electronics

MDPI

Article

Efficient n-th Root Computation on Microcontrollers Employing Magic Constants and Modified Newton and Householder Methods

Cezary J. Walczyk, Maciej Jurgielewicz and Jan L. Cieśliński *

Wydział Fizyki, Uniwersytet w Białymstoku, ul. Ciołkowskiego 1L, 15-245 Białystok, Poland; c.walczyk@uwb.edu.pl (C.J.W.); m.jurgielewicz@uwb.edu.pl (M.J.)
* Correspondence: j.cieslinski@uwb.edu.pl

Abstract

With the growing number of applications in embedded systems—such as IoT modules, smart sensors, and wearable devices—there is an increasing demand for fast and accurate computations on resource-constrained platforms. In this paper, we present a new method for computing n-th roots in floating-point arithmetic based on an initial estimate generated by a "magic constant," followed by one or two iterations of a modified Newton–Raphson or Householder algorithm. For cubic and quartic roots, we provide C implementations operating in single-precision floating-point format. The proposed algorithms are evaluated in terms of maximum relative error and execution time on selected microcontrollers. They exhibit high accuracy and noticeably reduced computation time. For example, our methods for computing cubic roots outperform the standard library function cbrtf() in both speed and precision. The results may be useful in a variety of fields, including biomedical and biophysical applications, statistical analysis, and real-time image and signal processing.

Keywords: embedded systems; floating-point arithmetic; piece-wise linear approximation; magic constants; cube root (cbrt); quartic root; ESP32; STM32

1. Introduction

Efficient evaluation of real-valued n-th roots is a fundamental building block in a wide range of numerical procedures, particularly in systems that must operate under strict constraints on computation time, memory footprint, and power consumption. Such constraints are especially pronounced in embedded systems, wearable devices, microcontroller-based biomedical instrumentation, and real-time signal-processing pipelines. Applications include dynamic-range and gamma-correction pipelines in modern imaging devices [1–5], fractional-order control algorithms implemented on microcontrollers [6], biomedical applications [7–9], and preprocessing stages in AI-driven statistical systems [10,11]. In all these contexts, root operations—square, cube, quartic, and inverse—can serve as essential primitives in normalization, error-transformation, spectral shaping, and nonlinear model components [12–15].

Power functions with sub-unit exponents can also appear in many of the same embedded perception and control pipelines. Gamma correction is a classical example: applying $Y_{\text{out}} = Y_{\text{in}}^{\gamma}$ with $0 < \gamma < 1$ lifts dark regions and protects highlights. Recent low-light methods make this operation spatially adaptive; adaptive chrominance gamma correction (ACGC) avoids hue shifts while enhancing detail [3]. A similar idea appears in rotation-aware object detection, where applying a small gamma on the Value channel of HSV improves robustness on aerial imagery [4]. Gradient-domain HDR compression is not

Check for updates

Academic Editors: Francesco Gagliardi and Massimo Piotto

Received: 19 November 2025
Revised: 17 December 2025
Accepted: 23 December 2025
Published: 26 December 2025

Copyright: © 2025 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license.

pointwise x^p, but relies on the same principle of compressing large variations while preserving fine structure [2]. These transformations are evaluated at every pixel, often per frame. On microcontrollers, large lookup tables are expensive and generic `pow` calls are slow; compact and adjustable power-law or root-like operations are therefore crucial.

Power transformations have long been a fundamental tool in data analysis, both in classical exploratory approaches and in statistical modeling. Tukey emphasized their practical value in exploratory data analysis through the ladder of powers [16], while Box and Cox formalized them in a parametric framework that enables continuous optimization of the transformation exponent [17]. In modern applications, including advanced biochemical and metabolomic analyses [18], suitable power transformations remain an essential component of preprocessing pipelines for signal normalization and variance stabilization.

Audio pipelines create similar demands. Dynamic-range compression uses concave envelope curves behaving like x^p; theoretical analyses link such mappings to envelope correlations, effective gain, and SNR, motivating signal-dependent compression strength and frequent evaluation of sub-unit power laws [1]. Speech dereverberation with learning-based models uses magnitude compression $|X|^b$ with $b \in (0,1)$, increasing the number of power evaluations per time–frequency bin [19]. For on-device processing, these costs accumulate; microcontroller-friendly kernels for x^p or $x^{1/n}$ can reduce latency and energy.

Control systems add further motivation. Fractional-order PID controllers have frequency responses scaling as ω^α with $\alpha \in (0,1)$; microcontroller studies [6] demonstrate that approximations of these behaviors require frequent evaluation of power-law components, demanding predictable and efficient numerical routines.

Commonly used methods for general n-th root evaluation often rely on unoptimized Newton iterations or achieve accuracy at the cost of computational expense or divergence on devices without hardware floating-point units (FPUs). Even where FPUs exist, unoptimized approaches may suffice, but often, e.g., on ESP-32-class devices [20], the overhead becomes substantial, limiting real-time performance.

This limitation is significant because embedded and wearable systems increasingly require on-device computation. EMG, EEG, and BCI systems need low-latency in-place processing [21,22]. Remote-sensing systems using hyperspectral data [23] rely on efficient local preprocessing. In many such pipelines, power laws and n-th roots appear repeatedly, for instance, in dynamic-range compression [1], HDR/gamma correction [2–5], geometric normalization [24], real-time RMS voltage estimation [25], and nonlinear state-estimation routines [26]. These operations must be both fast and accurate.

The present paper is motivated by simple, fast, and accurate algorithms for the inverse square root [27–29] historically rooted in early floating-point tricks [30] and followed by works on some other power functions, including square roots [31], reciprocal functions [32,33], inverse cube roots [34–36], and other negative exponents [37].

In this work, we present a new, very fast, and practically machine-accurate algorithm for computing $x^{1/n}$ for arbitrary integers $n \geq 2$, with emphasis on cube and quartic roots. Our approach combines a bit-level initial multi-interval approximation (a generalized integer-arithmetic magic-seed construction), and one or two modified Newton–Raphson or Householder iterations.

To validate performance of cube and quartic root computation across hardware architectures, we benchmark it on five microcontroller platforms: two with hardware FPUs and three without, including representative STM-32 and ESP-32 devices. Results show that the proposed approach yields a high-performance, low-power, and highly accurate solution suitable for a broad spectrum of embedded and wearable signal-processing applications.

2. A Floating-Point Approximation of the n-th Root Functions with Magic Numbers

In this article, we deal with FP numbers in the following form:

$$x = (-1)^{s_x} 2^{e_x} \cdot (1 + m_x), \tag{1}$$

where $s_x \in \{0, 1\}$ denotes the sign of the number, e_x is the number's exponent, and m_x is its mantissa. The IEEE-754 standard [38] defines two kinds of FP number representations—32-bit and 64-bit. Despite the fact that the theory in the present article refers to both representations, the practical testing of the accuracy and performance achieved by the implemented algorithms was conducted only for 32-bit precision. In the case of the standard IEEE-754, a floating-point number, the first bit corresponds to an s_x, the next 8 bits correspond to the biased exponent $E_x = e_x + \text{bias}$ (bias = 127), and the last 23 bits encode a mantissa. The range of available numbers for which we can determine n-th roots can be divided into disjoint intervals:

$$x \in \bigcup_j A_j \quad \text{where} \quad A_j = \left[2^{n \cdot j},\, 2^{n \cdot (j+1)}\right) \text{ for } j \in \{0, \pm 1, \pm 2, \dots\}. \tag{2}$$

If $x \in A_j$, then $y = \sqrt[n]{x}$ takes values within the interval $(2^j, 2^{j+1}]$. By introducing the new variables

$$\tilde{x} = 2^{-n \cdot j} x = 2^{e_{\tilde{x}}} \cdot (1 + m_{\tilde{x}}), \quad \tilde{y} = 2^{-n} y = 2^{e_{\tilde{y}}} \cdot (1 + m_{\tilde{y}}) \tag{3}$$

the whole problem may be restricted to computing the $\tilde{y} = \sqrt[n]{\tilde{x}}$ for the $\tilde{x} \in [1, 2^n)$ and $e_{\tilde{x}} \in \{0, 1, \dots n-1\}$; it follows from (3) that $e_{\tilde{y}} = 0$.

If we calculate the logarithm base 2 of $\tilde{y} = \sqrt[n]{\tilde{x}}$, taking into account (3), we obtain the formula

$$e_{\tilde{y}} + \log_2(1 + m_{\tilde{y}}) = \frac{1}{n} e_{\tilde{x}} + \frac{1}{n} \log_2(1 + m_{\tilde{x}}). \tag{4}$$

By applying a linear approximation of the logarithmic function

$$\log_2 a \simeq a + c_a,$$

where c_a is a certain undetermined constant, we obtain

$$1 + m_{\tilde{y}} + e_{\tilde{y}} \simeq \frac{1}{n} e_{\tilde{x}} + \frac{1}{n}(1 + m_{\tilde{x}}) + \underbrace{\left(\frac{c_{1+m_{\tilde{x}}}}{n} - c_{1+m_{\tilde{y}}}\right)}_{c}, \tag{5}$$

which, in equivalent form, is

$$\underbrace{2^p\left(E_{\tilde{y}} + m_{\tilde{y}}\right)}_{I_y} \simeq \underbrace{\left\lfloor 2^p\left(1 - \frac{1}{n}\right)(bias - 1) + 2^p c \right\rfloor}_{R \leftarrow \text{magic constant}} + \underbrace{\frac{2^p\left(E_{\tilde{x}} + m_{\tilde{x}}\right)}{n}}_{I_x/n}, \quad I_y,\, I_x,\, R \in \mathbb{N}^+, \tag{6}$$

where p is the number of mantissa bits. For 32-bit numbers, $p = 23$, and for 64-bit numbers, $p = 52$. Formula (6) is analogous to the equation $I_y \simeq R - I_x$ for reciprocal functions [32,33], and $I_y \simeq R - I_x/2$ and $I_y \simeq R - I_x/3$ for computing the reciprocal square and cube roots [27–31,34,35].

