Floats for Philosophers: A talk in three parts

Robert M. Corless

August 2024

Western University, Canada

Slides available at `rcorless.github.io`; please download them

Joint work with Nic Fillion

Maple Transactions

Announcing Maple Transactions

a "Diamond" class open access journal with no page charges

Now listed by DBLP

mapletransactions.org

Reaching an audience of Philosophers

- Many philosophers prefer a very high-level view
- I prefer to work up to high-level views by generalizing from examples (this may be a mismatch of styles, no more)
- Even though I have now been cross-appointed to Philosophy for nearly 20 years now, I haven't yet adapted to the differences in styles (the above is only one instance)
- I'll do my best, today, though

Contents

Floats vs "real" numbers

- Why don't computers use "real"* or complex numbers, but rather floating-point numbers? Answer: Because we *can't* use real arithmetic (most of the time)—they're not computable in general—and in the rare cases we *can*, it's too slow, and except in even rarer circumstances the effort would be wasted.
- What's so important about floats? Answer: They make *computer memory usage predictable*, and therefore fast.

You can argue about these claims, but you'll be arguing in a mostly empty room. Most of the computing world uses floating-point arithmetic. Even if they grumble, they use it. This talk takes IEEE floats as being graven in stone, and we will today examine consequences, and not contest antecedents.

* "Real numbers" should have been called "continuum numbers", or something even less ambiguous. The confusion caused by calling them "real" is just brutal.

Direct consequences

Admit, for instance, the existence of a minimum magnitude, and you will find that the minimum which you have introduced, small as it is, causes the greatest truths of mathematics to totter.

—Aristotle, *On The Heavens*

The most important observation is that the set of floating-point numbers is *finite*. This means that there is a largest float and a smallest positive float (vide Aristotle's warning above!). There are—there have to be—*gaps* between floats.

This means (among other catastrophes) that several venerable paradoxes rear their heads.

Zeno of Elea, circa 490–430BCE

The Stanford Encyclopedia entry on Zeno's paradoxes lists *nine* paradoxes attributed to Zeno. Let's choose one, and paraphrase it in two ways.

Suppose Atalanta (who was famous for her running speed) had to travel from $x = 1$ to $x = 2$ (say, in meters). Zeno notes that before Atalanta gets to 2, she first has to get to $1 + 1/2$. Before she gets *there*, she first has to get to $1 + 1/4$. Continuing in this way, Zeno arrived at the paradox that Atalanta couldn't even get started, because there were an infinite number of things she had to do before getting anywhere.

Atalanta in floating-point

```
one := 1.0
s := 1.0 # Atalanta wants to step at least one meter
zeno := 0
# The following loop requires specification
while one+s > one do
  s := s/2 # Reduce Atlanta's step size by half
  zeno := zeno + 1
end do:
```

What happens

In Matlab and in Python* using NumPy, the loop terminates with the variable "zeno" having the value 53, and the variable "s" having the value 2^{-53} which is approximately 1.1×10^{-16}. Zeno might have been surprised, but would doubtless point out that Atalanta *still* can't get anywhere, because taking a step of size 1.1×10^{-16} leaves you in exactly the same place! Which is reminiscent of some of his other paradoxes.

Recall that a proton's width is now widely accepted to be 0.84 femtometers, or 8.4×10^{-16} meters. Floating-point arithmetic provides a *finer* measurement of matter than a molecular or atomic model does!

* Double precision in Fortran does the same. With C, it's mostly the same. Maple does something different (but still terminates).

Starting from zero would be different

A crucial part of the floating-point idea (which I am not going to pursue much here): zero is different, in floating point, from all other floating-point numbers. The set of floats is *much* denser around 0 than it is around any nonzero float (such as 1). This has deep implications for simulation of dynamical systems. See Gora and Boyarsky, "Why computers like Lebesgue measure" for the beginnings of an explanation.

[We suspect Zeno, Lebesgue, and Cantor would have all got along fine.]

Wendy Parker on Simulation

Evidence and Knowledge from Computer Simulation is an example of a recent philosophical study of computer simulation. The author (Wendy Parker) does not mention floating-point arithmetic, except obliquely by the use of words like "approximation."

This talk is much less ambitious than Parker's work—we are looking today only at the effects of "rounding error," which most people think too boring and horrible to spend much time on.

I hope to convince you that this is not so, and instead that it's worthwhile to spend at least a *little* time with.

Some details about floats

I will only talk about IEEE Standard floats. For background, read Kahan's 1981 paper "Why do we need a floating-point standard?". See also his 1983 paper (one of my favourites—do read it!) Mathematics written in sand.

"A computer is deemed Reliable when its users are never surprised by something its designers must later apologize for." — from that just-mentioned paper by Kahan

The IEEE floating-point standard

The IEEE floating-point standard was established in 1985, and has been revised a number of times since. The current version was described in 2019, according to Wikipedia. [That page is an excellent one, by the way.]

I'll talk about the "binary64" or double-precision format, which encodes each such float into a 64-bit number, and about "binary16" or half-precision, which is becoming so popular for large-scale computations because each one uses just 16 bits.

