Accepted for Presentation
Joint Mathematics Meetings (JMM) 2026
AMS Special Session on Applied Mathematics for Digital Twins
Washington, DC — January 4–7, 2026
This is a working manuscript under revision.

OPTIMIZING THE MAGIC CONSTANT IN FAST INVERSE SQUARE ROOT USING DIFFERENTIAL EVOLUTION

XUANLIN ZHU

Abstract. The fast inverse square root algorithm, popularized by the Quake III Arena game engine, is a computational method that approximates $\frac{1}{\sqrt{x}}$ using bit manipulation and Newton's method. Central to this algorithm is the "magic number" constant that initializes the approximation. Traditional approaches to finding this constant have relied on analytical derivations based on assumptions on input distributions. This paper introduces a machine learning approach using Differential Evolution (DE) to optimize the magic number for 32-bit floating-point arithmetic. Our method eliminates the need for complex mathematical derivations while producing a highly accurate constant optimized for a specified range of inputs. We demonstrate that the DE-optimized magic number achieves a worst-case relative error of approximately 4.73×10^{-6} after two Newton iterations. This computational approach offers a generalizable framework for optimizing similar bit-manipulation algorithms, particularly when analytical solutions are unwieldy or when optimizing for specific input distributions.

1. Introduction

Computing the inverse square root function $f(x) = \frac{1}{\sqrt{x}}$ is a fundamental operation in various computational domains, including computer graphics, physics simulations, signal processing, and computational geometry. The traditional method of calculating this function—dividing one by the square root of a number—involves expensive floating-point operations. In performance-critical applications, particularly real-time 3D graphics where vector normalization is frequently required, more efficient approximation methods are desirable.

In the late 1990s, the Quake III Arena video game source code revealed an ingenious algorithm for fast inverse square root calculation. This algorithm achieved remarkable efficiency by combining bit-level manipulation of IEEE 754 floating-point representation with Newton's iterative method. The approach reduced computational cost while maintaining sufficient accuracy for graphics applications. At the heart of this algorithm lies a "magic number" constant—a specific bit pattern that, when used to initialize the approximation, minimizes the error across the input domain.[1]

Traditionally, the determination of this magic number has relied on analytical approaches that leverage the relationship between logarithms and the IEEE 754 format. However, these methods

often require complex mathematical derivations and may not account for the specific distribution of input values encountered in practical applications. Alternative approaches have included brute force searches and experimental tuning, which are computationally intensive and often sub-optimal.[2, 3]

This paper introduces a novel approach to determining the magic number using Differential Evolution (DE), an evolutionary algorithm designed for real-parameter optimization. Our method frames the magic number determination as a numerical optimization problem: finding the bit pattern that minimizes the maximum relative error across a representative sampling of the input space.

By applying DE to this problem, we demonstrate that machine learning techniques can effectively solve what has traditionally been approached through mathematical analysis. Our results show that the DE-optimized magic number achieves a good accuracy particularly when optimized for specific input ranges.

The remainder of this paper is organized as follows: Section 2 provides background on the fast inverse square root algorithm and reviews related work on magic number determination. Section 3 details our methodology, including the DE implementation and error measurement approach. Finally, Section 4 and 5 discusses implications and potential extensions of our work.

2. Related Work

2.1. **The Fast Inverse Square Root Algorithm.** The fast inverse square root algorithm gained widespread attention after its appearance in the Quake III Arena source code release.[1] Attributed to various contributors including John Carmack, Michael Abrash, and earlier developers at SGI, the algorithm was designed to accelerate 3D vector normalizations. The original implementation contained the now-famous "magic number" 0x5f3759df, accompanied by the comments expressing surprise at its effectiveness despite its seemingly arbitrary value.

As Lomont detailed in his comprehensive analysis, the algorithm's efficiency derives from exploiting the IEEE 754 floating-point representation standard and the logarithmic relationship between a number and its inverse square root.[3]

In IEEE 754, a 32-bit single-precision floating-point number consists of:

- 1 sign bit (s)
- 8 exponent bits (e), storing a biased exponent
- 23 mantissa bits (m), representing the fractional part

The value represented is: $(-1)^s \times 2^{e-127} \times (1.m)$, where $(1.m)$ indicates the implied leading 1 followed by the binary fraction represented by the mantissa.

