



HAL
open science

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

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

► **To cite this version:**

Nicolas Brisebarre, Guillaume Hanrot, Jean-Michel Muller, Paul Zimmermann. Correctly-rounded evaluation of a function: why, how, and at what cost?. 2024. hal-04474530v2

HAL Id: hal-04474530

<https://hal.science/hal-04474530v2>

Preprint submitted on 31 May 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

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 paper 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. We also give answers to common objections we have received over the last 10 years.

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 paper 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 [34, pp 58–59]:

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

Another possible starting point is the list of mathematical functions (not much different) defined by the C Standard [9].

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, Vandoeuvre-Lès-Nancy, France, F-54506, Paul.Zimmermann@inria.fr.

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 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 [9].

Organization of this article. We have contributed or are contributing to the CRLibm¹ or CORE-Math² 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 paper 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 math and computation required to design them can, in some cases, be rather complex. Finally, we present in Section 6 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 [16], and much of what is said in this paper 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.

Since our first works on the correct rounding of functions and the table maker’s dilemma [48], we have been asked many questions (either during face to face conversations, or anonymously through reviews) about the interest, feasibility and cost of correct rounding. In each section, we list the most relevant of these questions and try to give them an answer.

2 A REMINDER ON THE FLOATING-POINT ARITHMETIC FRAMEWORK

We only give the definitions and notation that are relevant for this paper. More information on floating-point arithmetic can be found in [5, 30, 60, 62].

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

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

where

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

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 [34].

In order to have a unique representation,³ we *normalize* the finite nonzero floating-point numbers by choosing the representation for which the exponent is minimum. A direct consequence is that if $|x| \geq 2^{e_{min}}$, then $2^{p-1} \leq |M| \leq 2^p - 1$. 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 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 paper turn out to be much easier. The same is actually almost⁴ true regarding binary32 arithmetic: 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.

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

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

Roughly speaking, $\text{ulp}(t)$ 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 [29].

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*⁵, 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 functions⁶; 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 the largest finite number from \mathcal{F}_p , $\circ_u(x) = +\infty$;

³The IEEE 754 Standard distinguishes two different zeros: $+0$ and -0 . While this is sometimes extremely useful [38] we do not need to make such a distinction in this paper.

⁴“Almost” because of *bivariate* binary32 functions, for which exhaustive search is not as easy.

⁵Called *rounding mode* or *rounding direction attribute* in the successive IEEE 754 jargons.

⁶More precisely, there is a sixth function: round-to-nearest *ties to zero*, but it is used only in special, *augmented* operations.

- 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 the smallest finite number from \mathcal{F}_p , $\circ_d(x) = -\infty$;
- 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 important⁷, we will write $\circ_n(x)$ for “ x rounded to nearest”. 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 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 [5], 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 $|t - \hat{t}| < \frac{1}{2}\text{ulp}(t)$ then $\hat{t} = \circ_n(t)$;*
- *if $\hat{t} = \circ_n(t)$ then $|t - \hat{t}| \leq \frac{1}{2}\text{ulp}(t)$;*
- *if $\hat{t} \in \{\circ_d(t), \circ_u(t)\}$ then $|t - \hat{t}| < \text{ulp}(t)$;*
- *if $|t - \hat{t}| < \text{ulp}(t)$ and $|\hat{t}|$ is not a power of 2 times $1 - 2^{-p}$ then $\hat{t} \in \{\circ_d(t), \circ_u(t)\}$.*

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 paper), have had a considerable impact on numerical software. They put an end to a chaos well described by Kahan [37], 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?

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 [56] writes “A software interface standard will aid in the development of portable software if it (\dots) 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

⁷This is a frequent case when the usual math functions are considered: the exponential, logarithm, sine, cosine of a floating-point number is never exactly halfway between two consecutive FP numbers. This is a consequence of Hermite-Lindemann’s theorem [74] that states that the exponential of a nonzero algebraic number is transcendental.

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 [29]) significantly between math libraries and platforms. And yet, most users expect them to be of the highest quality. Already in 1980, Cody [13] 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 [67], 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. Therefore verification methods willingly use floating-point arithmetic, not least because of its tremendous speed.

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 [64]. Their figures show that functions such as \exp , \ln , 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 [12]) 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 [28] 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 [34, Section 9.1] is a good starting point.

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. [4], 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

(\dots) 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, Demmel, and Nguyen [2], 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).⁸

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 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*?

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 [63] 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.

Now, it 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 [63] or Black et al. [4], but we aim at convincing the reader that now, this is feasible (Section 4) at a reasonable delay/energy cost (Section 5).

3.1 Answers to some usual questions

3.1.1 *OK, we need a specification of a kernel of functions, but specifying a relative/ulp error bound (say 0.501 ulp for rounding to nearest) suffices.* That would already be an improvement over the current situation, assuming the error bound is really proven,⁹ not just conjectured from random experiments. However, this would not guarantee unicity of the returned result (so that reproducibility of calculations would not be guaranteed). Furthermore, for large precisions, *guaranteeing* a tight error bound already requires a rather large effort anyway (formal proof, or a huge amount of exhaustive testing—years of computation for binary64, out of reach for binary128). Of course for small precisions one can easily perform an exhaustive test (but for these precisions, for the very same reason, correct rounding is easily implementable). However, why 0.501 and not 0.500001? Moreover, a proven error bound means that you have already implemented a “fast step” (see §4). Why stop there and not implement the “accurate step” as well? It is not more difficult.

3.1.2 *You want the computed sine, cosine, exponential... of a number to be uniquely defined for portability/debugging/reproducibility/legal purposes, but why not just specifying it as the result returned by a given algorithm?* This would fix in stone the mathematical function algorithms and stop all improvement. And do you really want to fix in stone a result that is not the best possible?

3.1.3 *We already guarantee correct rounding for the arithmetic functions, the square root, conversions. We can of course do that for some higher-level functions, but there are hundreds of functions in the math/physics bestiary: one has to stop at some point... which point?* There seems to be a reasonable consensus on which functions should be viewed as “atomic”: the list of functions in the IEEE

⁸A 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.

⁹To the best of our knowledge, this is extremely rare.