Half-precision

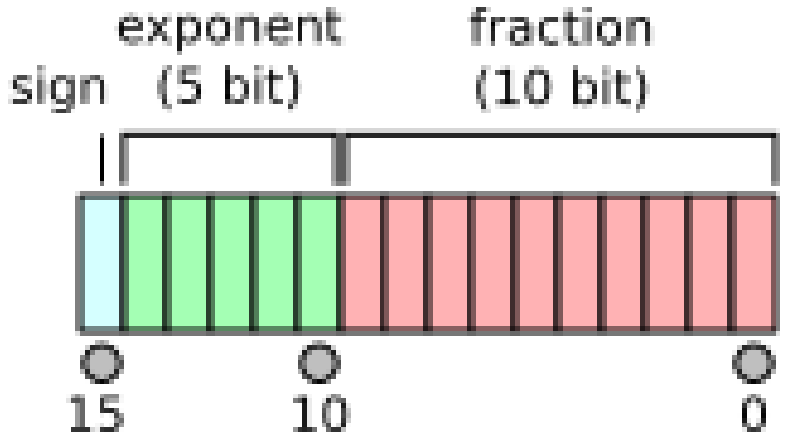

Figure 1: Image courtesy Wikipedia. Credit: Habbit/ Codekaizen - CC BY-SA 3.0

Double-precision

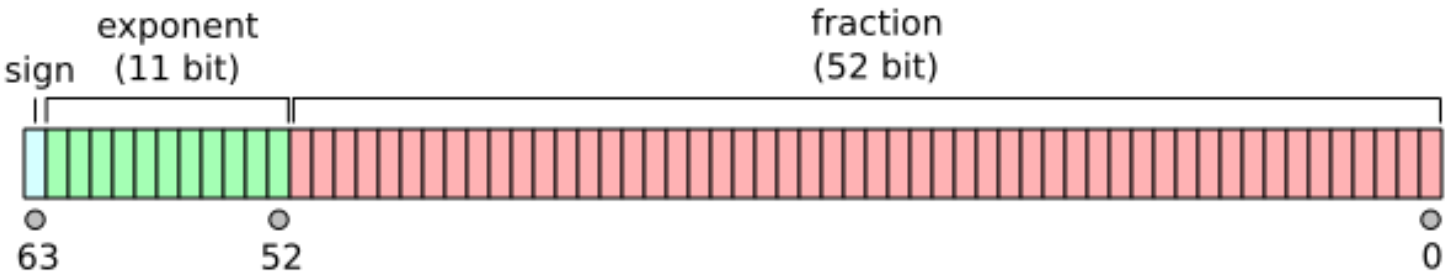

Figure 2: Image courtesy Wikipedia. Credit: Codekaizen - CC BY-SA 3.0

Aristotle's "minimum magnitude"

Floats have different "minimum magnitudes" depending on the context. The most important one (usually) is *half the distance to the next greater floating point number*. Starting at 1, this number is called the *unit roundoff* and is usually denoted u or μ. If $0 < s < \mu$, so s is a small positive number, then $1 + s$ is closer to 1 than it is to the next float larger than 1.

So it rounds *down* to 1 (Atalanta's step doesn't get her anywhere).

If $s = \mu$ exactly, then there is a choice, to round up or to round down; to reduce bias in long computations, the "round-to-even" rule is used and one rounds to the one of x_1 or x_2 whose last bit is *zero*. For $x_1 = 1$ and x_2 being the next larger number, the number is 1. Since Atalanta's step sizes were always powers of $1/2$, this actually happened in the loop.

Floats cannot be associative, then

It's a brutal shock to many that $(a + b) + c$ is not necessarily $a + (b + c)$ if one uses floating point arithmetic. Here is an example. Suppose $M = 1.0 \times 10^{22}$ (about the mass of the moon in kilograms). Then $100 + M - M$ will be 0 if computed as $(100 + M) - M$ and 100 if computed as $100 + (M - M)$.

This is a mathematical truth that has to totter when a minimum magnitude is introduced into the system.

Rebellion! We'll invent our own arithmetic!

The most popular alternatives to floating-point are *interval arithmetic*, its poor cousin *significance arithmetic*, and *arbitrary-precision* floating-point arithmetic. All have their own problems, but they have their uses.

But none of them are associative, either.

Arb (an interval arithmetic system) in Maple

```
> Digits := 15;
                                  Digits := 15                              (9)

> M := RealBox(10.^22);
                    M := ⟨RealBox: 1e+22 ± 1.67772e+07⟩                     (10)

> A := RealBox(100.)
                          A := ⟨RealBox: 100 ± 0⟩                           (11)

> A + (M - M);
                    ⟨RealBox: 100 ± 3.35544e+07⟩                            (12)

> (A + M) - M;
                    ⟨RealBox: 0 ± 5.03316e+07⟩                              (13)
```

Figure 3: Using interval arithmetic in Maple

Another important fact: Roundoff is Not Random

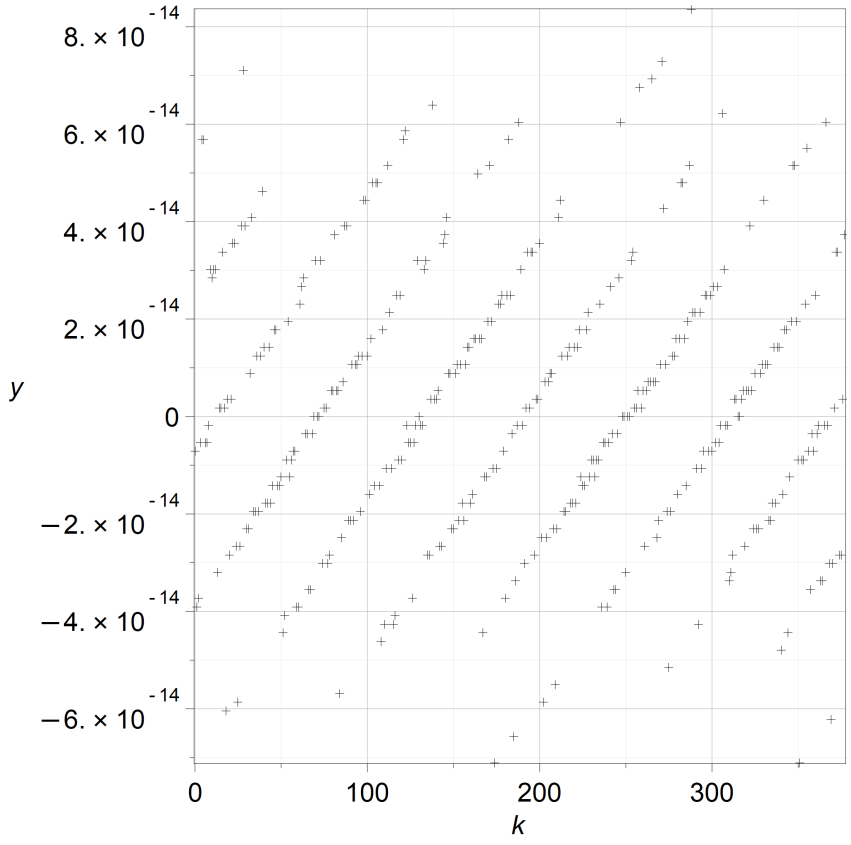

Figure 4: Roundoff is Not Random. This is the forward error in a certain rational function, evaluated at 378 neighbouring double-precision numbers. This example is a variation of an example of W. Kahan as used by Nick Higham (cf Figure 1.6 in his book *Accuracy and Stability of Numerical Algorithms*).

Yet in spite of all that

Floating-point arithmetic is a HUGE SUCCESS. And IEEE floats in particular are a huge part of that.

Do I need to multiply examples?

No, I do not. On to part II.

Part II: We can prove things

Floats can be extremely successful, and we can sometimes prove that they will be, ahead of time. At other times, we can provide a proof that explains the success of an algorithm.

