Correctly rounded evaluation of a function: why, how, and at what cost?

Nicolas Brisebarre, Guillaume Hanrot, Jean-Michel Muller, Paul Zimmermann

Correctly rounded evaluation of a function: why, how, and at what cost?

NICOLAS BRISEBARRE, Université de Lyon, CNRS, ENS de Lyon, Inria, Université Claude-Bernard Lyon 1, Laboratoire LIP (UMR 5668), France
GUILLAUME HANROT, Cryptolab, Inc. & Université de Lyon, CNRS, ENS de Lyon, Inria, Université Claude-Bernard Lyon 1, Laboratoire LIP (UMR 5668), France
JEAN-MICHEL MULLER, Université de Lyon, CNRS, ENS de Lyon, Inria, Université Claude-Bernard Lyon 1, Laboratoire LIP (UMR 5668), France
PAUL ZIMMERMANN, Université de Lorraine, CNRS, Inria, Loria, France

The goal of this article is to give a survey on the various computational and mathematical issues and progress related to the problem of providing efficient correctly rounded elementary functions in floating-point arithmetic. We also aim at convincing the reader that a future standard for floating-point arithmetic should require the availability of a correctly rounded version of a well-chosen core set of elementary functions. We discuss the interest and feasibility of this requirement.

Additional Key Words and Phrases: computer arithmetic, floating-point arithmetic, elementary functions, standardization, correct rounding, table maker's dilemma

1 MOTIVATION AND GENERAL ORGANIZATION OF THE ARTICLE

Motivation. The first goal of this article is to present an overview of the various computational and mathematical issues and progress related to the table-maker's dilemma, i.e., the problem of providing efficient, correctly rounded elementary functions in floating-point arithmetic. The second goal is to support the idea that future standards for floating-point arithmetic should require the availability of a correctly rounded version of a well-chosen core set of elementary functions. The exact contour of that set is still to be discussed, but it should contain the most frequently called functions, such as `exp`, `sin` (at least between $-\pi$ and π), `log`, etc., from which the other ones can be built. A possible good starting point is the set of the functions given in Table 9.1, *Additional mathematical operations*, of the IEEE 754-2019 Standard for Floating-Point Arithmetic [42, pp 58–59]:

$$\begin{gathered}
e^x, e^x - 1, 2^x, 2^x - 1, 10^x, 10^x - 1, \\
\log(x), \log_2(x), \log_{10}(x), \log(1+x), \log_2(1+x), \log_{10}(1+x), \\
\sqrt{x^2+y^2}, 1/\sqrt{x}, (1+x)^n, x^n, x^{1/n} (n \text{ is an integer}), x^y, \\
\sin(\pi x), \cos(\pi x), \tan(\pi x), \arcsin(x)/\pi, \arccos(x)/\pi, \arctan(x)/\pi, \arctan(y/x)/\pi, \\
\sin(x), \cos(x), \tan(x), \arcsin(x), \arccos(x), \arctan(x), \arctan(y/x), \\
\sinh(x), \cosh(x), \tanh(x), \operatorname{arcsinh}(x), \operatorname{arccosh}(x), \operatorname{arctanh}(x).
\end{gathered}$$

Another possible starting point is the list of mathematical functions (not much different) defined by the C Standard [12].

We do not claim that, for these functions, the correctly rounded implementation should be the only one available. One can imagine for each of these functions and each supported floating-point

This work was partly supported by the NuSCAP (ANR-20-CE48-0014) project of the French *Agence Nationale de la Recherche.*
Authors' addresses: Nicolas Brisebarre, Université de Lyon, CNRS, ENS de Lyon, Inria, Université Claude-Bernard Lyon 1, Laboratoire LIP (UMR 5668), Lyon, France, F-69000, Nicolas.Brisebarre@ens-lyon.fr; Guillaume Hanrot, Cryptolab, Inc. & Université de Lyon, CNRS, ENS de Lyon, Inria, Université Claude-Bernard Lyon 1, Laboratoire LIP (UMR 5668), Lyon, France, F-69000, Guillaume.Hanrot@cryptolab.co.kr; Jean-Michel Muller, Université de Lyon, CNRS, ENS de Lyon, Inria, Université Claude-Bernard Lyon 1, Laboratoire LIP (UMR 5668), Lyon, France, F-69000, Jean-Michel.Muller@ens-lyon.fr; Paul Zimmermann, Université de Lorraine, CNRS, Inria, Loria, Vandœuvre-lès-Nancy, France, F-54506, Paul.Zimmermann@inria.fr.

format two routines available to the end-user: a *fast* routine, and an *accurate* (correctly rounded) routine (although, as we are going to see in Section 5, it is not clear that a well-designed accurate implementation will be much slower than a fast implementation). The next version of the C Standard will simplify that possibility, since it will have reserved names, such as `cr_sin`, for correctly rounded mathematical functions [12].

Organization of this article. We have contributed or are contributing to the `CRLibm`[1] or `CORE-MATH`[2] libraries, which offer fast and correctly rounded evaluation of some of the functions mentioned above. Based on our expertise in the field, we wish to convince the reader that correct rounding of a core-set of functions is *useful* (or even *necessary*), that it is *feasible*, and that these correctly rounded functions can be evaluated at a *very reasonable (delay and energy) cost*. The article will be organized around these keywords. After a necessary reminder on the arithmetic framework in the sequel of this introduction, we address the *Why?* question in Section 3, the *How?* question in Section 4, and the *At what cost?* question in Section 5. Note that there are two different issues in each of these last two sections: *how and at what cost do we build function approximations whose evaluation is provably correctly rounded?* and *how and at what cost do we evaluate these approximations?* In practice, the final programs for function evaluation are quite simple. However, the preliminary mathematics and computation required to design them can, in some cases, be rather complex. Section 6 presents possible restrictions to the requirement of correct rounding. Finally, we present in Section 7 libraries that currently offer correctly rounded evaluations of mathematical functions for the IEEE 754 binary formats.

We deliberately choose to focus on *binary* floating-point arithmetic. Decimal arithmetic is an important issue [19], and much of what is said in this article can be extended to decimal, and yet the need for very accurate numerical computing with transcendental functions is probably less important in decimal applications—mainly financial calculations. Furthermore, the knowledge and tools required for designing correctly rounded functions (approximation tools, proof tools, knowledge of hardest to round cases, etc.) are less advanced in decimal arithmetic.

2 A REMINDER ON THE FLOATING-POINT ARITHMETIC FRAMEWORK

We only give the definitions and notation that are relevant for this article. More information on floating-point arithmetic can be found in [7, 38, 72, 74].

2.1 Floating-point numbers, basic binary formats

Definition 2.1. A binary, precision-p floating-point (FP) number is $\pm\infty$ or a number of the form

$$x = M \cdot 2^{e-p+1}, \tag{2.1}$$

where

- M is an integer, $|M| \leqslant 2^p - 1$, called the *integral significand* of the representation of x;
- e is an integer such that $e_{\min} \leqslant e \leqslant e_{\max}$, called the *exponent* of the representation of x.

In order to have a unique representation, we *normalize* the finite nonzero floating-point numbers by choosing the representation for which the exponent is minimum. A direct consequence is that if $|x| \geqslant 2^{e_{\min}}$, then $2^{p-1} \leqslant |M| \leqslant 2^p - 1$. Such a number x is said *normal*. If $|x| < 2^{e_{\min}}$, x is said *subnormal*. The largest finite floating-point number is $\Omega = 2^{e_{\max}+1} - 2^{e_{\max}-p+1}$.

The IEEE 754-2019 Standard for Floating-Point Arithmetic specifies various binary formats. The three *basic* binary formats are *binary32* (which was called *single precision* in the 1985 version of

[1] https://github.com/taschini/crlibm
[2] https://core-math.gitlabpages.inria.fr/

	precision p	minimal exponent e_{min}	maximal exponent e_{max}
binary32	24	−126	127
binary64	53	−1022	1023
binary128	113	−16382	16383

Table 1. Main parameters of the three basic binary formats (up to 128 bits) specified by IEEE 754 [42].

IEEE 754), *binary64* (formerly called *double precision*), and *binary128* (frequently called *quadruple precision*, it was not specified before the 2008 version of IEEE 754). The parameters of these formats that matter for this study are presented in Table 1. In the following, we denote $\mathcal{F}_p$ the set of all binary, precision-p FP numbers (to simplify, $e_{\min}$ and $e_{\max}$ are implicit).

For the sake of completeness, one should also mention the existence of a 16-bit format (binary16 in the IEEE 754 standard). Due to the small number of 16-bit floating-point numbers, exhaustive solutions are feasible and the problems studied in this article turn out to be much easier. The same is actually true regarding binary32 arithmetic, at least for univariate functions: the best way to make sure that an `arctan` program always returns a correctly rounded result is to try it with all possible 2^{32} input values and the various rounding functions, which takes at most a few hours on a modern laptop. The IEEE 754 Standard also defines a special value: NaN (Not a Number).

Definition 2.2. Assume a binary, precision-p, floating-point arithmetic. The *unit in the last place* of $t \in \mathbb{R}$ is the number

$$\operatorname{ulp}(t) = \begin{cases} 2^{\lfloor \log_2 |t| \rfloor - p + 1} & \text{if } |t| \geqslant 2^{e_{\min}}, \\ 2^{e_{\min} - p + 1} & \text{otherwise.} \end{cases}$$

Roughly speaking, $\operatorname{ulp}(t)$ is the distance between two consecutive FP numbers in the neighborhood of t. The error of "atomic calculations" (defined in §3) such as the elementary functions is in general expressed in ulps [37].

2.2 Correct rounding

The result of an arithmetic operation whose input values belong to $\mathcal{F}_p$ may not belong to $\mathcal{F}_p$ (in general it does not). Hence that result must be *rounded*. One of the most useful features brought by the IEEE 754 Standard is the requirement that the arithmetic operations and the square root should be *correctly rounded*: the user chooses a *rounding function*[3], and the four arithmetic operations, the FMA and the square root must return what would be obtained if their results were first computed exactly and then rounded to the target format. IEEE 754-2019 defines 5 different rounding functions; in the sequel, x is any real number to be rounded:

- round toward $+\infty$, or upwards: $\circ_u(x)$ is the smallest element of $\mathcal{F}_p$ that is greater than or equal to x. If x is larger than Ω, $\circ_u(x) = +\infty$;
- round toward $-\infty$, or downwards: $\circ_d(x)$ is the largest element of $\mathcal{F}_p$ that is less than or equal to x. If x is smaller than $-\Omega$, , $\circ_d(x) = -\infty$;
- round toward 0: $\circ_z(x)$ is equal to $\circ_u(x)$ if $x < 0$, and to $\circ_d(x)$ otherwise;
- round to nearest *ties to even*, denoted $\circ_{ne}(x)$ and round to nearest *ties to away*, denoted $\circ_{na}(x)$. If x is exactly halfway between two consecutive elements of $\mathcal{F}_p$, $\circ_{ne}(x)$ is the one for which the normalized integral significand M is an even number and $\circ_{na}(x)$ is the one for which $|M|$ is largest. Otherwise, both return the element of $\mathcal{F}_p$ that is the closest to x. When the tie-breaking rule is not important, we will write $\circ_n(x)$ for "x rounded to nearest".

[3]Called *rounding mode* or *rounding direction attribute* in the successive IEEE 754 jargons.

If applying this rounding rule with an unbounded exponent range would lead to a result of magnitude larger than Ω, then ∞ (with the appropriate sign) is returned.

The first three rounding functions are called *directed* rounding functions. The default rounding function is usually *round to nearest ties to even*. It is by far the most used in practice. The rounding functions are piecewise constant and increasing functions.

Although it is not a rounding function (it is not a function at all!), we say that $\hat{t}$ is a *faithful rounding* of t if $\hat{t} \in \{\circ_d(t), \circ_u(t)\}$. It is frequently considered that "correct rounding" (for round to nearest) is equivalent to "error less than 0.5ulp" or that faithful rounding is equivalent to "error less than 1ulp". As explained in [7], this is almost, but not entirely true. More precisely, we have,

PROPERTY 2.3. *Let $t \in \mathbb{R}$ and $\hat{t} \in \mathcal{F}_p$,*

- *if $|t - \hat{t}| < \frac{1}{2}\mathrm{ulp}(t)$ then $\hat{t} = \circ_n(t)$;*
- *if $\hat{t} = \circ_n(t)$ then $|t - \hat{t}| \leqslant \frac{1}{2}\mathrm{ulp}(t)$;*
- *if $\hat{t} \in \{\circ_d(t), \circ_u(t)\}$ then $|t - \hat{t}| < \mathrm{ulp}(t)$;*
- *if $|t - \hat{t}| < \mathrm{ulp}(t)$ and $|\hat{t}|$ is not of the form $(1 - 2^{-p})2^k, k \in \mathbb{Z}$, then $\hat{t} \in \{\circ_d(t), \circ_u(t)\}$.*

The first three items of Property 2.3 are rather intuitive. Let us explain the fourth. The distance between 1 and the FP number immediately *above* 1, say a, is 2^{-p+1}, whereas the distance between 1 and the FP number immediately *below* 1, say b, is 2^{-p}. Consider a real number t of the form $1 + \epsilon$, with $0 < \epsilon < 2^{-p}$ (which implies $1 < t < a$). That number t will be at distance $2^{-p} + \epsilon = \frac{1}{2}\mathrm{ulp}(t) + \epsilon < \mathrm{ulp}(t)$ from b. And yet $\circ_d(t) = 1$ and $\circ_u(t) = a$. Hence "directed rounding" and "error less than 1 ulp" are not strictly equivalent.

Definition 2.4. A *rounding breakpoint* (or simply, a breakpoint) is a point where the rounding function changes. For round-to-nearest functions, the rounding breakpoints are the exact middles of consecutive floating-point numbers (referred to as *midpoints* in the following). For the other rounding functions, they are the floating-point numbers themselves.

The requirement, since the original 1985 version of IEEE 754, that the four arithmetic operations, the square root (and some conversions) should be correctly rounded, together with the standardization of the handling of exceptions (not discussed in this article), have had a considerable impact on numerical software. They put an end to a chaos well described by Kahan [45], and today there is no serious disagreement that the arithmetic operations must be correctly rounded: this greatly facilitates the design, portability, and validation of numerical software.

We have no doubt that the same requirement regarding the set of elementary functions we are considering will also have a quite significant and positive impact on numerical software.

3 WHY?

Let us first explain why we feel that requiring the correct rounding of a set of elementary functions should be advisable.

3.1 The need for an unambiguous specification of the most frequent functions

Validating numerical programs, either by testing their behavior on a well-chosen set of input values or by providing a proof that they are correct, requires a clear and unambiguous specification of what they are supposed to compute. Clear specification is also essential for helping the design of portable software. Mooney [68] writes "*A software interface standard will aid in the development of portable software if it (· · ·) provides a* clear, complete *and* unambiguous specification…" This, in turn, requires a clear and unambiguous specification of the functions we can view as "atomic" in these programs. This has been done with success, in particular for the arithmetic operations and

the square root, by the IEEE 754 Standard on Floating-Point arithmetic. However, functions that appear in many numerical programs, such as the trigonometric functions or the various (bases e, 2, and 10) exponentials and logarithms are not fully specified, with the consequence that their quality and behavior may vary (and *does* vary [37]) significantly between math libraries and platforms. And yet, most users expect them to be of the highest quality. Already in 1980, Cody [16] wrote:

> *Software for the elementary functions normally resides in system libraries accompanying compilers for high level languages. Unless there is strong evidence of poor performance, users tend to regard these programs in the same way they regard the arithmetic operations in the computer. That is, they view them as friendly 'black boxes' that can be trusted to be efficient and accurate. Only careful preparation of software guarantees that the trust will not be violated.*

In a 2010 survey on verification methods [79], Rump wrote:

> *As another example, I personally believe that today's standard function libraries produce floating-point approximations accurate to at least the second-to-last bit. Nevertheless, they cannot be used 'as is' in verification methods because there is no proof of that property. In contrast, basic floating-point operations* $+, -, \cdot, /, \sqrt{\cdot}$*, according to IEEE 754, are defined precisely, and are accurate to the last bit.*

Natural questions that arise are *which functions* should be specified, and *what kind of specification* is desirable.

Concerning the choice of the functions that should be specified, the first criterion is the frequency with which they are called in numerical programs. This may of course vary from one application to another. The numbers of calls of the various mathematical functions in the simulation of proton collisions in the CERN CMS detector are given by Piparo and Innocente [76]. Their figures show that functions such as exp, log, and cos are very frequently called, and should be considered as "atomic", on nearly equal footing with the square root. The second criterion is which functions (called *primary* by Cody [15]) are frequently used as *basic building blocks* for writing software for the other functions. From that point of view, again, the exponentials, logarithms and trigonometric functions are frequently used for building the "special" functions [36] and, hence, are good candidates to be considered as "atomic". Hence, the list of functions given in the beginning of the introduction and extracted from [42, Section 9.1] is a good starting point. Incidentally, the list of functions in the C Standard, and the list of functions in Cody and Waite's book [14] are not so different.

