Elementary Functions and Approximate Computing

Jean-Michel Muller

Elementary Functions and Approximate Computing

Jean-Michel Muller* * Univ Lyon, CNRS, ENS de Lyon, Inria, Université Claude Bernard Lyon 1, LIP UMR 5668, F-69007 Lyon, France

***Abstract*—We review some of the classical methods used for quickly obtaining low-precision approximations to the elementary functions. Then, for each of the three main classes of elementary function algorithms (shift-and-add algorithms, polynomial or rational approximations, table-based methods, bit-manipulation techniques), we examine what can be done for obtaining very fast estimates of a function, at the cost of a (controlled) loss in terms of accuracy.**

I. Introduction

We call "elementary functions" the most commonly used mathematical functions: $\sin$, $\cos$, $\tan$, $\sin^{-1}$, $\cos^{-1}$, $\tan^{-1}$, $\sinh$, $\cosh$, $\tanh$, $\sinh^{-1}$, $\cosh^{-1}$, $\tanh^{-1}$, and exponentials, and logarithms in radices e, 2 and 10. This paper is devoted to the approximate evaluation of these functions (we will also add to the list, in Section VI the square root and the inverse square root). These functions cannot be obtained exactly through a finite number of arithmetic operations, one can only approximate them. Therefore, in a way, all elementary function algorithms are "approximate algorithms". The usual implementations of the elementary functions aim at very good accuracy (the ultimate goal being correct rounding: we always obtain the machine number nearest to the exact result). However, as pointed-out in [1] some applications (e.g., machine learning, pattern recognition, digital signal processing, robotics, and multimedia) only require at times a rather rough estimate of a function.

Hence, for these applications, it is worth examining if somehow relaxing the accuracy constraints can lead to a significant gain in terms of performance.

This of course is not a new idea. Before the advent of pocket calculators, many recipes were used for obtaining a quick approximation to some function in mental math. Most of them were linear or piecewise linear approximations, with very simple slopes so that multiplications could be avoided as much as possible. For example, in decimal arithmetic, a rule of thumb[1] for estimating the decimal logarithm of $m \times 10^k$ (where $1 \leq m < 10$) was to replace it by $k + m/10$. The maximum absolute error was less than 0.204: this is not big accuracy, but much enough for estimating the pH of an acid solution in mental math.

From a computational point of view, an essential difference between the elementary functions and the arithmetic operations is that the elementary functions are *nonlinear*. A consequence of that nonlinearity is that a small input error can sometimes imply a large output error. Hence a very careful error control is necessary, especially when approximations are cascaded: *approximate* computing cannot be *quick and dirty* computing. For instance, implementing x^y as $2^{y \log_2(x)}$ and using for that approximate radix-2 logarithm and exponential can lead to a very inaccurate result. To illustrate that, assume that we use the binary32/single precision format of the IEEE 754 Standard for Floating-Point Arithmetic [2], that the radix-2 logarithms and exponentials are computed using Mitchell's approximate algorithm (formulas (2) and (3) below), and that the multiplication $y \times \log_2(x)$ is correctly rounded in the binary32 format. With the floating-point inputs $x = 1093369/2^{20} \approx 1.042718$ and $y = 5421709/2^{14} \approx 330.915$, the computed result is $1191181/64 \approx 18612.2$, whereas the exact value of x^y is $1027254.94\cdots$. Even the order of magnitude is wrong.

There are, roughly speaking, three families of elementary function algorithms [3], and for each of these families, one can find strategies for improving speed at the cost of a (controlled!) loss in terms of accuracy:

- **shift-and-add algorithms** require a number of iterations proportional to the desired number of digits in the result. This is a rather slow convergence, but their main advantage is that the iterations are very simple, multiplication-free (they just use additions and shifts), and use a very small silicon area. The most famous algorithm in that family is CORDIC [4], [5];
- **polynomial or rational approximations** where the function to be evaluated is replaced by an approximating polynomial or rational function, preliminarily obtained using Remez's algorithm [6] or a more sophisticated method [7];
- **table-based methods** that range from "pure tabulation" of the function when the precision is very small (we just store its value for all possible inputs) to most sophisticated algorithms that use lookups in tables and a few arithmetic operations.

There are also in-between methods that combine two of these strategies (for example Tang's "table driven" methods, e.g., [8], [9], that use tables to reduce the input argument to a tiny interval, and polynomials of low degree for approximating the function in that interval). In general the shift-and-add algorithms are more suited for hardware implementations, the polynomial/rational approximations can be used in hardware or software, and there are table-based methods targeted either at software [10] or hardware [11] implementation.

To these three classical classes of methods, one can add another one, specific to approximate computing: **bit-manipulation techniques** [12], [13], [14], that use specific properties of the binary format used for representing numbers (for instance: in floating-point arithmetic, the exponent part of the representation of x encodes an approximation to $\log_2 |x|$).

The paper is organized as follows. In Section II, we review some classical approximate methods for rough elementary

[1]Useful in chemistry and described at https://www.studentdoctor.net/2018/12/20/quickly-calculate-logarithms-without-a-calculator-mcat-tips-and-tricks/

function approximation. As said above, many of them boil down to piecewise linear approximations, where the slopes are very simple (typically, powers of 2), so that multiplication by them is straightforward. Section III deals with shift-and-add algorithms. Section IV is devoted to polynomial and rational approximations of functions. In Section V, we examine some of the recent table-based methods. Finally, Section VI presents bit-manipulation techniques.

II. Some Classical Tricks of the Trade

for the radix-2 logarithm [15] is a well-known bit-manipulation technique. It consists in approximating

$$\log_2\left(2^k \cdot (1+f)\right), \tag{1}$$

where $0 \leq f < 1$, by

$$k+f, \tag{2}$$

with absolute error bounded by 0.0861. Mitchell's goal was to use an approximate radix-2 logarithm and exponential to implement approximate multiplication and division. Figure 1 shows a plot of Mitchell's approximation: it clearly illustrates the fact that Mitchell's algorithm computes a linear interpolation of the logarithm function at the powers of 2.

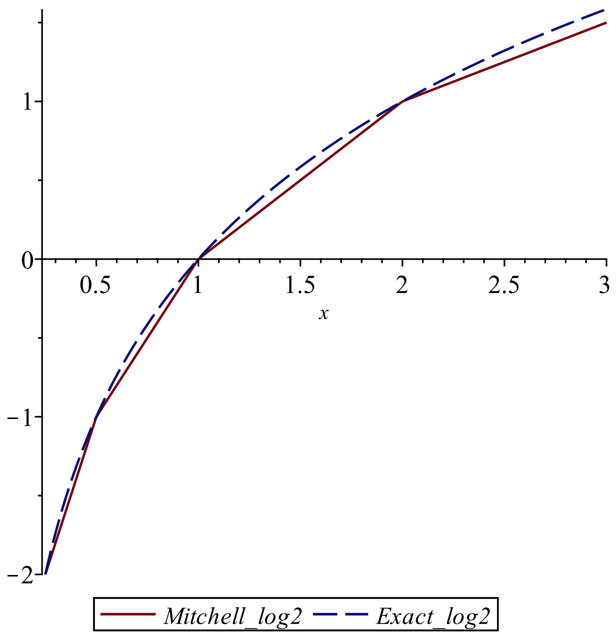

Fig. 1. Mitchell's approximation to $\log_2(x)$, along with the exact $\log_2$ function, plotted for $1/4 \leq x \leq 3$.

Mitchell's approximation is nothing but the piecewise linear interpolation of $\log_2(x)$ at the powers of 2. Its interesting property is that it comes at almost no cost: a hardware implementation of that method, assuming operands in binary fixed-point arithmetic, requires a leading zero counter for finding k, and a shifter for extracting the bits of x. We need no multiplication, and no table lookup. Since the approximation is always below the true radix-2 logarithm, one can diminish the absolute error at the cost of an extra addition, by adding half the maximum error of the approximation (2) to it.

The radix-2 exponential of x is, in a similar fashion, approximated by

$$2^{\lfloor x \rfloor} \cdot (1 + \mathrm{frac}(x)). \tag{3}$$

Variants of Mitchell's algorithm have been introduced, where $\log_2\left(2^k \cdot (1+x)\right)$ is approximated by $k+\ell(x)$, where $\ell(x)$ is a piecewise linear function of x [16], [17], with very simple linear coefficients (so that a multiplication by them reduces to a few additions and shifts). Marino suggests using $\ell(x)$ equal to x plus a piecewise order-2 correcting term, with a suitable approximation to x^2 [18].

Mitchell's technique, possibly with variants, is still used and in the recent period authors have suggested using it for implementing logarithmic number systems [19] and neural networks [20]. Exponentials and logarithms are important in deep neural networks [21]. Approximate multipliers that extend Mitchell's technique are presented and analyzed in [22].

Functions $\arctan(x)$ and $\arctan(y/x)$ are useful in signal processing and computer vision, hence several authors have suggested very simple approximations to them (a comparison of the methods known in 2011 is given in [23]). Abramowitz and Stegun [24] have suggested, for $|x| \leq 1$:

$$\arctan x \approx \frac{x}{1+0.28x^2}. \tag{4}$$

If $|x| > 1$, one can use $\arctan(x) = \pi/2 - \arctan(1/x)$. The absolute error of the approximation (4) is less than 4.883×10^{-3}. Unfortunately, the constant 0.28 is not representable with a finite number of bits in binary. It must be rounded and the resulting rounding error must be taken into consideration. Lyons [25] rather suggests, still for $|x| \leq 1$, the approximation

$$\arctan x \approx \frac{x}{1+0.28125x^2}. \tag{5}$$

The approximation error of (5) is slightly larger (just below 4.911×10^{-3}) than that of (4), but, even if is not straightforward when it is written in decimal, the constant 0.28125 is much more friendly: it is equal to $9/32 = 2^{-2} + 2^{-5}$, so that multiplying by that constant boils down to performing an addition and two shifts. Variants can be explored. Using the Sollya tool presented in Section IV, one finds the following approximation to $\arctan(x)$ in $[-1, 1]$:

$$\arctan x \approx \frac{x}{0.999755859375+0.03125\cdot|x|+0.24609375\cdot x^2}. \tag{6}$$

The absolute error of that approximation is 2.374×10^{-3}. Since $0.03125 = 2^{-5}$, multiplying by that coefficient reduces to performing a shift, and since $0.24609375 = 2^{-2} - 2^{-8}$, multiplying by that coefficient boils down to performing one addition and two shifts. The constant coefficient of the denominator, 0.999755859375, is in fact very simple: it is equal to $1 - 2^{-12}$.

