The CORE-MATH Project

Alexei Sibidanov
University of Victoria
British Columbia, Canada V8W 3P6
sibid@uvic.ca

Paul Zimmermann
Université de Lorraine
CNRS, Inria, LORIA
F-54000 Nancy, France
paul.zimmermann@inria.fr

Stéphane Glondu
Université de Lorraine
CNRS, Inria, LORIA
F-54000 Nancy, France
stephane.glondu@inria.fr

***Abstract*—The CORE-MATH project aims at providing open-source mathematical functions with correct rounding that can be integrated into current mathematical libraries. This article demonstrates the CORE-MATH methodology on two functions: the binary32 power function (`powf`) and the binary64 cube root function (`cbrt`). CORE-MATH already provides a full set of correctly rounded C99 functions for single precision (binary32). These functions provide similar or in some cases up to threefold speedups with respect to the GNU libc mathematic library, which is not correctly rounded. This work offers a prospect of the mandatory requirement of correct rounding for mathematical functions in the next revision of the IEEE-754 standard.**

***Index Terms*—IEEE 754, floating-point, correct rounding, efficiency.**

I. INTRODUCTION

Given a mathematical function f, a floating-point input x, the *correct rounding* of $f(x)$ is the floating-point y closest to $f(x)$ according to the given rounding mode. The IEEE 754-2019 standard [10] recommends correct rounding for mathematical functions, but does not require it. As a consequence, current mathematical libraries (GNU libc, Intel Math Library, AMD Libm, Newlib, OpenLibm, Musl, Apple Libm, LLVM-libc, CUDA libm, ROCm) may return different results for the same input, and yield in some cases huge errors in terms of units-in-last-place [11]. This prevents bit-to-bit reproductibility, which is more and more important in scientific applications [4]. Also, the next version of the C standard will have reserved names, say `cr_sin`, for correctly rounded mathematical functions [3]; efficient implementations will be mandatory for these functions.

A. Previous work

Previous work on correctly rounded mathematical functions was published through three libraries: MathLib, CRlibm, and RLibm. MathLib (also called libultim) is a library designed by Abraham Ziv, Moshe Olshansky, Ealan Henis and Anna Reitman from IBM [16], [19]. It implements the following double-precision functions: `asin`, `acos`, `atan`, `atan2`, `log`, `log2`, `exp`, `exp2`, `sin`, `cos`, `tan`, `cot`, `pow`. To our best knowledge, MathLib only provides correct rounding for rounding to nearest (with ties to even). Some of the MathLib routines were incorporated into GNU libc, but removed progressively after GNU libc 2.27, because the "accurate path" produced huge slowdowns in some rare cases. For example, the GNU libc 2.27 `benchtests` mechanism reports 440,000 cycles for the binary64 pow function in the "768-bit" path.

Another correctly rounded library is CR-LIBM [6], also targeting double precision. CR-LIBM provides the following functions: `exp`, `expm1`, `log`, `log1p`, `log2`, `log10`, `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `sinh`, `cosh`, `sinpi`, `cospi`, `tanpi`, `atanpi`, and an incomplete `pow` function. CR-LIBM can deal with all four IEEE-754 rounding modes, by providing four routines for each mathematical function, for example `exp_rn` for rounding to nearest, `exp_ru` for rounding towards $+\infty$, `exp_rd` for rounding towards $-\infty$, and `exp_rz` for rounding towards zero. These routines assume that the rounding precision is set to double precision (on some processors it is double-extended by default), and the current rounding mode is to nearest-even: the user has to call the `crlibm_init` routine before any floating-point computation. This differs from the philosophy of the C standard, with only one routine, say `cr_exp` or simply `exp`, with the rounding direction controlled by the current rounding mode (`fesetround` in the C language), and which should return correct rounding whatever the rounding precision (to double or double-extended). CR-LIBM uses modern instructions like the fused-multiply-add (FMA). Together with the knowledge of hardest-to-round cases, the accuracy of CR-LIBM's accurate path is better tuned, and its efficiency is better than MathLib. For example, for the `exp` function, [6] reports a ratio of about 6500 between the maximal and the average time for MathLib, against only 6.6 for CR-LIBM (using triple-double arithmetic). CR-LIBM provides different levels of proof of correctness: partial, paper proof, or formal proof. Alas, CR-LIBM was never integrated into widely used mathematical libraries.

The RLibm library [14] uses a different approach. From the knowledge of hard-to-round cases, it uses a linear programming approach to find polynomials that yield correct rounding in the given range. This new approach works well for univariate binary32 functions, but was not yet extended to bivariate functions or larger precisions. RLibm-32 provides `cosh`, `cospi`, `exp`, `exp10`, `exp2`, `log`, `log10`, `log2`, `sinh` and `sinpi`, but only for round to nearest.

B. Correct rounding methodology

The classical method to get correct rounding is the following:

1) a *fast path* routine computes an approximation y of $f(x)$ with a small error bound ε;
2) a *rounding test* checks whether the range $[y-\varepsilon, y+\varepsilon]$ crosses a *rounding boundary* (floating-point numbers for directed roundings, or the middle of two floating-point numbers for rounding to nearest). If this is not the case, rounding y yields the correctly rounded value;
3) otherwise, one calls the *accurate path*, which should always return the correctly rounded value.

The *fast path* should be fast as its name suggests, with a small probability of crossing a rounding boundary, say 0.1%. The *accurate path* might be up to 10 times slower, this will not impact much the average time. For the fast and accurate path, CORE-MATH relies on classical algorithms from the literature [15], which are specific to each function: argument reduction and reconstruction, table lookup, polynomial evaluation. The CORE-MATH know-how lies in particular in the efficient implementation of these algorithms, and in the optimal tuning of the accurate path, which relies on the knowledge of the hard-to-round (HR) cases.

C. Rounding mode and special values

The CORE-MATH routines are correctly rounded for any rounding mode: to nearest, towards zero, towards $\pm\infty$. To round `exp` towards zero, set the rounding mode towards zero using the `fesetenv` function, and call the CORE-MATH `cr_exp` function. The result should be correctly rounded whatever the rounding precision of the floating-point unit, the optimization level of the compiler, or the use of FMA.

Some CORE-MATH routines do not deal with special values (NaN, +Inf, -Inf), since the treatment of special values is library-specific: we let the developers of mathematical libraries deal with them when they integrate the CORE-MATH code. Similarly, they might raise spurious underflow, overflow, or inexact exceptions, since some libraries might not care about these exceptions.

D. Plan of the article

In Sections II and III, we focus on two functions to describe the CORE-MATH methodology: the binary32 power function (`powf`) and the binary64 cube root function (`cbrt`). For both functions, we detail the search for exact, midpoint and hard-to-round cases. Then Section IV explains how we validate the CORE-MATH routines and measure their efficiency.

