Exact Lookup Tables for the Evaluation of Trigonometric and Hyperbolic Functions

Hugues de Lassus Saint-Geniès, David Defour, Guillaume Revy

Exact Lookup Tables for the Evaluation of Trigonometric and Hyperbolic Functions

Hugues de Lassus Saint-Geniès, David Defour, and Guillaume Revy

Abstract—Elementary mathematical functions are pervasively used in many applications such as electronic calculators, computer simulations, or critical embedded systems. Their evaluation is always an approximation, which usually makes use of mathematical properties, precomputed tabulated values, and polynomial approximations. Each step generally combines error of approximation and error of evaluation on finite-precision arithmetic. When they are used, tabulated values generally embed rounding error inherent to the transcendence of elementary functions. In this article, we propose a general method to use error-free values that is worthy when two or more terms have to be tabulated in each table row. For the trigonometric and hyperbolic functions, we show that Pythagorean triples can lead to such tables in little time and memory usage. When targeting correct rounding in double precision for the same functions, we also show that this method saves memory and floating-point operations by up to 29% and 42%, respectively.

Index Terms—Elementary function approximation, table-based range reduction, exact lookup tables, computer arithmetic.

1 Introduction

The representation formats and the behavior of binary floating-point arithmetics available on general-purpose processors are defined by the IEEE 754-2008 standard [1]. For basic arithmetic operations such as $+$, $-$, $\times$, $\div$, $\sqrt{}$, or `fma`, this standard requires the system to return the *correct rounding* of the exact result, according to one of four rounding-direction attributes (to nearest ties to even, toward $-\infty$, toward $+\infty$, and toward 0): this property guarantees the quality of the result. However, due to the Table Maker's Dilemma (TMD) [2] and the difficulties in developing accurate and efficient evaluation schemes, correct rounding is only recommended for elementary functions defined in the IEEE 754-2008 standard.

The algorithms developed for the evaluation of elementary functions, such as the logarithm, the exponential, or the trigonometric and hyperbolic functions, can be classified into at least two categories. The first category concerns algorithms based on small and custom operators combined with tabulated values that target small accuracies, typically less than 30 bits [3], [4], [5]. The second category concerns algorithms that usually target single or double precision (24 or 53 bits of precision) with an implementation on general processors that relies on the available hardware units [6], [7].

Implementations of those functions that target correct rounding are usually divided into two or more phases [8]. A *quick phase* is first performed: it is based on a fast approximation which provides a few more bits than the targeted format, which makes correct rounding possible most of the time at a reasonable cost. When correct rounding is not possible, a much slower *accurate phase* is executed. The quick phase uses operations with a precision slightly greater than the targeted precision, while the accurate phase is based on a much larger precision. For example, in the correctly rounded library CR-Libm, the quick phase for the sine and cosine functions in double precision targets 66 bits while the accurate phase corresponds to 200 bits [6, § 9]. And then in order to guarantee that an implementation actually computes correctly rounded results, a proof of correctness has to be built. This proof is based on the mandatory number of bits required to ensure correct rounding, which is linked with the search for the worst cases for the TMD [9].

The design of such correctly rounded implementations is a real challenge, as it requires to control and limit every source of numerical error [10]. Indeed these implementations involve various steps, including range reduction, polynomial evaluation, and reconstruction. And at each of these steps, errors may occur. Since those errors accumulate and propagate up to the final result, any solution or algorithm that reduces them will have an impact on the simplicity of the proof of correctness and the performance of the implementation.

1.1 Evaluation of Elementary Functions

In this article, we deal with the implementation of trigonometric and hyperbolic functions. In order to better understand the challenges that have to be faced when designing such elementary function implementation, let us detail the function evaluation process with a focus on methods based on table lookups for trigonometric and hyperbolic sine and cosine. For this purpose, let y be a machine-representable floating-point number, taken as the input to the considered functions. The internal design of these implementations is based on the following four-step process:

1) A *first range reduction*, based on mathematical properties, narrows the domain of the function to a smaller one. For the trigonometric functions, properties of periodicity and symmetry lead to evaluating $f_q \in \{\pm\sin, \pm\cos\}$ at input
$$|x| = |y - q \cdot \pi/2| \in [0, \pi/4],$$

• *H. de Lassus Saint-Geniès, D. Defour, and G. Revy are with the DALI project-team, Université de Perpignan Via Domitia and LIRMM (CNRS: UMR 5506), Perpignan, France.*

Manuscript last updated March 2, 2017.

where $q \in \mathbb{Z}$ and the choice of f_q depends on the function to evaluate, $q \bmod 4$, and $\mathrm{sign}(x)$ [8]. For the hyperbolic functions, let

$$|x| = |y - q \cdot \ln(2)| \in [0, \ln(2)/2]\,,$$

with $q \in \mathbb{Z}$. Addition formulas can then be used together with the analytic expressions for the hyperbolic functions involving the exponential, which, for the sine, gives:

$$\begin{aligned}\sinh(y) = &\left(2^{q-1} - 2^{-q-1}\right) \cdot \cosh|x| \\ &\pm \left(2^{q-1} + 2^{-q-1}\right) \cdot \sinh|x|.\end{aligned}$$

2) A *second range reduction*, based on tabulated values, further reduces the range on which polynomial evaluations are to be performed. The argument x is split into two parts, x_h and x_ℓ, such that:

$$x = x_h + x_\ell \quad \text{with} \quad |x_\ell| \leq 2^{-p-1}. \tag{1}$$

The term x_h is the p-bit value of the form

$$x_h = i \cdot 2^{-p} \quad \text{with} \quad i = \lfloor x \cdot 2^p \rceil\,, \tag{2}$$

and where $i \in \mathbb{N}$. The integer i is used to address a table of $n = \left\lfloor \kappa \cdot 2^{p-1} \right\rceil + 1$ rows, with $\kappa \in \{\pi/2, \ln(2)\}$. This table holds precomputed values of either trigonometric or hyperbolic sines and cosines of x_h, which we indifferently name S_h and C_h.

3) Meanwhile, *polynomial approximations* to the trigonometric or hyperbolic sine and cosine on the interval $[-2^{p-1}, 2^{-p-1}]$, denoted by P_S and P_C, are evaluated at input x_ℓ.

4) Finally, a *reconstruction step* allows to compute the final result using the precomputed values retrieved at step 2 and the computed values from step 3. For the trigonometric sine, if we assume that $q = 0 \bmod 4$ and $\mathrm{sign}(x) = 1$ so that $f_q = \sin$, one has to perform the following reconstruction:

$$\sin(y) = S_h \cdot P_C(x_\ell) + C_h \cdot P_S(x_\ell),$$

while the reconstruction for the hyperbolic sine is:

$$\begin{aligned}\sinh(y) = &\left(2^{q-1} - 2^{-q-1}\right) \cdot \\ &\left(C_h \cdot P_C\left(x_\ell\right) + S_h \cdot P_S\left(x_\ell\right)\right) \\ &\pm \left(2^{q-1} + 2^{-q-1}\right) \cdot \\ &\left(S_h \cdot P_C\left(x_\ell\right) + C_h \cdot P_S\left(x_\ell\right)\right).\end{aligned}$$

Satisfactory solutions already exist to address the first range reduction [11], [12], [13], the generation and evaluation of accurate and efficient polynomial evaluation schemes [14], [15], [16], and the reconstruction step. The interested reader can find more details in [8].

In this article, we address the second range reduction based on tabulated values for the trigonometric and hyperbolic sine and cosine. At this step, each tabulated value embeds a rounding error. Our objective is to remove the error on these values, and consequently to concentrate the error due to range reduction in the reduced argument used in the polynomial evaluation. Precisely the proposed method relies on finding precomputed values with remarkable properties that simplify and accelerate the evaluation of these functions. These properties are threefold. First, each pair of values holds the exact images of a reference argument under the functions, that is, without any rounding error. For this purpose, we require these values to be rational numbers. Second, each numerator and denominator of these rational values should be exactly representable in a representation format available in hardware, that is, as an integer or a floating-point number. Third, all rational values must share the same denominator. This enables us to shift the division by this denominator into the polynomial coefficients. These three properties lead to tabulated values that are exact *and* representable on single machine words. For the trigonometric and hyperbolic functions, Pythagorean triples give rational numbers that fulfill these three properties.

1.2 Overview of this Article

In this article, we extend a previous work on the trigonometric functions $\sin$ and $\cos$ presented in [17] to the hyperbolic functions $\sinh$ and $\cosh$, and we show that our concept is general enough to be applied to other functions. More precisely, the contributions of this article are the following:

1) A general method that eliminates rounding errors from tabulated values;
2) And a formally justified algorithm to apply this method to trigonometric and hyperbolic functions.

This article is organized as follows: Section 2 gives some background on existing solutions for the second range reduction step. Section 3 details the properties of the proposed tables and demonstrates how it is possible to remove two sources of error involved during this step. Section 4 presents our approach to build exact lookup tables for trigonometric and hyperbolic functions using Pythagorean triples. Then, Section 5 presents some experimental results, which show that we can precompute exact lookup tables up to 10 indexing bits reasonably fast with the help of suitable heuristics. Some comparison results with classical approaches are given in Section 6, which show that our method can lower the table sizes and the number of floating-point operations performed during the reconstruction. Finally, an example with a toy table is also presented in Section 7, before a conclusion in Section 8.

2 Table-Lookup Range Reductions

While sometimes the second range reduction uses fast but inaccurate hardware approximations, in practice it is often implemented using table lookups.[1] This section presents three solutions that address this step with the sine function to illustrate them. Those methods are general and apply directly to the trigonometric cosine and the hyperbolic functions as well.

2.1 Tang's Tables

Tang proposed a general method to implement elementary functions that relies on hardware-tabulated values [18]. Given the reduced argument x as in Equation (1), Tang's method uses the upper part x_h to retrieve two tabulated

1. For instance, see the GNU `libm` in `glibc 2.25`.

values S_h and C_h that are good approximations of $\sin(x_h)$ and $\cos(x_h)$, respectively, rounded to the destination format. If an implementation targets correct-rounding, then those approximations are generally stored as *floating-point expansions* [19]. In practice, an expansion of size n consists in representing a given number as the unevaluated sum of n floating-point numbers so that the rounding error be reduced compared to a regular floating-point number. If we denote by $\circ_i(x)$ the rounded value of a real number x to the nearest floating-point number of precision i, and by a real number ε_{-i} the rounding error such that $|\varepsilon_{-i}| \leq 2^{-i}$, then, for the trigonometric functions, we have:

$$\begin{aligned} S_h &= \circ_{53j}(\sin(x_h)) = \sin(x_h) \cdot (1 + \varepsilon_{-53j}) \\ \text{and } C_h &= \circ_{53j}(\cos(x_h)) = \cos(x_h) \cdot (1 + \varepsilon_{-53j}), \end{aligned}$$

where j is the number of non-overlapping floating-point numbers used to represent S_h and C_h.

