The Last Bit

How machines evaluate **exp** and **sin**, why two *correct* answers can differ, and what a pair of disagreeing Excels taught us about forty years of silicon.

A TEA-LENGTH FIELD GUIDE · for a programmer, Excel devotee, maths instructor & finance researcher · July 2026

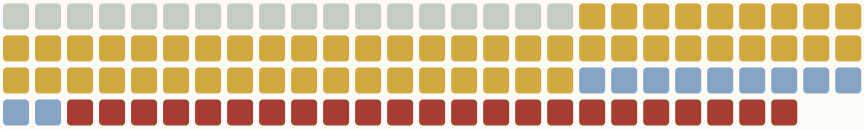
"The fast drives out the slow, even if the fast is wrong."
— William Kahan, architect of both the 8087 and the IEEE 754 standard, which is roughly the whole story in one sentence.

§ 1 TWO SPREADSHEETS WALK INTO A LAB

Last week we asked two computers the same question: the same build of Excel, the same formula, the same 64 bits of input. An Intel Core i7 and an AMD Ryzen 7 returned answers that differ — by exactly one bit, the very last one.[1] This document is about why that is not a bug, why it was nearly inevitable, and what it opens onto: a forty-year story involving a legendary coprocessor, a $475 million rounding error, a quiet revolution called SSE2, and the reason every AI model on Earth is, at heart, an exp-evaluating machine.

```
=EXP(0.24258475769541302)  x = 3fcf0d047067aa41
Intel i7-9750H → 1.2745392715968606  3ff464834a8f9e6d
AMD Ryzen 4800H → 1.2745392715968609  3ff464834a8f9e6e
true value → 1.2745392715968606…  (Intel is right, this time)
```

Across all 106 probes the shape of the disagreement is precise enough to read like a fingerprint:



18 control probes — everyone agrees, to the bit
54 "double-rounding traps" — Intel = AMD, both one bit off the true value
11 microcode probes where the vendors still agree
23 probes where Intel ≠ AMD — every gap exactly one bit

One hundred and six questions, four kinds of answer. The divergence is surgical: it appears *only* on inputs designed to stress the hardware's transcendental microcode, never on ordinary values. Where the two CPUs disagree, Intel is the one closer to the mathematically true value 18 times out of 23. And on 95 of 106 probes the Intel machine returns *exactly* what an idealized 1980s x87 coprocessor would return — a clue we will come back to in § 6.

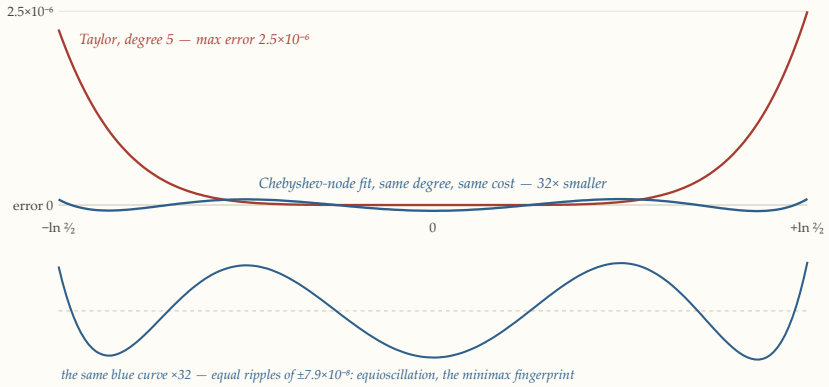
To see why two correct machines may differ, we need to know what a machine actually does when you type `=EXP(A1)`. It is lovelier than you might expect.

THREE SIPS, IF YOU READ NOTHING ELSE
① No law fixes the last bit of exp or sin — hardware and libraries are only required to be *close*, and different vendors are close in different directions.
② Excel's LN and EXP behave like a 1980s Intel coprocessor preserved in amber; the only place our two machines disagreed was inside that amber.
③ One wrong bit is harmless — until it crosses an `IF`, a barrier, a chaotic iteration, or a checksum. Then it is binary.

