Precision, Precisely: From Theory to Computation in High-Order Calculations

How precise are you ready to go?

Bakar Chargeishvili[1] Vitaly Magerya[2]

Wigner Research Centre for Physics, Budapest

September 25, 2026

[1]KIT, Institut für Theoretische Physik

[2]CERN

Motivation: $t\bar{t}H$ at NNLO

$pp \to t\bar{t}H$: Only direct measurement of the top-Yukawa coupling y_t

Deviation from SM prediction $\Rightarrow$ New Physics

↓

Current (LHC Run 2)
Experimental: $\sim$20%
Theory (NLO): $\sim$10–15%

$\Rightarrow$

HL-LHC (3000 fb^{-1})
Experimental: $\sim$2–3%
Theory (NNLO): needed!

↓

The Challenge: $2 \to 3$ at two loops with $m_t \neq 0$, $m_H \neq 0$

Millions of diagrams • Multiple mass scales • Extreme precision needed

$\Rightarrow$ **Flagship project at current computational limits**

Calculation Strategy

Calculating virtual amplitudes

Usual multi-loop amplitude calculation pipeline:

- Feynman diagrams: $\mathscr{M} =$  $+ \ldots$

  [QGRAF, FEYNARTS, GRC, etc]

- Feynman integrals: $\mathscr{M} = C_1$  $+ C_2$  $+ C_3$  $+ C_4$  $+ \ldots$

  [FEYNCALC, ALIBRARY, but mostly private code]

- IBP reduction: $\mathscr{M} = C_1'$  $+ C_2'$  $+ C_3'$  $+ \ldots$

  [KIRA, FIRE, FINITEFLOW, CARAVEL, RATRACER, NEATIBP, BLADE, LITERED, FORCER, REDUZE, AIR, FMFT, MATAD, etc]

  - Major bottleneck. First steps to GPU usage. [De Laurentis et al '25; Smirnov et al '25]

- Feynman integral evaluation:  $= ?$

  - Analytic methods. [Weinzierl '22, Smirnov '12, etc]
  - Sector decomposition: pySECDEC, FIESTA, SECTOR_DECOMPOSITION, etc.
  - Numerical differential equations: DIFFEXP, AMFLOW, SeaSyde, LINE, AMPRED, etc.
  - Mellin-Barnes representation: MB, AMBRE, etc.
  - Tropical sampling: FEYNTROP, MOMTROP.

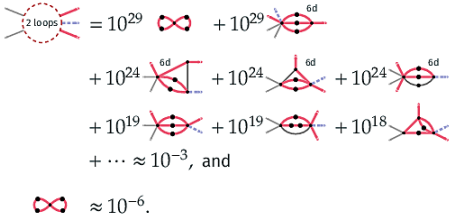
Our focus: *sector decomposition*.

4

Courtesy of V.Magerya

Cancellations between integrals

For $q\bar{q} \to t\bar{t}H$ calculation, we observed cancellations in parts of the high-energy region (above 4 TeV):



- Knowing these integrals at full double precision (16 digits) is not enough!
- The cancelling integrals are simple and converge well (with RQMC).
- Around 30 working precision digits would be needed to obtain these integrals.

Courtesy of V.Magerya

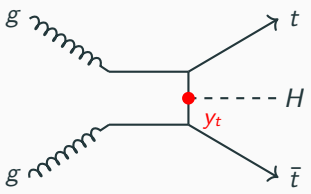
The Computational Challenge

What we compute: $|\mathcal{M}|^2$ for processes like

$$pp \to t\bar{t}H, \quad gg \to ZZ, \quad \text{etc.}$$

Typical structure:

$$|\mathcal{M}|^2 = \sum_{\text{diagrams}} \frac{g_1^4 \cdot P(\hat{s}, \hat{t}, \hat{u})}{(\hat{s} - M_W^2)^2 (\hat{t} - M_Z^2)^2}$$



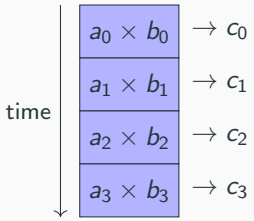
$\Rightarrow$ Bottleneck in MC generators!

Monte Carlo integration:

- 10^6–10^9 phase space points
- 10^{10}–10^{20} evaluations per point
- Each evaluation $\mathcal{O}(10^{40})$ floating-point ops
- Every order beyond LO adds logarithms: $\log(\hat{s}/\mu^2)$, $\log^2(\hat{s}/\mu^2)$ etc.

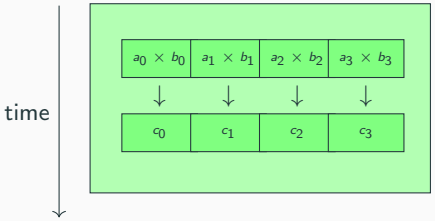
What is Vectorization? (SIMD)

Scalar execution: One operation at a time



4 cycles for 4 results

SIMD execution: Four operations simultaneously

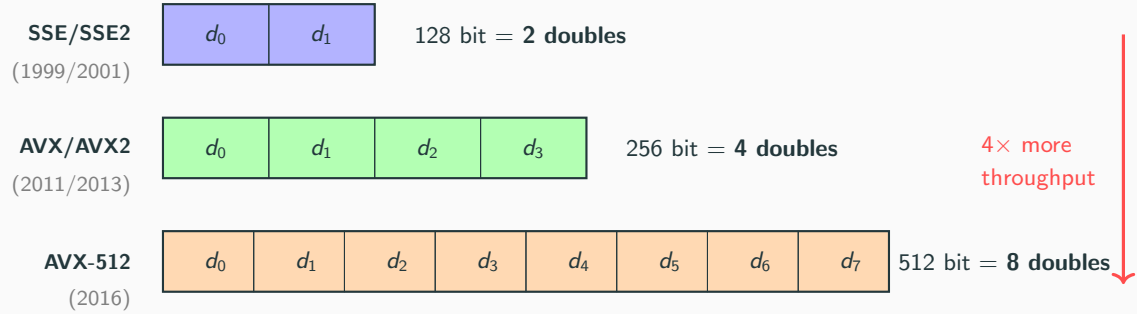


1 cycle for 4 results!

The Physics Analogy

Like computing $\sum_i p_i^\mu p_i^\nu$ — instead of looping over particles one-by-one, process 4 (or 8!) particles in a single instruction. Same idea as helicity sums done in parallel.

Evolution of SIMD: From 128 to 512 Bits



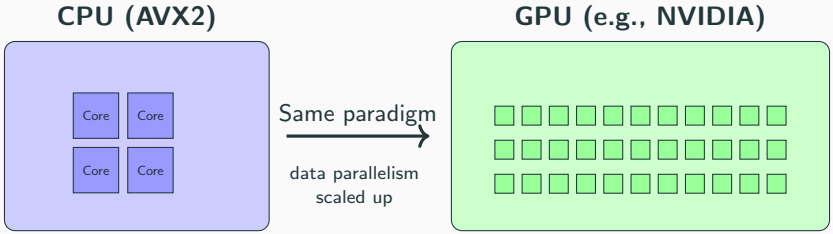
What this means for you:

- Your CPU has 16–32 of these wide registers
- Each holds multiple `doubles`
- One instruction operates on *all* elements
- Free parallelism — if your code allows it!

Catch
Data must be:

- Contiguous in memory
- Independent (no $c_i = f(c_{i-1})$)
- Aligned to 32/64 byte boundaries

Connection to GPUs: Same Idea, Extreme Scale



Key insight:

- SIMD-friendly code ≈ GPU-ready code
- Both require: independent operations on arrays
- If you vectorize for CPU, GPU port is easier
- Think: one phase-space point (or evaluation) per "lane"

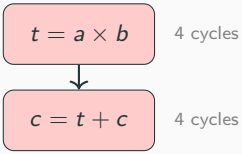
For MC integration

Each GPU thread evaluates $|\mathcal{M}|^2$ at one phase-space point — exactly like SIMD lanes!

MadGraph, Sherpa, Pepper already exploring this. pySecDec did it long ago ;)

Fused Multiply-Add (FMA): The Workhorse Operation

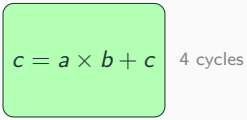
Without FMA:



2 instructions, 8 cycles

(+ rounding twice)

With FMA:



1 instruction, 4 cycles

(+ more accurate!)