In parallel to the extraction of the values S_h and C_h, the evaluation of two polynomial approximations $P_S(x)$ and $P_C(x)$ is performed. They respectively approach the sine and cosine functions over the interval covered by x_ℓ, namely $[-2^{-p-1}, 2^{-p-1}]$. Finally, the result of $\sin(x)$ is reconstructed as follows:

$$\sin(x) \approx S_h \cdot P_C(x_\ell) + C_h \cdot P_S(x_\ell).$$

Tang's method is well suited for hardware implementations on modern architectures. It takes advantage of the capability on these architectures to access tabulated values in memory and to perform floating-point computations concurrently. Once the argument x is split into the two parts x_h and x_ℓ, memory units can provide the two tabulated values S_h and C_h, while floating-point units evaluate the polynomials P_S and P_C. As the degree of the polynomials decreases when the table size increases, the objective is to find parameters so that the polynomial evaluations take as long as memory accesses, on average [20].

Tang's tables store images of regularly spaced inputs, rounded to the destination format. This rounding error is problematic when seeking a correctly rounded implementation in double precision, since worst cases for trigonometric and hyperbolic functions require more than 118 bits of accuracy [21], [22]. A solution consists in storing values on three double-precision floating-point numbers [6] and using costly extended-precision operators.

2.2 Gal's Accurate Tables

In Tang's method, S_h and C_h are approximations of $\sin(x_h)$ and $\cos(x_h)$, respectively. They are rounded according to the format used in the table and the targeted accuracy for the final result. To increase the accuracy of these tabulated values, Gal suggested a method to transfer some of the errors due to rounding over the reduced argument [23]. This consists in introducing small corrective terms on the values x_h, hereafter denoted by $corr$. For each input entry x_h of the table, one $corr$ term is carefully chosen to ensure that both $\sin(x_h + corr)$ and $\cos(x_h + corr)$ are "very close" to floating-point machine numbers. In [24], Gal and Bachelis were able to find $corr$ values such that

$$\begin{aligned} S_h &= \circ_{53j}(\sin(x_h + corr)) \\ &= \sin(x_h + corr) \cdot (1 + \varepsilon_{-10-53j}) \\ \text{and } C_h &= \circ_{53j}(\cos(x_h + corr)) \\ &= \cos(x_h + corr) \cdot (1 + \varepsilon_{-10-53j}) \end{aligned}$$

for each row of an 8-bit-indexed table. This corresponds to 10 extra bits of accuracy for both tabulated values of sine and cosine compared to Tang's tabulated values, thanks to a small perturbation $corr$ on the values x_h. For the trigonometric and hyperbolic functions, such $corr$ values are produced by a random sampling with an expected sample size of 2^{18} [25]. This method entails storing the corrective terms $corr$ along with the values S_h and C_h, which will usually make Gal's accurate tables larger than Tang's tables when targeting correct rounding. The value $\sin(x)$ is reconstructed as follows:

$$\sin(x) = S_h \cdot P_C(x_\ell - corr) + C_h \cdot P_S(x_\ell - corr),$$

where $P_C(x)$ and $P_S(x)$ are polynomials approximating $\cos(x)$ and $\sin(x)$, respectively, on the interval covered by $x_\ell - corr$.

Gal's solution requires an exhaustive search in order to find one corrective term for each entry x_h, within a search space that grows exponentially with the number of extra bits for S_h and C_h. Stehlé and Zimmermann proposed an improvement based on lattice reduction and worst cases to accelerate the precomputation [25]. They were able to increase the accuracy of Gal's tables by 11 extra bits, which translates to 21 extra bits compared to Tang's tables.

2.3 Brisebarre et al.'s (M, p, k)-Friendly Points

In 2012, Brisebarre, Ercegovac, and Muller proposed a new method for trigonometric sine and cosine evaluation with a few table lookups and additions in hardware [26], [27]. Their approach consists in tabulating four values a, b, z, and $\hat{x}$, defined as:

$$z = 1/\sqrt{a^2 + b^2} \quad \text{and} \quad \hat{x} \approx \arctan(b/a),$$

where a and b are *small* particular integers. The reconstruction then corresponds to:

$$\sin(x) = \big(b \cdot \cos(x - \hat{x}) + a \cdot \sin(x - \hat{x})\big) \cdot z.$$

The values $(a, b, z, \hat{x})$ are chosen amongst specific points with integer coordinates (a, b) called (M, p, k)-friendly points. These points are recoded using the *canonical recoding*, which minimizes the number of non-zero digits in table entries [28]. More precisely, a and b must be positive integers lower than M, so that the number $z = 1/\sqrt{a^2 + b^2}$ has less than k non-zero bits in the first p bits of its *canonical recoding*. These requirements make hardware multiplications by a, b, and z much more acceptable, since they are equivalent to just a few additions. Compared to other hardware evaluation schemes, this solution reduces by 50% the area on FPGAs for 24-bit accuracy. Overall, this is a suitable method for competitive low and medium-precision hardware approximations.

2.4 Exact Lookup Table for Trigonometric Functions

Gal's accurate tables presented in Section 2.2 consist in finding almost regularly spaced inputs for which the images by the functions to approximate are close to machine numbers. This virtually increases the accuracy of the stored values by a few extra bits, slightly reducing the error bounds in the reconstruction step. However, there are still errors in the reduced argument as well as in tabulated approximations.

In 2015, we proposed a method to tabulate error-free values for the sine and cosine functions [17]. It is an improvement over Gal's tables as tabulated approximations for the sine and cosine are stored exactly, so that extended precision computations involving these values shall be less expensive. This can be noticed through the accurate phase costs presented in the article. In an exact lookup table the corrective terms are inexact, but they are added to the reduced argument, which is already inexact. As this addition can be performed with arbitrary precision, the presented method *concentrates* the error in a single term besides freeing tabulated values from rounding error. This reduces the memory footprint of the table when targeting high precision, compared to Tang's and Gal's tables, which need to store asymptotically twice as many bits.

It is similar to the (M, p, k)-friendly points method, as it relies on integer values like a and b in Section 2.3. It differs from it though, since the equivalent of $1/z$ is made exactly representable as an integer. This allows for a reconstruction step like Gal's, at the difference that it involves two exact terms out of four. This improvement was made possible by using the interesting structure of Pythagorean triples [29].

3 Design of an Exact Lookup Table

In this article, the proposed improvement is based on the following observation: After the first range reduction, the reduced number x is assumed to be an irrational number. Therefore, it has to be rounded to some finite precision, which means that after the second range reduction, only x_ℓ contains a rounding error. Gal's method adds an exact corrective term $corr$ to x_ℓ that allows to increase the accuracy of transcendental tabulated values. Instead, we suggest not to worry about *inexact* corrective terms, as long as they make tabulated values *exactly representable*. This way, the error is solely concentrated in the reduced argument $x_\ell - corr$ used during the polynomial evaluations.

Now, let us characterize these exactly representable values and then explain how the evaluation scheme can benefit of this property. For this purpose, let f and g be the functions to approximate, $\kappa \in \{\pi/2, \ln(2)\}$ the additive constant for the first range reduction, and y an input floating-point number. As seen in the introduction, the reduced argument x obtained after the first range reduction is such that

$$x = |y - q \cdot \kappa| \quad \text{with} \quad q \in \mathbb{Z}, 0 \leq x \leq \kappa/2.$$

As y is rational, q is an integer, and κ is irrational, then unless $q = 0$, x must be irrational, and it has to be rounded to a floating-point number $\hat{x}$, with a precision j greater than the targeted format such that: $\hat{x} = x \cdot (1 + \varepsilon_{-j})$. We should mention that $\hat{x}$ is generally represented as a floating-point expansion of size 2 or more to reach an accuracy of at least j bits. As seen in Equation (1), the second range reduction splits $\hat{x}$ into two parts, x_h and x_ℓ, such that

$$\hat{x} = x_h + x_\ell \quad \text{and} \quad |x_\ell| \leq 2^{-p-1}.$$

As shown in Equation (2), the value x_h is then used to compute an address i in the table T made of n rows. This table T holds exact precomputed values of $k \cdot f(x_h + corr_i)$ and $k \cdot g(x_h + corr_i)$, where k is an integer scale factor that makes both tabulated values integers, and $corr_i$ is an irrational corrective term, precomputed for each table entry i, such that $|corr_i| \leq 2^{-p-1}$. To build such values, we assume the following properties:

1) The functions f and g are right invertible on the domain $[0, (n - 1/2) \cdot 2^{-p}]$. This allows corrective terms to be determined.
2) Corrective terms are such that $|corr_i| \leq 2^{-p-1}$.
3) For each table entry i, the values $f(x_h + corr_i)$ and $g(x_h + corr_i)$ are *rational* numbers with the same denominator, that is:
$$f(x_h + corr_i) = \frac{\eta_i}{k_i} \quad \text{and} \quad g(x_h + corr_i) = \frac{\gamma_i}{k_i}$$
with $\eta_i, \gamma_i \in \mathbb{Z}$ and $k_i \in \mathbb{Z}^*$.
4) Let $k = \operatorname{lcm}(k_0, \ldots, k_{n-1})$, i.e. the least common multiple (LCM) of the denominators k_i. Let $F_i = k \cdot f(x_h + corr_i)$ and $G_i = k \cdot g(x_h + corr_i)$. It is clear that the values $F_i = k \cdot \eta_i/k_i$ and $G_i = k \cdot \gamma_i/k_i$ are integers, but we will also assume that they are representable as one floating-point machine number each, e.g. they fit on 53 bits if using the binary64/double precision format.

With numbers satisfying those properties, the reconstruction step corresponds to

$$f(x) = G_i \cdot P_{f/k}(x_\ell - corr_i) + F_i \cdot P_{g/k}(x_\ell - corr_i)$$

for some i in $\{0, \ldots, n-1\}$, where

- $P_{f/k}(x)$ and $P_{g/k}(x)$ are polynomial approximations of the functions $f(x)/k$ and $g(x)/k$, respectively, on the interval
$$[-2^{-p-1} - \max_i(corr_i), 2^{-p-1} - \min_i(corr_i)].$$
- and F_i, G_i, and $\circ_j(corr_i)$ are gathered from T. The integers F_i and G_i are stored without error as one floating-point number each. The third tabulated value $\circ_j(corr_i)$ is an approximation of the corrective term $corr_i$ such that:
$$x_h + corr_i \in f^{-1}\left(\frac{F_i}{k}\right) \cup g^{-1}\left(\frac{G_i}{k}\right)$$
rounded to the targeted precision j.