Let us now consider the question of the kind of specification that is desirable. Requiring a *proven* relative error bound (e.g., 0.501ulp for round to nearest) would already be an improvement with respect to the current situation, and this was already suggested in 1984 by Black et al. [5], but it would not much ease the *reproducibility* of numerical calculations, which is becoming an important issue [1]. Ahrens et al. [2] define reproducibility as

> $(\cdots)$ *getting bitwise identical results from multiple runs of the same program, perhaps with different hardware resources or other changes that should not affect the answer.*

As pointed out by Ahrens et al., reproducibility is useful for debugging and testing software (one must for instance be able to "replay" a situation that led to an error), for reproducing simulations that produced rare events that need to be studied more carefully, for legal reasons (when several parties need to agree on the result of a calculation, or when one needs to justify a decision, after the fact, by the outcome of some simulation), and in the more and more frequent case when the same

quantity is computed at different places (and the result must be identical to allow for consistency of taken branches).[4]

For these reasons, we strongly believe that just specifying an error bound for the most frequent mathematical functions does not suffice, and that for each 4-tuple (function, rounding function, format, input value(s)) a *unique* result must be specified. The next question is: *which unique result?* One could argue that, for instance, the value returned by a predefined algorithm[5] might suffice. Not so. That would be the end of any incentive to improve mathematical function algorithms. Furthermore, would a standard survive for long if users could find libraries of functions here and there that claimed to be *better than the standard?*

3.2 The only specification that makes sense is correct rounding

For these reasons, the only viable solution is to require the returned result to be the best possible, i.e., the correct rounding of the exact mathematical result. In 1976, Paul and Wilson [75] considered the possibility of including the function library in the hardware, and reached the same conclusion:

> *The numerical result of each elementary function instruction will be equal to the nearest machine representable value which best approximates (rounded or truncated as appropriate) the infinite precision value for that exact finite precision argument for all possible machine representable input operands in the legal domain of the function.*

Of course, even with correct rounding, the behavior of the implementation of a function might differ from the mathematical behavior. For instance, the usage of the mathematical functions will not exhibit compositional behavior: the results of `exp(x+y)` and `exp(x)*exp(y)` will frequently differ. But this is already the case with the arithmetic operations: the results of `x*x-y*y` and `(x-y)*(x+y)` frequently differ. We also discuss an important issue about range violation in Section 6.4.

3.3 Now, correct rounding of many functions is feasible at a very reasonable cost

Correct rounding of the elementary functions was too strong a requirement at the time of Paul and Wilson [75] or Black et al. [5], but we aim at convincing the reader that now, this is feasible (Section 4) at a reasonable delay/energy cost (Section 5). Moreover, several implementations now do exist (Section 7).

4 HOW?

Let us now consider the problem of knowing how one can build correctly rounded function programs. The aim is to obtain fast and efficient programs, even if this is at the cost of a rather long pre-calculation of the various parameters (e.g., necessary accuracy of intermediate results, coefficients, special values to be tested, etc.) used by these programs. The reason is clear: the pre-calculation is done once and for all, whereas the evaluation programs will be used billions of times.

First, let us explain why the correct rounding of the transcendental functions is more complicated than the correct rounding of, say, addition. Suppose we want to evaluate function f at point x, where x is an FP number. Except in very special, rare cases, such as $\exp(0) = 1$, the exact value of $f(x)$ is generally not representable in finite-precision arithmetic. It can only be *approximated*. So, the only information we have is that $f(x)$ lies in some interval I. This interval may be very

[4] A surprising case is the online game industry, where one has to ensure that the game landscape is exactly the same for each player, and this landscape is generated locally on the player's computer.

[5] By this, we mean an *effective*, *practical*, algorithm that can actually be efficiently implemented. Of course, the definition itself of correct rounding (return what we would obtain if we could compute the function with infinite precision then round it to the target format) can be viewed as an "algorithm", but this definition is not efficiently implementable in practice.

narrow if we use high precision to compute the approximation, but it will still be nonzero in length. If all points of I round to the same FP number, then the correctly rounded value of $f(x)$ is that FP number. However, if I contains a rounding breakpoint, we cannot conclude. These two cases are illustrated by Figure 1, assuming that the rounding function is to nearest ($\circ_n$). When we cannot conclude, a possible solution is to recalculate successive approximations with increasing precisions (i.e., with intervals of decreasing length) until we are able to conclude. This is the essence of Ziv's strategy, presented in Section 4.1. The questions that naturally arise are: *Will this process eventually end?* Even if we have a proof that it does (which is the case for the most common functions), *does it terminate quickly*? These two questions form the *table maker's dilemma*, discussed in Section 4.2. The pre-calculations essentially consist in solving that problem, at least in some domains (in Section 4.3 we show that frequently, for tiny arguments, solving the table maker's dilemma is not necessary). The methods that can be used for the pre-calculation are described in Section 4.4. Once the pre-calculations are done, we can focus on the design of the programs that will evaluate the functions with correct rounding. The methods used in these programs will be presented in Section 4.5.

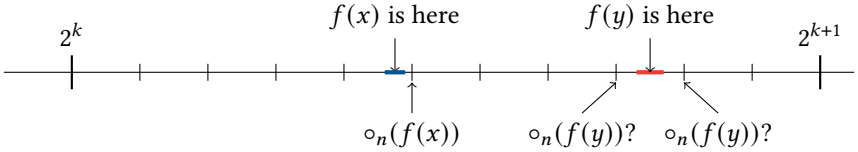

Fig. 1. The real line (between 2^k and 2^{k+1}) and the FP numbers (represented by the "ticks"). From the knowledge that $f(x)$ lies in the blue interval, we can deduce the value of $\circ_n(f(x))$. However, knowing that $f(y)$ lies in the red interval does not allow us to know if $\circ_n(f(y))$ is the FP number below $f(y)$ or the FP number above it.

4.1 Ziv's strategy

Ziv's onion peeling strategy [88] is a method to guarantee correct rounding at a cheap cost on average. Roughly speaking the idea is to start by approximating the function with an accuracy comparable to that of the current good-yet-not-correctly rounded libraries (to give an idea, something like 0.501ulp of the target format). Then a very simple test (such as the one presented in Section 4.5) allows one to know if the approximation suffices for returning a correctly rounded result. If it does, we are done. If it does not, we start the calculation again with a better accuracy, and so on until we are able to provide a correctly rounded result. More formally, we have a sequence of approximation functions f_k, $1 \leqslant k \leqslant n$, with increasing relative accuracy $p_1 < p_2 < \ldots < p_n$:

$$|f_k(x) - f(x)| \leqslant 2^{-p_k} \cdot |f(x)|.$$

On a uniform random input x, assuming a classical heuristic on the bits of $f(x)$ (see Section 4.2.2), if the target precision is p, it is possible to deduce a correctly rounded result from the approximation $f_k(x)$ with probability $\approx 1 - 2^{p-p_k}$. The key ingredients in Ziv's strategy are the following:

- to each function f_k is associated a *rounding test* (see Section 4.5) which, when it succeeds, *must* deliver the correct rounding;
- the sequence $f_1, \ldots, f_n$ is finite, and the last approximation $f_n(x)$ always delivers the correct rounding (*total Ziv's strategy*) or raises an exception or a flag (*partial Ziv's strategy*).

The second ingredient is crucial, and is strongly related to the table maker's dilemma (§4.2) and to the computation of "worst cases" (§4.4). In theory, the finiteness of the process (i.e., the fact

that $f_n(x)$ always delivers the correct rounding) requires either the knowledge that there are no nontrivial *exact* cases (i.e., FP numbers x such that $f(x)$ is a breakpoint), which is known to be true for the exponential, logarithmic and trigonometric functions thanks to Hermite-Lindemann's theorem [87], or the preliminary determination of all the exact cases. That task was for example accomplished for the binary64 power function by Lauter and Lefèvre [54], and for a function such as $\log_2$ the only exact cases are the trivial ones, i.e., $\log_2(2^k) = k$, with $k \in \mathbb{Z}$. In practice, for more complex functions, such as Γ,[6] finding a nontrivial exact point would be an extraordinary discovery. To be fully rigorous we can decide to raise a flag if f_n does not suffice to determine $\circ(f(x))$, while being almost certain that this will never happen if p_n is adequately chosen. Note that when the hardest-to-round cases are known, the last step of Ziv's strategy may sometimes be implemented by reading in a table the value of the function for the very few input values that remain possible; this strategy is used in some libraries.

4.2 The table maker's dilemma

The lack of requirement of correct rounding for elementary functions is mainly due to a difficult problem known as the *table maker's dilemma* (TMD), a term coined by Kahan [47]. When evaluating most elementary functions, one has to compute an approximation to the exact result, using an intermediate precision somewhat larger than the "target" precision p. The TMD is the problem of determining, given a function f, what this intermediate precision should be in order to make sure that rounding that approximation yields the same result as rounding the exact result. Ideally, we aim at getting the minimal such precision $\mathrm{htr}_f(p)$, that we call *hardness to round* of f (see Definition 4.3). That minimal precision derives from the minimal nonzero distance between the image of the function and a breakpoint. Hence, with directed rounding functions we need to know how close the image of the function can be to a FP number (which corresponds to Problem 4.1 below) and with rounding to nearest we need to know how close the image of the function can be to the middle of two consecutive FP numbers (which corresponds to Problem 4.2 below).

4.2.1 Formalization of the problem. Assume we wish to correctly round a real-valued function φ. We call *binade* an interval of the form $[2^k, 2^{k+1})$ or $(-2^{k+1}, -2^k]$ for $k \in \mathbb{Z}$. Note that if x is a "bad case" for φ (i.e., $\varphi(x)$ is difficult to round), then it is also a bad case for $-\varphi$ and $-x$ is a bad case for $t \mapsto \varphi(-t)$ and $t \mapsto -\varphi(-t)$. Hence we can assume that $x \geqslant 0$ and $\varphi(x) \geqslant 0$.

We consider that all input values are elements of $\mathcal{F}_p \cap [2^{e_1}, 2^{e_1+1})$. The method must be applied for each possible integer value of e_1.

If the values of $\varphi(x)$, for $x \in \left[2^{e_1}, 2^{e_1+1}\right)$, are not all included in a binade of the form $\left[2^{e_2}, 2^{e_2+1}\right)$, we split the input interval into subintervals such that for each subinterval, there is an integer e_2 such that the values $\varphi(x)$, for x in the subinterval, are in $[2^{e_2}, 2^{e_2+1})$. We stress that this task is easy for monotonic functions but might be harder for others (a typical example is trigonometric functions of large arguments).

We now restrict to one of those subintervals I included in $[2^{e_1}, 2^{e_1+1})$.

For directed rounding functions, the problem to be solved is the following:

Problem 4.1 (TMD, directed rounding functions). *Let $e_1, e_2 \in \mathbb{Z}$, $\varphi : I \subset [2^{e_1}, 2^{e_1+1}) \to [2^{e_2}, 2^{e_2+1})$.*

Let $p \in \mathbb{N}$, determine

[6] The possible trivial exact cases of the gamma function correspond to inputs that are positive integers, as $\Gamma(n) = (n-1)!$. For instance $\Gamma(14) = 13!$ is a breakpoint in binary32 arithmetic with a directed rounding, and $\Gamma(39) = 38!$ is a breakpoint in binary128 arithmetic with round-to-nearest.

- *the set of exact cases*

$$EC = \left\{ X \in [\![2^{p-1}, 2^p - 1]\!] \text{ such that } \frac{X}{2^{-e_1+p-1}} \in I \text{ and } 2^{p-1-e_2}\varphi\left(\frac{X}{2^{-e_1+p-1}}\right) \in \mathbb{Z} \right\}$$

  where $[\![2^{p-1}, 2^p - 1]\!] = \mathbb{Z} \cap [2^{p-1}, 2^p - 1]$*;*
- *the minimum* $\mu(p) \in \mathbb{Z}$ *such that, for* $X \notin EC$ *and for* $Y \in [\![2^{p-1}, 2^p]\!]$*, we have*

$$\left| 2^{p-1-e_2}\varphi\left(\frac{X}{2^{-e_1+p-1}}\right) - Y \right| \geqslant \frac{1}{2^{\mu(p)}}. \tag{4.1}$$

For rounding to nearest functions, the problem to be solved is the following:

Problem 4.2 (TMD, rounding to nearest functions). *Let* $e_1, e_2 \in \mathbb{Z}$*,* $\varphi : I \subset [2^{e_1}, 2^{e_1+1}) \to [2^{e_2}, 2^{e_2+1})$*.*

Let $p \in \mathbb{N}$*, determine*

- *the set of exact cases*

$$EC = \left\{ X \in [\![2^{p-1}, 2^p - 1]\!] \text{ such that } \frac{X}{2^{-e_1+p-1}} \in I \text{ and } 2^{p-1-e_2}\varphi\left(\frac{X}{2^{-e_1+p-1}}\right) - \frac{1}{2} \in \mathbb{Z} \right\};$$

- *the minimum* $\mu(p) \in \mathbb{Z}$ *such that, for* $X \notin EC$ *and for* $Y \in [\![2^{p-1}, 2^p - 1]\!]$*, we have*

$$\left| 2^{p-1-e_2}\varphi\left(\frac{X}{2^{-e_1+p-1}}\right) - Y - \frac{1}{2} \right| \geqslant \frac{1}{2^{\mu(p)}}.$$

In order to help the reader, we provide intuition for these two problems. The scaling factors 2^{-e_1} and 2^{-e_2} have the sole role of turning φ into a function $[1, 2) \to [1, 2)$; they should be disregarded for intuition. Setting them to 1 and dividing by 2^{p-1}, Equation (4.1) becomes $|\varphi(x) - y| \geqslant 2^{-\mu(p)-p+1}$, where $x = X/2^{p-1}$, $y = Y/2^{p-1}$ are arbitrary elements of $\mathcal{F}_p \cap [1, 2)$: a solution to Problem 4.1 implies that if $\varphi(x) \notin \mathcal{F}_p$, we can compute a correctly directed rounded value for $\varphi(x)$ by computing an approximation of it to the precision $\approx \mu(p) + p$ bits, and Problem 4.2 says the same regarding rounding to the nearest. The fact that $\mu(p)$ depends on p may seem nonobvious. We give an intuition for this using a heuristic argument in Section 4.2.2.

These problems lead to the following definition of the hardness to round. Other texts choose a different normalization, which differs from ours by an additive $p - 1$.

Definition 4.3 (hardness to round)**.** Let a precision p be given, $\circ$ be a rounding function and φ be a real valued function. Let x be a FP number in precision p and $e_2 \in \mathbb{Z}$ be the unique integer such that $\varphi(x) \in [2^{e_2}, 2^{e_2+1})$ (here again, we assume $\varphi(x) \geqslant 0$, since the extension to the other cases is straightforward).

If $\varphi(x)$ is not a breakpoint for $\circ$, the *hardness to round* $\varphi(x)$, denoted $\mathrm{htr}_{\varphi,\{x\},\circ}(p)$ is equal to the minimum $\mu(p) \in \mathbb{Z}$ such that the distance of $\varphi(x)$ to the nearest breakpoint is larger than or equal to $2^{-\mu(p)-p+1+e_2}$. If $\varphi(x)$ is a breakpoint for $\circ$ in precision p, we define $\mathrm{htr}_{\varphi,\{x\},\circ}(p)$ to be $-\infty$.[7]

The hardness to round φ over an interval I, denoted $\mathrm{htr}_{\varphi,I,\circ}(p)$, is then the maximum, over all $x \in \mathcal{F}_p \cap I$ of the hardness to round $\varphi(x)$, while the hardness to round φ is the hardness to round φ over $\mathbb{R}$, simply denoted $\mathrm{htr}_{\varphi,\circ}(p)$. When there is no ambiguity over the rounding function, we get rid of the symbol $\circ$.

Remark 4.4. Note that both Problem 4.1 and Problem 4.2 for precision p are subproblems of Problem 4.1 for precision $p + 1$.

[7]We stress that the natural choice in the last case would be $+\infty$. Setting the value to $-\infty$ is a convenience when we want to define the largest value of $\mu_{\varphi,\{x\},\circ}$ while excluding the breakpoints, as we now do.

Remark 4.5. If we assume that φ admits an inverse φ^{-1} and is differentiable over I and that we have a precise control over the image of φ' over I, it follows from the mean value theorem that addressing Problems 4.1 and 4.2 for φ over I is analogous to addressing Problems 4.1 and 4.2 for φ^{-1} over $\varphi(I)$. For instance, one can think of exp and log or $x \mapsto \sqrt[3]{x}$ and $x \mapsto x^3$. See Lemma 4.8 for an explicit statement.

4.2.2 A heuristic probabilistic approach and some partial results. If we have N FP numbers in the interval I being considered, it is expected that $\mathrm{htr}_{f,I}(p)$ is of the order of $\log_2(N)$ (hence p for a binade and most usual functions). This is supported by a probabilistic heuristic approach that is presented in detail in [70, 71] and that we now briefly recall.