Assuming from relation (5) that $e_{\tilde{x}}$ and $e_{\tilde{y}}$ are constant integer parameters, we may derive an approximate solution $\tilde{y}_{00} \simeq \tilde{y}$ that is linearly dependent on $\tilde{x}$:

$$\tilde{y}_{00}(\tilde{x}) = 2^{e_{\tilde{y}}}\left(-e_{\tilde{y}} + \frac{e_{\tilde{x}}}{n} + \frac{2^{-e_{\tilde{x}}}}{n}\tilde{x} + c\right). \tag{7}$$

Such an approximation is not optimal from the perspective of minimizing the relative error of n-th roots. To address this, it is sufficient to introduce an additional scaling parameter β:

$$\tilde{y}_0(\tilde{x}) = \beta \cdot \tilde{y}_{00}(\tilde{x}), \quad \delta_0(\tilde{x}) = \left(\tilde{y}_0(\tilde{x}) / \sqrt[n]{\tilde{x}}\right) - 1. \tag{8}$$

3. Multi-Interval Approximation

In order to obtain sufficiently accurate approximations, we divide the intervals $[2^{e_{\tilde{x}}}, 2^{e_{\tilde{x}}+1})$, where $e_{\tilde{x}} \in \{0, 1, \ldots, n-1\}$, into M equal parts:

$$\tilde{x} \in [2^{e_{\tilde{x}}}, 2^{e_{\tilde{x}}+1}) = \bigcup_{m=1}^{M} [x_{\mathbf{b},e_{\tilde{x}},m},\, x_{\mathbf{e},e_{\tilde{x}},m}), \quad \text{where} \quad \frac{x_{\mathbf{b},e_{\tilde{x}},m}}{2^{e_{\tilde{x}}}} = 1 + \frac{m-1}{M}, \; \frac{x_{\mathbf{e},e_{\tilde{x}},m}}{2^{e_{\tilde{x}}}} = 1 + \frac{m}{M}.$$

In every sub-interval $[x_{\mathbf{b},e_{\tilde{x}},m},\, x_{\mathbf{e},e_{\tilde{x}},m})$, we take a different piece-wise linear function of the form (8) as an initial approximation of the function $\sqrt[n]{\tilde{x}}$:

$$\tilde{y}_{0,e_x,m}^{[n\cdot M]}(\tilde{x}) = \beta_{e_x,m}^{[n\cdot M]} \cdot 2^{e_{\tilde{y}}}\left(-e_{\tilde{y}} + \frac{e_{\tilde{x}}}{n} + \frac{2^{-e_{\tilde{x}}}}{n}\tilde{x} + c_{e_x,m}^{[n\cdot M]}\right), \quad \delta_{0,e_x,m}^{[n\cdot M]}(\tilde{x}) = \frac{\tilde{y}_{0,e_x,m}^{[n\cdot M]}(\tilde{x})}{\sqrt[n]{\tilde{x}}} - 1. \tag{9}$$

Optimal solutions minimizing the relative error ${\delta_{0,e_x,m}}^{[n\cdot M]}(\tilde{x})$ are obtained for

$$c_{e_x,m}^{[n\cdot M]} = e_{\tilde{y}} - \frac{1}{n}e_{\tilde{x}} + \frac{1}{n}w_{n,m,M}, \tag{10}$$

$$\beta_{e_x,m}^{[n\cdot M]} = \frac{2^{1-e_{\tilde{y}}+e_{\tilde{x}}/n}\, n(n-1)^{1-1/n}\left(1+\frac{m-1}{M}\right)^{1/n}}{(n-1)^{1-1/n}\left(1+\frac{m-1}{M}+w_{n,m,M}\right) + n\left(1+\frac{m-1}{M}\right)^{1/n} w_{n,m,M}^{1-1/n}}, \tag{11}$$

where

$$w_{n,m,M} = -1 - \frac{m}{M} + \frac{1}{M}\frac{\sqrt[n]{1+\frac{m}{M}}}{\sqrt[n]{1+\frac{m}{M}} - \sqrt[n]{1+\frac{m-1}{M}}}. \tag{12}$$

The maximum error for a given sub-interval m does not depend on $e_{\tilde{y}}$ and equals

$$\delta_{0,e_x,m,\max}^{[n\cdot M]} = 1 - \frac{2n\left(1+\frac{m-1}{M}\right)^{1/n} w_{n,m,M}^{1-1/n}}{(n-1)^{1-1/n}\left(1+\frac{m-1}{M}+w_{n,m,M}\right) + n\left(1+\frac{m-1}{M}\right)^{1/n} w_{n,m,M}^{1-1/n}}. \tag{13}$$

In this case, the equations that symmetrize the extreme values of the errors are satisfied:

$$\delta_{0,e_x,m,\max}^{[n\cdot M]} = \tilde{y}_{0,e_x,m}^{[n\cdot M]}(x_{\mathbf{b},e_{\tilde{x}},m}) = \tilde{y}_{0,e_x,m}^{[n\cdot M]}(x_{\mathbf{e},e_{\tilde{x}},m}), \quad \delta_{0,e_x,m,\max}^{[n\cdot M]} = -\delta_{0,e_x,m,\min}^{[n\cdot M]},$$

where $\delta_{0,e_x,m,\min}^{[n\cdot M]}$ is the minimum of the error function:

$$\delta_{0,e_x,m,\min}^{[n\cdot M]} = \tilde{y}_{0,e_x,m}^{[n\cdot M]}(x_{\min}), \quad x_{\min} = \frac{2^{e_x}\left(e_x + n\, c_{e_x,m}^{[n\cdot M]} - n\, e_y\right)}{n-1}.$$

The plots of relative error functions $\delta_{0,e_x,m}^{[n\cdot M]}(\tilde{x})$ (8) for $n = 3$ and $n = 4$ are shown in Figures 1 and 2. The maximum error for all $e_{\tilde{x}} \in \{0, 1, \ldots, n-1\}$ corresponds to $m = 1$:

$$\delta_{0,\max}^{[n\cdot M]} = \delta_{0,e_x,1,\max}^{[n\cdot M]}. \tag{14}$$

The values of the maximal relative errors $\delta_{0,\max}^{[n\cdot M]}$ and their corresponding numbers of correct bits $ac_{\text{bits}}^{[n\cdot M]} = \lfloor -\log_2 \delta_{0,\max}^{[n\cdot M]} \rfloor$ for the approximation $\tilde{y}_{0,e_x,m}^{[n\cdot M]}(\tilde{x})$ with $n \in \{2, 3, 4\}$ and $M \in \{1, 2, 4, 8, 16\}$ are given in Table 1.

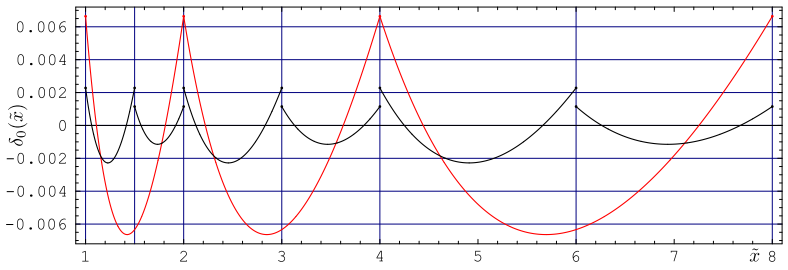


Figure 1. Error functions: $\delta_{0,e_x,1}^{[3\cdot M]}(\tilde{x})$ for $M = 1$ (red line), and $\delta_{0,e_x,m}^{[3\cdot M]}(\tilde{x})$ for $M = 2$ and $m \in \{1, 2\}$ (black line).

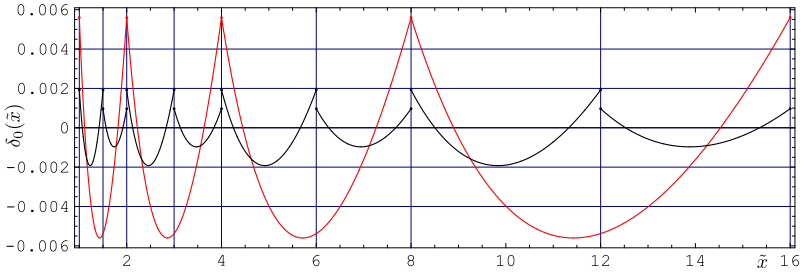


Figure 2. Error functions: $\delta_{0,e_x,1}^{[4\cdot M]}(\tilde{x})$ for $M = 1$ (red line), and $\delta_{0,e_x,m}^{[4\cdot M]}(\tilde{x})$ for $M = 2$ and $m \in \{1, 2\}$ (black line).

Table 1. Maximal relative errors $\delta_{0,\max}^{[n\cdot M]}$ and their corresponding numbers of correct bits $ac_{\text{bits}}^{[n\cdot M]} = \lfloor -\log_2 \delta_{0,\max}^{[n\cdot M]} \rfloor$ for the approximation $\tilde{y}_{0,e_x,m}^{[n\cdot M]}(\tilde{x})$.