1 The experiment: 106 probe values for `LN` and `EXP`, chosen adversarially, sent as a PowerShell script that drives Excel and reads back raw IEEE-754 bit patterns. "Agreement" below means *bit-for-bit identical*, the strictest possible standard.

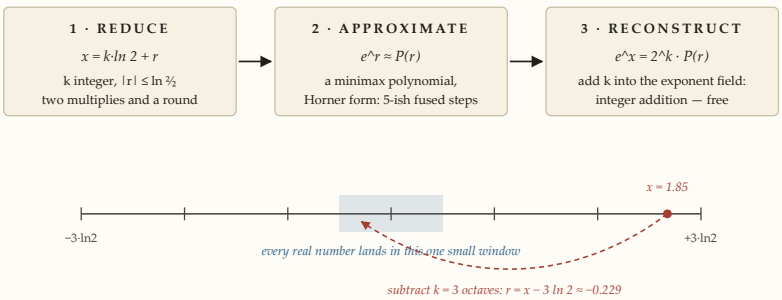
§ 2 HOW A MACHINE COMPUTES EXP (HINT: NOT WITH TAYLOR)

You know the Taylor series. So did the engineers — and they rejected it, for a reason worth internalizing: *Taylor series are infinitely accurate at a point, and you are paying for that accuracy exactly where you don't need it.*[2] A machine gets a fixed budget — say five multiplications — and must spread its accuracy evenly over a whole interval. The mathematics of doing that optimally (Chebyshev, equioscillation, the Remez exchange algorithm) predates computers, and it buys astonishing amounts: same degree, same cost, ~32× less error.



Same degree, same price, 32× the accuracy. Error of two degree-5 polynomial approximations to e^x on the interval a real libm uses. The Taylor polynomial (red) is perfect at 0 and pays for it at the edges; the Chebyshev-node fit (blue) spreads its error democratically. Production libraries push one step further (true minimax, via the Remez algorithm) and tune the coefficients so the polynomial is optimal *in floating-point arithmetic*, not just on paper. Both curves here are real computed data.

The polynomial is only act two of a three-act structure that every implementation of exp — 1980 microcode, glibc, your GPU — shares:



Reduce, approximate, reconstruct. The deep trick is choosing ln 2 as the step: then the coarse part of the answer, 2^k, costs *nothing* — it is an integer added into the exponent bits of the result. All the analytic work collapses onto a window half a unit wide. Serious libraries shrink it further with a small table (32 stored values of $2^{j/32}$ cut the window 32-fold[3]), after which a degree-4 or 5 polynomial reaches full double precision. Total cost: about a dozen arithmetic operations.

§ 3 SIN, AND THE TROUBLE WITH Π

exp got the friendly reduction constant. sin must be folded modulo π/2 — and π, unlike ln 2, has a habit of biting. When x is large, computing x − k·π/2 is a subtraction of two nearly equal huge numbers: catastrophic cancellation. To survive it you must carry π to absurd width — the standard method (Payne–Hanek, 1983) keeps about *1,100 bits* of 2/π on hand to reduce any double correctly.[4]

2 Taylor is also numerically suicidal for oscillating functions at large arguments: summing the series for sin(30) means wading through terms as large as 7.8×10^{11} that must cancel down to something ≤ 1. Twelve digits of your fifteen are destroyed before you begin. No library ever does this.