Let φ be a real-valued function, assume that after the p^{th} bit, the bits of the significands of the values $\varphi(x)$, where x is a floating-point number, are sequences of independent random 0 or 1 with equal probability $1/2$. The probability that after bit p, we have

- for **directed rounding functions**, the bit sequence

$$\underbrace{00\cdots0}_{k \text{ bits}} \quad \text{or} \quad \underbrace{11\cdots1}_{k \text{ bits}}$$

- or, for **rounding to nearest functions**, the bit sequence

$$\underbrace{100\cdots0}_{k \text{ bits}} \quad \text{or} \quad \underbrace{011\cdots1}_{k \text{ bits}}$$

is[8] 2^{-k+1}. One can find such an estimate used in [30] and a probabilistic study has been done in [35]. Hence, if we have N floating-point numbers in the domain being considered, the number of values x for which we will have a bit sequence of the form indicated above is, under the probabilistic model stated above, around $N2^{-k+1}$.

In [9], Brisebarre, Hanrot and Robert give, under a mild hypothesis on f'', solid theoretical foundations to some instances of this probabilistic heuristic, targeting in particular the parameters that the `CRLibm` or `CORE-MATH` libraries use in practice.

4.2.3 Diophantine approximation results. We now recall several theoretical results that can prove useful, yet insufficient, for algebraic[9] functions like $1/\sqrt{\cdot}$, $\sqrt[3]{\cdot}$, ... and the exponential function, the latter being pivotal for elementary functions.

Algebraic functions. When $x \in \mathcal{F}_p$ and f is an algebraic function, the value $f(x)$ is an algebraic number.[10] When α is an algebraic number, the minimal polynomial of α over $\mathbb{Z}$ is the polynomial $P_\alpha \in \mathbb{Z}[X] \setminus \{0\}$, with relatively prime coefficients and positive leading coefficient, of least degree such that $P_\alpha(\alpha) = 0$. Let d denote the degree of P_α; we then say that α is an algebraic number of degree d. As of today, the only uniform theoretical statement that we can take advantage of is an old result due to Liouville [62–64].

Theorem 4.6 (Liouville). *Let α be an algebraic number of degree $d \geqslant 2$. There exists an effective constant C_α such that, for all p, $q \in \mathbb{Z}$, $q \geqslant 1$,*

$$\left|\alpha - \frac{p}{q}\right| \geqslant \frac{C_\alpha}{q^d}.$$

[8] This may not be true for some functions in the case of very tiny input values because of their particular Taylor expansion (for instance if x is tiny, $\sin(x)$ is extremely close to x). We discuss this case in Section 4.3.

[9] A function φ is *algebraic* if there exists $P \in \mathbb{Z}[x, y] \setminus \{0\}$ such that for all x such that $\varphi(x)$ is defined, $P(x, \varphi(x)) = 0$.

[10] i.e., the root of a nonzero polynomial with rational coefficients.

We can take, for instance, $C_\alpha = \frac{1}{\max_{|t-\alpha|\leqslant 1/2}|P'_\alpha(t)|}$. Better lower bounds of the form ξ_α/q^γ can be obtained. The value of the exponent γ has been regularly improved to culminate in Roth's Theorem [78], that gives an exponent $2+\varepsilon$ for any $\varepsilon > 0$ instead of[11] d. Actually, when tackling the TMD, the integer q is a power of 2 and we can therefore take advantage of Ridout's improvement [77] over Roth's theorem: a valid exponent is now $1+\varepsilon$ for any $\varepsilon > 0$. Unfortunately, none of these results come together with an effective constant ξ_α, which makes them useful only in an asymptotic setting – and useless, except as qualitative information, in ours.

Remark 4.7. Liouville's theorem was improved in an effective way by Fel'dman [33]. Unfortunately, for the parameter sizes of interest to us, it does not yield an information more accurate than the one provided by Theorem 4.6.

In [10], Brisebarre and Muller followed Liouville's approach to obtain simple effective upper bounds on the hardness to round $\mathrm{htr}_f(p)$ for algebraic f. See also [43, 52] for similar results. For instance, let $a \in \mathbb{N}\setminus\{0\}$, and consider $f_a : t \mapsto t^{1/a}$ and the round-to-nearest function. First we notice that, for all nonnegative real number x, and for all $k \in \mathbb{Z}$, we have $f_a(x2^{ka}) = 2^k f_a(x)$. It therefore suffices to work on $\mathcal{F}_p \cap [1, 2^a)$ instead of the whole $\mathcal{F}_p$. Then, we have $\mathrm{htr}_{f_a}(p) \leqslant (a-1)p + a + \log_2(a) - 2$. We remind the reader that we rather expect $\mathrm{htr}_{f_a}(p)$ to be of the order of p.

The exponential function and its siblings. Addressing the question of worst cases (§4.4) from a theoretical point of view requires:

(1) the determination of all FP numbers α in precision p such that $f(\alpha)$ is a breakpoint ;
(2) for the remaining FP numbers α in precision p, proving a lower bound on the quantity $|f(\alpha) - \beta|$ when β is any breakpoint.

Regarding the exponential function, we know from Hermite-Lindemann's theorem that the only exact case is $e^0 = 1$. As for the second condition, it turns out that proving such bounds in the somewhat more general case where α, β are algebraic numbers has been a major line of research in transcendence theory since the second half of the XIX-th century.

In view of this, we now give a table (Table 2) showing how bad cases for trigonometric and hyperbolic functions are related to (algebraic) bad cases for the exponential function; this relation can be used, in relation with results on the exponential function coming from transcendental number theory, to give bounds on bad cases. We discuss this issue in depth in Appendix A.

Table 2 is to be read in the following way: the row related to a function f gives α', β', ε' as functions of α, β, ε such that $|f(\alpha) - \beta| \leqslant \varepsilon \Rightarrow |\exp(\alpha') - \beta'| \leqslant \varepsilon'$. In the cases where further assumptions on α, β are used to improve the bound, those are given in the last column.

Finally, the following Lemma, which is a direct consequence of the mean value theorem, gives a relation between bad cases for a function and bad cases for its reciprocal. We state it for a function $f : [1/2, 1) \to [1/2, 1)$ for the sake of simplicity.

Lemma 4.8. *Let $p \in \mathbb{N}$, let $f : [1/2,1) \to [1/2,1)$ be a continuously differentiable function, such that $\mu_f(p)$ is the solution of Problem 4.1 on $[\circ_d(a), \circ_u(b)]$, where f is strictly monotonic on $[\circ_d(a), \circ_u(b)]$ for $1/2 \leqslant a \leqslant b < 1$. Then, if $\mu_{f^{-1}}(p)$ is the solution of Problem 4.1 for f^{-1} on $f([a,b])$, we have*

$$\mu_{f^{-1}}(p) \leqslant \mu_f(p) + \log_2\left(\max_{[\circ_d(a),\circ_u(b)]} |f'|\right).$$

The same statement holds for Problem 4.2.

[11] Note that in the quadratic case, that is to say $d = 2$, Liouville's result remains better.

Table 2. Relating trigonometric & hyperbolic functions to exp.

Function	α'	β'	ε'	if...
cos	$i\alpha$	$\beta \pm i\sqrt{1-\beta^2}$	$\sqrt{2\varepsilon}$	
cos	$i\alpha$	$\beta \pm i\sqrt{1-\beta^2}$	$2\varepsilon/\sqrt{\delta}$	$1-\lvert\beta\rvert \geqslant \delta$
sin	$i\alpha$	$i\beta \pm \sqrt{1-\beta^2}$	$\sqrt{2\varepsilon}$	
sin	$i\alpha$	$i\beta \pm \sqrt{1-\beta^2}$	$2\varepsilon/\sqrt{\delta}$	$1-\lvert\beta\rvert \geqslant \delta$
cosh	$\pm\alpha$	$\beta \pm \sqrt{\beta^2-1}$	$\sqrt{2\varepsilon}$	
cosh	$\pm\alpha$	$\beta \pm \sqrt{\beta^2-1}$	$2\varepsilon/\sqrt{\delta}$	$\beta \geqslant 1+\delta$
sinh	$\pm\alpha$	$\beta \pm \sqrt{\beta^2+1}$	4ε	
tan, cot	$2i\alpha$	$(1 \pm i\beta)/(1 \mp i\beta)$	2ε	$\alpha \neq 0$ (cot)
tanh, coth	-2α	$(1 \mp \beta)/(1 \pm \beta)$	2ε	$\alpha \neq 0$ (coth)

Table 3. Estimates of current theoretical upper bounds obtained from Appendix A and Lemma 4.8 for the hardness to round for exp, trigonometric and hyperbolic functions in the binary128 format. For each function f, we report the values θ_f such that, over a given binade, the hardness to round f is less than $\theta_f \cdot 113$.

Binade	exp	sin	cos	sinh	cosh	tan	cot	tanh	coth
$[1/8, 1/4)$	226	1371	1359	688	682	303	302	303	298
$[1/4, 1/2)$	297	2070	2058	1062	1057	410	410	409	405
$[1/2, 1)$	403	3288	3281	1698	1695	604	604	600	598
$[1, 2)$	593	5678	6481	2889	2889	1194	1196	931	929
$[2, 4)$	920	11408	10266	5285	5285	1854	1855	1507	1504
$[4, 8)$	1485	20395	20395	10155	10155	3361	3360	2634	2631

Applying the theoretical results presented in Appendix A and Lemma 4.8, we obtain upper bounds[12] for the hardness to round of exp, trigonometric and hyperbolic functions over the first few binades surrounding 1; the reader will find those bounds gathered in Table 3.

Regarding reciprocal functions, Lemma 4.8 says that a bound for the hardness to round of some function f on some interval $[a, b]$ can be deduced from that of f^{-1} over $f([a, b])$. For example, from the bounds for exp and tan in Table 3, it is easy to deduce bounds for log and atan.

As the figures in these tables are rather large, we need to investigate algorithmic solutions to find better bounds on the hardness to round.

4.3 Special values

It is frequently advantageous to process the case of tiny input and output values separately, for two reasons:

- first, many classical results on the accuracy of FP calculations (including the evaluation of the polynomials that approximate the functions) hold only when the operands are not in the subnormal domain;
- second, the hardness-to-round (see Definition 4.3) of some functions near zero may be huge, because the probabilistic heuristic presented in Section 4.2.2 is no longer valid, due to specific Taylor expansions of these functions (for instance, when x is tiny, as $\sin(x)$ is very close to x the value of $\sin(x)$ is extremely close to a breakpoint with directed rounding

[12]The SageMath code computing these bounds is available at https://members.loria.fr/PZimmermann/papers/#crsurvey.

functions): due to that one may believe that correctly rounding such functions when the argument is near zero is very difficult, although it is in fact extremely simple, as we are going to see.

4.3.1 The special case of tiny input values. As explained above, when dealing with input variables that are very close to zero, it is in general not necessary to know the hardest-to-round cases and correct rounding is very easy. Assume that function f has a Taylor expansion near zero:

$$f(x) = a_0 + a_1 x + a_2 x^2 + \cdots .$$

- If $a_0 \neq 0$ is not a breakpoint (for the target rounding function) then, depending on the sign of the first nonzero coefficient a_i, $i \geqslant 1$, the reciprocal images of one or both of the two breakpoints surrounding a_0 will tell us in which domain one can safely return $\circ(a_0)$, where $\circ$ is the desired rounding function. For instance, with $f(x) = 2^x$ and rounding to nearest, for
$$\log_2\left(1 - 2^{-p-1}\right) \leqslant x \leqslant \log_2\left(1 + 2^{-p}\right),$$
we have $\circ_n(2^x) = 1$ (in binary32 arithmetic, that domain corresponds to $-12102203 \times 2^{-48} \leqslant x \leqslant 6051101 \times 2^{-46}$).
- The case where a_0 is a nonzero breakpoint (which is not a rare case: consider for instance functions e^x or $\cos(x)$, with directed rounding functions) is very similar to the previous one. The knowledge of the reciprocal images of the two breakpoints surrounding a_0 allows one to easily determine a small domain where, possibly depending on the sign of x, one should just return a constant result. For instance, for the exponential function, if $0 \leqslant x \leqslant 2^{-p+1} - 2^{-2p+1}$ then $\circ_d(e^x) = 1$, and if $-2^{-p} \leqslant x < 0$ then $\circ_d(e^x) = 1 - 2^{-p}$. Similarly, for the cosine function, if $|x| \leqslant \circ_d(2^{(-p+1)/2})$ then $\circ_u(\cos(x)) = 1$.
- If $a_0 = 0$ and a_1 is a nonzero power of 2, so that the product $a_1 x$ is exact, and assuming that product cannot underflow (this is a frequent case with the usual math functions: consider for example $f(x) = \sin(x)$, or $\tan(x)$, or $\operatorname{arctanh} x$, etc.), then a reasoning similar to the one of the case $a_0 \neq 0$ allows one to find a domain where $\circ(f(x))$ is always equal to $\circ(a_1 x) = a_1 x$ (or to the preceding/next FP number). Let us give an example with the function
$$f(x) = \arctan(x) = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \cdots$$
If we exclude the case where x is a power of 2 (to be processed separately), we need to know when
$$\rho(x) = \left|\frac{x^3}{3} - \frac{x^5}{5} + \frac{x^7}{7} - \cdots\right| < \frac{|x|^3}{3}, \text{ for } x \neq 0,$$
is less than $\frac{1}{2}\mathrm{ulp}(x)$ (for round-to-nearest) or less than $\mathrm{ulp}(x)$ (for the other rounding functions). By reasoning on the binade $2^k < |x| < 2^{k+1}$ where x lies, one finds that in binary64 arithmetic ($p = 53$) and with a round-to-nearest rounding function, we have $\circ_n(\arctan(x)) = x$ for $|x| \leqslant 6^{1/3} \cdot 2^{-27} = 1.817120\cdots \cdot 2^{-27}$. In practice the domain where such simplifications can be made is easily found by the means of a binary search.
- If $a_0 = 0$, and $a_1 \neq 0$ does not fit exactly in $p+1$ bits or less (which in particular implies that it is not a power of 2) then a continued-fraction analysis allows one to find a nonzero lower bound ϵ on the relative distance $|a_1 x - b|/|a_1 x|$ between $a_1 x$ and a breakpoint b. Then one can deduce from the a_i's a domain where
$$\left|\frac{a_2 x^2 + a_3 x^3 + \cdots}{a_1 x}\right| < \epsilon.$$

In that domain, computing $\mathrm{RN}(f(x))$ is equivalent to computing $\mathrm{RN}(a_1 x)$. For many (but not all) values of a_1 this can be done using an algorithm presented in [11]. This is in particular possible (in the formats specified by IEEE-754) in the case $a_1 = \pi$, which is of interest for the sinpi and tanpi functions. When a_1 fits exactly in $p+1$ bits or less, it may be possible to find values x for which $a_1 x$ is exactly a breakpoint, in that case the method we have just presented is not applicable.

For some functions, it may also happen that the hardest to round cases become very easy to compute for tiny values. For instance, for tiny x, $\mathrm{sinpi}(x) = \sin(\pi x)$ is close to a breakpoint if and only if πx is close to a breakpoint (and in such a case, $\mathrm{sinpi}(x/2)$, $\mathrm{sinpi}(x/4)$, etc. are close to a midpoint too).

4.3.2 The special case of tiny output values. The analysis given in the rest of the article implicitly uses the fact that the breakpoints are numbers of the form $\pm B \cdot 2^k$ (for directed rounding functions) or $\pm(2B+1) \cdot 2^k$ (for round-to-nearest rounding functions), where $B \in \mathbb{N}$, $2^{p-1} \leqslant B \leqslant 2^p - 1$ and $k \in \mathbb{Z}$. This is not true in the subnormal domain, so a separate study is required in the areas where the outputs have absolute value less than $2^{e_{\min}}$. In general, however, this separate study is not too much of a burden:

- very often, the output is in the subnormal domain when the input too is extremely small, so that the study is similar to what has been done in Section 4.3.1 (just to give an example, when x is subnormal, $\circ_n(\sin(x)) = \circ_n(\sinh(x)) = \circ_n(\tan(x)) = x$);
- a very simple continued-fraction analysis, introduced by Kahan[13] and described in detail in [70, p. 208], allows one to find, for a given format, which floating-point number is closest to a non-zero integer multiple of $\pi/2$. This makes it possible to know what is the smallest possible absolute value of a trigonometric function when the input is not close to zero. It turns out that these values are far from the subnormal domain. For example, the binary64 number greater than $\pi/4$ and closest to an integer multiple of $\pi/2$ is $6381956970095103 \cdot 2^{797}$. Its cosine is about -4.687×10^{-19} which is well above (in absolute value) the subnormal threshold of 2.225×10^{-308}.