	$M = 1$	$M = 2$	$M = 4$	$M = 8$	$M = 16$
$\delta_{0,\max}^{[2\cdot M]}$	$7.47 \cdot 10^{-3}$	$2.56 \cdot 10^{-3}$	$7.78 \cdot 10^{-4}$	$2.17 \cdot 10^{-4}$	$5.74 \cdot 10^{-5}$
$ac_{\text{bits}}^{[2\cdot M]}$	7	8	10	12	14
$\delta_{0,\max}^{[3\cdot M]}$	$6.64 \cdot 10^{-3}$	$2.28 \cdot 10^{-3}$	$6.91 \cdot 10^{-4}$	$1.93 \cdot 10^{-4}$	$5.10 \cdot 10^{-5}$
$ac_{\text{bits}}^{[3\cdot M]}$	7	8	10	12	14
$\delta_{0,\max}^{[4\cdot M]}$	$5.60 \cdot 10^{-3}$	$1.92 \cdot 10^{-3}$	$5.83 \cdot 10^{-4}$	$1.63 \cdot 10^{-4}$	$4.31 \cdot 10^{-5}$
$ac_{\text{bits}}^{[4\cdot M]}$	7	8	10	12	14

4. Modifications of the Newton–Raphson Method

In order to improve the accuracy, the zeroth approximation ($\tilde{y}_{0,e_x,m}^{[n\cdot M]}(\tilde{x})$) can be corrected using the Newton–Raphson method:

$$\tilde{y}_k^{(NR)} = \tilde{y}_{k-1}^{(NR)} - f(\tilde{y}_{k-1}^{(NR)}) / f'(\tilde{y}_{k-1}^{(NR)}), \quad k \in \{1, 2, \dots\}, \tag{15}$$

where $f(y) = y^n - x$ and $\tilde{y}_k^{(NR)} = \tilde{y}_{k,e_x,m}^{[n\cdot M]}(\tilde{x})$. Therefore,

$$\tilde{y}_k^{(NR)} = \frac{1}{n}\tilde{y}_{k-1}^{(NR)}\left(n - 1 + \tilde{x}\,\tilde{y}_{k-1}^{(NR)^{-n}}\right). \tag{16}$$

After the symmetrization of the relative error, the method takes the following form:

$$\tilde{y}_k^{(M1)} = \frac{\kappa_{1,k}}{n}\tilde{y}_{k-1}^{(M1)}\left((n-1)\kappa_{0,k} + \tilde{x}\,\tilde{y}_{k-1}^{(M1)^{-n}}\right), \tag{17}$$

where

$$\kappa_{0,k} = \frac{\left(1+\delta_{k-1,\max}^{(M1)}\right)^{n-1} - \left(1-\delta_{k-1,\max}^{(M1)}\right)^{n-1}}{2(n-1)\delta_{k-1,\max}^{(M1)}\left(1-\delta_{k-1,\max}^{(M1)^2}\right)^{n-1}}, \tag{18}$$

$$\kappa_{1,k} = \frac{4n\,\delta_{k-1,\max}^{(M1)}\left(1-\delta_{k-1,\max}^{(M1)^2}\right)^{n-1}}{2n\,\kappa_{0,k}^{1-1/n}\delta_{k-1,\max}^{(M1)}\left(1-\delta_{k-1,\max}^{(M1)^2}\right)^{n-1} + \left(1+\delta_{k-1,\max}^{(M1)}\right)^{n} - \left(1-\delta_{k-1,\max}^{(M1)}\right)^{n}}, \tag{19}$$

and $\delta_{k-1,\max}^{(M1)}$ is the maximum relative error of the $(k-1)$th iteration. Since the relationship between the relative errors of each iteration, according to Equation (17), is described by the equation

$$\delta_k^{(M1)} = \frac{\kappa_{1,k}}{n}\left(1+\delta_{k-1}^{(M1)}\right)\left((n-1)\kappa_{0,k} + \left(1+\delta_{k-1}^{(M1)}\right)^{-n}\right) - 1. \tag{20}$$

The maximum error of higher iterations can be expressed using the errors of earlier iterations:

$$\delta_{k,\max}^{(M1)} = \frac{\kappa_{1,k}}{2n}\frac{\left(1+\delta_{k-1,\max}^{(M1)}\right)^{n} - \left(1-\delta_{k-1,\max}^{(M1)}\right)^{n}}{\delta_{k-1,\max}^{(M1)}\left(1-\delta_{k-1,\max}^{(M1)^2}\right)^{n-1}} - 1. \tag{21}$$

The values of the maximal relative errors $\delta_{k,\max}^{(M1)}$ for initial approximation $\tilde{y}_{0,e_x,m}^{[n\cdot M]}(\tilde{x})$ with $n \in \{3, 4\}$ and $M \in \{1, 2, 4, 8, 16\}$ are given in Tables 2 and 3. Analyzing the data presented in the tables, it can be concluded that for floating-point calculations with a 23-bit mantissa, two iterations of type $M1$ are sufficient when $M = 1$, while for $M = 16$, only one iteration is needed. In the case of $M = 8$, using only one iteration, due to rounding errors, results in maximum relative errors greater than 10^{-7}.

Table 2. Maximal relative errors $\delta_{k,\max}^{(M1)}$ and their corresponding numbers of correct bits $ac_{\text{bits},k} = \lfloor -\log_2 \delta_{k,\max}^{(M1)} \rfloor$ for initial approximation $\sqrt[3]{\tilde{x}} \simeq \tilde{y}_{0,e_x,m}^{[3\cdot M]}(\tilde{x})$ and $k \in \{1, 2\}$.

	$M = 1$	$M = 2$	$M = 4$	$M = 8$	$M = 16$
$\delta_{1,\max}$	$2.20 \cdot 10^{-5}$	$2.60 \cdot 10^{-6}$	$2.39 \cdot 10^{-7}$	$1.86 \cdot 10^{-8}$	$1.30 \cdot 10^{-9}$
$ac_{\text{bits},1}$	15	18	21	25	29
$\delta_{2,\max}$	$2.43 \cdot 10^{-10}$	$3.37 \cdot 10^{-12}$	$2.85 \cdot 10^{-14}$	$1.72 \cdot 10^{-16}$	$8.49 \cdot 10^{-19}$
$ac_{\text{bits},2}$	31	38	44	52	60

4.1. Algorithms Based on Two Iterations

In this subsection, we present algorithms to compute $\sqrt[3]{x}$ and $\sqrt[4]{x}$, derived based on two modified Newton–Raphson (M1) iterations, which are described by Formula (17). In order to optimize the *C* implementation, two iterations are given in the following form:

$$z_1^{(M1)} = z_0^{(M1)} + k_1 \cdot x \cdot z_0^{(M1)^{1-n}}, \tag{22}$$

$$y_2^{(M1)} = z_1^{(M1)} + k_2 \cdot x \cdot z_1^{(M1)^{1-n}}, \tag{23}$$

where

$$\beta_2 = \left(1 - \frac{1}{n}\right)^2 \kappa_{0,1}\,\kappa_{0,2}\,\kappa_{1,1}\,\kappa_{1,2}\,, \qquad z_0^{(M1)} = \beta_2 \cdot y_0^{(M1)}\,, \tag{24}$$

$$k_1 = \frac{1}{n}\left(1 - \frac{1}{n}\right)^{2n-1} \kappa_{0,1}^{n-1}\kappa_{0,2}^{n}\,\kappa_{1,1}^{n}\,\kappa_{1,2}^{n}\,, \qquad k_2 = \frac{1}{n}\left(1 - \frac{1}{n}\right)^{n-1} \kappa_{0,2}^{n-1}\,\kappa_{1,2}^{n}\,. \tag{25}$$

Table 3. Maximal relative errors $\delta_{k,\max}^{(M1)}$ and their corresponding numbers of correct bits $ac_{\text{bits},k} = \lfloor -\log_2 \delta_{k,\max}^{(M1)} \rfloor$ for initial approximation $\sqrt[4]{\tilde{x}} \simeq \tilde{y}_{0,e_x,m}^{[4\cdot M]}(\tilde{x})$ and $k \in \{1, 2\}$.

	$M = 1$	$M = 2$	$M = 4$	$M = 8$	$M = 16$
$\delta_{1,\max}$	$2.35 \cdot 10^{-5}$	$2.77 \cdot 10^{-6}$	$2.55 \cdot 10^{-7}$	$1.98 \cdot 10^{-8}$	$1.39 \cdot 10^{-9}$
$ac_{\text{bits},1}$	15	18	21	25	29
$\delta_{2,\max}$	$4.15 \cdot 10^{-10}$	$5.77 \cdot 10^{-12}$	$4.84 \cdot 10^{-14}$	$2.95 \cdot 10^{-16}$	$1.45 \cdot 10^{-18}$
$ac_{\text{bits},2}$	31	37	44	51	59

Calculation of $\sqrt[3]{\tilde{x}}$ and $\sqrt[4]{\tilde{x}}$ is implemented in the *C* language through the functions *rt_M1_n3()* and *rt_M1_n4()*, in that order (Algorithms 1 and 2). Magic constant tables *R*3 and *R*4 were calculated based on Formulas (10) and (6), and table *beta*3 stores the multiplication of the β_2 parameter and the table, the coefficients of which are computed by $\beta_{e_x,m}^{[n,M]}$ (see (11)) for adequate $e_{\tilde{x}}$.

