C-solutions are difficult to diagnose or cure.
- Overly wide Unums due to uncertain data are difficult to diagnose, tedious to cure.

SORN/Unum arithmetics lack IEEE 754's *Flags* ; consequently …
Overly wide SORNs/Unums due to over/underflow are difficult to diagnose or cure;
lack of flags that point to sites where exceptions first occurred obscures them.

SORNs lack NaNs, lack Algebraic Integrity; cannot easily discover invalid operations.

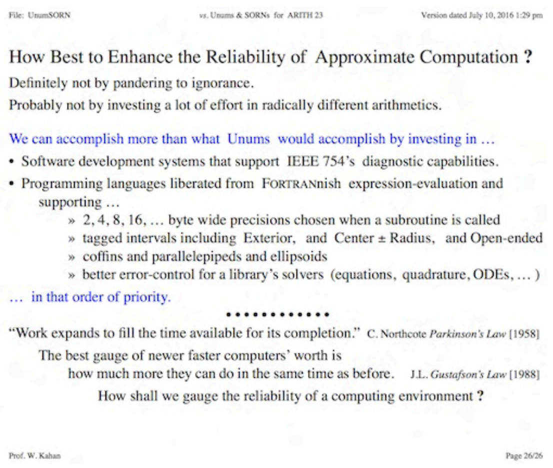
Unums have just one NaN instead of IEEE 754's "plethora" of them that can
serve as pointers to the program's sites where they were created.

Alas, IEEE 754's diagnostic capabilities are still supported poorly
by programming languages and software development systems, and
by computer architectures that trap floating-point exceptions
into the operating system instead of into the Math. library.
[see my …/Boulder.pdf]

Prof. W. Kahan Page 25/26

And then there's debugging. Ah, well. We spend at least three to four times as long debugging as we spend writing our engineering and scientific software. And so that should weigh on our minds. But debugging in unum arithmetic and SORN arithmetic, that is going to be a real headache.

OK, I can quit now.


File: UnumSORN vs. Unums & SORNs for ARITH 23 Version dated July 10, 2016 1:29 pm

How Best to Enhance the Reliability of Approximate Computation ?

Definitely not by pandering to ignorance.

Probably not by investing a lot of effort in radically different arithmetics.

We can accomplish more than what Unums would accomplish by investing in …

- Software development systems that support IEEE 754's diagnostic capabilities.
- Programming languages liberated from FORTRANnish expression-evaluation and supporting …
  - » 2, 4, 8, 16, … byte wide precisions chosen when a subroutine is called
  - » tagged intervals including Exterior, and Center ± Radius, and Open-ended
  - » coffins and parallelepipeds and ellipsoids
  - » better error-control for a library's solvers (equations, quadrature, ODEs, …)

… in that order of priority.

• • • • • • • • • • • •

"Work expands to fill the time available for its completion." C. Northcote *Parkinson's Law* [1958]

The best gauge of newer faster computers' worth is
how much more they can do in the same time as before. J.L. *Gustafson's Law* [1988]

How shall we gauge the reliability of a computing environment ?

Prof. W. Kahan Page 26/26

GUSTAFSON: May I respond, right away, to that last one?

DEMMEL: Let us thank both the speakers. [*applause*]

DEMMEL: Let the debate begin.

GUSTAFSON: So, regarding debugging; if you add information to a number, does that make it harder to debug or easier? Because all I've done is taken floating points and I've augmented them with three fields that are self-descriptive, that assist in finding out what went wrong with the calculation. I'm surprised by this attack. I think you should join me in actually supporting this kind of addition of information to a number so it will actually make it *easier* for people to find out what's wrong with the calculation.

KAHAN: They may found out… that it's *wrong*. But that doesn't really tell them what they want to know: *What's* wrong. [*struggles with microphone switch*]

GUSTAFSON: May I help you?

AUDIENCE MEMBER: Here… isn't it on?

KAHAN: It's got a tiny little switch…

GUSTAFSON: Point it at you.

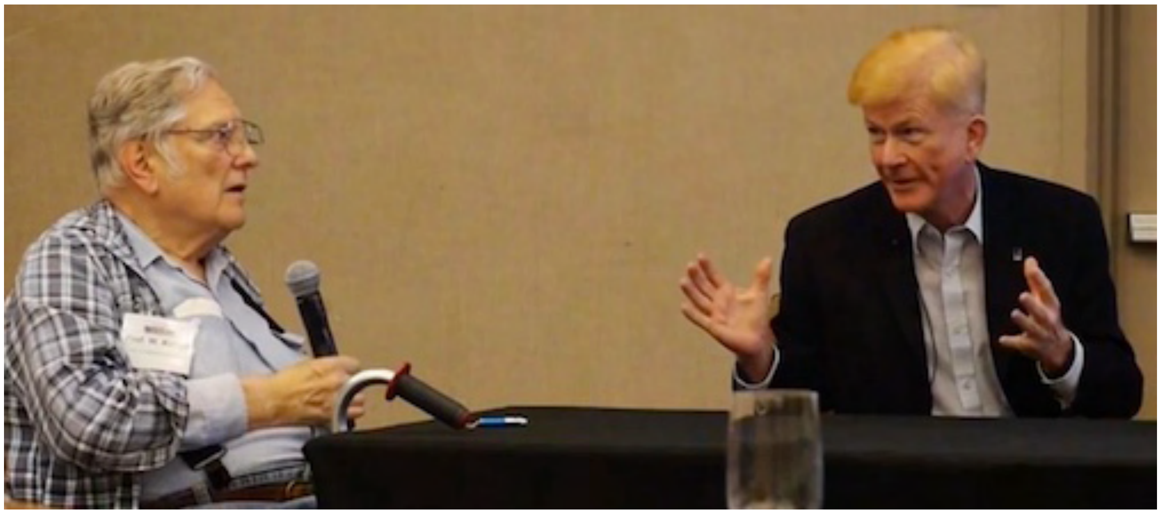
KAHAN: [*more struggling*] OK, I think it's working now. OK, it's one thing to know *something* is wrong. It's another thing to figure out *why* it's wrong.