In all urls given in that article, replace CORE-MATH by https://gitlab.inria.fr/core-math/core-math/-/blob/master.

II. THE BINARY32 POWER FUNCTION

This section explains how to get correct rounding for the binary32 power function x^y: one first has to compute exact and midpoint cases (§II-A), hard-to-round cases (§II-B), then the algorithm is detailed in §II-C.

A. Exact and midpoint cases

Exact cases for the binary32 power function are inputs x, y such that x^y is exactly representable in binary32. Midpoint cases are inputs such that x^y is exactly representable on 25 bits, but not on 24 bits, and lies in the binary32 range. We used the method of Lauter and Lefèvre [12] to generate exact and midpoint cases. We found a total of 1,071,899 exact and midpoint cases (842,073 exact and 229,826 midpoint). Since it would take 21Mb to store all these inputs, we only provide the program to count or generate them[1].

B. Search for hard-to-round cases

So far no clever algorithm exists for the search for hard-to-round cases of bivariate functions. Stehlé mentions an extension of the SLZ algorithm in his PhD thesis [17, Chapter III-1], but to our knowledge it was never implemented. Some recent progress was made by Brisebarre and Hanrot for the search for hard-to-round cases, which might apply to bivariate functions [2].

The binary32 power function has in principle about 2^{64} different pairs x, y of inputs. However, if one discards inputs yielding an overflow or an underflow, or a result rounding to 1 to nearest (when y is tiny), it remains about 2^{56} "regular" cases, which is still too large for a "naive exhaustive search" (explained below). For a positive integer m, we say that x^y is a m-HR case if there are at least m identical bits after the round bit in the (infinite precision) binary representation of x^y. We use Algorithm 1, where we assume for simplicity that $\mathrm{ulp}(x^y) = 2$, thus the round bit has weight 1.

Theorem 1: For given y and m, Algorithm 1 outputs all binary32 numbers x such that x^y is a m-HR case.

Proof: It suffices to verify that no m-HR case is missed in each interval $[x_0, x_1]$ in line 4. Assume $x = x_0 + i\mathrm{ulp}(x_0)$ is such a m-HR case, for $0 \le i < n$, with the notations of the algorithm. Then by definition, one has $|\mathrm{frac}(x^y)| < 2^{-m}$. From $|x^y - (a+bi+ci^2)| < di^3$, it follows $|\mathrm{frac}(a+bi+ci^2)| < 2^{-m} + di^3$. Multiplying by 2^{64}, and since $|\alpha - 2^{64}\mathrm{frac}(a)| < 1$, and similarly for β, b and γ, c, it yields $|\alpha + \beta i + \gamma i^2 \bmod 2^{64}| < 2^{64-m} + 2^{64}di^3 + 1 + i + i^2$. By the definitions of ε_a, ε_b, ε_c and ε_d, it follows $|\alpha + \beta i + \gamma i^2 \bmod 2^{64}| < p(i)$, where $p(i) := \varepsilon_a + i\varepsilon_b + i^2\varepsilon_c + i^3\varepsilon_d$. Thus one has $\alpha + \beta i + \gamma i^2 + p(i) \bmod 2^{64} < 2p(i)$. Lines 8 and 9 use the "table of differences method" to evaluate the polynomials $\alpha + \beta i + \gamma i^2 \bmod 2^{64}$ and $p(i)$: one can check by induction that at step i, one has $\alpha' = \alpha + \beta i + \gamma i^2 \bmod 2^{64}$, $\beta' = \beta + (2i+1)\gamma \bmod 2^{64}$, and $\gamma' = 2\gamma$. Similarly, $\varepsilon'_a = p(i), \varepsilon'_b = \varepsilon_b + (2i+1)\varepsilon_c + (3i^2+3i+1)\varepsilon_d$, $\varepsilon'_c = 2\varepsilon_c + 6(i+1)\varepsilon_d$. ■

Remark: in line 11 of the algorithm, $2\varepsilon'_a$ might exceed 2^{64}, in which case the test will always be true, and we will always perform an expensive check. In such a case, we restart from line 4 with $x_0 = x_0 + i\mathrm{ulp}(x_0)$ where i is the current index.

Algorithm 1 uses only a few cycles per x-value, plus some initialization time to compute the α', β', γ' values and the corresponding error bounds $\varepsilon'_a, \varepsilon'_b, \varepsilon'_c$ and ε'_d. Indeed, the

[1]CORE-MATH/src/binary32/pow/exact.c

Algorithm 1 Algorithm worst_powf

Input: a binary32 value y and a positive integer $m < 64$

Output: all hard-to-round cases of `powf(x,y)` with at least m identical bits after the round bit

compute the bounds $x_{\min}$ and $x_{\max}$ such that regular cases correspond to $x_{\min} \le x \le x_{\max}$
split $[x_{\min}, x_{\max}]$ into ranges included in a single binade
split further the inputs so that x^y lies in the same binade
on each sub-range $[x_0, x_1]$, compute n such that $x_1 = x_0 + n\mathrm{ulp}(x_0)$
compute a degree-2 Taylor approximation with error term $|x^y - (a + bi + ci^2)| < di^3$, for $x = x_0 + i\mathrm{ulp}(x_0)$
$\alpha \leftarrow \lfloor 2^{64}\mathrm{frac}(a)\rceil$, $\beta \leftarrow \lfloor 2^{64}\mathrm{frac}(b)\rceil$, $\gamma \leftarrow \lfloor 2^{64}\mathrm{frac}(c)\rceil$
$\varepsilon_a \leftarrow 1 + 2^{64-m}$, $\varepsilon_b \leftarrow 1$, $\varepsilon_c \leftarrow 1$, $\varepsilon_d \leftarrow \lceil 2^{64} d\rceil$
$\alpha' \leftarrow \alpha$, $\beta' \leftarrow \beta + \gamma$, $\gamma' \leftarrow 2\gamma$
$\varepsilon'_a \leftarrow \varepsilon_a$, $\varepsilon'_b \leftarrow \varepsilon_b + \varepsilon_c + \varepsilon_d$, $\varepsilon'_c \leftarrow 2\varepsilon_c + 6\varepsilon_d$, $\varepsilon'_d \leftarrow 6\varepsilon_d$
for i from 0 to $n - 1$ **do**
    if $\alpha' + \varepsilon'_a \bmod 2^{64} < 2\varepsilon'_a$, check whether x^y is a m-HR case, and if so output x, y
    $\alpha' \leftarrow \alpha' + \beta' \bmod 2^{64}$
    $\beta' \leftarrow \beta' + \gamma' \bmod 2^{64}$
    $\varepsilon'_a \leftarrow \varepsilon'_a + \varepsilon'_b$
    $\varepsilon'_b \leftarrow \varepsilon'_b + \varepsilon'_c$
    $\varepsilon'_c \leftarrow \varepsilon'_c + \varepsilon'_d$

critical loop uses only 64-bit additions, or additions modulo 2^{64}, which take one cycle (or less) on modern computers (this explains the limitation $m < 64$). On a 3.3Ghz Intel Core i5-4590, our implementation takes 12.1 seconds to check all x-values for exponent `0x1.921fb6p-1`, which corresponds to 18.6 cycles on average per x-value. Since for this exponent all $\approx 2^{31}$ positive x-values yield a regular x^y value, and one has $\approx 2^{56}$ regular inputs, this would scale to 13 core-years for the full hard-to-round check.