Algorithm 1 rt_M1_n3

```
float rt_M1_n3(float x){// x^1/3 -> II iter M1
  static const float beta3[3] = {0.553788056219f, 0.348864614605f,
      0.439541871504f};
  static const unsigned int R3[3] = {707012356, 712604762, 709808559};
  float y; unsigned int i, j;
  j = *(unsigned int*)&x;
  i = j » 23; j = j / 3; i = i - 3 * (j » 23);
  j = R3[i] + j; y = *(float*)&j; y *= beta3[i];
  y += x * 0.0438928459199f / (y * y); //k1=0.0438928459199f
  y = fmaf(x, 0.148148148040f / (y * y), y); //k2=0.148148148040f
  return y;
}
```

Algorithm 2 rt_M1_n4

```
float rt_M1_n4(float x){ // x^1/4 -> II iter M1
  static const float beta3[4] = {0.509100911370f, 0.605426426055f,
    0.719977413475f, 0.428101131373f};
  static const unsigned int R4[4] = {799613048, 797515896,
    795418744, 801710200};
  float y; unsigned int i, j;
  j = *(unsigned int*)&x; i = (j » 23) & 3; j = j » 2;
  j = R4[i] + j; y = *(float*)&j; y *= beta3[i];
  y += x * 0.0333678322765f / (y * y * y); //k1=0.0333678322765f
  y = fmaf(x, 0.105468749825f / (y * y * y), y); //k2=0.105468749825f
  return y;
}
```

4.2. Algorithms Based on One Iteration

We present algorithms computing $\sqrt[3]{x}$ and $\sqrt[4]{x}$ derived using one M1 iteration. To ensure sufficient precision, iteration will be performed for the approximations resulting from the partition of the ranges $[2^{e_{\bar{x}}}, 2^{e_{\bar{x}}+1}]$ into $M = 16$ equal sections. *C*-language optimized iteration takes the form

$$z_1^{(M1)} = z_0^{(M1)} + k_1 \cdot x \cdot {z_0^{(M1)}}^{1-n}, \tag{26}$$

where

$$z_0^{(M1)} = \beta_2 \cdot y_0^{(M1)}, \qquad \beta_2 = \left(1 - \frac{1}{n}\right)\kappa_{0,1}\,\kappa_{1,1}, \qquad k_1 = \frac{1}{n}\beta_2^{n-1}\,\kappa_{1,1}. \tag{27}$$

Cube root and fourth root approximations based on z_0 and z_i are implemented in the functions *rt_M1_n3_M16()* and *rt_M1_n4_M16()* (Algorithms 3 and 4). Table *beta*3 stores the multiplication of the β_2 parameter and the table, the coefficients of which are computed by $\beta_{e_x,m}^{[n,M]}$ (see (11)) for adequate $e_{\bar{x}}$.

Algorithm 3 rt_M1_n3_M16

```
float rt_M1_n3_M16(float x){ // x^1/3 -> I iter M1
  static const float beta3[48] = {
    1.03700808869f, 0.997092402893f, 0.960809533798f, 0.927657681256f,
    0.897225687452f, 0.869173170848f, 0.843215711384f, 0.819113640440f,
    0.796663445928f, 0.775691103009f, 0.756046842097f, 0.737601003018f,
    0.720240719391f, 0.703867244273f, 0.688393775961f, 0.673743677394f,
    0.653274159927f, 0.628128853547f, 0.605272078286f, 0.584387719855f,
    0.565216765063f, 0.547544786978f, 0.531192612188f, 0.516009258923f,
    0.501866522603f, 0.488654774449f, 0.476279665532f, 0.464659515063f,
    0.453723221676f, 0.443408578696f, 0.433660904475f, 0.424431920691f,
    0.823073865444f, 0.791392764631f, 0.762595032346f, 0.736282389546f,
    0.712128500057f, 0.689863202873f, 0.669260753644f, 0.650130927258f,
    0.632312196065f, 0.615666436459f, 0.600074776241f, 0.585434304062f,
    0.571655437816f, 0.558659802003f, 0.546378502065f, 0.534750711126f};
  static const unsigned int R3[48] = {
    704815480, 705165140, 705514786, 705864419, 706214041, 706563655,
    706913260, 707262859, 707612452, 707962040, 708311622, 708661201,
    709010776, 709360347, 709709916, 710059482, 710407886, 710757546,
    711107191, 711456824, 711806447, 712156060, 712505666, 712855265,
    713204857, 713554445, 713904028, 714253606, 714603181, 714952753,
    715302321, 715651887, 707611683, 707961343, 708310988, 708660621,
    709010244, 709359857, 709709463, 710059062, 710408655, 710758242,
    711107825, 711457404, 711806979, 712156550, 712506119, 712855684};
  float y; unsigned int i, j;
  j = *(unsigned int*)&x; i = j » 19; j = j / 3;
  i = i - 48 * (j » 23);
  j = R3[i] + j;
  y = *(float*)&j;
  y = beta3[i] * y;
  y = fmaf(x, 0.148148147569f / (y * y), y); //k1=0.148148147569f
  return y;
}
```

Algorithm 4 rt_M1_n4_M16

```
float rt_M1_n4_M16(float x){ // x^1/4 -> I iter M1
  static const float beta3[64] = {
    0.871769690734f, 0.834112093067f, 0.800044687381f, 0.769057226318f,
    0.740733518415f, 0.714730490373f, 0.690762640254f, 0.668590321642f,
    0.648010795939f, 0.628851315150f, 0.610963714722f, 0.594220143524f,
    0.578509660115f, 0.563735495983f, 0.549812837369f, 0.536667013976f,
    1.03671471886f, 0.991932035785f, 0.951418834553f, 0.914568325382f,
    0.880885570420f, 0.849962584461f, 0.821459846568f, 0.795092367519f,
    0.770619049129f, 0.747834458255f, 0.726562396556f, 0.706650822556f,
    0.687967803906f, 0.670398262803f, 0.653841338119f, 0.638208231408f,
    1.23286851990f, 1.17961263455f, 1.13143404740f, 1.08761115970f,
    1.04755538785f, 1.01078155293f, 0.976885894227f, 0.945529500538f,
    0.916425656181f, 0.889330058602f, 0.864033171478f, 0.840354186006f,
    0.818136207298f, 0.797242384011f, 0.777552771374f, 0.758961769644f,
    0.733068007865f, 0.701401868980f, 0.672754709661f, 0.646697464736f,
    0.622880160293f, 0.601014307227f, 0.580859827980f, 0.562215204742f,
    0.544909955351f, 0.528798816637f, 0.513757197560f, 0.499677588561f,
    0.486466699380f, 0.474043157724f, 0.462335644004f, 0.451281368238f};
  static const unsigned int R4[64] = {
    797111885, 797505243, 797898586, 798291915, 798685234, 799078543,
    799471843, 799865137, 800258424, 800651706, 801044982, 801438254,
    801831523, 802224787, 802618049, 803011308, 795014733, 795408091,
    795801434, 796194763, 796588082, 796981391, 797374691, 797767985,
    798161272, 798554554, 798947830, 799341102, 799734371, 800127635,
    800520897, 800914156, 792917581, 793310939, 793704282, 794097611,
    794490930, 794884239, 795277539, 795670833, 796064120, 796457402,
    796850678, 797243950, 797637219, 798030483, 798423745, 798817004,
    799209037, 799602395, 799995738, 800389067, 800782386, 801175695,
    801568995, 801962289, 802355576, 802748858, 803142134, 803535406,
    803928675, 804321939, 804715201, 805108460};
  float y; unsigned int i, j;
  i = *(unsigned int*)&x; j = (i >> 19) & 63;
  i = R4[j] + (i >> 2); y = *(float*)&i; y = beta3[j] * y;
  y = fmaf(x, 0.105468749413f / (y * y * y), y); //k1=0.105468749413f
  return y;
}
```

5. Modifications of the Householder Method

We derive equations which enable creation of the algorithms computing $\sqrt[n]{\tilde{x}}$ using iteration based on the modified Householder's method of the second order:

$$\tilde{y}_k^{(H)} = \tilde{y}_{k-1}^{(H)} - \frac{2\left(1/f(\tilde{y}_{k-1}^{(H)})\right)'}{\left(1/f(\tilde{y}_{k-1}^{(H)})\right)''}, \qquad k \in \{1, 2, \dots\}, \tag{28}$$

where $f(y) = y^n - x$ and $\tilde{y}_k^{(H)} = \tilde{y}_{k,e_x,m}^{[n\cdot M]}(\tilde{x})$.