GUSTAFSON: So does it hurt or help to have extra fields that tell all you about what's going on, with the uncertainty bit, and where it happened, and how big it is and at which bit it was not good… does that make it *harder* to actually use numerical methods and numerical analysis?

KAHAN: Well, what I want to know is *where* this happened, in my program.

GUSTAFSON: So if you have a signaling NaN, it will stop the system, and will go to the operating system where there is a call stack, and you can find out exactly what routine called it, and what instruction, what *line of the code*, in fact you can bring up the source code. *This is where this happened*. So instead of trying to encode numbers, hash pointers to where things went wrong in the five quintillion different kinds of NaN that is supported in 64-bit floats, all you have to do is stop, and let the operating system tell, "This is where your problem is." So I believe what we need is a quiet NaN that says, "Keep on going," or a noisy NaN, a signaling NaN, that will stop it and give you all the debug information that you need. The wish that somehow people are going to start using all these many, many different kinds of NaN to encode debug information… *never happened*. People don't use it. It doesn't exist.

We've waited thirty years for this to happen and it hasn't happened. I don't think it's going to happen.



KAHAN: Well, it has helped me, on some of my old computers. [*chuckles*] It doesn't happen on modern computers because Microsoft won't give you the source code for their operating system so I can't fiddle with it. [*laughter*] But wait; what you're saying about the other will work if you're debugging your own code and it's simple enough. But when you're debugging your code and it's a composite—it's got *your* stuff in it, and it's got *other people's* stuff in it—you *can't* just *stop*. Because, if you just stop, it'll be too late. You won't know what's going on. You'll stop in the middle of somebody else's program! And you don't know where it is. You don't *have* source code. And if you *did*, you wouldn't wanna read it.

GUSTAFSON: So what is your constructive solution to this?

KAHAN: Well, the solution is, to know first of all whose code caused the problem, and secondly, what kind of problem was it. And if the debugger cooperates, you can even tell a guy *where* it happened, when you ask him to fix it.

GUSTAFSON: Sounds great. I do want to comment on some of the other things.

From the very first page that you presented where you say a unum is either an exact number or within one-half ULP, I knew I was going to hear another half hour of complete misunderstandings of every definition in the book. That is *not* what an inexact unum is. An inexact unum is the open set between two exact unums, where those look exactly like floating-point numbers. I can now mark and fill in all the open spaces of the real number line. There's nothing like a plus-or-minus-half rounding that we get. But the habit of thinking in those terms is bending to, "Let's bend this to something I can attack and critique" even though that's not what I said.

"A ubox is the same as a *coffin*." "Coffin" is a classic Kahan dysphemism. "Let's find the most negative word we can use to describe something." If you have interval arithmetic, you do get these great big boxes. They get bigger and bigger; like, five to ten, and then suddenly they just blow up. Pretty soon your bound is minus infinity to infinity. *Uboxes* are *one ULP on a side*, or are *exact*, in whatever number of dimensions they are. They are not, like, from three to four, or whatever, and they split up into smaller boxes. So it's like confusing a box of Legos with the Legos themselves. You can build a lot of things out of Legos. You can build all kinds of shapes. It's not just "a box."

And… the definitions that you saw here, of what is— it's always just a slight misstatement of what I said. "Let's warp it over here, and then make it say something that wasn't said." It wasn't said that, "I can solve *all* ordinary differential equations." I said, "Here's one you can solve, and it's an interesting approach. It works for this *one*." Of *course* there's *tons* of things I can't solve that way. There's *lots* of problems I can't solve. And, I even mention some of them in the book. I did not exaggerate the universality of these approaches the way you're doing here. I said, "It works for this, it works for this… think about the other ways you get the method to work."

KAHAN: Well, the only thing I can do is quote from the book as accurately as I can, and I think I have. I think I've represented your ideas fairly. You're protesting because now you see the consequences of your ideas. You're trying to persuade people that they can do computations, some of them very complicated, without having to understand what's going on inside of the machine, what's going on mathematically. And I claim that that is a false promise.

GUSTAFSON: For all things, yes.

KAHAN: For…?

GUSTAFSON: It won't work on everything. Of course it won't.

KAHAN: That's right. Well, and then, if you don't know when it will work, and when it won't—

GUSTAFSON: Here's the problem. As you've said many times, we have ever more ambitious calculations that we are attempting with ever *less* numerical expertise. We have a million times more computing power now than we had when the IEEE Standard was formed. *What can we do* to improve this situation and make computers a little bit more robust and less error-prone and more self-managing when it comes to numerical calculations?