Girones et al. [26] consider approximations of $\arctan(y/x)$ suitable for object recognition in computer vision. For (x, y) in the first quadrant, their favored approximation is of the form

$$\arctan\left(\frac{y}{x}\right) \approx \frac{\pi}{2} \cdot \frac{kxy + y^2}{x^2 + 2kxy + y^2}, \tag{7}$$

with $k \approx 0.596227$. The absolute error of (7) is bounded by 0.00283. Girones et al. give similar approximations for the other quadrants. See also [27]. Approximation (7) corresponds to an approximation to $\arctan(z)$ (with $z = y/x$) valid for all real z:

$$\arctan(z) \approx \text{sign}(z) \cdot \frac{k \cdot |z| + z^2}{1 + 2k \cdot |z| + z^2}. \tag{8}$$

Such "uniform" approximations have larger errors than approximations in a closed interval, but they allow one to avoid tests and range reduction techniques. The simplest one foir tha arctangent function is the following [28]:

$$\arctan x \approx \frac{\pi}{2} \cdot \frac{x}{|x| + 1}, \tag{9}$$

for all x, with absolute error ≤ 0.072.

As noticed by Markstein [29], one frequently needs the cosine and sine of the same angle θ simultaneously (e.g., for performing rotations or computing Fourier transforms). Even in classical (i.e., nonapproximate) computing, this allows one to save delay, memory and power by sharing large parts of the calculation (range reduction, processing of "special" cases such as NaNs, even polynomial approximations) between the two functions. Below is an elegant method suggested in [30] for obtaining a (rather rough, however—there are slightly better, yet slightly less simple, other methods in [30]) approximation to these two values simultaneously. We assume $|\theta| \leq \pi/2$. First, compute

$$x = K \cdot \theta - \frac{1}{2}, \tag{10}$$

with

$$K = \frac{81}{128} = 0.1010001_2. \tag{11}$$

Then, compute

$$S = -x^2 + \frac{3}{4} + x, \tag{12}$$

and

$$C = -x^2 + \frac{3}{4} - x. \tag{13}$$

Note that the only difference between S and C is the last operation: most of the computation is common. Constant K is an approximation to $2/\pi$. Multiplying by K only requires three shifts and 2 additions, since its binary representation contains only 3 ones. We have

$$|S - \sin(\theta)| < 0.054, \tag{14}$$

and

$$|C - \cos(\theta)| \leq 0.063. \tag{15}$$

Fig. 2 gives a plot of C and S, compared with $\sin(\theta)$ and $\cos(\theta)$.

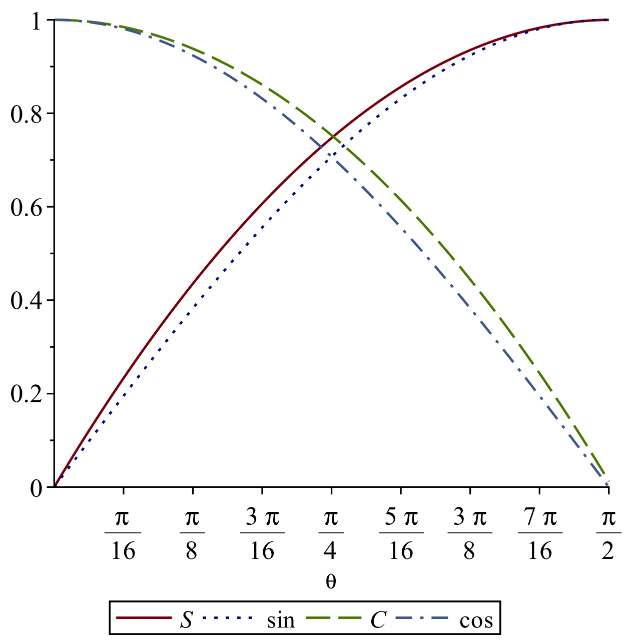

Fig. 2. The two approximations S and C to $\sin(\theta)$ and $\cos(\theta)$ given by (12) and (13) [30].

III. Shift-and-Add Algorithms

Shift-and-Add algorithms can be traced back to one of the first methods (the "radix method") designed by Henry Briggs (1561–1631) for building the first tables of logarithms [31]. In its first and simplest version, due to Volder [4], the CORDIC algorithm based upon the following iteration,

$$\left\{ \begin{array}{lll} x_{n+1} & = & x_n - d_n y_n 2^{-n} \\ y_{n+1} & = & y_n + d_n x_n 2^{-n} \\ z_{n+1} & = & z_n - d_n \arctan 2^{-n}, \end{array} \right. \tag{16}$$

where the terms $\arctan 2^{-n}$ are precomputed and stored, and $d_n = \pm 1$. Hence the only "multiplications" that appear in (16) are by ± 1 and by powers of 2. They are straightforwardly implemented in hardware (especially if n is small, which will be the case if we aim at low accuracy). Detailed descriptions of CORDIC can be found in [32], [33], [3].

In the *rotation mode* of CORDIC, we choose $d_n = \text{sign}(z_n)$. If

$$|z_0| \leq \sum_{k=0}^{\infty} \arctan 2^{-k} = 1.74328662047234 \cdots, \tag{17}$$

then $(x_n, y_n, z_n)^t$ converges to

$$K \times \begin{pmatrix} x_0 \cos z_0 - y_0 \sin z_0 \\ x_0 \sin z_0 + y_0 \cos z_0 \\ 0 \end{pmatrix}, \tag{18}$$

where $K = \prod_{n=0}^{\infty} \sqrt{1 + 2^{-2n}} = 1.646760258121 \cdots$. For instance, the sine and cosine of z_0 can be computed by choosing $x_0 = 1/K$ and $y_0 = 0$. In the *vectoring mode*, we choose $d_n = -\text{sign}(y_n)$, and we obtain

$$z_n \rightarrow z_0 + \arctan\left(\frac{y_0}{x_0}\right). \tag{19}$$

Slight modifications to (16) make it possible to evaluate several other functions with CORDIC: $\sqrt{x}$, $\cosh(x)$, $\sinh(x)$, $\arcsin(x)$, $\arccos(x)$, e^x, $\ln(x)$, etc.

A more sophisticated algorithm, for evaluating exponentials and logarithms, is the following (it is a variant of an algorithm introduced by Takagi in [34]). We perform the iteration

$$\begin{array}{rcl} L_{n+1} & = & L_n - \ln\left(1 + d_n 2^{-n}\right) \\ E_{n+1} & = & E_n\left(1 + d_n 2^{-n}\right) = E_n + d_n E_n 2^{-n}, \end{array} \tag{20}$$

with L_n and E_n represented in signed-digit arithmetic (i.e., radix 2 and digits -1, 0 and 1) to allow fully parallel, carry-free additions (a very similar strategy is possible with carry-save arithmetic), with the constants $\ln\left(1 \pm 2^{-n}\right)$ pre-calculated and stored, and where d_n is chosen as follows:

- for computing exponentials: if L_n^* is $2^n L_n$ truncated after the first fractional digit, we choose

$$d_n = \left\{ \begin{array}{rll} -1 & \text{if} & L_n^* \leq -1 \\ 0 & \text{if} & -1/2 \leq L_n^* \leq 0 \\ +1 & \text{if} & L_n^* \geq 1/2. \end{array} \right. \tag{21}$$

  This choice will ensure $E_n \rightarrow E_1 e^{L_1}$, provided that

$$\begin{array}{rcl} L_1 & \in & \left[\sum_{i=1}^{\infty} \ln\left(1 - 2^{-i}\right), \sum_{i=1}^{\infty} \ln\left(1 + 2^{-i}\right)\right] \\ & \approx & [-1.24206, 0.868876]. \end{array} \tag{22}$$

- for computing logarithms: if λ_n is $2^n(E_n - 1)$ truncated after the first fractional digit, we choose

$$d_n = \left\{ \begin{array}{rll} +1 & \text{if} & \lambda_n \leq -1/2 \\ 0 & \text{if} & \lambda_n = 0 \text{ or } 1/2 \\ -1 & \text{if} & \lambda_n \geq 1. \end{array} \right. \tag{23}$$

  This choice will ensure $L_n \rightarrow L_1 + \ln(E_1)$, provided that

$$\begin{array}{rcl} E_1 & \in & \left[\prod_{i=1}^{\infty}(1 - 2^{-i}), \prod_{i=1}^{\infty}(1 + 2^{-i})\right] \\ & \approx & [0.28879, 2.38423]. \end{array} \tag{24}$$

For details, proofs, similar algorithms, and how the choice (23) can be implemented just by examining a window of 4 digits, see [3].

A detailed *ad hoc* error analysis that takes into account the chosen algorithm, the input domain and the precision of the intermediate operands is necessary, but roughly speaking, to obtain n bits of accuracy in the result with these algorithms, we need to perform slightly more than n iterations.