The introduction of the three additional parameters s_k, $\kappa_{0,k}$, and $\kappa_{1,k}$ results in a modified iteration of the following form:

$$\tilde{y}_k^{(M2)} = \frac{4n\, s_k\, \kappa_{1,k}}{n^2-1} y_{k-1}^{(M2)} \left(\frac{\kappa_{0,k}(n-1)^2}{4n} + \frac{x}{x + \frac{n+1}{n-1} s_k^n\, y_{k-1}^{(M2)^n}} \right), \tag{29}$$

$$\delta_k^{(M2)} = \frac{4n\, s_k\, \kappa_{1,k}}{n^2-1} (\delta_{k-1}^{(M2)} + 1) \left(\frac{\kappa_{0,k}(n-1)^2}{4n} + \frac{x}{x + \frac{n+1}{n-1} s_k^n\, (\delta_{k-1}^{(M2)} + 1)^n} \right) - 1. \tag{30}$$

Relative error $\delta_k^{(M2)}$, in addition to a pair of derived maxima and minima, respectively, from the maxima and minima of $k-1$ iteration errors, also has a new pair of extremes for

$$\delta_{k-1}^{(M2)} = \delta_{k-1,\mp}^{(M2)} = \frac{1}{s_k} \sqrt[n]{\frac{2n - \kappa_{0,k}(n-1) \mp 2n\sqrt{1-\kappa_{0,k}}}{\kappa_{0,k}(1+n)}} - 1\,. \tag{31}$$

We find the parameters s_k and $\kappa_{0,k}$ by solving a system of equations:

$$\delta_k^{(M2)}\big|_{\kappa_{1,k}=1,\,\delta_{k-1}^{(M2)}=\delta_{k-1,+}^{(M2)}} = \delta_k^{(M2)}\big|_{\kappa_{1,k}=1\,,\,\delta_{k-1}^{(M2)}=\delta_{k-1,\min}^{(M2)}=-\delta_{k-1,\max}^{(M2)}}\,, \tag{32}$$

$$\delta_k^{(M2)}\big|_{\kappa_{1,k}=1,\,\delta_{k-1}^{(M2)}=\delta_{k-1,-}^{(M2)}} = \delta_k^{(M2)}\big|_{\kappa_{1,k}=1\,,\,\delta_{k-1}^{(M2)}=\delta_{k-1,\max}^{(M2)}}\,. \tag{33}$$

From the equation

$$\delta_{k-1}^{(M2)}\big|_{\delta_{k-1}^{(M2)}=\delta_{k-1,+}^{(M2)}} + \delta_{k-1}^{(M2)}\big|_{\delta_{k-1}^{(M2)}=\delta_{k-1,-}^{(M2)}} = 0 \tag{34}$$

we obtain the formula for $\kappa_{1,k}$:

$$\kappa_{1,k} = 2\sqrt[n]{\kappa_{0,k}}(1+n)^{1+1/n}\left(\sqrt[n]{2n\,a_1 - \kappa_{0,k}(n-1)}\,(2\,a_2 + \kappa_{0,k}(n-1)) + \right.$$
$$\left. + \sqrt[n]{2n\,a_2 - \kappa_{0,k}(n-1)}\,(2\,a_1 + \kappa_{0,k}(n-1))\right)^{-1}, \tag{35}$$

where

$$a_1 = 1 - \sqrt{1-\kappa_{0,k}}\,, \quad a_2 = 1 + \sqrt{1-\kappa_{0,k}}.$$

The values of $\delta_{1,\max} = \delta_1^{(M2)}\big|_{\delta_0^{(M2)}=\delta_{0,\max}^{(M2)}}$ for $n=3$ and $n=4$ are given in Tables 4 and 5.

Table 4. Maximal relative errors $\delta_{1,\max}^{(M2)}$ and their corresponding numbers of correct bits $ac_{\text{bits},1} = \lfloor -\log_2 \delta_{1,\max}^{(M2)} \rfloor$ for initial approximation $\sqrt[3]{\tilde{x}} \simeq \tilde{y}_{0,e_x,m}^{[3\cdot M]}(\tilde{x})$.

	$M=1$	$M=2$	$M=4$	$M=8$
$\delta_{1,\max}$	$4.88\cdot 10^{-8}$	$1.97\cdot 10^{-9}$	$5.50\cdot 10^{-11}$	$1.19\cdot 10^{-12}$
$ac_{\text{bits},1}$	24	28	34	39

Table 5. Maximal relative errors $\delta_{1,\max}^{(M2)}$ and their corresponding numbers of correct bits $ac_{\text{bits},1} = \lfloor -\log_2 \delta_{1,\max}^{(M2)} \rfloor$ for initial approximation $\sqrt[4]{\tilde{x}} \simeq \tilde{y}_{0,e_x,m}^{[4\cdot M]}(\tilde{x})$.

	$M=1$	$M=2$	$M=4$	$M=8$
$\delta_{1,\max}$	$5.49\cdot 10^{-8}$	$2.22\cdot 10^{-9}$	$6.20\cdot 10^{-11}$	$1.34\cdot 10^{-12}$
$ac_{\text{bits},1}$	24	28	33	39

Algorithm 5 for $\sqrt[3]{x}$ and Algorithm 6 for $\sqrt[4]{x}$ implement the presented method with $M=2$, i.e., partitioning the ranges $[2^{e_{\tilde{x}}}, 2^{e_{\tilde{x}}+1}]$ into two equal parts.

We use the following form of iteration in the algorithms:

$$z_1^{(M2)} = z_0^{(M2)}\left(k_0 + k_1\cdot x/\left(x + z_0^{(M2)^n}\right)\right), \qquad \text{where} \quad z_0^{(M2)} = \beta_2 \cdot y_0^{(M2)}, \tag{36}$$

and

$$\beta_2 = \sqrt[n]{\frac{n+1}{n-1}}\,s_k\,, \qquad k_0 = \left(\frac{n-1}{n+1}\right)^{1+1/n}\kappa_{0,1}\kappa_{1,1}\,, \qquad k_1 = \frac{4n}{n^2-1}\sqrt[n]{\frac{n-1}{n+1}}\kappa_{1,1}\,. \tag{37}$$

Multiplications of parameters β_2 and coefficient tables $\beta_{e_{\tilde{x}},m}^{[n,M]}$ (given by Equation (11)) for the corresponding ranges $\tilde{x} \in [2^{e_{\tilde{x}}}(1+\frac{m-1}{M}), 2^{e_{\tilde{x}}}(1+\frac{m}{M})]$ result in tables *beta*3.

Algorithm 5 rt_M2_n3_M2_H

```
float rt_M2_n3_M2_H(float x){ // x^1/3, I iter M2
  static const float beta3[6] = {
    1.74053475068f, 1.38407386750f, 1.09646818523f,
    0.871911900134f, 1.38146334711f, 1.09854015663f};
  static const unsigned int R3[6] = {705915582, 708748133,
     711507987, 714340538, 708711784, 711544336};
  float y; unsigned int i, j;
  j = *(unsigned int*)&x; i = j >> 22; j = j / 3; i = i - 6 * (j >> 23);
  j = R3[i] + j; y = *(float*)&j; y = beta3[i] * y;
  y *= fmaf(x, 1.19055388160f / (x + y * y * y), 0.396849232120f);
  return y;
}
```

Algorithm 6 rt_M2_n4_M2_H

```
float rt_M2_n4_M2_H(float x){ // x^1/4, I iter M2
  static const float beta3[8] = {
    1.15540928653f, 0.892930883379f, 1.37402094428f, 1.06187975972f,
    1.63399548310f, 1.26279496554f, 0.971579527194f, 0.750862378903f};
  static const unsigned int R4[8] = {
    798358146, 801542222, 796260994, 799445070,
    794163842, 797347918, 800455298, 803639374};
  float y, c; unsigned int i, j;
  i = *(unsigned int*)&x; j = (i >> 22) & 7; i = R4[j] + (i >> 2);
  y = *(float*)&i; y = beta3[j] * y; c = y * y;
  y *= fmaf(x, 0.938789107238f / (x + c * c), 0.528065821580f);
  return y;
}
```

6. Approximation with a Single Magic Constant

Modified iterations M2 do not provide sufficient precision when they are used as the last step. However, when used before iteration of the M1 type, they can be used when we want to introduce only one magic constant. We calculate constant R (see (6)), for roots of the n-th order, by finding constant c (see (7)), solving the following equation:

$$\delta_0(1) = \delta_0(2^n) \tag{38}$$

where $\delta_0(\tilde{x})$ is described by Formulas (7) and (8). We get (for any value of β)

$$c = 1 - 1/n. \tag{39}$$

Parameter β is given by the formula

$$\beta = \frac{2}{2 + \delta_0(1)|_{\beta=1} + \delta_0(2^{e_{\tilde{x},\max}})|_{\beta=1}}, \quad \text{where} \quad e_{\tilde{x},\max} = \underset{e_{\tilde{x}} \in \{1,\ldots,n-1\}}{\arg\max} \ \delta_0(2^{e_{\tilde{x}}})|_{\beta=1}. \tag{40}$$

where the subscript $\beta = 1$ means that we compute δ_0 from Equation (8) with $\beta = 1$. Values of the parameters $e_{\tilde{x},\max}$ and β and relative errors $\delta_{0,\max} = \delta_0(2^{e_{\tilde{x},\max}})$ for the initial approximation, for $n \in \{2, 3, 4, 5\}$, are given in Table 6. Figure 3 presents the relative error plots for the output approximations of the cube root and fourth root.

To achieve higher accuracy, we perform two iterations. They correspond (in this order) to the modified Householder method (29) and the modified Newton–Raphson method (17). Table 6 presents a comparison of the maximal relative errors $\delta_0(2^{e_{\tilde{x},\max}})$, initial approximation, and errors obtained after applying iterations M2 and M1, respectively.

Table 6. Values of the parameters $e_{\tilde{x},\max}$ and β and maximal relative errors for approximation $\sqrt[n]{x}$ ($n \in \{2, 3, 4, 5\}$) with a single magic constant: $\delta_{0,\max} = \delta_0(2^{e_{\tilde{x},\max}})$, $\delta_{1,\max} = \delta_{1,\max}^{(M2)}$, and $\delta_{2,\max} = \delta_{1,\max}^{(M1)}$.