As can be seen, the new reduced argument $x_\ell - corr_i$ covers a wider range than just x_ℓ. In the worst case, this interval can have its length doubled, making the polynomial approximations potentially more expensive. Actually, those polynomials approximate functions that usually do not variate much on such reduced intervals, so that their degree need rarely be increased to achieve the same precision as for an interval that may be up to twice as narrow. This was tested with some of our tables with Sollya's `guessdegree`

function [30]. In unfortunate cases for which the degree should be increased, rarely more than one additional monomial will be required, which barely adds two floating-point operations (one addition and one multiplication). Furthermore, such additional operations can often benefit from instruction parallelism in the evaluation scheme.

Also note that instead of considering polynomial approximations of $f(x)$ and $g(x)$ directly, we propose to incorporate a division by k into the polynomial coefficients. Therefore, we consider approximations $P_{f/k}(x)$ and $P_{g/k}(x)$ of $f(x)/k$ and $g(x)/k$, respectively, which avoid the prohibitive cost of the division and the associated rounding error.

4 Exact Lookup Tables for Trigonometric and Hyperbolic Functions

The proposed lookup table for the second range reduction brings several benefits over existing solutions. However, building such a table of error-free values is not trivial, since the integers F_i, G_i, and k are not always straightforward, especially for transcendental functions. For the trigonometric and hyperbolic functions, we rely on some useful results on *Pythagorean triples*. These objects are briefly described in the first part of this section. Then, we present a method to efficiently build exact lookup tables for the trigonometric and hyperbolic functions.

4.1 Pythagorean Triples

Pythagorean triples are a set of mathematical objects from Euclidean geometry which have been known and studied since ancient Babylonia [31, ch. 6]. There exist several definitions of Pythagorean triples that usually differ on authorized values. We choose the following one:

Definition 1. *A triple of non-negative integers $(a, b, c) \neq \vec{0}$ is a* Pythagorean triple *if and only if $a^2 + b^2 = c^2$.*

A Pythagorean triple (a, b, c) for which a, b, and c are coprime is called a *primitive Pythagorean triple* (PPT). In the following, we will refer to the set of PPTs as $\mathbb{PPT}$. Recall that we are looking for *several* rational numbers which hold exact values for the trigonometric and hyperbolic functions. Yet a PPT and its multiples share the same rational values a/b, $b/c, \ldots$ For example, the well known PPT $(3, 4, 5)$ and all its multiples can be associated to the ratio $a/b = 3/4$. Therefore we can restrict our search to primitive Pythagorean triples only, and apply a scale factor afterwards if needed.

According to the fundamental trigonometric and hyperbolic identities, we have

$$\forall x \in \mathbb{R}, \begin{cases} \cos(x)^2 + \sin(x)^2 = 1 \\ \cosh(x)^2 - \sinh(x)^2 = 1. \end{cases} \tag{3}$$

It follows from Definition 1 and Equation (3) that all Pythagorean triples can be mapped to rational values of trigonometric or hyperbolic sine and cosine. Indeed, let (a, b, c) be a Pythagorean triple. Without loss of generality, we can assume that $b \neq 0$. Then we have:

$$\begin{aligned} a^2 + b^2 = c^2 &\iff \left(\frac{a}{c}\right)^2 + \left(\frac{b}{c}\right)^2 = 1 \\ &\iff \left(\frac{c}{b}\right)^2 - \left(\frac{a}{b}\right)^2 = 1. \end{aligned}$$

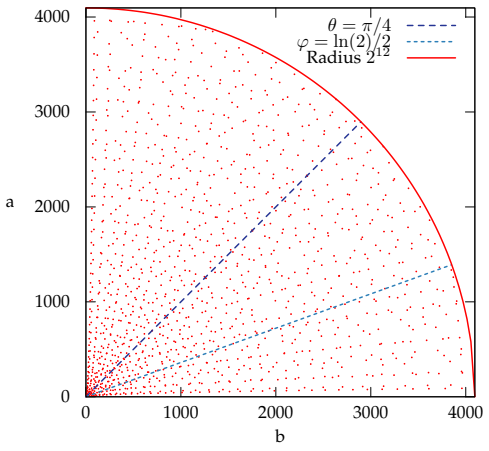

Fig. 1. Primitive Pythagorean triples with a hypotenuse $c < 2^{12}$.

Hence, for each Pythagorean triple (a, b, c), assuming $b \neq 0$,

$$\begin{aligned} \exists \theta \in [0, \pi/2[\,, \quad \cos(\theta) = \frac{b}{c} \quad \text{and} \quad \sin(\theta) = \frac{a}{c} \\ \exists \varphi \in \mathbb{R}_{\geq 0}, \quad \cosh(\varphi) = \frac{c}{b} \quad \text{and} \quad \sinh(\varphi) = \frac{a}{b}. \end{aligned}$$

The consequences of these properties are twofold: Firstly a PPT and its multiples share the same angles θ and φ. Secondly any Pythagorean triple can be mapped to the sides of a right triangle (possibly degenerated into a segment), the hypotenuse of which is the third item c of the triple. Hence, in the following, the word "hypotenuse" is used to refer to the third item of a Pythagorean triple, while the word "legs" is used to refer to its first and second items.

4.2 Building Subsets of Primitive Pythagorean Triples

The set of primitive Pythagorean triples $\mathbb{PPT}$ is unbounded. Figure 1 represents the subset of all PPTs with a hypotenuse $c < 2^{12}$. It shows that PPTs rapidly cover a wide range of angles over $[0, \pi/2]$ as c increases. But, as our exact lookup tables need one triple per row, one may ask if there will *always* be at least one PPT that matches each row, no matter how *narrow* the entry interval of a row can be.

Actually, a result due to P. Shiu states that the set of trigonometric angles covered by Pythagorean triples is dense in $[0, \pi/4]$. This is formalized by Theorem 1.

Theorem 1 (Shiu 1983 [32]). *$\forall \theta \in [0, \pi/4], \forall \delta > 0$, there exists a primitive Pythagorean triple with a corresponding trigonometric angle θ', such that*

$$|\theta - \theta'| < \delta.$$

Proof. (From [32].) Let $(a, b, c) \in \mathbb{PPT}$. It is well-known that, assuming that a is even, (a, b, c) can be rewritten $(2mn, m^2 - n^2, m^2 + n^2)$ where m, n are coprime integers satisfying $m > n > 0$. It follows that

$$\tan(\theta') = \frac{m^2 - n^2}{2mn} = \frac{1}{2}\left(\frac{m}{n} - \frac{n}{m}\right).$$

Let us write $t = \tan(\theta')$ and $r = m/n$ so that we have $r^2 - 2tr - 1 = 0$ and hence

$$r = t + \sqrt{t^2 + 1} = \tan(\theta') + \sec(\theta').$$

Note that we do not have $r = t - \sqrt{t^2+1}$ because $r > 0$.

It is now easy to prove the theorem. Let $0 \leq \theta \leq \pi/4$ and $u = \tan(\theta) + \sec(\theta)$. One can choose a sequence of rational numbers $r_0, r_1, r_2, \ldots$ converging to u, where

$$r_k = \frac{m_k}{n_k}, \quad k = 0, 1, 2, \ldots$$

such that m_k and n_k are positive and coprime. Let $x = 2m_k n_k, y = m_k^2 - n_k^2, z = m_k^2 + n_k^2$ with corresponding angle θ_k. The angles $\theta_0, \theta_1, \theta_2, \ldots$ tend to θ, therefore the θ_k's can approximate θ arbitrarily closely, as required. □

By symmetry of the Pythagorean triples (a, b, c) and (b, a, c), the density of the aforementioned set can be extended to the interval $[0, \pi/2]$. A corollary is the density of the set of hyperbolic angles covered by Pythagorean triples in $\mathbb{R}_+$. This is stated by Corollary 1 below.

Corollary 1. *$\forall \varphi \in \mathbb{R}_+, \forall \delta > 0$, there exists a primitive Pythagorean triple with an associated hyperbolic angle φ', such that*

$$|\varphi - \varphi'| < \delta.$$

Proof. By analogy, replace $\tan(\varphi')$ by $\sinh(\varphi')$, and $\sec(\varphi')$ by $\cosh(\varphi')$ in Shiu's proof of Theorem 1. We have

$$\sinh(\varphi') = \frac{m^2 - n^2}{2mn} = \frac{1}{2}\left(\frac{m}{n} - \frac{n}{m}\right).$$

Let us write $s = \sinh(\varphi')$ and $r = m/n$ so that we have $r^2 - 2sr - 1 = 0$ and hence

$$r = s + \sqrt{s^2+1} = \sinh(\varphi') + \cosh(\varphi').$$

Let $\varphi \geq 0$ and $u = \sinh(\varphi) + \cosh(\varphi)$. One can choose a sequence of rational numbers $r_0, r_1, r_2, \ldots$ converging to u, where

$$r_k = \frac{m_k}{n_k}, \quad k = 0, 1, 2, \ldots$$

such that m_k and n_k are positive and coprime. Let $x = 2m_k n_k, y = m_k^2 - n_k^2, z = m_k^2 + n_k^2$ with corresponding hyperbolic angle φ_k. It follows that the hyperbolic angles $\varphi_0, \varphi_1, \varphi_2, \ldots$ tend to φ and that the φ_k's can approximate φ arbitrarily closely, as required. □

Although we are now certain that there will always be an infinite number of PPTs for each row of our tables, these theorems do not give any bounds on the size of the PPTs. In practice, we will see that the PPT sizes allow us to build tables in double precision indexed by usual numbers of indexing bits (up to 13 or so). Also, we still have to find an easy means to generate PPTs efficiently. In this article, we make use of the Barning-Hall tree [33], [34], which exhibits a ternary structure that links any PPT to three different PPTs. From any PPT represented as a column vector, the Barning-Hall tree allows to compute three new PPTs by multiplying the former with the matrices

$$\begin{pmatrix} 1 & -2 & 2 \\ 2 & -1 & 2 \\ 2 & -2 & 3 \end{pmatrix}, \begin{pmatrix} -1 & 2 & 2 \\ -2 & 1 & 2 \\ -2 & 2 & 3 \end{pmatrix}, \text{ and } \begin{pmatrix} 1 & 2 & 2 \\ 2 & 1 & 2 \\ 2 & 2 & 3 \end{pmatrix}. \quad (4)$$

It has been proven that all PPTs can be generated from the root $(3, 4, 5)$ with increasing hypotenuse lengths [35]. For every generated PPT (a, b, c), we also consider its *symmetric* PPT (b, a, c), because it may be interesting for three reasons:

1) For the trigonometric functions,
   a) If $\arcsin(b/c) > \pi/4$, one has immediately $\arcsin(a/c) < \pi/4$, which falls into the range of the exact lookup table.
   b) Whenever $\arcsin(b/c) \approx \pi/4$, one also has $\arcsin(a/c) \approx \pi/4$, and both triples may either fall into two different subintervals of the exact lookup table, which can help reduce the value k (since they share a common hypotenuse), or fall into the same subinterval, which can help find a better corrective term.
2) For the hyperbolic functions, since the reduced argument x lies in $[0, \ln(2)/2]$, interesting PPTs must satisfy $a/b \in \sinh([0, (n + 1/2) \cdot 2^{-p}]) \supset [0, \sqrt{2}/4]$. Therefore, if $a \geq 4b/\sqrt{2}$, only the symmetric PPT (b, a, c) falls into the range of the lookup table.