754-2019 Standard for Floating-Point Arithmetic [34, Section 9.1], the list of functions in the C Standard, and the list of functions in Cody and Waite’s book [11] are not so different.

3.1.4 Consider a sine or cosine function. In general, the input is not “exact”: it comes from an earlier calculation or a measurement. When that input is large, due to the input error being non negligible compared to π , the output cannot be known accurately: what is the point in pretending returning correct rounding of that output? As Muller wrote in [58], our feeling is that the designer of a circuit/library has no right to assume that users are stupid. If someone wants to compute the sine of a very large number, he or she may have a good reason for doing so, and the software/hardware must provide a result that is as accurate as possible. Also, even if very large input values are somehow unrealistic, the robustness of a numerical software may depend on the preservation of symmetries, on the preservation of properties such as the fact that the computed value of $\sin^2 x + \cos^2 x$ must be as close to 1 as possible (i.e., even if we no longer perform *the good rotation*, we still do perform *a rotation*), and so on. All these relations are easily preserved if we have correctly rounded functions.

3.1.5 (also listed in the “How” section) 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 $\pi/2$. Should one return a value that violates the mathematical property $|\operatorname{asin} x| \leq \pi/2$?

You have a point. This might be a case when correct rounding is not always the best solution. For the (very few!) functions and formats for which such events may happen, we might consider an optional “range takes over correct rounding” behavior.

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 function evaluation 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. In Sections 4.1, 4.2 and 4.3 we explain the theoretical difficulties and general strategies behind correct rounding of functions. The methods that can be used for the pre-calculation are described in Section 4.4. The methods used for function evaluation are described in Section 4.5.

Let us briefly 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 $f(x)$ cannot be computed (and 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 $I_{f(x)}$. This interval may be very narrow if we use high precision to compute the approximation, but it will still be nonzero in length. If all points of $I_{f(x)}$ round to the same FP number, then the correctly-rounded value of $f(x)$ is that FP number. However, if $I_{f(x)}$ 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 4.3. Before that, in Section 4.2 we show that frequently, for tiny arguments, correct rounding is easily provided.

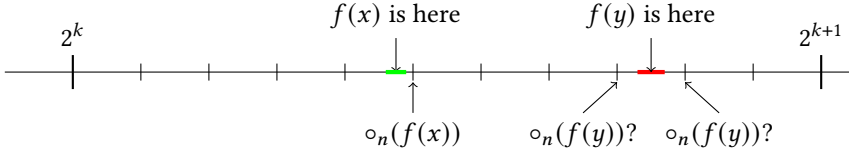


Fig. 1. The real line (between 2^k and 2^{k+1}) and the FP numbers (represented by the “ticks”). From the knowledge that $f(x)$ lies in the green interval, we can deduce the value of $\circ_n(f(x))$. However, knowing that $f(y)$ lies in the red interval does not allow us to know if $\circ_n(f(y))$ is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

4.1 Ziv’s strategy

Ziv’s onion peeling strategy [75] is a method to guarantee correct rounding at very 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 analyzed in [20] for round-to-nerarest) 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 , $1 \leq k \leq n$, with increasing relative accuracy $p_1 < p_2 < \dots < p_n$:

$$\frac{|f_k(x) - f(x)|}{|f(x)|} < 2^{-p_k}.$$

If the target precision is p , under some distributional assumptions (see Section 4.3.2), it is possible to deduce a correctly-rounded result from the approximation $f_k(x)$ with probability $1 - 2^{p-p_k}$. The key ingredients in Ziv’s strategy are the following:

- to each function f_k is associated a *rounding test* [20], which, when it succeeds, *must* deliver the correct rounding;
- the sequence f_1, \dots, 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 (§4.3) 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, or the preliminary determination of all the exact cases (that task was for example accomplished for the binary64 power function by Lauter and Lefèvre [44], 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 gamma¹⁰), finding a nontrivial exact point would be an extraordinary discovery (and 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.

¹⁰The possible trivial exact cases with the gamma function correspond to inputs that are positive integers, as $\Gamma(n) = (n-1)!$. For instance $\Gamma(14) = 13!$ is an exact point in binary32 arithmetic with a directed rounding, and $\Gamma(39) = 38!$ is a breakpoint in binary128 arithmetic with round-to-nearest.

4.2 Special values

For some extremal values, it is not necessary to use Ziv's strategy or to solve the table maker's dilemma (formalized in Section 4.3): returning a correctly-rounded result can be very easy, as we are going to see in the case of tiny *input* values (Section 4.2.1) or tiny *output* values (Section 4.2.2).

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

$$f(x) = a_0 + a_1x + a_2x^2 + \dots$$

- If 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) then the sign of $f(x) - a_0$ (which is immediately deduced from the sign of the first nonzero a_i , $i \geq 1$) and a bound on $|a_1x + a_2x^2 + \dots|$ allow 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 $0 \leq x \leq 2^{-p+1} - 2^{-2p+1}$ then $\circ_d(e^x) = 1$, and if $-2^{-p} \leq x < 0$ then $\circ_d(e^x) = 1 - 2^{-p}$. Similarly, for the cosine function, if $|x| \leq \circ_d(2^{(-p+1)/2})$ then $\circ_u(\cos(x)) = 1$.
- If $a_0 \neq 0$ is not a breakpoint then, depending on the sign of the first nonzero coefficient a_i , $i \geq 1$, 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 $f(x) = 2^x$, for

$$\log_2(1 - 2^{-p-1}) \leq x \leq \log_2(1 + 2^{-p}),$$

we have $\circ_n(2^x) = 1$ (in binary32 arithmetic, that domain corresponds to $-12102203 \times 2^{-48} \leq x \leq 6051101 \times 2^{-46}$).