	$n = 2$	$n = 3$	$n = 4$	$n = 5$
$e_{\tilde{x},\max}$	1	1	2	2
β	0.9705627485	0.9716910597	0.9705627485	0.9704019654
$\delta_{0,\max}$	$2.94 \cdot 10^{-2}$	$2.83 \cdot 10^{-2}$	$2.94 \cdot 10^{-2}$	$2.96 \cdot 10^{-2}$
$\delta_{1,\max}$	$1.60 \cdot 10^{-6}$	$3.78 \cdot 10^{-6}$	$7.97 \cdot 10^{-6}$	$1.30 \cdot 10^{-5}$
$\delta_{2,\max}$	$2.55 \cdot 10^{-12}$	$1.43 \cdot 10^{-11}$	$6.35 \cdot 10^{-11}$	$1.68 \cdot 10^{-10}$

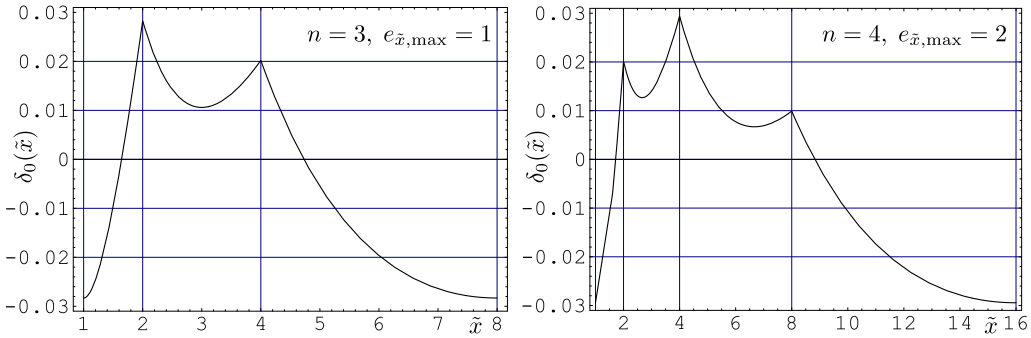


Figure 3. Error functions $\delta_0(\tilde{x})$ of the output approximations of the n-th roots, where $n \in \{3, 4\}$, with the use of one magic constant.

To implement the discussed method as a program, we need to represent each iteration in the following form:

$$z_0 = \beta_2 y_0\,,\quad z_1 = \frac{n-1}{n}\kappa_{1,2}\kappa_{0,2}\, y_1^{(M2)} = k_{1,1} z_0\left(k_{0,1} + \frac{x}{x + z_0^n}\right),\quad z_2 = y_2^{(M1)} = z_1 + \frac{k_{1,2}x}{z_1^{n-1}}\,,$$

where

$$\beta_2 = \sqrt[n]{\frac{n+1}{n-1}}\, s_1\,, \qquad k_{1,1} = \frac{4\,\kappa_{1,2}\kappa_{0,2}\kappa_{1,1}}{1+n}\sqrt[n]{\frac{n-1}{n+1}}\,,$$
$$k_{0,1} = \frac{(n-1)^2\kappa_{0,1}}{4n}\,, \qquad k_{1,2} = \frac{1}{n}\left(1 - \frac{1}{n}\right)^{n-1}\kappa_{1,2}^n\kappa_{0,2}^{n-1}\,.$$

We obtain parameters s_1, $\kappa_{0,1}$, and $\kappa_{1,1}$ using Equations (31)–(35) for $k = 1$. We then compute $\kappa_{0,2}$ and $\kappa_{1,2}$ using Formulas (18) and (19) for $k = 2$, with the error $\delta_{1,\max}^{(M1)}$ replaced by $\delta_{1,\max}^{(M2)}$. The resulting procedures for computing third and fourth roots are presented in Algorithm 7 (function *rt_3_M2M1*) and Algorithm 8 (function *rt_4_M2M1*). We use Formulas (39) and (6) to obtain magic constants R (710235477 and 799014912), while the products $\beta_2 \cdot \beta$ yield the constants $1.22486760675f$ and $1.10336983799f$.

Algorithm 7 rt_3_M2M1

```
float rt_3_M2M1(float x){ // x^(1/3)
  float y;
  unsigned int i; i=*(unsigned int*)&x;
  i=710235477+i/3; y=*(float*)&i; y*=1.22486760675f;
  y=0.794018666491f*y*(0.333066245882f+x/(x+y*y*y)); //M2
  y=fmaf(x, 0.148148148145f/(y*y), y);     //M1
  return y;
}
```

Algorithm 8 rt_4_M2M1

```
float rt_4_M2M1(float x){ // x^(1/4)
  float y, c;
  unsigned int i; i=*(unsigned int*)&x;
  i=799014912+(i»2); y=*(float*)&i; y*= 1.10336983799f;
  c=y*y;
  y=0.704661491656f*y*(0.561738942736f+x/(x+c*c)); //M2
  y=fmaf(x, 0.105468749980f/(y*y*y), y);     //M1
  return y;
}
```

7. Numerical Tests

Tests were performed on two microcontroller systems equipped with a hardware floating-point unit (FPU):

- STM32F767ZIT6 (ARM Cortex-**M7**), operating at $f = 216\,\text{MHz}$. The compiler used was `arm-none-eabi-gcc` with the following flags: `-mcpu=cortex-m7` (target core), `-mfpu=fpv4-sp-d16` (single-precision FPU), `-mfloat-abi=hard` (hardware floating-point instructions), `-mthumb` (Thumb instruction set), and `-O3` (maximum speed optimization).
- STM32L432KCU6 (ARM Cortex-**M4**), operating at $f = 80\,\text{MHz}$. The compiler used was `arm-none-eabi-gcc` with the following flags: `-mcpu=cortex-m4`, `-mfpu=fpv4-sp-d16`, `-mfloat-abi=hard`, `-mthumb`, and `-O3`.

Tests were also conducted on two systems without an FPU, performing floating-point computations using fast integer units such as MAC, DSP extensions, 8/16-bit SIMD operations, and a superscalar pipeline:

- ESP32-D0WDQ6 (Xtensa **LX6**), operating at $f = 240\,\text{MHz}$. The compiler used was `xtensa-esp32-elf-gcc` with the flag `-O3`.
- ESP32-S3 (Xtensa **LX7**), operating at $f = 240\,\text{MHz}$. The compiler used was `xtensa-esp32s3-elf-gcc` with the flag `-O3`.

Tests were also performed on one system without an FPU and without any DSP/-MAC/SIMD acceleration units:

- ESP32-C3 (**RISC-V** RV32IMC), operating at $f = 160\,\text{MHz}$. The compiler used was `riscv32-esp-elf-gcc` with the flag `-O3`.

Extreme values of relative errors ($\delta_{\min}$, $\delta_{\max}$) and relative mean squared errors (RMSEr) were determined by computing the errors for all float numbers x_i in the intervals $[1, 2^n)$:

$$\delta_{\min} = \min_{i\in\{0,1,\ldots,N\}} \delta(x_i)\,, \quad \delta_{\max} = \max_{i\in\{0,1,\ldots,N\}} \delta(x_i)\,, \quad \text{RMSEr} = \sqrt{\frac{1}{N+1}\sum_{i=0}^{N}\delta(x_i)^2}\,,$$

where

$$x_i = 2^{\lfloor i/2^{23} \rfloor} \left(1 + \frac{i \mod 2^{23}}{2^{23}}\right), \quad N = 2^{23}\, n - 1\,,$$

and

$$n = 3\,, \qquad \delta(x_i) = \frac{\tilde{y}_n(x_i) - \mathrm{cbrt}((\mathbf{double})x_i)}{\mathrm{cbrt}((\mathbf{double})x_i)} \qquad \text{for cube root,}$$

$$n = 4\,, \qquad \delta(x_i) = \frac{\tilde{y}_n(x_i) - \mathrm{pow}((\mathbf{double})x_i, 0.25)}{\mathrm{pow}((\mathbf{double})x_i, 0.25)} \qquad \text{for fourth root.}$$

In the program that tests the accuracy of the algorithms, `cbrt(○)` and `pow(○,○)` refer to functions with the same names that compute cube roots and powers of double-precision floating-point numbers (**double**). Tables 7 and 8 list the extreme relative errors and the RMSEr values. All tested configurations produced identical values.

It is worth emphasizing that the obtained values remain valid for the other intervals A_j (see Equation (2)) involving float numbers, with the exception of subnormal numbers.

Table 7. Extreme values of relative errors and execution times expressed in cycles ($c = t \cdot f$) for cube root approximations. Extreme values of relative errors were determined for all float numbers and were identical for all tested systems.

Method	Relative Errors			Execution Times				
	$\delta_{\min}$	$\delta_{\max}$	RMSEr	M7	M4	LX6	LX7	RISC-V
`cbrtf`	$-2.59 \cdot 10^{-7}$	$2.31 \cdot 10^{-7}$	$5.30 \cdot 10^{-8}$	101	93	222	229	1416
`rt_M1_n3`	$-8.44 \cdot 10^{-8}$	$9.39 \cdot 10^{-8}$	$2.78 \cdot 10^{-8}$	69	58	147	156	1616
`rt_M1_n3_M16`	$-8.90 \cdot 10^{-8}$	$9.53 \cdot 10^{-8}$	$2.78 \cdot 10^{-8}$	48	45	83	86	1058
`rt_M2_n3_M2_H`	$-1.30 \cdot 10^{-7}$	$1.79 \cdot 10^{-7}$	$4.63 \cdot 10^{-8}$	55	47	93	94	1394
`rt_3_M2M1`	$-8.88 \cdot 10^{-8}$	$9.40 \cdot 10^{-8}$	$2.78 \cdot 10^{-8}$	69	54	144	154	1932

Table 8. Extreme values of relative errors and execution times expressed in cycles ($c = t \cdot f$) for fourth root approximations. Extreme values of relative errors were determined for all float numbers and were identical for all tested systems.

