entropy

Article

Improving the Accuracy of the Fast Inverse Square Root by Modifying Newton–Raphson Corrections

Cezary J. Walczyk [1], Leonid V. Moroz [2] and Jan L. Cieśliński [1,*]

[1] Wydział Fizyki, Uniwersytet w Białymstoku, ul. Ciołkowskiego 1L, 15-245 Białystok, Poland; c.walczyk@uwb.edu.pl
[2] Department of Security Information and Technology, Lviv Polytechnic National University, st. Kn. Romana 1/3, 79000 Lviv, Ukraine; moroz_lv@polynet.lviv.ua
* Correspondence: j.cieslinski@uwb.edu.pl

Abstract: Direct computation of functions using low-complexity algorithms can be applied both for hardware constraints and in systems where storage capacity is a challenge for processing a large volume of data. We present improved algorithms for fast calculation of the inverse square root function for single-precision and double-precision floating-point numbers. Higher precision is also discussed. Our approach consists in minimizing maximal errors by finding optimal magic constants and modifying the Newton–Raphson coefficients. The obtained algorithms are much more accurate than the original fast inverse square root algorithm and have similar very low computational costs.

Keywords: approximation of functions; floating-point arithmetic; Newton–Raphson method; inverse square root; magic constant

Citation: Walczy, C.J.; Moroz, L.V.; Cieśliński, J.L. Improving the Accuracy of the Fast Inverse Square Root by Modifying Newton–Raphson Corrections. *Entropy* **2021**, *23*, 86. https://doi.org/10.3390/e23010086

Received: 29 November 2020
Accepted: 6 January 2021
Published: 9 January 2021

1. Introduction

Efficient performance of algebraic operations in the framework of floating-point arithmetic is a subject of considerable importance [1–6]. Approximations of elementary functions are crucial in scientific computing, computer graphics, signal processing, and other fields of engineering and science [7–10]. Our aim is to compute elementary functions at a very low computational cost without using memory resources. Direct evaluation of functions could be of interest in any systems where storage capabilities challenge the processing of a large volume of data. This problem is crucial, for instance, in high-energy physics experiments [11–13].

In this paper, we consider approximation and fast computation of the inverse square root function, which has numerous applications (see [8,10,14–17]), especially in 3D computer graphics, where it is needed for normalization of vectors [4,18,19]. The proposed algorithms are aimed primarily at floating-point platforms with limited hardware resources, such as microcontrollers, some field-programmable gate arrays (FPGAs), and graphics processing units (GPUs) that cannot use fast look-up table (LUT)-based hardware instructions, such as SSE (i.e., Streaming SIMD (single instruction, multiple data) Extensions) or Advanced Vector Extensions (AVX). We mean here devices and chips containing floating-point multipliers, adders–subtractors, and fused-multiply adders. Therefore, our algorithms can easily be implemented on such a platform. We also offer them as an alternative to library functions that provide full precision, but are very time consuming. This was the motivation for considering the cases of higher precision in Section 3.2. By selecting the precision and number of iterations, the desired accuracy can be obtained. We propose the use of our codes as direct insertions into more general algorithms without referring to the corresponding library of mathematical functions. In the double-precision mode, most modern processors do not have SSE instructions like rsqrt (such instructions appeared only in AVX-512, which is supported only by the latest processor models). In such cases, one can use our algorithms (with the appropriate number of iterations) as a fast alternative to the library function 1/sqrt(x).

In most cases, the initial seed needed to start the approximation is taken from a memory-consuming look-up table (LUT), although the so-called "bipartite table methods" (actually used on many current processors) make it possible to considerably lower the table sizes [20,21]. The "fast inverse square root" code works in a different way. It produces the initial seed in a cheap and effective way using the so-called magic constant [4,19,22–25]. We point out that this algorithm is still useful in numerous software applications and hardware implementations (see, e.g., [17,26–30]). Recently, we presented a new approach to the fast inverse square root code *InvSqrt*, presenting a rigorous derivation of the well-known code [31]. Then, this approach was used to construct a more accurate modification (called *InvSqrt1*) of the fast inverse square root (see [32]). It will be developed and generalized in the next sections, where we will show how to increase the accuracy of the *InvSqrt* code without losing its advantages, including the low computational cost. We will construct and test two new algorithms, *InvSqrt2* and *InvSqrt3*.

The main idea of the algorithm *InvSqrt* consists in interpreting bits of the input floating-point number as an integer [31]. In this paper, we consider positive floating-point normal numbers

$$x = (1 + m_x)2^{e_x}\,, \qquad m_x \in [0, 1)\,, \quad e_x \in \mathbb{Z}\,, \tag{1}$$

and, in Section 3.1, we also consider subnormal numbers. We use the standard IEEE-754, where single-precision floating-point numbers are encoded with 32 bits. For positive numbers, the first bit is zero. The next eight bits encode e_x, and the remaining 23 bits represent the mantissa m_x. The same 32 bits can be treated as an integer I_x:

$$I_x = N_m(B + e_x + m_x), \tag{2}$$

where $N_m = 2^{23}$ and $B = 127$. In this case $B + e_x$ is a natural number not exceeding 254. The case of higher precision is analogous (see Section 3.2).

The crucial step of the algorithm *InvSqrt* consists in shifting all bits to the right by one bit and subtracting the result of this operation from a "magic constant" R (and the optimum value of R has to be guessed or determined). In other words,

$$I_{y_0} = R - \lfloor I_x/2 \rfloor\,. \tag{3}$$

Originally, R was proposed as $0x5F3759DF$ (see [19,23]). Interpreted in terms of floating-point numbers, I_{y_0} approximates the inverse square root function surprisingly well ($y_0 \approx y = 1/\sqrt{x}$). This trick works because (3) is close to dividing the floating-point exponent by -2. The number R is needed because the floating-point exponents are biased (see (2)).

The magic constant R is usually given as a hexadecimal integer. The same bits encode the floating-point number R_f with an exponent e_R and mantissa m_R. According to (1), $R_f = (1 + m_R)2^{e_R}$. In [31], we have shown that if $e_R = \frac{1}{2}(B-1)$ (e.g., $e_R = 63$ in the 32-bit case), then the function (3) (defined on integers) is equivalent to the following piece-wise linear function (when interpreted in terms of corresponding floating-point numbers):

$$\tilde{y}_0(\tilde{x}, t) = \begin{cases} \frac{1}{8}(6 - 2\tilde{x} + t) & \text{for } \tilde{x} \in [1, 2) \\ \frac{1}{8}(4 + t - \tilde{x}) & \text{for } \tilde{x} \in [2, t) \\ \frac{1}{16}(8 + t - \tilde{x}) & \text{for } \tilde{x} \in [t, 4) \end{cases} \tag{4}$$

where

$$t = 2 + 4m_R + 2\mu_{\tilde{x}} N_m^{-1}, \tag{5}$$

m_R is the mantissa of R (i.e., $m_R := N_m^{-1}R - \lfloor N_m^{-1}R \rfloor$), and, finally, $\mu_{\tilde{x}} = 0$ or $\mu_{\tilde{x}} = 1$ depending on the parity of the last digit of the mantissa of $\tilde{x}$.

The function $\mu_{\tilde{x}}$ is two-valued, so a given parameter t may correspond to either two values of R or one value of R (when the term containing $\mu_{\tilde{x}}$ has no influence on the bits of the mantissa m_R). The function $y = 1/\sqrt{x}$, the function (3), and all Newton–Raphson corrections considered below are invariant under the scaling $\tilde{x} = 2^{-2n}x$ and $\tilde{y} = 2^n y$ for any integer n. Therefore, we can confine ourselves to numbers from the interval $[1, 4)$. Here and in the following, the tilde always denotes quantities defined on the interval $[1, 4)$.

In this paper, we focus on the Newton–Raphson corrections, which form the second part of the *InvSqrt* code. Following and developing ideas presented in our recent papers [31,32], we propose modifications of the Newton–Raphson formulas, which result in algorithms that have the same or similar computational cost as/to *InvSqrt*, but improve the accuracy of the original code, even by several times. The modifications consist in changing both the Newton–Raphson coefficients and the magic constant. Moreover, we extend our approach to subnormal numbers and to higher-precision cases.

2. Modified Newton–Raphson Formulas

The standard Newton–Raphson corrections $\tilde{y}_1$ and $\tilde{y}_2$ for the zeroth approximation $\tilde{y}_0$ given by (4) are given by the following formulas:

$$\begin{aligned} \tilde{y}_1(\tilde{x}, t) &= \frac{3}{2}\tilde{y}_0(\tilde{x}, t) - \frac{1}{2}\tilde{x}\,\tilde{y}_0^3(\tilde{x}, t)\,, \\ \tilde{y}_2(\tilde{x}, t) &= \frac{3}{2}\tilde{y}_1(\tilde{x}, t) - \frac{1}{2}\tilde{x}\,\tilde{y}_1^3(\tilde{x}, t)\,, \end{aligned} \tag{6}$$

(analogous formulas hold for the next corrections as well; see [31]). The relative error functions $\tilde{\delta}_j(\tilde{x}, t)$ (where $j = 0, 1, 2, \ldots$) can be expressed as:

$$\tilde{\delta}_j(\tilde{x}, t) = \sqrt{\tilde{x}}\,\tilde{y}_j(\tilde{x}, t) - 1\,. \tag{7}$$

The function $\tilde{\delta}_0(\tilde{x}, t)$, which is very important for the further analysis, is thoroughly described and discussed in [31]. Using (7), we substitute $\tilde{y}_j = (1 + \tilde{\delta}_j)/\sqrt{\tilde{x}}$ (for $j = 0, 1, 2, \ldots$) into (6), $\tilde{x}$ cancels out, and the formulas (6) assume the following form:

$$\tilde{\delta}_j = -\frac{1}{2}\tilde{\delta}_{j-1}^2(3 + \tilde{\delta}_{j-1})\,, \quad (j = 1, 2, \ldots), \tag{8}$$

where $\tilde{\delta}_j = \tilde{\delta}_j(\tilde{x}, t)$. We immediately see that every correction increases the accuracy, even by several orders of magnitude (due to the factor $\tilde{\delta}_{j-1}^2$). Thus, a very small number of corrections is sufficient to reach the machine precision (see the end of Section 4).