4.4 Search for exact, bad or worst cases - bounding the hardness to round from above

Let $\circ$ be a rounding function, we denote $\mathcal{B}_\circ$ the set of all breakpoints with respect to $\circ$. Let $y \in \mathbb{R}$, we define $\mathrm{dist}(y, \mathcal{B}_\circ) = \min\{|y - z|, z \in \mathcal{B}_\circ\}$.

Solving Problems 4.1 and 4.2 for a given function requires finding for which input values the function is either exact or closest to a breakpoint. The latter values will be called *worst cases*. More precisely, we have:

Definition 4.9. A *worst case* for a function f in a given format and rounding function $\circ$ is a FP number x in this format such that $2^{p-1-e_{f(x)}}\mathrm{dist}(f(x), \mathcal{B}_\circ)$ is minimal among all possible FP values of x such that $f(x)$ is not a breakpoint.

Here, we assume that $f(x)$ is not in the subnormal range. This is the case in practice, except for exp of large negative numbers.

Section 4.2.2 suggests that we should expect that apart from degenerate cases (for instance, $\cos(x)$ for x close to 0) this distance is of the order of $2^{-(p+\log_2(e_{\max}-e_{\min}+1))}$. Knowing the worst cases is of major importance in the context of correct rounding as this gives a bound on the final precision p_n such that Ziv's strategy succeeds.

More generally, m-bad cases are floating-point numbers x such that $2^{p-1-e_{f(x)}}\mathrm{dist}(f(x), \mathcal{B}_\circ) < 2^{-m}$, i.e.,

[13] See https://people.eecs.berkeley.edu/~wkahan/testpi/

- **[directed rounding functions]**

$$2^{p-1-e_{f(x)}} f(x) = \pm \underbrace{1\cdots\cdots}_{p \text{ bits}} . \underbrace{00\cdots0}_{m \text{ bits}} \cdots \quad \text{or} \quad \pm \underbrace{1\cdots\cdots}_{p \text{ bits}} . \underbrace{11\cdots1}_{m \text{ bits}} \cdots ;$$

- **[rounding to nearest functions]**

$$2^{p-1-e_{f(x)}} f(x) = \pm \underbrace{1\cdots\cdots}_{p \text{ bits}} . \underbrace{10\cdots0}_{m \text{ bits}} \cdots \quad \text{or} \quad \pm \underbrace{1\cdots\cdots}_{p \text{ bits}} . \underbrace{01\cdots1}_{m \text{ bits}} \cdots .$$

Note that a worst case is a FP number for which the length of the run of consecutive 0's or 1's after the round bit is longest. The above discussion suggests that $p + \log_2(e_{\max} - e_{\min} + 1)$ is a relevant estimate of this maximal length. Hence knowing m-bad cases for m slightly smaller than $p + \log_2(e_{\max} - e_{\min} + 1)$ is useful as a "stress test" for function implementation, whereas having the proof of the non-existence of m-bad cases for m somewhat larger than $p + \log_2(e_{\max} - e_{\min} + 1)$ is useful again as a bound on a final precision p_n such that Ziv's strategy succeeds.

We now present the existing algorithmic approaches for computing bad cases or establishing the value of the hardness-to-round (or an upper bound on that value).

4.4.1 Binary32 format. Apart from bivariate functions (see Sections 4.4.5 and 5.1.1 below) the table maker's dilemma in binary32 is easily solved for a given function by an exhaustive computation in a few hours on a modern laptop [80]. For each value of x one computes a sufficiently accurate interval approximation to $f(x)$ and determines the hardness to round $f(x)$. The same can of course be done with even smaller formats such as binary16 or bfloat16: the cost of the exhaustive approach is obviously proportional to the number of different FP numbers of the format under study, i.e., $(e_{\max} - e_{\min} + 2) \cdot 2^p - 1$.

4.4.2 Binary64 format. For univariate functions, the first idea that comes to mind is to do, as for binary32, exhaustive computations. While this seems difficult in software, it might be feasible in hardware as suggested in [26]. More subtle ideas proceed by splitting the domain into subintervals and replacing the function (assumed to be sufficiently smooth) by an approximation polynomial, often a Taylor approximation, over the interval under study; one is then reduced to study the problem in the polynomial case.

Lefèvre, together with Muller [55–57], studied the degree 1 case; in this case, the remaining problem is, given two real numbers α, β, to find two integers x, y, $|x| \leqslant X, |y| \leqslant Y$ such that $|\alpha x + \beta - y|$ is minimal. This problem can be solved by elementary arithmetic arguments, either the three distance theorem, or continued fractions (see e.g., [4]). These ideas lead to an algorithm of complexity $\tilde{O}(2^{2p/3})$ for floating-point numbers of precision p, which computes all bad cases for rounding in the domains under consideration. The bad cases published in [57] were obtained using that algorithm.

Higher degree approximations give rise to more complicated Diophantine problems. Stehlé, Lefèvre and Zimmermann [84], further refined by Stehlé [83], make use of a technique due to Coppersmith [17, 18] and based on lattice basis reduction to solve it. We will call Stehlé's variant SLZ algorithm in the sequel. We recall Corollaries 4 & 5 of [83], adapted to our context.

Theorem 4.10 (Stehlé [83]). *For all $\varepsilon > 0$, there exists a heuristic algorithm of complexity $2^{p(1+\varepsilon)/2}$ which, given a function f, returns all FP numbers $x \in [1/2, 1)$ of precision p such that the hardness to round $f(x)$ is $\geqslant p$.*

There exists a polynomial-time heuristic algorithm which returns all FP numbers $x \in [1/2, 1)$ of precision p such that the hardness to round $f(x)$ is $\geqslant 4p^2$; the latter works by reducing a lattice basis of dimension $O(p^2)$ of $\mathbb{R}^m$ for some $m = O(p^4)$.

This theorem can be extended to any fixed binade.

For most functions, the heuristic character of the algorithm is rather mild (i.e., the algorithm works in practice as expected on almost all inputs). Note that the algorithm obviously fails in a trivial case like $f(x) = x$ where the number of solutions to the problem is 2^{p-1}.

Recently, Brisebarre and Hanrot [8] presented an improvement over the SLZ algorithm. Their method (which we will call BH) is still based on lattice basis reduction, but instead of reducing the problem for f to the same problem for an approximation (Taylor) polynomial for f as it is done in [83, 84], they work on the function f itself as long as possible. This is made possible thanks to rigorous uniform approximation techniques based on Chebyshev interpolation.

This brings significant improvements on several algorithmic aspects, resulting in major improvements in the polynomial part of the asymptotic complexity (lattice basis reduction). Their approach also permits an improvement on the second part of Theorem 4.10 for regular functions with moderate growth at ∞ (exp, sin, cos). They prove the following theorem:

Theorem 4.11 (Brisebarre, Hanrot [8]). *Let $f \in \{\exp, \sin, \cos, \sinh, \cosh\}$. For all $\varepsilon > 0$, there exists a polynomial-time heuristic algorithm which returns all FP numbers $x \in [1/2, 1)$ of precision p such that the hardness to round $f(x)$ is $\geqslant (1+\varepsilon)p^2/\log p$; the latter works by reducing a lattice of dimension $O((p/\log p)^2)$ of $\mathbb{R}^m$ for some $m = O((p/\log p)^2)$.*

Actually, the theorem can be stated for more functions, precisely *entire functions of finite order* (this includes, for instance, the erf function). Note also that by combining this result with Lemma 4.8, we can extend the theorem to the reciprocal functions.

4.4.3 Binary128 format. As of today, finding the worst cases in binary128 seems by far out of reach. However, a somewhat easier task is to show that, for the usual functions that we target, there are no nontrivial (θp)-bad cases, for a reasonable constant θ ($\theta = 5, 6, 7$, say). The SLZ and BH algorithms can perform this task (see [85] for the SLZ algorithm, and see Section 5.1.3).

4.4.4 Large arguments. In general, the approximation-based strategies of Section 4.4.2 and Section 4.4.3 are only useful for moderate arguments. More precisely, we need the validity domain of the approximation to contain a significant number of floating-point numbers for that strategy to be better than the exhaustive approach; for very large arguments this requires using a polynomial approximation of very large degree. For functions with fast growth (or functions such that f^{-1} has fast growth, by reducing to f^{-1} thanks to Lemma 4.8) this is not an issue since large arguments can be discarded as they lead to overflow (think of exp).

For periodic functions, a modification to SLZ is described in [39]. It leads to a modified version of Theorem 4.10 where $(1+\varepsilon)/2$ is replaced by the slightly worse exponent $(7-2\sqrt{10})(1+\varepsilon) \approx 0.68(1+\varepsilon)$.

As a consequence, in binary64 arithmetic, the bad cases for the sine and cosine functions are not known for input arguments larger than 2^{11}. Various strategies can be used to cope with this problem: only require/implement correct rounding for small (typically $< 2^{11}$ in absolute value) arguments, use Ziv's technique with an arbitrarily large number of steps, implement functions that are "correctly rounded with high probability" (using the probabilistic argument presented in Section 4.2.2), etc.

4.4.5 Bivariate functions. The landscape is much less clear regarding bivariate functions – except for the hypot function for which general results concerning algebraic functions apply.

It is highly likely that all the previous algorithms (Lefèvre, SLZ, BH) can be adapted to this setting, see for example [86].

4.4.6 Formal verification. All these large computations using rather complicated pieces of software may raise concerns, particularly in the binary128 case since the output of the quest of $5p$ or $7p$-bad cases is typically "No", namely that such bad cases do not exist. The need for formal proof certification of those results seems obvious; preliminary work has been performed in this direction [67] in the case of the SLZ algorithm. Most of the computation time of this algorithm is spent finding *auxiliary polynomials*. Once computed, they can be stored as part of *certificates*, which allows for a verification that can be much faster than the actual computation. This verification can be performed using a proof assistant such as Coq, or, for a faster but weaker verification, simply by another tool written independently from the first one. The BH algorithm is based on similar ideas, and extending this work to the BH algorithm seems feasible; a large part of the Coq formalization can actually be re-used.

4.5 Implementation of correctly rounded evaluation routines

Several works in the literature describe the design and implementation of algorithms for correct rounding of mathematical functions: [20, 22, 24, 28, 29, 34, 41, 50, 53, 58–60, 88] to cite only a few.

A program must first filter special cases (infinities, NaNs, possibly tiny values using the properties presented in 4.3) and exact cases : we will not describe this task.

The main idea for implementing correctly rounded functions is to use a few steps (in general, 2) of Ziv's strategy (presented in Section 4.1): we first use a *fast path* delivering an approximation to the function accurate to, say, $p + 10$ bits; then we perform a *rounding test* that checks if the result of the fast path suffices to deduce the correctly rounded result; and, only if needed, we use an *accurate path*, that must be accurate enough to guarantee correct rounding for all inputs (possibly with a very few exceptions, handled by the means of a table that contains the value of the function for these exceptional cases or, rarely, by an even more accurate third path). To design the accurate path, we take into account the accuracy it should provide, deduced from the upper bound on the hardness to round obtained using the methods presented in Section 4.4. The accurate path is necessarily slower, but is called only in rare cases.

To implement each path, one has to use efficient algorithms. These algorithms are in general very similar to those used in the current mathematical libraries (references to such algorithms are [3, 66, 70]). They consist in three steps:

(1) argument/range reduction (if available for the function to be implemented),
(2) evaluation of a precomputed minimax polynomial (or, more rarely a minimax rational fraction), and
(3) argument reconstruction (if argument reduction was performed).

The major difference with the not correctly rounded libraries is that one must carefully bound the total error of the algorithm (including error of the range reduction and reconstruction, approximation error of the polynomials, rounding errors of the arithmetic operations), because that error is used by the rounding test. A simple rounding test, if y is the computed approximation to the function, ε a bound on the total error, and $\circ$ the rounding function, consists in computing $\ell = \circ(y - \varepsilon)$ and $h = \circ(y + \varepsilon)$. If both ℓ and h round to the same FP number y^* then, due to the monotonicity of the rounding function, all values in $[y - \varepsilon, y + \varepsilon]$ (including, therefore, the "exact" value of the function) round to y^*. This simple test works for any of the IEEE 754 rounding functions. More specific tests do exist for rounding to nearest [25] and for some functions [54].

The bound on the error can be obtained by paper-and-pencil calculations, but such calculations are tedious and error-prone. Fortunately tools are available to alleviate that task. The error of range reduction can be deduced using techniques presented in [6, 46, 82]. Sollya [13] provides near-optimal polynomial approximations under constraints (such as the requirement that all coefficients must

be representable in binary64), and delivers rigorous bounds of these approximations (i.e., bounds on the distance between the exact value of the function and the exact value of the polynomial). Gappa [23] allows one to compute the rounding error that occurs when evaluating the polynomial approximations. One can even build a formal proof of the error bounds with tools that are reasonably easy to use [27].

At times, the function evaluation will need to perform arithmetic operations with a precision higher than the target precision p. This is obvious for the accurate path, and for the fast path, with some care, this need can be limited to a very few operations only: for instance if the absolute value of the reduced argument is significantly smaller than 1, and if the absolute values of the coefficients of the approximating polynomial decrease – which is in practice the case for many functions – then only the very few last steps of a Horner evaluation of the approximating polynomial will need a large precision. To implement these high-precision operations, when a wider format is available in hardware, it is usually the best choice (for instance, if the target format is binary32, one can use binary64 for internal computations). When this is not the case, an alternative is to use double-word arithmetic, which offers about $2p$ bits of precision [61], or even triple-word arithmetic, which offers about $3p$ bits [32]. Efficient double-word algorithms are known with tight and rigorous error bounds [44], and have even been formally proven [69]. Another solution, in some cases, is integer-based arithmetic [65].

Let us detail these steps on the example of computing $\exp(x)$ in binary64 arithmetic. The solution briefly described below is similar (albeit slightly different) to that used in CORE-MATH at the time of writing.

Argument reduction–fast path. Since the binary64 range has absolute values from 2^{-1074} to 2^{1024} (excluding zero), once underflow and overflow have been filtered, one can assume $|x| < 745$. The input x is reduced to a smaller range as follows:

$$x = n \log 2 + x',$$

where n is an integer, $|n| \leqslant 1075$, and $|x'| \leqslant \log 2/2$, which yields $\exp(x) = 2^n \cdot \exp(x')$. The number $\log 2$ is represented by the sum of two binary64 numbers, L_h and L_ℓ, where the significand of L_h fits in 42 bits, which since $|n| < 2^{11}$ ensures the floating-point multiplication $n \cdot L_h$ is exact (this is Cody and Waite's range reduction algorithm [14]). The reduced argument x' is further reduced as follows:

$$x' = i2^{-k} + \ell,$$

where i is an integer, and $|\ell| \leqslant 2^{-k-1}$. Then we have $\exp(x') = \exp(i2^{-k}) \exp(\ell)$, and the values $\exp(i2^{-k})$ can be tabulated. For example with $k = 6$, we have $|i| \leqslant 22$, a small table of 45 values suffices, and the reduced argument satisfies $|\ell| \leqslant 2^{-7}$.

Polynomial evaluation–fast path. We want a polynomial that approximates $\exp(\ell)$ for $|\ell| \leqslant 2^{-7}$, with at least $p + 10 = 63$ bits of accuracy. Sollya provides such a polynomial $p(\ell)$, with binary64 coefficients, and also delivers a rigorous bound on the mathematical relative error $(|p(\ell) - \exp(\ell)|)/\exp(\ell)$ for $|\ell| \leqslant 2^{-7}$. Sollya finds a degree-7 polynomial, with a relative error about 2^{-77}.

To evaluate this polynomial efficiently, we can use Estrin's scheme [31]: if the polynomial is $p_0 + p_1\ell + \cdots + p_7\ell^7$, we first evaluate $y = \ell^2$, $a_0 = p_0 + p_1\ell$, $a_2 = p_2 + p_3\ell$, $a_4 = p_4 + p_5\ell$, $a_6 = p_6 + p_7\ell$, then $z = y^2$, $b_0 = a_0 + a_2 y$, $b_4 = a_4 + a_6 y$, and finally $t = b_0 + b_4 z$. In this case, Estrin's scheme has depth 3, whereas Horner's scheme has depth 7, thus this reduces the latency if the processor can perform several operations simultaneously. Note that the high order terms a_4, a_6, b_4, y, z can be computed in binary64 arithmetic, whereas the low order terms should be computed with a larger precision (for example using double-word arithmetic).

Argument reconstruction–fast path. If t is the approximation to $\exp(\ell)$, then $\exp x \approx 2^n \exp(i2^{-k})t$. It thus suffices to multiply t by the tabulated value $\exp(i2^{-k})$. The multiplication by 2^n is trivial in binary: we just have to add n to the floating-point exponent.