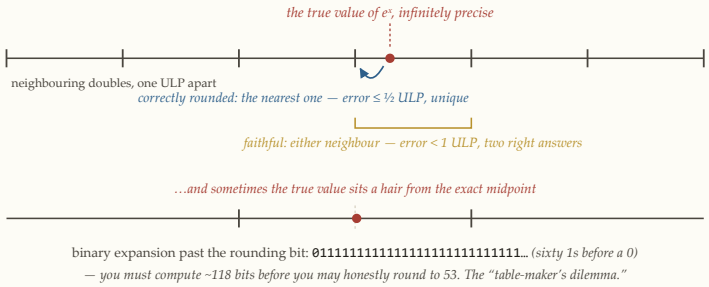
3 Table-driven methods (Tang, late 1980s) are the standard recipe to this day. Also standard since 2013: the fused multiply-add instruction, which computes a·b+c with a *single* rounding. Polynomials are evaluated as a chain of FMAs — faster *and* more accurate, one of the rare free lunches.

Intel's hardware instruction `FSIN` carries 66 bits of π. For decades its manual claimed 1-unit accuracy across the range. In 2014 Bruce Dawson checked, and the resulting blog post — "Intel Underestimates Error Bounds by 1.3 quintillion"[5] — forced a quiet rewrite of the documentation. Near multiples of π, FSIN returns answers where only a dozen-and-a-half of the 53 bits are correct. The instruction was not so much wrong as *honest late*: the sin of a number near π is tiny, the 66-bit π isn't nearly tiny-compatible, and everyone who knew numerical analysis had quietly stopped using FSIN years before.

> *Every large-argument sin evaluation is a little miracle of bookkeeping: the function is easy; knowing where you are on the circle is hard.*

§ 4 WHAT DOES "RIGHT" EVEN MEAN?

A double is a fraction with 53 significant bits. The true value of e^x almost never lands exactly on one, so some neighbouring representable number must stand in for it. Now the vocabulary of the whole field, in one figure:



Correct vs. faithful, and why the last bit is expensive. Correct rounding gives a unique answer but requires, in the worst cases, computing to roughly *twice* the target precision before the 53rd bit settles (Lefèvre & Muller found the actual worst cases by exhaustive search — that is how we know 118-ish suffices for double exp). Faithful rounding tolerates either neighbour and is what nearly everything actually ships.

Here is the institutional punchline. IEEE 754 *requires* correct rounding for +, −, ×, ÷ and √ — and for the transcendentals it merely, since 2008, *recommends* it.[6] The last bit of exp is not law; it is custom. Every libm, every chip vendor, every era of microcode is free to return either faithful neighbour, and they exercise that freedom differently. Two machines disagreeing in the last bit of EXP are not one right and one broken. They are two citizens of a standard that deliberately left that bit unlegislated.

The frontier is moving: crlibm (Lyon, 2004) proved correct rounding practical; the core-math project and Rutgers' RLIBM now produce correctly rounded routines *as fast as* the sloppy ones, and they are flowing into LLVM's libc.[7] Your grandchildren's spreadsheets may finally have a unique right answer. Ours do not: on our 106 probes, Excel is faithful on every single row — and correctly rounded on only 41. That is not an Excel scandal; that is the ecosystem's default posture.

§ 5 HOW ONE BIT ESCAPES THE CAGE

A last-bit error is 2^{-52} relative — on a billion-dollar notional, about a tenth of a micro-cent. The money is never the problem. The problem is that spreadsheets, and finance, are full of machinery that converts *continuous* errors into *discrete* events:

- **Thresholds.** `IF(EXP(x) > K, …)` flips on a one-bit change. A barrier option whose underlying is priced through exp can knock in on one machine and not the other[8] — a one-bit difference becomes the full rebate.