Hence the first step of PPT generation using the Barning-Hall tree is the following: multiplying the matrices in Equation (4) by the root $(3, 4, 5)$ taken as a column vector, one gets the three new PPTs $(5, 12, 13)$, $(15, 8, 17)$, and $(21, 20, 29)$, and their symmetric counterparts $(12, 5, 13)$, $(8, 15, 17)$, and $(20, 21, 29)$. In the following, note that we always consider the "degenerated" PPT $(0, 1, 1)$, because it gives us an exact corrective term for the first table entry, without making the LCM k grow.

4.3 Selection of Primitive Pythagorean Triples

For each row i of the table indexed by $\lfloor x_h \cdot 2^p \rceil$, we want to select exactly one PPT (a, b, c) with a corresponding value $\theta = \arcsin(a/c)$ or $\varphi = \operatorname{asinh}(a/b)$ such that:

$$|x_h - \theta| < 2^{-p-1} \quad \text{for trigonometric functions, or}$$
$$|x_h - \varphi| < 2^{-p-1} \quad \text{for hyperbolic functions.}$$

The existence of such values θ and φ is a consequence of Theorem 1 and its Corollary 1.

In the following, we define our *corrective terms*, which we denote by $corr$, as:

$$corr = \theta - x_h \quad \text{for trigonometric functions, or}$$
$$corr = \varphi - x_h \quad \text{for hyperbolic functions.}$$

Once one PPT has been selected for each row i, a naive solution would consist in storing exactly each a_i, b_i, c_i in the lookup table T, plus an approximation of $corr$ on as many bits as necessary. Instead, as presented in Section 3, we suggest to store two integers C_h and S_h of the form

$$C_h^{(i)} = \frac{b_i}{c_i} \cdot k \quad \text{and} \quad S_h^{(i)} = \frac{a_i}{c_i} \cdot k \quad (5)$$

for the trigonometric functions, or

$$C_h^{(i)} = \frac{c_i}{b_i} \cdot k \quad \text{and} \quad S_h^{(i)} = \frac{a_i}{b_i} \cdot k \quad (6)$$

for the hyperbolic functions, where $k \in \mathbb{N}^*$ is the same for all table entries. In this article, in order to reduce the memory footprint of the table T, we look for *small* table entries C_h and S_h. This entails looking for a value k that is rather small,

which can be simplified as a search for a small value in a set of least common multiples.

It was shown in [17] that a straightforward approach, consisting in computing the set of all possible LCMs, was quickly out of reach with current technology, as soon as tables had more than a few rows.

To reduce the computational time, the solution proposed in [17] consists in looking for an LCM directly *amongst* generated values. We call this solution "exhaustive search" in the sequel of this article, as it always gives the smallest LCM. This claim may be counter-intuitive since one may ask why the smallest LCM would necessarily appear as the denominator side of some PPT. To prove it, we start by proving that for any table, there is a set of PPTs that fills it, for which the denominator sides (either hypotenuses or bigger legs) have the least LCM possible:

Lemma 1. *If $k \in \mathbb{N}^*$ is the least element in the set of possible LCMs for a table of n rows, then there is a set of* primitive *Pythagorean triples $\{T_i = (a_i, b_i, c_i)\}_{i\in\{0,\dots,n-1\}}$, with T_i belonging to row i, such that:*

$$\operatorname{lcm}(c_0, \dots, c_{n-1}) = k \quad \textit{for the trigonometric functions, and}$$
$$\operatorname{lcm}(b_0, \dots, b_{n-1}) = k \quad \textit{for the hyperbolic functions.}$$

Proof. By construction. For readability, and without loss of generality, we will only consider the LCM for the trigonometric functions. The set of possible LCMs for a table T of n rows is non-empty as a consequence of Theorem 1. Thus, by the Well-Ordering Principle, we know that k exists.

Now, let $\{T'_0, \dots, T'_{n-1}\}$ be a set of primitive *or* non-primitive Pythagorean triples that fill the table T and such that $\operatorname{lcm}(c'_0, \dots, c'_{n-1}) = k$. If there exists $i \in \{0, \dots, n-1\}$ such that T'_i is non primitive, then there exists an integer $\alpha > 1$ and $T_i \in \mathbb{PPT}$ such that $T'_i = \alpha \cdot T_i$. Thus, we have

$$k = \operatorname{lcm}(c'_0, \dots, \alpha \cdot c_i, \dots, c'_{n-1}).$$

Hence α divides k. Since k is the *least* element of the set of possible LCMs, $k/\alpha < k$ cannot be a valid LCM. Therefore the prime factors of α must be shared with other c'_j. Hence, it is possible to write

$$k = \operatorname{lcm}(c'_0, \dots, c_i, \dots, c'_{n-1}).$$

As mentioned in Section 4.1, T_i and T'_i share the same angles θ and φ, and consequently they belong to the same subdivision of the table T. By repeating this process c'_i by c'_i as long as necessary, one can construct a set of *primitive* Pythagorean triples $\{T_i\}_{i\in\{0,\dots,n-1\}}$ such that $k = \operatorname{lcm}(c_0, \dots, c_{n-1})$, which concludes the proof. □

Second, let us recall two important results about primitive Pythagorean triples.

Theorem 2. *Let n be a positive integer. First n is the hypotenuse of a primitive Pythagorean triple if and only if all of its prime factors are of the form $4k + 1$. Second, when $n > 2$, n is a leg of a primitive Pythagorean triple if and only if $n \not\equiv 2 \mod 4$.*

Proof. For the first result, a proof is given by Sierpiński in [36, ch. XI.3]. A more recent one can be found in [37]. For the second result, different proofs are given in [36, ch. II.3], and [38, p. 116] □

Now, for the trigonometric functions, we formulate Theorem 3, which states that the smallest LCM k is the hypothenuse of a PPT.

Theorem 3. *If $k \in \mathbb{N}^*$ is the least element of the set of possible LCMs for a table of n rows for trigonometric functions, then k is the hypotenuse of a* primitive *Pythagorean triple.*

Proof. Lemma 1 tells us that each T_i can be made primitive. Thus, by Theorem 2, we have

$$\forall i \in \{0, \dots, n-1\},\ \exists \{\beta_{p,i}\}_{p\in\mathcal{P}_\pi} \in \mathbb{N}^{\mathcal{P}_\pi},\ c_i = \prod_{p\in\mathcal{P}_\pi} p^{\beta_{p,i}}$$

where $\mathcal{P}_\pi$ is the set of Pythagorean primes, i.e. the set of prime numbers congruent to 1 modulo 4. By construction, we have

$$k = \operatorname{lcm}(c_0, \dots, c_{n-1}) = \prod_{p\in\mathcal{P}_\pi} p^{\max_i(\beta_{p,i})},$$

which means that k is a product of Pythagorean primes. By Theorem 2, k is the hypotenuse of at least one primitive Pythagorean triple, which concludes the proof. □

Finally, for the case of hyperbolic functions, Theorem 4 shows that the smallest LCM k is a leg of a PPT.

Theorem 4. *If $k \in \mathbb{N}^*$ is the least element of the set of possible LCMs for a table of n rows for hyperbolic functions, then k is a leg of a* primitive *Pythagorean triple.*

Proof. By Theorem 2, we only need to prove that $k \not\equiv 2 \mod 4$.

Using Lemma 1, we know that each $T_i = (a_i, b_i, c_i)$ can be made primitive. Thus, using the aforementioned equivalence, we have

$$\forall i \in \{0, \dots, n-1\},\ b_i \not\equiv 2 \mod 4. \tag{7}$$

Hence, either every b_i is odd, or there exists $i \in \{0, \dots, n-1\}$ such that b_i is even. In the first case, the LCM k of all b_i's is obviously odd too. In the latter case, b_i is even implies that it is a multiple of 4 as a consequence of Equation (7), which entails that k is also a multiple of 4. Therefore, by construction,

$$k = \operatorname{lcm}(b_0, \dots, b_{n-1}) \not\equiv 2 \mod 4,$$

which concludes the proof. □

Theorems 3 and 4 justify why we call our search amongst generated hypotenuses or legs "exhaustive", because by searching *exhaustively* amongst increasing PPT hypotenuses or legs, the smallest LCM *will* be found eventually.

5 Implementation and Numeric Results

In this section, we present how we implemented the proposed method to generate exact lookup tables that have been described in Section 4. We have designed two solutions to look for a small common multiple k. The first solution is based on an *exhaustive* search and allows us to build tables indexed by up to 7 bits in a reasonable amount of time. The second solution uses a *heuristic* approach, which reduces memory consumption during the execution and the search time, allowing us to build tables indexed by 10 bits much faster than the exhaustive search.