We can compare to the BaCSeL tool (branch pow, revision 4bd8cce), which was modified to deal with the power function (for a fixed exponent y): on the same computer it takes 0.8s to check the binade $1/2 \le x < 1$ for the same exponent y as above, with BaCSeL parameters $d = \alpha = 2$, which extrapolates to 200 seconds for the $\approx 2^{31}$ positive x-values, and to 218 core-years for the full search. Algorithm 1 is thus faster.

1) Naive exhaustive search: The "naive exhaustive search" consists of using GNU MPFR [8] for each input pair x, y: compute x^y with $24+m$ bits, correctly rounded to nearest, then round this value z down to a 25-bit number t (still to nearest), and if $z = t$, x^y is a m-HR case. Indeed, we have $|x^y - z| < \frac{1}{2}\mathrm{ulp}(z)$ thus since $\mathrm{ulp}(z) = 2^{-(m-1)}\mathrm{ulp}(t)$, if $z = t$ we deduce $|x^y - t| < 2^{-m}\mathrm{ulp}(t)$, which means that x^y has at least m identical bits after the round bit. Despite the efficiency of MPFR, the naive exhaustive search is slow: checking the full binade $1/2 \le x < 1$ for $y =$ `0x1.921fb6p-1` takes about 37 seconds on an Intel Core i5-4590, which would extrapolate to about 10000 core-years for the full search of $\approx 2^{56}$ inputs.

2) Using degree 1: Algorithm 1, which uses a degree-2 Taylor expansion with explicit error term, can be extended to any degree-d expansion. We tried degree 1, which uses a linear expansion, but the error term was quite large, so we had to recompute the Taylor expansion too often, and it was less efficient than degree 2.

3) Using the inverse function: When y is small in absolute value, the function x^y is contracting. Instead of checking each value of x, it is faster to check the inverse function $z^{1/y}$, since there are much fewer values of z to check: they correspond to the image of the binary32 range by $x \to x^y$. However, the wanted number m' of identical bits after the round bit for $z^{1/y}$ is smaller than m, which makes Algorithm 1 slower, because ε'_a and thus ε_a is larger. Since one also wants hard-to-round cases for rounding to nearest, one has to consider 25-bit values z, which doubles the number of values to check in the z-range.

4) Results: In the end we used the inverse function—with Algorithm 1 and degree 2—for $|y| < 2^{-9}$, a naive search for $|y| > 2^{14}$, where only very few x-values give x^y in the binary32 range, and Algorithm 1 with degree 2 everywhere else. The program we used is available from the CORE-MATH page[2] as well as the HR cases[3]. We found 129,173 x, y pairs giving at least 44 identical bits after the round bit, among which x^y rounds to nearest to 1 for 2952 values. The worst cases for $y > 0$ have 57 identical bits after the round bit, whereas the worst case for $y < 0$ has 66 identical bits after the round bit (Table I).

`0x1.762d7ep+104,0x1.df50fep-10`	57
`0x1.f5ec58p+121,0x1.7857e4p-12`	57
`0x1.a2a5d8p+90,0x1.16a37ap-20`	57
`0x1.1f49dap-105,0x1.4966fep-24`	57
`0x1.c8b072p-2,0x1.1be8f6p-3`	57
`0x1.00001p+0,-0x1.00000ap-2`	61
`0x1.ffffdp-1,-0x1.ffffe2p-3`	60
`0x1.ffffep-1,-0x1.ffffecp-3`	62
`0x1.fffffp-1,-0x1.fffff6p-3`	66
`0x1.46ee2p+67,-0x1.acbb3ap-7`	61

TABLE I
ALL m-HR CASES FOR BINARY32 x^y WITH $m \ge 57$ FOR $y > 0$ (TOP), AND $m \ge 60$ FOR $y < 0$ (BOTTOM).

Remark: Algorithm 1 applies to other bivariate functions, and also to univariate functions, since y is fixed in the algorithm. When the number of possible inputs x is not too large, as for binary32, it might be faster than other approaches like Lefèvre's algorithm or SLZ [18].

Due to lack of space, we cannot detail the search for hard-to-round cases for the two other C99 binary32 bivariate functions, namely `hypot` and `atan2`. For `hypot`, we first test on Pythagorean triples $x^2 + y^2 = z^2$, with x, y, z integers, $2^{23} \le y < 2^{24}$, $2^{23+k} \le x < 2^{24+k}$, and z exactly representable on 25 bits. The hard-to-round cases correspond to "almost Pythagorean triples" $x^2 + y^2 = z^2 \pm 1$ and the

[2]CORE-MATH/src/binary32/pow/worst.c
[3]CORE-MATH/src/binary32/pow/powf.wc

same bounds. In both cases, only a few values of the exponent difference k have to be considered.

For the atan2 function, we used the following algorithm. For each 25-bit floating-point value z lying in the binary32 range, we look for two binary32 values x, y such that $\operatorname{atan}(y/x)$ is very close to z. This means that y/x is very close to $\tan z$. Compute a continued fraction decomposition of $\tan z$, and take the last convergent that is exactly representable as y/x for two binary32 values x and y. This is a hard-to-round case.

C. Correct rounding

The binary32 power function x^y is computed using the well known relation:

$$x^y = 2^{y \log_2 x}.$$

In the fast path, the approximation of the binary logarithm exploits the floating point format $x = 2^{e_x} \cdot 1.m_x$ as

$$\log_2 x = e_x + \log_2 1.m_x,$$

where e_x is the exponent and $x' = 1.m_x$ is the 24-bit mantissa of x in the binary format. Since the polynomal approximation of the logarithm function converges very slowly in the $[1, 2]$ range, it is reduced into 32 equal regions $[1+(i-0.5)/32, 1+(i+0.5)/32]$, where $i \in [0, 31]$ is the region number. The reduced variable in each region is

$$z = x' \times r_i - 1,$$

where r_i is the reciprocal of $1 + i/32$ rounded in such a way that z is exact in the binary64 format.