Hence, to obtain fast approximations with these algorithms, a first solution is just to *stop the iterations* as soon as we have enough accuracy for the target application. This allows for a fine tuning of the tradeoff between speed and accuracy. This is what Mikaitis et al. do for implementing the exponential function on a neuromorphic chip [35], using iteration (20). This is a clear advantage of shift-and-add algorithms: with the other classes of algorithms (polynomials, tables), the implementation is designed for a specific target accuracy, whereas the same implementation of CORDIC, for instance, can be used for 4-bit or for 64-bit approximations, it suffices to stop the iterations at the desired step. However, the intrinsically linear convergence of these algorithms (number of bits of accuracy proportional to the number of iterations) makes them less interesting than polynomial approximations beyond single precision.

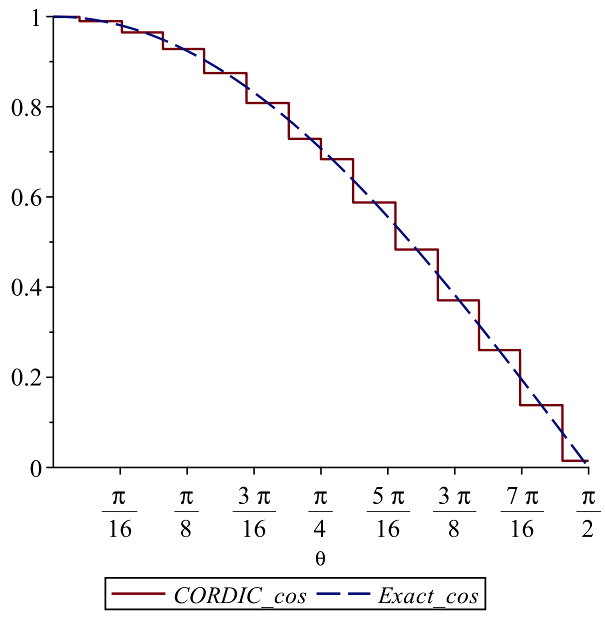

Fig. 3. CORDIC approximation to $\cos(x)$ with 5 iterations only, plotted with the real function $\cos(x)$. The number of scales doubles each time we perform one more iteration.

A more subtle way of simplifying the calculations at the cost of a small loss in accuracy is to notice that the constants $\arctan(2^{-n})$ and $\ln(1 \pm 2^{-n})$ that appear in (16) and (20) satisy

$$\left|\arctan(2^{-n}) - 2^{-n}\right| \approx \frac{2^{-3n}}{3}, \tag{25}$$

$$\left|\ln(1 + 2^{-n}) - 2^{-n}\right| \approx \frac{2^{-2n}}{2}, \tag{26}$$

and

$$\left|\ln(1 - 2^{-n}) + 2^{-n}\right| \approx \frac{2^{-2n}}{2}. \tag{27}$$

Hence, if 2^{-t} is the target accuracy, as soon as $n > t/3$, $\arctan(2^{-n})$ can be replaced by 2^{-n} in (16), and as soon as $n > t/2$, $\ln(1 + d_n 2^{-n})$ can be replaced by $d_n 2^{-n}$ in (20). Beyond the obvious gain in terms of silicon area, this can be exploited in several ways: Ahmed [36] uses that property for replacing the last $t/2$ iterations of CORDIC by one multiplication (by remarking that when $\arctan(2^{-n})$ is replaced by 2^{-n} starting from step n_0, iteration (16) is nothing but a multiplication by z_{n_0}). Baker [37] uses that property for avoiding the comparisons required for choosing d_n. In a similar fashion, Juang et al. [38] use that property for generating the terms d_n in (16) in parallel in two phases in a dedicated architecture, and Chen et al. [39], [40] modify their architecture to avoid the need for two phases for generating the terms d_n, at the cost of being nonexact: their algorithm is an intrinsically approximate CORDIC algorithm.

Another way of accelerating shift-and-add algorithms is to consider high radix variants [41], that require less iterations at the cost of significantly more complex iterations. To our knowledge, this has never been done for doing approximate computing.

IV. Polynomial and Rational Approximations

Polynomial approximations are the most widely used method for designing elementary function software. They can be used for any accuracy target (a few bits to hundreds of bits). If $\mathcal{P}_n$ is the set of the polynomials of degree less than or equal to n, the minimax degree-n polynomial for function f in $[a,b]$ is the polynomial p^* that satisfies

$$\max_{x\in[a,b]} |f(x)-p^*(x)| = \min_{q\in\mathcal{P}_n}\left(\max_{x\in[a,b]}|f(x)-q(x)|\right). \quad (28)$$

An algorithm due to Remez [6], [42] computes p^*. The approximation error decreases with n, but the speed with which it decreases depends much on the function (see [3, Table 3.3 and Fig. 3.9]).

A straightforward way of saving delay, silicon area (for hardware implementations) and energy using polynomial approximations is to lower the degree of the approximating polynomial. However, one can replace that solution or combine it with a more involved trick: one can force some of the polynomial coefficients to be very simple (say, they fit into a very small number k of bits), so that a multiplication by them reduces to a very small number of additions. One can even force some small coefficients to be zero. However, all this must be done with care: as we are going to see below with an example, rounding to k bits the coefficients of the minimax polynomial does not, in general, give the best approximation to f among the elements of $\mathcal{P}_n$ whose coefficients fit into k bits.

Such “sparse-Coefficient” polynomial approximations have been considered in [43]. A first strategy for obtaining them was introduced in [44]. The state-of-the-art tool for obtaining such approximations is Sollya[2], developed by Chevillard, Joldes and Lauter [45]. Sollya uses techniques presented in [7] by Brisebarre and Chevillard, based on the theory of Euclidean lattice reduction.

To illustrate what can be done with Sollya, let us consider the following example. One wishes to approximate the exponential function in $[0,1]$ by a very small polynomial: a degree-2 polynomial, with coefficients that fit in at most 4 bits (so that a multiplication by such coefficients is straightforward).

First, let us eliminate the degree-2 Taylor expansion of the exponential at the origin: it approximates the exponential in $[0,1]$ with maximum error around 0.218, which is quite large.

The minimax approximation to e^x in $[0,1]$ is given by the Remez algorithm. It is equal to

$$\begin{aligned} P_1(x) &= 1.0087560221136893228\cdots \\ &+ 0.8547425734330620925\cdots \times x \\ &+ 0.8460272107986044971 9\cdots \times x^2. \end{aligned} \quad (29)$$

Such a polynomial can be generated by the Sollya `Remez` command. The error of that approximation can be obtained using the Sollya command

```
supnorm(P1,exp(x),[0;1],absolute,2^(-40));
```

That command generates an interval, of length bounded by around 2^{-40} that is *guaranteed* to contain the approximation error. Here, the obtained error is 8.7561×10^{-3}. Unfortunately, the polynomial P_1 does not satisfy our requirements: its coefficients do not fit in 4 bits. In general, the coefficients of the minimax polynomial to a nontrivial function do not fit exactly in a finite number of bits. The first idea that springs in mind is to round each of the coefficients of P_1 to 4 bits. This gives a polynomial P_2 equal to

$$P_2(x) = 1 + \frac{7}{8}x + \frac{7}{8}x^2. \quad (30)$$

That polynomial approximates e^x in $[0,1]$ with maximum error 3.671×10^{-2}: we have lost much accuracy by rounding the coefficients of P_1. However, there is no reason for P_2 to be the best approximating polynomial to e^x among the degree-2 polynomials with 4-bits coefficients.

We obtain a best or nearly best approximating polynomial to e^x among the degree-2 polynomials with 4-bits coefficients using the Sollya command line

```
P3 = fpminimax(exp(x),2,[|4...|],[0;1],absolute);
```

The obtained polynomial is

$$P_3(x) = 1 + \frac{15}{16}x + \frac{3}{4}x^2, \quad (31)$$

and the approximation error is less than 3.0782×10^{-2}, which is significantly better than the one of P_2. Sollya returns a *certified approximation error* (we can ask it to output a proof), obtained using techniques presented in [46]. The only “real” multiplication[3] required when evaluating P_3 is for computing x^2: the multiplications by $15/16$ and $3/4$ are straightforwardly replaced by one shift and one subtraction. Fig. 4 plots the difference between $P_3(x)$ and e^x, and the difference between $P_2(x)$ and e^x.

With polynomial approximations, chance plays a role: depending on the function, the domain, and the requested accuracy, we may have very low degree and very simple coefficients (such as powers of 2) which will make these approximations very attractive, or not.

To obtain the overall error when using a polynomial approximation p to some function f, we must add to the “approximation error” (i.e., the maximum of $|f-p|$ in the considered interval) the error committed when evaluating p. Errors on evaluating a polynomial can be computed using Melquiond’s Gappa tool [47], [48], [49].

[2]http://sollya.gforge.inria.fr

[3]And of course it can be an *approximate* multiplication: there is no need for it to be much more accurate than the final approximation to the function.

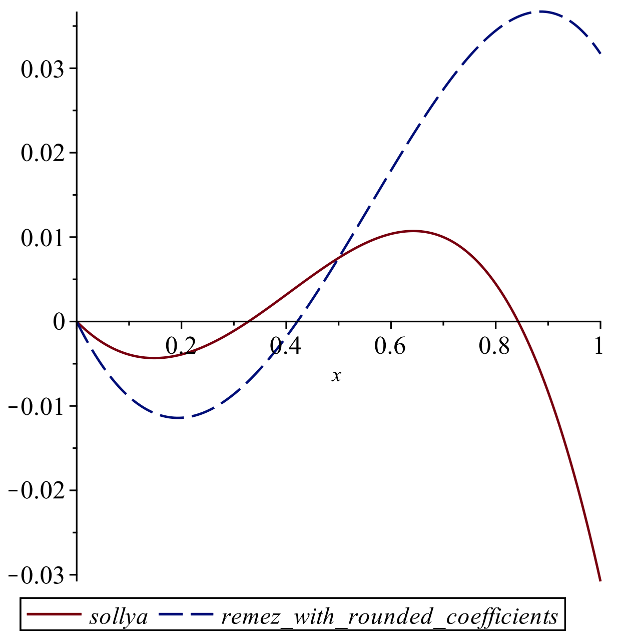

Fig. 4. Plot of $P_3(x)-e^x$ (obtained through Sollya) and $P_2(x)-e^x$ (Remez polynomial with coefficients rounded to 4 bits), for $x \in [0,1]$.