TABLE 1
Exhaustive Search Results for $\sin$ and $\cos$.

p	k	n	time (s)	PPTs	Hypotenuses
3	425	9	$\ll 1$	87	33
4	5525	13	$\ll 1$	1405	428
5	160,225	18	0.2	42,329	11,765
6	1,698,385	21	7	335,345	87,633
7	6,569,225	23	31	1,347,954	335,645
8	$> 2^{27}$	> 27	> 6700	$> 21{,}407{,}993$	$> 4{,}976{,}110$

TABLE 2
Exhaustive Search Results for $\sinh$ and $\cosh$.

p	k	n	time (s)	PPTs	Bigger legs
3	144	8	$\ll 1$	23	12
4	840	10	$\ll 1$	86	43
5	10,080	14	$\ll 1$	1202	610
6	171,360	18	9	18,674	9312
7	1,081,080	21	328	147,748	72,476
8	$> 2^{24}$	> 24	$> 60{,}000$	$> 1{,}188{,}585$	$> 574{,}800$

5.1 Exhaustive Search

This subsection first describes the design of our exhaustive algorithm, then discusses the different choices that we have when several PPTs are available inside one table row for the found LCM. Finally, numeric results are presented for various precisions p along with two toy exact lookup tables for the trigonometric and hyperbolic functions.

5.1.1 Algorithm

As seen in Section 4.3, we restrain our search to the set of generated hypotenuses c or to the set of generated legs b, for the trigonometric and hyperbolic functions, respectively. In other words, we require that the LCM k be the hypotenuse or a leg of one of the generated PPTs. Moreover, for the hyperbolic functions, one has $\sinh(\ln(2)/2) < 1$, so that the search can safely be limited to the set of bigger legs only.

To perform such an exhaustive search, we designed a C++ program that takes as input the number p of bits used to index the table T and tries to generate an exact lookup table. To achieve this, the algorithm looks for the smallest integer k, amongst the generated values, that is a multiple of at least one PPT hypotenuse or leg per entry. By Theorems 3 and 4, this search is guaranteed to find the smallest LCM. The algorithm is the following: Start with a maximum denominator size (in bits) of $n = 4$ and then follow these four steps:

1) Generate all PPTs (a, b, c) such that $c \leq 2^n$ or $c \leq 2^n / \cos\left((N - 1/2) \cdot 2^{-p}\right)$, where N is the number of table entries, for the trigonometric and hyperbolic functions, respectively. The latter inequality guarantees that every interesting leg for the search step will be generated. It is during this step that the Barning-Hall matrices from Equation (4) are used.
2) Store only the PPTs that belong to a table entry, i.e. the potentially interesting ones.
3) Search for the LCM k among the PPT hypotenuses c or legs b that lie in $[2^{n-1}, 2^n]$, for the trigonometric or the hyperbolic functions, respectively.
4) If such a k is found, build values $(S_h, C_h, \circ_j(corr_i))$ for every row using Equation (5) or (6) and return an exact lookup table. Otherwise, set $n \leftarrow n + 1$ and go back to step 1.

Sometimes, several primitive Pythagorean triples are possible for a same table row. Then the selection of only one of them depends on which goal we want to achieve. In this article, we made the choice of selecting the one for which $\arcsin(a/c)$ or $\operatorname{asinh}(a/b)$ is the closest to x_h, as this shifts the bits of the corrective term to the right compared to the reduced argument x_ℓ. This virtually increases the precision of the "corrected" reduced argument, by having null bits in front of the mantissa of the corrective term.

But we see at least three alternatives to this solution. First, we could try to minimize the value $\max_i(corr_i) - \min_j(corr_j)$. This could be interesting to store smaller corrective terms $\delta_i = corr_i - \min_j(corr_j)$, and incorporate $\min_j(corr_j)$ into the polynomial approximations. But minimizing such a value might be much more expensive as its time complexity is $\prod_{i=0}^{n-1} m_i$, where m_i is the number of possible triples for entry i. Second, we could choose the PPT for which the corrective term has as many identical bits (0's or 1's) as possible after its least significant bits. If there are enough, the precision of the corrective term is also virtually extended, which could allow for smaller tables. Finally, we could relax the constraints on the k_i's to allow greater values of their LCM k, so that the corrective terms may have even lower magnitudes or be even closer to machine numbers. This can be done in different fashions, but we will only describe one to explain the idea to the reader. Let us assume that we want a 10-bit-indexed table T. What we do is generate a $10 + m$-bit-indexed table T' instead: T' has roughly 2^m times as many rows as the table that we want. Now let us make sets of about 2^m contiguous rows in this table. A subsequent step selects among each set of rows which of the available corrective terms is the best fit for the table T, and merges the rows into a single one for T. This technique could increase the precision of each corrective term at the cost of greater PPTs. As long as the exact stored values are still machine-representable, then this method stays interesting for software implementations. However, for hardware implementations, this might be less interesting since it involves more storage bits than a regular table.

5.1.2 Numeric Results

Tables 1 and 2 show the results obtained for the trigonometric and hyperbolic functions, respectively. Timings were measured on a server with an Intel® Xeon® E5-2650 v2 @ 2.6 GHz processor (16 physical cores) with 125 GB of RAM running on GNU/Linux. For the number p of bits that is targeted, the tables describe the value k that was found, followed by the number n of bits used to represent k (that is, $n = \lceil \log_2(k) \rceil$), the time (in seconds) taken by our program, and the numbers of PPTs and denominators considered during the LCM search.

As can be seen, it was possible to find k and to build tables indexed by up to $p = 7$ bits in a reasonable amount of time. However, it is clear that the number of dynamic memory allocations, which are mainly used to store the

TABLE 3
A trigonometric exact lookup table computed for $p = 4$.

Index	S_h	C_h	$corr$
0	0	5525	+0x0.0000000000000p+0
1	235	5520	-0x1.46e9e7603049fp-6
2	612	5491	-0x1.cad996fe25a24p-7
3	1036	5427	+0x1.27ac440de0a8cp-10
4	1360	5355	-0x1.522b2a9e8491dp-10
5	1547	5304	-0x1.d6513b89c7237p-6
6	2044	5133	+0x1.038b12ae4eba1p-8
7	2340	5005	-0x1.53f734851f48bp-13
8	2600	4875	-0x1.49140da6fe454p-7
9	2880	4715	-0x1.d02973d03a1f6p-7
10	3315	4420	+0x1.2f1f464d3dc25p-6
11	3500	4275	-0x1.7caa112f287aep-10
12	3720	4085	-0x1.735972faced77p-7
13	3952	3861	-0x1.fa6ed9240ab1ap-7

TABLE 4
A hyperbolic exact lookup table computed for $p = 5$.

Index	S_h	C_h	$corr$
0	0	10,080	+0x0.0000000000000p+0
1	284	10,084	-0x1.93963974f0cb6p-9
2	651	10,101	+0x1.0b316b3c740d1p-9
3	1064	10,136	+0x1.7c74108520aebp-7
4	1190	10,150	-0x1.d8f891d50d1a1p-8
5	1560	10,200	-0x1.13297ef8b55bbp-9
6	1848	10,248	-0x1.535fdc36d3139p-8
7	2222	10,322	-0x1.fe04ef1053a97p-15
8	2560	10,400	+0x1.5891c9eaef76ap-10
9	2940	10,500	+0x1.a58844d36e49ep-8
10	3237	10,587	+0x1.b77a5031ebc86p-9
11	3456	10,656	-0x1.dcf49bb32dc17p-8

triples and the denominators, grows exponentially with p. Consequently, it was not possible to find k for $p \geq 8$ with our hardware configuration.

Table 3 describes an exact lookup table for the trigonometric functions when $p = 4$, where $k = 5525$ and the absolute value of the corrective term is at most 0x1.d6513b89c7237p-6, that is, ≈ 0.0287 for input index $i = 5$. Table 4 presents an exact lookup table for the hyperbolic functions when $p = 5$, where $k = 10{,}080$ and the absolute value of the corrective term is at most 0x1.7c74108520aebp-7, that is, ≈ 0.0116 for input index $i = 3$. Figures 2 and 3 give a visual representation of those tables in the Euclidean plane, depicting the stored triples by dots and the limits of each entry interval by dashed lines.

5.2 Heuristic Search

To build tables indexed by a larger number of bits, a more efficient solution must be found. In order to drastically reduce the search space, we have developed two heuristics, one for the trigonometric functions and one for the hyperbolic functions. These heuristics try to characterize the denominator sides (hypotenuses or bigger legs) to either keep or reject PPTs during the generation step.

For this purpose, let us observe the decomposition in prime factors of each k found using the exhaustive search. Such decompositions are given in Tables 5 and 6 for both the trigonometric and the hyperbolic tables. These factorizations show that every k in the table is a composite number

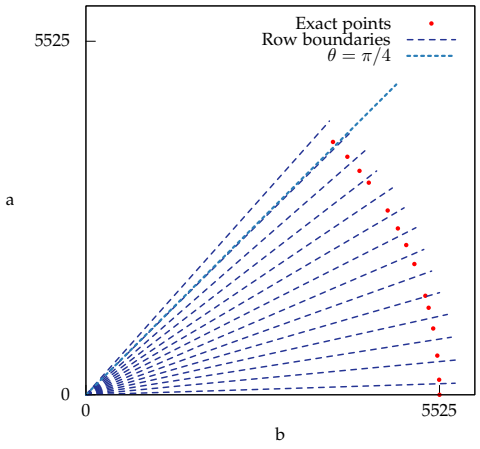

Fig. 2. Visual representation of the trigonometric exact lookup table from Table 3 in the Euclidean plane.

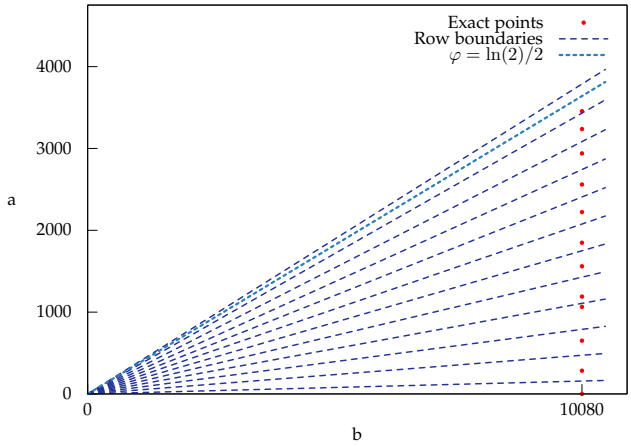

Fig. 3. Visual representation of the hyperbolic exact lookup table from Table 4 in the Euclidean plane.