- If $a_0 = 0$ and a_1 is a nonzero power of 2, so that the product a_1x is exact, and assuming that 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 $\circ(a_1x) = a_1x$ (or to the preceding/next FP number). Let us give an example with function

$$f(x) = \arctan(x) = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \dots$$

If we exclude the case where x is a power of 2 (to be processed separately), we need to know when

$$\rho(x) = \left| \frac{x^3}{3} - \frac{x^5}{5} + \frac{x^7}{7} - \dots \right|$$

is less than $\frac{1}{2}\operatorname{ulp}(x)$ (for round-to-nearest) or less than $\operatorname{ulp}(x)$ (for the other rounding functions). By reasoning on the binade $2^k < |x| < 2^{k+1}$ where x lies, one finds that

- if p is even then $\rho(x) < \operatorname{ulp}(x)$ as soon as $|x| \leq \circ_d\left(\left(\frac{3}{2}\right)^{\frac{1}{3}}\right) \cdot 2^{-\frac{p}{2}+1}$, and $\rho(x) < \frac{1}{2}\operatorname{ulp}(x)$ as soon as $|x| \leq \circ_d\left(3^{\frac{1}{3}}\right) \cdot 2^{-\frac{p}{2}}$;
- if p is odd then $\rho(x) < \operatorname{ulp}(x)$ as soon as $|x| \leq \circ_d\left(3^{\frac{1}{3}}\right) \cdot 2^{-\frac{p+1}{2}}$ and $\rho(x) < \frac{1}{2}\operatorname{ulp}(x)$ as soon as $|x| \leq \circ_d\left(\left(\frac{3}{2}\right)^{\frac{1}{3}}\right) \cdot 2^{-\frac{p+1}{2}}$.

For instance, in binary64 arithmetic ($p = 53$) and with a round-to-nearest rounding function, we have $\circ_n(\arctan(x)) = x$ for $|x| \leq \circ_d\left(\left(\frac{3}{2}\right)^{\frac{1}{3}}\right) \cdot 2^{-26} = 1.1447 \dots \cdot 2^{-26}$ and $|x| \neq 2^{-26}$, and

one easily checks that $\circ_n(\arctan(2^{-26}))$ is the FP predecessor of 2^{-26} . In practice the domain where such simplifications can be made is easily found by the means of a binary search.

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 , $\sinpi(x) = \sin(\pi x)$ is close to a breakpoint if and only if πx is close to a breakpoint (and in such a case, $\sinpi(x/2)$, $\sinpi(x/4)$, etc. are close to a midpoint too).

4.2.2 The special case of tiny output values. The analysis given in the rest of the paper 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 $B \in \mathbb{N}$, $2^{p-1} \leq B \leq 2^p - 1$. 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 4.2.1 (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 Kahan¹¹ and described in detail in [58, 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 the subnormal threshold of 2.225×10^{-308} .

4.3 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 [39]. 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 $\text{htr}_f(p)$, that we call *hardness to round* of f (see Definition 4.3).

4.3.1 Formalization of the problem. Assume we wish to correctly round a real-valued function φ . 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 $x \geq 0$ and $\varphi(x) \geq 0$.

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 $x \in [2^{e_1}, 2^{e_1+1})$, are not all included in a binade of the form $[2^{e_2}, 2^{e_2+1})$ ¹², 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 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:

¹¹See <https://people.eecs.berkeley.edu/~wkahan/testpi/>

¹²A binade is an interval of the form $[2^k, 2^{k+1})$ or $(-2^{k+1}, -2^k]$ for $k \in \mathbb{Z}$.

PROBLEM 4.1 (TMD, DIRECTED ROUNDING FUNCTIONS). *What is the minimum $\mu(p) \in \mathbb{Z}$ such that, for $2^{p-1} \leq X \leq 2^p - 1$ (and, possibly, the restrictions implied by $X/2^{-e_1+p-1} \in I$) such that $\varphi(X2^{e_1-p+1}) \notin \mathcal{F}_p$ and for $2^{p-1} \leq Y \leq 2^p$, we have*

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

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

PROBLEM 4.2 (TMD, ROUNDING TO NEAREST FUNCTIONS). *What is the minimum $\mu(p) \in \mathbb{Z}$ such that, for $2^{p-1} \leq X \leq 2^p - 1$ (and, possibly, the restrictions implied by $X/2^{-e_1+p-1} \in I$) such that $\varphi(X2^{e_1-p+1})$ is not the middle of two consecutive elements of \mathcal{F}_p and for $2^{p-1} \leq Y \leq 2^p - 1$, we have*

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

These statements lead to the following definition.

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 x and $\varphi(x) \geq 0$, since the extension to the other cases is straightforward).

The *hardness to round* $\varphi(x)$, denoted $\text{htr}_{\varphi, \{x\}, \circ}(p)$ is equal¹³ to:

- $+\infty$ if $\varphi(x)$ is a breakpoint;
- the minimum $\mu(p) \in \mathbb{Z}$ such that the distance of $\varphi(x)$ to the nearest breakpoint is larger than or equal to $2^{-\mu(p)-p+1+e_2}$.

The hardness to round φ over an interval I , denoted $\text{htr}_{\varphi, I, \circ}(p)$, is then the maximum of the hardness to round $\varphi(x)$ for all FP $x \in \mathcal{F}_p \cap I$, while the hardness to round φ is the hardness to round φ over \mathbb{R} , simply denoted $\text{htr}_{\varphi, \circ}(p)$. 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 $p + 1$.

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 \ln or $x \mapsto \sqrt[3]{x}$ and $x \mapsto x^3$. See Lemma 4.8 for an explicit statement.

4.3.2 A heuristic probabilistic approach and some partial results. If we have N FP numbers in the domain being considered, it is expected that $\text{htr}_f(p)$ is of the order of $\log_2(N)$ (hence p for most usual functions and binades). This is supported by a probabilistic heuristic approach that is presented in detail in [58, 59] and that we know 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

$$\underbrace{00 \cdots 0}_{k \text{ bits}} \quad \text{or} \quad \underbrace{11 \cdots 1}_{k \text{ bits}}$$

¹³One may find in other texts the same value shifted by $p - 1$.

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

$$\underbrace{100 \cdots 0}_{k \text{ bits}} \quad \text{or} \quad \underbrace{011 \cdots 1}_{k \text{ bits}}$$

is 2^{-k+1} . One can find such an estimate used in [23] and a probabilistic study has been done in [27]. 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 [7], 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 cases that the CRLibm or CORE-Math libraries use in practice.