For a positive floating-point number x:

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

The inverse square root is:

$$\frac{1}{\sqrt{x}} = 2^{-(e-127)/2} \times \frac{1}{\sqrt{1.m}}$$

Taking the binary logarithm:

$$\log_2(\frac{1}{\sqrt{x}}) = -\frac{e-127}{2} - \frac{1}{2}\log_2(1.m)$$

$$2$$

The algorithm approximates $\log_2(1.m)$ using a linear function $y \approx x + b$, where the constant b is determined empirically or analytically to minimize the error. This linear approximation is where the "magic" happens.

When interpreted as an integer, the bit pattern of a floating-point number roughly corresponds to $2^{23} \times (e + \log_2(1.m))$. By performing integer subtraction from a specially chosen constant (the magic number), the algorithm effectively computes an approximation of the bit pattern corresponding to the inverse square root.

2.2. **Newton-Raphson Iteration.** The initial approximation provided by the bit manipulation is typically refined using one or more Newton-Raphson iterations. For computing $\frac{1}{\sqrt{x}}$, the Newton iteration is:

$$y_{n+1} = y_n \times (1.5 - 0.5 \times x \times y_n^2)$$

Each iteration approximately doubles the number of correct bits, quickly converging to a highly accurate approximation. The original Quake III implementation used a single iteration, while modern implementations often use two iterations for greater accuracy.

2.3. **Differential Evolution.** Differential Evolution (DE) is an evolutionary algorithm introduced by Storn and Price for optimizing real-parameter functions.[4] DE operates on a population of candidate solutions, evolving them through mechanisms of mutation, crossover, and selection. Unlike traditional gradient-based optimization methods, DE does not require the objective function to be differentiable or even continuous, making it suitable for complex, multi-modal optimization problems.

DE has been successfully applied to various engineering and scientific problems, including parameter estimation, neural network training, and image processing. Its ability to efficiently explore large search spaces while maintaining diversity makes it particularly appropriate for problems where the landscape of the objective function is unknown or difficult to characterize analytically.[5]

Until now, DE has not been systematically applied to the optimization of bit-manipulation algorithms like the fast inverse square root, representing a novel intersection of evolutionary computation and low-level numerical methods.

3. Methodology

3.1. **Data Sampling and Range Justification.** To robustly fit the 32-bit magic constant R that initializes the fast inverse square root, we construct a dataset capturing both critical exponent transitions and broad dynamic range. Specifically, our input samples x_j consist of two parts:

(1) **Exponent anchors:** $x = 2^k$ for $k = -10, -9, \ldots, 10$. These points correspond to exact boundaries in the exponent field of IEEE-754 single precision.
(2) **Log-uniform random samples:** $x = \exp(u)$ where $u \sim \text{Uniform}(\ln 10^{-3}, \ln 10^{3})$, covering six orders of magnitude and ensuring equal representation of relative error across scales.

The combined set $\mathcal{X} = \{2^k\} \cup \{\exp(u)\}$ totals $N = 21 + 200{,}000$ samples. We compute the ground truth $y_j = 1/\sqrt{x_j}$ in IEEE-754 single precision arithmetic to reflect target precision.

3.2. **Floating-Point Range Constraints.** IEEE-754 single precision uses 1 sign bit, 8 exponent bits, and 23 mantissa bits, giving an unbiased exponent range of $[-126, +127]$. Valid bit patterns

for normal numbers lie in $[0x00800000, 0x7f7fffff]$. We therefore constrain the candidate magic constant R to

$$\text{LOW} = \texttt{0x00800000}, \quad \text{HIGH} = \texttt{0x7f800000},$$

clamping any trial R to this interval to avoid subnormals, infinities, or NaNs during bit reinterpretation.

3.3. **Loss Function Design.** To emphasize worst-case accuracy, we define the *maximum relative error* loss:

$$L_\infty(R) = \max_j \Big| \frac{y_2(R; x_j) - x_j^{-1/2}}{x_j^{-1/2}} \Big|.$$

This loss is piecewise constant with sharp kinks at points where different samples become extremal, ensuring that shifts in R that affect any boundary sample alter the loss. To prevent invalid values, any non-finite relative error is substituted with a large constant (e.g. 10^3), guaranteeing L_∞ remains finite.

3.4. **Differential Evolution Algorithm.** Differential Evolution (DE) is a population-based global optimization heuristic. At each generation, a population of real-valued candidates r_i is evolved via mutation, crossover, and selection. For each target r_i, three distinct individuals are chosen and combined to form a mutant vector

$$v = r_a + F\,(r_b - r_c),$$

where F is the mutation factor. A crossover step with probability C_r then produces a trial vector u, which replaces r_i if it yields lower loss. Key parameters are population size N_p, mutation factor F, and crossover rate C_r.