Why this matters for $|\mathcal{M}|^2$:

$$\underbrace{g^4 \cdot \frac{1}{(s - M^2)^2}}_{\text{couplings}} \cdot \underbrace{(s^2 + 2st + t^2 + u^2)}_{\text{polynomial in Mandelstams}} \quad \leftarrow \text{Full of } a \times b + c \text{ patterns!}$$

Enable with

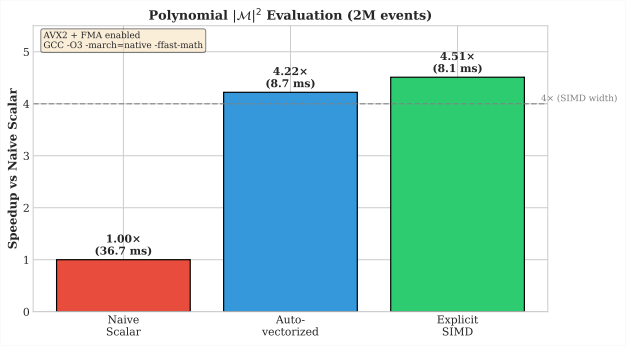
`gcc -mfma` or `-march=native`

Bonus

FMA is *more accurate* than separate mul+add: only one rounding instead of two!

Results: Polynomial $|\mathcal{M}|^2$ (Tree-Level Like)

Setup: simplified $2 \to 2$ matrix element squared — 6 propagators, interference terms, EW/QCD-like polynomial kinematics; 2M phase-space points; AVX2+FMA CPU; `gcc -O3 -march=native`; best of 5 runs.



Method	Time	Speedup
Naive scalar	36.7 ms	1.0×
Auto-vectorized	8.7 ms	4.2×
Explicit SIMD	8.1 ms	4.5×

Key Insight

Auto-vectorization achieves **93%** of hand-tuned SIMD performance!

$> 4\times$ speedup from:

- Function call elimination
- SIMD parallelism

Same Loop, Two Ways of Writing It

```
double eval_msq(double s, double t,
                double u, ...) {
  double prop_s = 1.0/(s - m1sq);
  // ... propagators, kinematics,
  //     interference ... ~30 lines
  return (color + interf) * flux;
}

// hot loop: one call per event
for (i = 0; i < 2000000; i++)
  result[i] = eval_msq(s[i], t[i],
                       u[i], ...);
```

Naive: one call per event

- ✗ call overhead: ∼20–30 cycles
- ✗ compiler can't see across the call ⇒ no vectorization

```
void eval(double *restrict s,
          double *restrict t,
          double *restrict res, size_t n) {
  #pragma omp simd  // safe to vectorize
  for (i = 0; i < n; i++) {
    double prop_s = 1.0/(s[i]-m1sq[i]);
    // ... all computation inline ...
    res[i] = (color + interf) * flux;
  }
}
```

Vectorization-friendly

- ✓ `restrict`: promises no aliasing
- ✓ `#pragma omp simd`: vectorize hint
- ✓ aligned memory: `aligned_alloc(64, ...)`
- ✓ everything visible: auto-inlining

The Hard Way: Explicit SIMD Intrinsics

```
__m256d v_one = _mm256_set1_pd(1.0);

for (size_t i = 0; i < n; i += 4) {  // Process 4 events at once
    __m256d si = _mm256_load_pd(&ps->s[i]);
    __m256d ti = _mm256_load_pd(&ps->t[i]);
    __m256d m1 = _mm256_load_pd(&ps->m1sq[i]);

    // Propagator: 1.0 / (s - m1)
    __m256d prop_s = _mm256_div_pd(v_one, _mm256_sub_pd(si, m1));

    // FMA: a*b + c in one instruction
    __m256d kin = _mm256_fmadd_pd(v_two, _mm256_mul_pd(si,ti),
                  _mm256_add_pd(s2, _mm256_add_pd(t2, u2)));
    // ... 50+ more lines of intrinsics ...

    _mm256_store_pd(&ps->result[i], final);
}
```

Hand-vectorized with AVX2 intrinsics

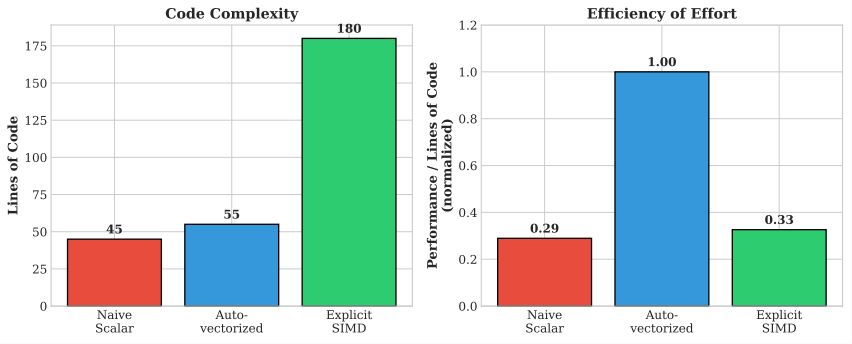
Downsides

- ✗ 3× more code
- ✗ Hardware-specific
- ✗ Easy to make mistakes
- ✗ Harder to maintain

Upsides

- ✓ Full control
- ✓ Predictable performance
- ✓ Can beat compiler (sometimes)

Code Complexity Comparison



The Trade-off

7% more performance vs **3× more code complexity**

Is it worth it? Usually not.

Practical Recommendations for Vectorization

DO

- Use `restrict` on pointers
- Align data to 32/64 bytes
- Keep loop bodies simple
- Use `#pragma omp simd`
- Compile with `-O3 -march=native`
- Add `-ffast-math` if precision allows
- Check compiler output: `-fopt-info-vec`

DON'T

- Put function calls in hot loops
- Use complex control flow
- Mix data types carelessly
- Assume intrinsics are faster
- Fight the compiler blindly

Golden Rule
Write clean code, let the compiler optimize.
Hand-tune only after profiling!

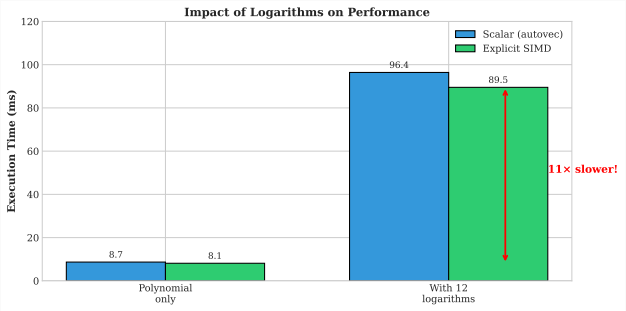
Interlude: The Logarithm Problem

(N$^{(n)}$)NLO calculations need logarithms

Virtual: $\frac{\alpha_s}{\pi}\left[-2\left(\log^2\frac{\hat{s}}{\mu^2}+\log^2\frac{\hat{t}}{\mu^2}\right)+\cdots\right]$

And logarithms destroy vectorization.

Results: Adding Logarithms (NLO-Like)



Method	Time	vs base
Poly only (SIMD)	8.1 ms	1.0×
+ Logs (scalar)	96.4 ms	11.9×
+ Logs (SIMD)	89.5 ms	11.0×

The Disaster

- 12 logarithms per event
- **11× slowdown!**
- SIMD advantage: only 1.08×

SIMD with logs is barely faster than scalar!