4.3.3 Diophantine approximation results. We now recall several theoretical results that can prove useful, yet insufficient, for algebraic¹⁴ functions like $1/\sqrt{\cdot}$, $\sqrt[3]{\cdot}$, \dots 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. 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 [50–52].

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

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

We can take, for instance, $C_\alpha = \frac{1}{\max_{|t-\alpha| \leq 1/2} |P'_\alpha(t)|}$. 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 [66], that gives an exponent $2 + \varepsilon$ for any $\varepsilon > 0$ instead of d . Actually, when tackling the TMD, the integer q is a power of 2 and we can therefore take advantage of Ridout's improvement [65] 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 [25]. 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 [8], Brisebarre and Muller followed Liouville's approach to obtain simple effective upper bounds on the hardness to round $\text{htr}_f(p)$ for algebraic f . See also [35, 42] 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 $x \in \mathcal{F}_p \cap [1, 2^a)$, since 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 $\text{htr}_{f_a}(p) \leq (a-1)p + a + \log_2(a) - 2$. We remind the reader that we rather expect $\text{htr}_{f_a}(p)$ to be of the order of p .

¹⁴A 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, $P(x, \varphi(x)) = 0$.

¹⁵Note that in the quadratic, that is to say $d = 2$, case, Liouville's result remains better.

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 [74] 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 numbers¹⁶ 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 $\alpha', \beta', \epsilon'$ as functions of α, β, ϵ such that $|f(\alpha) - \beta| \leq \epsilon \Rightarrow |\exp(\alpha') - \beta'| \leq \epsilon'$. In the cases where further assumptions on α, β are used to improve the bound, those are given in the last column.

Table 2. Relating trigonometric & hyperbolic functions to exp

Function	α'	β'	ϵ'	if...
cos	$i\alpha$	$\beta \pm i\sqrt{1 - \beta^2}$	$\sqrt{2\epsilon}$	
cos	$i\alpha$	$\beta \pm i\sqrt{1 - \beta^2}$	$2\epsilon/\sqrt{\delta}$	$1 - \beta \geq \delta$
sin	$i\alpha$	$i\beta \pm \sqrt{1 - \beta^2}$	$\sqrt{2\epsilon}$	
sin	$i\alpha$	$i\beta \pm \sqrt{1 - \beta^2}$	$2\epsilon/\sqrt{\delta}$	$1 - \beta \geq \delta$
cosh	$\pm\alpha$	$\beta \pm \sqrt{\beta^2 - 1}$	$\sqrt{2\epsilon}$	
cosh	$\pm\alpha$	$\beta \pm \sqrt{\beta^2 - 1}$	$2\epsilon/\sqrt{\delta}$	$\beta \geq 1 + \delta$
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)

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.

LEMMA 4.8. *Let $f : I \subset \mathbb{R} \rightarrow \mathbb{R}$ be a continuously differentiable function. Assume that for all $\alpha \in [a, b] \subset I, \beta \in \mathbb{R}, |f(\alpha) - \beta| \geq \epsilon$. Then, for all $\alpha \in [a, b], \beta \in f(I)$, we have*

$$|f^{-1}(\beta) - \alpha| \geq \frac{\epsilon}{\sup_{[\alpha, f^{-1}(\beta)]} |f'|}.$$

Applying the theoretical results presented in Appendix A and Lemma 4.8, we obtain upper bounds¹⁷ 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 $f^{-1}([a, b])$. For example, from the bounds for exp and tan in Table 3, it is easy to deduce bounds for ln and atan.

¹⁶that is to say roots of polynomials with rational coefficients.

¹⁷The SageMath code computing these bounds is available at <https://members.loria.fr/PZimmermann/papers/#crsurvey>.

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

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

4.4 Search for 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 $\text{dist}(y, \mathcal{B}_\circ) = \min\{|y - z|, z \in \mathcal{B}_\circ\}$.

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 $2^{p-1-e_f(x)} \text{dist}(f(x), \mathcal{B}_\circ)$ is minimal among all possible FP values of x such that $f(x)$ is not a breakpoint. Section 4.3.2 suggests that we should expect that apart from degenerate cases (for instance, $\cos(x)$ for x close to 0) this quantity 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-points numbers x such that $2^{p-1-e_f(x)} \text{dist}(f(x), \mathcal{B}_\circ) < 2^{-m}$, i.e.,

- [directed rounding functions]

$$2^{p-1-e_f(x)} f(x) = \pm \underbrace{1 \dots \dots}_{p \text{ bits}} \cdot \underbrace{00 \dots 0}_{m \text{ bits}} \dots \quad \text{or} \quad \pm \underbrace{1 \dots \dots}_{p \text{ bits}} \cdot \underbrace{11 \dots 1}_{m \text{ bits}} \dots ;$$

- [rounding to nearest functions]

$$2^{p-1-e_f(x)} f(x) = \pm \underbrace{1 \dots \dots}_{p \text{ bits}} \cdot \underbrace{10 \dots 0}_{m \text{ bits}} \dots \quad \text{or} \quad \pm \underbrace{1 \dots \dots}_{p \text{ bits}} \cdot \underbrace{01 \dots 1}_{m \text{ bits}} \dots .$$

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 rounding 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). Some timings are provided in Section 5.1.

4.4.1 Binary32 format. Apart for bivariate functions (see sections 4.4.4 and 5.1.1 below) the table maker's dilemma in binary32 is easily solved for a given function by an exhaustive computation [68] in a few hours on a modern laptop. 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 [21]. More subtle ideas proceed by splitting the domain into subintervals and replacing the function (assumed to be sufficiently smooth) by a 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 [45–47], studied the degree 1 case; in this case, the remaining Diophantine problem is to find two integers x, y , $|x| \leq X, |y| \leq Y$ such that $|ax + \beta - y|$ is minimal, which is solved by elementary Diophantine arguments, either the three distance theorem, or continued fractions (see e.g., [3]). These ideas lead to an algorithm of complexity $\tilde{O}(2^{2p/3})$ for floating-point numbers of precision p , as $p \rightarrow \infty$, which computes all worst cases for rounding in the domains under consideration. The worst cases published in [47] were obtained using that algorithm.

