# Optimizing the Linearization Constant in Fast Inverse Square Root Using Differential Evolution

Xuanlin Zhu

College of William & Mary
Department of Mathematics

Joint Mathematics Meetings 2026
AMS Special Session on Applied Mathematics for Digital Twins
January 6, 2026

# Outline

# The Hardware Constraint (1999)

**Pentium III Instruction Latencies:**

| Instruction | Cycles |
|---|---|
| `FSQRT` | $\sim$70 |
| `FDIV` | $\sim$40 |
| `FMUL` | $\sim$5 |
| Integer `SUB`, `SHR` | 1 |

**Traditional $1/\sqrt{x}$:**

- `FSQRT` + `FDIV` $\approx$ **110 cycles**

**Fast inverse sqrt:**

- Bit ops + 1 Newton $\approx$ **20 cycles**
- **5$\times$ speedup**

Quake III Arena (1999)

**Bottleneck Analysis (Pentium III, 500 MHz)**

**Load:** $640 \times 480$ px @ 60 fps $\approx$ **18.4 million** ops/sec

**CPU Requirement:**

- Standard (110 cyc): $\sim$ **2.0** GHz (**Impossible**)
- Fast (20 cyc): $\sim$ **368** MHz (**Feasible**)

# Connection to Digital Twins

**FPS Games as Early Simulations:**

- Real-time collision detection
- Projectile physics, light transport
- Constraint: 30+ fps on consumer hardware

**Digital Twin = Simulation + Real Data**

- Often runs on constrained hardware:
  - Embedded controllers (IoT)
  - Edge devices, FPGA soft-cores
  - Custom AI accelerators
- Same 1999 constraint: arithmetic bottleneck

**Why This Matters**

Optimizing the arithmetic substrate of simulations benefits digital twins on resource-constrained platforms.

**This Talk**

Revisit a 1999 algorithm, understand *why* it works, and show how to optimize it automatically.

# The Code

Quake III Arena (1999)

```
float Q_rsqrt(float number) {
  long i;
  float x2, y;
  x2 = number * 0.5F;
  y = number;
  i = *(long *)&y;              // reinterpret bits
  i = 0x5f3759df - (i >> 1); // what the ...?
  y = *(float *)&i;
  y = y * (1.5F - x2*y*y);    // Newton iteration
  return y;
}
```

**Two Components:**

1. **Bit manipulation** (lines 6–8)
   ∼8 cycles, gives rough estimate
2. **Newton-Raphson** (line 9)
   ∼12 cycles, refines accuracy

**Questions:**

- Where does `0x5f3759df` come from?
- Why does bit reinterpretation work?

# The Arithmetic Substrate: IEEE 754

The "Magic" relies on how single-precision floats are packed into 32 bits:

| Sign (1 bit) | Exponent (8 bits) | Mantissa (23 bits) |
|---|---|---|
| $s$ | $e = E + 127$ | $m = M/2^{23}$ |

**The Value Equation:**

$$\text{Value} = (-1)^s \times (1 + m) \times 2^{e-127}$$

- **Sign ($s$):** 0 for positive, 1 for negative.
- **Exponent ($e$):** Biased by 127 to avoid negative integers.
- **Mantissa ($m$):** A fractional value in the range $[0, 1)$.

# Numerical Trace: $x = 2.0$

**Step 1: Reinterpret bits**

$$x = 2.0 \quad \text{(float)}$$
$$I_x = \texttt{0x40000000} \quad \text{(as integer)}$$

**Step 2: Bit manipulation**

$$I_x \gg 1 = \texttt{0x20000000}$$
$$R - (I_x \gg 1) = \texttt{0x5f3759df} - \texttt{0x20000000}$$
$$= \texttt{0x3f3759df}$$

**Step 3: Reinterpret as float**

$$y_0 = 0.7162... \quad \text{(true: } 0.7071...\text{)}$$

**Error: 1.29%**

**Step 4: Newton iteration**

$$y_1 = y_0 \cdot (1.5 - 0.5 \cdot x \cdot y_0^2)$$

$$y_1 = 0.70693...$$

**Error: 0.025%**

**Step 5: Second Newton**

$$y_2 = 0.7071066...$$

**Error: 0.000017%**

**Observation**

Each Newton iteration squares the error (quadratic convergence).