Sometimes floats fail spectacularly, and we can prove that this has to happen (and give circumstances to avoid).

One more bad example first

Consider the quadratic equation $x^2 - 2bx + 1 = 0$. Its solutions are

$$x = b \pm \sqrt{b^2 - 1}\,. \tag{1}$$

This formula was (probably) known to the Babylonians. Every schoolchild that is old enough knows it (with $ax^2 + bx + c = 0$, the above is just a special case).

We'll solve it in (say) Matlab.

Matlab

We choose a lot of values of b, from 10^6 up to 10^9, logarithmically spaced. `b = logspace(6,9,2024);`

```
r1 = b + sqrt( b.^2 - 1 );
r2 = b - sqrt( b.^2 - 1 );
one = r1.*r2;
figure(1), semilogx( b, one, 'b.'  ), grid on
```

The results

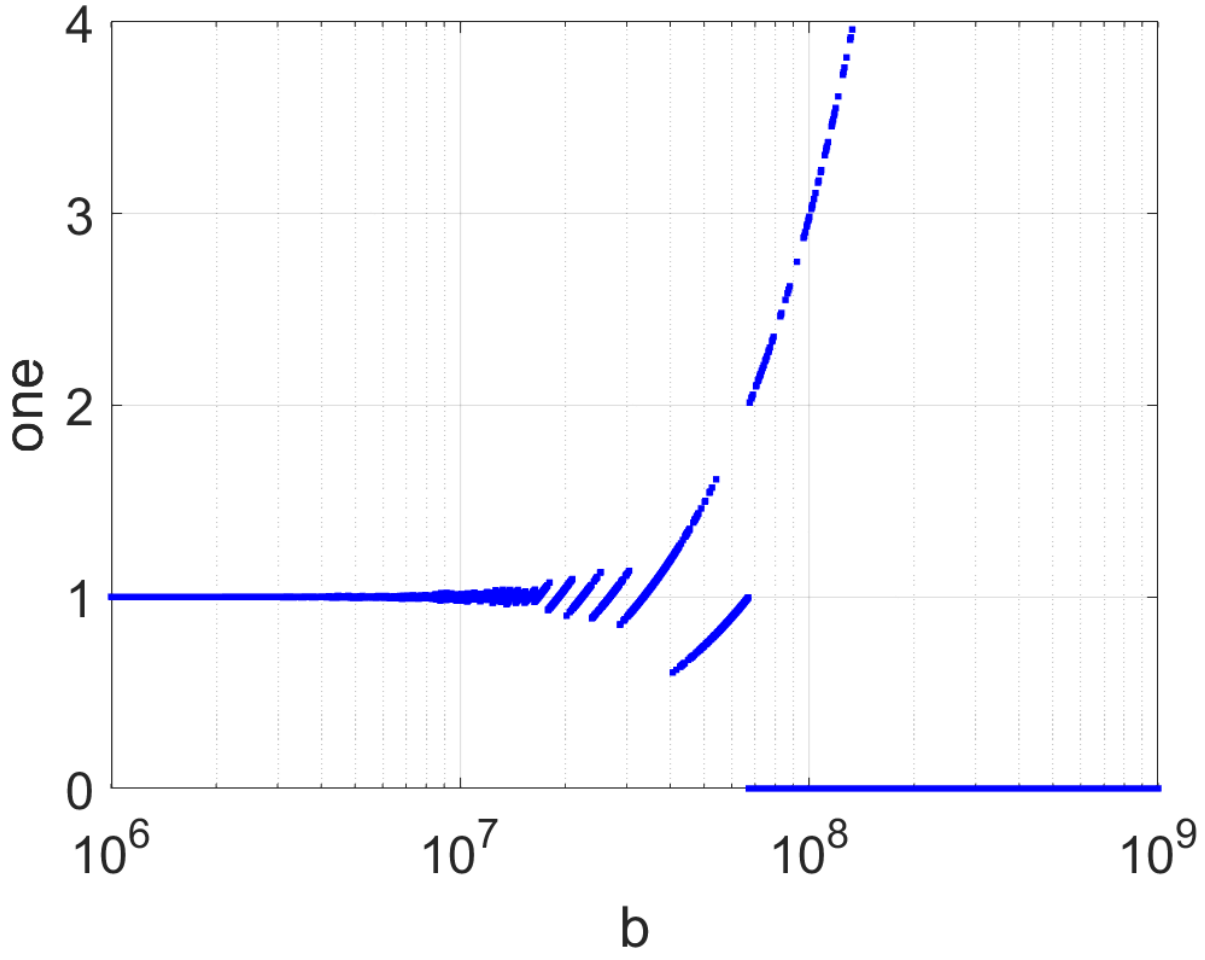

Figure 5: $(b + \sqrt{b^2 - 1}) \cdot (b - \sqrt{b^2 - 1})$, which should be 1. Rounding errors are not random! Analysis can reveal the culprit, here.

The relation to real analysis

The big thing that the IEEE standard does for us is The floating-point result of any basic operation on pairs of numbers must be the exact result, rounded to the nearest machine number. This is the *best possible outcome*.

In symbols, (for $+$ but can replace by $-$, $\cdot$, and $/$)

$$\mathrm{fl}(a+b) = (a+b)(1+\delta) \tag{2}$$

where $|\delta| \leq \mu$, the unit roundoff. No such guarantees for operations on three or more floats.

Using these, we may painstakingly prove theorems about certain algorithms. Quite astonishingly, many important modern algorithms (e.g. partial pivoting* for Gaussian elimination, or QR decomposition) *rely* on rounding errors cancelling themselves during the process. These don't work well in interval arithmetic!

* Embarassingly, there is no full proof for this. Trefethen & Schreiber 1990 is the best we know.

Errors in the quadratic formula

The floating-point value of b^2 is $b^2(1+\delta_1)$ for some small δ_1. The floating-point value of b^2-1 is thus $(b^2(1+\delta_1)-1)(1+\delta_2)$ for some small δ_2. Then the square root is $\sqrt{b^2(1+\delta_1)(1+\delta_2)-(1+\delta_2)}$. This is quite close to b, if b is large. Then subtracting this from b gives us another formula we can analyze.

In words, subtracting two nearly equal quantities reveals the earlier rounding errors because the correct significant figures cancel. This is called "catastrophic cancellation." [There is such a thing as "benign cancellation," by the way.]

It's interesting, though, that those revealed rounding errors are *not* random.