The above approximations depend on the parameter t (which can be expressed by the magic constant R, see (5)). The best approximation is obtained for $t = t_k$ minimizing $||\tilde{\delta}_k(t)||$, i.e.,

$$||\tilde{\delta}_k(t_k)|| = \inf_{t \in (2,4)} ||\tilde{\delta}_k(t)|| \equiv \inf_{t \in (2,4)} \left(\sup_{\tilde{x} \in [1,4)} |\tilde{\delta}_k(\tilde{x}, t)| \right). \tag{9}$$

In this paper we confine ourselves to the case $t = t_1$ (i.e., we assume $t_2 = t_1$) because the more general case (where the magic constant is also optimized with respect to the assumed number of iterations) is much more cumbersome, and the related increase in accuracy is negligible. Then, we get

$$t_1^{(0)} \approx 3.7298003, \qquad R^{(0)} = 0x5F375A86, \tag{10}$$

for details, see [31]. The theoretical relative errors are given by

$$\Delta_{1\,\max}^{(0)} \equiv ||\tilde{\delta}_1(t_1)|| \approx 1.75118 \cdot 10^{-3}, \quad \Delta_{2\,\max}^{(0)} \equiv ||\tilde{\delta}_2(t_1)|| \approx 4.60 \cdot 10^{-6}. \tag{11}$$

The superscript (0) indicates values corresponding to the algorithm *InvSqrt* (other superscripts will denote modifications of this algorithm).

The idea of increasing the accuracy by a modification of the Newton–Raphson formulas is motivated by the fact that $\tilde{\delta}_k(\tilde{x},t) \leqslant 0$ for any $\tilde{x}$ (see [31,32]). Therefore, we can try to shift the graph of $\tilde{\delta}_1$ upwards (making it more symmetric with respect to the horizontal axis). Then, the errors of the first correction are expected to decrease twice and the errors of the second correction are expected to decrease by about eight times (for more details, see [32]). Indeed, according to (8), reducing the first correction by a factor of 2 will reduce the second correction by a factor of 4. The second correction is also non-positive, so we may shift the graph of $\tilde{\delta}_2$, once more improving the accuracy by the factor of 2. This procedure can be formalized by postulating the following modification of the Newton–Raphson formulas (6):

$$\begin{aligned} \tilde{y}_1 &= \tfrac{1}{2}\tilde{y}_0(3-\tilde{y}_0^2\,\tilde{x}) + \tfrac{1}{2}d_1(a_1\tilde{y}_0 + b_1\tilde{y}_1), \\ \tilde{y}_2 &= \tfrac{1}{2}\tilde{y}_1(3-\tilde{y}_1^2\,\tilde{x}) + \tfrac{1}{2}d_2(a_2\tilde{y}_1 + b_2\tilde{y}_2), \end{aligned} \tag{12}$$

where $a_k + b_k = 1$ for $k = 1,2$. Thus, we have four independent parameters (d_1, d_2, a_1, and a_2) to be determined. In other words,

$$\begin{aligned} \tilde{y}_1 &= c_{11}\,\tilde{y}_0 - c_{21}\,\tilde{x}\,\tilde{y}_0^3, \\ \tilde{y}_2 &= c_{12}\,\tilde{y}_1 - c_{22}\,\tilde{x}\,\tilde{y}_1^3, \end{aligned} \tag{13}$$

where four coefficients c_{jk} can be expressed by the four coefficients a_k and d_k:

$$c_{1k} = \frac{3 + a_k d_k}{2-(1-a_k)d_k}\,, \qquad c_{2k} = \frac{1}{2-(1-a_k)d_k} \qquad (k=1,2)\,. \tag{14}$$

We point out that the Newton–Raphson corrections and any of their modifications of the form (13) are obviously invariant with respect to the scaling mentioned at the end of Section 1. Therefore, we can continue to confine our analysis to the interval $[1,4)$.

Below, we present three different algorithms (*InvSqrt1*, *InvSqrt2*, *InvSqrt3*) constructed along the above principles (the last two of them are first introduced in this paper). They will be denoted by superscripts in parentheses, e.g., $\tilde{y}_k^{(N)}$ means the kth modified Newton–Raphson correction to the algorithm *InvSqrt N*. We always assume that the zeroth approximation is given by (4), i.e.,

$$\tilde{y}_0^{(N)} = \tilde{y}_0 \qquad (N = 1,2,3)\,, \tag{15}$$

and relative error functions, $\Delta_j^{(N)}$, are expressed as

$$\Delta_j^{(N)}(\tilde{x},t) = \sqrt{\tilde{x}}\,\tilde{y}_j^{(N)}(\tilde{x},t) - 1\,. \tag{16}$$

We point out that the coefficients of our algorithms are obtained without taking rounding errors into account. This issue will be shortly discussed at the end of Section 4.

2.1. Algorithm InvSqrt1

Assuming $a_1 = a_2 = 0$ and $b_1 = b_2 = 1$, we transform (12) into

$$\begin{aligned} \tilde{y}_1^{(1)} &= \frac{1}{2}\tilde{y}_0^{(1)}\left(3 - (\tilde{y}_0^{(1)})^2\,\tilde{x}\right) + \frac{1}{2}d_1\tilde{y}_1^{(1)}, \\ \tilde{y}_2^{(1)} &= \frac{1}{2}\tilde{y}_1^{(1)}\left(3 - (\tilde{y}_1^{(1)})^2\,\tilde{x}\right) + \frac{1}{2}d_2\tilde{y}_2^{(1)}. \end{aligned} \tag{17}$$

Therefore, $\tilde{y}_1^{(1)}$ and $\tilde{y}_2^{(1)}$ depend on $\tilde{x}, t, d_1$, and d_2. Parameters $t = t_1^{(1)}$ and $d_1 = d_1^{(1)}$ are determined by minimization of $||\Delta_1^{(1)}(\tilde{x}, t)||$. Then, the parameter $d_2 = d_2^{(1)}$ is determined by minimization of $||\Delta_2^{(1)}(\tilde{x}, t_1^{(1)})||$ (for details, see [32]). As a result, we get:

$$d_1^{(1)} \approx 1.75118 \cdot 10^{-3}\,, \qquad d_2^{(1)} \approx 1.15234 \cdot 10^{-6}\,, \tag{18}$$

and $t_1^{(1)} = t_1^{(0)}$ (see (10)). Therefore, $R^{(1)} = R^{(0)}$, i.e., *InvSqrt1* has the same magic constant as *InvSqrt*. The theoretical relative errors are given by

$$\begin{aligned} \Delta_{1\,\mathrm{max}}^{(1)} &\equiv ||\Delta_1^{(1)}(\tilde{x}, t_1^{(1)})|| \approx 0.87636 \cdot 10^{-3}, \\ \Delta_{2\,\mathrm{max}}^{(1)} &\equiv ||\Delta_2^{(1)}(\tilde{x}, t_1^{(1)})|| \approx 5.76 \cdot 10^{-7}. \end{aligned} \tag{19}$$

The algorithm (17) can be written in the form (13), where:

$$c_{1k}^{(1)} = \frac{3}{2 - d_k^{(1)}}\,, \qquad c_{2k}^{(1)} = \frac{1}{2 - d_k^{(1)}} \qquad (k = 1, 2)\,. \tag{20}$$

Taking into account numerical values for $d_1^{(1)}$ and $d_2^{(1)}$, we obtain the following values of the parameters $c_{jk}^{(1)}$:

$$\begin{aligned} c_{11}^{(1)} &\approx 1.5013145387528147176730252470373223, \\ c_{21}^{(1)} &\approx 0.50043817958427157255767508234577407, \\ c_{12}^{(1)} &\approx 1.5000008642589575005473878767725752, \\ c_{22}^{(1)} &\approx c_{21}^{(1)} \cdot 0.99912498383253616899527502360939620. \end{aligned} \tag{21}$$

This large number of digits, which is much higher than that needed for the single-precision computations, will be useful later in the case of higher precision.

Thus, finally, we obtained a new algorithm *InvSqrt1* that has the same structure as *InvSqrt*, but with different values of numerical coefficients (see [32]). In the case of two iterations, the code *InvSqrt*1 has more algebraic operations (one additional multiplication) in comparison to *InvSqrt*.

2.2. *InvSqrt2 Algorithm*

Assuming $a_1 = a_2 = 1$ and $b_1 = b_2 = 0$, we transform (12) into

$$\begin{aligned} \tilde{y}_1^{(2)} &= \frac{1}{2}\tilde{y}_0^{(2)}\left(3 - (\tilde{y}_0^{(2)})^2\,\tilde{x}\right) + \frac{1}{2}d_1\tilde{y}_0^{(2)}, \\ \tilde{y}_2^{(2)} &= \frac{1}{2}\tilde{y}_1^{(2)}\left(3 - (\tilde{y}_1^{(2)})^2\,\tilde{x}\right) + \frac{1}{2}d_2\tilde{y}_1^{(2)}, \end{aligned} \tag{22}$$

where $\tilde{y}_1^{(2)}$ and $\tilde{y}_2^{(2)}$ depend on $\tilde{x}, t, d_1$, and d_2.