The binary logarithm $\log_2(1 + z)$ is approximated in the range $|z| \leq 1/64$ by a degree-8 polynomial with relative error smaller than 3×10^{-18}. This is about two orders of magnitude better than what the binary64 format provides and thus only the rounding errors of polynomial evaluation dominate in the final logarithm error. In order to fully exploit this precision, the logarithms of r_i are tabulated with 61-bit precision and stored as two numbers in the binary64 format.

The polynomial approximation of the binary exponential 2^t behaves well in the $[0, 1]$ range but still requires a significant number of terms to reach the full binary64 format precision, which negatively affects the function performance. Thus again the $[0, 1]$ range is reduced to $[0, 1/16]$ and within this range 2^t is approximated by a degree-7 polynomial with $\sim 7 \times 10^{-18}$ absolute error.

The argument of the binary exponential, the product $y \log_2 x$, is evaluated and reduced to the $[0, 1/16]$ range with as many significant bits as possible.

After all intermediate calculations, the final result is represented in the binary64 format and has to be rounded in the binary32 format. The last 28 bits of the binary64 result are extracted and tested against an empirically found largest error and if the result lies within this error the accurate path of the binary32 power function is invoked. Otherwise the binary64 result is rounded to the binary32 format according to the current rounding mode and returned.

The worst case requires more than $24+1+66 = 91$ bits of internal precision as shown in Table I. This precision does fit in the double-double format and thus can be relatively efficiently calculated on ordinary hardware. To avoid large lookup tables the logarithm calculation is based on the hyperbolic arctangent:

$$\log_2 x = \frac{2}{\log 2} \tanh^{-1} \frac{x - x_m}{x + x_m} + (x_m - 1),$$

where $x \in [1, 2]$ and $x_m = 1$ for $x < \sqrt{2}$ and $x_m = 2$ otherwise. The arctangent argument is relatively small and its absolute value does not exceed 0.172. The arctangent is approximated by a degree-27 polynomial with relative error smaller than 2×10^{-29}. Since the arctangent is an odd function only 13 coefficients are needed. The binary exponential is approximated by a degree-17 polynomial in the $[-0.5, 0.5]$ range with maximal absolute error about 6×10^{-30}.

Then the result of the binary exponential in the double-double format is rounded into the binary32 format and returned.

The largest error in the fast path is found to be 44 ulp—in the binary64 format—and thus the accurate path should be invoked only in 1 case out of 3×10^6 function calls assuming random inputs. The accurate path takes on average 925 cycles per input on an Intel i5-4590.

III. The binary64 cube root function

A. The algorithm

The cube root $x = \sqrt[3]{a}$ is a real root of an algebraic equation:

$$f(x) = x^3 - a = 0. \tag{1}$$

There is a closed form solution for Eq. (1) but it requires the cube root function so other methods have to be employed, e.g., *Newton iteration*.

Let x_0 be an initial approximation of the cube root then

$$h_0 = f(x_0)/a = (x_0^3 - a)/a = (x_0^3 - a) r_a \tag{2}$$

is the relative error of Eq. (1) with respect to a, where $r_a = 1/a$ is the reciprocal of a. The next better approximation x_1 can be derived as

$$x_1 = x_0 - \frac{1}{3} x_0 h_0 \tag{3}$$

with about twice more significant figures than x_0. This procedure is repeated until it reaches enough accuracy.

The generalization of Newton's iteration to higher orders gives the following rule:

$$\begin{aligned} x_{i+1} &= x_i \left(1 - \frac{1}{3} h_i + \frac{2}{9} h_i^2 - \frac{14}{81} h_i^3 + \frac{35}{243} h_i^4 - \frac{91}{729} h_i^5 \right. \\ &+ \left. \frac{728}{6561} h_i^6 - \frac{1976}{19683} h_i^7 + \frac{5434}{59049} h_i^8 - \cdots \right), \end{aligned} \tag{4}$$

where each additional term reduces the error of the next approximation x_{i+1}. The coefficients of (4) are given by the series expansion of

$$\frac{1}{\sqrt[3]{1+h}} = \sum_{j=0}^{\infty} c_j h^j.$$

We can reduce the input argument a to the $[1, 8]$ range without any error, to get its cube root $x \in [1, 2]$ and then scale it accordingly to get the final result. The binary scaling is a cheap and exact operation in the binary64 format and particularly for the cube root without risk of overflow or underflow, due to the limited exponent range of the final result. The argument a can be further reduced to the $[1, 2]$ range if the corresponding cube root is scaled by $2^{1/3}$ or $2^{2/3}$; but this should be done *before* the rounding test, since the values of $2^{n/3}$ with $n = 1, 2$ are inexact. (Here by Newton's iteration of order k, we mean multiplying by k the accuracy.)

Algorithm 2 Fast path for binary64 cube root

Input: a binary64 value a

Output: correct rounding of $a^{1/3}$

1: scale a to $[1, 2)$

2: compute an initial 3rd-order minimax approximation x_0 in double precision with corresponding relative error $|h_0| < 0.3 \cdot 10^{-3}$ (Fig. 1, top-right)

3: perform a first Newton's iteration of order 3 to deduce an approximation x_1 in double precision with relative error $|h_1| < 6 \cdot 10^{-12}$ (Fig. 2, top-right)

4: perform a second Newton's iteration of order 2 to deduce an approximation x_2 in double-double with relative error $|h_2| < 1.32 \cdot 10^{-23}$ (Fig. 3)

The error h_0 of the minimax approximations of the cube root function by the second, third, fourth and fifth order polynomials is shown in Fig. 1. The error h_1 after the first step is shown in Fig. 2 for the third order Newton iteration. These calculations are performed in the binary64 format so the limited precision of the format is immediately seen even after the first high order iteration (see jitter at the bottom of Fig. 2). So the cube root calculated by this method cannot be correctly rounded due to intermediate rounding errors. A final refinement step using a compensated algorithm is needed.

The final step has to be as simple as possible so it is the second order Newton iteration (Eq. (2) and (3)) where intermediate values are represented as an unevaluated sum of two binary64 numbers so the internal precision should be about 100 bits which largely exceeds the target precision of the result of 53 bits in binary64.

The precision of the result before the final step should be good enough that after the refinement—which doubles the number of significant figures—an additional refinement has to be done only in very rare cases when the rounding test fails. Based on this consideration and performance tests we select the initial third order polynomial approximation and the third order Newton iteration step, see the top-right plots in Fig. 1 and 2.

After the refinement with the compensated algorithm, the cube root value is represented as an unevaluated sum $1 \le a + b \le 2$ of two binary64 numbers, where $|a| \ge |b|$. We then apply the Fast2Sum algorithm to compute $x_2^{\text{high}} \leftarrow \circ(a + b)$, $z \leftarrow \circ(x_2^{\text{high}} - a)$, $x_2^{\text{low}} \leftarrow \circ(b - z)$, where $\circ()$ denotes the current rounding mode.