Backward Error: a tool for computational epistemology

Although backward analysis is a perfectly straightforward concept there is strong evidence that a training in classical mathematics leaves one unprepared to adopt it. ... I have even detected a note of moral disapproval in the attitude of many to its use and there is a tendency to seek a forward error analysis even when a backward error analysis has been spectacularly successful.

—J. H. Wilkinson. The state of the art in error analysis. NAG Newsletter, 2/85:5–28, 1985.

Backward error for the quadratic formula

The two computed roots x_1 and x_2 were the exact roots of $x^2 - 2bx + \mathrm{one}$ (where "one" was the quantity we plotted). Compared to $2b$, which was about 10^8 when the difficulties started to be visible, "one" is pretty small, though not *that* small. In that sense, we had computed the exact roots of a "nearby" polynomial—but it wasn't terribly close to the original; certainly not 16 figures close. This backward error view shows that our computation wasn't very good (and we did not have to compare to the exact answer, which would be a "forward" error analysis). We also didn't need those δ_ks, either. Backward error analysis is frequently simpler than forward analyses.

Error in the $2b$ coefficient

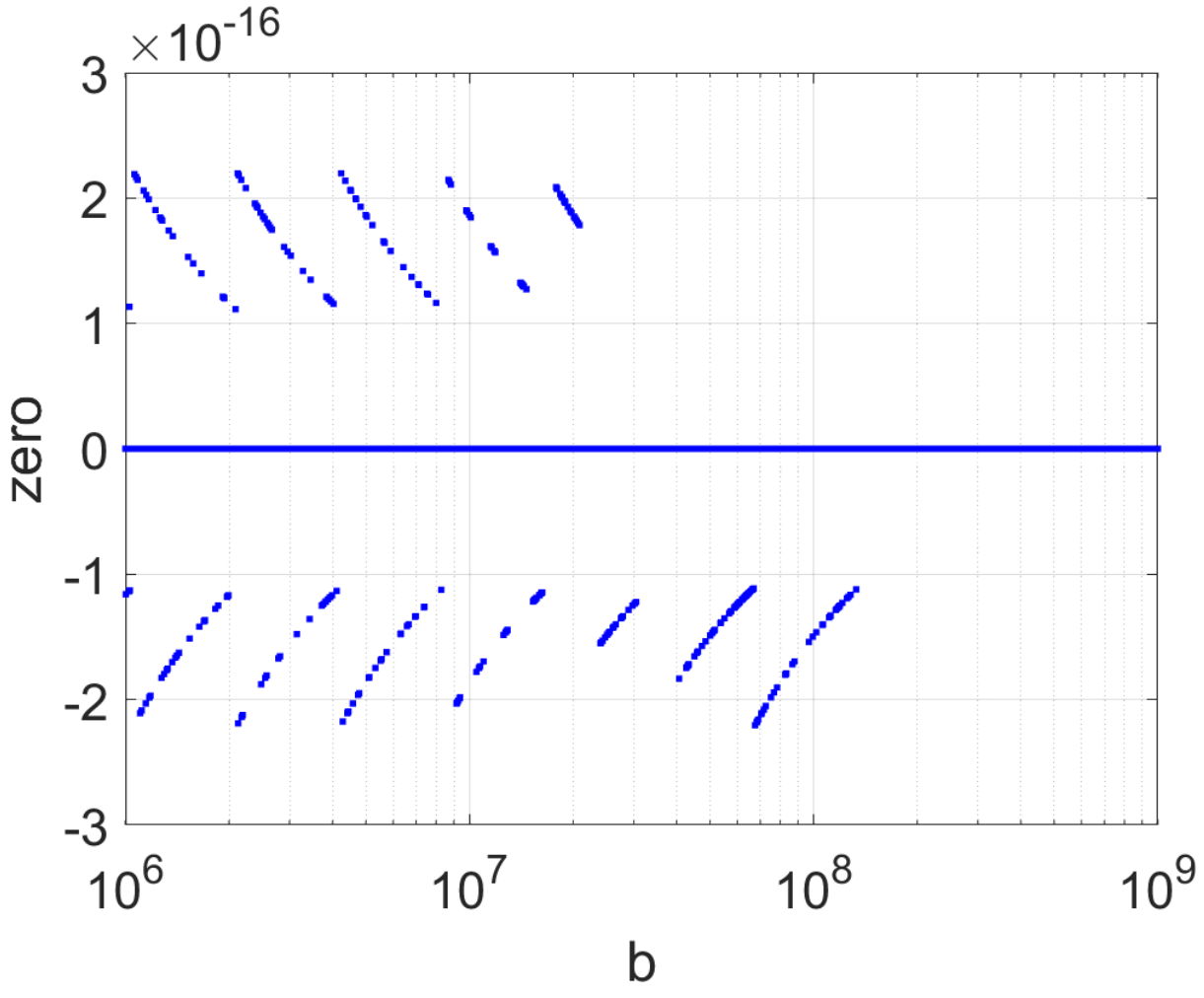

Figure 6: The relative error $(x_1 + x_2 - 2b)/(2b)$. This is uniformly small (but again we see it is *not* random).

A typical backward error result

Theorem: When evaluating the polynomial*

$$p(x) = a_0 + a_1 x + a_2 x^2 + \cdots a_n x^n \tag{3}$$

(with real floating-point coefficients a_k, and at a real floating-point value of x) by use of Horner's method using an IEEE-854 floating-point compliant system with unit roundoff μ, and if the degree n is smaller than $1/(3\mu)$, then the computed result is the exact value of

$$\tilde{p}(x) = a_0(1+\delta_0) + a_1(1+\delta_1)x + a_2(1+\delta_2)x^2 + \cdots a_n(1+\delta_n)x^n \tag{4}$$

where each δ_k is bounded by $|\delta_k| \leq 3n\mu/(1-3n\mu)$.

NB: Using *double precision* $\mu = 2^{-53} \approx 1.1 \times 10^{-16}$.

* And by the way polynomials are *really* important in scientific computing. All computers can really do is add, multiply, subtract, and divide; if you only do the first three, that's a polynomial.

Horner's method

Horner's method in pseudocode is

```
p := c[n];
for k from n-1 by -1 to 0 do
  p := c[k] + x*p;
end do;
```

The proof of the theorem uses mathematical induction and the IEEE guarantees about floating-point operations, namely that they must give the *exact result*, *correctly rounded to the nearest floating-point number* (with a special tie-breaking rule for those rare times where the exact result is half-way between floating-point numbers, called "round to even").

In words, Horner's method is *numerically stable*.