Parameters $t = t_1^{(2)}$ and $d_1 = d_1^{(2)}$ are determined by the minimization of $||\Delta_1^{(2)}(\tilde{x}, t)||$. Then, the parameter $d_2 = d_2^{(2)}$ is determined by the minimization of $||\Delta_2^{(2)}(\tilde{x}, t_1^{(2)})||$ (see Appendix A.1 for details). As a result, we get:

$$d_1^{(2)} = 1.75791023259 \cdot 10^{-3}\,, \qquad d_2^{(2)} \simeq 1.159352515 \cdot 10^{-6}\,, \tag{23}$$

and

$$t_1^{(2)} \equiv t^{(2)} \approx 3.73157124016\,, \qquad R^{(2)} = 1597466888 = 0x5F376908\,. \tag{24}$$

The theoretical relative errors are given by

$$\begin{aligned} \Delta^{(2)}_{1\,\max} &\equiv ||\Delta^{(2)}_1(\tilde{x}, t^{(2)}_1)|| \approx 0.87908 \cdot 10^{-3}, \\ \Delta^{(2)}_{2\,\max} &\equiv ||\Delta^{(2)}_2(\tilde{x}, t^{(2)}_1)|| \approx 5.80 \cdot 10^{-7}. \end{aligned} \tag{25}$$

The coefficients in (13) are given by

$$c^{(2)}_{1k} = \frac{3 + d^{(2)}_k}{2}\,, \qquad c^{(2)}_{2k} = \frac{1}{2}\,. \tag{26}$$

Taking into account the numerical values for $d^{(2)}_1$ and $d^{(2)}_2$ (see (23)), we obtain the following values of the parameters $c^{(2)}_{jk}$:

$$\begin{aligned} c^{(2)}_{11} &\approx 1.5008789551163345746409291568502392, \\ c^{(2)}_{12} &\approx 1.5000005796762576644996810350809289, \\ c^{(2)}_{21} &= c^{(2)}_{22} = 0.5, \end{aligned} \tag{27}$$

where the large number of digits will be useful later in the case of higher precision. Thus, we completed the derivation of the code *InvSqrt*2:

```
1. float InvSqrt2(float x){
2.       float halfx = 0.5f*x;
3.       int i = *(int*) &x;
4.       i = 0x5F376908 - (i>>1);
5.       float y = *(float*) &i;
6.       y* = 1.50087896f - halfx*y*y;
7.       y* = 1.50000057f - halfx*y*y;
8.       return y;
9. }
```

The code *InvSqrt2* contains a new magic constant ($R^{(2)}$) and has two lines (6 and 7) that were modified in comparison with the code *InvSqrt*. We point out that *InvSqrt*2 has the same number of algebraic operations as *InvSqrt*.

2.3. *InvSqrt3 Algorithm*

Now, we consider the algorithm (13) in its most general form:

$$\begin{aligned} \tilde{y}^{(3)}_1 &= k_1 \tilde{y}^{(3)}_0 \left(k_2 - \tilde{x}(\tilde{y}^{(3)}_0)^2\right), \\ \tilde{y}^{(3)}_2 &= k_3 \tilde{y}^{(3)}_1 \left(k_4 - \tilde{x}(\tilde{y}^{(3)}_1)^2\right), \end{aligned} \tag{28}$$

where k_j, k_2, k_3, and k_4 are constant. In Appendix A.2, we determine parameters $t = t^{(3)}_1$, k_1, and k_2 by minimization of $||\Delta^{(3)}_1(\tilde{x}, t)||$. Then, the parameters k_3 and k_4 are determined by minimization of $||\Delta^{(3)}_2(\tilde{x}, t^{(3)}_1)||$. As a result, we get:

$$\begin{aligned} k_1 &\approx 0.70395201\,, \qquad & k_2 &\approx 2.3892451\,, \\ k_3 &\approx 0.50000005\,, \qquad & k_4 &\approx 3.0000004\,, \end{aligned} \tag{29}$$

and

$$t^{(3)}_1 \equiv t^{(3)} = 3, \qquad R^{(3)} = 1595932672 = 0x5F200000\,. \tag{30}$$

The theoretical relative errors are given by

$$\Delta_{1\,\mathrm{max}}^{(3)} \equiv ||\Delta_1^{(3)}(\tilde{x}, t_1^{(3)})|| \approx 0.65007 \cdot 10^{-3},$$
$$\Delta_{2\,\mathrm{max}}^{(3)} \equiv ||\Delta_2^{(3)}(\tilde{x}, t_1^{(3)})|| \approx 3.17 \cdot 10^{-7}. \tag{31}$$

They are significantly smaller (by 26% and 45%, respectively) than the analogous errors for *InvSqrt1* and *InvSqrt2* (see (19) and (25)). The comparison of error functions for *InvSqrt2* and *InvSqrt3* (in the case of one correction) is presented in Figure 1.

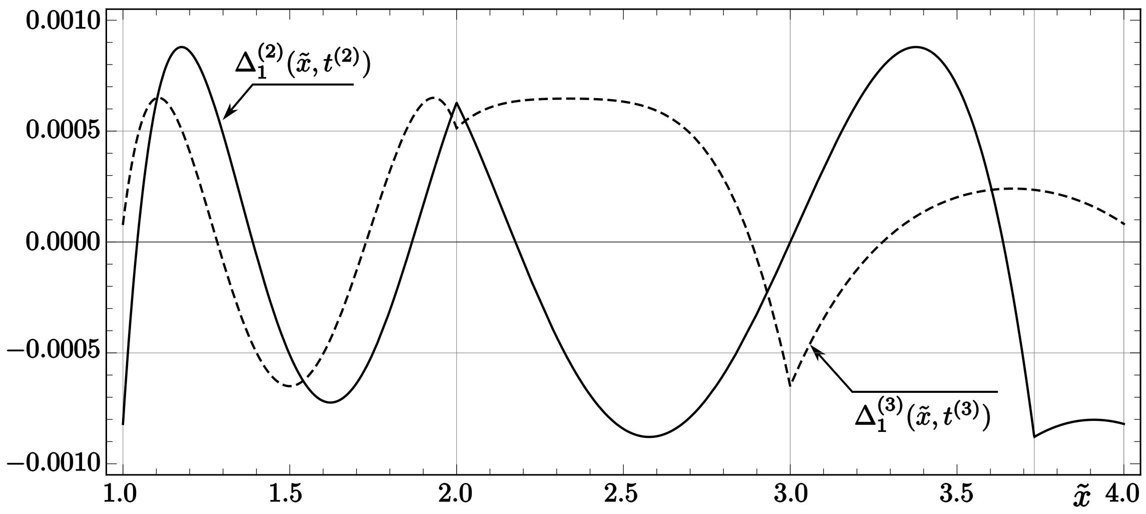

Figure 1. Theoretical relative errors of the first correction for the codes *InvSqrt2* and *InvSqrt3*. The solid line represents the function $\Delta_1^{(2)}(\tilde{x}, t^{(2)})$, while the dashed line represents $\Delta_1^{(3)}(\tilde{x}, t^{(3)})$.

The numerical values of coefficients $c_{ij}^{(3)}$ (compare with (13)) are given by:

$$\begin{aligned} c_{11}^{(3)} &= k_1 k_2 \approx 1.68191390868723079, \\ c_{21}^{(3)} &= k_1 \approx 0.703952009104829370, \\ c_{12}^{(3)} &= k_3 k_4 \approx 1.50000036976749938, \\ c_{22}^{(3)} &= k_3 \approx 0.500000052823927419. \end{aligned} \tag{32}$$

Thus, we obtained the following code, called *InvSqrt3*:

```
float InvSqrt3(float x){
    int i = *(int*) &x;
    i = 0x5F200000 - (i>>1);
    float y = *(float*) &i;
    y* = 1.68191391f - 0.703952009f*x*y*y;
    y* = 1.50000036f - 0.500000053f*x*y*y;
    return y;
}
```

The code *InvSqrt3* has the same number of multiplications as *InvSqrt1*, which means that it is slightly more expensive than *InvSqrt* and *InvSqrt2*.

3. Generalizations

The codes presented in Section 2 can only be applied to normal numbers (1) of the type **float**. In this section, we show how to extend these results to subnormal numbers and to higher-precision formats.

3.1. Subnormal Numbers

Subnormal numbers are smaller than any normal number of the form of (1). In the single-precision case, positive subnormals can be represented as $m_x \cdot 2^{-126}$, where $m_x \in (0,1)$. They can also be characterized by nine first bits equal to zero (which also includes the case where $x = 0$). In order to identify subnormals, we will make a bitwise conjunction (AND) of a given number with the integer $0x7f800000$, which has all eight exponent bits equal to 1 and all 23 mantissa bits equal to 0. This bitwise conjunction is zero if and only if the given number is subnormal (including 0).

In the case of the single precision, the multiplication by 2^{24} transforms any subnormal number into a normal number. Therefore, we make this transformation; then, we apply one of our algorithms and, finally, make the inverse transformation (i.e., multiplying the result by 2^{-12}). Thus, we get an approximated value of the inverse square root of the subnormal number. Note that 2^{24} is the smallest power of 2 with an even exponent that transforms all subnormals into normal numbers.

In the case of *InvSqrt3*, the procedure described above can be written in the form of the following code.

```
float InvSqrt3s(float x){
    int i = *(int*) &x;
    int k = i & 0x7f800000;
    if (k==0) {
        x = 16777216.f*x;    //16777216.f=pow(2.0f, 24)
        i = *(int*) &x;
    }
    i = 0x5F200000 - (i>>1);
    float y = *(float*) &i;
    y* = 1.68191391f - 0.703952009f*x*y*y;
    y* = 1.50000036f - 0.500000053f*x*y*y;
    if (k==0) return 4096.f*y;    //4096.f=pow(2.0f, 12)
    return y;
}
```

The maximum relative errors for this code are presented in Section 4 (see Table 1).

Table 1. Relative numerical errors for the first and second corrections in the case of the type **float** (compiler 32-bit) for subnormal numbers.

Algorithm	$\Delta^{(i)}_{1,N\,\mathrm{min}}$	$\Delta^{(i)}_{1,N\,\mathrm{max}}$	$\Delta^{(i)}_{2,N\,\mathrm{min}}$	$\Delta^{(i)}_{2,N\,\mathrm{max}}$
InvSqrt1	-0.87642×10^{-3}	0.87644×10^{-3}	-0.66221×10^{-6}	0.63442×10^{-6}
InvSqrt2	-0.87916×10^{-3}	0.87911×10^{-3}	-0.62060×10^{-6}	0.65285×10^{-6}
InvSqrt3	-0.65016×10^{-3}	0.65006×10^{-3}	-0.38701×10^{-6}	0.35196×10^{-6}

3.2. Higher Precision

The above analysis was confined to the single-precision floating-point format. This is sufficient for many applications (especially microcontrollers), although the double-precision format is more popular. A trade-off between accuracy, computational cost, and memory usage is welcome [33]. In this subsection, we extend our analysis to double- and higher-precision formats. The calculations are almost the same. We just have to compute all involved constants with an appropriate accuracy. Low-bit arithmetic cases could be treated in exactly the same way. In this paper, however, we are focused on increasing the accuracy and on possible applications in distributed systems, so only the cases of higher precision are explicitly presented.