Some authors have attempted to generate function approximations automatically [50], [51]. This allows one to explore a large design space (one can try many different solutions), and to compute "just right": the approximations are especially suited to the exact needs of the considered application. "Computing just right" is the motto of the Flopoco tool[4]. That tool was built by De Dinechin and his students [52], [53]. It mainly (but not solely) targets FPGA implementation of arithmetic operators and functions. An example of the use of Flopoco for computing reasonably low precision (12 or 16 bits) $\arctan(y/x)$ is given in [54].

Polynomials approximations are valid in a bounded interval I only. One needs to use *ad hoc* (i.e., dependent on algebraic properties of the function being approximated) techniques to reduce the initial input argument to some element of I. Some choices may seem obvious at first glance, but one needs to be cautious. For instance, Goldberg [55] shows that for the $\log_2$ function, the counterintuitive choice $I = [0.75, 1.5]$ is far better than the straightforward choice $I = [1, 2]$.

Division is a significantly slower arithmetic operation than multiplication. As a consequence, when very fast approximations are at stake, rational approximations to functions are seldom advisable. However, there are exceptions to that rule:

- some functions have a behavior (such as poles, or finite limits at $\pm\infty$) that is "highly nonpolynomial". Correctly approximating these functions by polynomials would require large degrees and/or dividing the input domain into many subdomains, with a different approximation for each subdomain (and, at run time, many tests necessary for finding in which subdomain the input variable lies);
- in some cases, one may want to obtain *uniform* approximations to a given function in the whole real line (or in a half line), to avoid having to perform a preliminary range reduction. This will, in general, be impossible to do with polynomials: they have only one kind of behavior at $\pm\infty$.

[4]http://flopoco.gforge.inria.fr

An illustration of these exceptions is the approximation (9) to the arctangent, valid for all real inputs. Incidentally, one can improve (9) slightly. For instance, one can obtain

$$\arctan(x) \approx \frac{x}{\frac{85}{128} + \frac{157}{256} \cdot |x|}, \qquad (32)$$

for all real x, with absolute error less than 6.24×10^{-2}, or

$$\arctan(x) \approx \frac{x + x^2}{\frac{31}{32} + \frac{61}{64} \cdot x + \frac{655}{1024} \cdot x^2}, \qquad (33)$$

for all positive real x, with error less than 7.44×10^{-3}.

Winitzki [28] gives the approximation (9). He also gives the following uniform (valid in $[0, +\infty)$) approximation to $\operatorname{erf}(x)$:

$$\operatorname{erf}(x) = 1 - \frac{e^{-x^2}}{x\sqrt{\pi}} \cdot g(x), \qquad (34)$$

with

$$g(x) \approx \frac{x\sqrt{\pi} + (\pi - 2)x^2}{1 + x\sqrt{\pi} + (\pi - 2)x^2}. \qquad (35)$$

Note in (35) the similarity between the numerator and denominator: almost all the computation necessary for their evaluation is shared). Winitzki claims that it provides a uniform approximation to erf with an error less than 0.02. He also gives uniform approximations to Bessel functions, the Airy function, and Lambert's W function. His approximations somehow generalize Padé-Hermite approximants [56]: he builds rational functions whose first terms of the series expansions at 0 and ∞ coincide with those of the function.

V. Table-Based Methods

A. *Simple tabulation of the function*

When we need less than around 8 to 12 bits of accuracy, it is tempting to simply tabulate the function in a table addressed by the most significant bits of the input operand. Let f be the function, let $x = 0.x_1x_2x_3 \cdots x_n$ be the input operand, represented in binary fixed point arithmetic (assuming $0 \le x < 1$ to simplify the presentation), and assume that we build a table with p address bits (i.e., 2^p elements). Let $T_{x_1x_2x_3\cdots x_p}$ be the value stored in the table at the location addressed by the first p bits of x. For all numbers in the interval

$$I_{x_1x_2x_3\cdots x_p} = \left[0.x_1x_2x_3\cdots x_p, 0.x_1x_2x_3\cdots x_p + 2^{-p}\right),$$

the same result will be returned. What is the best value to store? The first idea that springs in mind is to choose

$$T_{x_1x_2x_3\cdots x_p} = f(0.x_1x_2x_3\cdots x_p). \qquad (36)$$

In general (36) is not the best idea: in most cases, f is monotonic in the interval $I_{x_1x_2x_3\cdots x_p}$, so that in that interval, $f(x)$ will either be always larger or always less than $f(0.x_1x_2x_3\cdots x_p)$. The value that minimizes the absolute error is

$$T_{x_1x_2\cdots x_p} = \frac{1}{2}\left(\min_{I_{x_1x_2\cdots x_p}} f(x) + \max_{I_{x_1x_2\cdots x_p}} f(x)\right). \qquad (37)$$

Since in general f is monotonic in $I_{x_1x_2x_3\cdots x_p}$, the "min" and the "max" in (37) are almost always attained at the two extremal points $0.x_1x_2\cdots x_p$ and $0.x_1x_2\cdots x_p + 2^{-p}$. Interestingly enough, if the value being tabulated serves as a seed value for a few Newton-Raphson iterations, storing the value (37) that minimizes the *initial error* (i.e., the distance between the seed value and the value of the function) is not necessarily the best choice: one can store in the table a value that minimizes the *final error* (i.e., the distance between the result obtained after the iterations and the value of the function). This is investigated in [57].

B. Bipartite and multipartite table methods

Since the size of a table increases exponentially with the number of address bits, simple tabulation of a function (as described in the previous section) becomes impossible if accurate results are at stake. The bipartite and multipartite table methods make it possible to extend the domain of applicability of table methods to around 20 bits of accuracy. Sunderland et al. [58], have suggested a way of computing an approximation to the sine of a 12-bit number x less than $\pi/2$, using small tables. Their method consists in splitting the binary representation of x into three 4-bit words A, B, and C, with $A < \pi/2$, $B < \pi/32$ and $C < \pi/512$, so that $x = A+B+C$, and to use

$$\sin(A+B+C) \approx \sin(A+B) + \cos(A)\sin(C). \tag{38}$$

Formula (38) is easily deduced from

$$\begin{array}{rcl} \sin(A+B+C) & = & \sin(A+B)\cos(C) \\ & + & \cos(A+B)\sin(C), \end{array} \tag{39}$$

and the fact that, since C is small, $\cos(C)$ is very close to 1 and $\sin(C)$ is small enough so that approximating $\cos(A+B)$ by $\cos(A)$ in (39) has little influence on the final result.

By using (38), instead of one table with 12 address bits (i.e., with 2^{12} elements), one needs two tables—one for $\sin(A+B)$ and one for $\cos(A)\sin(C)$)—each of them with 8 address bits only. This results in a total table size 8 times smaller. The largest absolute error committed by using (38) is $8.765 \times 10^{-4} = 2^{-10.16}$.

This was the first use of what is now called the *bipartite table method*. That method was re-discovered (in a totally different context) and named *bipartite* by DasSarma and Matula [59], who wanted to generate seed values for computing reciprocals using the Newton–Raphson iteration. The bipartite method was later on generalized to arbitrary functions by Schulte and Stine [60], [61], [62].

Assume we wish to approximate $f(x)$, where x is a p-bit fixed-point number in $[0,1]$. Define $k = \lceil p/3 \rceil$. We split x into three k-bit numbers x_0, x_1, and x_2, so that

$$x = x_0 + 2^{-k}x_1 + 2^{-2k}x_2,$$

where x_i is a multiple of 2^{-k} and $0 \leq x_i < 1$. The bipartite method consists in approximating $f(x)$ by

$$A(x_0,x_1) + B(x_0,x_2) \tag{40}$$

where

$$\begin{array}{rcl} A(x_0,x_1) & = & f(x_0 + 2^{-k}x_1) \\ B(x_0,x_2) & = & 2^{-2k}x_2 \cdot f'(x_0). \end{array} \tag{41}$$

This is illustrated by Figure 5.

How does it work? First, the order-1 Taylor-Lagrange formula for f at point $x_0 + 2^{-k}x_1$ gives

$$\begin{array}{c} f(x_0 + 2^{-k}x_1 + 2^{-2k}x_2) \\ = f(x_0 + 2^{-k}x_1) + 2^{-2k}x_2 \cdot f'(x_0 + 2^{-k}x_1) + \epsilon_1, \end{array} \tag{42}$$

with

$$|\epsilon_1| \leq \frac{1}{2}\cdot(2^{-2k}x_2)^2 \max_{[0,1]}|f''| \leq 2^{-4k-1}\max_{[0,1]}|f''|. \tag{43}$$

Then, we can replace the term $f'(x_0 + 2^{-k}x_1)$ in (42) by $f'(x_0)$, using the order-0 Taylor-Lagrange formula for f':

$$f'(x_0 + 2^{-k}x_1) = f'(x_0) + \epsilon_2, \tag{44}$$

with

$$|\epsilon_2| \leq 2^{-k}\cdot|x_1|\cdot\max_{[0,1]}|f''| \leq 2^{-k}\max_{[0,1]}|f''|. \tag{45}$$

Combining (42), (43), (44), and (45), we obtain

$$f(x_0 + 2^{-k}x_1 + 2^{-2k}x_2) = A(x_0,x_1) + B(x_0,x_2) + \epsilon, \tag{46}$$

with

$$|\epsilon| = |\epsilon_1 + 2^{-2k}x_2\epsilon_2| \leq \left(2^{-4k-1} + 2^{-3k}\right)\cdot\max_{[0,1]}|f''|. \tag{47}$$

If the values of $A(x_0,x_1)$ and $B(x_0,x_2)$ stored in the tables are rounded to the nearest p-bit number, then the total error of the approximation is bounded by

$$2^{-p} + \left(2^{-4k-1} + 2^{-3k}\right)\max_{[0,1]}|f''|. \tag{48}$$

Hence, if $|f''|$ has the same order of magnitude as $|f|$ (otherwise the method needs to be slightly modified), we obtain an error only slightly larger than the one we would get by fully tabulating f, with 2 tables of $2p/3$ address bits instead of one with p address bits, which is a very significant improvement.