Higher degree approximations give rise to more complicated Diophantine problems. Stehlé, Lefèvre and Zimmermann [71], further refined by Stehlé [70], make use of a technique due to Coppersmith [14, 15] and based on lattice basis reduction to solve it¹⁸. We recall Corollaries 4 & 5 of [70], adapted to our context.

THEOREM 4.9 (STEHLÉ [70]). *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 $\geq p$.*

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 $\geq 4p^2$; the latter works by reducing a lattice of dimension $O(p^2)$ of \mathbb{R}^m for some $m = O(p^4)$.

This theorem can be extended to any fixed binade, but in practice it works well only for binades that are not too large (in absolute value); this is generally true for all the known methods that rely on a local polynomial approximation of the function under study.

However, for most functions, these methods are sufficient to cover the appropriate range, because outside of that range, either the function or its reciprocal overflows. The main exception to this rule is the case of periodic functions. The previous algorithms have been extended to the case of periodic functions in [31]. The first part of Theorem 4.9 still holds, except that the constant $(1 + \varepsilon)/2$ in the exponent of the complexity must be replaced by $(7 - 2\sqrt{10})(1 + \varepsilon) \approx 0.68$.

The heuristic character of the algorithm is rather mild (i.e., the algorithm works in practice as expected on almost all inputs).

Recently, Brisebarre and Hanrot [6] 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 [70, 71], 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.

For general functions, their approach allows one to recover the results of Theorem 4.9. However, first, they significantly improve on several algorithmic aspects, resulting in major improvements in the polynomial part of the asymptotic complexity (lattice basis reduction). Second, their approach

¹⁸We will call Stehlé’s variant SLZ algorithm in the sequel.

permits an improvement on the second part of Theorem 4.9 for regular functions with moderate growth at ∞ (exp, sin, cos). They prove the following theorem:

THEOREM 4.10 (BRISEBARRE, HANROT [6]). *Let $f \in \{\exp, \sin, \cos, \sinh, \cosh\}$. 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 $\geq (1 + \varepsilon)p^2/\ln p$; the latter works by reducing a lattice of dimension $O((p/\ln p)^2)$ of \mathbb{R}^m for some $m = O((p/\ln p)^2)$.*

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 however is to show that, for the usual functions that we target, there are no nontrivial $\theta \cdot p$ -bad cases, for a reasonable constant θ ($\theta = 5, 6, 7$, say). The SLZ and BH algorithms can perform this task (see [72] for the SLZ algorithm), see Figure 3.

4.4.4 Bivariate functions. The landscape is much more unclear 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 [73].

4.4.5 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 [55] 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

To implement efficiently correctly-rounded evaluation routines, one first has to use efficient algorithms. These algorithms are specific to each function, and are well known in the literature. These algorithms are usually the same as those used in current mathematical libraries, which do not provide correct rounding. A very good reference is the book by Peter Markstein [54]. A crucial difference, however, is that each implementation must provide a rigorous error bound to be used in the corresponding rounding test (§4.1). This error bound can be computed by hand, or by using software tools such as Gappa [18]. One can construct a formal proof of the error bounds with tools that are reasonably easy to use [22].

These algorithms consist in:

- (1) argument reduction (if available for the function to be implemented),
- (2) evaluation of a precomputed minimax polynomial, and
- (3) argument reconstruction (if argument reduction was performed).

Tools such as Sollya [10] provide good minimax polynomials of a given degree for a given function over a given interval, with optimal or near-optimal absolute or relative error, and possibly under

additional constraints (such as the requirement that all coefficients must be representable in binary64).

To evaluate a given minimax polynomial, or for the argument reduction or reconstruction, one should try to use only floating-point additions/subtractions and multiplications (possibly fused-multiply adds which are now very common in hardware).

Usually, if the target format is binary64, the first phase of Ziv's strategy (§4.1) will use double-double arithmetic with a target accuracy of about 70 bits for the binary64 format. Efficient double-double algorithms are known with tight and rigorous error bounds [36], and have even been formally proven [57].

For the second and further phases (if any), still for the binary64 format, the implementor has the choice between double-double arithmetic (with a target accuracy of at most 106 bits), and integer-based arithmetic [53].

4.6 Answers to some usual questions

4.6.1 You will not be able to obtain hardest to round cases for all functions in their whole domain. For these functions, when you cannot obtain a reasonable bound on the hardness to round, you will have to use Ziv's initial approach, i.e., you will just stop the calculation at some point, without any proof that this suffices.

We already have hardest-to-round cases in binary64 arithmetic for most usual functions, so that very efficient correctly-rounded calculation of univariate functions in binary64 is at hand. Section 4.3 shows that even when the worst cases are out of reach one can anyway bound the hardness-to-round (the bounds are, admittedly, rather large but calculations with that precision is feasible).

4.6.2 Correct rounding also comes with the cost of more complex algorithms. I am not sure whether this is beneficial for the design of formal proofs.

Formal proof requires a complete specification of what is computed, and correct rounding is the simplest, neatest possible specification. Furthermore, the algorithms are not that complex, and a significant part of the effort of building formal proof for correctly rounded functions has already been accomplished [18, 55].

4.6.3 Question 3.1.5 also falls into the "How" category.

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 4.3 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.

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

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 [69] for the most useful of them, namely power, hypotenuse and $\arctan 2$ (with a possible adaptation to $\operatorname{atan} 2\pi$). Work still needs to be done on 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 [32], which is multi-threaded and can therefore run efficiently on a multi-core processor. In Figure 2, we see

algorithm	exp	cbt
SLZ	2.2 hours (20.3)	2.0 hours (20.6)

Fig. 2. Time to search worst cases (with at least 43 identical bits after the round bit) in binary64 for e^x and $x^{1/3}$ using BaCSeL on a 64-core AMD EPYC 7282, for $1/2 \leq x < 1$. BaCSeL was compiled with the AUTOMATIC flag. Values in parentheses indicate the average log-2 size of sub-intervals, for example 20.3 corresponds to sub-intervals of 1,300,000 elements.

that for the binary64 format, 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 $x^{1/3}$, worst cases in $[2^{e+3k-1}, 2^{e+3k})$ are those in $[2^{e-1}, 2^e)$ multiplied by 2^k , 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 [58] gives the hardest-to-round points for functions $\ln(x)$ and $\ln(1+x)$ in binary64 arithmetic.