Method	Relative Errors			Execution Times				
	$\delta_{\min}$	$\delta_{\max}$	RMSEr	M7	M4	LX6	LX7	RISC-V
$2 \times$ `sqrtf`	$-8.94 \cdot 10^{-8}$	$8.93 \cdot 10^{-8}$	$2.80 \cdot 10^{-8}$	42	37	152	137	1532
`rt_M1_n4`	$-8.79 \cdot 10^{-8}$	$8.87 \cdot 10^{-8}$	$2.74 \cdot 10^{-8}$	71	58	148	160	1862
`rt_M1_n4_M16`	$-8.85 \cdot 10^{-8}$	$9.17 \cdot 10^{-8}$	$2.74 \cdot 10^{-8}$	45	43	80	81	1173
`rt_M2_n4_M2_H`	$-1.62 \cdot 10^{-7}$	$1.29 \cdot 10^{-7}$	$3.97 \cdot 10^{-8}$	49	41	84	86	1396
`rt_4_M2M1`	$-8.90 \cdot 10^{-8}$	$8.73 \cdot 10^{-8}$	$2.74 \cdot 10^{-8}$	70	55	146	154	2069

All data on extreme values of relative errors for cube root approximations are contained in Table 7. In the first row, we present, as a benchmark, the standard `cbrtf` function, as implemented on microcontrollers such as STM32 and ESP32. In the next rows of Table 7, we show the data for four codes. All of them are based on the "magic constant approach", which consists of determining the initial approximation using a magic constant and then performing Newton–Raphson-like or Householder-like iterations. Table 8 contains data corresponding to fourth roots. Its structure mirrors that of Table 7, but instead of using `cbrtf` as the reference function, we apply double square rooting: `sqrtf(sqrtf(○))`, denoted as $2 \times$ `sqrtf`.

The numerical experiments confirm that all proposed algorithms achieve high accuracy, with relative errors largely influenced by machine precision rather than by the iterative schemes themselves. Although the observed error ranges are wider than the theoretical bounds, all cube root methods consistently outperform the standard cbrtf function in terms of precision.

The next part of the analysis focuses on the methods' execution-time performance. All functions were tested on 1000 independently generated sets of random numbers, each consisting of 1400 elements. For each set, all functions were executed sequentially, and after a given function processed the entire set, its execution time was recorded in microseconds. The value t represents the average execution time of 1,400,000 calls of each function and is computed as the mean of 1000 such measurements. Table 7 presents these results expressed in processor cycles as $f \cdot t$. No significant deviations from the mean value were observed in the collected measurements.

Execution times vary across processors; however, on microcontrollers equipped with an FPU or DSP/MAC/SIMD acceleration units, the proposed algorithms are considerably faster than `cbrtf`. On devices without acceleration units (e.g., ESP32-C3), only the single-iteration variants provide a clear performance advantage, while the double-iteration versions achieve comparable speed.

The methods from Table 8 exhibit accuracy and timing similar to their counterparts in Table 7. Their precision matches that of the double-square-root approach; however, on STM32 platforms with hardware support for `sqrtf`, two-iteration schemes are slower, whereas single-iteration ones reach similar performance. On ESP32-D0WDQ6 and ESP32-S3, which lack hardware support for `sqrtf` or reciprocal operations, the two-iteration algorithms perform comparably, while the single-iteration algorithms are markedly faster.

8. Memory Usage Analysis

The memory usage analysis was performed using identical algorithmic functions. For each of the evaluated platforms, a separate compilation was carried out using the `-O3` optimization level. To enable analysis of each function individually—in terms of both its Flash code size and its generated SU stack report—the `-fno-inline` flag was used to prevent the compiler from inlining functions. Additionally, the `-fstack-usage` flag was applied to generate SU reports indicating the stack usage of each function. The distribution of program code and static data in Flash memory was analyzed based on the ELF output.

In Tables 9–12, two categories of memory are distinguished: program memory (Program) and static data (Data). In the ESP32-D0WDQ6 and ESP32-S3 platforms, static data are stored in Flash memory but are copied to RAM during system startup (these entries are marked in Tables 10 and 11 as $\mathrm{DATA_{F/R}}$). The ESP32-C3 platform uses the same mechanism, as shown in Table 12. In contrast, on STM32F7 and STM32L4, all static data remained in Flash and were not copied to RAM; hence, Table 9 uses the designation $\mathrm{DATA_F}$.

Table 9. Comparison of program FLASH memory (Program) and FLASH data memory ($\mathrm{DATA_F}$) (in bytes) for Algorithms 1–8 based on ELF analysis, on STM32F767ZIT6 and STM32L432KCU6.

Method (n = 3)	Program	$\mathrm{DATA_F}$	Method (n = 4)	Program	$\mathrm{DATA_F}$
`rt_M1_n3`	104B	24B	`rt_M1_n4`	96B	32B
`rt_M1_n3_M16`	80B	384B	`rt_M1_n4_M16`	68B	512B
`rt_M2_n3_M2_H`	100B	48B	`rt_M2_n4_M2_H`	80B	64B
`rt_3_M2M1`	112B	0B	`rt_4_M2M1`	104B	0B

Table 10. Comparison of program FLASH memory (Program) and FLASH/RAM data memory ($DATA_{F/R}$) (in bytes) for Algorithms 1–8 based on ELF analysis, on ESP32-D0WDQ6 (Xtensa LX6).

Method ($n = 3$)	Program	$DATA_{F/R}$	Method ($n = 4$)	Program	$DATA_{F/R}$
`rt_M1_n3`	129B	24B	`rt_M1_n4`	123B	32B
`rt_M1_n3_M16`	91B	384B	`rt_M1_n4_M16`	76B	512B
`rt_M2_n3_M2_H`	108B	48B	`rt_M2_n4_M2_H`	92B	64B
`rt_3_M2M1`	119B	0B	`rt_4_M2M1`	117B	0B

Table 11. Comparison of program FLASH memory (Program) and FLASH/RAM data memory ($DATA_{F/R}$) (in bytes) for Algorithms 1–8 based on ELF analysis, on ESP32-S3 (Xtensa LX7).

Method ($n = 3$)	Program	$DATA_{F/R}$	Method ($n = 4$)	Program	$DATA_{F/R}$
`rt_M1_n3`	133B	24B	`rt_M1_n4`	127B	32B
`rt_M1_n3_M16`	91B	384B	`rt_M1_n4_M16`	76B	512B
`rt_M2_n3_M2_H`	112B	48B	`rt_M2_n4_M2_H`	96B	64B
`rt_3_M2M1`	127B	0B	`rt_4_M2M1`	121B	0B

Table 12. Comparison of program FLASH memory (Program) and FLASH/RAM data memory ($DATA_{F/R}$) (in bytes) for Algorithms 1–8 based on ELF analysis, on ESP32-C3 (RISC-V).

Method ($n = 3$)	Program	$DATA_{F/R}$	Method ($n = 4$)	Program	$DATA_{F/R}$
`rt_M1_n3`	182B	24B	`rt_M1_n4`	178B	32B
`rt_M1_n3_M16`	124B	384B	`rt_M1_n4_M16`	110B	512B
`rt_M2_n3_M2_H`	168B	48B	`rt_M2_n4_M2_H`	142B	64B
`rt_3_M2M1`	192B	0B	`rt_4_M2M1`	188B	0B

The ELF analysis showed that the program code size was similar across all platforms and ranged from several dozen to slightly over one hundred bytes. Static tables ranged from a few dozen bytes to 512 B. Differences between platforms resulted solely from the different compilers (Xtensa, RISC-V, and ARM) and their optimization strategies, despite identical source code.

The stack usage analysis revealed clear differences between architectures. On ESP32-D0WDQ6 and ESP32-S3, each function used exactly 48 B of stack memory, while on ESP32-C3, the stack usage was 16 B. On both STM32 platforms, all analyzed functions reported 0 B stack usage, because all local variables were allocated in registers and no stack frame was required. As a result, STM32F7 and STM32L4 did not use RAM either for static data or for the stack in the analyzed functions.

9. Conclusions

This work presented a complete derivation, analysis, and implementation of fast algorithms for computing n-th roots, with detailed specializations for the cubic and quartic cases. The proposed framework generalizes the structural advantages of inverse-square-root methods (see, e.g., [27,29]) and provides a mathematically transparent error analysis, with theoretical bounds confirmed in practice up to machine precision. Although the observed relative errors are unavoidably affected by floating-point limitations—particularly the 2^{-24} granularity of single-precision arithmetic—the experimental results remain consistent with the analytical predictions. In all cases, the accuracy of the cubic root solvers clearly exceeds that of the standard library cbrtf function, and the method of [35] is less accurate by more than two orders of magnitude.

The hardware evaluation, conducted on five representative microcontroller platforms (including both FPU-equipped and non-FPU devices), confirmed the efficiency and porta-

bility of the proposed approach. In particular, on all microcontrollers except ESP32-C3 (RISC-V), every implementation from Table 7 achieves markedly better execution times than cbrtf. The quartic root implementations from Table 8 reach accuracy and speed comparable to double square rooting, with single-iteration schemes matching its performance on STM32 devices, and two-iteration variants being slower due to the availability of hardware-accelerated sqrtf.

Overall, the results demonstrate that the proposed family of algorithms provides a high-performance, low-power, and accurate solution for embedded, wearable, and real-time signal-processing systems. The combination of rigorous analysis, provable accuracy, and strong cross-platform performance suggests that these methods offer a compelling alternative to currently used root-extraction routines in resource-constrained environments.