Why `log()` Kills Vectorization

The Problem:

`libm` provides only **scalar** `log()`

SIMD code must:

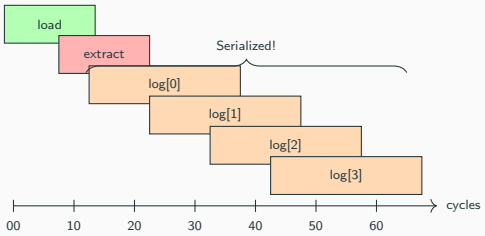
1. Extract 4 values from vector
2. Call scalar `log()` 4 times
3. Pack results back into vector

```
// What actually happens:
__m256d x = _mm256_load_pd(&data[i]);
double tmp[4];
_mm256_store_pd(tmp, x);

tmp[0] = log(tmp[0]);  // Scalar!
tmp[1] = log(tmp[1]);  // Scalar!
tmp[2] = log(tmp[2]);  // Scalar!
tmp[3] = log(tmp[3]);  // Scalar!

x = _mm256_load_pd(tmp);  // Repack
```

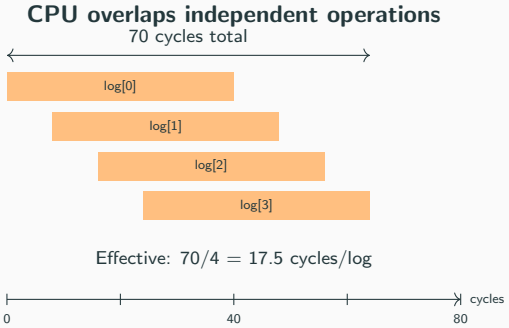
Pipeline stalls everywhere



Cost breakdown:

- Each `log()`: ∼40–60 cycles
- Extract/repack: ∼10 cycles
- vs FMA: ∼4 cycles latency!

The Saving Grace: Out-of-Order Execution



How it works:

The 4 log calls are *independent*!

Modern CPUs:

- Detect independence
- Issue multiple operations
- Execute in parallel
- Reorder results

Result

40 cycles per log becomes

$\sim$**17 cycles** effective

Still bad, but not catastrophic!

Solutions for Logarithm-Heavy Code

1. Vectorized Math Libraries

- **Intel SVML** (proprietary)
- **AMD AOCL-LibM** (free)
- **Sleef** (open source)

Provide true vectorized transcendentals:

```
_mm256_log_pd(x)
```

4 logs in ∼50 cycles total!

2. Polynomial Approximations
Trade accuracy for speed:

$$\log(x) \approx 2\frac{x-1}{x+1}\left(1 + \frac{(x-1)^2}{3(x+1)^2} + \cdots\right)$$

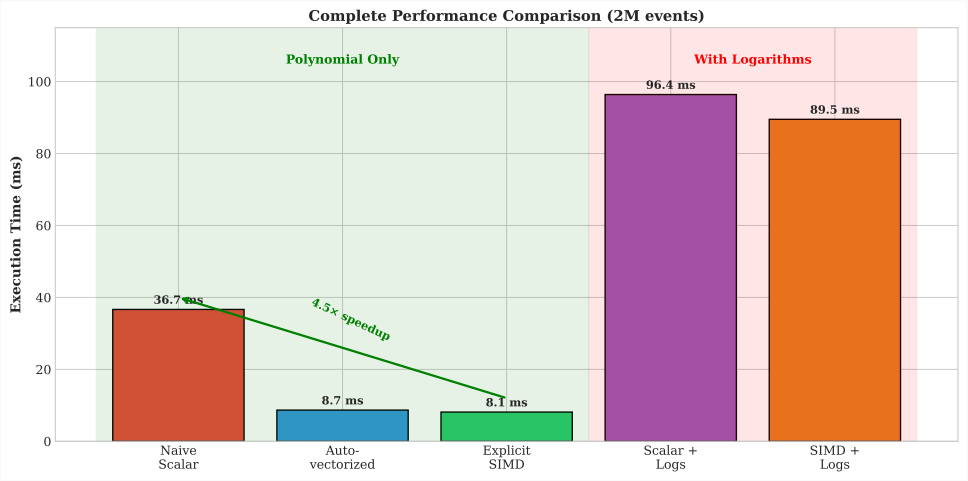
3. Algorithmic Changes

- Cache/precompute logs where possible
- Compute log ratios: $\log(a/b) = \log a - \log b$
- Batch computations
- Use lookup tables + interpolation

Best Practice
For MC generators:

1. Profile first!
2. Try libraries
3. Derive your approx as last resort

Performance Summary



Key Takeaways

Vectorization

- ✓ **4–5**× speedup achievable
- ✓ Auto-vectorization works well
- ✓ Intrinsics rarely worth the effort
- ✓ Write clean, simple loops
- ✓ Use `-O3 -march=native -ffast-math`

Logarithms

- × Standard `log()` is scalar-only
- × **10–15**× slowdown in NLO code
- × SIMD gains largely lost
- ✓ OoO execution helps somewhat
- ✓ Vectorized math libs are the fix

Don't fight the compiler. Help it help you.

Profile before optimizing!

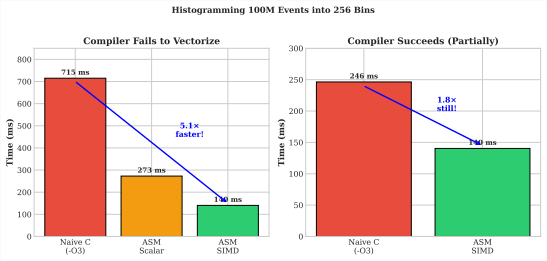
Case Study: Histogramming — Where the Compiler Fails

The beautifully simple code:

```
for (i = 0; i < n; i++) {
  int bin = (int)((E[i]-E_MIN)*scale);
  if (bin < 0) bin = 0;              // clamp
  if (bin >= N_BINS) bin = N_BINS-1;
  bins[bin]++;     // <-- the problem!
}
```

8 lanes may hit the *same* bin ⇒ random memory access, order matters ⇒ fundamentally serial, per element.

Fix: extract all 8 indices to *different* registers first (`vpextrd`), then let out-of-order execution overlap the 8 scattered increments.



100M events, 256 bins

Observations

- When vectorization misses: **5**× on the table
- When it hits: still **1.8**× left for hand-tuning
- 50 lines of x86 assembly bought it

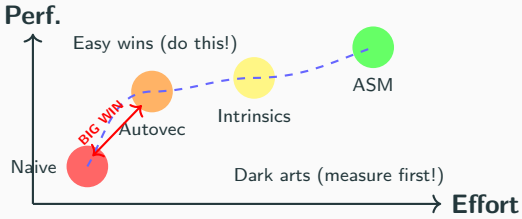
Hand Optimization: Worth It? Rarely — But Sometimes Yes

Maybe worth it

- Histogramming / binning, scatter/gather
- Hot inner loops (>50% of runtime)
- Production code running for years
- When the profiler says "this is it"

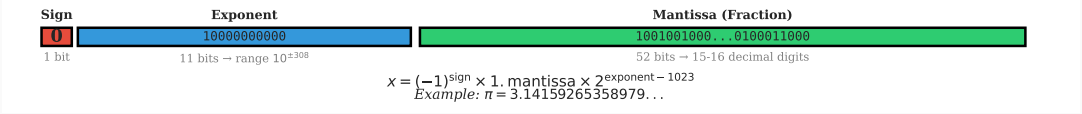
Usually not

- General matrix-element code
- Code that changes frequently
- Multi-platform deployment
- When the compiler does fine (check first!)



The hierarchy: clean C + flags → OpenMP SIMD → intrinsics → assembly (last resort).

Floating Point: The Physicist's Workhorse



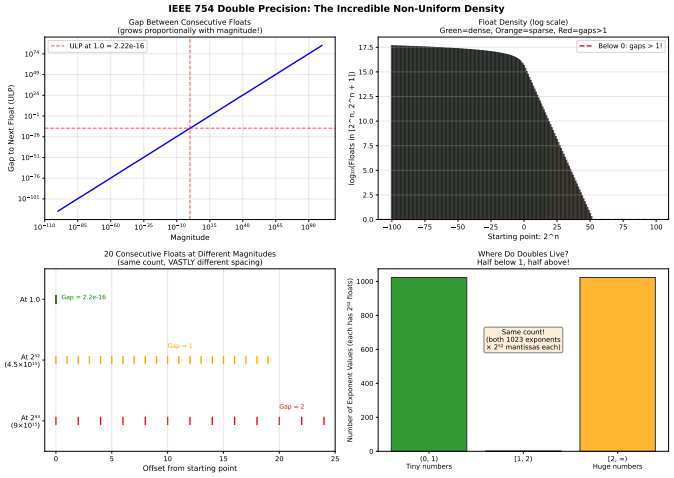
What you get

- **15–16 significant digits**
- Dynamic range: 10^{-308} to 10^{308}
- Relative precision: $\epsilon \approx 2.2 \times 10^{-16}$
- **Hardware speed** (same as integer!)