Function	Domain	Argument	Truncated result	Trailing bits
$\ln(x)$	$[2^{-1074}, 2^{-1})$	1.EA71D85CEE020P-509	-1.60296A66B42FFP8	11 ⁶⁰ 0000...
		1.9476E304CD7C7P-384	-1.09B60CAF47B35P8	10 ⁶⁰ 1010...
		1.26E9C4D327960P-232	-1.4156584BCD084P7	00 ⁶⁰ 1001...
1.613955DC802F8P-35		-1.7F02F9BAF6035P4	01 ⁶⁰ 0011...	
	$[2^{-1}, 2^1)$	1.BADED30CBF1C4P-1	-1.290EA09E36478P-3	11 ⁵⁴ 0110...
	$[2^1, 2^{1024})$	1.C90810D354618P245	1.54CD1FEA76639P7	11 ⁶³ 0101...
		1.62A88613629B6P678	1.D6479EBA7C971P8	00 ⁶⁴ 1110...
$\ln(1+x)$	$(2^{-51}, 2^{1024})$	1.80000000000003P-50	1.7FFFFFFFFFFFFEP-50	10 ⁹⁹ 1000...
	$(-1, -2^{-51})$	-1.7FFFFFFFFFFFFDP-50	-1.8000000000001P-50	01 ⁹⁹ 0110...

Table 4. Non trivial hardest-to-round points for functions $\ln(x)$ and $\ln(1+x)$ [58]. The values given here suffice to round functions $\ln(x)$ and $\ln(1+x)$ correctly in the full binary64 range (when $|x| \leq 2^{-51}$, the correct rounding of $\ln(1+x)$ can be obtained using the technique presented in Section 4.2.1).

In light of this data, the table maker’s dilemma can be considered as solved for the binary64 format for univariate functions (bivariate functions appear, in the current state of knowledge, to be out of reach); known worst cases for the binary64 format are available on Lefèvre’s page <https://www.vinc17.net/research/testlib/> 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, worst cases for $\cos(x)$ and $\sin(x)$ are known only for $|x| \leq 2^{11}$, and worst cases for $\tan(x)$ are known only for $|x| \leq 10.5\pi$).

5.1.3 *Binary128 format.* In Figure 3, we compare the performance of implementations of SLZ and BH algorithms, namely BaCSeL¹⁹ and LACoR²⁰, when tackling the determination of $5p$ -bad cases over a binade. We can see a reasonable computation time, which opens the way to a guaranteed

¹⁹<https://gitlab.inria.fr/zimmerma/bacsel>

²⁰<https://perso.ens-lyon.fr/nicolas.brisebarre/lacor.tgz>

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).

SLZ	BH
305 years	445 days

Fig. 3. Time to search $5p$ worst cases of e^x for $x \in [1/4, 1/2]$ for quadruple precision on a Core i7-8700 at 3.20GHz. The SLZ time is obtained with BaCSeL parameters $d = 16$, $\alpha = 5$, $t = 76$; the BH time is obtained with $d = 12$, $\rho_1 = 536870912$, $t = 88.5$, $n_1 = 58$, $n_2 = 3$.

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.

5.1.4 Formal verification. The cost of the certificate-based approach has been evaluated in [55]. 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 ≤ 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 than to verify. 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

The function evaluation algorithms used in libraries such as CRLibm or CORE-MATH (see Section 6) 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^{-2}). The number of coefficients of these first approximations is comparable to those used in the good current libraries. 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 the good (but not correctly-rounded) current math libraries (a few figures are given in Table 5. For more, see <https://core-math.gitlabpages.inria.fr>). 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^{-2}). 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.

5.3 Answers to some usual questions

5.3.1 *The cost (delay, memory) of correct rounding is too high.*

With a good choice of the accuracy of the first, “fast” step of Ziv’s strategy, the average cost of correct rounding is only slightly more than the cost of that fast step (i.e., essentially the same as the cost of the other math libraries).

5.3.2 *OK, I understand that on average the cost of correct rounding can be made reasonable, but the worst case cost is too large (and for real-time applications, one has to consider worst case delays).*

Examples with the CORE-MATH library [69] show that the worst case delays can be made reasonable. We do not require that correctly-rounded functions should be the *only* available functions: if really the additional delay is not acceptable, a possible solution is to “unplug” the rounding test at the end of the first step of Ziv’s strategy (i.e., to perform the first step only).

5.3.3 *You will not be able to do correct rounding in a reasonable delay for vector applications.*

Same answer as for the previous question.

5.3.4 *You have invested a lot of expertise and time in designing correctly-rounded routines, if you had invested the same amount in non correctly-rounded routines, what would be the speedup?*

As said above, the delay is on average slightly more than the delay of the first step. And because of this, it is *that* step that receives much optimization effort. An optimized not-correctly-rounded routine would essentially be that first step. So in a way we *do* have invested the same amount of expertise and time in designing non correctly-rounded routines.

5.3.5 *Correctly-rounded routines are more complex to maintain. How will you be able to maintain them?*

Precisely for the reason given in the answer to the previous question, the largest effort is on the first step. Hence, correctly-rounded routines are not much more complex to maintain. 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).

5.3.6 *Correct rounding comes with the cost of having to compute with a larger intermediate precision than necessary. Even when this cost is only about 10%, this still leads to a huge waste of energy whenever correct rounding is not needed.*

Concerning energy consumption, what matters is the *average* consumption of a function call. So it is roughly the same as for the other (non correctly-rounded) libraries. And in some (admittedly difficult to quantify) cases of iterative calculations, a more accurate result may imply faster convergence.

6 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 January 2024), and whether the algorithms are documented. Unless stated otherwise, when the algorithms are documented, correct rounding is obtained with Ziv’s onion peeling strategy [19].

MathLib. Also called LibUltim, it is a library developed by IBM around 1990 [75]. 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 [75]. 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 [33]. 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 [17]. 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 [24, 43]. For example, for the binary64 `exp` function, [17] 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). It provides binary32 functions only: `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. The originality of *RLIBM* is that it uses a new approach based on linear programming, to find polynomials that yield correct rounding [49]. 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/rllibm/>.