Figure 6 gives a plot of the bipartite approximation to $\ln(x)$ in $[1/4,1]$, assuming $k=2$ (this is a toy example, since in this case a $3k$-address-bit table would be very cheap: with a larger k, the approximation would not be discernable from the function on the plot). In Fig. 6, the approximation is always larger than the logarithm. A variant of the bipartite method, called symmetric bipartite tables method [60], [61], [62] makes it possible to use that phenomenon to lower the error.

Figure 7 compares the error of the bipartite approximation to $\ln(x)$ in $[1/2,1]$ for $k=5$ with the error resulting from reading the logarithm in a 15-address-bit table. This corresponds to a rather cheap implementation: we just need to add two values read in tables with $2^{10} = 1024$ elements, instead of reading one value in a table with $2^{15} = 32768$ elements. The price to pay is a significant loss in accuracy: maximum error around 1.2×10^{-4} instead of 6.1×10^{-5}.

The bipartite method is not the only method that decomposes a function into a sum of simpler-to-tabulate functions.

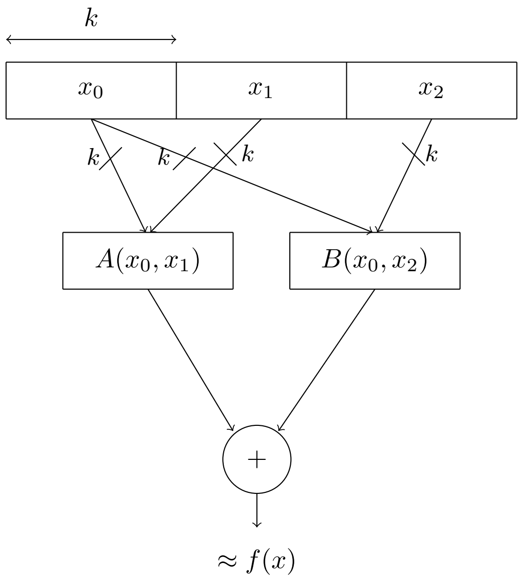

Fig. 5. The bipartite method. The $3k$-bit input x is split into three k-bit numbers x_0, x_1, and x_3. Instead of using one $3k$-address-bit table for storing the values $f(x)$, we use two $2k$-address-bit tables: one stores the values $A(x_0, x_1) = f(x_0 + 2^{-k}x_1)$, and the other one stores the values $B(x_0, x_2) = 2^{-2k}x_2 \cdot f'(x_0)$. The number $f(x)$ is approximated by $A(x_0, x_1) + B(x_0, x_2)$.

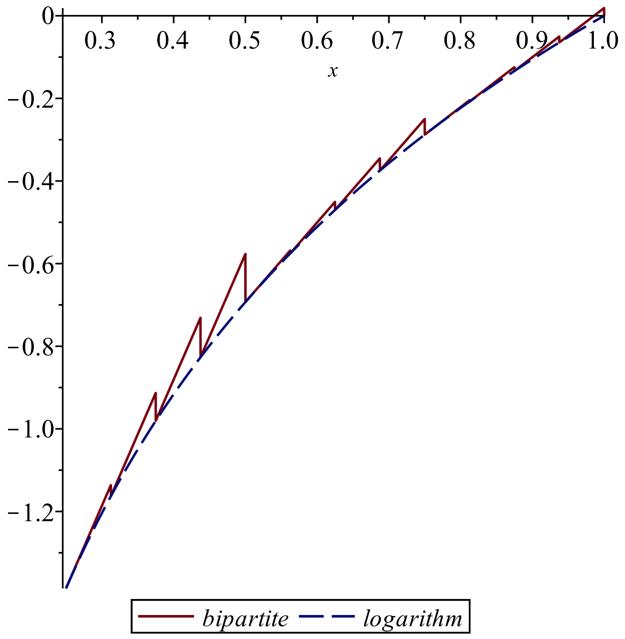

Fig. 6. Bipartite approximation of $\ln(x)$ in $[1/4, 1]$ with $k = 2$, along with the exact $\ln$ function.

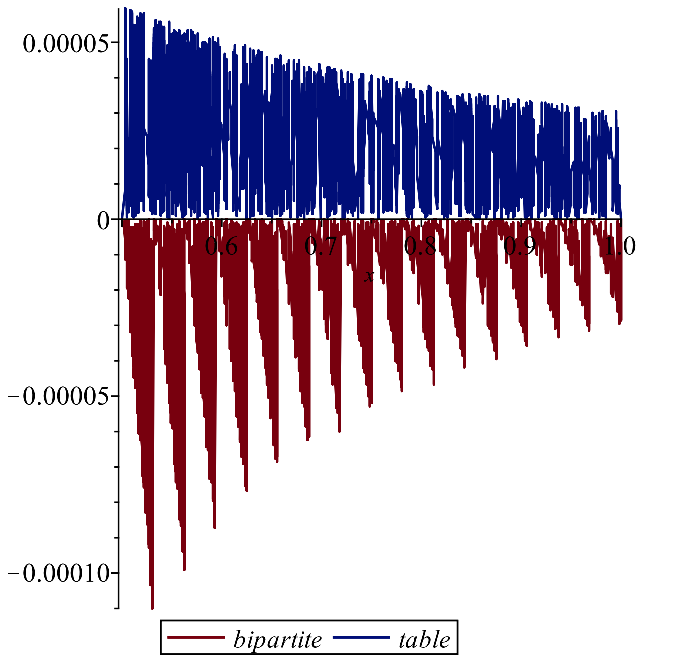

Fig. 7. Error of a bipartite approximation of $\ln(x)$ in $[1/2, 1]$ with $k = 5$, compared with the error of a 15-address-bit table.

Hassler and Takagi have presented such a method, based on the representation of the function by partial product arrays [63].

To improve the bipartite method, instead of using two tables, one can try to use several, even smaller, tables. This leads to the idea of *multipartite table methods*, introduced by Schulte and Stine [62]. Later on, De Dinechin and Tisserand improved these multipartite methods [64] and made them very attractive. A recent analysis of that class of algorithms and a new variant called *hierarchical multipartite methods* are presented by Hsiao et al. in [65].

VI. Bit-Manipulation Techniques

So far, we have adapted to the context of approximate computing techniques that are used in classical computing. Let us now examine methods that are *specific* to approximate computing. These methods are very fast, but do not allow one to obtain more than a few (say, 5-6) bits of precision. Several algorithms use the fact that the exponent part of the floating-point representation of a number x encodes $\lfloor \log_2 |x| \rfloor$. Using shifts and integer operations, one can modify the exponent, or move it to the fraction part. For instance, dividing by two the exponent gives a rough approximation to $\sqrt{x}$ (we will give a somehow better approximation below). That division by 2 cannot be implemented just by a right-shift, because in IEEE-754, the exponents are represented with a *bias*: if x is not a *subnormal* number and $e_x = \lfloor \log_2 |x| \rfloor$, the number stored in the exponent part is $e_x + b$, where b is equal to 15 in the binary16 format, 127 in binary32/single precision, and 1023 in binary64/double precision. Hence in practice, one performs a right-bit shift, and adds (with an *integer* addition, not a floating-point one) a "magic constant" to compensate for the bias (the magic constant can be tuned to minimize the maximum relative error). This gives rise to fast methods, used in computer graphics, for computing square roots or inverse square roots [13], [14] (frequently, the initial estimate is improved by performing one or two steps of the Newton-Raphson iteration). Let us now give a few examples.

Consider the IEEE-754 binary32 (a.k.a. single precision) representation of a floating-point number x, depicted Fig. 8. It is made-up with a 1-bit sign S_x, a 8-bit biased exponent

E_x, and a 23-bit fraction F_x such that

$$x = (-1)^S \times 2^{E_x - 127} \times \left(1 + 2^{-23} \cdot F_x\right). \tag{49}$$

Alternatively, the same bit-chain, if interpreted as 2's complement integer, represents the number

$$I_x = (1 - 2S_x) \cdot 2^{31} + \left(2^{23} \cdot E_x + F_x\right). \tag{50}$$

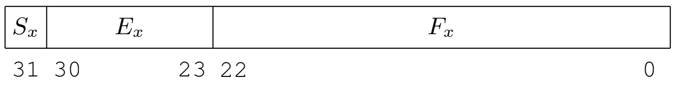

Fig. 8. The IEEE-754 binary32 (a.k.a. single precision) representation of a floating-point number x. The floating-point exponent e_x of x is $E_x - 127$, the significand m_x of x is $1 + 2^{-23} \cdot F_x$, so that $x = (-1)^S \times 2^{E_x - 127} \times \left(1 + 2^{-23} \cdot F_x\right)$.

Let us assume that a fast conversion from integer to floating-point is available. Call `Float` the corresponding function: `Float`(I) is the floating-point number mathematically equal to I. Beware, if $y =$ `Float`(J), and $I_x = J$, x is *not* equal to y: we have *mathematical* equality of the integer J and the real y, and equality of the *binary representations* of J and x.