The magic

- Precision is *relative*, not absolute
- Works from quark masses to cosmological scales
- Same relative error whether $x = 10^{-10}$ or 10^{10}

The Catch: Non-Uniform Distribution

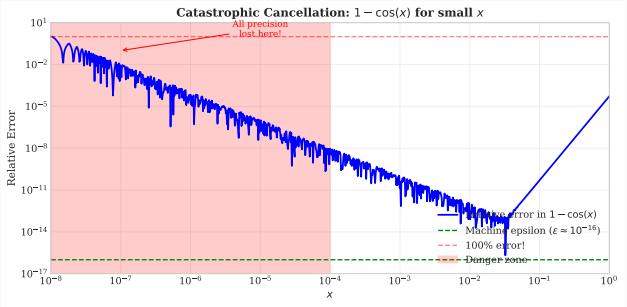


Key insight

Floating point numbers are **logarithmically distributed**:

- Half of all representable numbers are between -1 and $+1$
- Gap between neighbors grows with magnitude: $\Delta x \approx \epsilon \cdot |x|$
- Adding a small number to a large one? The small one may vanish!

Danger #1: Catastrophic Cancellation



What happens:
Subtracting nearly equal numbers destroys precision!

Example: $1 - \cos(x)$ for small x

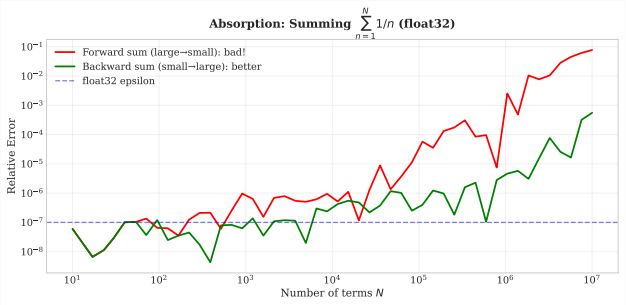
$$\cos(0.001) = 0.999999500...$$
$$1 - \cos = 0.000000500...$$

Started with 16 digits, ended with only 6!

In physics
Common in:

- Gram determinants
- ΔR calculations
- Deep kinematic limits (soft/collinear)

Danger #2: Absorption



What happens:

Adding small numbers to large ones — the small contributions get "absorbed"

The classic:

$$10^{16} + 1 - 10^{16} = 0 \quad (!)$$

In double precision: 1 is below the last digit of 10^{16}

Fix: Kahan Summation

- Track running error
- Add it back next iteration
- Cost: 4× more FLOPs

Danger #3: Non-Associativity

$$(a + b) + c \neq a + (b + c)$$

```c
double a = 1e16;
double b = -1e16;
double c = 1.0;

printf("(a+b)+c = %f\n", (a+b)+c);   // Prints: 1.000000
printf("a+(b+c) = %f\n", a+(b+c));   // Prints: 0.000000
```

Implications

- `-ffast-math` breaks this
- Parallel reductions not reproducible
- Different compilers → different results

What to do

- Understand your numerical stability
- Be careful with "optimizing" compilers
- Test sensitivity to ordering

The Golden Rules of Floating Point

Never do

- **Compare with ==**

```
if (x == 0.3) // NEVER!
```

- **Subtract similar magnitudes**

```
y = sqrt(x+1) - sqrt(x);  // BAD
```

- **Sum many small to one large**

```
for(i=0; i<N; i++)
  total += tiny[i];  // Danger
```

- **Trust compiler “optimizations”**

Best practices

- **Use relative tolerance**

```
if (fabs(x-y) < eps*fabs(x))
```

- **Reformulate expressions**

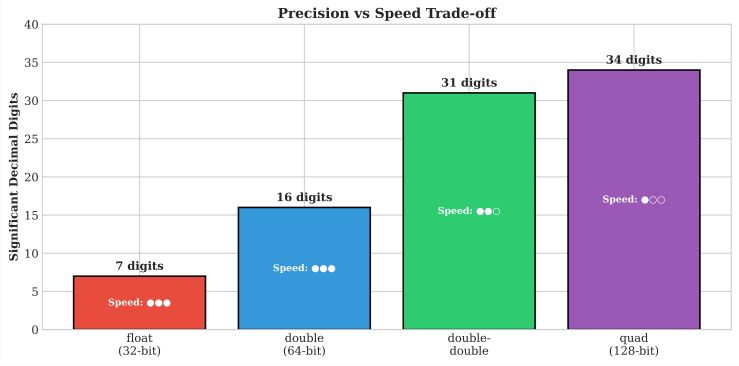
```
y = 1.0/(sqrt(x+1)+sqrt(x));
```

- **Sort before summing**

```
sort(tiny); // smallest first
for(...) total += tiny[i];
```

- **Use Kahan summation**
- **Understand your algorithm**

When Double Isn't Enough: Extended Precision

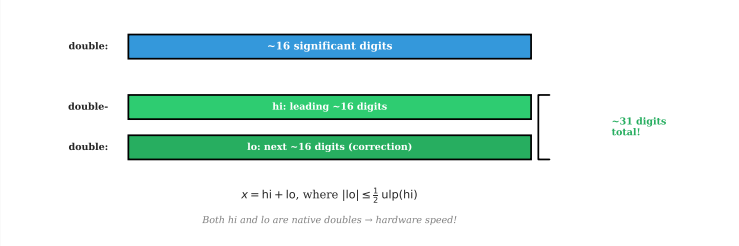


When you need more:

- Multi-loop integrals with near-singular kinematics
- Gram determinant evaluations
- High-precision constants / cross-checks

But `__float128` (quad) is 10–100× slower than double!

Double-Double: The Sweet Spot



The idea
Represent one number as sum of two doubles:

$$x = x_{\text{hi}} + x_{\text{lo}}$$

where x_{hi} holds leading bits, x_{lo} holds the rest.

Result: $\sim$31 decimal digits using native double hardware!

Why not quad?

	DD	Quad
Digits	31	34
Speed	$\sim$10$\times$ slower	$\sim$50$\times$ slower
Hardware	Native FPU	Software emu

Double-double gives 90% of the precision at 20% of the cost!

Double-Double: How It Works

The key primitive — Two-Sum: Given a, b, compute $s = a + b$ *exactly* as:

$s = \text{hi} + \text{lo}$, lo holds the exact rounding error

```
void two_sum(double a, double b,
             double *hi, double *lo) {
    double s = a + b;
    double v = s - a;
    *hi = s;
    *lo = (a - (s - v)) + (b - v);
}
```

No precision lost! The "error" goes into `lo`.

Similarly: Two-Product for $a \times b$

```
// Uses FMA for exact error
*hi = a * b;
*lo = fma(a, b, -(*hi));
```

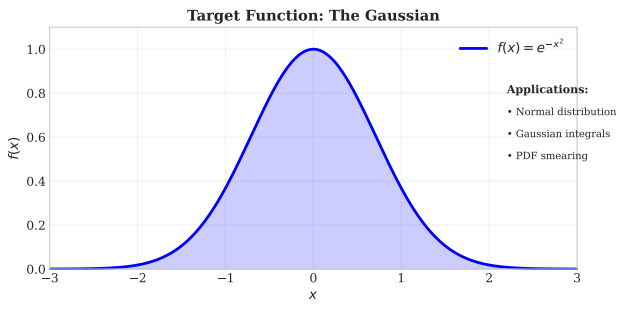
Building operations:
DD addition: ∼10 double ops
DD multiplication: ∼9 double ops
DD division: ∼30 double ops

The Catch

- **No auto SIMD support** — must do by hand
- **No GPU support** — no native DD libs
- **Limited library support**
- **Tricky edge cases** — inf, NaN handling
- **Compiler can break it** — beware `-ffast-math`!

Ongoing effort in pySecDec

Function Approximation: The Problem



The situation:

You need to evaluate $f(x) = e^{-x^2}$ millions of times in your MC.