“Cashing that out”—see, I know some philosopherese

- Computation with floating-point is different from computation with “real” numbers. In particular, inevitable rounding errors can accumulate.
- Backward error analysis puts such differences *on the same footing as data errors*.
- Since you have to understand the effects of data errors anyway (typically by perturbation methods) this represents a potentially significant economy of human effort.

A strong backward error result

Suppose that we want to solve a *triangular linear system of equations* (about the easiest kind there is, almost). Say $\boldsymbol{T}x = b$ where $\boldsymbol{T}$ is an $n \times n$ matrix that happens to be "upper triangular" in shape—that is, the entries below the main diagonal are all zero. The given vector b is just a collection of n numbers, and the desired solution x is a vector of unknowns.

The theorem

Then computation with IEEE standard floating point arithmetic guarantees that using back substitution (absolutely the most natural algorithm) gives you a computed $\hat{x}$ which is the *exact* solution of

$$(\boldsymbol{T} + \Delta\boldsymbol{T})\,\hat{x} = b \tag{5}$$

where each entry of $\Delta t_{i,j}$ of $\Delta\boldsymbol{T}$ satisfies $|\Delta t_{i,j}| \leq \gamma_{n+9}|t_{i,j}|$. In words, the computed solution solves a linear system of exactly the same type, where the nonzero entries $t_{i,j}(1+\delta_{i,j})$ are very nearly the same as the original. **Zero entries are not disturbed.** Each perturbed entry has been changed by at most $(n+9)\mu/(1-(n+9)\mu)$; for $n+9 < 10^d$ this preserves $16-d$ decimal digits in the data. If n is a thousand, and you are using double precision, that means you got the exact solution to a system of the same kind as you wanted, where the data was the same up to about 12 figures. If you are an astronomer, you might have data so precise; but usually not otherwise.

Nothing special has to be done—this is a guarantee for the most natural algorithm.

An amazingly strong BEA theorem*

Simulating the Gauss map $x_{k+1} = \mathrm{frac}(1/x_k)$ from (0,1) to (0,1) in IEEE floating point gives an orbit of floating-point numbers x_0, x_1, x_2, ... which is *uniformly* shadowed by an orbit of the true (continuum numbers) Gauss map where the initial value $\hat{x}_0$ is $O(\mu)$ close to the floating-point x_0. [The shadowing is done by the orbit of the continued fraction that the numerical map generates.]

This means that *every* computed orbit is shadowed by a real one, with a really nearby initial condition. Corless, 1992, Continued Fractions and Chaos

We will investigate this in this talk using half-precision.

* Well, I think so

Does that mean the forward error is small?

Not necessarily. "Ill-conditioned" problems are *sensitive to changes*. "Chaotic" problems are exponentially sensitive. A difference of $O(\varepsilon)$ in initial conditions will, in $O(\ln \varepsilon)$ time, produce $O(1)$ changes in the solution.

But then they are sensitive to *changes in the model or data* as well, and you needed to know that anyway.

[If *nothing* about the problem is insensitive to perturbations, you will not be able to predict anything about it. So there has to be something.]

The basic method of BEA

- First one designs an algorithm
- Ideally, one tests it, sees that it works well, and then proves that it's numerically stable
- Then one shows how to compute the sensitivities of the problem the algorithm solves. In some cases one gets a "condition number" $\mathcal{K}$ so that for input x and output y one can estimate the relative effects of changes Δx to the input

$$\frac{\Delta y}{y} \approx \mathcal{K} \frac{\Delta x}{x} \tag{6}$$

Reminder: floats are finite in number

There are fewer than $2^{16} = 65{,}536$ half-precision floats. [Some of the patterns encode things like NaNs (NaN=Not-A-Number) and infinities.]

There are fewer than $2^{64} \approx 1.844 \times 10^{19}$ double-precision numbers.

Theorem: Every deterministic discrete dynamical system $x_{k+1} = F(x_k)$ ultimately cycles, for every floating point starting value x_0, when executed in floating-point arithmetic. In particular, there is no chaos, or even quasiperiodicity. There are only transients, then cycles. The transients and cycles may all be surprisingly short.

A schematic

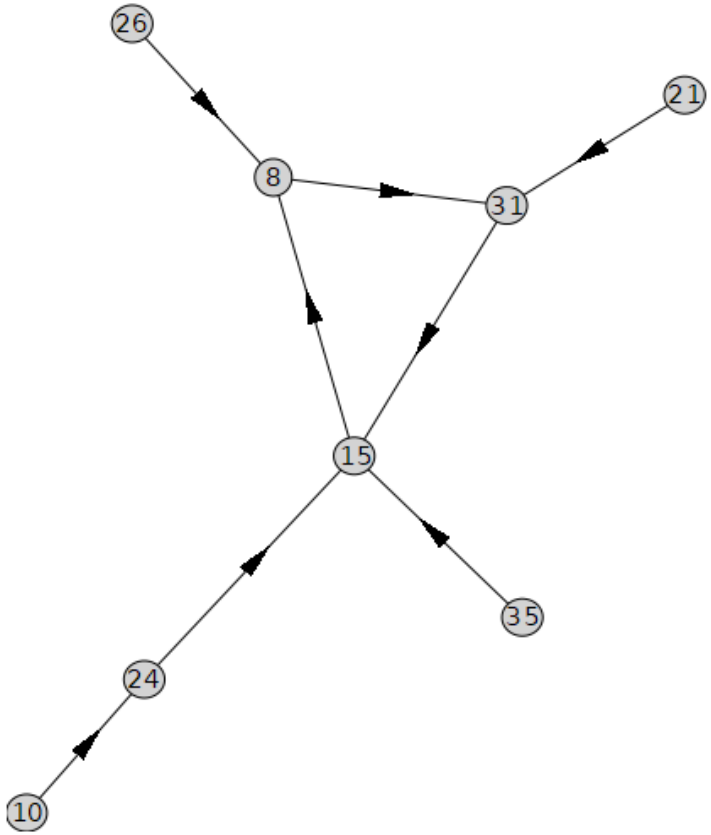

Figure 7: A schematic of a dynamical system acting on a subset of floating-point numbers. Each node represents a floating-point number. The arrow represents the map $F(x)$ from one float to another. Ultimately the orbit must cycle (perhaps of length one, which is called a "fixed point").

The length of the transient and the length of the cycles

If the dynamical system F is chosen "at random" (whatever that means) then one can expect that the length of the longest cycle and the length of the longest transient will be $O(N^{d/2})$ where N is the number of floats available, and d is the dimension of the attracting set.