Lemma 1: Whatever the rounding mode, we have $|x_2^{\text{low}}| < 2^{-52}$.

Proof: For rounding to nearest, this is a direct consequence of the Fast2Sum algorithm, since in that case we have $a + b = x_2^{\text{high}} + x_2^{\text{low}}$ exactly, and since x_2^{high} is the rounding to nearest of $a + b$, we have $x_2^{\text{low}} \le \frac{1}{2}\text{ulp}(x_2^{\text{high}})$. For directed rounding, according to [1, Theorem 3.1], x_2^{low} is a faithful rounding of the error in the FP addition $x_2^{\text{high}} = \circ(a+b)$. Let $\varepsilon = (a+b) - x_2^{\text{high}}$ be that error. Since $1 \le a+b \le 2$, and x_2^{high} is a directed rounding of $a+b$, we have $|\varepsilon| < \text{ulp}(1) = 2^{-52}$, thus a faithful rounding of that error cannot exceed 2^{-52}. Now if a faithful rounding of ε is $\pm 2^{-52}$, this implies $|\varepsilon| > 2^{-52} - 2^{-105}$, since $2^{-52} - 2^{-105}$ is representable in binary64. This in turn implies $\text{ulp}(b) < 2^{-105}$, otherwise $a + b$ would be an integer multiple of 2^{-105}, which would contradict $2^{-52} - 2^{-105} < |\varepsilon| < 2^{-52}$. But since $|b| < 2^{53}\text{ulp}(b)$ this yields $|b| < 2^{-52}$. In the Fast2Sum algorithm, when $x_2^{\text{high}} = \circ(a + b)$ is rounded towards a, we get $z = 0$ and $x_2^{\text{low}} = b$, thus $|x_2^{\text{low}}| < 2^{-52}$. If $x_2^{\text{high}} = \circ(a + b)$ is rounded away from a, say upwards if $b > 0$, then $z = 2^{-52}$, and since $x_2^{\text{low}} = \circ(b - z)$ is rounded in the *same* direction, we get $x_2^{\text{low}} > -z$. The same reasoning when rounding downwards for $b < 0$ also gives $|x_2^{\text{low}}| < 2^{-52}$. ∎

According to Lemma 1, we thus get an approximation $x_2^{\text{high}} + x_2^{\text{low}}$ of the cube root with $|x_2^{\text{low}}| < 2^{-52}$. The difference of this approximation with the exact cube root value is shown in Fig. 3. The maximal found error is $1.32 \times 10^{-23} < 2^{-76}$ and it occurs near the upper bound of the range. Thus to perform the rounding test in the round-to-nearest mode we need to check that $||x_2^{\text{low}}| - 2^{-53}| > 2^{-76}$ which means that x_2^{high} is a correctly rounded cube root value in binary64. In the directional modes we need to check both borders $|x_2^{\text{low}}| > 2^{-76}$ and $||x_2^{\text{low}}| - 2^{-52}| > 2^{-76}$ to be sure that x_2^{high} is correctly rounded. For safety the limit 2^{-76} is doubled to 2^{-75}. Considering this limit we can conclude that the probability to fail the rounding test is about $2^{-75}/2^{-52} \sim 10^{-7}$. (The check for exact cube is described below.)

If the rounding test fails, we perform an additional second order Newton iteration step starting from x_2^{high} which is known to be very close to the correctly rounded cube root, but might be 1 ulp off. The difference of x_3 (again $x_3 = x_3^{\text{high}} + x_3^{\text{low}}$) with the exact cube root value is shown in Fig. 4, 5, 6, 7 when the FPU is operating in various rounding modes. As it is seen the maximal visible error is about 2^{-102} on the limited number of arguments.

Unfortunately even the last refinement is not enough for the worst cases to provide the correctly rounded results, fortunately there are only a few such cases so we can test arguments and return already precomputed correctly rounded values.

1) Exact cases: In the round-to-nearest mode, the exact cases, when both a and x are exactly representable in the binary64 format, always pass the first rounding test and round to correct values. In the directed rounding modes, both rounding tests fail for exact cube roots and x_3^{high} can be 1 ulp off of the correctly rounded value. Such cases have to be detected.

There are 104032 distinct binary64 numbers x in the $[1, 2]$ range which might be exact solutions of Eq. (1), the one with largest numerator being $208063/2^{17}$, where $208063 = \lfloor 2^{53/3} \rfloor$. Thus, for exact cube roots, and rounding to nearest, at least 35 last bits of x_2^{high} have to be zero. For exact cube roots with a directed rounding mode, the last 35 bits of x_3^{high} should be all 0 or 1 (note that the first rounding test will always fail in that case). Testing the last bits of x_3^{high} alone to detect the exact cases is not enough since there are cases when the cube root of a has 35 zero bits but it is not an exact root. For example, when we have the exact relation $x = \sqrt[3]{a}$ in binary64 then $\sqrt[3]{a \pm 1 \text{ ulp}}$ would be also very close to x and thus would inherit the property of the last 35 bits. So we also need to test that the difference between $x = x_3^{\text{high}} + x_3^{\text{low}}$ and its rounded-to-nearest value in binary64 is smaller than the smallest difference between the cube root values of two consecutive binary64 values to detect exact cases.

Lemma 2: Let a be a binary64 number such that $1 \le a < 8$, and $a^{1/3}$ is not exactly representable in binary64. Let x be a binary64 number such that x^3 is also a binary64 number, and x is closest to $a^{1/3}$ (in case of tie, any value is ok). Then the distance from $a^{1/3}$ to x is at least $4.66 \cdot 10^{-17}$.

Proof: We first deal with the special cases where a is a power of 2. First a cannot be 1, since $1^{1/3}$ is exactly representable in binary64. If $a = 2$, we get $x = 165140/2^{17}$, and $|a^{1/3} - x| > 2 \cdot 10^{-6}$. If $a = 4$, we get $x = 104032/2^{16}$, and $|a^{1/3} - x| > 1 \cdot 10^{-6}$. Now assume that a is not a power of 2. Since x^3 is a binary64 number, and $x^3 \ne a$, we have $|x^3 - a| \ge \text{ulp}(a)$ (since a is not a power of 2). Write $a^{1/3} = x + \varepsilon$. Then $a = x^3 + 3x^2\varepsilon + 3x\varepsilon^2 + \varepsilon^3$. Thus $|3x^2\varepsilon + 3x\varepsilon^2 + \varepsilon^3| \ge \text{ulp}(a)$. In the case where $1 \le a < 2$, we have $\text{ulp}(a) = 2^{-52}$, and writing $\delta = |\varepsilon|$:

$$\delta \ge \frac{2^{-52}}{3x^2} - \frac{\delta^2}{x} - \frac{\delta^3}{3x^2},$$