# The Insight: Floats as Fixed-Point Logarithms

**Key Observation:** IEEE 754 bits encode an approximate logarithm.

**IEEE 754 Single Precision:**

$$x = 2^{e-127} \times (1 + m)$$

Taking $\log_2$:

$$\log_2 x = (e - 127) + \log_2(1 + m)$$

Since $\log_2(1 + m) \approx m + \sigma$:

$$\log_2 x \approx e + m - 127 + \sigma$$

**Integer Interpretation:**
The 32-bit pattern as integer:

$$I_x = 2^{23}(e + m)$$

Therefore:

$$\boxed{I_x \approx 2^{23}(\log_2 x + 127 + \sigma)}$$

The integer bits are a **fixed-point logarithm** with implicit bias.

# The Linearization Approximation

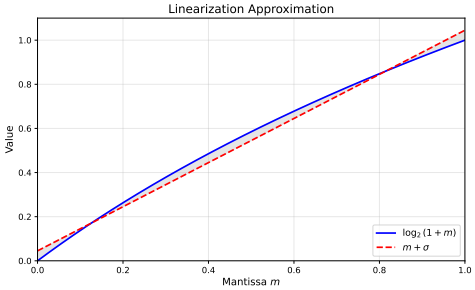
**The Approximation:**

$$\log_2(1+m) \approx m + \sigma$$

where $\sigma \approx 0.0450466$ minimizes max error.

**Quality:**

- Max linearization error: $\sim$4.5%
- This error propagates to the initial guess
- The constant $R$ encodes $\sigma$




Blue: $\log_2(1+m)$. Red: linear approx.

# Deriving the Constant $R$

**Goal:** $y = x^{-1/2}$, so $\log_2 y = -\frac{1}{2}\log_2 x$.

**In the integer domain:**

$$\begin{aligned} I_y &\approx 2^{23}(\log_2 y + 127 + \sigma) \\ &= 2^{23}\left(-\tfrac{1}{2}\log_2 x + 127 + \sigma\right) \\ &= 2^{23}\left(-\tfrac{1}{2}\left(\frac{I_x}{2^{23}} - 127 - \sigma\right) + 127 + \sigma\right) \\ &= \underbrace{\tfrac{3}{2}\cdot 2^{23}(127+\sigma)}_{R} - \tfrac{1}{2}I_x \end{aligned}$$

**Result:**

$$\boxed{I_y = R - (I_x \gg 1)} \quad \text{where} \quad R = \tfrac{3}{2}\cdot 2^{23}(127+\sigma)$$

The constant $R$ encodes the bias (127) and linearization parameter ($\sigma$).

# Newton-Raphson: Convergence Analysis

**The Iteration:**
For $f(y) = y^{-2} - x = 0$:

$$y_{n+1} = y_n \cdot \left(\tfrac{3}{2} - \tfrac{1}{2} x y_n^2\right)$$

**Error Propagation:**
Let $\varepsilon_n = \frac{y_n - y^*}{y^*}$ (relative error). Then:

$$\varepsilon_{n+1} = -\tfrac{3}{2}\varepsilon_n^2(1 + \varepsilon_n/3)$$

For small $\varepsilon_n$:

$$\boxed{|\varepsilon_{n+1}| \approx \tfrac{3}{2}\varepsilon_n^2}$$

**Verified Experimentally:**

| Stage | Max Error |
|---|---|
| Initial (bit hack) | 3.44% |
| After 1 Newton | 0.175% |
| After 2 Newton | $4.7 \times 10^{-4}$% |

Check: $(3/2) \times 0.0344^2 \approx 0.00178$ ✓

**Key Point**

Better initial guess ⇒ smaller $\varepsilon_0$ ⇒ smaller final error.
The constant $R$ determines $\varepsilon_0$.

# A Class of Linear Exponent Transformations

**Observation:** Several fast approximations share the same structure.

| **Function** | $f(x)$ | **Transformation** | **Source** |
|---|---|---|---|
| Inv. Sqrt | $x^{-1/2}$ | $I_y = R - \lfloor I_x/2 \rfloor$ | id Software |
| Inv. Cube Root | $x^{-1/3}$ | $I_y = R - \lfloor I_x/3 \rfloor$ | (extension) |
| Exponential | $2^x$ | $I_y = R + c \cdot I_x$ | Schraudolph |

**General Form:** $I_{\text{out}} = R + \alpha \cdot I_{\text{in}}$

For $y = x^{-1/n}$: $\alpha = -1/n$, and $R = \frac{n+1}{n} \cdot 2^{23}(127 + \sigma)$

## The Point

Individual algorithms are known. The contribution is recognizing them as a **unified class** where the same optimization framework applies.