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. It provides all binary32 functions from the C99 standard (except `atan2`, `cbt`, `erfc`), and a few binary64 functions (`exp`, `exp2`, `exp10`, `expm1`, `hypot`, `log`, `log10`, `log1p`, and `log2`). 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. It provides all binary32 and binary64 functions from the C99 standard, as well as new functions from the C23 standard. *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 [33]. Most routines use a “fast path” that delivers about $p + 13$ extra bits for a target precision of p bits, with a rigorous error bound. Then a “rounding test” yields the correct rounding for 99% of the inputs. In case the rounding test fails, an “accurate path” delivers an approximation with about $2p$ correct bits. The hardest-to-round cases for which the accurate path would deliver a wrong result are treated separately; there are very few such “exceptional cases”. 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. A few figures extracted

Table 5. This table, extracted from <https://core-math.gitlabpages.inria.fr>, compares the reciprocal throughput (average value) of some binary64 functions of CORE-MATH (revision 55c872b) with the GNU Libc (version 3.37), Intel Math (from icx 2023.2.0), and LLVM (revision d099dbb) libraries, on an Intel Xeon Silver 4214. CORE-MATH and LLVM offer correctly-rounded functions.

Function	CORE-MATH	GNU libc	Intel Math Library	LLVM
cos	59.9	42.5	16.8	85.5
exp	16.3	12.7	9.8	23.7
erf	36.9	50.4	59.9	
log1p	21.9	23.6	18.6	26.0
pow	52.7	42.0	34.2	
hypot	21.8	36.4	23.1	213.8

from that website are presented in Table 5. The CORE-MATH code also contains large tables of hardest-to-round inputs, generated using the BaCSeL software tool [32]. These tables are used to check the correctness of the CORE-MATH routines, but can also be used to check 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.

GNU MPFR. Though it is not focused on IEEE formats, we also mention the MPFR library [26] 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 unary 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

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

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

In the binary64 format, correctly-rounded implementations of the 1-variable functions

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

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

In all other cases (decimal formats, binary128, 2-variable 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 paper. 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

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

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

ACKNOWLEDGMENTS

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, 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] 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>
- [4] 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 (Dec. 1984), 361–366.
- [5] 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>
- [6] 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.
- [7] 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>
- [8] 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>
- [9] C Working Group. 2023. *Programming Languages – C, Working draft*. Available at <https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3096.pdf>.
- [10] 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.
- [11] W. Cody and W. Waite. 1980. *Software Manual for the Elementary Functions*. Prentice-Hall, Englewood Cliffs, NJ.
- [12] W. J. Cody. 1971. Software for the Elementary Functions. In *Mathematical Software*, John R. Rice (Ed.). Academic Press, 171–186.
- [13] 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.
- [14] 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>
- [15] 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.
- [16] M. F. Cowlishaw. 2003. Decimal Floating-Point: algorithm 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.
- [17] 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://lipforge.ens-lyon.fr/frs/download.php/99/crlibm-0.18beta1.pdf>.
- [18] 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>
- [19] 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>
- [20] 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 (July 2013), 19 pages. <https://doi.org/10.1145/2491491.2491495>
- [21] 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>
- [22] P. G. de Lamarrière, 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>
- [23] C. B. Dunham. 1990. Feasibility of “Perfect” Function Evaluation. *SIGNAL Newsletter* 25, 4 (Oct. 1990), 25–26.
- [24] 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>
- [25] N. I. Fel’dman. 1971. An effective power sharpening of a theorem of Liouville. *Izv. Akad. Nauk SSSR Ser. Mat.* 35 (1971), 973–990.
- [26] 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>
- [27] S. Gal and B. Bachelis. 1991. An Accurate Elementary Mathematical Library for the IEEE Floating Point Standard. *ACM Trans. Math. Software* 17, 1 (March 1991), 26–45.

- [28] 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.
- [29] B. Gladman, V. Innocente, J. Mather, and P. Zimmermann. 2024. Accuracy of Mathematical Functions in Single, Double, Extended Double and Quadruple Precision. (2024). Working paper or preprint, available at <https://hal.inria.fr/hal-03141101>.
- [30] D. Goldberg. 1991. What Every Computer Scientist Should Know About Floating-Point Arithmetic. *Comput. Surveys* 23 (March 1991), 5–48.
- [31] 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 2007), 25-27 June 2007, Montpellier, France*. IEEE Computer Society, 133–140. <https://doi.org/10.1109/ARITH.2007.37>
- [32] G. Hanrot, V. Lefèvre, D. Stehlé, and P. Zimmermann. 2020. BaCSeL. <https://gitlab.inria.fr/zimmerma/bacsel>. (2020). Version 4.0.
- [33] 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>
- [34] IEEE. 2019. *IEEE Standard for Floating-Point Arithmetic (IEEE Std 754-2019)*. 1–84 pages. <https://doi.org/10.1109/IEEESTD.2019.8766229> Available at <https://ieeexplore.ieee.org/servlet/opac?punumber=8766227>.
- [35] 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.
- [36] 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>
- [37] 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>.
- [38] W. Kahan. 1987. Branch Cuts for Complex Elementary Functions. In *The State of the Art in Numerical Analysis*, A. Islerles and M. J. D. Powell (Eds.). Clarendon Press, Oxford, 165–211.
- [39] W. Kahan. 2004. A Logarithm Too Clever by Half. (2004). Available at <http://http.cs.berkeley.edu/~wkahan/LOG10HAF.TXT>.
- [40] 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.
- [41] 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>
- [42] 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.
- [43] 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>
- [44] C. Q. Lauter and V. Lefèvre. 2009. An Efficient Rounding Boundary Test for $\text{pow}(x, y)$ in Double Precision. *IEEE Trans. Comput.* 58, 2 (Feb. 2009), 197–207.
- [45] V. Lefèvre. 2000. *Moyens Arithmétiques Pour un Calcul Fiable*. Ph.D. Dissertation. École Normale Supérieure de Lyon, Lyon, France.
- [46] 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.
- [47] 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>
- [48] V. Lefèvre, J.-M. Muller, and A. Tisserand. 1997. Towards Correctly Rounded Transcendentals. In *Proceedings of the 13th IEEE Symposium on Computer Arithmetic*. IEEE Computer Society Press, Los Alamitos, CA, 132–137.
- [49] 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>
- [50] J. Liouville. 1844. Nouvelle démonstration d'un théorème sur les irrationnelles algébriques. *C. R. Acad. Sci. Paris* 18 (1844), 910–911.
- [51] 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.