I think we have now the ability to do things with all those transistors that can actually save bandwidth, which is another thing I want to correct you on. It's a subtle thing, but you talked about memory management being very expensive, it's the wires? The *management* is not expensive. The *management* is inside the CPU. That's cheap. It's *actually moving the data* that costs you. It is trying to reduce the number of bits. The reason I invented unums was to reduce the number of bits going back and forth between the *DRAM* and the *CPU*, to the absolute minimum, because otherwise we're not going to get to exascale. It will actually blow our power budget.

We just heard from the Stanford professor, actually the question-and-answer-session, I said "what about DRAM?" and he said "Oh, that's sixty to seventy percent of all the energy." So if you can cut *that* in half, now you're getting somewhere.

KAHAN: Oh no, no, you— you're ignoring the fact that if the unums vary in width in the course of the computation, you're going to have to put them onto a heap. And if you have to put them onto a heap, that means with a unum, you've gotta fetch an address, also.

GUSTAFSON: Right. So you will go between *fixed-size storage* unums, and occasionally we will wrap them all up and send them off to disk or send them off to DRAM, but we will do it intelligently as needed, but most of the time they will probably exist in unpacked form.

KAHAN: Well the example in your book with the Fast Fourier Transform didn't do that.

GUSTAFSON: The Fast Fourier Transform actually can be arranged so you are almost always accessing linear-ordered, contiguous groups. It is possible to do an in-place Fourier Transform that keeps on rearranging the data, always packed. But that said, I would certainly unpack, probably, an FFT to make the indexing simple. It would use less energy and do a better job of bounding the answer with fewer bits.

KAHAN: So that means somebody can't use this thing mindlessly, can they?

GUSTAFSON: Not completely mindlessly, no. I did make *tools*, as which— certain things actually can be managed with less effort to find what's going wrong. Because, as I said, I'm *adding* information to a number; I'm not taking it away. So, if your objection is that, I used the word "mindlessly," and of course you can't completely use a computer mindlessly—we wish!—you have to use some amount of thought. But does *everybody* have to know numerical analysis? Does everyone have to know calculus? You know, only about *eight percent* of the people in the world know calculus? There have been studies to try to estimate what is the overall numerical literacy

of the human race, and if only eight percent of the people can do numerical methods on computers, uh, we're in trouble.

KAHAN: Are ninety-eight percent of the population going to write numerical programs? [*laughter*] Look…

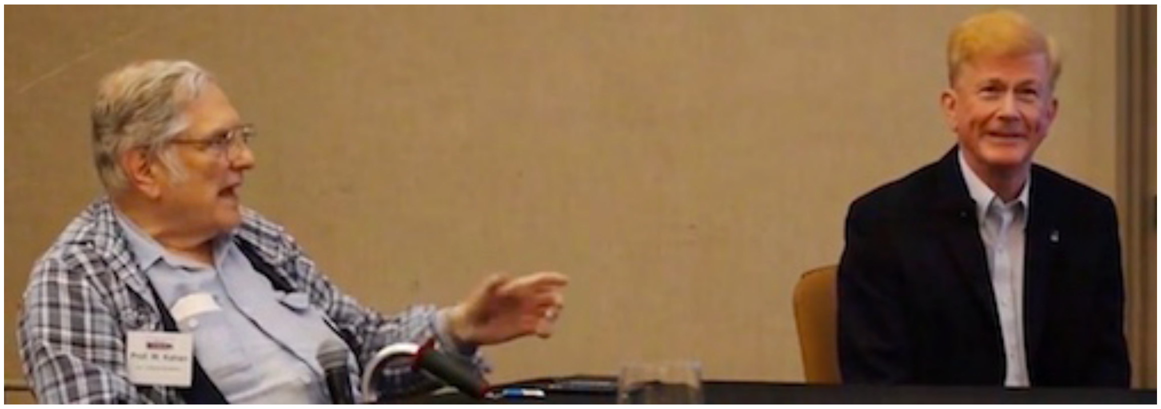
GUSTAFSON: There are lots of people using spreadsheets who don't know what they're doing.

KAHAN: I understand the desire to make it easier for people to use computers… to make it possible for them to do more, and know less. And there's this little box here, the HP-15c which is exactly that.

Yes, you can evaluate an integral. You can solve an equation. You can do complex variables… on this tiny calculator. So I'm in favor, of making things easier. But I'm not in favor of telling people that *you don't have to know anything*. You have to know something, and it's difficult to say what you have to know, but unless people are willing to learn a little bit about numerical analysis, they are very likely to harm themselves, or, you. We all are going to depend upon software that we get through the web. And if any idiot can write the software, and then send it out and say, "Hey, I'm using unums, so it should be OK," well…

GUSTAFSON: Either OK or it will say, "Look, you just had a big error! You'd better figure this out," whereas with floats, it won't tell you a thing. At least you get the warning with unums. You know something went wrong.

KAHAN: But there is a way, with floats, we saw a discussion about yesterday, when someone described Monte Carlo rounding. Now, he's got the right idea, except a wrong implementation. Let's see if you can *simplify* the implementation a lot, and reduce the price, a lot, and *still* get the benefit, which says, with *very* high probability, you'll discover whether your results are contaminated excessively with high roundoff. And, when it comes to uncertain data, that's the same sort of thing, that if you re-curve the data and run a lot of experiments, you can find that out. But with unum computation, you can conclude, because you say, “Well, I put in my data as unums, each one is uncertain by a half a unit in the last place, because that’s as many bits as you have per unum [*laughter*] work, how uncertain the result is, because of the uncertainty of data, well that’s *very* optimistic.