This "square root" behaviour (the factor $1/2$ in the exponent) means that the cycle length can be surprisingly short. In half-precision for $d = 1$ this is just 181, so one ought not to be surprised if the *longest* cycle has about 200 elements in it.

Now on to some extended examples.

Part III: Two extended half-precision examples

In this section we look at two dynamical systems, which we can prove some things about (although not everything).

We will then stand back and marvel at how successful they are, in spite of their limitations. Even in half precision!

Our first example: The Gauss Map

The discrete dynamical system given by the Gauss map $G(x) = \mathrm{frac}(1/x)$, taking the fractional part of $1/x$, is well-defined on (0,1). Its iterations show up in the theory of *continued fractions*. If $x = 0$ then we specially define $G(0) = 0$ and so the iteration reaches an artificial fixed point (an orbit of period 1).

When implemented in floating-point, all its orbits are ultimately periodic. I wrote about this thirty years ago and proved some theorems. Today we'll talk about what happens in half-precision, float16. I used the Julia implementation of float16 for my computations, but I analyzed the output in Maple.

Nic and I co-supervised a student, Irene Claudia Noharinaivo, at AIMS in 2020, the African Institute of Mathematical Sciences, who worked in Python on this project. I'll take her work further in this talk.

Continued fractions

We can write

$$\sqrt{2} = 1 + \cfrac{1}{2 + \cfrac{1}{2 + \cfrac{1}{2 + \cfrac{1}{2 + \cfrac{1}{\cdots}}}}} \tag{7}$$

That's hard to typeset so we write $\sqrt{2} = 1 + [2, 2, 2, \ldots]$.

A general simple continued fraction in (0,1) can be written $[a_1, a_2, a_3, \ldots]$. If the number is rational, the continued fraction terminates.

The Gauss map takes $[a_1, a_2, a_3, \ldots]$ to $[a_2, a_3, a_4, \ldots]$. That is, it shifts the entries of the continued fraction by 1 and discards the first entry.

See C.D. Olds, "Continued Fractions." He won a Chauvenet prize for his paper on the simple continued fraction of e, and his book is likewise utterly lucid.

The unit interval

There are 15,361 float16s between 0 and 1, inclusive. We generate them by putting x[1]=Float16(1) (the Julia syntax for creating a float16 copy of the number 1) and generating the next one smaller by the command x[k+1] = prevfloat(x[k]). To find the number 15,361 I simply counted them all—I did not deduce what the number should have been from the definition, although I could have, because $15360 = 15 \times 2^{10}$.

The 256 smallest nonzero numbers, from $x_{15,105}$ down to $x_{15,360}$, behave badly under the Gauss map. When you invert them, they *overflow* because they are too big to fit in the float16 format. [These are the smallest "subnormal" numbers, though I don't really want to talk about them.]

The next smallest nonzero numbers, from x_{10225} down to $x_{15,104}$, are not so bad, but when you invert them you get numbers which are "big integers" in that their fractional parts fall off the end of the float16, so the fractional part is 0. So all of these numbers get mapped to 0.

Everything else

We analyze the orbits by using the GraphTheory package in Maple. This tells us that there are 9 connected components of the graph. Each vertex in the graph represents a distinct float16, and each directed edge tells which float16 the Gauss map takes it to.

Float8

Nobody uses 8-bit floats. But to make some understandable pictures I simulated them in Maple. One bit for sign, three bits for exponent, and four bits for significand. This gives 49 float8s in [0,1]. The Gauss map has 4 components on this reduced set, with $[17, 14, 8, 10]$ elements, respectively (a one-cycle, two two-cycles, and a three-cycle).