We present detailed results for double precision and some results (magic constants) for quadruple precision. Performing computations in **C**, we use the GCC Quad-Precision

Math Library (working with numbers of type **_float128**). The crucial point is to express the magic constant R through the corresponding parameter t, which can be done with the formula:

$$R = N_m\,(3B-1)/2 + \lfloor N_m(t-2)/4 - \mu_{\tilde{x}}/2 \rceil \tag{33}$$

where $\mu_{\tilde{x}} \in \{0,1\}$, t depends on the considered algorithm and N_m and B depend on the precision format used. Namely,

$$\begin{aligned} &\text{Single precision (32-bit):} && N_m = 2^{23}, && B = 2^7 - 1\,, \\ &\text{Double precision (64-bit):} && N_m = 2^{52}, && B = 2^{10} - 1\,, \\ &\text{Quadruple precision (128-bit):} && N_m = 2^{112}, && B = 2^{14} - 1\,. \end{aligned} \tag{34}$$

In the case of the zeroth approximation (without Newton–Raphson corrections), the parameter t is given by:

$$t_0 = 3.7309795598377727818740863479840422, \tag{35}$$

which can be compared with [31]. The corresponding magic constants computed from the formula (33) read:

$$\begin{aligned} &\text{32-bit:} && R^{(0)} = 0x5F37642F \\ &\text{64-bit:} && R^{(0D)} = 0x5FE6EC85E7DE30DA \\ &\text{128-bit:} && R^{(0Q)} = 0x5FFE6EC85E7DE30DAABC602711840B0F. \end{aligned} \tag{36}$$

In this paper, we focus on the case of Newton–Raphson corrections, where the value of the parameter t may depend on the algorithm. For *InvSqrt* and *InvSqrt1*, we have:

$$t_1^{(0)} = t_1^{(1)} = 3.7298003391605705687151317499871860, \tag{37}$$

(see Section (2.1); compare with [31,32]). Then, (33) yields the following magic constants:

$$\begin{aligned} &\text{32-bit:} && R^{(1)} = 0x5F375A86 \\ &\text{64-bit:} && R^{(1D)} = 0x5FE6EB50C7B537A9 \\ &\text{128-bit:} && R^{(1Q)} = 0x5FFE6EB50C7B537A9CD9F02E504FCFC0. \end{aligned} \tag{38}$$

Actually, the above value of R in the 64-bit case (i.e., $R^{(1D)}$) corresponds to $\mu_{\tilde{x}} = 1$ (the same value of R was obtained by Robertson for *InvSqrt* [24] with a different method). For $\mu_{\tilde{x}} = 0$, we got an R greater by 1 (other results reported in this section do not depend on $\mu_{\tilde{x}}$). In the 128-bit case, Robertson obtained an R that was 1 less than our value (i.e., $R^{(1Q)}$).

In the case of *InvSqrt2*, we have

$$t^{(2)} = 3.7315712401613957182292407381942955 \tag{39}$$

(compare with (A11)), which yields:

$$\begin{aligned} &\text{32-bit:} && R^{(2)} = 0x5F376908 \\ &\text{64-bit:} && R^{(2D)} = 0x5FE6ED2102DCBFDA \\ &\text{128-bit:} && R^{(2Q)} = 0x5FFE6ED2102DCBFDA59415059AC483B5. \end{aligned} \tag{40}$$

Finally, for *InvSqrt3*, we obtained:

$$t^{(3)} = 3 \tag{41}$$

(see (A29)). The corresponding magic constants are given by:

$$\begin{aligned} &\text{32-bit:} \quad R^{(3)} = 0x5F200000 \\ &\text{64-bit:} \quad R^{(3D)} = 0x5FE400000000000C \\ &\text{128-bit:} \quad R^{(3Q)} = 0x5FFE4000000000000000000000000000. \end{aligned} \tag{42}$$

The parameters of the modified Newton–Raphson corrections for the higher-precision codes can be computed from the theoretical formulas used in the single-precision cases, taking into account an appropriate number of significant digits. In numerical experiments, we tested the algorithms *InvSqrt1D*, *InvSqrt2D*, and *InvSqrt3D* with the magic constants $R^{(1D)}$, $R^{(2D)}$, and $R^{(3D)}$, respectively, and the following coefficients in the modified Newton–Raphson iterations (compare with (21), (27), and (32), respectively):

$$\begin{aligned} c_{11}^{(1D)} &= 1.50131453875281472, \\ c_{21}^{(1D)} &= 0.500438179584271573, \\ c_{12}^{(1D)} &= 1.50000086425895750, \\ c_{22}^{(1D)} &= c_{21}^{(1D)} \cdot 0.999124983832536169. \end{aligned} \tag{43}$$

$$\begin{aligned} c_{11}^{(2D)} &= 1.50087895511633457, \\ c_{12}^{(2D)} &= 1.50000057967625766, \\ c_{21}^{(2D)} &= c_{22}^{(2D)} = 0.5, \end{aligned} \tag{44}$$

$$\begin{aligned} c_{11}^{(3D)} &= 1.68191390868723079, \\ c_{21}^{(3D)} &= 0.703952009104829370, \\ c_{12}^{(3D)} &= 1.50000036976749938, \\ c_{22}^{(3D)} &= 0.500000052823927419. \end{aligned} \tag{45}$$

The algorithm *InvSqrt* and its improved versions are usually implemented in the single-precision case with no more than two Newton–Raphson corrections. However, in the case of higher precision, higher accuracy of the result is welcome. Then, a higher number of modified Newton–Raphson iterations could be considered. As an example, we present the algorithm *InvSqrt2D* with four iterations:

```
1.  double InvSqrt2D(double x){
2.      double halfx=0.5*x;
3.      long long i=*(long long*) &x;
4.      i=0x5FE6ED2102DCBFDA - (i>>1);
5.      double y =*(double*) &i;
6.      y* = 1.50087895511633457 - halfx*y*y;
7.      y* = 1.50000057967625766 - halfx*y*y;
8.      y* = 1.5000000000002520 - halfx*y*y;
9.      y* = 1.5000000000000000 - halfx*y*y;
10.     return y;
11. }
```

By removing Line *9*, we obtain the code *InvSqrt2D* with three iterations, and by also removing Line *8*, we get the code defined by (44). The maximum relative errors for this code are presented in Section 4 (see (52)).

4. Numerical Experiments

The numerical tests for the codes derived and presented in this paper were performed on an Intel Core i5-3470 processor using the TDM-GCC 4.9.2 32-bit compiler (when repeating these tests on the Intel i7-5700 processor, we obtained the same results, and comparisons with some other processors and compilers are given in Appendix B). In this section, we discuss round-off errors for the algorithms *InvSqrt2* and *InvSqrt3* (the case of single precision and two Newton–Raphson iterations) and then present the final results of analogous analysis for other codes described in this paper.

Applying algorithms *InvSqrt2* and *InvSqrt3*, we obtain relative errors that differ slightly, due to round-off errors, from their analytical values (see Figures 2 and 3; compare with [32] for an analogous discussion concerning *InvSqrt1*). Although we present only randomly chosen values in the figures, calculations were done for all **float** numbers x such that $e_x \in [-126, 128)$.

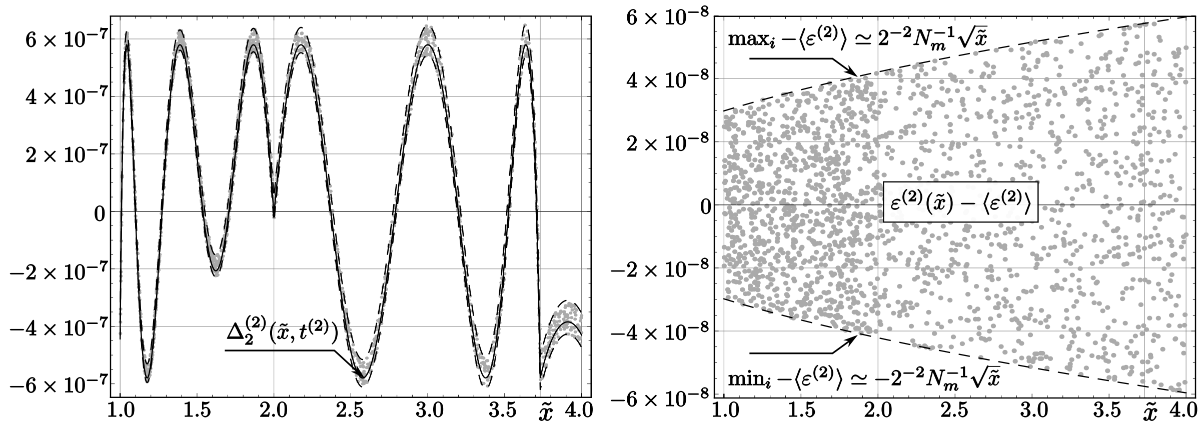

Figure 2. Theoretical and rounding errors of the code *InvSqrt2* (with two Newton–Raphson corrections). Left: The solid line represents $\Delta_2^{(2)}(\tilde{x}, t^{(2)})$, dashed lines correspond to $\Delta_2^{(2)}(\tilde{x}, t^{(2)}) \pm 2^{-2}N_m^{-1}\sqrt{\tilde{x}} + \langle\varepsilon^{(2)}\rangle$, and dots represent errors for 4000 floating-point numbers x randomly chosen from the interval $(2^{-126}, 2^{128})$. Right: relative error $\varepsilon^{(2)}$ (see (47)). Dashed lines correspond to the minimum and maximum values of these errors, and dots denote errors for 2000 values $\tilde{x}$ randomly chosen from the interval $[1, 4)$.