DEMMEL: So I would like to just briefly ask, are there any questions from the audience? We can let the debate go on, but you are, you know, close to six-thirty, and I just wanted to make sure that—

AUDIENCE MEMBER: Well, I have a question for you, Dr. Demmel. A lot of people use libraries, such as LAPACK, that I know you’re expert in, or statistical packages like R that may depend on large libraries underneath. What’s gonna—what do you think might happen in a world with a different number representation with error encoded in it? Running through these large libraries, which you’ve done, a lot of error analysis yourself.

DEMMEL: So this is one question that you already have addressed and I was going to ask that, but let me ask you: So imagine I want to write a matrix algorithm. And I refer to submatrices all the time, and ask for them in other routines, and so if all of the objects are variable size, referring to a submatrix suddenly becomes a complicated kind of thing. Would you say that’s something to be unpacked?

GUSTAFSON: Absolutely you would unpack that. And as long as we're talking about linear algebra, the biggest benefit of the unum definition and its environment, is the support of things like fused dot product. So you don't do any kind of rounding or approximation until you've done a complete dot product. This is the idea from Ulrich Kulisch that he unfortunately *patented*, which effectively prevented the world from getting to it for however long the patent lasted, and now the patent has expired so I think it deserves some very serious consideration again. Because this is a very, very good way to, at a possibly *slight* cost of time, get a much more reliable answer, that will obey associativity for example, as long as you're doing the calculation exactly, in the scratchpad. So the quality of linear algebra could improve vastly, simply by using exact dot products throughout the kernels, and not making *that* rounding error.

DEMMEL: So, so I just wanted to point out that our philosophy in building LAPACK has always been that if we can do something that will in fact keep the error down, and, but the number of people who actually use those error bounds is incredibly tiny, compared to the number of people who type A back-slash b.

GUSTAFSON: Yes.

DEMMEL: And so that's why I'm just, and so—we're not going to give up that philosophy, but you know, it's— it's had not as much impact as we would have liked. And, so the question comes up in, you know, all the approaches we're talking about here too, is that, is it worth the cost.

KAHAN: I think you've used the word "unpacking" in a glib way. Because, you can unpack if all you're ever going to do is read. But if you're going to *write* unums, you have to allow that unums may vary in length. And if that's the case, you've got to unpack them, and then you've got something wider, where are you going to *put* it?

GUSTAFSON: You're talking about the management of a heap. You've taken something out of the heap, and now it's gotten bigger, so you can't put it back, that's what you're saying.

KAHAN: You can't put it back where it was, that's right.

GUSTAFSON: You're not understanding what "unpacking" is. An unpacked is at the *maximum size*. So you allow the maximum size that a unum can be. Which still might be a very usable number of bits and a very large exponent range. The extra bits that you get can be used to mark whether something is a NaN or infinity such that a single-bit test actually is faster than a float.

KAHAN: So you've gotta make provision for what may be the largest width that will turn up in a computation.

GUSTAFSON: When it's on the CPU, it probably will pay to do that.

DEMMEL: So, so let me just continue that line… if you want to implement Gaussian elimination? And you want to use blocking, which is what the standard optimization—

KAHAN: Is your microphone on? We can't hear you.

DEMMEL: OK, let me try one more time. So, let's ask how we would implement Gaussian elimination where we want to use the standard optimization to get it to run at decent speed, which is blocking, so it seems like we'd have to—so we're going to be writing to that matrix all the time, because that's how Gaussian elimination works, so it seems like the algorithm would have to run in sort of standard arithmetic, basically, because you're going to reading and writing the matrix over and over again, and so, you're effectively running in standard floating point. Maybe with your representation, but the semantics would be the same.

GUSTAFSON: It would be similar. I think I'd find a way to do it up to a certain point where as soon as the matrix gets so large, that you discover you have to store it, say, on disk, then it's worth paying the price to pack and unpack these things. It's always an option. You don't *have* to do it. But right now, memory is two hundred times slower. It takes two hundred times as long to fetch an operand from main DRAM as it does to do a multiply-add on it. You can get an awful lot of management-type things done in that time, and figure out what's going to go where. In your gut you said, "Oh, that's going to be really expensive!" Spec it out! Actually do the calculation of how long these things take. It's not something that you can just, from an armchair, say, "Oh, that's going to be really expensive." You'd be very surprised at what the design rules of 2016 say you should do, as expensive versus not expensive. Moving data between DRAM and a CPU is a choke. You want to do as little of that as possible, and you want every bit to count. That's what I'm trying to accomplish. So any time you can—

DEMMEL: So, I would hope that, maybe that a future challenge is to write optimized block Gaussian elimination using your arithmetic and see what the overhead might be.

GUSTAFSON: Sure.

KAHAN: Well, there's a second problem. When you minimize communications costs, you end up using algorithms which, in some sense, have to anticipate what is going on. In that case, you may require rather more precision than you thought, in order to survive. And I give a citation to that on one of the documents on my web page, where a graduate student worked it out and found, "My goodness! You may have to use double precision or more, when actually your data is in floats! Or, smaller!" This means it's hard for you to predict how much precision will be needed, if you're using an algorithm that economizes very greatly on memory movement. The algorithms that economize on memory movement are sufficiently strange, that some of them require extraordinary precision in order to survive.