The problem:

- `exp()` is slow (∼50 cycles)
- Or you're on a GPU/FPGA with no `exp`
- Or you need a vectorized version

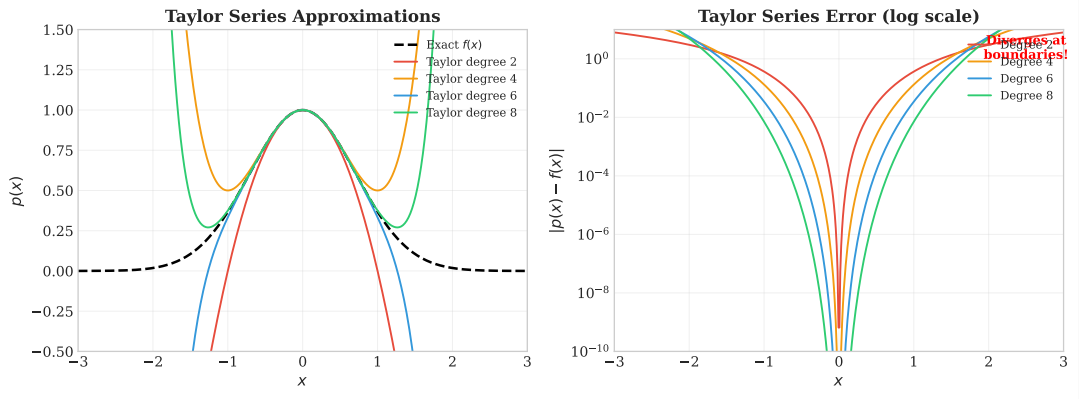
The solution:

Approximate with a polynomial!

$$p(x) = c_0 + c_1 x + c_2 x^2 + \cdots$$

Only needs: `+`, `*`, maybe `FMA`

First Attempt: Taylor Series



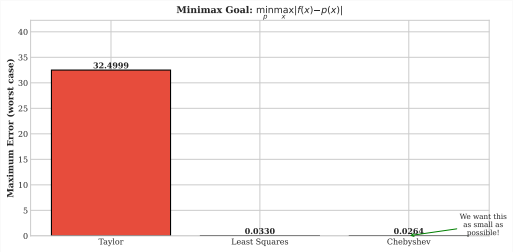
Taylor series for e^{-x^2} around $x = 0$:

$$p(x) = 1 - x^2 + \frac{x^4}{2!} - \frac{x^6}{3!} + \frac{x^8}{4!} - \cdots$$

The problem

Taylor optimizes at **one point** ($x = 0$), not over a **range**!

The Right Goal: Minimax Approximation



What we actually want:

Find $p(x)$ that minimizes the worst-case error:

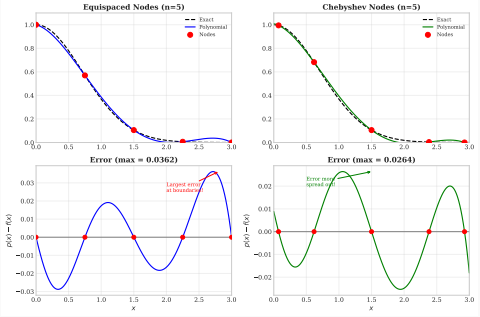
$$\min_{p} \max_{x \in [a,b]} |f(x) - p(x)|$$

Why worst-case?

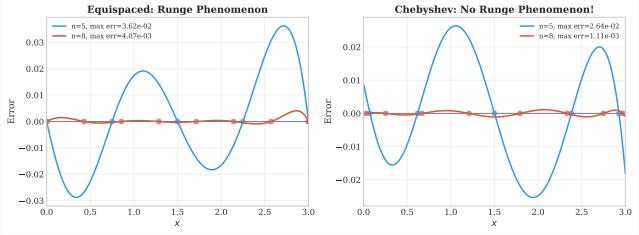
- "Accurate to 12 digits" = hard guarantee
- Least-squares minimizes average, not max
- One bad point ruins your physics!

Pro tip: Exploit symmetry, reduce the range! $f(-x) = f(x)$ means we only need $[0, 3]$.

Node Placement Matters: Chebyshev Prevents Runge



Equispaced nodes: simple, but error concentrates at the boundaries

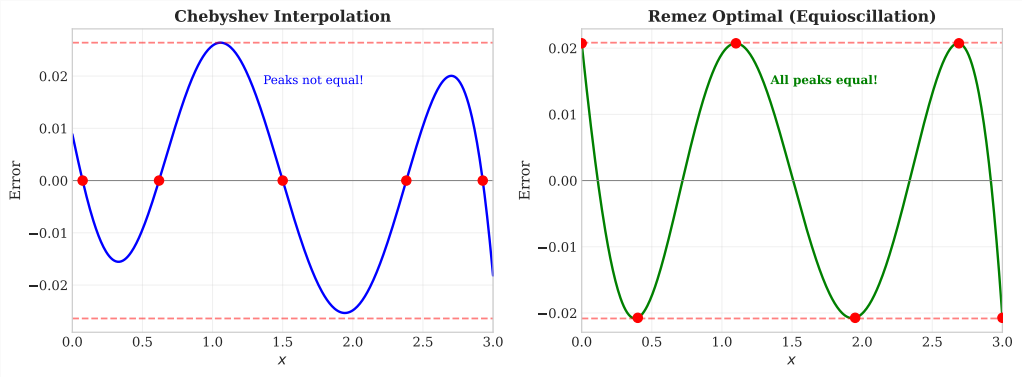


Chebyshev nodes $x_i = \frac{a+b}{2} + \frac{b-a}{2}\cos\frac{(2i+1)\pi}{2n}$: clustered at the edges $\Rightarrow$ error spreads out

Key insight

With **equispaced** nodes, adding more points makes the boundaries *worse* (Runge). With **Chebyshev** nodes, more points $\Rightarrow$ uniformly better approximation.

The Equioscillation Theorem



Theorem (Chebyshev, 1850s)
An n-term polynomial $p(x)$ is the **optimal** minimax approximation if and only if the error $f(x) - p(x)$ oscillates between $+E$ and $-E$ at least $n + 1$ times.

Intuition: If error is uneven, we could "redistribute" it and reduce the max!

Remez Algorithm: The Key Idea

Insight #1: Instead of forcing $p(x_i) = f(x_i)$, allow controlled error:

$$p(x_i) = f(x_i) \pm E$$

with $\pm$ alternating signs. This gives $n + 2$ equations for $n + 1$ coefficients + E.

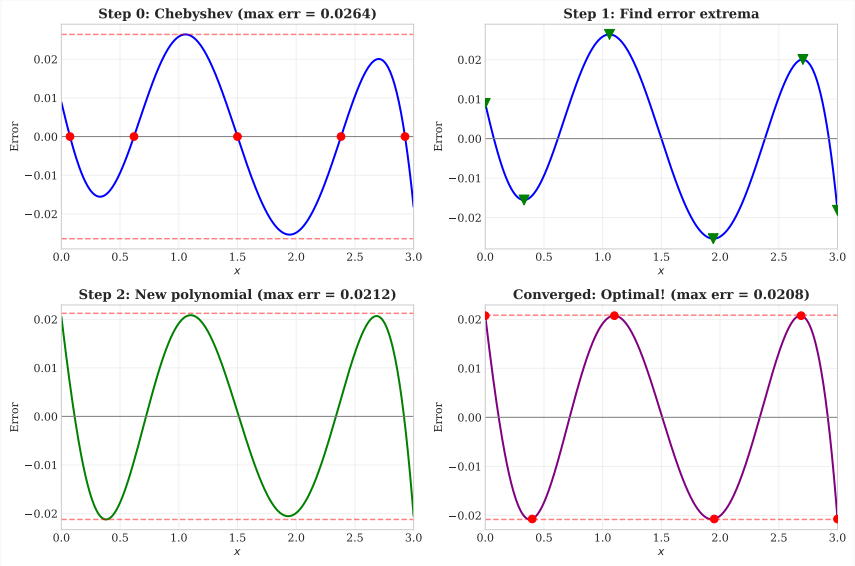
Insight #2: If you start with approximate extrema, solving gives better ones!

Algorithm

1. Start with Chebyshev nodes (good initial guess)
2. Solve linear system for coefficients + error E
3. Find actual error extrema of resulting polynomial
4. Use those as new nodes
5. Repeat until E converges (usually 3–5 iterations)

Named after Evgeny Yakovlevich Remez, Kyiv University

The Remez Algorithm (1934)



Modern Tools: Sollya's `fpminimax`

Remez assumes exact arithmetic. But we have floating point!