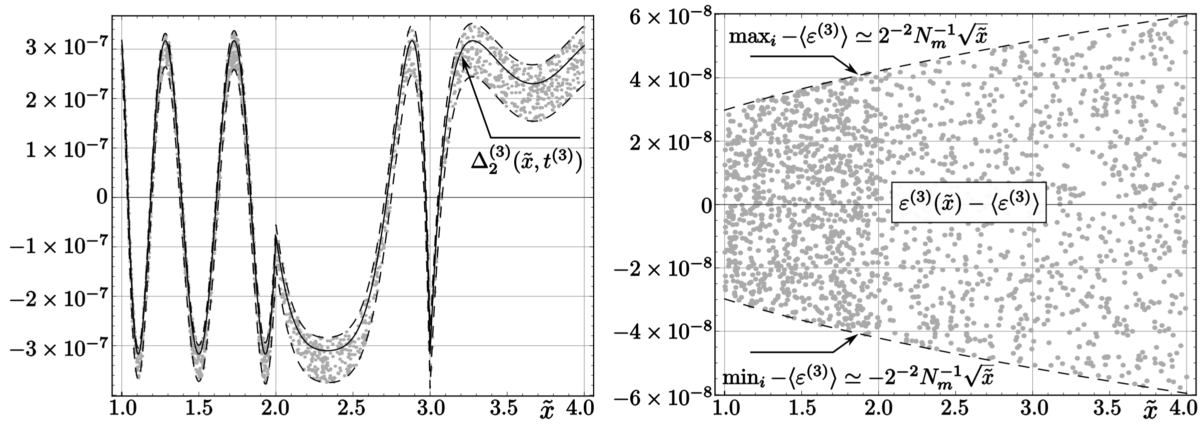

Figure 3. Theoretical and rounding errors of the code *InvSqrt3* (with two Newton–Raphson corrections). Left: The solid line represents $\Delta_2^{(3)}(\tilde{x}, t^{(3)})$, dashed lines correspond to $\Delta_2^{(3)}(\tilde{x}, t^{(3)}) \pm 2^{-2}N_m^{-1}\sqrt{\tilde{x}} + \langle\varepsilon^{(3)}\rangle$, and dots represent errors for 4000 floating-point numbers x randomly chosen from the interval $(2^{-126}, 2^{128})$. Right: relative error $\varepsilon^{(3)}$. Dashed lines correspond to minimum and maximum values of these errors, and dots denote errors for 2000 values $\tilde{x}$ randomly chosen from the interval $[1, 4)$.

The errors of numerical values returned by *InvSqrt2*

$$\Delta_{2;N}^{(2)}(x) = \mathrm{sqrt}(x) * \mathit{InvSqrt2}(x) - 1 \tag{46}$$

belong (for $e_x \neq -126$) to the interval $(-6.21 \cdot 10^{-7}, 6.53 \cdot 10^{-7})$. For $e_x = -126$, we get a wider interval: $[-6.46 \cdot 10^{-7}, 6.84 \cdot 10^{-7}]$. These errors differ from the errors of $\tilde{y}_2^{(3)}(\tilde{x}, t^{(2)})$, which were determined analytically (compare (25)). We define:

$$\varepsilon^{(2)}(\tilde{x}) = \frac{\mathit{InvSqrt2(x)} - \tilde{y}_2^{(2)}(\tilde{x}, t^{(2)})}{\tilde{y}_2^{(2)}(\tilde{x}, t^{(2)})}. \tag{47}$$

This function, representing the observed blur of the **float** approximation of the *InvSqrt2* output, is symmetric with respect to its mean value

$$\langle \varepsilon^{(2)} \rangle = 2^{-1} N_m^{-1} \sum_{x \in [1,4)} \varepsilon^{(2)}(\tilde{x}) = 1.636 \cdot 10^{-8} \tag{48}$$

(see the right part of Figure 2), and covers the following range of values:

$$\varepsilon^{(2)}(\tilde{x}) \in \left[-4.333 \cdot 10^{-8}, 7.596 \cdot 10^{-8}\right]. \tag{49}$$

Analogous results for the code *InvSqrt3* read:

$$\langle \varepsilon^{(3)} \rangle = 2^{-1} N_m^{-1} \sum_{x \in [1,4)} \varepsilon^{(3)}(\tilde{x}) = -1.890 \cdot 10^{-8} \tag{50}$$

$$\varepsilon^{(3)}(\tilde{x}) \in \left[-7.850 \cdot 10^{-8}, 4.067 \cdot 10^{-8}\right]. \tag{51}$$

The results produced by the same hardware with a 64-bit compiler have a greater amplitude of the error oscillations as compared with the 32-bit case (also compare Appendix B).

The maximum errors for the code *InvSqrt* and all codes presented in the previous sections are given in Table 2 (for codes with just one Newton–Raphson iteration) and Table 3 (the same codes but with two iterations).

Table 2. Relative numerical errors for the first correction in the case of type **float** (compiler 32-bit). In the case of type **double**, the errors are equal to theoretical errors $\pm\Delta_{1\,\max}^{(i)}$ up to the accuracy given in the table.

Algorithm	i	$\Delta_{1\,\max}^{(i)}$	$\Delta_{1,N\,\min}^{(i)}$	$\Delta_{1,N\,\max}^{(i)}$	$\Delta_{1,N}^{(i)}$
InvSqrt	0	1.75118×10^{-3}	-1.75124×10^{-3}	0.00008×10^{-3}	1.75124×10^{-3}
InvSqrt1	1	0.87636×10^{-3}	-0.87642×10^{-3}	0.87645×10^{-3}	0.87645×10^{-3}
InvSqrt2	2	0.87908×10^{-3}	-0.87916×10^{-3}	0.87914×10^{-3}	0.87916×10^{-3}
InvSqrt3	3	0.65007×10^{-3}	-0.65017×10^{-3}	$0.65006 \cdot 10^{-3}$	0.65017×10^{-3}

Looking at the last column of Table 2 (this is the case of one iteration), we see that the code *InvSqrt1* is slightly more accurate than *InvSqrt2*, and both are roughly almost two times more accurate than *InvSqrt*. However, it is the code *InvSqrt3* that has the best accuracy. The computational costs of all these codes are practically the same (four multiplications in every case).

Table 3. Relative numerical errors for the second correction in the case of type **float** (compiler 32-bit). In the case of type **double**, the errors are equal to theoretical errors $\pm\Delta^{(i)}_{2\,\max}$ up to the accuracy given in the table.

Algorithm	i	$\Delta^{(i)}_{2\,\max}$	$\Delta^{(i)}_{2,N\,\min}$	$\Delta^{(i)}_{2,N\,\max}$	$\Delta^{(i)}_{2,N}$
InvSqrt	0	4.59728×10^{-6}	-4.65441×10^{-6}	0.08336×10^{-6}	4.65441×10^{-6}
InvSqrt1	1	0.57617×10^{-6}	-0.67207×10^{-6}	0.64871×10^{-6}	0.67207×10^{-6}
InvSqrt2	2	0.57968×10^{-6}	-0.64591×10^{-6}	0.68363×10^{-6}	0.68363×10^{-6}
InvSqrt3	3	0.31694×10^{-6}	-0.38701×10^{-6}	0.35198×10^{-6}	0.38701×10^{-6}

In the case of two iterations (Table 3), the code *InvSqrt3* is the most accurate as well. Compared with *InvSqrt*, its accuracy is 12 times higher for single precision and 14.5 times higher for double precision. However, the computational costs of *InvSqrt1* and *InvSqrt3* (eight multiplications) are higher than the cost of *InvSqrt* (seven multiplications). Therefore, the code *InvSqrt2* has some advantage, as it is less accurate than *InvSqrt3* but cheaper. In the single-precision case the code *InvSqrt2* is 6.8 times more accurate than *InvSqrt*.

We point out that the round-off errors in the single-precision case significantly decrease the gain of the accuracy of the new algorithms as compared with the theoretical values, especially in the case of two Newton–Raphson corrections (compare the third and the last column of Table 3).

The range of errors in the case of subnormal numbers (using the codes described in Section 3.1) is shown in Table 1. One can easily see that the relative errors are similar—in fact, even slightly lower—than in the case of normal numbers.

Although the original *InvSqrt* code used only one Newton–Raphson iteration, and in this paper, we focus mostly on two iterations, it is worthwhile to also briefly consider the case of more iterations. Then, the increased computational cost is accompanied by increased accuracy. We confine ourselves to the code *InvSqrt2* (see the end of Section 3.2), which is less expensive than *InvSqrt3* (and the advantage of *InvSqrt2* increases with the number of iterations). In the double-precision case, the maximum error for three Newton–Raphson corrections is much lower, and the fourth correction yields the best possible accuracy.

$$\begin{aligned} \Delta^{(2)}_{1D,N} &= 0.87908 \times 10^{-3}\,, \\ \Delta^{(2)}_{2D,N} &= 0.57968 \times 10^{-6}\,, \\ \Delta^{(2)}_{3D,N} &= 2.5213 \times 10^{-13}\,, \\ \Delta^{(2)}_{4D,N} &= 1.1103 \times 10^{-16}\,. \end{aligned} \tag{52}$$

In the case of single precision, we already get the best possible accuracy for the third correction, given by adding the line `y* = 1.5f - halfx*y*y` as Line *8* in the code *InvSqrt2* (see Section 2.2).

$$\begin{aligned} \Delta^{(2)}_{1,N} &= 0.87916 \times 10^{-3}\,, \\ \Delta^{(2)}_{2,N} &= 0.68363 \times 10^{-6}\,, \\ \Delta^{(2)}_{3,N} &= 0.89367 \times 10^{-7} \end{aligned} \tag{53}$$

The derivation of all numerical codes presented in this paper did not take rounding errors into account. Therefore, the best floating-point parameters can be slightly different from the rounding of the best real parameters, all the more so since the distribution of the errors is still not exactly symmetric (compare fourth and fifth columns in Tables 2 and 3). The full analysis of this problem is much more difficult than the analogous analysis for the original *InvSqrt* code because we now have several parameters to be optimized instead of a single magic constant. At the same time, the increase in accuracy is negligible. Actually, much greater differences in the accuracy appear in numerical experiments as a result of using different devices (see Appendix B).