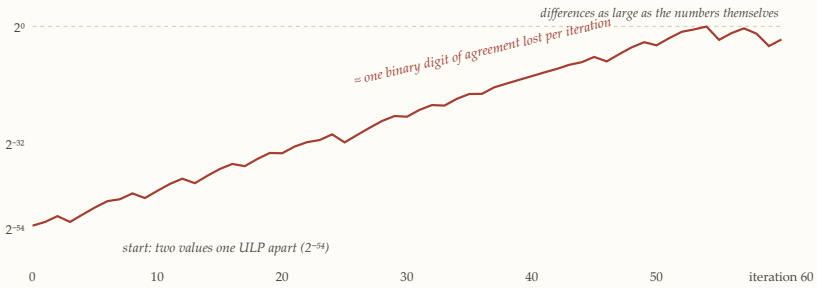
4 Why 1,100: the worst double is astronomically close to a multiple of π/2, and the reduction must survive up to ~2^{1024}-scale cancellation. Finding those worst cases is itself lovely mathematics — continued fractions of π do the hunting.

5 randomascii.wordpress.com, October 2014. Glibc had already switched to software sin; the post mostly changed the paperwork. A theme of this story: *documentation converging on long-known truth.*

6 Why the carve-out? In 1985 nobody could prove how much precision correct rounding of exp would need — the worst case might have been arbitrarily deep. The exhaustive searches that bounded it came fifteen years later. Standards codify what can be promised.

7 A precedent for "bit-identical everywhere" exists: Java shipped `StrictMath` — Sun's 1993 fdlibm, frozen — so every JVM on Earth returns the same sin. The cost was speed; Java 17 (2021) could afford strict semantics by default because post-x87 hardware made them free.

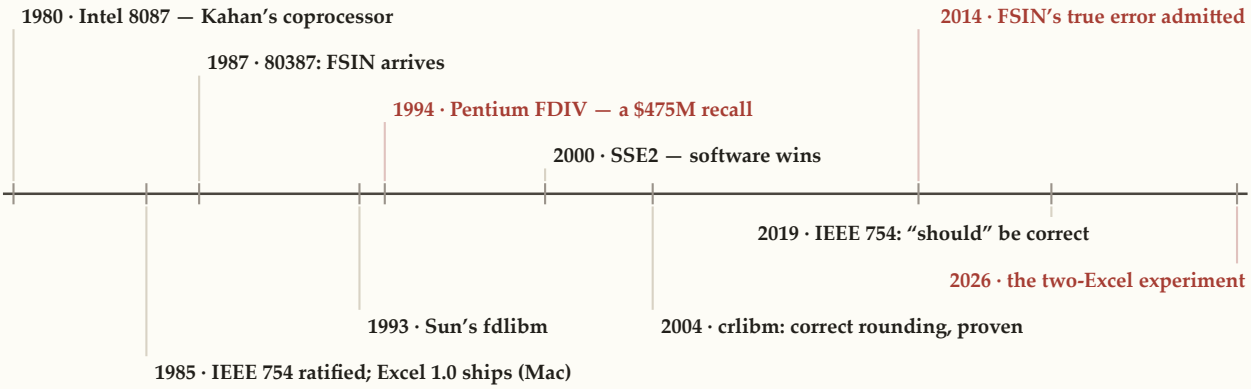
- **Iteration.** Feed a function's output back into itself and errors compound — or explode, if the dynamics are chaotic (figure below). Root-finders like XIRR and Goal Seek stop at tolerances; a last bit can move which side of the tolerance you stop on.
- **Identity.** Checksums, golden-file tests, "does the rerun match the filing" — reproducibility questions are binary. One bit in one cell of a million-cell model changes the hash exactly as much as a wrong sign would.



Chaos is a bit-amplifier. Two copies of the logistic map x ← 4x(1−x), seeded one ULP apart at 0.3, iterated in double precision — sixty rows of a spreadsheet. The gap grows by almost exactly one binary digit per iteration (the Lyapunov exponent is ln 2), so 53 bits of agreement are fully spent by row ~50; the trajectories are visibly different numbers by row 48. Real computed data — drag the formula down and watch.[9]

8 Digital options, barrier monitors, VaR breach flags, margin triggers — quantitative finance is unusually rich in cliffs. Smooth-payoff pricing is robust to the last bit; anything with an indicator function in it is not.