The problem:
Optimal math coefficients $\neq$ optimal FP coefficients!

Rounding the coefficients to `double` adds error that Remez doesn't see.

Sollya's `fpminimax`:

- Accounts for coefficient rounding
- Optimizes in floating-point space
- Gives truly optimal FP polynomial

```
> fpminimax(exp(-x^2), 5, [|D...|],
            [0;3], floating);
```

Limitation
`fpminimax` models standard polynomial evaluation:

$$c_0 + c_1 x + c_2 x^2 + \cdots$$

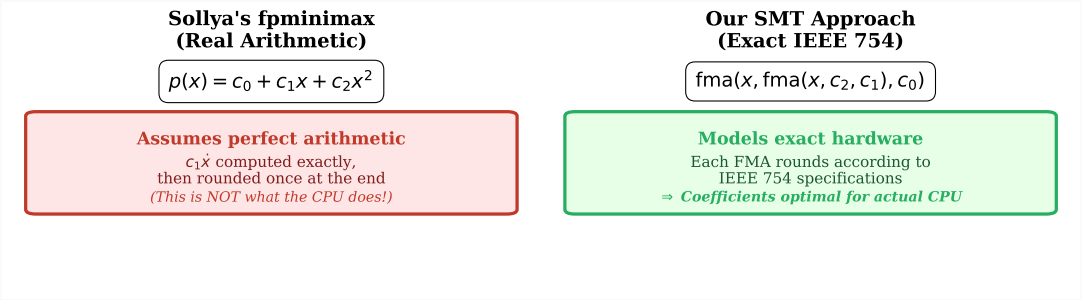
It **cannot** account for:

- FMA instructions
- Horner's method rounding
- Hardware-specific effects

We'll see an alternative approach later!

`https://sollya.org/` — Free, powerful, essential tool

The Problem with Traditional Tools



The Gap
Sollya optimizes for $\mathbb{R}$, but your CPU computes in $\mathbb{F}_{64}$!

Every intermediate result gets rounded. The "optimal" real coefficients are not optimal for floating-point.

Our Solution: Model the Actual Hardware

Key insight: Use an SMT solver to model exact IEEE 754 semantics.

What we want to find:

Coefficients $c_0, c_1, \ldots, c_n$ that minimize

$$\max_{u \in [a,b]} |P_{\text{FP}}(u) - f(u)|$$

where P_{FP} is the *actual floating-point evaluation*:

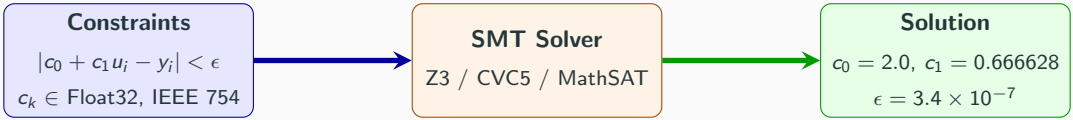
$$P_{\text{FP}}(u) = \text{fma}(u, \text{fma}(u, c_2, c_1), c_0)$$

Why this is hard:

- Non-linear optimization
- Discontinuous (rounding!)
- 2^{64} possible inputs
- Different optima for FMA vs MUL+ADD

Our weapon: SMT solvers can handle exact FP arithmetic!

What is an SMT Solver? Think "Smart Constraint Solver"



You give it:

- Variables to find
- Constraints they must satisfy
- Objective to optimize

It returns:

- SAT + concrete values, or
- UNSAT (provably impossible)

SMT = SAT + Theories:

- **SAT:** Boolean logic ($x \wedge (y \vee \neg z)$)
- **Theory:** arithmetic over integers, reals, IEEE 754 floats!

Every FP operation in the constraints is modeled exactly, bit for bit.

Popular solvers: Z3 (Microsoft), CVC5, MathSAT — all free!

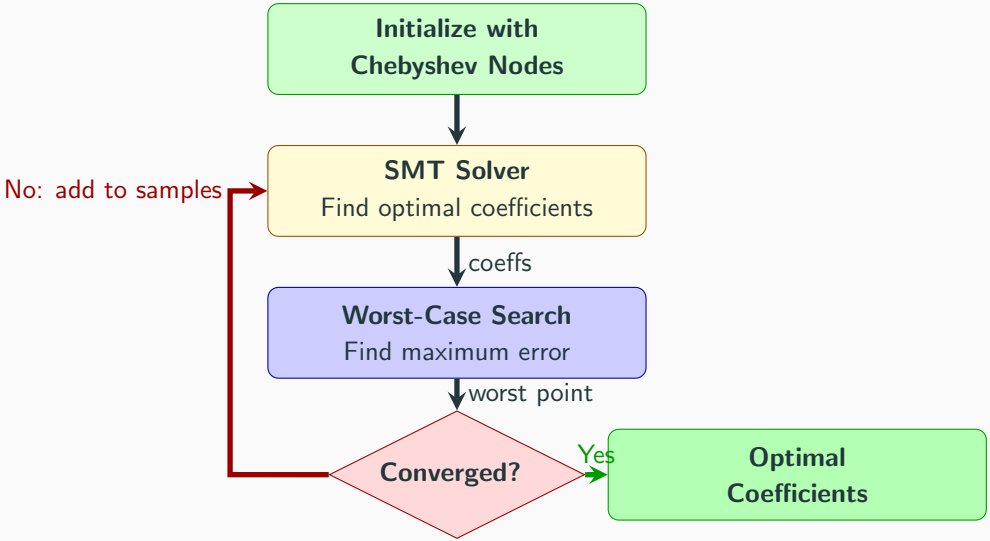
SMT Beyond Floating Point: It Can Even Do Group Theory

Multiplet decomposition for 100 gluons in SU(3):

$$
\begin{aligned}
8^{\otimes 100} &= 33443500918456193511928172773662367301269374339874884516344743861337119538197216896 \otimes 1 \\
&\oplus 257258780658463513006697646375852917026704950659634633269714393804169309523120243200 \otimes 8 \\
&\oplus \cdots \text{ (thousands of irreps later) } \cdots \\
&\oplus 4850 \otimes \overline{999100} \oplus 4850 \otimes 999100 \oplus 9900 \otimes 1000000 \oplus 99 \otimes \overline{1014849} \oplus 99 \otimes 1014849 \oplus 1 \otimes 1030301 \\
&= 203703597633448608626844568840937816105146839366593625063614044935438129976333670618339737 6 \\
&= 2.04 \cdot 10^{90}
\end{aligned}
$$

Decomposition computed with `OrthoBase` — B. Chargeishvili, *Automatic Generation of Orthogonal Multiplet Bases in* $\mathrm{SU}(N_c)$ *Color Space*, arXiv:2404.02443.

The Algorithm for function approximation: SMT + Iterative Search



SMT Encoding Example

Horner with FMA:

$$P(u) = \mathrm{fma}(u, \mathrm{fma}(u, c_2, c_1), c_0)$$

$$\text{Inner} = \mathrm{round}(u \cdot c_2 + c_1)$$
$$\text{Result} = \mathrm{round}(u \cdot \text{ Inner } + c_0)$$

Each round: IEEE 754 rules!