where $x \le x_0 = 165141/2^{17}$. Thus

$$\delta \ge \frac{2^{-52}}{3x_0^2} - \delta^2 - \frac{\delta^3}{3}.$$

The corresponding equation has a single real root $\delta_0 \approx 4.66 \cdot 10^{-17}$, and for $\delta < \delta_0$, the above inequality does not hold. In the case where $2 \le a < 4$, we have $\text{ulp}(a) = 2^{-51}$, and writing $\delta = |\varepsilon|$:

$$\delta \ge \frac{2^{-51}}{3x^2} - \frac{\delta^2}{x} - \frac{\delta^3}{3x^2},$$

where $x \le x_1 = 104032/2^{16}$. Thus

$$\delta \ge \frac{2^{-51}}{3x_1^2} - \delta^2 - \frac{\delta^3}{3}.$$

The corresponding equation has a single real root $\delta_1 \approx 5.87 \cdot 10^{-17}$, and for $\delta < \delta_1$, the above inequality does not hold. In the case where $4 \le a < 8$, we have $\text{ulp}(a) = 2^{-50}$, and writing $\delta = |\varepsilon|$:

$$\delta \ge \frac{2^{-50}}{3x^2} - \frac{\delta^2}{x} - \frac{\delta^3}{3x^2},$$

where $x \le x_2 = 2$. Thus

$$\delta \ge \frac{2^{-51}}{3x_2^2} - \delta^2 - \frac{\delta^3}{3}.$$

The corresponding equation has a single real root $\delta_2 \approx 7.40 \cdot 10^{-17}$, and for $\delta < \delta_2$, the above inequality does not hold. In summary, for $|\varepsilon| \le \min(\delta_0, \delta_1, \delta_2)$, the inequality does not hold, thus we have $|\varepsilon| > \min(\delta_0, \delta_1, \delta_2) \ge 4.66 \cdot 10^{-17}$. ■

As a consequence of Lemma 2, if the distance from the approximation $x_2^{\text{high}} + x_2^{\text{low}}$—or $x_3^{\text{high}} + x_3^{\text{low}}$—to the nearest binary64 number x is less than $2^{-53}/3$, then $a^{1/3}$ is exactly representable. Indeed, since $|x_2^{\text{high}} + x_2^{\text{low}} - a^{1/3}| < 2^{-76}$, and $|x_2^{\text{high}} + x_2^{\text{low}} - x| < 2^{-53}/3$, this yields $|a^{1/3} - x| < 2^{-53}/3 + 2^{-76} < 4.66 \cdot 10^{-17}$.

To cover the exact cases we test that the last 35 bits of x are identical, then to cover the directed modes we round x to the nearest value independently of the FPU status register in the general purpose registers assuming the exact case. Then we subtract from the rounded value x_2 or x_3 depending on the rounding mode and check that the difference is less than $2^{-53}/3$ according to Lemma 2. In fact the threshold can be any value between 2^{-76} and $2^{-53}/3$ and in the function it is set to 2^{-60}. If the result passes the test we return the rounded value.

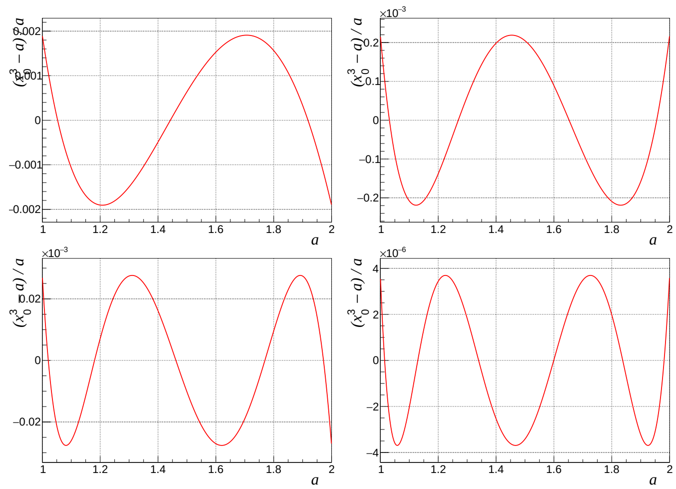

Fig. 1. The error h_0 of initial approximations. Top-left plot – second order, top-right – third order, bottom-left – fourth order, and bottom-right – the fifth order polynomial.

B. Validation on worst cases

We have computed the hard-to-round cases of the cube root function using the BaCSeL software tool [9]. Since $(8x)^{1/3} = 2x^{1/3}$, it suffices to search in three consecutive binades, for example $0.5 \le x < 4$. We kept hard-to-round cases with 44 identical bits after the round bit. We found 1496 such inputs,

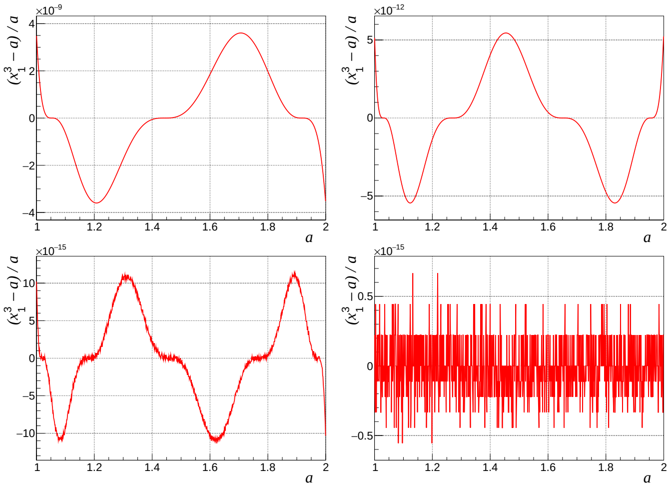

Fig. 2. The error h_1 after the first third order Newton iteration step for various initial approximations. The plot order is the same as in Fig. 1.

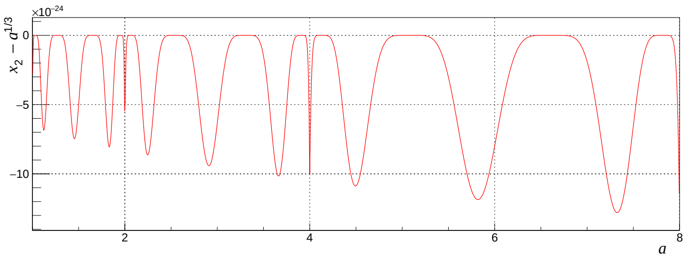

Fig. 3. The error of the cube root evaluation after the refinement step where the root x_2 is represented as an unevaluated sum $x_2^{\text{high}} + x_2^{\text{low}}$ of two binary64 numbers.

the worst one being `0x1.9b78223aa307cp+1` with 55 identical bits after the round bit. (In the file `testlibm-data` from [13], Lefèvre gives 138 inputs with at least 46 identical bits after the round bit.)