# Problem Statement

## Objective

Find integer $R^*$ minimizing worst-case relative error:

$$R^* = \arg\min_{R \in \mathbb{Z}} \max_{x \in \mathcal{X}} \left| \frac{y_2(R; x) - x^{-1/2}}{x^{-1/2}} \right|$$

**Challenges:**

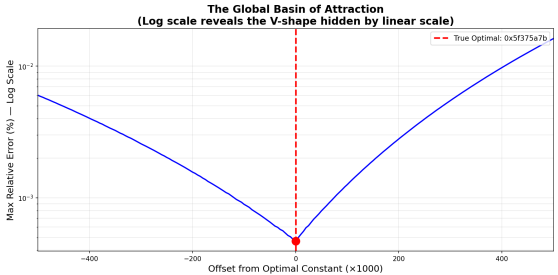
- **Discrete:** $R$ must be a 32-bit integer
- **Non-smooth:** Floor operations break differentiability
- **Rough:** Many local minima at integer resolution

**Previous Approaches:**

- id Software: theoretical derivation + manual tuning
- Lomont (2003): analytical with assumptions

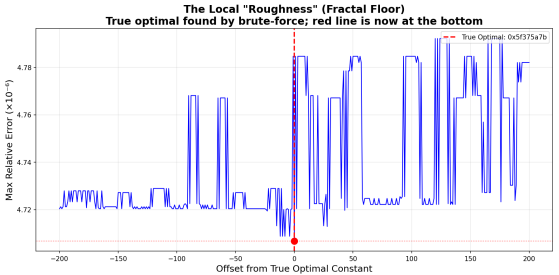
# The Error Landscape: Macro vs. Micro

**Macro Scale (log y-axis):**




Global Funnel Landscape.

**Micro Scale (linear):**




Local roughness—many minima.

## Structure

**Globally Funnel Landscape, locally rough.** Gradient descent fails (no consistent descent direction), but population-based methods can exploit the global structure.

# What is Differential Evolution?

**Background:** Storn & Price (1997)

A population-based, gradient-free optimizer.

- Not new—25+ years old, well-understood
- Simple: mutation, crossover, selection
- Works on non-differentiable objectives

**Why DE for this problem?**

- Not convex
- But globally Funnel Landscape
- DE exploits exactly this structure

## The Intuition

A population "feels" the global basin through averaging, while mutations hop over local roughness.

Think of the population as a **low-pass filter**: it smooths out high-frequency noise while tracking the macro trend.

# The DE Algorithm

**Algorithm 1** Differential Evolution

**Init:** $\{r_1, \ldots, r_N\} \sim \mathcal{U}[L, H]$
**for** generation $g = 1$ to $G_{\max}$ **do**
  **for** each $r_i$ **do**
    Pick 3 distinct: $r_a, r_b, r_c$
    **Mutate:** $v \leftarrow r_a + F(r_b - r_c)$
    **Cross:** $u \leftarrow \text{mix}(v, r_i, C_r)$
    **Round:** $R \leftarrow \lfloor u + 0.5 \rfloor$
    **Select:** if $f(R) < f(r_i)$: $r_i \leftarrow u$
  **end for**
**end for**
**return** best

**Parameters:**

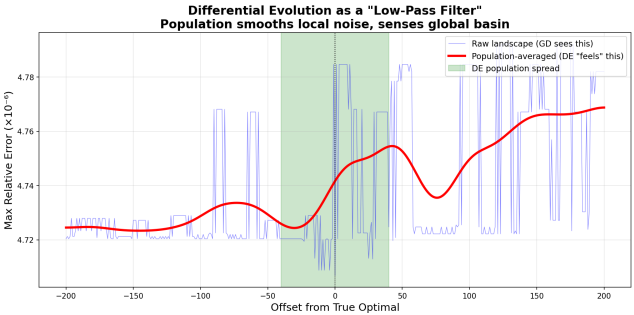
- Population: $N = 15$
- Mutation: $F = 0.5$
- Crossover: $C_r = 0.9$
- Generations: 50

**Cost:**

- $\sim$375 evaluations
- $<$1 second

Compare: brute force on FP32 needs $\sim 2 \times 10^9$ evaluations.