DEMMEL: So I guess we're talking about solving least-squared normal equations, just to be—

KAHAN: Exactly.

DEMMEL: —properly, because you want to compute a few x transpose x and help, you sort of have to do that ahead of time.

And there's a question?

AUDIENCE MEMBER: Yes, I have a question about the cost of performing with your basic set unum. Suppose you had an iterative big solver, where you multiply a matrix. You have this nice size, let's say, 128 by 128. Chances are, every time, if you impose a limit on the number of bits you represent, every time you do an FMA, practically, you well end up rounding. But with unums you're going to have imprecise bit set. Right? OK. Now for matrix multiplication I have to do, in this case 128 FMAs, one after the other, and chances are very high, every step, I'm going to have an imprecise value. So if I understand correctly, after computing one element of my matrix I'm going to end up with 128 boxes.

GUSTAFSON: You're talking about the dot product?

AUDIENCE MEMBER: —representation but the point I'm making here—

GUSTAFSON: This is the dot product you're talking about? You're doing matrix-vector multiply?

AUDIENCE MEMBER: You know, matrix multiplication, where you take two vectors, you know, to get one result, right?

GUSTAFSON: There is one rounding error at the very end. Or, it wouldn't be called a rounding error. It's done as if it was an integer multiplication.

AUDIENCE MEMBER: My main question is, if you have an imprecise result every time you do a plus, how many units of storage do you need, to store the result?

GUSTAFSON: My answer is you do *not* do it in an inexact after every plus. You do the entire vector; you do a dot product all in *exact* arithmetic. Then—

AUDIENCE MEMBER: So you are proposing implementing an arithmetic unit that takes together 128 values—

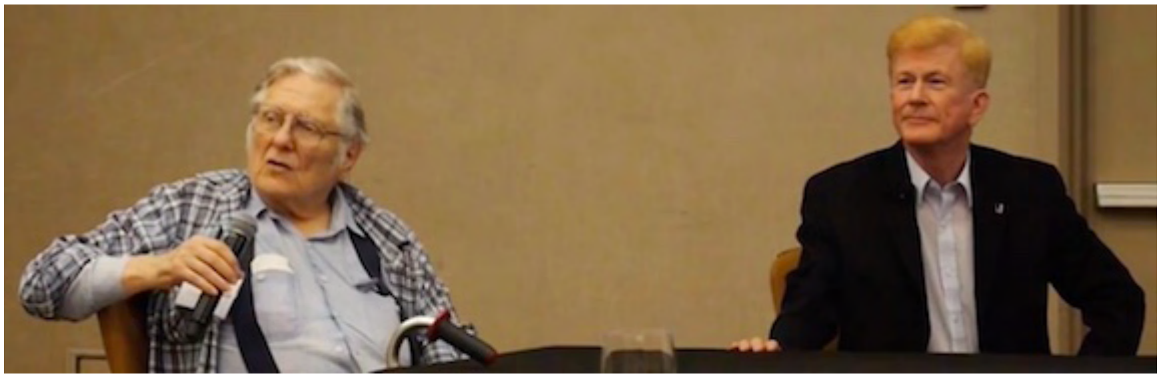
GUSTAFSON: It's called the Exact Accumulator.

AUDIENCE MEMBER: And that's, exact—

KAHAN: Look! You *don't* use—

AUDIENCE MEMBER: Because if you use 1024 matrices, I should beef up my—

KAHAN: OK, look. *Most* of the time, with *extremely* rare exceptions, you can do a scalar product, accumulating the products with ever manys if you want, or you can accumulate them in variables that have twice the width of the operands. So you have A times B, say A and B are floats, A times B, you compute that in double and add it into a double sum. And *almost always*, what you'll get will be perfectly satisfactory.



GUSTAFSON: "Almost."

KAHAN: And that's one of the reasons why Kulisch's scheme, which would evaluate it *exactly*, isn't very popular, is because it hardly adds anything to the value. Only in extremely special circumstances. So for your purposes, in graphics, I think that's what you have in mind, right?

AUDIENCE MEMBER: Not, not really. we have GEMMs, all over the place—

KAHAN: I can't hear you.

AUDIENCE MEMBER: GEMM? Right? Matrix Multiply? I need matrix multiplication, and my big question is, "how much memory do I need to store my result, under random circumstances?"

KAHAN: Well, well, whatever it is—

AUDIENCE MEMBER: If I use unums. I understand what works for floating point and how to deal with error. I'm just curious for unum, how much memory do I need to store my results, or how much price I need to pay for the exact implementation?

GUSTAFSON: Are you afraid that somehow it's going to become a gigantic number? Is that your fear? Because you set the environment; you say, "I want a maximum of this many bits of exponent—

AUDIENCE MEMBER: So it's a *really big* problem I'm facing. Basically I understand the benefits. I do not understand the cost.