IV. Correctness and Efficiency

In CORE-MATH, each function is implemented in its own, standalone, file that can be directly integrated in a third-party codebase. Each file is in a dedicated directory, for example the implementation of the binary32 arc-cosine function is in CORE-MATH/src/binary32/acos/acosf.c. In addition to the mathematical functions themselves, CORE-MATH provides

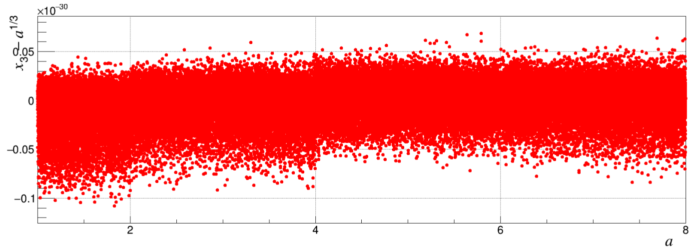

Fig. 5. The error of the cube root evaluation for the worst cases when the rounding test fails and the additional Newton iteration step is taken. FPU is operating in the downward mode.

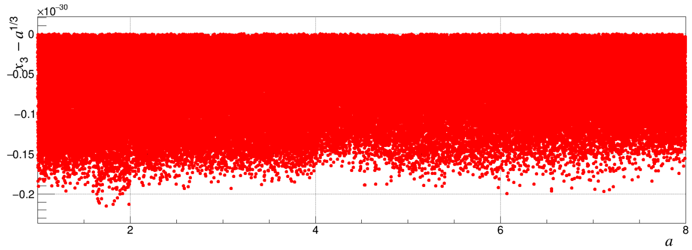

Fig. 6. The error of the cube root evaluation for the worst cases when the rounding test fails and the additional Newton iteration step is taken. FPU is operating in the upward mode.

infrastructure for assessing correctness and efficiency of our code.

A. Correctness

For each function, we have written a reference implementation using the MPFR library, which we use to evaluate the correctness. The `check.sh` script uses the aforementioned infrastructure and allows one to perform several kinds of checks:

- *exhaustive* checks: for univariate binary32 functions, the domain is so small that exhaustive checks can be done in a short time;
- *worst case* checks: for all other functions, exhaustive checks would take too long, and instead only worst cases are checked;
- *special* checks: for some functions, some interesting special cases can be easily computed, for example exact or midpoint cases (Section II-A).

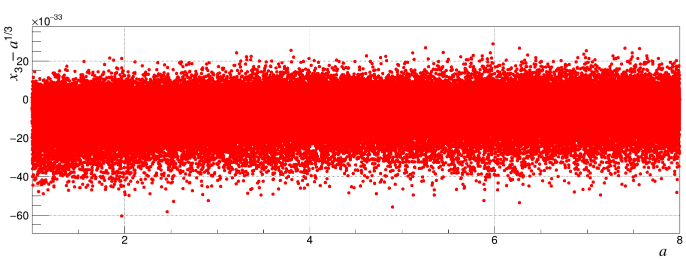

Fig. 4. The error of the cube root evaluation for the worst cases when the rounding test fails and the additional Newton iteration step is taken. FPU is operating in the round-to-nearest mode.

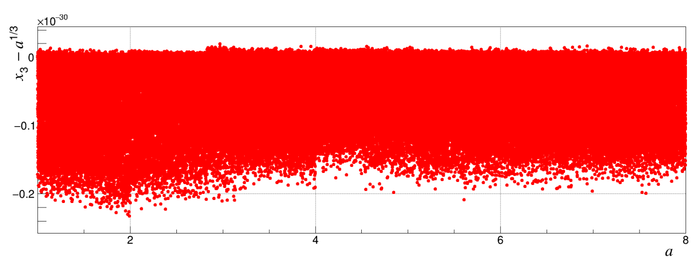

Fig. 7. The error of the cube root evaluation for the worst cases when the rounding test fails and the additional Newton iteration step is taken. FPU is operating in the toward-zero mode.

All these checks are run for each function and the four rounding modes. For exhaustive checks, OpenMP is used to parallelize iteration on all possible values.

B. *Efficiency*

Measuring reciprocal throughput and latency: To evaluate the efficiency of our implementations and other existing ones, we use either the `perf` tool or the (x86-specific) `rdtsc` instruction, both of which give similar results.

For each function, we define an input domain and, in a first process, we sample N_1 inputs and save them to a file. Then, in a second process, to measure the reciprocal throughput, we load these inputs and call the function under test on all these inputs N_2 times. The function is therefore called $N_1 \times N_2$ times. To measure latency, we introduce a dependency between each call so that one call needs to wait for the result of the previous call to proceed. In both cases, we measure the number of cycles taken by the second process, and divide by $N_1 \times N_2$. N_1 and N_2 are chosen so that the loading time is negligible in the overall measurement. Moreover, we try to choose N_1 big enough to avoid aggressive optimizations. Typically, $N_1 = 10^6$ and $N_2 = 10^3$.

The `perf.sh` script allows one to benchmark a specific CORE-MATH function, or a function from the system math library, or of any other library. The `perf-all.sh` evaluates all functions implemented in CORE-MATH and is used to get the figures of this article. By default `perf.sh` uses the following optimization flags: `CFLAGS=-O3 -march=native -ffinite-math-only`, which you can override as follows, for example if you want to disable the use of FMA:

```
$ ... CFLAGS="-O3" ./perf.sh atan2f
34.488
83.554
```

The first figure is for CORE-MATH (in number of cycles), the second one is for the system library (here GNU libc).

At the time we write this article, all C99 binary32 functions are available in CORE-MATH. A few of them are also correctly rounded in LLVM-libc, and we also compare with the GNU libc 2.35 (not correctly rounded), and with the Intel Math Library (from the docker image `intel/oneapi-hpckit`). Table II compares the reciprocal throughput of these four libraries for all C99 binary32 functions (plus `exp10f` which is not in C99), and on three binary64 functions (`acos`, `cbrt`, and `exp`) for which a first implementation is already available in CORE-MATH. Table II shows that the CORE-MATH routines are quite competitive with respect to the (incorrectly rounded) GNU libc and IML libraries, and for several functions even faster.