Accurate path. Before designing the accurate path, one has to know hard-to-round cases. If one neglects tiny values of x (for which the exponential is easily obtained: see Section 4.3), the hardest-to-round input for exp in binary64 is

$$x = \texttt{-0x1.12d31a20fb38bp+5} = -34.353077180544538293816003715619444847106933593 75,$$

whose exponential has 58 identical bits after the round bit: its binary representation is

$$1.0101101100001011111100110010010001001000000100000 10$$
$$1001111111111111111111111111111111111111111111111111111111111001\cdots\times 2^{-50}.$$

One thus needs an accurate path with an accuracy of about $53 + 1 + 58 = 112$ bits.

The argument reduction is very similar to that of the fast path, with $\log 2$ now represented by the sum of three FP numbers, and the need of two tables containing $\exp(i/2^6)$ for $|i| \leqslant 22$ and $\exp(j/2^{12})$ for $0 \leqslant j < 2^6$: we now have

$$x = n\log 2 + \frac{i}{2^6} + \frac{j}{2^{12}} + \ell,$$

with $|\ell| \leqslant 2^{-13}$. Sollya[14] finds a degree-6 polynomial whose first three coefficients are $1 + x + x^2/2$, whose degree-3 coefficient is a double-word number (in binary64 arithmetic), and whose coefficients of degree 4, 5, 6 are binary64 numbers, that approximates the exponential function in $[-2^{-13}, 2^{-13}]$ with a relative accuracy better than 2^{-108}. The corresponding global error bound does not guarantee 112 bits of accuracy. However, it is easy to check if the few hard-to-round cases that "exceed" the global bound are correctly handled by the program, and if not, to handle them separately as exceptional cases.

The argument reconstruction is similar to the fast path: if t is the approximation to $\exp(\ell)$, then $\exp x \approx 2^n \exp(i/2^6)\exp(j/2^{12})t$, where the double-word approximations of $\exp(i/2^6)$ and $\exp(j/2^{12})$ are tabulated.

5 AT WHAT COST?

It is important to make a clear distinction between the cost of *designing* the correctly rounded function evaluation algorithms (which is essentially the cost of solving the table maker's dilemma for the function under consideration) and the cost of running the algorithm once it has been designed. The former is large, as one might expect after reading Sections 4.2 and 4.4, but the corresponding work is done once and for all. On contrary the function evaluation programs (especially for very frequent functions such as exp or cos) will be run billions of times, and they are simple and fast. We address the cost of designing the algorithms in Section 5.1, and the cost of running the elementary function algorithms in Section 5.2.

5.1 Search for bad or worst cases - bounding the hardness to round from above

We now consider the cost of running the algorithms when tackling the three main binary formats and comment on the cost of formal verification of the output results.

5.1.1 Binary32 format. The table maker's dilemma in binary32 is easily solved for a given univariate function by an exhaustive computation in a few hours on a modern laptop. As for the bivariate functions, solutions have been proposed in [81] for the most useful of them, namely power, hypotenuse and arctan2 (with a possible adaptation to atan2pi). Work still needs to be done on

[14]Called with the command line `P = fpminimax(exp(x),6,[|1,1,1,DD,D,D,D|],[-2^-13;2^-13],relative)`.

more general bivariate functions in binary32, but they do not seem out of reach (the difficulty of handling them should be similar to the difficulty of handling univariate functions in binary64).

5.1.2 Binary64 format. The SLZ algorithm is implemented in the BaCSeL software tool [40]. The time to search bad cases (with at least 43 identical bits after the round bit) in binary64 for $1/2 \leqslant x \leqslant 1$ is about 120 core hours for e^x and 90 hours for $x^{1/3}$ using BaCSeL on an AMD EPYC 7282 with gcc 14.2.0. These figures show that a full binade can be checked in a few hours on a 64-core processor. Given the fact that for some functions like e^x and $x^{1/3}$, only a few binades need to be checked (for e^x, one gets underflow or overflow for $|x| > 745$, and for $x^{1/3}$, bad cases in $[2^{e+3k-1}, 2^{e+3k})$ are those in $[2^{e-1}, 2^e)$ multiplied by 2^{3k}, thus only three binades need to be checked), the total time to check a binary64 function usually takes from a few hours for $x^{1/3}$ to a few days. As an example, Table 4, extracted from [70], gives the hardest-to-round points for functions $\log(x)$ and $\log(1+x)$ in binary64 arithmetic.

Function	Domain	Argument	Truncated result	Trailing bits
$\log(x)$	$[2^{-1074}, 2^{-1})$	1.EA71D85CEE020P-509	-1.60296A66B42FFP8	$1\,1^{60}\,0000\cdots$
		1.9476E304CD7C7P-384	-1.09B60CAF47B35P8	$1\,0^{60}\,1010\cdots$
		1.26E9C4D327960P-232	-1.4156584BCD084P7	$0\,0^{60}\,1001\cdots$
		1.613955DC802F8P-35	-1.7F02F9BAF6035P4	$0\,1^{60}\,0011\cdots$
	$[2^{-1}, 2^{1})$	1.BADED30CBF1C4P-1	-1.290EA09E36478P-3	$1\,1^{54}\,0110\cdots$
	$[2^{1}, 2^{1024})$	1.C90810D354618P245	1.54CD1FEA76639P7	$1\,1^{63}\,0101\cdots$
		1.62A88613629B6P678	1.D6479EBA7C971P8	$0\,0^{64}\,1110\cdots$
$\log(1+x)$	$(2^{-51}, 2^{1024})$	1.8000000000003P-50	1.7FFFFFFFFFFFEP-50	$1\,0^{99}\,1000\cdots$
	$(-1, -2^{-51}]$	-1.7FFFFFFFFFFFDP-50	-1.8000000000001P-50	$0\,1^{99}\,0110\cdots$

Table 4. *Non trivial hardest-to-round points for functions* $\log(x)$ *and* $\log(1+x)$ *[70]. The values given here suffice to round functions* $\log(x)$ *and* $\log(1+x)$ *correctly in the full binary64 range (when* $|x| \leqslant 2^{-51}$*, the correct rounding of* $\log(1+x)$ *can be obtained using the technique presented in Section 4.3.1).*

In light of this data, the table maker's dilemma can be considered as solved for the binary64 format for univariate functions with the possible exception of trigonometric functions of large arguments (bivariate functions appear, in the current state of knowledge, to be out of reach); known bad cases for the binary64 format are available on Lefèvre's page https://www.vinc17.net/research/testlibm/ and in the CORE-MATH source code (for example https://gitlab.inria.fr/core-math/core-math/-/blob/master/src/binary64/exp/exp.wc for the exp function). The latter also provides m-bad cases with $m = 44$ for several functions (with a restricted range for some trigonometric functions: as we are writing these lines, the bad cases for $\cos(x)$ and $\sin(x)$ are known only for $|x| \leqslant 2^{11}$, and the bad cases for $\tan(x)$ are known only for $|x| \leqslant 10.5\pi$).

5.1.3 Binary128 format. We have compared the performance of implementations of SLZ and BH algorithms, namely BaCSeL[15] and LACoR[16], when tackling the determination of $5p$-bad cases over a binade. The time to search $5p$-bad cases of e^x for $x \in [1/4, 1/2]$ for quadruple precision on a Core i7-8700 at 3.20GHz was estimated to 305 years for SLZ with BaCSeL parameters $d = 16$, $\alpha = 5$, $t = 76$; and estimated to 445 days for BH with parameters $d = 12$, $\rho_1 = 536870912$, $t = 88.5$, $n_1 = 58$, $n_2 = 3$.

[15] https://gitlab.inria.fr/zimmerma/bacsel
[16] https://perso.ens-lyon.fr/nicolas.brisebarre/lacor.tgz

Of course, this remains large, and the corresponding energy consumption is far from negligible. However, this is done once for all, for a few core functions only, and will be amortized over billions of function evaluations. In view of this, LACoR timings remain acceptable, and open the way to a guaranteed implementation of Ziv's strategy for univariate functions over a restricted domain containing the most commonly used binades (bivariate functions seem to be out of reach at the current state of knowledge).

As in binary128 we only have an overestimate of the hardness to round, a correctly rounded binary128 function may, at times, be significantly slower than a state-of-the-art function (which is not the case, for instance, in binary64). As a consequence, we feel that, although correct rounding in the binary128 format is desirable, it is difficult, as of today, to make it mandatory.

Even though these computation times may seem important, it should be noted that all the underlying algorithms are embarrassingly parallel, and the corresponding computations can be distributed over a large number of cores / processors / machines in order to reduce the real computation time.

In view of these timings, one may wonder why we ran both SLZ and BH. First, the discrepancy depends on the precise choise of parameters and may even be smaller in other cases. Second, SLZ has been published for a long time while as of today BH is still under review; similarly, BaCSeL has been extensively tested while LACoR is still an experimental implementation. Finally, as in most cases these libraries only return $\emptyset$, having the second one as a check of the results (in small selected domains) of the first one is a useful feature. Developing an independent certificate checker, as mentioned in Section 4.4.6, would allow one to only use the most efficient of the two libraries in all cases, depending on the parameter values.

5.1.4 Formal verification. The cost of the formal proof + certificate-based approach has been evaluated in [67]. The actual running time in Coq depends strongly on the underlying arithmetic chosen, but also on the parameter choices. For binary64, verifying SLZ-based certificates that prove that the hardness to round for exp is $\leqslant 247$ bits took less than one day on an 8-core Xeon X5550 running at 2.67GHz, and each certificate took 6 times longer to generate (in C) than to verify (in Coq). It should be noted, however, that in these experiments, the parameter choices were biased towards harder generation / easier verification.

5.2 Cost of the function evaluation algorithms

Let us first consider the case of scalar functions. The function evaluation algorithms used in libraries such as CRLibm or CORE-MATH (see Section 7) are based on Ziv's strategy, presented in Section 4.1, with the first polynomial approximations tailored so that the probability of not obtaining a correctly rounded result from these first approximations is small (e.g., of the order of 10^{-3}). The number of coefficients of these first approximations is comparable to the one used in the libraries currently provided by the chip or compiler designers. The direct consequence of this is that the cost of the CORE-MATH library in terms of latency is on average very similar to the cost of these not correctly rounded current math libraries (a few figures are given in Table 5. For more, see https://core-math.gitlabpages.inria.fr). Examples with the CORE-MATH library [81] show that the worst case delays can be made reasonable. The energy cost will essentially be the cost of the range reduction and the first polynomial evaluation, so it will also be more or less the same as the other libraries.

There is still a small theoretical overhead between a correctly rounded function and a function that returns a result within around 1 ulp:

- a correctly rounded function needs a fast path with about $p + 10$ correct bits, where p is the target precision, so that the accurate path is called with small probability (say 10^{-3}). In contrast, a function aiming at 1 ulp accuracy can deliver only about p correct bits;
- the rounding test costs a few cycles. For a correctly rounded function, this test is required to decide whether the accurate path is needed.

With clever algorithms and coding, this theoretical overhead can be made quite small. However, starting from a correctly rounded function, if one disables the rounding test and returns the value of the fast path, one will always get a faster routine. By the way, if one wishes to implement two versions for each function, a correctly rounded one and a "fast" one – for instance for high-throughput vector calculations, this disabling might well be the right solution.

The case of vector functions is more complicated. If the same function is evaluated in parallel for m different inputs, even if only one of these inputs requires the use of the accurate path, the calculation is slowed down for all m inputs. We are not aware of any publications in the literature on correctly rounded vector functions. The solution will probably be to build a fast path that is significantly more accurate than for scalar functions, making the probability of having to use the accurate path much smaller. This small probability will also have a benefit in terms of control: if the same function is called many times in a given application, then the branch prediction will be very efficient, so that the cost of the rounding test will be negligible. And, of course, if the function is called infrequently, its performance is of little concern.

Another important point is the memory cost. Indeed, especially for functions that do not benefit from argument reduction, the accurate step requires precomputed tables that might take a lot of memory. However, there is a time-memory trade-off: either use large tables and low-degree polynomials, or small tables and high-degree polynomials. The goal of the MetaLibm [51] project was precisely to automatically generate mathematical routines for a wide range of targets: general-purpose processors, embedded processors, graphical processing units, and even FPGA and digital circuits.

The cost of maintaining and debugging the libraries is also to be considered. As performance essentially depends on the first approximation, the largest effort is on that first approximation. Hence, correctly rounded routines are not much more complex to maintain than other libraries. And having a clear specification (not a fuzzy one of the kind "the returned value is not too far from the exact result") helps detecting and correcting bugs (either in the code or in the error analysis).

6 SOME LIMITATIONS

In this section, we discuss possible restrictions to the requirement of correct rounding which might offer a better compromise between what is desirable and what is achievable at a reasonable cost. Some of these restrictions can be lifted when progress is made on the computation of bounds on the hardness to round in these settings.

In the following first three cases, one may consider requiring "reasonably fast" correct rounding in a limited domain only, and outside of that domain, either relax the correct rounding requirement, or accept a significant loss of performance using a strategy of the kind *while we are not able to correctly round the result, increase the intermediate precision with which the function is approximated.* This is Ziv's total strategy, presented in Section 4.1. We can also decide to adopt Ziv's partial strategy (see Section 4.1), stopping the iterative precision increase at some point and printing a warning or raising an exception.

6.1 The case of binary128 functions.

The currently high cost of even simply deriving upper bounds for the hardness-to-round in binary128, joined with the larger exponent range, suggests that one may want to restrict the requirement of correct rounding to a limited number of binades with exponents around 0.

6.2 The case of bivariate functions.

For x^y, $\arctan(y/x)$, $\arctan(y/x)/\pi$ or $\mathrm{hypot}(x, y)$, the situation for a given format is similar to that for a univariate function and a format twice as large. The same restriction should thus be considered. This restriction may be asymmetrical, for instance for x^y where we may want to restrict more strongly the range on y than on x.

6.3 The case of trigonometric functions.

A difficult issue is trigonometric functions of very large arguments in binary64 arithmetic (or wider formats). First, one may argue that, as the input values result in general from earlier calculations or measurements, when the input is large the absolute error on that input may be non negligible in front of π. In such a case, the calculated output may be anyway very far from the exact result. Second, as we have seen, in binary64 arithmetic and in wider formats, the cost of computing hardest to round values for sines or cosines of huge arguments (at the time we are writing these lines, they are known in binary64 for arguments less than 2^{11}) is significantly larger than for small arguments.

6.4 Violation of range constraints

Sometimes correct rounding is incompatible with range constraints: for example in double-extended precision ($p = 64$), the correctly rounded arcsine of 1 is `0x1.921fb54442d1846ap+0`, which is larger than $\pi/2$. Returning a value that violates the mathematical property $|\mathrm{asin}\, x| \leqslant \pi/2$ may in some cases be inappropriate. For the (very few!) functions and formats for which such events may happen, we might consider an optional "range takes over correct rounding" behavior.

7 EXISTING CORRECTLY ROUNDED IMPLEMENTATIONS

We list here implementations yielding correct rounding of mathematical functions for the IEEE 754 binary formats. For each implementation we mention whether it is still maintained or not, which IEEE formats and rounding functions it supports, which functions it provides (as of February 2025), and whether the algorithms are documented. Unless stated otherwise, when the algorithms are documented, correct rounding is obtained with Ziv's strategy (see Section 4.1).

A detailed assessment of the accuracy of various libraries is given in [37]. This reference is regularly updated.

MathLib. Also called LibUltim, it is a library developed by IBM around 1990. It provides the following binary64 functions: acos, asin, atan, atan2, exp, exp2, log, log2, cos, sin, tan, cot, and pow. It only supports rounding to nearest-even. Some high-level algorithms are described in [88]. MathLib is no longer maintained, but it was integrated into GNU libc version 2.27 (2018), except for acos, exp2, log2 and cot. After GNU libc 2.27, the accurate path was removed by the GNU libc developers because it was too slow for some corner-case inputs (for example up to 440,000 cycles for the binary64 power function, using 768-bit arithmetic). No known bug exists. An unofficial copy is available at https://github.com/dreal-deps/mathlib.

LIBMCR. It was developed by Sun Microsystems until 2004. It also targets binary64 only and rounding to nearest-even. It provides the following functions: exp, log, pow, atan, sin, cos, and tan. Algorithms are not detailed. Some tests with the power function reveal several issues [41]: first, for

some inputs it does not terminate; secondly, for some inputs, it gives a result which is far from the correct one. A non-official copy is available from https://github.com/simonbyrne/libmcr.

CRLibm. It was developed by the Arénaire team in the LIP Laboratory (Lyon, France) until 2006 [21]. It provides the following binary64 functions: exp, expm1, log, log1p, log2, log10, sin, cos, tan, asin, acos, atan, sinh, cosh, sinpi, cospi, tanpi, atanpi, and (with restrictions) pow. For each function, there are four entry points corresponding to the four IEEE 754 rounding functions (at that time rounding to nearest ties-to-away was not yet standardized). For example for the exponential function: `exp_rn`, `exp_rz`, `exp_ru`, `exp_rd` for rounding to nearest ties-to-even, towards zero, towards $+\infty$ and towards $-\infty$ respectively. CRLibm assumes the rounding precision is set to binary64, and the processor rounding function is set to nearest ties-to-even. CRLibm makes use of modern instructions such as the fused-multiply add (FMA), it benefits from the knowledge of hardest-to-round cases, and thus has a better tuning of the accurate path, which uses triple-double arithmetic [32, 53]. For example, for the binary64 exp function, [21] reports a maximum/average time ratio of 6500 for MathLib, compared to only 6.6 for CRLibm. Although CRLibm is no longer maintained, an unofficial copy is available from https://github.com/taschini/crlibm.