GUSTAFSON: And I'm telling you that—

AUDIENCE MEMBER: What I heard today, it's either going to be an *extremely*, *extremely* costly arithmetic unit, or it's going to be an extreme cost because of storage. But if I use unums. I mean if it's my—

GUSTAFSON: I don't think you understand. It's not unlike saying: "I'm going to do this in double precision." You know the result's going to be 64 bits. I'm going to say, "This is the biggest exponent I can have; this is the biggest mantissa I can have; that's the size you can get to." It's not gonna go beyond that, without your control. You can set— you can try that, and see if it works. And if it doesn't work—

AUDIENCE MEMBER: Let me make this simpler. Suppose you have value A, and value B. And they are both inexact, right? Which means, the only thing to, you have to worry about, whatever the Unit in the Last Place you get with these unums, I add them. A plus B. All of a sudden instead of, let's say, one *inexact* bit at the end, one unum bit at the end, I'm going to have *two* unum bits.

GUSTAFSON: What?

DEMMEL: What width have the intervals. When you have two intervals. Width—

AUDIENCE MEMBER: So I'm adding two numbers that are not exact, I cannot store the result using only one unum.

GUSTAFSON: Sometimes, you can.

AUDIENCE MEMBER: I have— Sometimes, but if you don't know the update, no, you cannot. So I will have, for simple addition, two inexact numbers I have to store; I have to double my storage!

GUSTAFSON: Yeah.

AUDIENCE MEMBER: Now this is *one step*—

GUSTAFSON: You really need to read the book. It's—

AUDIENCE MEMBER: And we're talking about the very same matrix multiplication, which will end up having an N cubed cost in terms of storage.

GUSTAFSON: No.

AUDIENCE MEMBER: On the average case. That seems a little bit extreme.

GUSTAFSON: That's not the way they work. As I said—

AUDIENCE MEMBER: And my question is then, how?

GUSTAFSON: You're getting into something where you'd just about have to read two or three chapters of the book to understand—

AUDIENCE MEMBER: Can, can you explain in a few word why am I wrong.

GUSTAFSON: Well, first of all, if you have— let's say your numbers are— where you would store floats, you have replaced them with unums. So you have inexact quantities, here. You—

AUDIENCE MEMBER: It's not about replacing with unums. It's about how unums *work*. I don't think you explained that. You made a claim about how *wonderful* it would be—

GUSTAFSON: In general, a unum is not a closed—

AUDIENCE MEMBER: —to have a world in which things work magically, but I do not understand the magic. That's my problem right now.

KAHAN: I think interval arithmetic was designed to help you with your problem. Namely, each of your arguments is uncertain, and if you use, not the conventional two real number, brackets, for intervals, but you use the center and radius scheme for intervals, then you'll find that no matter how wide your operands are, if the uncertainties are small, you will *not* have to worry about a great deal of storage, or the accumulation of uncertainty.

But: you will have to worry about the fact that if your uncertainties are *correlated*, the correlation will not be taken into account, and you may end up with a very pessimistic estimate. But otherwise than that, interval arithmetic was designed to cope with your problem. I *think* I understand it. And there isn't a simpler way to deal with it. If you knew in advance that none of your arguments was more uncertain than, say, one part in a million, then it would be possible to say something in advance about the inherent uncertainty of the scalar product. But it would be very pessimistic.

DEMMEL: So let me just say, we've been reminded that dinner is at seven, [*chuckles*] and so I would like to invite both of our speakers to make some closing remarks. So maybe, one minute of closing remarks each. And then of course, the debate can continue at dinner, if you desire. [*laughter*]

GUSTAFSON: As I said, we have much more power than we used to have. Maybe I didn't get it right, not on all counts, but I tried to convince people— get people to think about the fact that we need to use the vast advances in technology to improve the quality of computer arithmetic, and we have the opportunity to do so. I would *very* much like to do so with Dr. Kahan's help, not as an adversary. He has been a man I've admired all my life. I've been doing numerical work since I was 15. I've been making use of his calculators since I was in— probably a freshman in college. I've learned *so much* from this man. I don't want to— I'm not happy being on the other side of the fence from him, because I think we *do* want to accomplish many of the same things.

He may argue with my *style* a little bit, or a lot, but I think we really do want to see the same thing come out of numerical analysis.

KAHAN: OK. My position is that we should not try to oversell the schemes. We should try to be conservative in what we offer, because people will depend on it, and they depend on things that *can* fail from time to time, then it's necessary for us to estimate: What is the *insurance premium* that should be paid, in case the system fails? So, if a system isn't foolproof, what is the probability of failure, how much will the failure cost, and when you put that all together, you add that up over all the possible failure modes, and that gives us an idea of how much an insurance premium should cost.

There's the choice. Either the system has to be *foolproof*, or, if you *say* that it's foolproof, but it isn't, then I would like to know what insurance premium will Lloyd's of London charge you? Or, you can tell people, "If you want the answer in a hurry, you're going to have to take a certain risk. And we're *accustomed* to that in life. There are some situations in life where, you have to accept certain risks in order to get things done in a reasonable time. In numerical computation, there's a lot of that. And it doesn't mean that we're just *guessing*. It means that we try to strike a balance between getting something done in a reasonable length of time and having reasonable confidence.

I don't think that the world of computation is *quite* as uncertain as John says it is, but then, since I'm a numerical analyst, I would see things differently. I do agree, however, that there *are* people who will write computer programs who are *completely* innocent of any exposure to the hazards of floating point. I agree that that is going to happen. And unfortunately, it is going to happen no matter what John and I say or do.