3.4.1. *Objective Function $f(r)$.* We treat $r \in \mathbb{R}$ as a continuous surrogate for the 32-bit magic constant. Define

$$R = \text{clip}\big(\,\text{round}(r),\, \text{LOW},\, \text{HIGH}\big),$$

where

$$\text{LOW} = \texttt{0x00800000}, \quad \text{HIGH} = \texttt{0x7f800000},$$

so that R always corresponds to a valid IEEE-754 single-precision bit pattern. Then compute the two-step Newton approximation

$$y_0 = \texttt{bitcast\_float}(R - (I_x \gg 1)), \quad y_1 = y_0\big(1.5 - 0.5\,x\,y_0^2\big), \quad y_2 = y_1\big(1.5 - 0.5\,x\,y_1^2\big),$$

over a sample $\{x_j\}$ comprising both exact powers of two and log-uniform draws. The loss

$$f(r) \;=\; \max_j \Big| \tfrac{y_2(R;x_j) - x_j^{-1/2}}{x_j^{-1/2}} \Big|$$

measures the worst-case relative error after two Newton steps. Any NaN or infinite relative-error entries are clamped to a large penalty (e.g. 10^3) so that $f(r)$ is always finite.

Algorithm 1 EVALUATE $f(r)$

Require: continuous r, sample $\{x_j\}$, bounds LOW, HIGH

$R \leftarrow \mathrm{round}(r)$
$R \leftarrow \min(\max(R, \mathrm{LOW}), \mathrm{HIGH})$
for all x_j in sample **do**
  $I \leftarrow \texttt{bitcast\_uint}(x_j)$
  $y_0 \leftarrow \texttt{bitcast\_float}\big(R - (I \gg 1)\big)$
  $y_1 \leftarrow y_0\big(1.5 - 0.5\, x_j\, y_0^2\big)$
  $y_2 \leftarrow y_1\big(1.5 - 0.5\, x_j\, y_1^2\big)$
  $e_j \leftarrow \big|\, y_2 - x_j^{-1/2}\big| / x_j^{-1/2}$
end for
replace any NaN or ∞ in $\{e_j\}$ with 10^3
return $\max_j e_j$

3.4.2. *Binomial Crossover.* In differential evolution, after generating a mutant vector v, we produce a trial vector u by *binomial crossover*. Given parent r_i, mutant v, crossover probability $C_r \in [0, 1]$, and dimensionality $D = 1$ here, binomial crossover is:

$$u_j = \begin{cases} v_j, & \text{if } \mathrm{rand}_j \le C_r \quad \vee \quad j = j_{\mathrm{rand}}, \\ r_{i,j}, & \text{otherwise,} \end{cases}$$

where $\mathrm{rand}_j \sim \mathrm{Uniform}(0, 1)$ and $j_{\mathrm{rand}} \in \{1, \ldots, D\}$ is a single index guaranteed to be inherited from the mutant to ensure $u \neq r_i$. In our one-dimensional case, this simplifies to:

$$u = \begin{cases} v, & \text{if rand} \le C_r, \\ r_i, & \text{otherwise.} \end{cases}$$

Algorithm 2 BINOMIALCROSSOVER(r_i, v, C_r)

Require: parent r_i, mutant v, crossover probability C_r

rand $\leftarrow \mathrm{Uniform}(0, 1)$
if rand $\le C_r$ **then**
  $u \leftarrow v$
else
  $u \leftarrow r_i$
end if
return u

We typically set $C_r = 0.9$, ensuring high mixing from the mutant, and verify in experiments that varying C_r between 0.5 and 0.95 does not significantly affect convergence in this one-dimensional problem.

3.5. **DE Parameter Choices and Justification.** We select DE parameters based on the single-dimensional nature of our problem and computational constraints:

- $N_p = 15$, providing sufficient diversity without excessive evaluations.
- Mutation factor $F = 0.5$, a standard compromise between exploration and stability.

- Crossover rate $C_r = 0.9$, favoring incorporation of mutant information while retaining existing structure.
- Maximum generations = 50 and tolerance 10^{-3}, since convergence in one dimension is rapid and evaluation cost is low.

These choices enable DE to locate the optimum within a few hundred function evaluations.

3.6. **Experimental Implementation.** We implement DE using SciPy's `differential_evolution`. The objective function wraps rounding and clamping of each candidate to the valid bit-pattern range, computes two Newton iterations, and evaluates L_∞. Pseudocode is given in Algorithm 3.