Author Contributions: Conceptualization, C.J.W.; methodology, C.J.W.; software, C.J.W.; validation, C.J.W.; formal analysis, C.J.W.; data curation, C.J.W. and M.J.; writing—original draft preparation, C.J.W., J.L.C. and M.J.; writing—review and editing, C.J.W. and J.L.C.; visualization, C.J.W.; supervision, J.L.C. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: Data is contained within the article.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Corey, R.M.; Singer, A.C. Modeling the Effects of Dynamic Range Compression on Signals in Noise. *J. Acoust. Soc. Am.* **2021**, *150*, 159–170. [CrossRef]
2. Fattal, R.; Lischinski, D.; Werman, M. Gradient Domain High Dynamic Range Compression. In *Seminal Graphics Papers: Pushing the Boundaries*; Whitton, M.C., Ed.; ACM: New York, NY, USA, 2023; Volume 2, pp. 671–678.
3. Severoglu, N.; Demir, Y.; Kaplan, N.H.; Kucuk, S. ACGC: Adaptive Chrominance Gamma Correction for Low-Light Image Enhancement. *J. Vis. Commun. Image Represent.* **2025**, *107*, 104402. [CrossRef]
4. Nie, Z.; Long, T.; Zhou, Z. Research on Rotational Object Recognition Based on HSV Color Space and Gamma Correction. In Proceedings of the 6th International Symposium on Advanced Technologies and Applications in the Internet of Things, Kusatsu, Japan, 19–22 August 2024; pp. 19–22.
5. Ngo, D.; Kang, B. Taylor-Series-Based Reconfigurability of Gamma Correction in Hardware Designs. *Electronics* **2021**, *10*, 1959. [CrossRef]
6. Oprzędkiewicz, K.; Rosół, M.; Żegleń-Włodarczyk, J. The Frequency and Real-Time Properties of the Microcontroller Implementation of Fractional-Order PID Controller. *Electronics* **2021**, *10*, 524. [CrossRef]
7. Ferdi, Y. Some applications of fractional order calculus to design digital filters for biomedical signal processing. *J. Mech. Med. Biol.* **2012**, *12*, 1240008. [CrossRef]
8. Zhu, J.Y.; Snowden, J.C.; Verdejo, J.; Chen, E.; Zhang, P.; Ghaednia, H.; Schwab, J.H.; Mueller, S. EIT-kit: An Electrical Impedance Tomography Toolkit for Health and Motion Sensing. In Proceedings of the 34th Annual ACM Symposium on User Interface Software and Technology (UIST '21), Virtual Event, 10–14 October 2021; pp. 400–413.
9. Telceken, M. A new feature extraction method for AI based classification of heart sounds: Dual-frequency cepstral coefficients (DFCCs). *Eur. Phys. J. Spec. Top.* **2025**, *234*, 4699–4710. [CrossRef]
10. Fu, J.B.; Zhang, Y.; Liu, J.; Lian, X.C.; Tang, J.; Zhu, F. Pharmacometabonomics: Data processing and statistical analysis. *Brief. Bioinform.* **2021**, *22*, 1–25. [CrossRef]
11. Echoukairi, H.; Ettaki, W.; Ghmary, M.E. Enhancing Call Detail Record Prediction with Machine Learning for Communication. In *Connected Objects, Artificial Intelligence, Telecommunications and Electronics Engineering. COCIA 2025*; Mejdoub, Y., Elamri, A., Kardouchi, M., Eds.; Lecture Notes in Networks and Systems; Springer: Cham, Switzerland, 2026; Volume 1584.
12. Dike, A.O.; Chikezie, D.C.; Otuonye, E.L. Application of cube root transformation of error component of multiplicative time stress model. *Eur. J. Stat. Probab.* **2020**, *8*, 25–40.
13. Murugesan, N.; Ramasamy, A.M.S. A numerical algorithm for n^{th} root. *J. Fundam. Sci.* **2011**, *7*, 35–40. [CrossRef]
14. Jo, J.; Park, I.-C. Low-Latency Low-Cost Architecture for Square and Cube Roots. *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.* **2017**, *E100-A*, 1951–1955. [CrossRef]

15. Aggarwal, S.; Garg1, N.; Kaur, G. Performance Evaluation of Various Dispersion Compensation Modules. *Wirel. Pers. Commun.* **2022**, *123*, 2011–2025. [CrossRef]
16. Tukey, J.W. *Exploratory Data Analysis*; Addison-Wesley: Reading, MA, USA, 1977.
17. Box, G.E.P.; Cox, D.R. An Analysis of Transformations. *J. R. Stat. Soc. Ser. B* **1964**, *26*, 211–243. [CrossRef]
18. Zheng, H.; Cai, A.; Zhou, Q.; Xu, P.T.; Zhao, L.C.; Li, C.; Dong, B.J.; Gao, H.C. Optimal preprocessing of serum and urine metabolomic data fusion for staging prostate cancer through design of experiment. *Anal. Chim. Acta* **2017**, *991*, 68–75. [CrossRef]
19. Li, A.; Zheng, C.; Peng, R.; Li, X. On the Importance of Power Compression and Phase Estimation in Monaural Speech Dereverberation. *JASA Express Lett.* **2021**, *1*, 014802. [CrossRef]
20. Bendib, H.-E.; Bahri, M.; Boumehraz, M.; Mallem, A.; Hamzi, M. Implementation of a New Versatile Bio-Potential Measurement System. *J. Circuits Syst. Comput.* **2023**, *32*, 2350128. [CrossRef]
21. Rakhmatulin, I. Low-Cost Shield MicroBCI to Measure EEG with STM32. *arXiv* **2025**, arXiv:2509.16229.
22. Santos, P.; Marquês, F.; Quintão, C.; Quaresma, C. Electromyographic and Kinematic Analysis of the Upper Limb During Drinking and Eating Using a Wearable Device Prototype. *Sensors* **2025**, *25*, 5227. [CrossRef]
23. Altamimi, A.; Ben Youssef, B. Leveraging Seed Generation for Efficient Hardware Acceleration of Lossless Compression of Remotely Sensed Hyperspectral Images. *Electronics* **2024**, *13*, 2164. [CrossRef]
24. Kusaka, T.; Tanaka, T. Pseudo-Normalization via Integer Fast Inverse Square Root and Its Application to Fast Computation without Division. *Electronics* **2024**, *13*, 2955. [CrossRef]
25. Yu, B. Averaging-Based Method for Real-Time Estimation of Voltage Effective Value in Grid-Connected Inverters. *Electronics* **2025**, *14*, 3733. [CrossRef]
26. Qin, L.; Wang, W.; Shi, M.; Liang, Y.; Tan, P. Attitude Calculation Method of Drilling Tools Based on Cross-Correlation Extraction and ASRUKF. *Electronics* **2024**, *13*, 1707. [CrossRef]
27. Moroz, L.; Walczyk, C.J.; Hrynchyshyn, A.; Holimath, V.; Cieśliński, J.L. Fast calculation of inverse square root with the use of magic constant—Analytical approach. *Appl. Math. Comput.* **2018**, *316*, 245–255. [CrossRef]
28. Walczyk, C.J.; Moroz, L.; Cieśliński, J.L. A modification of the fast inverse square root algorithm. *Computation* **2019**, *7*, 41. [CrossRef]
29. Walczyk, C.J.; Moroz, L.; Cieśliński, J.L. Improving the accuracy of the fast inverse square root by modifying Newton-Raphson corrections. *Entropy* **2021**, *23*, 86. [CrossRef]
30. Blinn, J. Floating-point tricks. *IEEE Comput. Graph. Appl. Mag.* **1997**, *17*, 80–84. [CrossRef]
31. Moroz, L.; Samotyy, V.; Horyachyy, O.; Dzelendzyak, U. Algorithms for calculating the square root and inverse square root based on the second-order Householder's method. In Proceedings of the 2019 10th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS), Metz, France, 18–21 September 2019; pp. 436–442.
32. Moroz, L.; Samotyy, V. Efficient floating-point division for digital signal processing application. *IEEE Signal Process. Mag.* **2019**, *36*, 159–163. [CrossRef]
33. Walczyk, C.J.; Moroz, L.V.; Samotyy, V.; Cieśliński, J.L. Optimal approximation of the 1/x function using Chebyshev polynomials and magic constants. *ACM Trans. Math. Softw.* **2025**, *51*, 2. [CrossRef]
34. Walczyk, C.; Gepner, P.; Ltaief, H.; Gavkalova, N. On Floating Point Approximation of the Reciprocal Cube Root Function. In *Computational Science—ICCS 2025 Workshops*; Paszynski, M., Barnard, A.S., Zhang, Y.J., Eds.; Springer Nature: Cham, Switzerland, 2025; pp. 89–103.
35. Moroz, L.; Samotyy, V.; Walczyk, C.J.; Cieśliński, J.L. Fast calculation of cube and inverse cube root using a magic constant and its implementation on microcontrollers. *Energies* **2021**, *14*, 1058. [CrossRef]
36. Moroz, L.; Samotyy, V.; Walczyk, C. Fast Algorithms for Computing Floating-Point Reciprocal Cube Root Functions. *Int. J. Comput. Sci. Netw. Secur.* **2023**, *23*, 84–90. [CrossRef]
37. Day, M. Generalising the Fast Reciprocal Square Root Algorithm. *arXiv* **2023**, arXiv:2307.15600.
38. IEEE Computer Society; Standards Committee; Working Group of the Microprocessor Standards Subcommittee; American National Standards Institute. *IEEE Standard for Binary Floating-Point Arithmetic*; IEEE: New York, NY, USA, 1985; Volume 754.

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.