§ 6 FORTY YEARS OF SILICON, ONE TIMELINE



Two tracks, one story. Above the line, silicon; below it, standards and software. Note the rhyme at 1985 — the floating-point constitution and Excel are the same age — and the long silence after 2000, when new instruction sets simply stopped including transcendentals.

9 A counter-surprise from making this figure: seeded at 0.4 instead of 0.3, the one-ULP perturbation is *rounded away* on the very first iteration and the trajectories stay identical forever. Errors don't always grow. The dynamics decide — which is exactly why you can't wave the question away.

The 8087 (1980) is one of history's great chips: William Kahan helped Intel design it, then led the committee that turned its arithmetic into IEEE 754, the constitution every computer now obeys.[10] Memory was precious, so the transcendentals went into the chip as microcode — CORDIC-style shift-and-add routines of the same family HP calculators used — operating on a luxurious 80-bit internal format meant to absorb rounding sins before the result was cut down to 64 bits. The 8087 couldn't even do sin directly (it offered a partial tangent); `FSIN` arrived with the 387 in 1987.

Then two things happened. First, AMD: from the mid-90s it built its own x87 microcode rather than copying Intel's — legally it had to, and the standard, as we saw, only demanded *closeness* for transcendentals. Two implementations, both faithful, free to disagree in the last bit. Sound familiar? Second, the Pentium FDIV affair (1994): a lookup table in the new division hardware was missing five entries,

10 Kahan received the Turing Award in 1989 for this body of work. The 80-bit format was his belt-and-braces philosophy in silicon: compute with more than you promise.

a mathematician doing prime-number research noticed his reciprocals were wrong, and Intel eventually took a $475 million charge.[11] The contrast is the lesson: division is one of the operations IEEE 754 *requires* exactly right, so FDIV was a broken promise and a scandal. FSIN's far larger errors broke no promise at all. Nobody recalled anything.

The quiet revolution came in 2000 with SSE2: a whole new floating-point instruction set with *no transcendental instructions whatsoever*. By then, software running on fast multipliers beat microcode on both speed and accuracy, so the industry simply stopped. The x87 unit was left running for backward compatibility — a museum wing attached to every modern CPU. Libraries moved on: Sun's fdlibm (1993) became the genetic ancestor of nearly every libm alive[12]; Microsoft's C runtime was rebuilt on SSE2 in the UCRT era; results shifted, slightly and legally, from one Windows and glibc version to the next. A libm is not a constant of nature. It is a versioned opinion.

Which brings us home. Our probes say Excel's LN and EXP, on both machines, behave *exactly* like an idealized x87: on the Intel box, 95 of 106 outputs match a model of `FYL2X`/`F2XM1` — 80-bit internal computation, then rounding to double — and the two error populations we found are precisely the two this design predicts. The 54 gold cells are **double-rounding** casualties: inputs engineered so that rounding to 80 bits and then to 53 lands one bit away from rounding once, directly — identical on both CPUs, because it is a property of the algorithm, not the vendor.[13] The 23 red cells are the vendors' 80-bit microcode disagreeing *internally* by a whisker, which surfaces after the cut to 53 bits. Excel's EXP is, functionally, a 1980 coprocessor preserved in amber[14] — and before you judge, remember what Excel sells: forty years of files that must recalculate to the same answer. In that business, bit-stability *is* the feature. This is the company that still honours Lotus 1-2-3's fictional 29 February 1900 on purpose.

11 Thomas Nicely, computing sums of reciprocal twin primes, got different answers on different machines — the great tradition of noticing that our experiment humbly joins.