RLIBM. It is developed by the group of Santosh Nagarakatte (Rutgers University). As of February 2025, it provides 16 binary32 functions: acos, asin, atan, cos, cosh, cospi, exp, exp10, exp2, log, log10, log2, sin, sinh, sinpi, and tan. It supports all IEEE rounding functions. Additionally, it also provides routines for posits. One originality of RLIBM is that it uses a new approach based on linear programming, to find polynomials that yield correct rounding [59]. Another interesting fact is that RLIBM uses a correctly rounded result for a 34-bit significant and the "round-to-odd" rounding mode, which when rounded to a smaller precision $p \leqslant 32$ and with another rounding mode always produces the correctly rounded result [60]. However, it is not clear whether that new approach scales for larger precisions. The code and an extensive bibliography are available from https://people.cs.rutgers.edu/~sn349/rlibm/.

LLVM libc. It is the C library that comes with the LLVM compiler, supported by Google. It contains a mathematical library, whose aim is to provide only correctly rounded functions, for all IEEE rounding functions. As of February 2025, it provides all binary32 functions from the C99 standard except erfc, a few binary64 functions (cbrt, cos, exp, exp2, exp10, expm1, hypot, log, log10, log1p, log2, sin, sincos, and tan), and also some functions for the binary16 format. For some binary32 functions, LLVM uses polynomials generated by the RLIBM developers. A few performance comparisons with other libraries are presented in Table 5 (more figures can be found at https://core-math.gitlabpages.inria.fr). Except for the binary64 hypot function, its efficiency is within a factor of two of the GNU libc and/or the Intel library. The code is available from https://libc.llvm.org/.

CORE-MATH. It is not a real mathematical library, but rather a collection of standalone correctly rounded routines that can be integrated into mathematical libraries, or used directly in specific applications. As of February 2025, it provides all binary32 and binary64 functions from the C23 standard except compound and lgamma, and a few functions for double-extended precision (cbrt, exp, exp2, log2, pow, rsqrt). CORE-MATH supports all four IEEE rounding functions available in the C language. Algorithms are detailed either as comments in the source code, or as scientific publications [41]. The implementation follows the strategy described in §4.5. The code and an extensive bibliography are available from https://core-math.gitlabpages.inria.fr/, where one can also find a comparison of the efficiency with respect to other libraries. The CORE-MATH code also contains large tables of hard-to-round inputs, generated using the BaCSeL software tool [40]. These tables are used to check the correctness of the CORE-MATH routines, but can also be used to check

Table 5. Reciprocal throughput (average value) of some binary64 functions of CORE-MATH (revision 95d99b4) with the GNU Libc (version 2.40), Intel Math (from icx 2025.0.0), and LLVM (revision 22687aa) libraries, on an Intel Xeon Silver 4214. Figures in italics are not correctly rounded (including pow for LLVM). The CORE-MATH code is compiled with gcc 14.2.0, the Intel Math Library with the Intel compiler, and the LLVM code with clang 19.1.6.

Function	CORE-MATH	GNU libc	Intel Math Library	LLVM
cos	57.6	*40.4*	*15.4*	32.5
exp	15.5	*11.6*	*9.4*	18.5
erf	35.5	*50.1*	*17.3*	
log1p	21.7	*21.8*	*17.4*	29.9
pow	51.4	*38.8*	*32.3*	*34.1*
hypot	21.4	*29.6*	*22.7*	209.9

the correctness of other libraries. Like LLVM libc, the efficiency of CORE-MATH is within a factor of two of the GNU libc and/or the Intel library, and for some functions it is even faster (see Table 5).

As of February 2025, the maintained and widespread correctly rounded libraries are RLIBM, LLVM libc and CORE-MATH. Part of CORE-MATH is already integrated into mathematical libraries: the binary32 tanh function into the AMD library, and the binary32 acos, acosh, asin, asinh, atan, atan2, atanh, cbrt, cosh, erf, erfc, expm1, exp2m1, exp10m1, tgamma, lgamma, log10, log1p, log2p1, log10p1, sinh, tan, tanh functions into GNU libc.

GNU MPFR. Though it is not focused on IEEE formats, we also mention the MPFR library [34] because it has helped popularize correct-rounding evaluation and it is also a useful verification tool for the libraries mentioned above. GNU MPFR is an arbitrary precision library with correct rounding. To emulate some IEEE 754 format, one has to use the corresponding precision, and so set the corresponding exponent range. MPFR does not have subnormal numbers, but they can be emulated with the `mpfr_subnormalize` function.

CONCLUSION AND RECOMMENDATION

Knowledge of the worst cases for the common univariate functions in binary64 arithmetic, as well as the existence of efficient correctly rounded implementations for these functions, leads us to believe that there are no longer reasons not to require correct rounding of these functions in the binary16, binary32, and binary64 formats. We understand that high-throughput vector computations may be penalized by the tests required when running correctly rounded function programs. Therefore, we suggest that, along with the correctly rounded version, a faster version of the functions may be available. Some vendors might be reluctant to accept a (very small, as we have seen) loss in performance and ease of implementation for a benefit that may not seem clear at first glance. Similar objections were raised early in the discussions leading up to IEEE 754-1985, and no one now seriously doubts the obvious interest in having well-specified additions and multiplications. The use of formal methods to validate critical software is growing rapidly, and this justifies by itself the need for fully specified functions. With a little pedagogy and explanatory skill, providing "ultimate" functions at a reasonable cost could be a very good selling point.

Our suggestion is therefore:

- In the binary16 and binary32 formats, correctly rounded implementations of the functions

$$e^x, e^x - 1, 2^x, 2^x - 1, 10^x, 10^x - 1,$$
$$\log(x), \log_2(x), \log_{10}(x), \log(1+x), \log_2(1+x), \log_{10}(1+x),$$
$$1/\sqrt{x}, \sin(\pi x), \cos(\pi x), \tan(\pi x), \arcsin(x)/\pi, \arccos(x)/\pi, \arctan(x)/\pi,$$
$$\sin(x), \cos(x), \tan(x), \arcsin(x), \arccos(x), \arctan(x),$$
$$\sinh(x), \cosh(x), \tanh(x), \operatorname{arcsinh}(x), \operatorname{arccosh}(x), \operatorname{arctanh}(x)$$
$$\sqrt{x^2+y^2}, x^y, \arctan(y/x)/\pi, \arctan(y/x)$$

  shall be provided. Along with these correctly rounded implementations, other implementations may be provided.
- In the binary64 format, correctly rounded implementations of the univariate functions

$$e^x, e^x - 1, 2^x, 2^x - 1, 10^x, 10^x - 1,$$
$$\log(x), \log_2(x), \log_{10}(x), \log(1+x), \log_2(1+x), \log_{10}(1+x),$$
$$1/\sqrt{x}, \sin(\pi x), \cos(\pi x), \tan(\pi x), \arcsin(x)/\pi, \arccos(x)/\pi, \arctan(x)/\pi,$$
$$\sin(x), \cos(x), \tan(x), \arcsin(x), \arccos(x), \arctan(x),$$
$$\sinh(x), \cosh(x), \tanh(x), \operatorname{arcsinh}(x), \operatorname{arccosh}(x), \operatorname{arctanh}(x)$$

  shall be provided (at least, for the sine and cosine functions, for $|x| \leqslant 2^{11}$, and for the tan function, for $|x| \leqslant 10.5\pi$). Along with these correctly rounded implementations, other implementations may be provided.

In all other cases (decimal formats, binary128, bivariate functions in binary64), correct rounding is still desirable, but since the worst cases are not known, we can only use bounds on the hardness-to-round, obtained using the techniques presented in this article. This may lead to implementations that are slower in (hopefully extremely rare!) bad cases. We therefore suggest that for these formats and functions, correct rounding should be recommended, but not required. This gives:

- In all other cases, correct rounding of the functions

$$e^x, e^x - 1, 2^x, 2^x - 1, 10^x, 10^x - 1,$$
$$\log(x), \log_2(x), \log_{10}(x), \log(1+x), \log_2(1+x), \log_{10}(1+x),$$
$$1/\sqrt{x}, \sin(\pi x), \cos(\pi x), \tan(\pi x), \arcsin(x)/\pi, \arccos(x)/\pi, \arctan(x)/\pi,$$
$$\sin(x), \cos(x), \tan(x), \arcsin(x), \arccos(x), \arctan(x),$$
$$\sinh(x), \cosh(x), \tanh(x), \operatorname{arcsinh}(x), \operatorname{arccosh}(x), \operatorname{arctanh}(x)$$
$$\sqrt{x^2+y^2}, x^y, \arctan(y/x)/\pi, \arctan(y/x)$$

  should be provided. Along with these correctly rounded implementations, other implementations may be provided.

ACKNOWLEDGMENTS

We wish to thank the anonymous referees for their very helpful comments and suggestions. Many colleagues and students contributed over the years to these questions, by working on the theory of the table maker's dilemma, pointing out relevant mathematical results, writing software for solving the TMD or writing function programs, running tests, writing formal proofs, reading preliminary versions of this text, or giving advice. We are especially grateful to Sylvie Boldo, Sylvain Chevillard, Sélène Corbineau, Catherine Daramy, David Defour, Florent de Dinechin, Pierre Fortin, Mourad Gouicem, Stef Graillat, Tom Hubrecht, Claude-Pierre Jeannerod, Mioara Joldes, Tomas Lang, Christoph Lauter, Vincent Lefèvre, Erik Martin-Dorel, Micaela Mayero, Guillaume

Melquiond, Laurence Rideau, Mike Schulte, Alexei Sibidanov, Damien Stehlé, Earl Swartzlander, Peter Tang, Laurent Théry, Arnaud Tisserand, Serge Torres, and Michel Waldschmidt.

REFERENCES

[1] D. H. Ahn, A. H. Baker, M. Bentley, I. Briggs, G. Gopalakrishnan, D. M. Hammerling, I. Laguna, G. L. Lee, D. J. Milroy, and M. Vertenstein. 2021. Keeping Science on Keel When Software Moves. *Commun. ACM* 64, 2 (2021), 66–74. https://doi.org/10.1145/3382037

[2] P. Ahrens, J. Demmel, and H. D. Nguyen. 2020. Algorithms for Efficient Reproducible Floating Point Summation. *ACM Trans. Math. Software* 46, 3, Article 22 (2020), 49 pages. https://doi.org/10.1145/3389360

[3] N. H. Beebe. 2017. *The Mathematical-Function Computation Handbook.* Springer.

[4] V. Berthé and L. Imbert. 2009. Diophantine Approximation, Ostrowski Numeration and the Double-Base Number System. *Discrete Math. Theor. Comput. Sci.* 11, 1 (2009), 153–172. http://dmtcs.episciences.org/450

[5] C. M. Black, R. P. Burton, and T. H. Miller. 1984. The Need for an Industry Standard of Accuracy for Elementary-Function Programs. *ACM Trans. Math. Software* 10, 4 (1984), 361–366.

[6] S. Boldo, M. Daumas, and R.-C. Li. 2009. Formally Verified Argument Reduction with a Fused Multiply-Add. *IEEE Trans. Comput.* 58, 8 (2009), 1139–1145. https://doi.org/10.1109/TC.2008.216

[7] S. Boldo, C.-P. Jeannerod, G. Melquiond, and J.-M. Muller. 2023. Floating-Point Arithmetic. *Acta Numer.* 32 (2023), 203–290. https://doi.org/10.1017/S0962492922000101

[8] N. Brisebarre and G. Hanrot. 2023. Integer points close to a transcendental curve and correctly-rounded evaluation of a function. (2023). https://hal.science/hal-03240179 Working paper or preprint.

[9] N. Brisebarre, G. Hanrot, and O. Robert. 2017. Exponential Sums and Correctly-Rounded Functions. *IEEE Trans. Comput.* 66, 12 (2017), 2044–2057. https://doi.org/10.1109/TC.2017.2690850

[10] N. Brisebarre and J.-M. Muller. 2007. Correct rounding of algebraic functions. *Theor. Inform. Appl.* 41, 1 (2007), 71–83. https://doi.org/10.1051/ita:2007002

[11] N. Brisebarre and J.-M. Muller. 2008. Correctly rounded multiplication by arbitrary precision constants. *IEEE Trans. Comput.* 57, 2 (2008), 165–174. https://doi.org/10.1109/TC.2007.70813

[12] C Working Group. 2024. *Programming Languages – C, Working draft.* Available at https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3435.pdf.

[13] S. Chevillard, M. Joldeş, and C. Lauter. 2010. Sollya: An environment for the development of numerical codes. In *International Congress on Mathematical Software*. Springer, 28–31.

[14] W. Cody and W. Waite. 1980. *Software Manual for the Elementary Functions.* Prentice-Hall, Englewood Cliffs, NJ.

[15] W. J. Cody. 1971. Software for the Elementary Functions. In *Mathematical Software*, John R. Rice (Ed.). Academic Press, 171–186.

[16] W. J. Cody. 1980. Implementation and Testing of Function Software. In *Problems and Methodologies in Mathematical Software Production, International Seminar*. Springer-Verlag, Berlin, Heidelberg, 24–47.

[17] D. Coppersmith. 1997. Small Solutions to Polynomial Equations, and Low Exponent RSA Vulnerabilities. *J. Cryptology* 10, 4 (1997), 233–260. https://doi.org/10.1007/s001459900030

[18] D. Coppersmith. 2001. Finding Small Solutions to Small Degree Polynomials. In *Proceedings of Cryptography and Lattices (CaLC) (Lecture Notes in Computer Science)*, J. H. Silverman (Ed.), Vol. 2146. Springer-Verlag, Berlin, 20–31.

[19] M. F. Cowlishaw. 2003. Decimal Floating-Point: algorism for Computers. In *Proceedings of the 16th IEEE Symposium on Computer Arithmetic (ARITH-16)*, Bajard and Schulte (Eds.). IEEE Computer Society Press, Los Alamitos, CA, 104–111.

[20] C. Daramy, D. Defour, F. de Dinechin, and J.-M. Muller. 2003. CR-LIBM, a Correctly Rounded Elementary Function Library. In *SPIE 48th Annual Meeting International Symposium on Optical Science and Technology*.

[21] C. Daramy-Loirat, D. Defour, F. de Dinechin, M. Gallet, N. Gast, C. Q. Lauter, and J.-M. Muller. 2006. *CR-LIBM, A library of correctly-rounded elementary functions in double-precision.* Technical Report. LIP Laboratory, Arenaire team, Available at https://ens-lyon.hal.science/ensl-01529804/file/crlibm.pdf.

[22] F. de Dinechin, A. V. Ershov, and N. Gast. 2005. Towards the Post-Ultimate libm. In *Proceedings of the 17th IEEE Symposium on Computer Arithmetic (ARITH'05)*. IEEE Computer Society, Washington, DC, USA, 288–295. https://doi.org/10.1109/ARITH.2005.46

[23] F. de Dinechin, C. Lauter, and G. Melquiond. 2011. Certifying the Floating-point Implementation of an Elementary Function Using Gappa. *IEEE Trans. Comput.* 60, 2 (2011), 242–253. https://doi.org/10.1109/TC.2010.128

[24] F. de Dinechin, C. Lauter, and J.-M. Muller. 2007. Fast and correctly rounded logarithms in double-precision. *Theor. Inform. Appl.* 41, 1 (2007), 85–102. https://doi.org/10.1051/ita:2007003

[25] F. de Dinechin, C. Lauter, J.-M. Muller, and S. Torres. 2013. On Ziv's Rounding Test. *ACM Trans. Math. Software* 39, 4, Article 25 (2013), 19 pages. https://doi.org/10.1145/2491491.2491495

[26] F. de Dinechin, J.-M. Muller, B. Pasca, and A. Plesco. 2011. An FPGA architecture for solving the Table Maker's Dilemma. In *Application-Specific Systems, Architectures and Processors (ASAP), 2011 IEEE International Conference on*. IEEE Computer Society, Santa Monica, United States, 187–194. https://doi.org/10.1109/ASAP.2011.6043267

[27] P. G. de Lamarlière, G. Melquiond, and F. Faissole. 2023. Slimmer Formal Proofs for Mathematical Libraries. In *2023 IEEE 30th Symposium on Computer Arithmetic (ARITH)*. 32–35. https://doi.org/10.1109/ARITH58626.2023.00026

[28] D. Defour. 2003. *Fonctions élémentaires: algorithmes et implémentations efficaces pour l'arrondi correct en double précision*. Ph.D. Dissertation. École normale supérieure de Lyon, Lyon, France. http://www.ens-lyon.fr/LIP/Pub/Rapports/PhD/PhD2003/PhD2003-01.ps.gz

[29] D. Defour and J. Muller. 2001. Correctly Rounded Exponential Function in Double Precision Arithmetic. In *SPIE, 46th Annual Meeting International Symposium on Optical Science and Technology*.