Assume x is a nonnegative FP number:

$$x = (1 + f_x) \cdot 2^{e_x} = 2^{E_x - 127} \times \left(1 + 2^{-23} \cdot F_x\right). \tag{51}$$

From (50) and $I_1 = 127 \times 2^{23}$, one easily deduces

$$\begin{aligned} 2^{-23}(I_x - I_1) &= (E_x - 127) + 2^{-23} \cdot F_x \\ &= e_x + f_x \\ &\approx \log_2(x). \end{aligned} \tag{52}$$

Blinn [13] suggests to use (52), i.e., to approximate $\log_2(x)$ by $2^{-23} \cdot$ `Float`$(I_x - I_1)$. One easily notices that this is the same approximation (hence, with the same maximum absolute error bound 0.0861) as Mitchell's approximation (2).

The square root approximation suggested by Blinn is slightly more complex. Still consider a floating-point number x as in (51).

- If e_x is even (i.e., E_x is odd), we use the approximation
$$\sqrt{(1 + f_x) \cdot 2^{e_x}} \approx \left(1 + \frac{f_x}{2}\right) \cdot 2^{\frac{e_x}{2}}, \tag{53}$$
i.e., we use the Taylor series for $\sqrt{1+f}$ at $f = 0$;
- if e_x is odd (i.e., E_x is even), we use the approximation
$$\begin{aligned} \sqrt{(1 + f_x) \cdot 2^{e_x}} &= \sqrt{(2 + 2f_x) \cdot 2^{e_x - 1}} \\ &= \sqrt{4 + \epsilon_x} \cdot 2^{\frac{e_x - 1}{2}} \\ &\approx \left(2 + \tfrac{\epsilon_x}{4}\right) \cdot 2^{\frac{e_x - 1}{2}} \\ &= \left(\tfrac{3}{2} + \tfrac{f_x}{2}\right) \cdot 2^{\frac{e_x - 1}{2}}, \end{aligned} \tag{54}$$
i.e., we use the Taylor series for $\sqrt{4 + \epsilon_x}$ at $\epsilon_x = 0$, with $\epsilon_x = 2f_x - 2$.

In both cases, this consists in approximating x by y, such that

$$I_y = \left\lfloor \frac{I_x}{2} \right\rfloor + 127 \cdot 2^{22}. \tag{55}$$

That approximation is *very fast*: one just performs a one-position right shift and an integer addition. However, as one can see on Fig. 9, it is a rather rough approximation. The largest relative error, obtained by exhaustive testing (subnormal numbers excluded) is 0.0607. As one can notice, the approximation is always larger then the exact square-root. Hence it is possible to obtain a better approximation by replacing the constant $127 \cdot 2^{22} = 532676608$ in (55) by a slightly smaller number. Figure 10 shows the relative difference between the approximation and the square root if we replace the constant by 532369100. With that constant, the largest relative error becomes 0.03476.

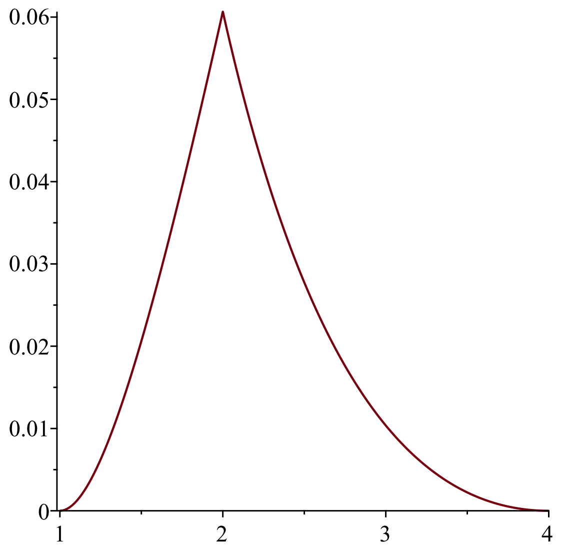

Fig. 9. Relative difference between Blinn's approximation to $\sqrt{x}$ and the actual $\sqrt{x}$ in $[1, 4]$ (function Asqrt in [13]).

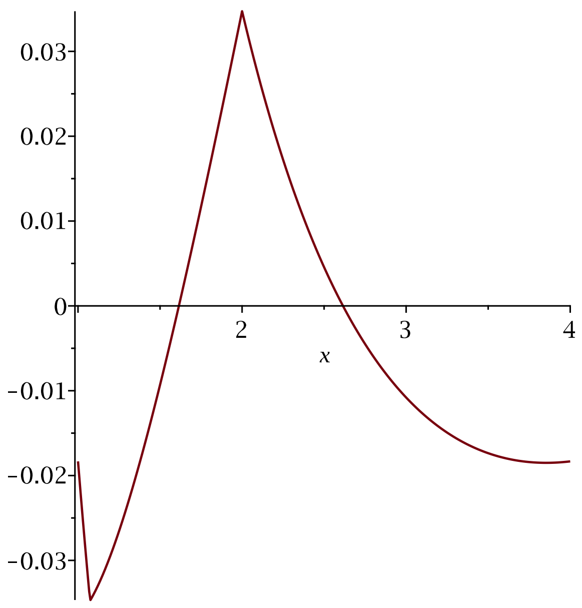

Fig. 10. Relative difference between Approximation (55) to $\sqrt{x}$ with the constant $127 \cdot 2^{22}$ replaced by 532369100 and the actual $\sqrt{x}$ in $[1, 4]$.

Very similarly, the inverse square root of x is approximated by y such that [13]

$$I_y = -\left\lfloor \frac{I_x}{2} \right\rfloor + \mathcal{M}, \tag{56}$$

with

$$\mathcal{M} = 127 \cdot \left(2^{23} + 2^{22}\right). \tag{57}$$

Again, this is a very fast (one shift and one integer addition) yet rather crude approximation, as shown in Fig. 12. The maximum relative error (obtained through exhaustive testing) is 0.0887. As we can notice from Fig. 12, the approximation is

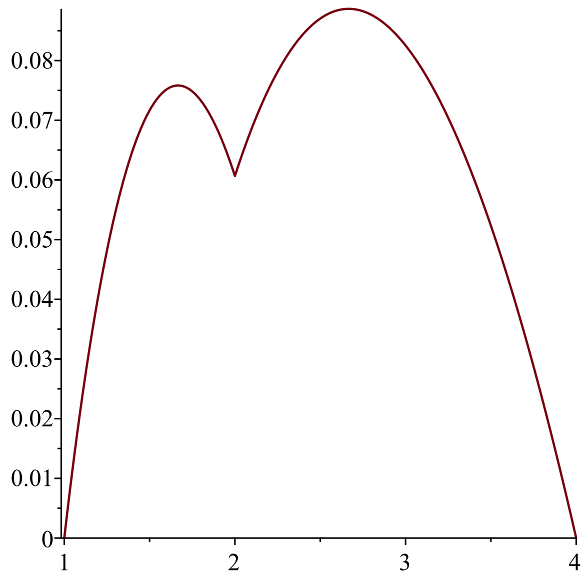

Fig. 11. Relative difference between Blinn's approximation (56) to $1/\sqrt{x}$ and the actual $1/\sqrt{x}$ in $[1, 4]$ (function Ainversesqrt in [13]).

always larger than $1/\sqrt{x}$. One can therefore get a significantly better approximation by replacing in (56) the constant given in Eq. (57) by a slightly different value:

$$\mathcal{M} = 1597463007. \tag{58}$$

That "magic constant" (probably better known by its hexadecimal floating-point representation `0x5F3759DF`) has an interesting history [66]. It has been used in a famous video game. The largest relative error (obtained through exhaustive testing) using that constant is 0.0344. The magic constant (58) is not optimal: Moroz et al. [14] give a slightly better constant, $\mathcal{M} = 1597465647$, for which the largest relative error is 0.03422.

This approximation can be very significantly improved by performing one or two steps of the Newton-Raphson iteration for $1/\sqrt{x}$:

$$y_{n+1} = y_n \cdot \left(\frac{3}{2} - \frac{1}{2} \cdot x \cdot y_n^2\right). \tag{59}$$

Iteration (59) converges to $1/\sqrt{x}$ as soon as $y_0 \in (0, \sqrt{3}/\sqrt{x})$. Convergence is very fast (quadratic) if y_0 is close to $1/\sqrt{x}$. That iteration is frequently used [67] for implementing the inverse square root and the square root ($\sqrt{x}$ is obtained as $x \times (1/\sqrt{x})$). See [68] for more details.

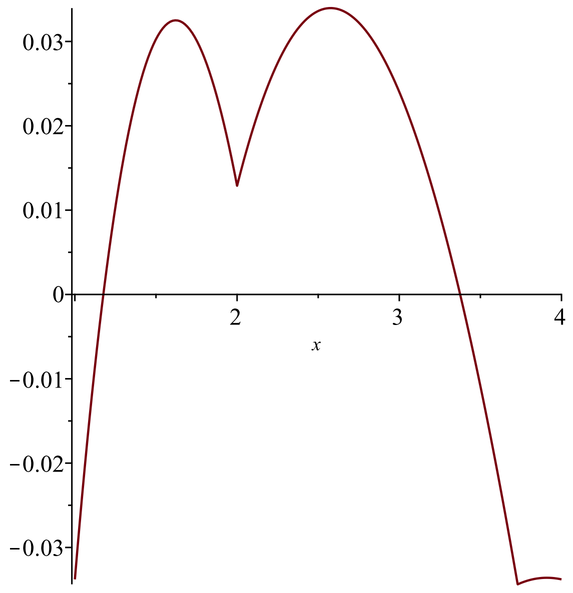

Fig. 12. Relative difference between approximation (56) to $1/\sqrt{x}$ and the actual $1/\sqrt{x}$ in $[1, 4]$ with the "magic constant" $127 \cdot \left(2^{23} + 2^{22}\right)$ replaced by 1597463007.