[*Applause*]

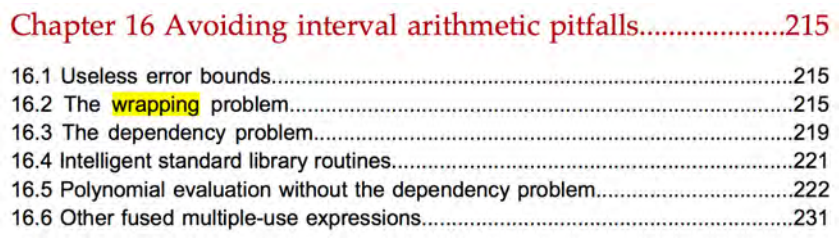
DEMMEL: So, hopefully this video will be made available to all of you so you can review all the claims and counterclaims, and read all the documents. ■

Afterword

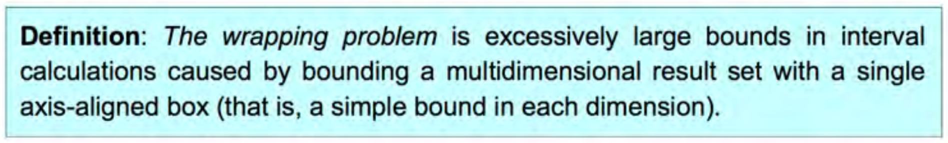
Perhaps the strongest evidence that Kahan did not actually read the book *The End of Error: Unum Computing* that he is so critical of is this part of his position statement (page 23 of this document):

> “A third failure mode is called the ‘wrapping effect.’ Now, we’re calling— the word wrapping effect doesn’t appear in the book.”

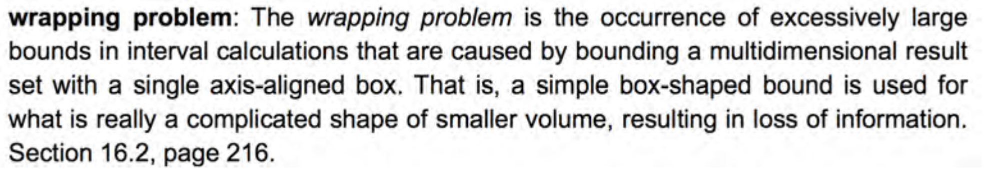
This is untrue. Here it is in the Table of Contents:



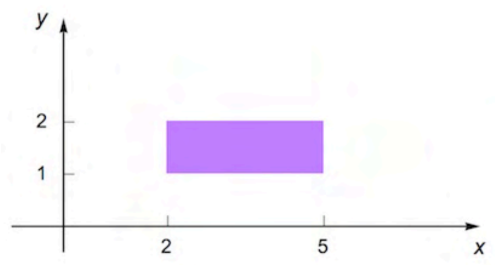
To make sure the reader does not miss it, it is clearly defined in a bright blue definition box:


Definition: *The wrapping problem* is excessively large bounds in interval calculations caused by bounding a multidimensional result set with a single axis-aligned box (that is, a simple bound in each dimension).

It is discussed thereafter in six different places in the book, with examples and an exercise for the reader showing the limitations of the ubox method. It also appears In the Glossary:

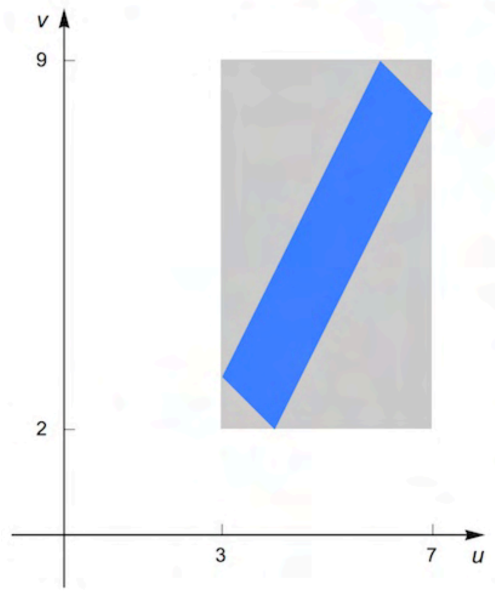

wrapping problem: The *wrapping problem* is the occurrence of excessively large bounds in interval calculations that are caused by bounding a multidimensional result set with a single axis-aligned box. That is, a simple box-shaped bound is used for what is really a complicated shape of smaller volume, resulting in loss of information. Section 16.2, page 216.

The diagrams of the wrapping effect fill entire pages and are similar to ones Kahan himself uses in his “coffin” discussions:



The interval result hides the fact that a great deal of information has been lost, using "information" as defined in the previous chapter. Graphically, we start with the rectangular bound shown in purple in the plot on the left.

The true range of values of *u* and *v* are shown by the tilted blue parallelogram below.