TABLE 5
Prime Factorization of Found Common Multiples for sin and cos.

k	Prime factorization
425	$5^2 \cdot 17$
5525	$5^2 \cdot 13 \cdot 17$
160,225	$5^2 \cdot 13 \cdot 17 \cdot 29$
1,698,385	$5 \cdot 13 \cdot 17 \cdot 29 \cdot 53$
6,569,225	$5^2 \cdot 13 \cdot 17 \cdot 29 \cdot 41$

TABLE 6
Prime Factorization of Found Common Multiples for sinh and cosh.

k	Prime factorization
144	$2^4 \cdot 3^2$
840	$2^3 \cdot 3 \cdot 5 \cdot 7$
10,080	$2^5 \cdot 3^2 \cdot 5 \cdot 7$
180,180	$2^2 \cdot 3^2 \cdot 5 \cdot 7 \cdot 11 \cdot 13$
1,081,080	$2^3 \cdot 3^3 \cdot 5 \cdot 7 \cdot 11 \cdot 13$

divisible by relatively small primes. Furthermore, for the trigonometric functions, all those small primes are of the form $4n + 1$, as per Theorem 2.

TABLE 7
Heuristic Search Results for $\sin$ and $\cos$.

p	k	n	time (s)	PPTs	Hypotenuses
3	425	9	$\ll 1$	42	8
4	5525	13	$\ll 1$	211	17
5	160,225	18	$\ll 1$	996	40
6	1,698,385	21	0.1	2172	66
7	6,569,225	23	0.4	3453	69
8	314,201,225	29	9.5	10,468	100
9	12,882,250,225	34	294	20,312	109
10	279,827,610,985	39	9393	33,057	110

TABLE 8
Heuristic Search Results for $\sinh$ and $\cosh$.

p	k	n	time (s)	PPTs	Bigger legs
3	144	8	$\ll 1$	23	12
4	840	10	$\ll 1$	65	24
5	10,080	14	$\ll 1$	247	79
6	180,180	18	$\ll 1$	917	193
7	1,081,080	21	0.3	1743	248
8	17,907,120	25	3.2	3909	388
9	147,026,880	28	23	5802	400
10	2,793,510,720	32	350	9012	502

Therefore, each heuristic that we propose follows a simple rule: for the trigonometric functions, only store primitive Pythagorean triples with a hypotenuse of the form:

$$c = \prod_{p \in \mathcal{P}_\pi} p^{r_p} \text{ with } \begin{cases} r_p \in \mathbb{N}^* & \text{if } p = 5 \\ r_p \in \{0,1\} & \text{if } 13 \le p \le 73 \\ r_p = 0 & \text{if } p > 73 \end{cases}, \quad (8)$$

where $\mathcal{P}_\pi$ is the set of Pythagorean primes:

$$\mathcal{P}_\pi = \{5, 13, 17, 29, 37, 41, 53, 61, 73, \dots\}.$$

For the hyperbolic functions, the heuristic only stores primitive Pythagorean triples with a bigger leg of the form:

$$b = \prod_{p \in \mathcal{P}_{\le 23}} p^{r_p} \text{ with } \begin{cases} r_p \in \mathbb{N}^* & \text{if } p < 5 \\ r_p \in \{0,1\} & \text{if } 5 \le p \le 23 \end{cases}, \quad (9)$$

where $\mathcal{P}_{\le 23}$ is the set of primes lower than or equal to 23:

$$\mathcal{P}_{\le 23} = \{2, 3, 5, 7, 11, 13, 17, 19, 23\}.$$

Results, timings, and numbers of considered triples and potential LCMs (hypotenuses or legs) for this heuristic are given in Tables 7 and 8 for the trigonometric and the hyperbolic functions, respectively. As can be seen, this algorithm considers a number of potential LCMs several orders of magnitude lower than the exhaustive search solution. This reduces drastically the memory usage and execution times. For instance, for the trigonometric functions, when $p = 7$, only 3453 triples are stored, compared to the 1,347,954 triples for the exhaustive algorithm. In this first case, the execution time was reduced from 31 seconds to 0.4 seconds. And for the hyperbolic functions, when $p = 7$, only 1743 triples are stored, while there were 147,748 when using the exhaustive algorithm. In this second case, the execution time was reduced from 328 seconds to 0.3 seconds.

With this heuristic, the bottleneck is no longer the memory but the selection of PPTs during their generation. Indeed, this selection is carried out on the elements of an exponentially-growing set. And checking if a given integer satisfies either Equation (8) or Equation (9) requires checking if it is multiple of certain prime numbers, which is a rather expensive test involving integer division.

TABLE 9
Costs of Addition and Multiplication of Expansions.

Operation	Floating-point operations	
	Without FMA	With FMA
$E_2 = E_1 + E_2$	11	
$E_2 = E_2 + E_2$	12	
$E_3 = E_1 + E_3$	16	
$E_3 = E_3 + E_3$	27	
$E_2 = E_1 \times E_2$	20	6
$E_2 = E_2 \times E_2$	26	11
$E_3 = E_1 \times E_3$	47	19
$E_3 = E_3 \times E_3$	107	43

6 Comparisons with Other Methods

We have presented a range reduction for the trigonometric and hyperbolic functions based on exact lookup tables and a method to efficiently build such tables. In order to compare this solution with the other solutions presented in Section 2, we consider a two-phase evaluation scheme for the trigonometric and hyperbolic functions that targets correct rounding for the rounding to nearest in double precision. The quick and the accurate phases target a relative error less than 2^{-66} and 2^{-150}, respectively [9], [39]. We choose $p = 10$ which corresponds to 805 rows in the trigonometric table and 356 rows in the hyperbolic table.

In order to ease comparisons, we consider only the number of memory accesses required by the second range reduction and the number of floating-point operations (FLOPs) involved in the reconstruction step, and table sizes. We will consider that expansion algorithms are used whenever high accuracy is required as it is the case in the correctly rounded library CR-Libm [6]. Let us recall that an expansion of size n consists of an unevaluated sum of n floating-point numbers that represents a given number with a larger precision than what is available in hardware [19]. We will take Table 9 extracted from [6, § 2.3], [40], and [41, § 3.2] as the reference costs for those algorithms. The notation E_n stands for an expansion of size n in double precision, so that, with this notation, E_1 represents a regular binary64/double precision floating-point number.

6.1 Costs for the Trigonometric Functions

6.1.1 Tang's Solution

In order to reach an accuracy of 66 bits, Tang's solution requires access to tabulated values S_h and C_h that are stored as expansions of size 2. These values need to be multiplied by the two results of the polynomial evaluations, which can be considered stored as expansion of size 2 as well. The total cost of the quick phase becomes: 4 double-precision memory accesses, 2 multiplications $E_2 \times E_2$, and 1 addition $E_2 + E_2$, that is, 34 FLOPs (64 without FMA).

TABLE 10
Computation and Memory Access Costs for Three Table-Based Range Reductions for Trigonometric and Hyperbolic Functions. The number of memory accesses and the number of floating-point operations (without FMA in parentheses) are reported.

Solution	Memory Accesses		Floating-point Operations (without FMA)				Bytes per Row
	Quick	Accurate	Trigonometric		Hyperbolic		
			Quick	Accurate	Quick	Accurate	
Tang	4	6	**34** (64)	113 (241)	102 (192)	339 (723)	48
Gal	**3**	7	35 **(63)**	129 (257)	**93 (179)**	355 (739)	56
Proposed	4	**5**	36 (64)	**92 (148)**	94 (180)	**270 (510)**	**40**

In case the quick phase failed to return correct rounding, the accurate phase is launched. This requires access to 2 extra tabulated values to represent S_h and C_h as expansions of size 3. Those values need to be multiplied by the two results of the polynomial evaluations, which can be considered stored as expansion of size 3 as well. The total cost of the accurate phase becomes: 2 extra memory accesses, 2 multiplications $E_3 \times E_3$, and 1 addition $E_3 + E_3$. This corresponds to 6 memory accesses, 113 FLOPs (241 without FMA), and a $38{,}640$ byte table.

6.1.2 Gal's Solution

Using Gal's method, the corrective terms allow around 63 bits of accuracy, and Stehlé and Zimmermann's improvement allows to reach 74 bits. By considering Stehlé's approach, only one double-precision number is required for S_h, C_h and the corrective term, which fits on about 20 bits, to reach an accuracy of 66 bits. Again, S_h and C_h need to be multiplied by the two results of the polynomial evaluations, which can be considered stored as expansions of size 2. Thus the quick phase requires $2 + 1$ double-precision memory accesses, 1 addition $E_2 = E_1 + E_2$ for the corrective term, 2 multiplications $E_1 \times E_2$, and 1 addition $E_2 + E_2$. The cost of the quick phase with this table becomes 3 memory accesses and 35 FLOPs (63 without FMA).

To reach the 150-bit accuracy required by the accurate phase, it is necessary to get 2 extra floating-point numbers for S_h and C_h, which are not exact. The corrective term is then incorporated in the final result using an addition $E_3 = E_1 + E_3$. The remaining operations need to be done using size-3 expansions. The total cost for the accurate phase becomes: 4 extra memory accesses, 2 multiplications $E_3 \times E_3$, 1 addition $E_1 + E_3$, and 1 addition $E_3 + E_3$, that is, 7 memory accesses, 129 FLOPs (257 without FMA), and a $45{,}080$-byte table.

6.1.3 Exact Lookup Tables Solution

With our solution, as shown in Table 7, at most 39 bits are required to store S_h and C_h, that is, only one floating-point number per entry. Our corrective terms are inexact, so that the cost of the quick phase is the same as Gal's approach in Section 6.1.2, except that the addition for the corrective term is an addition $E_2 = E_2 + E_2$, which accounts for 1 extra FLOP and 1 extra memory access. Hence, the quick phase for the exact lookup table solution is 36 FLOPs (64 without FMA), which is as much as Tang's solution.

However, for the accurate phase, values S_h and C_h that were accessed during the quick phase are exact, and do not require any extra memory access. The corrective term is stored as an expansion of size 3 and it requires 1 extra memory access to reach the 150 bits of accuracy. The addition for the corrective term is performed using an addition with expansions of size 3. Multiplications correspond to $E_3 = E_1 \times E_3$ as the results of the polynomial evaluations can be considered as expansions of size 3. The final addition is done using an $E_3 = E_3 + E_3$ operation. That is, the total cost of this step becomes 5 memory accesses and 92 FLOPs (148 without FMA), for a $32{,}200$ byte table.

6.1.4 Comparison Results

Table 10 synthesizes the estimated costs for those three range reduction algorithms based on tabulated values. This table reports the number of memory accesses and FLOPs for the quick and accurate phases, together with the size in bytes of each row of the precomputed table.