SMT-LIB2 code:

```
; Point 28: s=8.292931e-03, u=6.877269e-05,
    target=2.00004578
(declare-const u28 Float32)
(declare-const y28 Float32)
(declare-const p28 Float32)
(assert (= u28 (fp #b0 #x71 #
    b00100000011101000001111)))
(assert (= y28 (fp #b0 #x80 #
    b00000000000000011000000)))
(assert (= p28 (fp.fma roundNearestTiesToEven u28
     (fp.fma roundNearestTiesToEven u28 c2 c1) c0
    )))
(assert (fp.lt (fp.abs (fp.sub
    roundNearestTiesToEven p28 y28)) eps))
(declare-const c0 Float32)
(declare-const c1 Float32)
(declare-const c2 Float32)
```

Phase 1: SMT Finds Optimal Coefficients

Given: Sample points $\{(u_i, y_i)\}$ where $y_i = f(u_i)$

Find: Coefficients minimizing max error *at those points*

The encoding:

1. Declare coefficient variables
2. For each sample point:
   - Assert exact FP evaluation
   - Assert $|P(u_i) - y_i| < \epsilon$
3. Minimize ϵ

Minimization trick:

Decompose ϵ into IEEE 754 parts:

$$\epsilon = (-1)^s \cdot 2^{e-127} \cdot (1.m)$$

Minimize lexicographically:

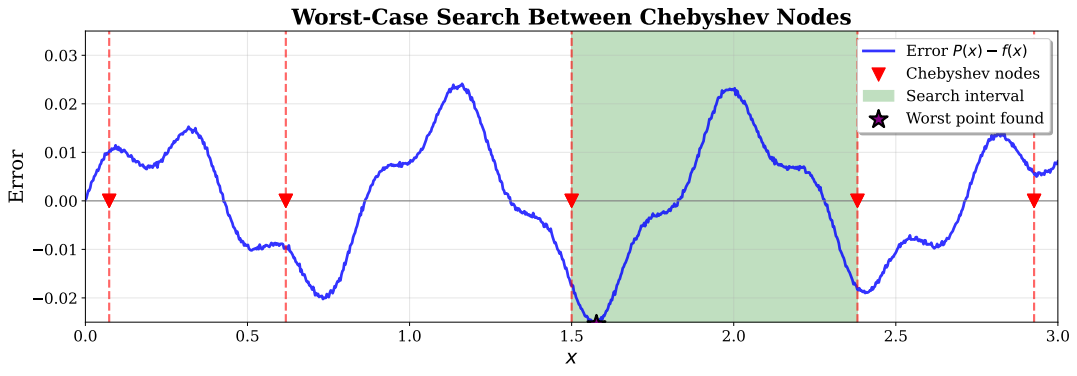
1. First minimize exponent e
2. Then minimize mantissa m

Limitation

SMT only guarantees optimality *at the sample points*!

We need to find where the *actual* maximum error occurs...

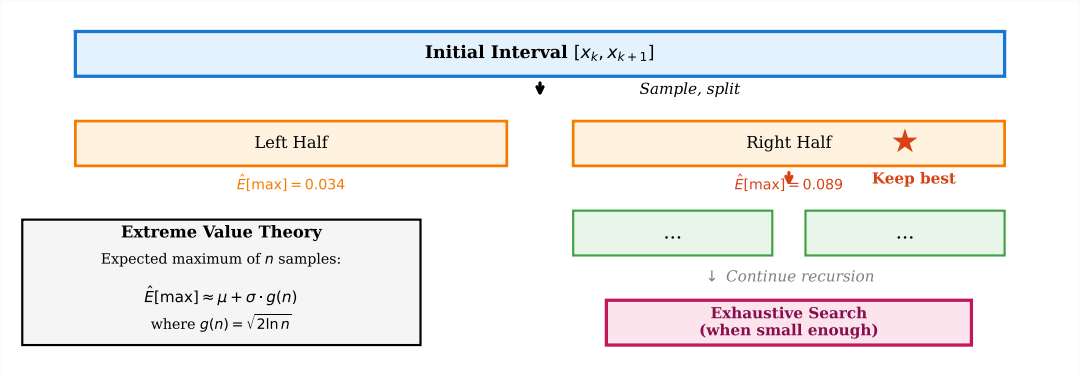
Phase 2: Worst-Case Search



Strategy: Search intervals between Chebyshev nodes (where errors are expected to peak by equioscillation).

Scaling to Float64: Recursive Subdivision

Problem: Can't exhaustively search 2^{64} values!



Search strategy based on: B. Gladman, V. Innocente, J. Mather & P. Zimmermann, *Accuracy of Mathematical Functions in Single, Double, Double Extended, and Quadruple Precision*, LORIA tech. rep. (regularly updated).

Extreme Value Theory: Smart Interval Selection

The idea:

From a few random samples in an interval, estimate the likely maximum error.

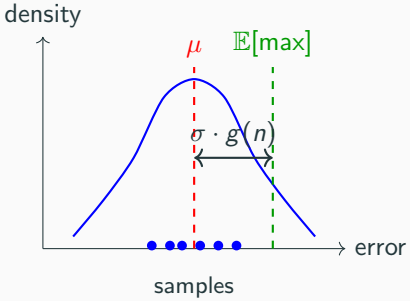
Gumbel approximation:

For n i.i.d. samples with mean μ and std σ:

$$\mathbb{E}[\max] \approx \mu + \sigma \cdot g(n)$$

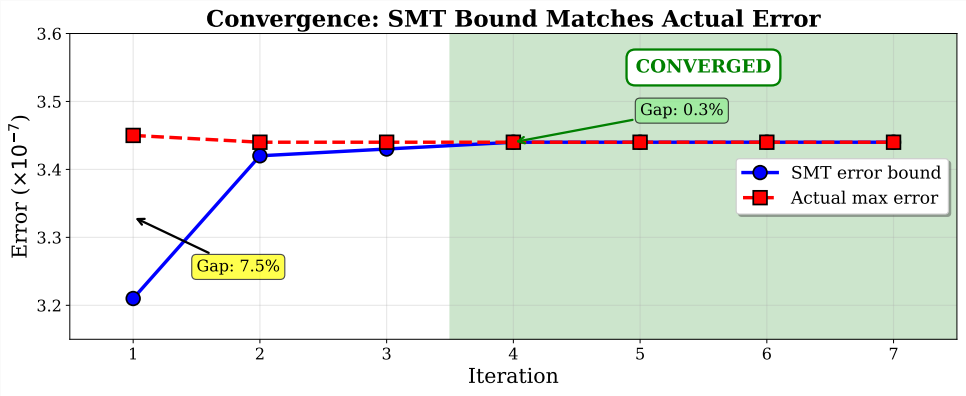
where

$$g(n) = \sqrt{2 \ln n} - \frac{\ln \ln n}{2\sqrt{2 \ln n}}$$



Use: Prioritize intervals with high $\mathbb{E}[\max]$

Convergence: Remez-Like Iteration

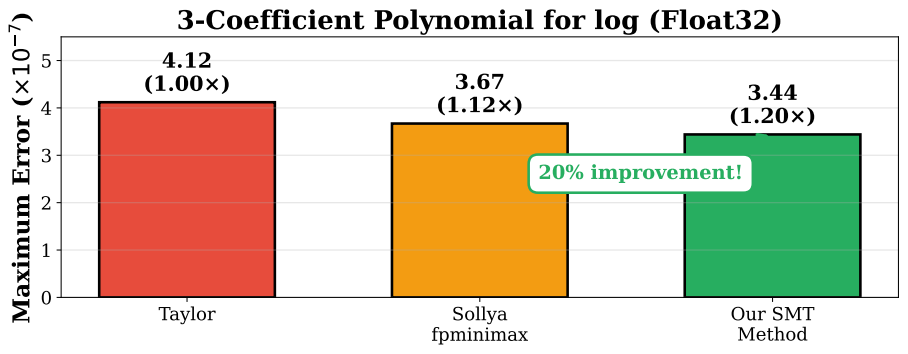


Convergence criterion
Stop when SMT error bound $\approx$ actual max error.

This means we've found *all* the worst points!

Typically converges in 10–30 iterations.

Results: We Beat Sollya!



Why we win:

- Model exact FMA rounding
- Account for Horner evaluation
- Optimize for actual hardware

Improvement varies:

- 5–20% for Float32
- Similar for Float64
- Bigger gains for more coefficients

Scaling Across Precisions

Precision	SMT Theory	Worst-Case Search	Time
Float32	`Float32`	Exhaustive OK	Minutes
Float64	`Float64`	Recursive subdivision	Hours
Double-Double	$2 \times$ `Float64`	Parallel subdivision	Days

Double-Double encoding:

Model $x = x_{\text{hi}} + x_{\text{lo}}$ as two Float64 values.

Explicitly encode `two_sum`, `two_prod` algorithms.

Compute cost
This is an *offline* optimization — run once, use forever!

Even days of compute time is fine for a log/exp implementation.

From Toy to Production: The 106-Bit Building Blocks

Real high-precision calculations rest on a handful of kernels.

How accurate are they, really?