As an example, we present the results of an experimental search in the case of the code *InvSqrt3* with one Newton–Raphson correction (three parameters to be optimized). The modified Newton–Raphson coefficients are found to be

$$c_{11\ num}^{(3)} = 1.681911588f\ , \qquad c_{21\ num}^{(3)} = k_1 = 0.7039490938f\ . \tag{54}$$

Figure 4 summarizes the last step of this analysis. The dependence of maximum errors on R shows clearly that the optimum value for the magic constant is slightly shifted as compared to the theoretical (real) value:

$$R_{num}^{(3)} = 17 + 0x5f200000 = 0x5f200011\ . \tag{55}$$

The corresponding errors given by

$$\Delta_{1,N\,\max}^{(3)} = 6.50112284 \cdot 10^{-4}\ , \qquad \Delta_{1,N\,\min}^{(3)} = -6.501092575 \cdot 10^{-4} \tag{56}$$

are nearly symmetric. They are smaller than the maximum error $\Delta_{1,N}^{(3)}$ corresponding to our theoretical values, but only by about 0.001% (see Table 2).

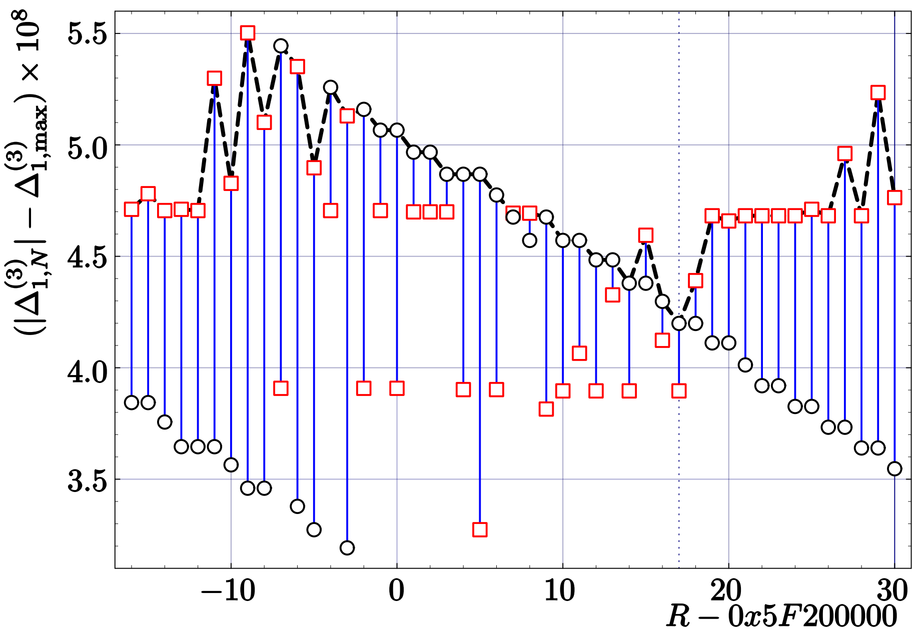

Figure 4. Maximum relative errors for the first Newton–Raphson correction in the code *InvSqrt3* as a function of R in the case of $k_1 = 0.7039490938f$ and $k_1k_2 = 1.681911588f$. Circles denote maximum errors ($\Delta_{1,N\,\max}^{(3)}$), while squares denote minimum errors ($|\Delta_{1,N\,\min}^{(3)}|$). The maximum error (shown by the dashed line) was determined by minimizing the maximum error for all floating-point numbers from $[1,4)$.

5. Conclusions

We presented two new modifications (*InvSqrt2* and *InvSqrt3*) of the fast inverse square root code in single-, double-, and higher-precision versions. Each code has its own magic constant. All new algorithms are much more accurate than the original code *InvSqrt*. One of the new algorithms, *InvSqrt2*, has the same computational cost as *InvSqrt* in the case of any precision. Another code, *InvSqrt3*, has the best accuracy, but is more expensive if the number of Newton–Raphson corrections is greater than 1. However, its gain in accuracy is very high, even by more than 12 times for two iterations (see Table 3 in Section 4).

Our approach was to modify the Newton–Raphson method by introducing arbitrary parameters, which are then determined by minimizing the maximum relative error. It is expected that such modifications will provide a significant increase in accuracy, especially in the case of asymmetric error distribution for Newton–Raphson corrections (and this is the case with the inverse square root function when these corrections are non-positive). One has to remember that due to rounding errors, our theoretical results may differ from the best floating-point parameters, but the difference is negligible (see the end of Section 4). In fact, parameters (magic constants and modified Newton–Raphson coefficients) from a certain range near the values obtained in this article seem equally good for all practical purposes.

Concerning potential applications, we have to acknowledge that for general-purpose computing, the SSE and AVX reciprocal square root instructions are faster and more accurate. We hope, however, that the proposed algorithms can be applied in embedded systems and microcontrollers without a hardware floating-point divider, and potentially in FPGAs. Moreover, in contrast to the SSE and AVX instructions, our approach can be easily extended to computational platforms of high precision, like 256-bit or 512-bit platforms.

Author Contributions: Conceptualization, L.V.M.; Formal analysis, C.J.W.; Investigation, C.J.W., L.V.M., and J.L.C.; Methodology, C.J.W. and L.V.M.; Visualization, C.J.W.; Writing—original draft, J.L.C.; Writing—review and editing, 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 conflict of interest.

Appendix A. Analytical Derivation of Modified Newton–Raphson Coefficients

Appendix A.1. Algorithm InvSqrt2

We will determine the parameters d_1 and d_2 in formulas (22) that minimize the maximum error. Substituting (16) (with $n = 2$) into (22), we get:

$$\Delta_1^{(2)}(\tilde{x}, t, d_1) = \frac{1}{2}d_1\big(1 + \tilde{\delta}_0(\tilde{x}, t)\big) - \frac{1}{2}\tilde{\delta}_0^2(\tilde{x}, t)\big(3 + \tilde{\delta}_0(\tilde{x}, t)\big) \tag{A1}$$

$$\Delta_2^{(2)}(\tilde{x}, t, d_1, d_2) = \frac{1}{2}d_2\Big(1 + \Delta_1^{(2)}\Big) - \frac{1}{2}\Big(\Delta_1^{(2)}\Big)^2\Big(3 + \Delta_1^{(2)}\Big), \tag{A2}$$

where $\Delta_1^{(2)} \equiv \Delta_1^{(2)}(\tilde{x}, t, d_1)$ and $\tilde{\delta}_0(\tilde{x}, t)$ is the relative error of the zeroth approximation (the function $\tilde{\delta}_0(\tilde{x}, t)$ is presented and discussed in [31,32]).

First, we are going to determine the t and $d_1^{(2)}$ that minimize the maximum absolute value of the relative error of the first correction. We have to solve the following equation:

$$0 = \frac{\partial \Delta_1^{(2)}(\tilde{x}, t)}{\partial \tilde{\delta}_0(\tilde{x}, t)} = \frac{1}{2}d_1^{(2)} - 3\tilde{\delta}_0(\tilde{x}, t) - \frac{3}{2}\tilde{\delta}_0^2(\tilde{x}, t). \tag{A3}$$

Its solution

$$\tilde{\delta}^+ = \sqrt{1 + d_1^{(2)}/3} - 1 \tag{A4}$$

corresponds to the value

$$\Delta_{1\,\max}^{(2)} = \frac{1}{2}d_1^{(2)}(1 + \tilde{\delta}^+) - \frac{1}{2}\tilde{\delta}^{+2}(3 + \tilde{\delta}^+), \tag{A5}$$

which is a maximum of $\Delta_1^{(2)}(\tilde{x},t)$ because its second derivative with respect to $\tilde{x}$, i.e.,

$$\partial^2_{\tilde{x}}\Delta_1^{(2)}(\tilde{x},t) = \partial^2_{\tilde{x}}\tilde{\delta}_0(\tilde{x},t)\partial_{\tilde{\delta}_0}\Delta_1^{(2)}(\tilde{x},t) - 3(1+\tilde{\delta}_0(\tilde{x},t))(\partial_x\tilde{\delta}_0(\tilde{x},t))^2, \tag{A6}$$

is negative. In order to determine the dependence of $d_1^{(2)}$ on the parameter t, we solve the equation

$$-\Delta_1^{(2)}(t,t) = \Delta_{1\,\max}^{(2)}, \tag{A7}$$

which (for some $t = t_1^{(2)}$) equates the maximum value of error with the modulus of the minimum value of error. Thus, we obtain the following relations:

$$\delta^+ = -1 - \frac{1}{4}\sqrt{t} + \frac{1}{8}\frac{t}{f(t)} + \frac{1}{2}f(t), \tag{A8}$$

$$d_1^{(2)} = -3 + \frac{9}{16}t + \frac{3}{64}t^2 f^{-2}(t) - \frac{3}{16}t^{3/2}f^{-1}(t) - \frac{3}{4}\sqrt{t}f(t) + \frac{3}{4}f^2(t), \tag{A9}$$

where

$$f(t) = \left[8 + t^{3/2}/8 + 4\sqrt{4 + t^{3/2}/8}\right]^{1/3}.$$

The last step consists in equating the minimum boundary value of the error of analyzed correction with its smallest local minimum:

$$\Delta_1^{(2)}(t,t) = \frac{1}{2}d_1^{(2)}(1+\tilde{\delta}_0(\tilde{x}_0^{II},t)) - \frac{1}{2}\tilde{\delta}_0^2(\tilde{x}_0^{II},t)(3+\tilde{\delta}_0(\tilde{x}_0^{II},t)), \tag{A10}$$

where $x_0^{II} = (4+t)/3$ (see [31]). Solving the Equation (A10) numerically, we obtain the following value of t:

$$t_1^{(2)} \simeq 3.73157124016, \tag{A11}$$

which corresponds to the following magic constant:

$$R^{(2)} = 1597466888 = 0x5F376908\,. \tag{A12}$$

Taking into account (A8), we compute

$$d_1^{(2)} = 1.75791023259 \cdot 10^{-3}, \tag{A13}$$

and, using (A4) and (A5), we get

$$\Delta_{1\,\max}^{(2)} \simeq 8.7908386407 \cdot 10^{-4} \simeq \frac{\tilde{\delta}_{1\,\max}}{1.99}. \tag{A14}$$