First, it can be seen that the proposed table-based range reduction requires less memory per table row than the other solutions. Tang's method requires 48 bytes per row, Gal's method 56 bytes and our exact lookup table 40 bytes. This is an improvement in memory usage of $\approx 17\%$ and $\approx 29\%$ over Tang's and Gal's methods, respectively.

Second, regarding the number of operations, our solution requires two more FLOPs than Tang's solution for the quick phase (none without FMA), and one more FLOP than Gal's solution. This represents an overhead of $\approx 6\%$ (0% without FMA) and $\approx 2\%$, respectively. This small downside comes with two advantages: First, as two out of the four terms in the reconstruction are exact, error bounds will be easier to determine. Second, the accurate phase is much faster. Indeed, for this phase, there is an improvement in favor of our approach of $\approx 19\%$ (39% without FMA) and $\approx 29\%$ (42% without FMA) over Tang and Gal, respectively.

Third, the solution we propose reduces the number of memory accesses during the accurate phase. For this phase, the number of accesses is reduced from 7 to 5 compared to Gal's approach, This is an improvement of $\approx 29\%$, which translates to $\approx 17\%$ compared to Tang's solution.

6.2 Costs for the Hyperbolic Functions

The costs for memory accesses and table sizes for the hyperbolic functions are the same as the ones for the trigonometric functions. Indeed, although the exact lookup tables are smaller for the hyperbolic functions because $\ln(2)/2 \approx 0.347$ is lower than $\pi/4 \approx 0.785$, they both

contain as many words per row. We recall the reconstruction step for the hyperbolic functions:

$$\begin{aligned}\sinh(y) = &\left(2^{n-1} - 2^{-n-1}\right) \cdot \\ &\left(C_h \cdot P_C\left(x_\ell\right) + S_h \cdot P_S\left(x_\ell\right)\right) \\ &\pm \left(2^{n-1} + 2^{-n-1}\right) \cdot \\ &\left(S_h \cdot P_C\left(x_\ell\right) + C_h \cdot P_S\left(x_\ell\right)\right).\end{aligned} \tag{10}$$

Note that if we do not count the computations of the terms $\left(2^{n-1} \pm 2^{-n-1}\right)$, there are 6 multiplications, and 3 additions. Indeed these terms can be neglected in the cost computations as they are easy to generate using shifts and adds in integer arithmetics.

6.2.1 Tang's and Gal's Solutions

On the first hand, let us consider Tang's solution: Overall, the table requires 17,088 bytes. For the quick phase, we must use E_2 expansions for 6 multiplications and 3 additions, as we have seen in Equation (10). These extended-precisions operations account for 102 FLOPs (192 without FMA). During the accurate phase, all computations are done using E_3 expansions, which require a total of 339 FLOPs (723 without FMA).

On the other hand, using Gal's method, the table takes 19,936 bytes and one addition $E_2 = E_1 + E_2$ for $x_\ell - corr$ is added to the cost computations. The quick phase accounts now for 4 multiplications $E_1 \times E_2$ between tabulated values and polynomial results, 2 multiplications $E_2 \times E_2$ for the remaining terms, 1 addition $E_2 = E_1 + E_2$, and 3 additions $E_2 = E_2 + E_2$. Hence, the cost of the quick phase with this table is 93 FLOPs (179 without FMA). For the accurate phase, all operations need to be done using size-3 expansions, except for the addition of the corrective term $E_3 = E_1 + E_3$. The total cost for the accurate phase becomes: 355 floating-point operations (739 without FMA).

6.2.2 Exact Lookup Tables Solution

Again, as for the trigonometric functions, the cost of the quick phase is the same as Gal's approach plus one extra memory access and one extra FLOP for the addition of the inexact corrective term, which is 94 FLOPs (180 without FMA). However, the table is smaller as it takes only 14,240 bytes.

For the accurate phase, as it was the case for the trigonometric functions, values S_h and C_h are exact and do not require any extra memory access. Only the corrective term is stored as an expansion of size 3 and requires 1 extra memory accesses in order to reach an accuracy of 150 bits. Hence, the accurate phase corresponds to 4 multiplications $E_1 \times E_3$, 2 multiplications $E_3 \times E_3$, and 4 additions $E_3 + E_3$. The total cost of this step is thus 270 floating-point operations (510 without FMA).

6.2.3 Comparison Results

As in Section 6.1.4, we compare the three methods using synthesized results from Table 10.

Regarding memory usage, we have quantitatively the same benefits as for the trigonometric functions.

Regarding the number of floating-point operations, our solution has the same small overhead of one FLOP for the quick phase compared to Gal's solution, but it requires 8% (6% without FMA) less FLOPs than Tang's solution. For the accurate phase, there is an improvement in favor of our approach of $\approx$ 20% (29% without FMA) and $\approx$ 24% (31% without FMA) over Tang and Gal, respectively. This is mainly due to the huge cost of multiplications $E_3 \times E_3$ that our method avoids in 4 out of 6 multiplications.

TABLE 11
Tang's table for $p = 4$ with 16-bit precision

Index	S_h	C_h
0	0x0.0000p+0	0x1.0000p+0
1	0x1.ffaap-5	0x1.ff00p-1
2	0x1.feaap-4	0x1.fc02p-1
3	0x1.7dc2p-3	0x1.f706p-1
4	0x1.faaep-3	0x1.f016p-1
5	0x1.3ad2p-2	0x1.e734p-1
6	0x1.7710p-2	0x1.dc6cp-1
7	0x1.b1d8p-2	0x1.cfc6p-1
8	0x1.eaeep-2	0x1.c152p-1
9	0x1.110ep-1	0x1.b11ep-1
10	0x1.2b92p-1	0x1.9f36p-1
11	0x1.44ecp-1	0x1.8bb2p-1
12	0x1.5d00p-1	0x1.76a0p-1
13	0x1.73b8p-1	0x1.6018p-1

7 Example on the Trigonometric Sine

We propose to illustrate the use of the proposed error-free tabulated values compared to Tang's solution, which is probably the most pervasive, using Table 3 and Table 11. We consider the evaluation of the sine function for the input value $y = 10$ when targeting 8 bits of accuracy. When necessary, storage precision and intermediate computations will be done on 16 bits of precision in order to emulate 8-bit expansions of size 2, like what is commonly used for the quick phase. For clarity, bit fields will be represented in radix 2 (suffix "$_2$").

First, the reduced argument x is computed on 16 bits:

$$\begin{aligned}q &= \lfloor 10 \cdot 2/\pi \rceil = 6 \\ x &= 10 - q \cdot \pi/2 \\ &= 1.00100110100001_2 \cdot 2^{-1} \cdot (1 + \varepsilon_{-16}) \\ &\approx 0.575225830078125.\end{aligned}$$

As explained earlier, x is essentially inexact and has to be rounded (it is rounded to 16 bits in this case). By construction, x belongs to the interval $[-\pi/4, \pi/4]$, but in practice, properties of symmetry are used so that it finally lies in $[0, \pi/4]$. We have $q = 6$, which entails $q \equiv 2 \mod 4$. This means that an additional rotation by π radians was made during the first range reduction, so that the sign changed and we now have $\sin(10) = -\sin(x)$.

Now let us compute x_h and x_ℓ, such that $x = x_h + x_\ell$ with $x_h = i \cdot 2^{-4}, i \in [0, 13]$ and $|x_\ell| \leq 2^{-5}$. This leads to

$$\begin{aligned}x_h &= 9 \cdot 2^{-4} \quad \text{and} \\ x_\ell &= x - x_h \\ &= 1.10100001_2 \cdot 2^{-7} \cdot (1 + \varepsilon_{-16}).\end{aligned}$$

For row index $i = 9$, Table 11 gives us

$$\begin{aligned}S_h &= 1.000100010000111_2 \cdot 2^{-1} \cdot (1 + \varepsilon_{-16}) \\ C_h &= 1.101100010001111_2 \cdot 2^{-1} \cdot (1 + \varepsilon_{-16}),\end{aligned}$$

while Table 3 gives us

$$\begin{aligned} S_h &= 2880 \\ C_h &= 4715 \\ corr &= -1.110100000010101_2 \cdot 2^{-7} \cdot (1+\varepsilon_{-16}) \end{aligned}$$

with $k = 5525$ as mentioned in Section 5.1.2.

Now is when the different table-based methods diverge. They will be analyzed separately.

Tang's Table Lookup Method. Using Tang's table, the evaluation of two polynomial approximations to $\sin(x)$ and $\cos(x)$ is performed with an output precision of 12 bits at input $x_\ell = 1.10100001_2 \cdot 2^{-7}$:

$$\begin{aligned} \sin(x_\ell) &\approx 1.10100001_2 \cdot 2^{-7} \cdot (1+\varepsilon_{-12}) \\ \cos(x_\ell) &\approx 1.0_2 \cdot (1+\varepsilon_{-12}). \end{aligned} \tag{11}$$

The final result is then computed on 8 bits using tabulated values S_h and C_h with the results from Equation (11):

$$\begin{aligned} \sin(10) &= -\sin(x) \\ &\approx -S_h \cdot 1.0_2 - C_h \cdot 1.10100001_2 \cdot 2^{-7} \\ &\approx -1.0001011_2 \cdot 2^{-1} \cdot (1+\varepsilon_{-8}) \\ &\approx -0.54296875. \end{aligned}$$

Exact Table Lookup Method. With the proposed solution, the evaluation of polynomial approximations to $\sin(x)/k$ and $\cos(x)/k$ are performed with an output precision of 12 bits at input $x_\ell - corr \approx 1.10111000100101_2 \cdot 2^{-6}$:

$$\begin{aligned} \sin(x_\ell - corr)/k &\approx 1.01000110101_2 \cdot 2^{-18} \\ \cos(x_\ell - corr)/k &\approx 1.011110111_2 \cdot 2^{-13}. \end{aligned} \tag{12}$$

Finally, the result is reconstructed as follows, using the error-free tabulated values S_h and C_h from Table 3 and the approximations from Equation (12):

$$\begin{aligned} \sin(10) &= -\sin(x) \\ &= -S_h \cdot \cos(x_\ell - corr)/k - C_h \cdot \sin(x_\ell - corr)/k \\ &\approx -1.0001011_2 \cdot 2^{-1} \\ &\approx -0.54296875. \end{aligned}$$

Comparison Between Both Methods. One can see that for the exact lookup table method, the error is quickly concentrated into the reduced argument x_ℓ, then into $x_\ell - corr$ and finally into the polynomial approximations. The main advantage of this method relies on the fact that S_h and C_h embed no rounding error at all, which allows for an easier, faster, and more elegant reconstruction step.