function	CORE-MATH	GNU libc	LLVM-libc	IML
acosf	31	*28*		***27***
acoshf	17	*22*		***13***
asinf	**25**	*27*		***25***
asinhf	25	*34*		***15***
atanf	19	*29*		***11***
atanhf	23	*64*		***15***
atan2f	25	*81*		***19***
cbrtf	17	*33*		***13***
cosf	**17**	*26*	*30*	*24*
coshf	18	*16*		***12***
erff	**14**	*53*		*24*
erfcf	**46**	*62*		*66*
expf	9	***6***	10	*8*
exp2f	10	***6***	23	*8*
exp10f	**10**	***10***		***10***
expm1f	**10**	*36*	11	*12*
hypotf	9	***8***	21	***9***
logf	11	***8***	9	*10*
log2f	10	***8***	11	*13*
log10f	12	*19*	**10**	*12*
log1pf	**13**	*22*	**13**	***13***
powf	32	***20***		*49*
sinf	**17**	*24*	*29*	*24*
sinhf	17	*51*		***13***
tanf	**16**	*48*		*32*
tanhf	13	*50*		***10***
acos	41	*48*		***30***
cbrt	44	*36*		***17***
exp	34	*13*		*23*

TABLE II
COMPARISON OF THE RECIPROCAL THROUGHPUT (IN CYCLES) OF CORE-MATH (COMMIT `F359CE4`), GNU LIBC 2.35, LLVM-LIBC (COMMIT `BEC8DFF`) AND INTEL MATH LIBRARY (IML, SHIPPED WITH ONEAPI COMPILER 2022.0.0), OBTAINED WITH `PERF.SH` USING THE `RDTSC` INSTRUCTION, ON AN AMD EPYC 7282, WITH GCC 10.2.1. BOTH CORE-MATH AND GNU LIBC WERE COMPILED WITH `CFLAGS=-O3 -MARCH=NATIVE -FFINITE-MATH-ONLY`, LLVM-LIBC WAS COMPILED WITH ITS DEFAULT FLAGS. CYCLES ARE ROUNDED TO THE NEAREST INTEGER. ITALICS VALUES CORRESPOND TO INCORRECT ROUNDING.

V. CONCLUSION

We exhibit for the first time a full set of C99 single-precision functions with correct rounding, for all rounding modes. These functions can either be used directly by the end-user in her/his application, or integrated into the current mathematical libraries. The CORE-MATH implementation outperforms for many functions the GNU libc, and for some functions the Intel math library, both being not correctly rounded. The efficiency of the CORE-MATH routines comes from several factors: (a) state-of-the-art argument reduction algorithms; (b) optimal minimax polynomials generated by Sollya [5]; (c) exploiting the knowledge of hard-to-round cases, and if they are not known, computing them using BaCSeL or new algorithms. We hope this will motivate the developers of mathematical libraries to provide correctly rounded functions, either as additional functions `cr_xxx`, or replacing their incorrectly-rounded routines. We also hope it will motivate the next revision of the IEEE-754 standard to *require* correct rounding for mathematical functions.

ACKNOWLEDGMENTS

The authors are grateful to the three anonymous reviewers for their useful comments. The search for hard-to-round cases of the single precision power function was possible thanks to the use of the Grid 5000 testbed and the EXPLOR centre hosted by the Université de Lorraine, for which we thank Patrice Ringot.

References

[1] Boldo, S., Graillat, S., and Muller, J.-M. On the robustness of the 2Sum and Fast2Sum algorithms. *ACM Transactions on Mathematical Software 44*, 1 (July 2017).

[2] Brisebarre, N., and Hanrot, G. Integer points close to a transcendental curve and correctly-rounded evaluation of a function. Working paper or preprint, https://hal.archives-ouvertes.fr/hal-03240179, Nov. 2021.

[3] Programming languages - C, working draft, N2731, ISO/IEC 9899:202x. http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2731.pdf, Oct. 2021.

[4] Charpilloz, C., Arteaga, A., Fuhrer, O., Montek, T., and Harrop, C. Reproducible climate and weather simulations: an application to the COSMO model. Platform fo Advanced Scientific Computing, Lugano. Switzerland, 2017. https://pasc17.pasc-conference.org/fileadmin/user_upload/pasc17/program/post125s2.pdf.

[5] Chevillard, S., Joldes, M. M., and Lauter, C. Sollya: an environment for the development of numerical codes. In *Third International Congress on Mathematical Software - ICMS 2010* (Kobe, Japan, Sept. 2010), K. Fukuda, J. van der Hoeven, M. Joswig, and N. Takayama, Eds., vol. 6327 of *Lecture Notes in Computer Science*, Springer, pp. 28 – 31.

[6] Daramy-Loirat, C., Defour, D., de Dinechin, F., Gallet, M., Gast, N., Lauter, C., and Muller, J.-M. CR-LIBM: A library of correctly rounded elementary functions in double-precision. Research report, LIP, 2006. https://hal-ens-lyon.archives-ouvertes.fr/ensl-01529804.

[7] de Lassus Saint-Genies, H., Defour, D., and Revy, G. Exact lookup tables for the evaluation of trigonometric and hyperbolic functions. *IEEE Trans. Computers 66*, 12 (2017), 2058–2071.

[8] Fousse, L., Hanrot, G., Lefèvre, V., Pélissier, P., and Zimmermann, P. MPFR: A multiple-precision binary floating-point library with correct rounding. *ACM Trans. Math. Softw. 33*, 2 (2007), article 13.

[9] Hanrot, G., Lefèvre, V., Stehlé, D., and Zimmermann, P. The BaCSeL software tool. https://gitlab.inria.fr/zimmerma/bacsel.

[10] IEEE standard for floating-point arithmetic, 2019. 84 pages.

[11] Innocente, V., and Zimmermann, P. Accuracy of mathematical functions in single, double, extended double and quadruple precision. https://members.loria.fr/PZimmermann/papers/accuracy.pdf, 2022. Version of February 11, 21 pages.

[12] Lauter, C. Q., and Lefèvre, V. An efficient rounding boundary test for $\mathrm{pow}(x, y)$ in double precision. *IEEE Trans. Comput. 58*, 2 (2009), 197–207.

[13] Lefèvre, V. Test of mathematical functions of the standard C library. https://www.vinc17.net/research/testlibm/.

[14] Lim, J. P., and Nagarakatte, S. High performance correctly rounded math libraries for 32-bit floating point representations. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021* (2021), S. N. Freund and E. Yahav, Eds., ACM, pp. 359–374.

[15] Markstein, P. *IA-64 and Elementary Functions: Speed and Precision*. Prentice Hall, 2000. Hewlett-Packard Professional Books.

[16] IBM MathLib. https://github.com/dreal-deps/mathlib. Non-official copy of the original library.

[17] Stehlé, D. *Algorithmique de la réduction de réseaux et application à la recherche de pires cas pour l'arrondi de fonctions mathématiques*. Thèse de doctorat, Université Henri Poincaré Nancy 1, 2005.

[18] Stehlé, D., Lefèvre, V., and Zimmermann, P. Searching worst cases of a one-variable function using lattice reduction. *IEEE Transactions on Computers 54*, 3 (2005), 340–346.

[19] Ziv, A. Fast evaluation of elementary mathematical functions with correctly rounded last bit. *ACM Trans. Math. Softw. 17*, 3 (1991), 410–423.