[30] C. B. Dunham. 1990. Feasibility of "Perfect" Function Evaluation. *SIGNUM Newsletter* 25, 4 (1990), 25–26.

[31] G. Estrin. 1960. Organization of computer systems – the fixed plus variable structure computer. In *Proceedings Western Joint Computing Conference 17*. 33–40.

[32] N. Fabiano, J.-M. Muller, and J. Picot. 2019. Algorithms for Triple-Word Arithmetic. *IEEE Trans. Comput.* 68, 11 (2019), 1573–1583. https://doi.org/10.1109/TC.2019.2918451

[33] N. I. Fel'dman. 1971. An effective power sharpening of a theorem of Liouville. *Izv. Akad. Nauk SSSR Ser. Mat.* 35 (1971), 973–990.

[34] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann. 2007. MPFR: A Multiple-Precision Binary Floating-Point Library with Correct Rounding. *ACM Trans. Math. Software* 33, 2 (2007). https://doi.org/10.1145/1236463.1236468

[35] S. Gal and B. Bachelis. 1991. An Accurate Elementary Mathematical Library for the IEEE Floating Point Standard. *ACM Trans. Math. Software* 17, 1 (1991), 26–45.

[36] A. Gil, J. Segura, and N. M. Temme. 2011. Basic Methods for Computing Special Functions. In *Recent Advances in Computational and Applied Mathematics*, Theodore E. Simos (Ed.). Springer Netherlands, Dordrecht, 67–121.

[37] B. Gladman, V. Innocente, J. Mather, and P. Zimmermann. 2025. Accuracy of Mathematical Functions in Single, Double, Extended Double and Quadruple Precision. (2025). Working paper or preprint, available at https://hal.inria.fr/hal-03141101.

[38] D. Goldberg. 1991. What Every Computer Scientist Should Know About Floating-Point Arithmetic. *Comput. Surveys* 23 (1991), 5–48.

[39] G. Hanrot, V. Lefèvre, D. Stehlé, and P. Zimmermann. 2007. Worst Cases of a Periodic Function for Large Arguments. In *18th IEEE Symposium on Computer Arithmetic (ARITH-18)*. IEEE Computer Society, 133–140. https://doi.org/10.1109/ARITH.2007.37

[40] G. Hanrot, V. Lefèvre, D. Stehlé, and P. Zimmermann. 2020. BaCSeL. https://gitlab.inria.fr/zimmerma/bacsel. (2020). Version 4.0.

[41] T. Hubrecht, C.-P. Jeannerod, and P. Zimmermann. 2023. Towards a correctly-rounded and fast power function in binary64 arithmetic. In *2023 IEEE 30th Symposium on Computer Arithmetic (ARITH 2023)*. Portland, Oregon (USA), United States. https://inria.hal.science/hal-04326201

[42] IEEE. 2019. *IEEE Standard for Floating-Point Arithmetic (IEEE Std 754-2019)*. 84 pages. https://doi.org/10.1109/IEEESTD.2019.8766229

[43] C. Iordache and D. W. Matula. 1999. On Infinitely Precise Rounding for Division, Square Root, Reciprocal and Square Root Reciprocal. In *Proceedings of the 14th IEEE Symposium on Computer Arithmetic (Adelaide, Australia)*, Koren and Kornerup (Eds.). IEEE Computer Society Press, Los Alamitos, CA, 233–240.

[44] M. Joldes, J. Muller, and V. Popescu. 2017. Tight and Rigorous Error Bounds for Basic Building Blocks of Double-Word Arithmetic. *ACM Trans. Math. Software* 44, 2 (2017). https://doi.org/10.1145/3121432

[45] W. Kahan. 1981. *Why do we Need a Floating-Point Standard?* Technical Report. Computer Science, UC Berkeley. Available at https://www.cs.berkeley.edu/~wkahan/ieee754status/why-ieee.pdf.

[46] W. Kahan. 1983. Minimizing $qm - n$. Text accessible electronically at https://people.eecs.berkeley.edu/~wkahan/testpi/nearpi.c. (1983). At the beginning of the file "nearpi.c".

[47] W. Kahan. 2004. A Logarithm Too Clever by Half. (2004). Available at http://http.cs.berkeley.edu/~wkahan/LOG10HAF.TXT.

[48] S. Khémira and P. Voutier. 2011. Approximation diophantienne et approximants de Hermite-Padé de type I de fonctions exponentielles. *Ann. Sci. Math. Québec* 35, 1 (2011), 85–116.

[49] S. Khémira. 2005. *Approximants de Hermite-Padé, déterminants d'interpolation et approximation diophantienne*. Ph.D. Dissertation. Université Paris 6, Paris, France. https://tel.archives-ouvertes.fr/tel-00657843

[50] P. Kornerup, C. Lauter, V. Lefèvre, N. Louvet, and J.-M. Muller. 2010. Computing correctly rounded integer powers in floating-point arithmetic. *ACM Trans. Math. Software* 37, 1 (2010). https://doi.org/10.1145/1644001.1644005

[51] O. Kupriianova and C. Lauter. 2014. Metalibm: A Mathematical Functions Code Generator. In *Mathematical Software – ICMS 2014*, Hoon Hong and Chee Yap (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 713–717.

[52] T. Lang and J.-M. Muller. 2001. Bound on Run of Zeros and Ones for Algebraic Functions. In *Proceedings of the 15th IEEE Symposium on Computer Arithmetic (ARITH-16)*, N. Burgess and L. Ciminiera (Eds.). 13–20.

[53] C. Q. Lauter. 2008. *Arrondi Correct de Fonctions Mathématiques.* Ph.D. Dissertation. École Normale Supérieure de Lyon, Lyon, France. https://www.christoph-lauter.org/these.pdf

[54] C. Q. Lauter and V. Lefèvre. 2009. An Efficient Rounding Boundary Test for $\mathrm{pow}(x, y)$ in Double Precision. *IEEE Trans. Comput.* 58, 2 (2009), 197–207.

[55] V. Lefèvre. 2000. *Moyens Arithmétiques Pour un Calcul Fiable.* Ph.D. Dissertation. École Normale Supérieure de Lyon, Lyon, France.

[56] V. Lefèvre. 2005. New Results on the Distance Between a Segment and $\mathbb{Z}^2$. Application to the Exact Rounding. In *Proceedings of the 17th IEEE Symposium on Computer Arithmetic (ARITH-17)*. IEEE Computer Society Press, Los Alamitos, CA, 68–75.

[57] V. Lefèvre and J.-M. Muller. 2001. Worst Cases for Correct Rounding of the Elementary Functions in Double Precision. In *Proceedings of the 15th IEEE Symposium on Computer Arithmetic (ARITH-16)*, N. Burgess and L. Ciminiera (Eds.). Vail, CO, 111–118. https://doi.org/10.1109/ARITH.2001.930110

[58] V. Lefèvre, J.-M. Muller, and A. Tisserand. 1998. Toward Correctly Rounded Transcendentals. *IEEE Trans. Comput.* 47, 11 (1998), 1235–1243.

[59] J. P. Lim and S. Nagarakatte. 2021. 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*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 359–374. https://doi.org/10.1145/3453483.3454049

[60] J. P. Lim and S. Nagarakatte. 2022. One polynomial approximation to produce correctly rounded results of an elementary function for multiple representations and rounding modes. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–28. https://doi.org/10.1145/3498664

[61] S. Linnainmaa. 1981. Software for Doubled-Precision Floating-Point Computations. *ACM Trans. Math. Software* 7, 3 (1981), 272–283. https://doi.org/10.1145/355958.355960

[62] J. Liouville. 1844. Nouvelle démonstration d'un théorème sur les irrationnelles algébriques. *C. R. Acad. Sci. Paris* 18 (1844), 910–911.

[63] J. Liouville. 1844. Remarques relatives à des classes très-étendues de quantités dont la valeur n'est ni algébrique ni même réductible à des irrationnelles algébriques. *C. R. Acad. Sci. Paris* 18 (1844), 883–885.

[64] J. Liouville. 1851. Sur des classes très étendues de quantités dont la valeur n'est ni algébrique ni même réductible à des irrationnelles algébriques. *J. Math. Pures Appl.* 16 (1851), 133–142.

[65] J. L. Maire, N. Brunie, F. de Dinechin, and J. Muller. 2016. Computing floating-point logarithms with fixed-point operations. In *23nd IEEE Symposium on Computer Arithmetic, ARITH 2016, Silicon Valley, CA, USA, July 10-13, 2016*, Paolo Montuschi, Michael J. Schulte, Javier Hormigo, Stuart F. Oberman, and Nathalie Revol (Eds.). IEEE Computer Society, 156–163. https://doi.org/10.1109/ARITH.2016.24

[66] P. Markstein. 2000. *IA-64 and Elementary Functions: Speed and Precision.* Prentice-Hall, Englewood Cliffs, NJ.

[67] É. Martin-Dorel, G. Hanrot, M. Mayero, and L. Théry. 2015. Formally Verified Certificate Checkers for Hardest-to-Round Computation. *J. Autom. Reason.* 54, 1 (2015), 1–29. https://doi.org/10.1007/s10817-014-9312-2

[68] J. D. Mooney. 2004. Developing Portable Software. In *Information Technology*, Ricardo Reis (Ed.). Springer US, Boston, MA, 55–84.

[69] J. Muller and L. Rideau. 2022. Formalization of Double-Word Arithmetic, and Comments on "Tight and Rigorous Error Bounds for Basic Building Blocks of Double-Word Arithmetic". *ACM Trans. Math. Software* 48, 1 (2022), 9:1–9:24. https://doi.org/10.1145/3484514

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

[71] J.-M. Muller, N. Brisebarre, F. de Dinechin, C.-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, and S. Torres. 2010. *Handbook of Floating-Point Arithmetic.* Birkhäuser. 572 pages.

[72] J.-M. Muller, N. Brunie, F. de Dinechin, C.-P. Jeannerod, M. Joldeş, V. Lefèvre, G. Melquiond, N. Revol, and S. Torres. 2018. *Handbook of Floating-Point Arithmetic (2nd Edition).* Birkhäuser. 627 pages.

[73] Y. V. Nesterenko and M. Waldschmidt. 1996. On the approximation of the values of exponential function and logarithm by algebraic numbers (in Russian). *Mat. Zapiski* 2 (1996), 23–42. Available in English at https://arxiv.org/abs/math/0002047.

[74] M. L. Overton. 2001. *Numerical Computing with IEEE Floating-Point Arithmetic.* SIAM, Philadelphia, PA.

[75] G. Paul and M. W. Wilson. 1976. Should the elementary function library be incorporated into computer instruction sets? *ACM Trans. Math. Software* 2, 2 (1976), 132–142.

[76] D. Piparo and V. Innocente. 2016. The CptnHook Profiler - A tool to investigate usage patterns of mathematical functions. *Journal of Physics: Conference Series* 762, 1 (2016), 012038. https://doi.org/10.1088/1742-6596/762/1/012038

[77] D. Ridout. 1957. Rational approximations to algebraic numbers. *Mathematika* 4 (1957), 125–131. https://doi.org/10.1112/S0025579300001182

[78] K. F. Roth. 1955. Rational Approximations to Algebraic Numbers. *Mathematika* 2 (1955), 1–20.

[79] S. M. Rump. 2010. Verification methods: Rigorous results using floating-point arithmetic. *Acta Numer.* 19 (2010), 287–449. https://doi.org/10.1017/S096249291000005X

[80] M. J. Schulte and E. E. Swartzlander. 1993. Exact rounding of certain elementary functions. In *Proceedings of the 11th IEEE Symposium on Computer Arithmetic*, E. E. Swartzlander, M. J. Irwin, and G. Jullien (Eds.). IEEE Computer Society Press, Los Alamitos, CA, 138–145.

[81] A. Sibidanov, P. Zimmermann, and S. Glondu. 2022. The CORE-MATH Project. In *29th IEEE Symposium on Computer Arithmetic*. Preprint available at https://hal.inria.fr/hal-03721525.

[82] R. A. Smith. 1995. A Continued-Fraction Analysis of Trigonometric Argument Reduction. *IEEE Trans. Comput.* 44, 11 (1995), 1348–1351.

[83] D. Stehlé. 2006. On the Randomness of Bits Generated by Sufficiently Smooth Functions. In *Proceedings of the 7th Algorithmic Number Theory Symposium, ANTS VII (Lecture Notes in Computer Science)*, F. Hess, S. Pauli, and M. E. Pohst (Eds.), Vol. 4076. Springer-Verlag, Berlin, 257–274.

[84] D. Stehlé, V. Lefèvre, and P. Zimmermann. 2005. Searching Worst Cases of a One-Variable Function Using Lattice Reduction. *IEEE Trans. Comput.* 54, 3 (2005), 340–346.

[85] S. Torres. 2016. *Tools for the Design of Reliable and Efficient Functions Evaluation Libraries.* Ph.D. Dissertation. École normale supérieure de Lyon – Université de Lyon, Lyon, France. https://tel.archives-ouvertes.fr/tel-01396907

[86] L. Turelier. 2022. *Extension of the SLZ algorithm to bivariate functions.* Research Report. INRIA Nancy. https://inria.hal.science/hal-03740209

[87] M. Waldschmidt. 2000. *Diophantine approximation on linear algebraic groups.* Grundlehren der Mathematischen Wissenschaften [Fundamental Principles of Mathematical Sciences], Vol. 326. Springer-Verlag, Berlin. xxiv+633 pages. https://doi.org/10.1007/978-3-662-11569-5 Transcendence properties of the exponential function in several variables.

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

A TRANSCENDENCE RESULTS

Khémira-Voutier's theorem for exp

The exponential function is central to the study of the correctly rounded evaluation of the elementary functions by `libms`: relevant information about its hardness to round yields relevant information as well about the trigonometric and hyperbolic functions and their respective reciprocals (see [71, §12.4.4], Remark 4.5), the logarithm function and the inverse trigonometric functions.

Following the works [73] and [49], Khémira and Voutier proved in [48] a lower bound (called *transcendence measure*) for the expression $|e^{\alpha} - \beta|$, where α and β are algebraic numbers, $\alpha \neq 0$. When specialized in FP numbers, their result provides interesting upper bounds for $\mathrm{htr}_{\exp}(p)$.

Furthermore, the strong relationship between exp and many functions such as log, the trigonometric, hyperbolic, inverse trigonometric and inverse hyperbolic functions allows one to deduce from their Theorem statements that apply to all of these functions.

Let m and $n \in \mathbb{N}$, we introduce the quantities

$$d_n = \text{l.c.m.}(1, \ldots, n) \text{ and } \mathcal{D}_{m,n} = \frac{m!}{\prod\limits_{\substack{q \leqslant n, \\ q \text{ prime}}} q^{v_q(m!)}},$$

where $v_q(m!)$ is the the largest integer k such that q^k divides $m!$.

Let α be an algebraic number of degree d over $\mathbb{Q}$, so that the minimal polynomial over $\mathbb{Z}$ is written $a \prod_{i=1}^{d} (X - \alpha^{(i)})$, the roots $\alpha^{(i)}$ being complex numbers. We denote by

$$h(\alpha) = \frac{1}{d} \left(\log |a| + \sum_{i=1}^{d} \log \max \left(1, \left| \alpha^{(i)} \right| \right) \right) \tag{A.1}$$

the absolute logarithmic Weil height of the algebraic number α.

We now state Khémira and Voutier's Theorem in its full generality.

Theorem A.1 (Theorem 1.1 from [48]). *Let α and $\beta \in \mathbb{C}$ be two algebraic numbers, $\alpha \neq 0$. Define $D_{\alpha,\beta} = [\mathbb{Q}(\alpha, \beta) : \mathbb{Q}]/[\mathbb{R}(\alpha, \beta) : \mathbb{R}]$, $\mathcal{A}_{\alpha,\beta} \geqslant 1, \mathcal{B}_{\alpha,\beta} \geqslant 1$ be such that*

$$\begin{aligned} D_{\alpha,\beta} h(\alpha) - \log(\max(1, |\alpha|)) &\leqslant \log \mathcal{A}_{\alpha,\beta}, \\ D_{\alpha,\beta} h(\beta) - \log(\max(1, |\beta|)) &\leqslant \log \mathcal{B}_{\alpha,\beta}. \end{aligned}$$

For all integers $K \geqslant 1$, $L \geqslant 2$ and real number $E > 1$ such that

$$\begin{aligned} KL \log E \geqslant\ & D_{\alpha,\beta} KL \log 2 + D_{\alpha,\beta}(K-1) \log \left(e\sqrt{3L} d_{L-1} \right) + D_{\alpha,\beta} \log(\mathcal{D}_{K-1,L-1}) \\ & + D_{\alpha,\beta}(1 + 2\log 2)(L-1) + D_{\alpha,\beta} \log \left(\min \left(d_{L-2}^{K-1}, (L-2)! \right) \right) + \log((K-1)!) \\ & + (K-1) \log(\mathcal{A}_{\alpha,\beta}/2) + LE|\alpha| + L \log E + (L-1) \log(\mathcal{B}_{\alpha,\beta}/2), \end{aligned} \tag{A.2}$$

we have $|\exp(\alpha) - \beta| \geqslant E^{-KL}$.