# Why DE Works Here




Blue: raw landscape. Red: population average. Green: spread.

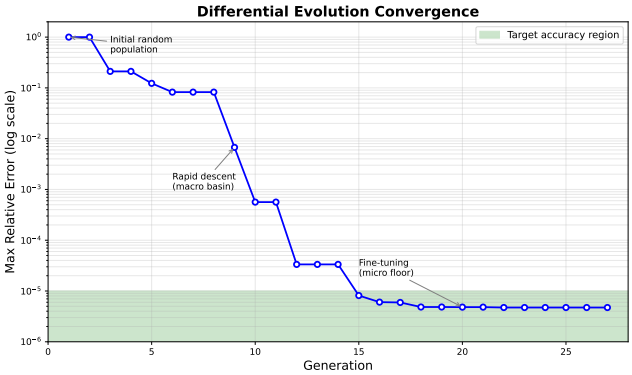
**Mechanism:**

1. **Population spread** covers basin $\Rightarrow$ senses global trend
2. **Mutation** generates trial points that hop over local minima
3. **Selection** follows macro gradient

**Critical:**
Population variance $>$ roughness wavelength

This is why DE works but gradient descent fails.

# Convergence




**Two Phases:**

**Phase 1 (Gen 1–9):**
Rapid descent through basin.
Error: $1 \to 10^{-3}$

**Phase 2 (Gen 10–25):**
Fine-tuning on rough floor.
Error: $\to 4.7 \times 10^{-6}$

**Total:** 375 evaluations

# Theoretical Derivation: The Chebyshev Baseline

**Problem:** Approximating $\log_2(1+x)$ with a line $x+\sigma$ on $[0,1]$ to minimize the $L_\infty$ error.

Let error function $E(x) = \log_2(1+x) - (x+\sigma)$.

**1. Find Critical Point $x^*$:**

$$E'(x) = \frac{1}{(1+x)\ln 2} - 1 = 0 \implies x^* = \frac{1}{\ln 2} - 1 \approx 0.4427$$

**2. Apply Equioscillation Condition:** To minimize max error, the peak error must equal the negative endpoint error:

$$E(x^*) = -E(0)$$

$$\underbrace{\log_2(1+x^*) - x^* - \sigma}_{\text{Peak (Positive)}} = \underbrace{-(0-\sigma)}_{\text{Endpoint (Negative)}}$$

**Optimal Sigma**

$$2\sigma = \log_2(1+x^*) - x^*$$

$$\sigma_{\text{theo}} \approx \frac{0.08607}{2} \approx \mathbf{0.0430357}$$

**Implied Constant:**

$$R = \frac{3}{2}L(127 - \sigma_{\text{theo}})$$

$$R \approx \texttt{0x5F38146F}$$

**This $\sigma$ minimizes the error of the log-linear step, but NOT the final Newton output.*

# The Theoretical Baseline: Chebyshev Optimization

## The Analytical Approach

Classical approximation theory seeks to minimize the maximum error ($L_\infty$ norm) of the log-linear step:

$$\min_{\sigma} \| \log_2(1+x) - (x+\sigma) \|_\infty \quad \text{for } x \in [0,1]$$

Using **Chebyshev Equioscillation**, the derived optimal shift is:

$$\sigma_{\text{theo}} \approx 0.0430357$$

**The implied "Perfect" Constant:**

$$R_{\text{cheby}} = \frac{3}{2} L(B - \sigma_{\text{theo}}) \approx \texttt{0x5F38146F}$$

## The Discrepancy

Our DE algorithm found $R_{\text{DE}} = \texttt{0x5F375966}$.
Why did the machine deviate from the mathematically proven optimum?