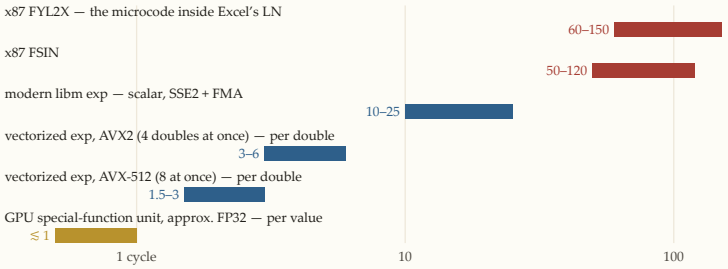
12 fdlibm's children: FreeBSD's msun, glibc, musl, Julia's openlibm, Java's StrictMath (bit-frozen). Written by K.C. Ng and colleagues — arguably the most-executed mathematics ever authored by so few people.

13 Double rounding in one line: round(round(v, 64 bits), 53 bits) ≠ round(v, 53 bits) whenever the first rounding nudges v across the second one's midpoint. Rare for random inputs (~1 in 2,000), designable at will — which is what our 54 "bias" probes did.

14 Meanwhile `SIN` in modern Office builds appears to travel the SSE2/library route — the amber is selective, which is its own kind of archaeological evidence.

§ 7 THE PRICE OF A TRANSCENDENTAL, 1980–2026

"Surely microcode is faster than doing it yourself?" It was — in 1980, when the alternative was a memory-starved 8086 grinding for milliseconds. Today the table has fully turned:



Cycles per evaluation, roughly, on Skylake-era hardware (log scale). Red is 1980s microcode still executing today; blue is software; gold is GPU hardware fast-path. Order-of-magnitude figures from vendor tables and Agner Fog's measurements[15] — take each bar with ±2× salt; the shape of the staircase is the point. The microcoded instruction Excel leans on is ~30× slower than a modern vectorized routine of equal or better accuracy.

Three inventions did this. **Pipelined multipliers** made polynomial terms nearly free. **FMA** (2013 on desktops) halved both the operation count and the rounding noise of a Horner chain. And **SIMD** changed the economics wholesale: a polynomial has no branches, so it vectorizes perfectly — compilers now silently rewrite a loop over `exp()` into calls that compute eight at a time. Meanwhile the

15 agner.org/optimize — the community's instruction-timing bible for two decades. The x87 instructions are microcode sequences internally, tens of µops; a vector exp is a handful of FMAs retiring four or eight results wide.

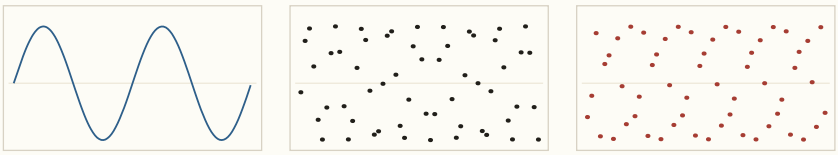
argument-reduction bookkeeping, the tables, the exponent surgery — all of it turned out to be ordinary integer work modern cores do in their sleep. The transcendental became a commodity.

§ 8 SMALLER FLOATS, STRANGER WORLDS

GPUs run the same three-act play with the fun turned up. NVIDIA chips carry *special function units* — actual dedicated hardware for approximate sin, cos, $\exp_2$, $\log_2$ — accurate to a couple of FP32 bits' shy of full, one result per clock, meant for the use-case where nobody sues: pixels.[16] Ask politely (`sinf`) and you get careful software; ask fast (`__sinf`, fast-math) and you get the SFU. The same "right vs. good enough" treaty, now with a menu.