The coefficients $3/2$ and $1/2$ in (59) are the best ones only asymptotically (as the number of iterations goes to infinity or, equivalently, as y_n goes to $1/\sqrt{x}$). However, as noticed by Blinn [13], if one performs just one or two iterations with the first term y_0 deduced from (56), one can get a better accuracy with slightly different coefficients. Blinn suggests

$$y_1 = y_n \cdot \left(1.47 - 0.47 \cdot y_n^2\right). \tag{60}$$

Figure 13 plots the relative error of these various approximations.

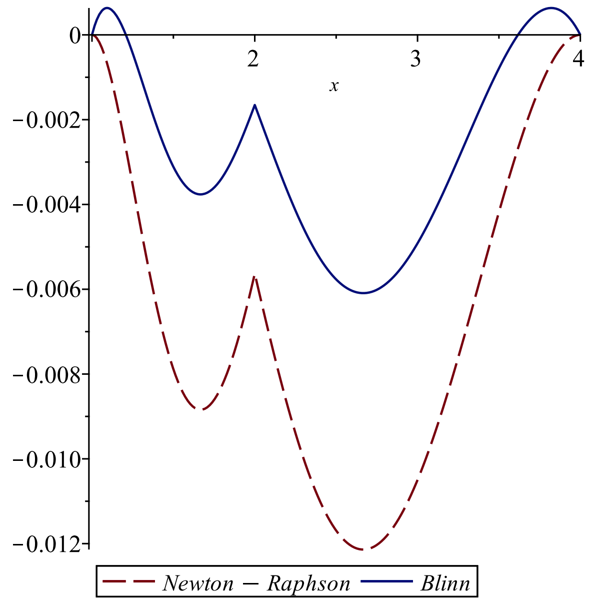

Fig. 13. Relative difference between the approximation obtained by getting y_0 from (56) and then performing (59) or (60) and the exact $1/\sqrt{x}$.

A careful error analysis of this class of methods for approximating the inverse square root, along with a derivation of the optimal values of the "magic constant" $\mathcal{M}$ in (56) is given by Moroz et al. [14]. In particular, the best choice for the constant slightly depends on whether we directly use the estimate (56) as an approximation to $1/\sqrt{x}$ or we perform one or two Newton-Raphson iterations to get a better approximation. In a recent paper [69] based on a similar analysis, Walczyk et al. also suggest modified Newton-Raphson iterations that give a better result.

VII. Discussion

It is difficult to compare the methods presented in this paper, since they differ in terms of versatility (which functions can be implemented with the method?), accuracy and scalability (does the considered method still work if we need better accuracy?). However, let us summarize their main properties.

- for software implementations, if rather rough approximations suffice (typically, relative error of a few percents), bit-manipulations techniques are hard to beat. Typically, a logarithm or a square root are approximated with one addition and a shift. These approximations can also be used as "seed values" for obtaining better approximations using Newton-Raphson iterations. However, these methods are limited to a small set of functions (radix-2 exponentials and logarithms, small powers);
- for hardware implementations, shift-and-add algorithms have the great advantage of allowing fine tuning of the speed-vs-accuracy compromise. Obtaining n bits of accuracy with these methods requires to perform a number of shifts and additions proportional to n (and we need to store around n n-bit constants). These methods scale-up quite well (even if for large n—which is not the topic of this paper—they become less interesting that polynomial approximations), but they are interesting for moderate precision (for instance, De Dinechin et al. [54] find that when low-precision FPGA implementation of the $\arctan$ function, CORDIC is a good candidate). These methods cannot be used for all functions (they are all based on an algebraic property of the function being computed, such as $e^{x+y} = e^x e^y$), but the set of implementable functions is rather large: it includes exponentials, logarithms, trigonometric functions, square roots;
- table-based methods are in-between, they have been used in hardware and software implementations. They make it possible to get higher accuracies than bit-manipulation techniques and are faster than shift-and-add algorithms, but they don't scale up well: even if bipartite and multipartite methods are very helpful for reducing the table size, that size remains an exponential function of the number of input bits. They are very versatile (any continuous function can be tabulated);
- polynomial approximations have been used either in software and hardware. They are very versatile and scale-up well: any continuous function can be approximated within any desired accuracy by a polynomial. However, the necessary degree (for a given input interval and a given accuracy target) depends much on the function. Depending on the function, it may be possible to find approximations with simple coefficients.

VIII. Conclusion

We have reviewed the main methods that can be used for getting fast approximations to the elementary functions. No general advice can be given on which method is to be preferred: this depends much on the function being approximated the requested accuracy, the underlying technology, and possible constraints on table size.

References

[1] W. Liu, F. Lombardi, and M. Schulte, "A retrospective and prospective view of approximate computing [point of view]," *Proceedings of the IEEE*, vol. 108, no. 3, pp. 394–399, March 2020.

[2] IEEE, "IEEE 754-2019 standard for floating-point arithmetic," July 2019. [Online]. Available: https://ieeexplore.ieee.org/servlet/opac?punumber=8766227

[3] J.-M. Muller, *Elementary Functions, Algorithms and Implementation*, 3rd ed. Birkhäuser Boston, MA, 2016.

[4] J. E. Volder, "The CORDIC trigonometric computing technique," *IRE Transactions on Electronic Computers*, vol. EC-8, no. 3, pp. 330–334, Sep. 1959.

[5] ——, "The birth of CORDIC," *Journal of VLSI Signal Processing Systems*, vol. 25, no. 2, pp. 101–105, Jun. 2000.

[6] E. Remez, "Sur un procédé convergent d'approximations successives pour déterminer les polynômes d'approximation," *C.R. Académie des Sciences, Paris*, vol. 198, pp. 2063–2065, 1934, in French.

[7] N. Brisebarre and S. Chevillard, "Efficient polynomial L^∞ approximations," in *18th IEEE Symposium on Computer Arithmetic (ARITH-18)*, Montpellier, France, 2007, pp. 169–176.

[8] P. T. P. Tang, "Table-driven implementation of the exponential function in IEEE floating-point arithmetic," *ACM Transactions on Mathematical Software*, vol. 15, no. 2, pp. 144–157, Jun. 1989.

[9] ——, "Table-driven implementation of the logarithm function in IEEE floating-point arithmetic," *ACM Transactions on Mathematical Software*, vol. 16, no. 4, pp. 378–400, Dec. 1990.

[10] S. Gal, "Computing elementary functions: a new approach for achieving high accuracy and good performance," in *Accurate Scientific Computations. Lecture Notes in Computer Science*, vol. 235. Springer-Verlag, Berlin, 1986, pp. 1–16.

[11] W. F. Wong and E. Goto, "Fast hardware-based algorithms for elementary function computations using rectangular multipliers," *IEEE Transactions on Computers*, vol. 43, no. 3, pp. 278–294, Mar. 1994.

[12] N. N. Schraudolph, "A fast, compact approximation of the exponential function," *Neural Computation*, vol. 11, no. 4, pp. 853–862, 1999.

[13] J. F. Blinn, *Notation, Notation, Notation, Jim Blinn's Corner*, ser. The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling. Morgan Kaufmann, 2003.

[14] L. Moroz, C. Walczyk, A. Hrynchyshyn, V. Holimath, and J. Cieśliński, "Fast calculation of inverse square root with the use of magic constant – analytical approach," *Applied Mathematics and Computation*, vol. 316, pp. 245 – 255, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0096300317305763

[15] J. N. Mitchell, "Computer multiplication and division using binary logarithms," *IRE Transactions on Electronic Computers*, vol. EC-11, no. 4, pp. 512–517, Aug 1962.

[16] K. H. Abed and R. E. Siferd, "CMOS VLSI implementation of a low-power logarithmic converter," *IEEE Transactions on Computers*, vol. 52, no. 11, pp. 1421–1433, Nov 2003.

[17] T. Juang, S. Chen, and H. Cheng, "A lower error and rom-free logarithmic converter for digital signal processing applications," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 56, no. 12, pp. 931–935, Dec 2009.

[18] D. Marino, "New algorithms for the approximate evaluation in hardware of binary logarithms and elementary functions," *IEEE Transactions on Computers*, vol. C-21, pp. 1416–1421, 1972, reprinted in E. E. Swartzlander, *Computer Arithmetic*, Vol. 1, IEEE Computer Society Press, Los Alamitos, CA, 1990.

[19] V. Mahalingam and N. Ranganathan, "Improving accuracy in Mitchell's logarithmic multiplication using operand decomposition," *IEEE Transactions on Computers*, vol. 55, no. 12, pp. 1523–1535, Dec 2006.

[20] M. S. Kim, A. A. D. Barrio, L. T. Oliveira, R. Hermida, and N. Bagherzadeh, "Efficient Mitchell's approximate log multipliers for convolutional neural networks," *IEEE Transactions on Computers*, vol. 68, no. 5, pp. 660–675, May 2019.

[21] B. Yuan, "Efficient hardware architecture of softmax layer in deep neural network," in *2016 29th IEEE International System-on-Chip Conference (SOCC)*, Sep. 2016, pp. 323–326.

[22] W. Liu, J. Xu, D. Wang, C. Wang, P. Montuschi, and F. Lombardi, "Design and evaluation of approximate logarithmic multipliers for low power error-tolerant applications," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 65, no. 9, pp. 2856–2868, Sep. 2018.

[23] A. Ukil, V. H. Shah, and B. Deck, "Fast computation of arctangent functions for embedded applications: A comparative analysis," in *2011 IEEE International Symposium on Industrial Electronics*, June 2011, pp. 1206–1211.

[24] M. Abramowitz and I. A. Stegun, *Handbook of mathematical functions with formulas, graphs and mathematical tables*, ser. Applied Math. Series 55. National Bureau of Standards, Washington, D.C., 1964.