# Validation: The "Greedy" vs. "Strategic" Trade-off

We compared the error profiles of the Analytical (Chebyshev) constant vs. our DE constant.

| Metric | Chebyshev (Theory) | DE (Ours) | Winner |
|---|---|---|---|
| Initial Error ($y_0$) | **3.43%** | 3.85% | Analytical |
| After 1 Newton ($y_1$) | 0.22% | **0.17%** | DE |
| Global Max Error | High Peaks | **Squashed** | DE |

**Key Insight:**

- **Analytical is "Greedy":** It guarantees the best possible *start* ($y_0$).
- **DE is "Strategic":** It sacrifices initial accuracy to position the estimate in a "sweet spot" for the Newton iteration, optimizing the *finish* ($y_{final}$).

# The "Hidden Variable": Bit-Shift Bias

Why does the Analytical derivation fail at the finish line?

**The Math Model**

$$y \approx R - \frac{I_x}{2}$$

Assumes continuous division.

**The Hardware Reality**

$$y \approx R - (I_x \gg 1)$$

Performs **integer truncation**.

## Discovery: Quantization Noise

- For **odd exponents**, the bit-shift $(I_x \gg 1)$ truncates the LSB.
- This creates a systematic **bias** (the subtracted term is smaller than expected, making the result too high).
- **DE's Correction:** It learned to **lower** the constant (from `...38` to `...37`) to compensate for this hardware-specific truncation.

# The "DE Win": Strategic vs. Greedy Optimization

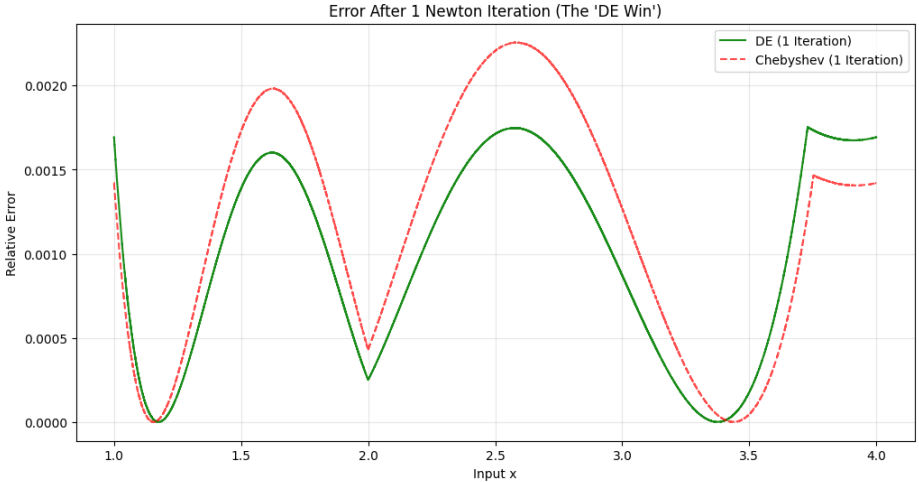



Figure: Relative Error After One Newton-Raphson Iteration ($y_1$)

# Where Does This Matter Today?

**Modern x86/ARM:**

- `rsqrtss` (SSE): 12-bit, 1 cycle
- `vrsqrt14ps` (AVX-512): vectorized
- Software approximation obsolete here

**Where Software Still Matters:**

- **ARM Cortex-M:** No FPU (embedded)
- **RISC-V soft-cores:** Minimal ISA
- **FPGA:** Custom datapaths
- **AI accelerators:** Non-standard formats (FP8, BF16)

**The Niche**

The algorithm lives today in embedded systems, custom silicon, and non-standard precision.
These are exactly where digital twins run on edge devices.

# The Agile Pipeline: Removing the Human-in-the-Loop

**Microcontroller Design Flow:**

1. **Spec:** Define bit-width (FP32, FP16, or Custom FP8/19-bit).
2. **Layout:** AI-automated floorplanning and routing.
3. **Logic:** RTL generation for custom arithmetic units.
4. **Tuning:** Find "Magic Constants" for math primitives.
5. **Verify:** Final performance and error validation.

## The Optimization Gap

**Current State:**
Step 4 often requires manual re-derivation by a human expert whenever Step 1 changes.

**Our DE Solution:**
Acts as an **Automated Math Backend**. Define the format, run DE ($<$1s), and generate the optimal constant.