In the case of the second correction, we keep the obtained value $t = t_1^{(2)}$ and determine the parameter $d_2^{(2)}$, which equates the maximum value of the error with the modulus of its global minimum. $\Delta_2^{(2)}(\tilde{x},t_1^{(2)})$ is increasing (decreasing) with respect to negative (positive) $\Delta_1^{(2)}(\tilde{x},t_1^{(2)})$ and has local minima that come only from positive maxima and negative minima. Therefore, the global minimum should correspond to the global minimum $-\Delta_{1\,\max}^{(2)}$ or to the global maximum $\Delta_{1\,\max}^{(2)}$. Substituting these values into Equation (A2) in the place of $\Delta_1^{(2)}(\tilde{x},t_1^{(2)})$, we obtain that deeper minima of $(-\Delta_{2\,\max}^{(2)})$ come from the global minimum of the first correction:

$$-\Delta_{2\,\max}^{(2)} = \frac{1}{2}d_2^{(2)}(1-\Delta_{1\,\max}^{(2)}) - \frac{1}{2}\Delta_{1\,\max}^{(2)2}(3-\Delta_{1\,\max}^{(2)}), \tag{A15}$$

and the maximum, by analogy to the first correction, corresponds to the following value of $\Delta_1^{(2)}(\tilde{x}, t_1^{(2)})$:

$$\Delta^+ = \sqrt{1 + d_2^{(2)}/3} - 1. \tag{A16}$$

Solving the equation

$$\Delta_{2\,\max}^{(2)} = \frac{1}{2} d_2^{(2)}(1 + \Delta^+) - \frac{1}{2}\Delta^{+2}(3 + \Delta^+), \tag{A17}$$

we get

$$d_2^{(2)} \simeq 1.159352515 \cdot 10^{-6} \quad \text{and} \quad \Delta_{2\,\max}^{(2)} \simeq 5.796763137 \cdot 10^{-7} \simeq \frac{\tilde{\delta}_{2\,\max}}{7.93}. \tag{A18}$$

Appendix A.2. Algorithm InvSqrt3

Parameters k_1, k_2, k_3, and k_4 in the formula (28) will be determined by minimization of the maximum error. The relative error functions for (28) are given by:

$$\Delta_j^{(3)} = \sqrt{\tilde{x}}\,\tilde{y}_j^{(3)} - 1, \tag{A19}$$

where $j = 1, 2$. Substituting (A19) into (28), we obtain:

$$\begin{aligned} \Delta_1^{(3)}(\tilde{x}, t, k_1, k_2) &= k_1 k_2(\tilde{\delta}_0(\tilde{x}, t) + 1) - k_1(\tilde{\delta}_0(\tilde{x}, t) + 1)^3 - 1\,, \\ \Delta_2^{(3)}(\tilde{x}, t, k) &= k_3 k_4(\Delta_1^{(3)}(\tilde{x}, t, k_1, k_2) + 1) - k_3(\Delta_1^{(3)}(\tilde{x}, t, k_1, k_2) + 1)^3 - 1\,, \end{aligned} \tag{A20}$$

where $k = (k_1, k_2, k_3, k_4)$ and $\tilde{\delta}_0(\tilde{x}, t)$ is the relative error of the zeroth approximation (see [31,32]).

We are going to find parameters t and k such that the error functions take extreme values. We begin with $\Delta_1^{(3)}$.

$$\partial_{\tilde{x}}\Delta_1^{(3)}(\tilde{x}, t, k_1, k_2) = \Big(k_1 k_2 - 3k_1(\tilde{\delta}_0(\tilde{x}, t) + 1)^2\Big)\partial_{\tilde{x}}\tilde{\delta}_0(\tilde{x}, t). \tag{A21}$$

Therefore, the local extremes of $\Delta_1^{(3)}$ can be located either at the same points as the extremes of $\tilde{\delta}_0(\tilde{x}, t)$ or at the $\tilde{x}$ satisfying the equation:

$$\tilde{\delta}_0(\tilde{x}, t) = \delta_1^{(e)} \tag{A22}$$

where

$$\delta_1^{(e)} = \sqrt{k_2/3} - 1\,. \tag{A23}$$

The extremes corresponding to $\delta_1^{(e)}$ are global maxima equal to

$$\Delta_{1\,\max}^{(3)} = k_1 k_2(\delta_1^{(e)} + 1) - k_1(\delta_1^{(e)} + 1)^3 - 1 = 2k_1\left(\frac{k_2}{3}\right)^{3/2} - 1\,. \tag{A24}$$

The two extremes of $\tilde{\delta}_0(\tilde{x}, t)$, located at $\tilde{x}_0^I = (6 + t)/6$ and $\tilde{x}_0^{II} = (4 + t)/3$ (see [31]), can correspond either to minima or to maxima of $\Delta_1^{(3)}$ (depending on parameters t, k_1, and k_2). If $\tilde{\delta}_0(\tilde{x}_0^{I/II}, t) < \delta_1^{(e)}$, we have a maximum. The case $\tilde{\delta}_0(\tilde{x}_0^{I/II}, t) > \delta_1^{(e)}$ corresponds to a minimum.

Global extremes of $\Delta_1^{(3)}$ can be either local extremes or boundary values (which, in turn, correspond to global extremes of $\tilde{\delta}_0(\tilde{x}, t)$). We recall that the global minimum of $\tilde{\delta}_0(\tilde{x}, t)$ is at $\tilde{x} = t$ and the global maximum is at $\tilde{x}_0^I$ or $\tilde{x}_0^{II}$ [31]. Therefore, in order to find the parameters k_1 and k_2 that minimize the maximal value of $|\Delta_1^{(3)}|$, we have to solve two equations:

$$\Delta_{1\,\max}^{(3)} + \Delta_1^{(3)}(t,t,k_1,k_2) = 0, \tag{A25}$$

$$\Delta_1^{(3)}(\tilde{x}_0^N,t,k_1,k_2) = \Delta_1^{(3)}(t,t,k_1,k_2), \tag{A26}$$

where $N = I$ for $t \in (2, 2^{5/3} + 2^{4/3} - 2)$ and $N = II$ for $t \in (2^{5/3} + 2^{4/3} - 2, 4)$. Note that $\tilde{\delta}(\tilde{x}_0^N, t) = \max\{\tilde{\delta}(\tilde{x}_0^I, t), \tilde{\delta}(\tilde{x}_0^{II}, t)\}$ and $\Delta_1^{(3)}(\tilde{x}_0^N, t, k_1, k_2)$ is a minimum of $\Delta_1^{(3)}$.

The solution of the system (A25) and (A26) corresponds to the case $N = I$ and is given by:

$$\begin{cases} k_1 = 2\left(2\cdot 3^{-3/2}k_2^{3/2} + t^{1/2}k_2/2 - t^{3/2}/8\right)^{-1} \\ k_2 = \left[(1+t/6)^3 + \sqrt{t}(1+t/6)^{3/2} + t\right]/4\,. \end{cases} \tag{A27}$$

Thus, $\Delta_{1\,\max}^{(3)}$, given by (A24), is a function of one variable t, and we can easily find its minimum value. It is enough to compute

$$\frac{d\,\Delta_{1\,\max}^{(3)}}{dt} \equiv \sqrt{\frac{k_2}{3}}\left(\frac{2}{3}k_2\frac{d\,k_1}{dt} + k_1\frac{d\,k_2}{dt}\right) = 0\,. \tag{A28}$$

We obtain $t = t_1^{(3)}$, where

$$t_1^{(3)} = 3, \qquad R^{(3)} = 1595932672 = 0x5F200000\,, \tag{A29}$$

and, inserting $t = t_1^{(3)}$ into (A24) and (A27):

$$\Delta_{1\,\max}^{(3)} = \frac{-54\sqrt{3} - 36\sqrt{6} + \sqrt{2}(17+6\sqrt{2})^{3/2}}{54\sqrt{3} + 36\sqrt{6} + \sqrt{2}(17+6\sqrt{2})^{3/2}} \simeq 6.5007\cdot 10^{-4}, \tag{A30}$$

$$\begin{aligned} k_1 &= \frac{256}{54\sqrt{3} + 36\sqrt{6} + 12\sqrt{17+6\sqrt{2}} + 17\sqrt{34+12\sqrt{2}}} \approx 0.7039520, \\ k_2 &= \frac{51+18\sqrt{2}}{32} \approx 2.3892451. \end{aligned} \tag{A31}$$

In order to find the parameters k_3 and k_4 that minimize the maximal relative error of the second correction, we fix $t = t_1^{(3)}$. Then, we have to solve the following Equations (obtained in the same way as Equations (A25) and (A26)):

$$\begin{cases} \Delta_2^{(3)}(t,t,k_1,k_2,k_3,k_4) = & k_3k_4(\Delta_{1\,\max}^{(3)}+1) - k_3(\Delta_{1\,\max}^{(3)}+1)^3 - 1\,, \\ \Delta_2^{(3)}(t,t,k_1,k_2,k_3,k_4) = & -k_3k_4(\delta_2^{(e)}+1) + k_3(\delta_2^{(e)}+1)^3 + 1\,, \end{cases} \tag{A32}$$

where

$$\delta_2^{(e)} = \sqrt{k_4/3} - 1 \tag{A33}$$

corresponds to $\Delta_1^{(3)}(\tilde{x},t,k_1,k_2)$ satisfying:

$$0 = \partial_{\Delta_1^{(3)}(\tilde{x},t,k_1,k_2)}\Delta_2^{(3)}(\tilde{x},t,k_1,k_2,k_3,k_4) = k_3k_4 - 3k_3(\Delta_1^{(3)}(\tilde{x},t,k_1,k_2)+1)^2.$$

Solving the system (A32), we get:

$$k_4 = 3 + (\Delta^{(3)}_{1\,\max})^2\,,$$
$$k_3 = 2\Big(\Big(2 - \Delta^{(3)}_{1\,\max} + \delta^{(e)}_2\Big)\Big(2 - \delta^{(e)}_2(1 + \Delta^{(3)}_{1\,\max} + \delta^{(e)}_2) + \Delta^{(3)}_{1\,\max}\Big)\Big)^{-1}\,, \tag{A34}$$
$$\Delta^{(3)}_{2\,\max} = k_3 k_4(\delta^{(e)}_2 + 1) - k_3(\delta^{(e)}_2 + 1)^3 - 1\,.$$

Then, using (A30) and (A33), we get numerical values:

$$k_3 \approx 0.50000005\,, \quad k_4 \approx 3.0000004\,, \quad \Delta^{(3)}_{2\,\max} \approx 3.16943579 \cdot 10^{-7}\,. \tag{A35}$$

One can easily see that the obtained error $\Delta^{(3)}_{2\,\max}$ is only about half (55%) of the error $\Delta^{(2)}_{2\,\max}$.