This result is readily specialized to the case where α and β are both precision p floating-point number by noting that in that case $D_{\alpha,\beta} = 1$, $\log \mathcal{A}_{\alpha,\beta} = \max(0, p - 1 - e_\alpha) \log 2$, $\log \mathcal{B}_{\alpha,\beta} = \max(0, p - 1 - e_\beta) \log 2$, where e_α and e_β are the respective exponents of α and β; this gives an estimate for the hardness to round for directed rounding functions.

Remark A.2. In this section, we only address the directed rounding function case. For the round-to-nearest rounding function, we have to assume that β is the middle of two consecutive FP numbers

and that the numbers β and e^α are in the same binade. The theorem should then be used with e_β replaced by $e_\beta - 1$, which in practice has very little impact.

We now turn to show how this result applies to other functions.

Consequences for hyperbolic and trigonometric functions

LEMMA A.3. *Let α, β with $\alpha \neq 0$, $|\beta| < 1$ be two floating-point numbers in precision p. Assume that $|\cos(\alpha) - \beta| \leqslant \varepsilon$.*

Then there exist two algebraic numbers α', β' with $D_{\alpha',\beta'} \leqslant 2$, $\log \mathcal{A}_{\alpha',\beta'} = 2\max(0, p-1-e_\alpha)\log(2)+\log(\max(1,|\alpha|))$ and $\log \mathcal{B}_{\alpha',\beta'} = \max(0, p-2-e_\beta)\log(2)$, $|\alpha'| = |\alpha|$ such that $|\exp(\alpha') - \beta'| \leqslant \sqrt{2\varepsilon}$.

If, further, $1 - |\beta| \geqslant \delta$, we then have $|\exp(\alpha') - \beta'| \leqslant 2\varepsilon/\sqrt{\delta}$.

These statements also hold for the sine function.

PROOF. We write

$$|\cos\alpha - \beta| = \frac{1}{2}\left|\exp(2i\alpha) - 2\beta\exp(i\alpha) + 1\right| = \frac{1}{2}\left|\exp(i\alpha) - \beta + i\sqrt{1-\beta^2}\right| \cdot \left|\exp(i\alpha) - \beta - i\sqrt{1-\beta^2}\right|.$$

As the product of the last two terms is at most 2ε, one of the two must be at most $\sqrt{2\varepsilon}$.

Put $\alpha' = i\alpha$, $\beta' = \beta + si\sqrt{1-\beta^2}$, for $s \in \{-1, 1\}$ such that $|\exp(\alpha') - \beta'|$ is minimal. Then, unless $1 - \beta^2$ is a perfect square (which is equivalent to $\beta = 0$, as 2^{2p} is not a sum of two squares of integers), $[\mathbb{Q}(\alpha', \beta') : \mathbb{Q}] = 4$ whereas $[\mathbb{R}(\alpha', \beta') : \mathbb{R}] = 2$, so $D_{\alpha',\beta'} = 2$. If $1 - \beta^2$ is a perfect square, we have $D_{\alpha',\beta'} = 1$.

Further, the minimal polynomial of α' over $\mathbb{Z}$ is $2^{2\max(0,p-1-e_\alpha)}(X^2 + \alpha^2)$, from which we deduce easily that $h(\alpha') = h(\alpha)$; thus in all cases $D_{\alpha',\beta'}h(\alpha') - \log(\max(1,|\alpha'|)) = D_{\alpha',\beta'}h(\alpha) - \log(\max(1,|\alpha|)) \leqslant 2\max(0, p-1-e_\alpha)\log(2) + \log(\max(1,|\alpha|))$.

Finally, the minimal polynomial of β' over $\mathbb{Z}$ is $2^{\max(0,p-2-e_\beta)}(X^2 - 2\beta X + 1)$, and as the two roots of this polynomial have modulus 1, $h(\beta') = \max(0, p-2-e_\beta)\log(2)/2$; hence, $D_{\alpha',\beta'}h(\beta') - \log(\max(1,|\beta'|)) \leqslant \max(0, p-2-e_\beta)\log(2)$ in all cases.

Assume now that $1 - |\beta| \geqslant \delta$; then, $|(\beta + i\sqrt{1-\beta^2}|) - (\beta - i\sqrt{1-\beta^2}|)| = 2\sqrt{1-\beta^2} \geqslant 2\sqrt{\delta}$. Thus, from the triangle inequality, at least one of $|\exp(i\alpha) - \beta + i\sqrt{1-\beta^2}|$, $|\exp(i\alpha) - \beta - i\sqrt{1-\beta^2}|$ must be $\geqslant \sqrt{\delta}$, and the other one is $\leqslant 2\varepsilon/\sqrt{\delta}$. We conclude as in the first case.

The proof follows the same lines for the sine function, with $\beta' = s\sqrt{1-\beta^2} + i\beta$, for $s \in \{-1, 1\}$. If $1 - |\beta| \geqslant \delta$, $|(\sqrt{1-\beta^2} + i\beta) - (-\sqrt{1-\beta^2} + i\beta)| \geqslant 2\sqrt{\delta}$, so that for at least one $\beta' \in \{\pm\sqrt{1-\beta^2} + i\beta\}$, we have $|\exp(i\alpha) - \beta'| \leqslant 2\varepsilon/\sqrt{\delta}$. □

The simplest way to use the previous Lemma is via the first inequality, which directly reduces the problem of finding a lower bound for $|\cos\alpha - \beta|$ to finding a lower bound for $|\exp(\alpha') - \beta'|$. Without loss of generality for our problem, we can restrict to the case where β is a floating-point closest to $\cos(\alpha)$.

The second inequality is much more efficient in practice: over a given binade $[2^k, 2^{k+1})$, it is quite easy to find the precision-p floating-point number closest to a multiple of π (resp. to an odd multiple of $\pi/2$ for the sine function, resp. to 0 for cosh), which gives a bound for δ over this interval.

If this value of δ is less than 2^{-p} we can improve this bound by noting that β precision-p floating-point number and $|\beta| < 1$ implies $1 - |\beta| \geqslant 2^{-p}$.

For instance, over $[4, 8]$, the precision-113 floating-point number closest to 2π (which is the only multiple of π in the interval under consideration) is

$$2^2 \cdot \frac{8156040833015188200833743081374136}{2^{112}},$$

the cosine of which is $\leqslant 1 - 1.5 \cdot 10^{-68}$. We then use the better bound $\delta = 2^{-113}$.

Lemma A.4. *Let α, β with $\alpha \neq 0$, $\beta \geqslant 1$ be floating-point numbers in precision p. Assume that $|\cosh(\alpha) - \beta| \leqslant \varepsilon$.*

Then there exist two algebraic numbers α', β' with $D_{\alpha',\beta'} \leqslant 2$, $\log \mathcal{A}_{\alpha',\beta'} = \max(0, p - 1 - e_\alpha)\log(2) + \log(\max(1, |\alpha|))$, $|\alpha'| = |\alpha|$, $\log \mathcal{B}_{\alpha',\beta'} = \max(0, p - 2 - e_\beta)\log(2) + |\log(2\beta)|$, such that $|\exp(\alpha') - \beta'| \leqslant \sqrt{2\varepsilon}$.

If, further, $\beta \geqslant 1 + \delta$, then we have $|\exp(\alpha') - \beta'| \leqslant 2\varepsilon/\sqrt{\delta}$.

Proof. We have

$$|\cosh(\alpha) - \beta| = \frac{\exp(-\alpha)}{2} \cdot \left|\exp(\alpha) - \beta + \sqrt{\beta^2 - 1}\right| \cdot \left|\exp(\alpha) - \beta - \sqrt{\beta^2 - 1}\right|.$$

Up to changing α to $-\alpha$, we can assume without loss of generality that $\alpha < 0$, in which case the product of the last two terms is again upper bounded by 2ε.

We thus take $\alpha' = \pm\alpha$ and choose $\beta' \in \{\beta \pm \sqrt{\beta^2 - 1}\}$ such that $|\exp(\alpha') - \beta'|$ is minimal; in this case, $D_{\alpha',\beta'} \leqslant 2$ and

$$\begin{aligned} D_{\alpha',\beta'} h(\alpha') - \log(\max(1, |\alpha'|)) &\leqslant 2h(\alpha) - \log(\max(1, |\alpha|)) \\ &= \max(0, p - 1 - e_\alpha)\log 2 + \log(\max(1, |\alpha|)). \end{aligned}$$

Further, β' is a root of the polynomial with integer coefficients $2^{\max(0,p-2-e_\beta)}(X^2 - 2\beta X + 1)$. As the other root is $1/\beta'$, the sum in the definition of $h(\beta')$ (see A.1) is at most equal to $\max(\log|\beta'^{-1}|, \log|\beta'|) = |\log|\beta'||$. This shows that $h(\beta') \leqslant \frac{1}{2}\left(\max(0, p - 2 - e_\beta)\log 2 + |\log|\beta'||\right)$, so that

$$\begin{aligned} D_{\alpha',\beta'} h(\beta') - \log(\max(1, |\beta'|)) &\leqslant \max(0, p - 2 - e_\beta)\log 2 + |\log|\beta'|| - \log(\max(1, |\beta'|)) \\ &\leqslant \max(0, p - 2 - e_\beta)\log 2 + |\log(2\beta)|. \end{aligned}$$

If now $\beta - 1 \geqslant \delta$, as $|(\beta + \sqrt{\beta^2 - 1}) - (\beta - \sqrt{\beta^2 - 1})| = 2\sqrt{\beta^2 - 1} \geqslant 2\sqrt{\delta}$, so that either $\exp(\alpha) - (\beta - \sqrt{\beta^2 - 1})$ or $\exp(\alpha) - (\beta + \sqrt{\beta^2 - 1})$ is $\geqslant \sqrt{\delta}$, so that the other one is $\leqslant 2\varepsilon/\sqrt{\delta}$. □

Lemma A.5. *Let α, β be two floating-point numbers in precision p. Assume that $|\sinh(\alpha) - \beta| \leqslant \varepsilon$.*

Then there exist two algebraic numbers α', β' with $D_{\alpha',\beta'} \leqslant 2$, $\log \mathcal{A}_{\alpha',\beta'} = \max(0, p-1-e_\alpha)\log(2) + \log(\max(1, |\alpha|))$ and $|\alpha'| = |\alpha|$, $\log \mathcal{B}_{\alpha',\beta'} = \max(0, p - 2 - e_\beta)\log(2) + |\log(|\beta| + \sqrt{1 + \beta^2})|$, such that $|\exp(\alpha') - \beta'| \leqslant 4\varepsilon$.

Proof. We have $\varepsilon \geqslant |\sinh(\alpha) - \beta| = \frac{\exp(-\alpha)}{2}|\exp(\alpha) - \beta + \sqrt{\beta^2 + 1}||\exp(\alpha) - \beta - \sqrt{\beta^2 + 1}|$.

By a similar argument as before, we can restrict to $\alpha \leqslant 0$. As the difference of the last two terms is $2\sqrt{\beta^2 + 1} \geqslant 1$, one of these terms is $\geqslant 1/2$, from which we get the result. □

We now turn our attention to the tangent and cotangent functions.

Lemma A.6. *Let $\alpha \neq 0$, $\beta \neq 0$ be two floating-point numbers in precision p.*

Assume that $|\tan(\alpha) - \beta| \leqslant \varepsilon$. Then there exist two algebraic numbers α', β' with $D_{\alpha',\beta'} = 1$, $\log \mathcal{A}_{\alpha',\beta'} = \max(1, p - 1 - e_\alpha)\log 2$, $|\alpha'| = 2|\alpha|$, $\log \mathcal{B}_{\alpha',\beta'} = \max(p - 1 - e_\beta, 0)\log 2 + \frac{1}{2}\log(1 + \beta^2)$, such that $|\exp(\alpha') - \beta'| \leqslant 2\varepsilon$. The same results holds for the cotangent function.

Proof. We write

$$\begin{aligned}|\tan(\alpha)-\beta| &= \left|\frac{\exp(2i\alpha)-1}{i(1+\exp(2i\alpha))}-\beta\right| \\ &= |1+\exp(2i\alpha)|^{-1}\cdot|\exp(2i\alpha)-1-i\beta(1+\exp(2i\alpha))| \\ &\geqslant \frac{1}{2}\,|\exp(2i\alpha)(1-i\beta)-(1+i\beta)| \\ &= \frac{|1-i\beta|}{2}\left|\exp(2i\alpha)-\frac{1+i\beta}{1-i\beta}\right| \\ &\geqslant \frac{1}{2}\left|\exp(2i\alpha)-\frac{1+i\beta}{1-i\beta}\right|.\end{aligned}$$

We put $\alpha' = 2i\alpha$ and $\beta' = (1+i\beta)/(1-i\beta)$. Obviously, we have $D_{\alpha',\beta'} = 1$.

The minimal polynomial of α' over $\mathbb{Z}$ is $2^{2\max(0,p-2-e_\alpha)}(X^2+4\alpha^2)$, from which we deduce that $h(\alpha') = \max(p-2-e_\alpha,0)\log(2) + \log\max(2|\alpha|,1) \leqslant \max(p-1-e_\alpha,1)\log 2 + \log(\max(|\alpha|,1))$. The claim on $\log \mathcal{A}_{\alpha',\beta'}$ follows.

The minimal polynomial of β' over $\mathbb{Z}[X]$ is $2^{\max(2p-2-2e_\beta,0)}((\beta^2+1)X^2+(2\beta^2-2)X+(\beta^2+1))$; as $|\beta'| = 1 = |\overline{\beta'}|$, the logarithmic height of β' is equal to $\max(p-1-e_\beta,0)\log 2 + \frac{1}{2}\log(1+\beta^2)$, which concludes the proof for the tangent function.

The proof for the cotangent function is the same with $\beta' = (1-i\beta)/(1+i\beta)$. □

We now turn our attention to the hyperbolic tangent and cotangent functions.

Lemma A.7. *Let $\alpha > 0, \beta > 0$ be two floating-point numbers in precision p. Assume that $|\tanh(\alpha)-\beta| \leqslant \varepsilon$. Then there exist two algebraic numbers α', β' with $D_{\alpha',\beta'} = 1$, $\log \mathcal{A}_{\alpha',\beta'} = \max(0,p-2-e_\alpha)\log 2$, $|\alpha'| = 2\alpha$, $\log \mathcal{B}_{\alpha',\beta'} = \max(0,p-1-e_\beta)\log(2) + \log(1+\beta)$ such that*

$$|\exp(\alpha')-\beta'| \leqslant 2\varepsilon.$$

The same result holds for the hyperbolic cotangent function, assuming $\beta > 1$.

Proof. We have

$$\begin{aligned}|\tanh(\alpha)-\beta| &= \left|\frac{1-\exp(-2\alpha)}{1+\exp(-2\alpha)}-\beta\right| \\ &= |1+\exp(-2\alpha)|^{-1}\cdot|-\exp(-2\alpha)+1-\beta(1+\exp(-2\alpha))| \\ &\geqslant \frac{1}{2}\,|\exp(-2\alpha)(1+\beta)-(1-\beta)| \\ &= \frac{|1+\beta|}{2}\cdot\left|\exp(-2\alpha)-\frac{1-\beta}{1+\beta}\right| \\ &\geqslant \frac{1}{2}\left|\exp(-2\alpha)-\frac{1-\beta}{1+\beta}\right|.\end{aligned}$$

We take $\alpha' = -2\alpha$ and $\beta' = (1-\beta)/(1+\beta)$. Obviously $D_{\alpha',\beta'} = 1$, $\log \mathcal{A}_{\alpha',\beta'} = \max(0,p-2-e_\alpha)\log 2$, $|\beta'| = |1-\beta|/|1+\beta|$. The minimal polynomial of β' over $\mathbb{Z}[X]$ is $2^{\max(0,p-1-e_\beta)}((1+\beta)X-(1-\beta))$, so that $h(\beta') = \max(0,p-1-e_\beta)\log(2) + \log|1+\beta| = \log \mathcal{B}_{\alpha',\beta'}$.

For the hyperbolic cotangent, assuming $\alpha > 0$, $\beta > 1$, we write

$$|\operatorname{cotanh}(\alpha)-\beta| = |\operatorname{cotanh}(\alpha)\beta|\,|\tanh(\alpha)-1/\beta| \geqslant |\tanh(\alpha)-1/\beta| \geqslant \frac{1}{2}\left|\exp(-2\alpha)-\frac{\beta-1}{\beta+1}\right|,$$

which eventually gives the same estimate. □