- [52] 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.
- [53] J. L. Maire, N. Brunie, F. de Dinechin, and J. Muller. 2016. Computing floating-point logarithms with fixed-point operations. In *23rd 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>
- [54] P. Markstein. 2000. *IA-64 and Elementary Functions: Speed and Precision*. Prentice-Hall, Englewood Cliffs, NJ.
- [55] 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>
- [56] J. D. Mooney. 2004. Developing Portable Software. In *Information Technology*, Ricardo Reis (Ed.). Springer US, Boston, MA, 55–84.
- [57] 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>
- [58] J.-M. Muller. 2016. *Elementary Functions, Algorithms and Implementation* (3rd ed.). Birkhäuser, Boston.
- [59] 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.
- [60] 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*. Birkhäuser. 627 pages.
- [61] 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>.
- [62] M. L. Overton. 2001. *Numerical Computing with IEEE Floating-Point Arithmetic*. SIAM, Philadelphia, PA.
- [63] 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.
- [64] 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>
- [65] D. Ridout. 1957. Rational approximations to algebraic numbers. *Mathematika* 4 (1957), 125–131. <https://doi.org/10.1112/S0025579300001182>
- [66] K. F. Roth. 1955. Rational Approximations to Algebraic Numbers. *Mathematika* 2 (1955), 1–20.
- [67] S. M. Rump. 2010. Verification methods: Rigorous results using floating-point arithmetic. *Acta Numer.* 19 (2010), 287–449. <https://doi.org/10.1017/S096249291000005X>
- [68] 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.
- [69] 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>.
- [70] 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.
- [71] D. Stehlé, V. Lefèvre, and P. Zimmermann. 2005. Searching Worst Cases of a One-Variable Function Using Lattice Reduction. *IEEE Trans. Comput.* 54, 3 (March 2005), 340–346.
- [72] 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>
- [73] L. Turelier. 2022. *Extension of the SLZ algorithm to bivariate functions*. Research Report. INRIA Nancy. <https://inria.hal.science/hal-03740209>
- [74] 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.
- [75] A. Ziv. 1991. Fast evaluation of elementary mathematical functions with correctly rounded last bit. *ACM Trans. Math. Software* 17, 3 (Sept. 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 [59, §12.4.4], Remark 4.5), the logarithm function and the inverse trigonometric functions.

Following the works [61] and [41], Khémira and Voutier proved in [40] 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 $\text{htr}_{\text{exp}}(p)$.

Furthermore, the strong relationship between exp and many functions such as ln, 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

$$d_n = \text{l.c.m.}(1, \dots, n) \text{ and } \mathcal{D}_{m,n} = \frac{m!}{\prod_{\substack{q \leq n \\ q \text{ prime}}} q^{v_q(m!)}}$$

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

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

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

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

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

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

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

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 $D_{\alpha,\beta} = 1$, $\ln \mathcal{A}_{\alpha,\beta} = \max(0, p-1-e_\alpha) \ln 2$, $\ln \mathcal{B}_{\alpha,\beta} = \max(0, p-1-e_\beta) \ln 2$, 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 $|\cos(\alpha) - \beta| \leq \varepsilon$.*

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

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

These statements also hold for the sine function.

PROOF. We write

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

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 $D_{\alpha', \beta'} h(\alpha') - \ln(\max(1, |\alpha'|)) = D_{\alpha', \beta'} h(\alpha) - \ln(\max(1, |\alpha|)) \leq 2 \max(0, p-1-e_\alpha) \ln(2) + \ln(\max(1, |\alpha|))$.

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, $h(\beta') = \max(0, p-2-e_\beta) \ln(2)/2$; hence, $D_{\alpha', \beta'} h(\beta') - \ln(\max(1, |\beta'|)) \leq \max(0, p-2-e_\beta) \ln(2)$ in all cases.

Assume now that $1 - |\beta| \geq \delta$; then, $|(\beta + i\sqrt{1-\beta^2}) - (\beta - i\sqrt{1-\beta^2})| = 2\sqrt{1-\beta^2} \geq 2\sqrt{\delta}$. 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 $\geq \sqrt{\delta}$, and the other one is $\leq 2\varepsilon/\sqrt{\delta}$. 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 $1 - |\beta| \geq \delta, |(\sqrt{1-\beta^2} + i\beta) - (-\sqrt{1-\beta^2} + i\beta)| \geq 2\sqrt{\delta}$, so that for at least one $\beta' \in \{\pm\sqrt{1-\beta^2} + i\beta\}$, we have $|\exp(i\alpha) - \beta'| \leq 2\varepsilon/\sqrt{\delta}$. \square

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 $1 - |\beta| \geq 2^{-p}$.

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 $\leq 1 - 1.5 \cdot 10^{-68}$. We then use the better bound $\delta = 2^{-113}$.

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

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

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

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, $D_{\alpha', \beta'} \leq 2$ and

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

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 $\max(\ln |\beta'^{-1}|, \ln |\beta'|) = |\ln |\beta'||$. This shows that $h(\beta') \leq \frac{1}{2}(\max(0, p-2-e_\beta) \ln 2 + |\ln |\beta'||)$, so that

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

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

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

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

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

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

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

PROOF. We write

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

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 $h(\alpha') = \max(p-2-e_\alpha, 0) \ln(2) + \ln \max(2|\alpha|, 1) \leq \max(p-1-e_\alpha, 1) \ln 2 + \ln(\max(|\alpha|, 1))$.

The claim on $\ln \mathcal{A}_{\alpha', \beta'}$ 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 $\max(p-1-e_\beta, 0) \ln 2 + \frac{1}{2} \ln(1 + \beta^2)$, which concludes the proof for the tangent function.

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

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

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

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

PROOF. We have

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

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

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

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

which eventually gives the same estimate. \square