*Contribution: This framework enables true agility by automating the algorithmic tuning layer that physical design tools ignore.*

# Validation: FP16

**Setup:** IEEE 754 half-precision (5-bit exp, 10-bit mantissa)

## Brute Force

4,096 evaluations
$R^* =$ 0x59ba
Error: $4.84 \times 10^{-6}$

## Differential Evolution

225 evaluations
$R^* =$ 0x59ba
Error: $4.84 \times 10^{-6}$

DE finds the **exact** global optimum with 18$\times$ fewer evaluations.

## Practical Use

For a custom format (e.g., 19-bit on FPGA), brute force is tedious to set up. DE is quick: $<$1 sec, parallelizable, adapts to any format.

# Generalization: Inverse Cube Root

**Function:** $y = x^{-1/3}$

**Derivation:** Following the same logic as inverse sqrt:

$$\log_2 y = -\tfrac{1}{3} \log_2 x$$

$$I_y = R - \lfloor I_x/3 \rfloor \quad \text{where} \quad R = \tfrac{4}{3} \cdot 2^{23}(127 + \sigma)$$

**Newton iteration:**

$$y_{n+1} = y_n \cdot \left(\tfrac{4}{3} - \tfrac{1}{3} x y_n^3\right)$$

**DE Result:** $R = \texttt{0x54a21dbe}$, Error: $1.10 \times 10^{-5}$

The same framework applies to any $x^{-1/n}$ without modification.

# Summary: Automating the Algorithmic Layer

**Key Contributions:**

1. **Landscape Characterization:** Identified the "Fractal Floor" that causes classical gradient methods to fail.
2. **Stochastic Optimization:** Applied Differential Evolution to "filter" local noise and resolve the global basin.
3. **Hardware-Aware Discovery:** DE "learned" to compensate for bit-shift bias and truncation missed by analytical models.
4. **Efficiency:** Achieved optimum in 375 evals (<1s), enabling real-time design iteration.

## The DE-Found Constant

`0x5f375966`

Optimized for Newton Output

## Industrial Impact

Removes the **Human-in-the-Loop** for Agile Hardware design.

As custom formats (FP8/FP4) proliferate, DE acts as a **Constant Compiler** for the next generation of AI accelerators.

# Future Work

1. Other transcendental functions ($\sin$, $\exp$, $\log$)
2. Multi-objective: accuracy vs. iteration count
3. Distribution-specific constants (Gaussian weights, etc.)
4. Emerging formats: FP8, FP4, custom ML formats
5. Theoretical analysis of the rough basin structure

# Thank You

Questions?

**Email:** xzhu09@wm.edu
**Code:** https://github.com/Myouza/fast-inverse-sqrt-research

Joint Mathematics Meetings 2026
Applied Mathematics for Digital Twins

# Appendix: Error Bound Derivation

**Newton iteration:** $y_{n+1} = y_n(3/2 - xy_n^2/2)$
Let $y^* = x^{-1/2}$ and $\varepsilon_n = (y_n - y^*)/y^*$.
Then $y_n = y^*(1+\varepsilon_n)$ and $y_n^2 = (y^*)^2(1+\varepsilon_n)^2 = x^{-1}(1+\varepsilon_n)^2$.

$$
\begin{aligned}
y_{n+1} &= y^*(1+\varepsilon_n)\left(\frac{3}{2} - \frac{x \cdot x^{-1}(1+\varepsilon_n)^2}{2}\right) \\
&= y^*(1+\varepsilon_n)\left(\frac{3}{2} - \frac{(1+\varepsilon_n)^2}{2}\right) \\
&= y^*(1+\varepsilon_n)\left(\frac{3-1-2\varepsilon_n-\varepsilon_n^2}{2}\right) \\
&= y^*(1+\varepsilon_n)\left(1-\varepsilon_n-\frac{\varepsilon_n^2}{2}\right)
\end{aligned}
$$

Expanding: $\varepsilon_{n+1} = -\frac{3}{2}\varepsilon_n^2 - \frac{1}{2}\varepsilon_n^3$

For small $\varepsilon_n$: $\boxed{|\varepsilon_{n+1}| \approx \frac{3}{2}\varepsilon_n^2}$

# Appendix: References



id Software. *Quake III Arena GPL Source*. 2005.


C. Lomont. “Fast Inverse Square Root.” 2003.


R. Storn, K. Price. “Differential Evolution.” *J. Global Opt.*, 1997.


N. Schraudolph. “Fast Exponential.” *Neural Comp.*, 1999.