The Setting: Double-Double Arithmetic in Production

Where DD shows up:

- Loop integrals (sector decomposition): cancellation demands $\sim$106 bits
- Code generation emits expression graphs of DD *kernels*:
  `add`, `mul`, `div`, `sqr`, `inv`, `log`, `atan2`, ...
- Each kernel = fixed sequence of FP64/FMA instructions (TwoSum, TwoProd, ...)

The literature provides *proven* worst-case bounds for these building blocks (Joldes–Muller–Popescu '17; corrected and Coq-verified by Muller–Rideau '22), in units of $u^2 = 2^{-106}$.

The questions we ask

1. What error does the code we *actually run* produce — worst case, not average?
2. Do measurements respect the proven bounds?
3. Can we re-engineer: more accuracy at no speed cost?

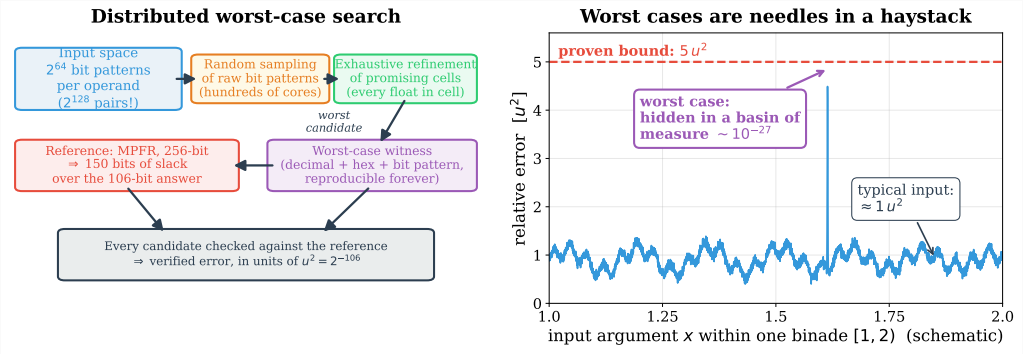
The metric

rel. error in u^2, $u = 2^{-53}$

reference: MPFR at 256 bits

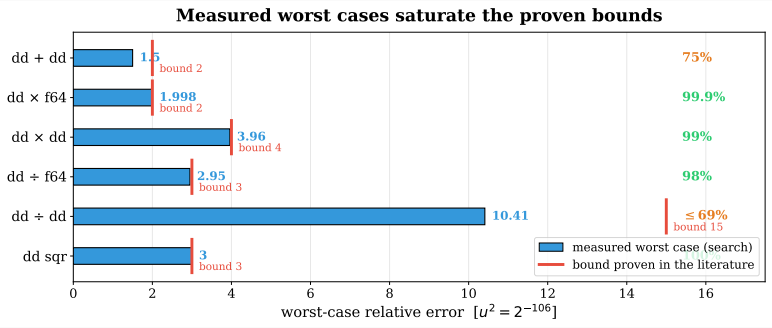
exhaustive: over all 2^{64} inputs

Worst-Case Search at Cluster Scale



- Raw bit patterns as search space: **no distribution assumptions**, no “nice” inputs
- Randomized sampling on hundreds of cores + **exhaustive refinement** of promising cells
- Worst cases hide in exponentially small needles — plain random sampling is not enough
- Every candidate verified against a 256-bit MPFR reference

Result 1: Worst Cases Saturate the Proven Bounds



Validation

- ✅ No measurement exceeds any proven bound
- ✅ Many kernels at 98–100% of the bound — proof and experiment *agree*: the bounds are tight

Discovery

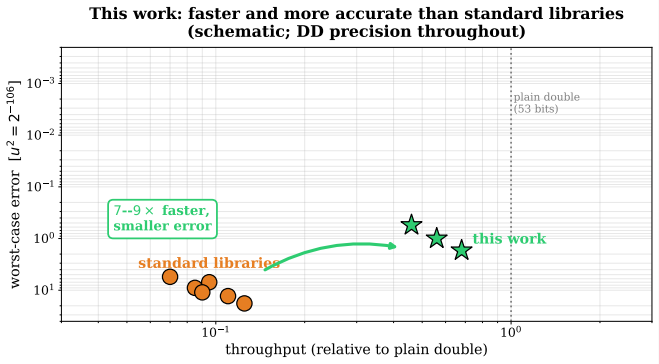
- Some bounds are **loose** (dd÷dd: $\leq$69%) $\Rightarrow$ better algorithms possible
- The search finds worse inputs than the *published* examples!

Result 2: Closing the Loop — Never Slower, Much More Accurate

Re-engineering recipe:

1. Take the kernel's approximation core (its polynomial)
2. Feed the worst points from the search into the SMT optimizer as constraints
3. Re-optimize coefficients for **exact IEEE 754 semantics**
4. Re-run the exhaustive search: verify the improvement

Exactly the loop we demonstrated on the e^{-x^2} toy — now on real kernels.



Outcome

- Re-engineering: up to **15.8× smaller** worst-case error at the same speed (`atan2` is even **2.5% faster**)
- vs. standard libraries: 7–9× faster than the standard DD library, ∼10% faster than the best quad `log` — *and* tighter worst cases

Current Status & Future Directions

What's Working

- ✓ SMT coefficient optimization (beats Sollya's `fpminimax`)
- ✓ Recursive worst-case search with EVT: FP32 → FP64
- ✓ DD kernels at cluster scale: worst cases saturate proven bounds
- ✓ Worst points feed back into the optimizer

Work in Progress

- Re-engineered DD special functions: smaller error, same speed
- Tunable precision/performance trade-offs, AVX-512/CUDA vectorization

Open Research Questions

- **Mixed precision:** which parts of an expression need high precision — *a priori*?
- **FLOP economy:** Double-Double only where needed, Float64 elsewhere
- **Error propagation:** automatic precision assignment from sensitivity

Goal

Right precision at the right place

$\Rightarrow$ Maximum speed, guaranteed accuracy

Summary: What We Covered

I. Vectorization

- SIMD basics: SSE → AVX → AVX-512, FMA
- 4–5× speedup achievable; auto-vectorization often sufficient
- Connection to GPU programming

II. The Logarithm Problem

- `log()` is not vectorized in libm: 10–15× slowdown
- Solutions: vectorized math libraries, custom approximations

III. Hand Optimization

- Histogramming: compiler struggles, 5× available
- Rarely justified — profile first!

IV. Floating Point

- Know the dangers: cancellation, absorption, non-associativity
- Double-double: 31 digits at 10× the cost

V. Function Approximation

- Taylor: wrong tool; Chebyshev/Remez: right one
- Sollya's `fpminimax`: FP-aware, but not FP-exact

VI. SMT-Based Optimization

- Model exact IEEE 754 semantics (FMA, rounding)
- SMT + worst-case search: correct coefficients for actual hardware

VII. Production: DD Special Functions

- Exhaustive cluster search: worst cases saturate the proven bounds
- Re-engineered kernels: never slower, up to 16× more accurate
- vs standard DD libraries: 7–9× faster and more accurate

Moral: reliability is not an afterthought — *precision, precisely.*

To be continued...

Backup: Compiler Flags Cheatsheet

Flag	Effect
`-O3`	Aggressive optimization
`-march=native`	Use all CPU features
`-ffast-math`	Allow FP reordering (careful!)
`-fopenmp-simd`	Enable OpenMP SIMD pragmas
`-fopt-info-vec`	Report vectorization success/failure
`-mfma`	Enable FMA instructions
`-mavx2`	Enable AVX2
`-funroll-loops`	Unroll loops (sometimes helps)

Recommended compilation:

```
gcc -O3 -march=native -ffast-math -fopenmp-simd \
    -fopt-info-vec-optimized -o mycode mycode.c -lm
```

Backup: Cycle Counts Reference

Operation	Latency (cycles)	Throughput (cycles)
ADD/SUB	3–4	0.5
MUL	3–5	0.5
FMA	4–5	0.5
DIV	13–20	4–6
SQRT	15–20	4–6
`log()`	40–80	40–80
`exp()`	40–80	40–80
`sin/cos()`	50–100	50–100

Key insight: Latency = time for one operation. Throughput = how fast you can start new operations. With independent operations, throughput matters more!