[25] R. Lyons, "Another contender in the arctangent race," *IEEE Signal Processing Magazine*, vol. 21, no. 1, pp. 109–110, Jan 2004.

[26] X. Girones, C. Julia, and D. Puig, "Full quadrant approximations for the arctangent function [tips and tricks]," *IEEE Signal Processing Magazine*, vol. 30, no. 1, pp. 130–135, Jan 2013.

[27] S. Rajan, Sichun Wang, R. Inkol, and A. Joyal, "Efficient approximations for the arctangent function," *IEEE Signal Processing Magazine*, vol. 23, no. 3, pp. 108–111, May 2006.

[28] S. Winitzki, "Uniform approximations for transcendental functions," in *Computational Science and Its Applications — ICCSA 2003*, V. Kumar, M. L. Gavrilova, C. J. K. Tan, and P. L'Ecuyer, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 780–789.

[29] P. Markstein, "Accelerating sine and cosine evaluation with compiler assistance," in *16th IEEE Symposium on Computer Arithmetic (ARITH16)*. IEEE Computer Society Press, Los Alamitos, CA, Jun. 2003, pp. 137–140.

[30] O. Niemitalo, "DSP trick: Simultaneous parabolic approximation of sin and cos," 2001, available at https://dspguru.com/dsp/tricks/parabolic-approximation-of-sin-and-cos/.

[31] D. Roegel, "A reconstruction of the tables of Briggs' Arithmetica logarithmica (1624)," Inria, France, Tech. Rep. inria-00543939, 2010, available at https://hal.inria.fr/inria-00543939.

[32] M. D. Ercegovac and T. Lang, *Digital Arithmetic*. Morgan Kaufmann Publishers, San Francisco, CA, 2004.

[33] P. Meher, J. Valls, T.-B. Juang, K. Sridharan, and K. Maharatna, "50 years of CORDIC: Algorithms, architectures, and applications," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 56, no. 9, pp. 1893–1907, Sept 2009.

[34] N. Takagi, "Studies on hardware algorithms for arithmetic operations with a redundant binary representation," Ph.D. dissertation, Dept. Info. Sci., Kyoto University, Japan, 1987.

[35] M. Mikaitis, D. R. Lester, D. Shang, S. Furber, G. Liu, J. Garside, S. Scholze, S. Höppner, and A. Dixius, "Approximate fixed-point elementary function accelerator for the spinnaker-2 neuromorphic chip," in *2018 IEEE 25th Symposium on Computer Arithmetic (ARITH)*, June 2018, pp. 37–44.

[36] H. Ahmed, "Efficient elementary function generation with multipliers," in *9th IEEE Symposium on Computer Arithmetic*, 1989, pp. 52–59.

[37] P. W. Baker, "Suggestion for a fast binary sine/cosine generator," *IEEE Transactions on Computers*, vol. C-25, no. 11, Nov. 1976.

[38] T.-B. Juang, S.-F. Hsiao, and M.-Y. Tsai, "Para-CORDIC: parallel CORDIC rotation algorithm," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 51, no. 8, pp. 1515–1524, Aug 2004.

[39] L. Chen, F. Lombardi, Jie Han, and Weiqiang Liu, "A fully parallel approximate CORDIC design," in *2016 IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH)*, July 2016, pp. 197–202.

[40] L. Chen, J. Han, W. Liu, and F. Lombardi, "Algorithm and design of a fully parallel approximate coordinate rotation digital computer (cordic)," *IEEE Transactions on Multi-Scale Computing Systems*, vol. 3, no. 3, pp. 139–151, July 2017.

[41] M. Ercegovac, "Radix-16 evaluation of certain elementary functions," *IEEE Transactions on Computers*, vol. C-22, no. 6, pp. 561–566, 1973.

[42] L. Trefethen, *Approximation Theory and Approximation Practice*. SIAM, 2013.

[43] N. Brisebarre, J.-M. Muller, and A. Tisserand, "Sparse-coefficient polynomial approximations for hardware implementations," in *38th IEEE Conference on Signals, Systems and Computers*. IEEE, Nov. 2004.

[44] ——, "Computing machine-efficient polynomial approximations," *ACM Transactions on Mathematical Software*, vol. 32, no. 2, pp. 236–256, Jun. 2006.

[45] S. Chevillard, M. Joldeş, and C. Lauter, "Sollya: An environment for the development of numerical codes," in *3rd International Congress on Mathematical Software (ICMS)*, ser. Lecture Notes in Computer Science, vol. 6327. Springer, Heidelberg, Germany, September 2010, pp. 28–31.

[46] S. Chevillard, J. Harrison, M. Joldeş, and C. Lauter, "Efficient and accurate computation of upper bounds of approximation errors," *Theoretical Computer Science*, vol. 412, no. 16, pp. 1523–1543, 2011.

[47] M. Daumas and G. Melquiond, "Certification of bounds on expressions involving rounded operators," *ACM Transactions on Mathematical Software*, vol. 37, no. 1, pp. 2:1–2:20, Jan. 2010.

[48] F. de Dinechin, C. Lauter, and G. Melquiond, "Assisted verification of elementary functions using Gappa," in *Proceedings of the 2006 ACM Symposium on Applied Computing*, 2006, pp. 1318–1322.

[49] ——, "Certifying the floating-point implementation of an elementary function using Gappa," *IEEE Transactions on Computers*, vol. 60, no. 2, pp. 242–253, 2011.

[50] N. Brunie, F. de Dinechin, O. Kupriianova, and C. Lauter, "Code generators for mathematical functions," in *22nd IEEE Symposium on Computer Arithmetic*, Jun. 2015, pp. 66–73.

[51] O. Kupriianova and C. Lauter, "Metalibm: A mathematical functions code generator," in *Mathematical Software – ICMS 2014*, H. Hong and C. Yap, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 713–717.

[52] F. de Dinechin and B. Pasca, "Designing custom arithmetic data paths with FloPoCo," *IEEE Design & Test of Computers*, vol. 28, no. 4, pp. 18–27, Jul. 2011.

[53] F. de Dinechin, "Reflections on 10 years of FloPoCo," in *ARITH 2019 - 26th IEEE Symposium on Computer Arithmetic*. Kyoto, Japan: IEEE, Jun. 2019, pp. 1–3. [Online]. Available: https://hal.inria.fr/hal-02161527

[54] F. de Dinechin and M. Istoan, "Hardware implementations of fixed-point Atan2," in *22nd Symposium of Computer Arithmetic*. IEEE, Jun. 2015.

[55] D. Goldberg, "Fast approximate logarithms, part i: The basics," may 205, ebay Tech Blog "Applied math in engineering", available at https://tech.ebayinc.com/engineering/fast-approximate-logarithms-part-i-the-basics/.

[56] G. A. Baker, *Essentials of Padé Approximants*. Academic Press, New York, NY, 1975.

[57] P. Kornerup and J.-M. Muller, "Choosing starting values for certain Newton–Raphson iterations," *Theoretical Computer Science*, vol. 351, no. 1, pp. 101–110, Feb. 2006.

[58] D. A. Sunderland, R. A. Strauch, S. W. Wharfield, H. T. Peterson, and C. R. Cole, "CMOS/SOS frequency synthesizer LSI circuit for spread spectrum communications," *IEEE Journal of Solid State Circuits*, vol. SC-19, no. 4, pp. 497–506, 1984.

[59] D. DasSarma and D. W. Matula, "Faithful bipartite ROM reciprocal tables," in *12th IEEE Symposium on Computer Arithmetic (ARITH-12)*, Jun. 1995, pp. 17–28.

[60] M. J. Schulte and J. Stine, "Symmetric bipartite tables for accurate function approximation," in *13th IEEE Symposium on Computer Arithmetic*, 1997.

[61] M. J. Schulte and J. E. Stine, "Accurate function evaluation by symmetric table lookup and addition," in *IEEE International Conference on Application-Specific Systems, Architectures and Processors (Zurich, Switzerland)*, 1997, pp. 144–153.

[62] ——, "Approximating elementary functions with symmetric bipartite tables," *IEEE Transactions on Computers*, vol. 48, no. 8, pp. 842–847, Aug. 1999.

[63] H. Hassler and N. Takagi, "Function evaluation by table look-up and addition," in *12th IEEE Symposium on Computer Arithmetic*, Bath, UK, Jul. 1995.

[64] F. de Dinechin and A. Tisserand, "Multipartite table methods," *IEEE Transactions on Computers*, vol. 54, no. 3, pp. 319–330, Mar. 2005.

[65] S. Hsiao, C. Wen, Y. Chen, and K. Huang, "Hierarchical multipartite function evaluation," *IEEE Transactions on Computers*, vol. 66, no. 1, pp. 89–99, Jan 2017.

[66] Wikipedia contributors, "Fast inverse square root — Wikipedia, the free encyclopedia," 2020, [Online; accessed 11-March-2020]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Fast_inverse_square_root&oldid=940101226

[67] M. A. Cornea-Hasegan, R. A. Golliver, and P. Markstein, “Correctness proofs outline for Newton–Raphson based floating-point divide and square root algorithms,” in *14th IEEE Symposium on Computer Arithmetic (ARITH-14)*, Apr. 1999, pp. 96–105.

[68] J.-M. Muller, N. Brunie, F. de Dinechin, C.-P. Jeannerod, M. Joldes, V. Lefèvre, G. Melquiond, N. Revol, and S. Torres, *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston, 2018, ACM G.1.0; G.1.2; G.4; B.2.0; B.2.4; F.2.1., ISBN 978-3-319-76525-9.

[69] C. J. Walczyk, L. V. Moroz, and J. L. Cieśliński, “A modification of the fast inverse square root algorithm,” *Computation*, vol. 7, no. 3, 2019.