The 1st component

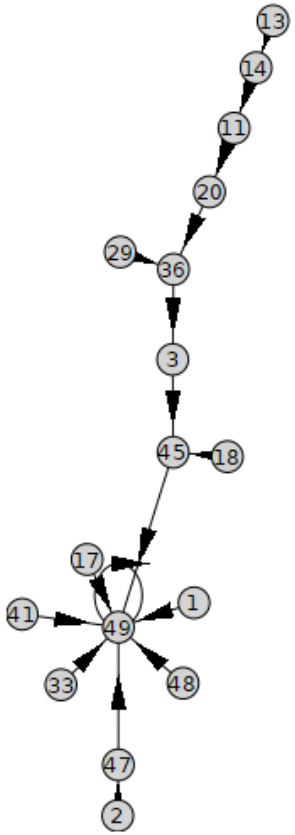

Figure 8: The first component, ending at 0

The 2nd component

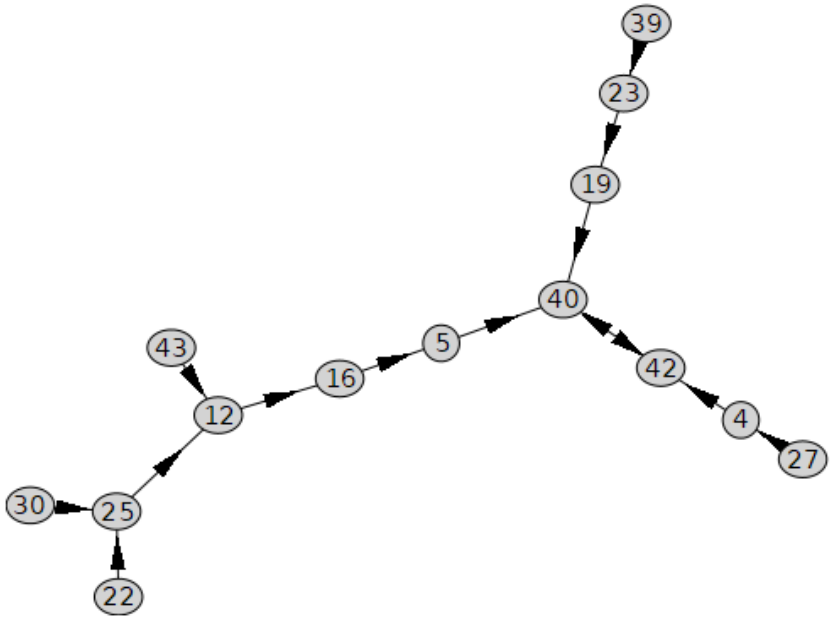

Figure 9: The second component, ending at a two-cycle

The third component

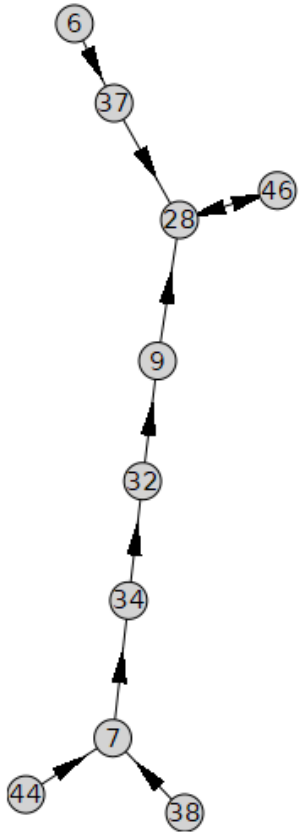

Figure 10: The third component, ending at another two-cycle

The 4th component

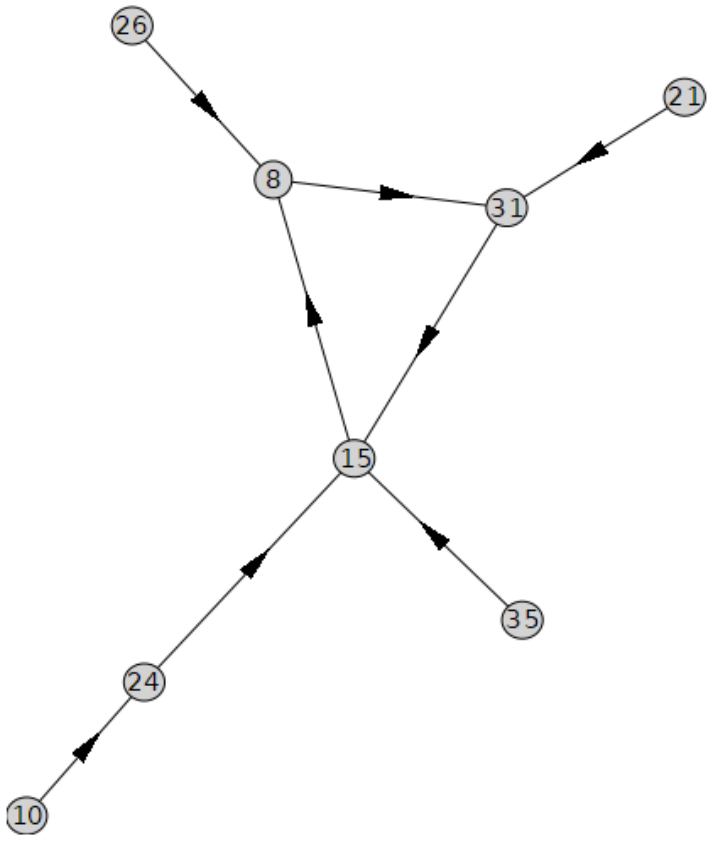

Figure 11: The fourth component, with the three-cycle $1.1001_2 \times 2^{-1}$, $0.1001_2 \times 2^{-2}$ (a subnormal number), and $1.0010_2 \times 2^{-1}$.

Components in Float16

Back to half-precision, Float16, and the components of the Gauss map on the 15,361 float16s in (0,1). The sizes of these components are (after removing 256 entries from the first component that actually lead to NaN16s)

$$\{11119, 1273, 286, 587, 198, 1477, 126, 38, 1\} \tag{8}$$

The first component is the largest. Every element in that component ultimately goes to 0 under iteration of $x_{k+1} = G(x_k)$. All rational numbers *should* go to 0 ultimately.

The smallest component

In contrast, the smallest component has just one entry, $x_{783} = 0.618$ (in decimal form), and it has $G(x_{783}) = x_{783}$, a fixed point. Moreover, no other float16 is mapped to that fixed point. It's not an accident that this is the closest float16 to the golden ratio. This means that we can correctly compute the infinite continued fraction for $\phi = (1 + \sqrt{5})/2 = 1.618\ldots = 1 + [1,1,1,\ldots]$ using half-precision.

Next smallest component

The 8th component has 38 elements. Following each possible path here leads to the fixed point $x_{1833} = 0.3027$. NB $(\sqrt{13} - 3)/2 = 0.3027756\ldots = [3,3,3,\ldots]$. The *longest* such path has five elements in it.

Next smallest after that

The 7th component has 127 elements. Following each possible path leads to the fixed two-cycle $a = 0.3281$ and $b = 0.04688$. The longest path to that cycle has just six elements in it. NB $(\sqrt{469} - 21)/2 = 0.3282039\ldots = [3,21,3,21,3,21,\ldots]$ while the other number is $(\sqrt{469} - 21)/14 = [21,3,21,3,\ldots]$.

The 6th component

The sixth component has the longest cycle, with 18 entries (a random map with this many floats in it could be expected to have a cycle of length 123). The longest transient leading to this cycle has 26 elements in it, including the element in the cycle that it hits. [The 2nd component has a longer transient, with 30 elements in it. The 2nd component has a 6-cycle.]

More about the largest component

The largest component has 11,119 elements in it, plus 256 elements in it that lead to NaN16s and should have their own category. This is about 2/3 of the available float16s. But about 5,000 of these are so small that on inversion they are "automatic" integers on rounding.

All 11,119 entries go to zero in twenty or fewer steps of the map. There are just three entries which take exactly twenty steps: $x_{740} = 0.639$, $x_{1476} = 0.39$, and $x_{1924} = 0.2805$. Every other transient is shorter.

Contrast with the "true" Gauss map

Almost all real numbers have infinite orbits under the Gauss map. With probability one, the patterns of the orbit never repeat. The Lyapunov exponent is positive ($\pi^2/(6 \ln 2)$ for almost all initial values); this map is chaotic, and forward error wouldn't help.

All rational numbers (set of measure 0) have finite orbits, terminating at zero. The longest exact orbit of any exact rational float16 has 14 entries, and only two achieve that (1893/65536 and 1809/524288).

All ultimately periodic orbits are quadratic irrationals (this is a theorem of Lagrange). All purely periodic numbers are a specific subset of those (this is a theorem of Galois).

The floating-point orbits couldn't be more different in kind, even though every floating-point orbit is shadowed by a true orbit starting $O(\mu)$ close to the initial point.

It gets the distribution mostly right, though!

I claim that the float16 orbits, even though they are wrong in character (not just wrong by rounding errors) can *still* be useful. One can compute the Lyapunov exponent pretty well by the long transient plus long cycle orbit.

That this works is not fully explained, though.