And exp quietly became the most important function in computing. Every softmax in every transformer is a wall of exponentials; every token generated by every large language model spends its attention step evaluating exp by the million. That workload runs in FP16 and bfloat16 — formats so small they bend the rules of this whole story:

- FP16 has 65,536 bit patterns, total. You can test a function on *every possible input* in a millisecond — the table-maker's dilemma dies of exhaustion. You could even ship the entire function as a 128 KB lookup table.
- Range gets absurd: e^x overflows FP16 at $x \approx 11.09$.[17] This is why every softmax implementation subtracts the row maximum before exponentiating — a numerical-analysis move performed trillions of times a day by people who've never heard of it.
- And sin? Behold:



x near 6 — grid step 0.008: a sine | *x = 1024…1079 — step 1.0: aliased* | *x = 4096…4316 — step 4.0: noise*

sin(x), evaluated exactly, at consecutive representable FP16 numbers. The function isn't approximated here — every dot is the true sine of the input. The *format* is the problem: past 1024 the half-precision grid steps by 1 radian; past 4096 it steps by 4 — more than π — and neighbouring inputs land on unrelated points of the circle. At the top of FP16's range, adjacent numbers are 32 radians apart: five full turns. sin in FP16 at large arguments isn't inaccurate — it's *meaningless*, and the ML world knows it: the sines in transformer positional encodings are only ever fed small, pre-scaled arguments.

§ 9 WHAT THE TEA LEAVES SAY

Back to our two Excels, which now read like a core sample through everything above. The control probes agree because ordinary inputs hide the politics of the last bit. The gold probes disagree with *the truth* — identically on both machines — because Excel's EXP and LN still run an 80-bit, double-rounding x87 algorithm from 1980, and that algorithm has a signature. The red probes disagree *between machines* because two vendors wrote that microcode independently under a standard that only demanded closeness. Nothing is broken. Everything is explained. And it took raw bit patterns to see any of it: at fifteen displayed digits, both machines look serenely identical.

16 Lineage note for the historically inclined: hardware sin/cos for graphics is CORDIC's old stomping ground too — the algorithm was invented in 1959 to aim a bomber's navigation system, and has never once been out of a job.

17 Double exp holds out to $x \approx 709.78$ — Excel's `=EXP(710)` returns #NUM! for exactly this reason. Overflow thresholds are load-bearing trivia.

IF HE REMEMBERS THREE THINGS OVER THE SECOND CUP

1. **Transcendentals are computed by treaty, not by theorem.** IEEE 754 mandates the last bit of ×, ÷, √ — and only *recommends* it for exp, ln, sin. Faithful (either neighbour) is the world's default; every library and vendor chooses its neighbours differently, and all of them are "right."
2. **The machinery is three moves:** fold the argument into a tiny window (the hard, dangerous part — ask sin about π), evaluate a minimax polynomial there (never Taylor — same cost, ~32× worse), then reconstruct the scale by editing exponent bits (free). Microcode, glibc, CUDA: same play, different theatres.
3. **A last bit is harmless until something discrete touches it** — an IF, a barrier, a chaotic recurrence eating one bit per iteration, a checksum. If your work needs bit-identical answers across machines, that is a property you must *engineer* (one library, one code path — the Java StrictMath move), never one you may assume.

There is something quietly wonderful about the whole affair. A number leaves a spreadsheet cell, is folded through a thousand-bit shadow of π or a ladder of ln 2, runs down a polynomial tuned by an exchange algorithm older than the computer, has an integer stitched into its exponent field, and arrives — fifteen digits certain, the sixteenth a matter of vendor temperament and forty years of compatibility promises. Your friend's i7 and our Ryzen aren't disagreeing. They're each keeping a different, perfectly honourable, 1980s promise.

FURTHER STEEPING

· J.-M. Muller, *Elementary Functions: Algorithms and Implementation* — the book on all of this.
· D. Goldberg, "What Every Computer Scientist Should Know About Floating-Point Arithmetic" (1991).
· B. Dawson's randomascii blog — the FSIN exposé and much else.
· core-math.org — correctly rounded, and fast, today.
· Agner Fog, agner.org/optimize — where the cycle counts live.

Set on cream in Palatino, margins in the spirit of E. R. Tufte. Every curve, dot, ripple and waffle cell in this document is real computed data — nothing is an artist's impression. Errors of history are the author's; errors past the 53rd bit are the vendor's. July 2026.