Algorithm 3 Differential Evolution for Magic Constant

Define $f(r)$ by rounding $R = \text{round}(r)$, clamping to [LOW,HIGH], computing y_2 and returning $L_\infty(R)$.
Initialize population $\{r_i\}$ uniformly in [LOW,HIGH].
for generation = 1 **to** max_gen **do**
  for each r_i in population **do**
    Randomly select r_a, r_b, r_c distinct.
    Mutant $v = r_a + F\,(r_b - r_c)$.
    Trial $u = \text{binomial\_crossover}(v, r_i, C_r)$.
    Evaluate $f(u)$ and $f(r_i)$.
    if $f(u) < f(r_i)$ **then**
      Replace $r_i \leftarrow u$
    end if
  end for
  if convergence criterion met **then**
    break
  end if
end for
return $R^* = \text{round}(\arg\min_i f(r_i))$

3.7. **Experimental Results.** We applied DE with the above settings to our sample of 220 001 points. The progression of the population's best objective f was recorded in Table 1:

After refinement, DE converged to

$$\boxed{R^* = \texttt{0x5f375966},}$$

achieving a worst-case relative error of 4.7254×10^{-6}.

4. Discussion

This result is both quantitatively excellent and practically acceptable for real-time applications.

Crucially, DE's global search capability allows it to *jump out of local minima* and traverse the very flat plateaus created by the Newton–Raphson refinement steps. In contrast, traditional gradient-based methods would stall on such plateaus, unable to find a descent direction among the numerous subtle local minima.

Furthermore, the DE approach is agnostic to the input distribution: the same framework can tune the magic constant even for distributions that defy analytic treatment.

Generation	$\mathbf{f}_{\max}$
1	0.9999859333038330
2	0.9991989731788635
3	0.2114003151655197
4	0.2114003151655197
5	0.12296796590089798
6	0.08253248035907745
7	0.08253248035907745
8	0.08253248035907745
9	0.00673156650736928
10	0.0005632959655486047
11	0.0005632959655486047
12	0.00003340846160426736
13	0.00003340846160426736
14	0.00003340846160426736
15	0.000008141875696310308
16	0.000006036231752659660
17	0.000005946666533418465
18	0.000004837864253204316
19	0.000004837864253204316
20	0.000004812646238860907
21	0.000004812646238860907
22	0.000004729200099973241
23	0.000004729200099973241
24	0.000004729200099973241
25	0.000004729200099973241
26	0.000004727283794636605
27	0.000004725406142824795

TABLE 1. Evolution of the worst-case relative error $f(r)$ over DE generations.

5. Future Work

(1) **Extending the bitwise hack to other transcendental functions.** Functions such as $\exp(x)$ can also be accelerated using floating-point bit-manipulations. In particular, the MIT paper on exponential approximations by Schraudolph has demonstrated how a carefully chosen "magic" bias constant can yield efficient exp evaluations [6]. Applying DE to discover or refine these magic constants promises similar improvements.

(2) **Adapting to alternative precision formats.** Modern machine-learning hardware increasingly employs 16-bit and 8-bit floating-point formats to reduce memory and energy costs [7, 8]. We may apply the DE framework to search for optimal magic constants under these reduced-precision schemes, as well as 64-bit double precision, thereby broadening the bitwise-trick's applicability across diverse computing environments.

References

[1] id Software. Quake III Arena GPL source release. GitHub repository, 2005.

[2] M. Robertson. *A Brief History of InvSqrt*. Ph.D. dissertation, University of New Brunswick, 2012.

[3] C. Lomont. *Fast inverse square root*. Technical report, 2003.

[4] R. Storn and K. Price. Differential Evolution—A Simple and Efficient Heuristic for Global Optimization over Continuous Spaces. *Journal of Global Optimization*, 11(4):341–359, 1997.

[5] S. Das and P. N. Suganthan. Differential Evolution: A Survey of the State-of-the-Art. *IEEE Transactions on Evolutionary Computation*, 15(1):4–31, 2011.

[6] N. N. Schraudolph. A Fast, Compact Approximation of the Exponential Function. *Neural Computation*, 11(4):853–862, 1999.

[7] P. Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David García, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. Mixed Precision Training. *International Conference on Learning Representations (ICLR)*, 2018.

[8] M. Courbariaux, Y. Bengio, and J. David. BinaryConnect: Training Deep Neural Networks with binary weights during propagations. *Advances in Neural Information Processing Systems (NeurIPS)*, 2015.