The Logistic Map

Nic used the Logistic map $L(x) = 4x(1-x)$ in one of his talks. This, too, has an analytical solution and reduces to the tent map, so we know that this is also chaotic. If we repeat all the graph theoretical analysis above, this time for the float16 version of the Logistic map, we find that 5902 of the initial points lead, after at most 55 iterations, to 0. The value $3/4$ is a fixed point, and $1/4$ goes to $3/4$. The remaining 10,267 half-precision numbers go, after at most 83 iterations to a cycle of length 40.

That cycle

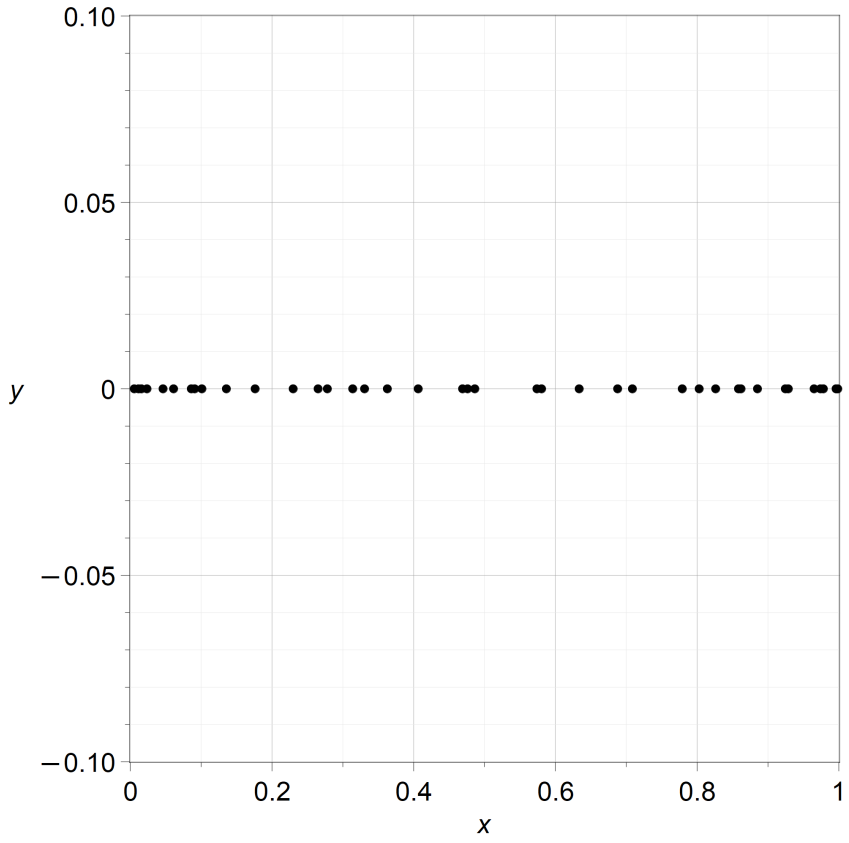

Figure 12: Where the forty elements of the half-precision logistic cycle lie.

Comparing to the true distribution

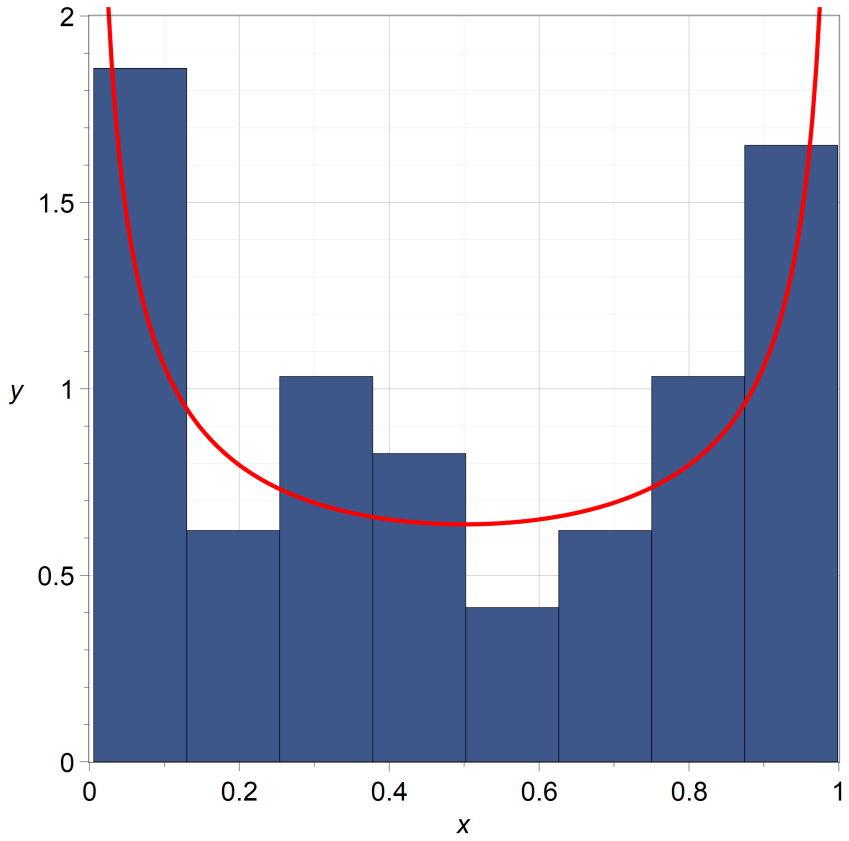

Figure 13: The true distribution of the "real" logistic map is $1/(\pi\sqrt{x(1-x)})$ (red line). The distribution of the forty element cycle is shown as a histogram. Float16 FTW

Concluding remarks

What you should take away:

- Floats are amazing
- Backward error analysis can tell you if an algorithm is numerically stable, which means that it gives you the *exact* answer for a nearby question (data, model) **economising human effort**.
- Some problems (e.g. chaotic problems) are sensitive to changes
- Even the strongest backward error results can be tricky: your nearby problem may not be "typical" of the problems in the neighbourhood. This is an open problem in modelling, even though many physicists want to believe that it's solved. See e.g. Paul Tupper's 2009 paper on Molecular Dynamics to see that it is *not* solved.
- I didn't say this, but although Backward Error Analysis (BEA) can be used for floating-point arithmetic, it can also be used for other approximation methods. It's quite useful.

Acknowledgements

Irene Claudia Noharinaivo got us interested in the Gauss map again. Tony Roberts sent us the quote from Wilkinson. Paul Tupper has sent several useful remarks, in addition to the reference to his extremely interesting 2009 paper.

I also thank Bill Harper for getting me involved with Philosophy at Western, so many years ago, by inviting me to co-teach his course on gravity. That was a lot of fun, and has led to a lot more over the years.

Thank you for listening.