Appendix B. Numerical Experiments Using Different Processors and Compilers

The accuracy of our codes depends, to some extent, on the devices used for testing. In Section 4, we limited ourselves to the Intel Core i5 with the 32-bit compiler. In this Appendix, we present, for comparison, data from other devices (Tables A1 and A2). All data are for the type **float** (single precision).

The first two columns with data correspond to the Intel Core i5 with the 32-bit compiler (described, in more detail, in Section 4), the next two columns correspond to the same processor, but with the 64-bit compiler. Then, we have results (the same) for three microcontrollers: STM32L432KC and TM4C123GH6PM (ARM Cortex-M4), as well as STM32F767ZIT6 (ARM Cortex-M7). The last two columns contain results for the ESP32-D0WDQ5 system with two Xtensa LX6 microprocessors.

Table A1. The range of numerical errors for the first correction depending on the CPU used.

Code	**Intel Core i5 (32-bit)**		**Intel Core i5 (64-bit)**		**ARM Cortex M4-7**		**ESP32**	
	$10^3\Delta_{\min}$	$10^3\Delta_{\max}$	$10^3\Delta_{\min}$	$10^3\Delta_{\max}$	$10^3\Delta_{\min}$	$10^3\Delta_{\max}$	$10^3\Delta_{\min}$	$10^3\Delta_{\max}$
InvSqrt1	−0.87642	0.87644	−0.87646	0.87654	−0.87649	0.87656	−0.87646	0.87652
InvSqrt2	−0.87916	0.87911	−0.87922	0.87924	−0.87923	0.87927	−0.87921	0.87922
InvSqrt3	−0.65017	0.65006	−0.65029	0.65017	−0.65028	0.65018	−0.65025	0.65014

Table A2. The range of numerical errors for the second correction depending on the processor used.

Code	**Intel Core i5 (32-bit)**		**Intel Core i5 (64-bit)**		**ARM Cortex M4-7**		**ESP32**	
	$10^6\Delta_{\min}$	$10^6\Delta_{\max}$	$10^6\Delta_{\min}$	$10^6\Delta_{\max}$	$10^6\Delta_{\min}$	$10^6\Delta_{\max}$	$10^6\Delta_{\min}$	$10^6\Delta_{\max}$
InvSqrt1	−0.66221	0.63504	−0.75813	0.78832	−0.80341	0.81933	−0.75548	0.77074
InvSqrt2	−0.62060	0.65287	−0.70266	0.77609	−0.74198	0.81605	−0.69991	0.76667
InvSqrt3	−0.38701	0.35198	−0.48605	0.45363	−0.51755	0.48057	−0.47314	0.44122

References

1. Ercegovac, M.D.; Lang, T. *Digital Arithmetic*; Morgan Kaufmann: San Francisco, CA, USA, 2003.
2. Parhami, B. *Computer Arithmetic: Algorithms and Hardware Designs*, 2nd ed.; Oxford University Press: New York, NY, USA, 2010.
3. Muller, J.-M.; Brunie, N.; Dinechin, F.; Jeannerod, C.-P.; Joldes, M.; Lefèvre, V.; Melquiond, G.; Revol, N.; Torres, S. *Handbook of Floating-Point Arithmetic*, 2nd ed.; Birkhäuser: Basel, Switzerland, 2018.
4. Eberly, D.H. *GPGPU Programming for Games and Science*; CRC Press: Boca Raton, FL, USA, 2015.
5. Beebe, N.H.F. *The Mathematical-Function Computation Handbook: Programming Using the MathCW Portable Software Library*; Springer: Berlin/Heidelberg, Germany, 2017.

6. Blanchard, P.; Higham, N.J.; Mary, T. A class of fast and accurate summation algorithms. *SIAM J. Sci. Comput.* **2020**, *42*, A1541–A1557. [CrossRef]
7. Moroz, L.; Samotyy, W. Efficient floating-point division for digital signal processing application. *IEEE Signal Process. Mag.* **2019**, *36*, 159–163. [CrossRef]
8. Liu, W.; Nannarelli, A. Power Efficient Division and Square root Unit. *IEEE Trans. Comp.* **2012**, *61*, 1059–1070. [CrossRef]
9. Viitanen, T.; Jääskeläinen, P.; Esko, O.; Takala, J. Simplified floating-point division and square root. In Proceedings of the IEEE International Conference Acoustics Speech and Signal Process,Vancouver, BC, Canada, 26–31 May 2013; pp. 2707–2711.
10. Cornea, M. *Intel® AVX-512 Instructions and Their Use in the Implementation of Math Functions*; Intel Corporation: Santa Clara, CA, USA, 2015.
11. Ivanchenko, V.N.; Apostolakis, J.; Bagulya, A.; Bogdanov, A.; Grichine, V.; Incerti, S.; Ivantchenko, A.; Maire, M.; Pandola, L.; Pokorski, W.; et al. Geant4 Electromagnetic Physics for LHC Upgrade. *J. Phys. Conf. Seriies* **2014**, 513, 022015. [CrossRef]
12. Piparo, D.; Innocente, V.; Hauth, T. Speeding up HEP experiment software with a library of fast and auto-vectorisable mathematical functions. *J. Phys. Conf. Seriies* **2014**, *513*, 052027. [CrossRef]
13. Faerber, C. Acceleration of Cherenkov angle reconstruction with the new Intel Xeon/FPGA compute platform for the particle identification in the LHCb Upgrade. *J. Phys. Conf. Seriies* **2017**, *898*, 032044. [CrossRef]
14. Ercegovac, M.D.; Lang, T. *Division and Square Root: Digit Recurrence Algorithms and Implementations*; Kluwer Academic Publishers: Boston, MA, USA, 1994.
15. Kwon, T.J.; Draper, J. Floating-point Division and Square root Implementation using a Taylor-Series Expansion Algorithm with Reduced Look-Up Table, In Proceedings of the 51st Midwest Symposium on Circuits and Systems, Knoxville, TN, USA, 10–13 August 2008.
16. Aguilera-Galicia, C.R. Design and Implementation of Reciprocal Square Root Units on Digital ASIC Technology for Low Power Embedded Applications. Ph.D. Thesis, ITESO, Tlaquepaque, Jalisco, Mexico, 2019.
17. Lemaitre, F. Tracking Haute Frequence Pour Architectures SIMD: Optimization de la Reconstruction LHCb. Ph.D. Thesis, Paris University IV (Sorbonne), Paris, France, 2019; (CERN-THESIS-2019-014).
18. Oberman, S.; Favor, G.; Weber, F. AMD 3DNow! technology: Architecture and implementations. *IEEE Micro* **1999**, *19*, 37–48. [CrossRef]
19. id software, quake3-1.32b/code/game/q_math.c, Quake III Arena, 1999. Available online: https://github.com/id-Software/Quake-III-Arena/blob/master/code/game/q_math.c (accessed on 8 January 2021).
20. Sarma, D.D.; Matula, D.W. Faithful bipartite ROM reciprocal tables. In Proceedings of the 12th IEEE Symposium on Computer Arithmetic, Bath, UK, 19–21 July 1995; pp. 17–29.
21. Stine, J.E.; Schulte, M.J. The symmetric table addition method for accurate function approximation. *J. VLSI Signal Process.* **1999**, *11*, 1–12.
22. Blinn, J. Floating-point tricks. *IEEE Comput. Graph. Appl.* **1997**, *17*, 80–84. [CrossRef]
23. Lomont, C. Fast Inverse Square Root, Purdue University, Technical Report, 2003. Available online: http://www.lomont.org/papers/2003/InvSqrt.pdf (accessed on 8 January 2021).
24. Robertson, M. A Brief History of InvSqrt. Bachelor's Thesis, University of New Brunswick, Fredericton, NB, Canada, 2012.
25. Warren, H.S. *Hacker's Delight*, 2nd ed.; Pearson Education: Upper Saddle River, NJ, USA, 2013.
26. Hänninen, T.; Janhunen, J.; Juntti, M. Novel detector implementations for 3G LTE downlink and uplink. *Analog. Integr. Circ. Sig. Process.* **2014**, *78*, 645–655. [CrossRef]
27. Hsu, C.J.; Chen, J.L.; Chen, L.G. An Efficient Hardware Implementation of HON4D Feature Extraction for Real-time Action Recognition. In Proceedings of the 2015 IEEE International Symposium on Consumer Electronics (ISCE), Madrid, Spain, 17 June 2015.
28. Hsieh, C.H.; Chiu, Y.F.; Shen, Y.H.; Chu, T.S.; Huang, Y.H. A UWB Radar Signal Processing Platform for Real-Time Human Respiratory Feature Extraction Based on Four-Segment Linear Waveform Model. *IEEE Trans. Biomed. Circ. Syst.* **2016**, *10*, 219–230. [CrossRef] [PubMed]
29. Sangeetha, D.; Deepa, P. Efficient Scale Invariant Human Detection using Histogram of Oriented Gradients for IoT Services. In *2017 30th International Conference on VLSI Design and 2017 16th International Conference on Embedded Systems*; IEEE: Piscataway, NJ, USA, 2016; pp. 61–66 .
30. Hasnat, A.; Bhattacharyya, T.; Dey, A.; Halder, S.; Bhattacharjee, D. A fast FPGA based architecture for computation of square root and Inverse Square Root. In Proceedings of the 2nd International Conference on Devices for Integrated Circuit, DevIC 2017, Kalyani, Nadia, India, 23–24 March 2017; pp. 383–387.
31. 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]
32. Walczyk, C.J.; Moroz, L.V.; Cieśliński, J.L. A Modification of the Fast Inverse Square Root Algorithm. *Computation* **2019**, *7*, 41. [CrossRef]
33. Graillat, S.; Jézéquel, F.; Picot, R.; Févotte, F.; Lathuilière, B. Auto-tuning for floating-point precision with Discrete Stochastic Arithmetic. *J. Comput. Sci.* **2019**, *36*, 101017. [CrossRef]