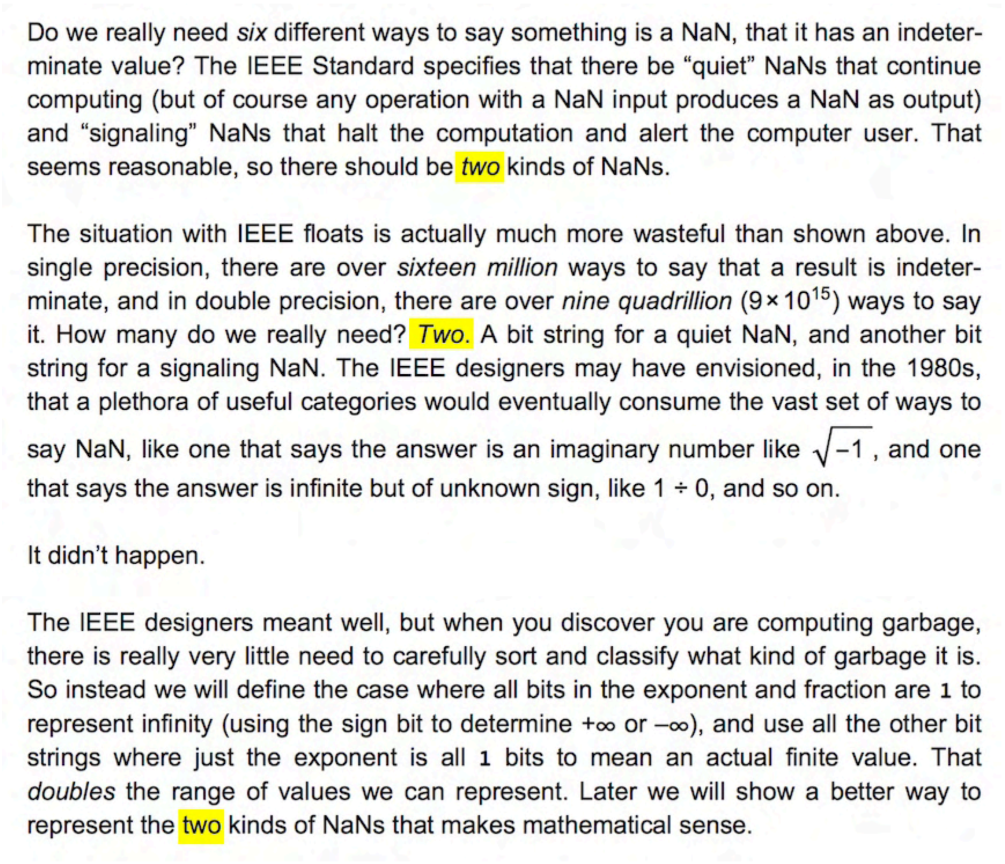
However, what interval arithmetic gives us is the *entire rectangle containing the parallelogram*, which is the "wrapper" for the solution set: $3 \leq x \leq 7$ and $2 \leq y \leq 9$. The area of the blue parallelogram is 9, whereas the area of the gray wrapping rectangle is 28, which is more than a *threefold* loss of information about the actual *c*-solution. If the crude description of the *u* and *v* bounds is then passed as input to another simple routine, information about the shape of the solution set is again lost. Without considerable care and expertise, interval arithmetic gives a bound too loose to be useful.

Had he even glanced through the book, spending one second per page, this diagram would have caught his eye instantly as an example of the wrapping effect. Yet he stated with confidence that the book shows no awareness of the problem.

We could speculate: Did Kahan read only the first part of the 400-page book, stopping before he got to that part of the discussion? No, that cannot be either. Another mistake, here visible only in his statement, visible on his next-to-last slide (page 33 of this document) and repeatedly asserted in his posted documents, that

"unums have only one NaN."

That means he did not even get as far as *page 23*, or he would find the fact that there are two kinds of NaN, italicized for emphasis. The three times the word "two" appears just on that page for the number of NaN types are highlighted in the following excerpt:



Do we really need *six* different ways to say something is a NaN, that it has an indeterminate value? The IEEE Standard specifies that there be "quiet" NaNs that continue computing (but of course any operation with a NaN input produces a NaN as output) and "signaling" NaNs that halt the computation and alert the computer user. That seems reasonable, so there should be *two* kinds of NaNs.

The situation with IEEE floats is actually much more wasteful than shown above. 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}) ways to say it. How many do we really need? *Two.* A bit string for a quiet NaN, and another bit string for a signaling NaN. The IEEE designers may have envisioned, in the 1980s, that a plethora of useful categories would eventually consume the vast set of ways to say NaN, like one that says the answer is an imaginary number like $\sqrt{-1}$, and one that says the answer is infinite but of unknown sign, like $1 \div 0$, and so on.

It didn't happen.

The IEEE designers meant well, but when you discover you are computing garbage, there is really very little need to carefully sort and classify what kind of garbage it is. So instead we will define the case where all bits in the exponent and fraction are `1` to represent infinity (using the sign bit to determine $+\infty$ or $-\infty$), and use all the other bit strings where just the exponent is all `1` bits to mean an actual finite value. That *doubles* the range of values we can represent. Later we will show a better way to represent the two kinds of NaNs that makes mathematical sense.

The point that there are two unums is emphasized throughout the book and is visible in the code listings in the Appendices as well.

When reading through this transcription and thinking about what Kahan asserts, bear in mind the above examples. There are many other examples of Kahan statements about unum arithmetic or the book that are blatantly incorrect.

Any statements Kahan makes about the contents or meaning of *The End of Error: Unum Arithmetic* are probably incorrect, since he provably did not read it. Similarly, his statements about what his own software would do show that he did not even test it by running it, as shown by the bug in his Compensated Summation code. His statements about what unum arithmetic would do show that he did not try running the prototype unum environment, or he would have discovered that his speculations were false.

—John Gustafson
7 January 2017