8 Conclusions and Perspectives

In this article, we have presented a new approach to address table-based range reductions in the evaluation process of elementary functions. We have shown that this method allows for simpler and faster evaluation of these functions. For the trigonometric and hyperbolic functions, we have made use of Pythagorean triples to build exact lookup tables, which concentrate the error into the polynomial evaluations. Compared to other solutions, exact lookup tables eliminate the rounding error on certain tabulated values, and transfer this error to the remaining reduced argument. We have focused on those functions, as they both benefit from the use of Pythagorean triples. However, the concept remains valid for other functions, provided that exact values can be found for the tables. Compared to the state of the art, we have shown that it was possible to reduce the table sizes, the number of memory accesses and the number of floating-point operations involved in the reconstruction process by up to 29%, 29% and 42%, respectively, when targeting correct rounding in double precision. This benefit would be even higher for larger targeted accuracies since the tabulated values are exactly stored and do not depend on the targeted accuracy.

As future work, it would be interesting to further characterize potential small LCMs in order to speed up the table precomputation process. The objective is to compute the interesting triples directly, possibly using one of Fässler's algorithms [42], instead of generating huge sets of triples and then selecting the relevant ones. Finally, in the proposed algorithms we focused on looking for the smallest possible LCM so that tabulated values would be stored on the minimal number of bits. This property is essential for hardware implementations where each bit is important, but not for software implementations. In that case, it could be interesting to look for tabulated values that fit in a given format (e.g. 53 bits for double-precision floating-point numbers) but that would also increase the precision or reduce the magnitude of the corrective terms. This could save some extra memory accesses and associated floating-point operations.

Acknowledgments

This work was supported by the MetaLibm[2] ANR project (*Ingénierie Numérique et Sécurité 2013*, ANR-13-INSE-0007). We thank the reviewers for their constructive comments on a previous version of this article.

References

[1] *IEEE Standard for Floating-Point Arithmetic*, 2008.
[2] V. Lefèvre, J.-M. Muller, and A. Tisserand, "Toward correctly rounded transcendentals," *IEEE Transactions on Computers*, vol. 47, no. 11, pp. 1235–1243, 1998.
[3] D. Das Sarma and D. W. Matula, "Faithful bipartite ROM reciprocal tables," in *12th IEEE Symposium on Computer Arithmetic*. IEEE Computer Society, 1995, pp. 17–28.
[4] F. de Dinechin and A. Tisserand, "Multipartite table methods," *IEEE Transactions on Computers*, vol. 54, no. 3, pp. 319–330, 2005.
[5] D. Defour, F. de Dinechin, and J.-M. Muller, "A new scheme for table-based evaluation of functions," in *36th Asilomar Conference on Signals, Systems, and Computers*, 2002, pp. 1608–1613.
[6] "CR-Libm, a library of correctly rounded elementary functions in double-precision," http://lipforge.ens-lyon.fr/www/crlibm/.
[7] A. Ziv, "Fast evaluation of elementary mathematical functions with correctly rounded last bit," *ACM Transactions on Mathematical Software*, vol. 17, pp. 410–423, 1991.
[8] J.-M. Muller, *Elementary Functions: Algorithms and Implementation*, 2nd ed. Birkhäuser, 2006.
[9] V. Lefèvre, "Hardest-to-round cases," ENS Lyon, Tech. Rep., 2010.
[10] S. Chevillard, J. Harrison, M. Joldeş, and C. Lauter, "Efficient and accurate computation of upper bounds of approximation errors," *Theoretical Computer Science*, vol. 412, no. 16, pp. 1523–1543, 2011.
[11] M. H. Payne and R. N. Hanek, "Radian reduction for trigonometric functions," *SIGNUM Newsletter*, vol. 18, pp. 19–24, 1983.

2. See www.metalibm.org/ANRMetaLibm/.

[12] M. Daumas, C. Mazenc, X. Merrheim, and J.-M. Muller, *Modular Range Reduction: a New Algorithm for Fast and Accurate Computation of the Elementary Functions*. Springer Berlin Heidelberg, 1995, pp. 162–175.
[13] N. Brisebarre, D. Defour, P. Kornerup, J.-M. Muller, and N. Revol, "A new range-reduction algorithm," *IEEE Transactions on Computers*, vol. 54, 2005.
[14] F. de Dinechin and C. Q. Lauter, "Optimizing polynomials for floating-point implementation," in *Proceedings of the 8th Conference on Real Numbers and Computers*, 2008, pp. 7–16.
[15] N. Brisebarre and S. Chevillard, "Efficient polynomial L^∞-approximations," in *18th IEEE Symposium on Computer Arithmetic*, 2007, pp. 169–176.
[16] C. Mouilleron and G. Revy, "Automatic generation of fast and certified code for polynomial evaluation," in *20th IEEE Symposium on Computer Arithmetic*, 2011, pp. 233–242.
[17] H. de Lassus Saint-Geniès, D. Defour, and G. Revy, "Range reduction based on pythagorean triples for trigonometric function evaluation," in *IEEE 26th International Conference on Application-specific Systems, Architectures and Processors*, 2015, pp. 74–81.
[18] P. T. P. Tang, "Table-lookup algorithms for elementary functions and their error analysis," in *10th IEEE Symposium on Computer Arithmetic*, P. Kornerup and D. W. Matula, Eds. IEEE Computer Society Press, 1991, pp. 232–236.
[19] J. R. Shewchuk, "Adaptive precision floating-point arithmetic and fast robust geometric predicates," *Discrete & Computational Geometry*, vol. 18, pp. 305–363, 1997.
[20] D. Defour, "Cache-optimised methods for the evaluation of elementary functions," Laboratoire de l'informatique du parallélisme, Research report, 2002.
[21] V. Lefevre and J. M. Muller, "Worst cases for correct rounding of the elementary functions in double precision," in *15th IEEE Symposium on Computer Arithmetic*, 2001, pp. 111–118.
[22] ——, "Worst cases for correct rounding of the elementary functions in double precision," 2003, http://perso.ens-lyon.fr/jean-michel.muller/TMDworstcases.pdf.
[23] S. Gal, "Computing elementary functions: A new approach for achieving high accuracy and good performance," in *Accurate Scientific Computations*, ser. Lecture Notes in Computer Science, W. L. Miranker and R. A. Toupin, Eds. Springer Berlin Heidelberg, 1986, vol. 235, pp. 1–16.
[24] S. Gal and B. Bachelis, "An accurate elementary mathematical library for the IEEE floating point standard," *ACM Transactions on Mathematical Software*, vol. 17, pp. 26–45, 1991.
[25] D. Stehlé and P. Zimmermann, "Gal's accurate tables method revisited," in *17th IEEE Symposium on Computer Arithmetic*. IEEE Computer Society, 2005, pp. 257–264.
[26] N. Brisebarre, M. D. Ercegovac, and J.-M. Muller, "(M, p, k)-friendly points: A table-based method for trigonometric function evaluation," in *ASAP*. IEEE Computer Society, 2012, pp. 46–52.
[27] D. Wang, J.-M. Muller, N. Brisebarre, and M. D. Ercegovac, "(M, p, k)-friendly points: A table-based method to evaluate trigonometric functions," *IEEE Transactions on Circuits and Systems*, vol. 61-II, no. 9, pp. 711–715, 2014.
[28] G. W. Reitwiesner, "Binary arithmetic," in *Advances in Computers*, A. D. Booth and R. E. Meager, Eds. Academic Press, 1960, vol. 1, pp. 231–308.
[29] W. Sierpiński, *Pythagorean triangles*. Graduate School of Science, Yeshiva University, 1962.
[30] S. Chevillard, M. Joldeş, and C. Lauter, "Sollya: An environment for the development of numerical codes," in *Mathematical Software – ICMS 2010*, ser. Lecture Notes in Computer Science, K. Fukuda, J. van der Hoeven, M. Joswig, and N. Takayama, Eds., vol. 6327. Heidelberg, Germany: Springer, Sep. 2010, pp. 28–31.
[31] J. Conway and R. Guy, *The Book of Numbers*, ser. Copernicus Series. Springer New York, 1998.
[32] P. Shiu, "The shapes and sizes of pythagorean triangles," *The Mathematical Gazette*, vol. 67, no. 439, pp. 33–38, 1983.
[33] F. J. M. Barning, "On pythagorean and quasi-pythagorean triangles and a generation process with the help of unimodular matrices." *(Dutch) Math. Centrum Amsterdam Afd. Zuivere Wisk. ZW-001*, 1963.
[34] A. Hall, "232. genealogy of pythagorean triads," *The Mathematical Gazette*, vol. 54, no. 390, pp. 377–379, 1970.
[35] H. L. Price, "The Pythagorean Tree: A New Species," *ArXiv e-prints*, Sep. 2008.
[36] W. Sierpiński, *Elementary Theory of Numbers: Second English Edition (edited by A. Schinzel)*. Elsevier, 1988, vol. 31.
[37] A. V. Dominic Vella, "When is n a member of a pythagorean triple?" *The Mathematical Gazette*, vol. 87, no. 508, pp. 102–105, 2003.
[38] A. H. Beiler, *Recreations in the theory of numbers: The queen of mathematics entertains*. Dover Publications, 1964.
[39] J.-M. Muller, N. Brisebarre, F. de Dinechin, C.-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, and S. Torres, *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston, 2010.
[40] C. Q. Lauter, "Basic building blocks for a triple-double intermediate format," Inria, Research Report, 2005.
[41] ——, "Arrondi correct de fonctions mathématiques - fonctions univariées et bivariées, certification et automatisation," Ph.D. dissertation, École Normale Supérieure de Lyon, October 2008.
[42] A. Fässler, "Multiple Pythagorean number triples," *American Mathematical Monthly*, vol. 98, no. 6, pp. 505–517, 1991.

Hugues de Lassus Saint-Geniès was born in Paris, France, in 1990. He received the *Diplôme d'ingénieur* (Master's degree) in Physics, Electronics and Materials from the Grenoble Institute of Technology in 2014, where he majored in Embedded systems and software. He is now a Ph.D. candidate in Computer science at the University of Perpignan Via Domitia. His research interests include floating-point and fixed-point arithmetics, elementary mathematical function approximation, and high performance code generation.

David Defour is an Associate Professor at the University of Perpignan, Perpignan, France, since 2004. He received his Ph.D. degree in computer science from ENS, Lyon, France, in 2003. He is currently leading the evaluation committee of the High Performance Computing Center (HPC@LR). Defour is primarily interested in floating-point arithmetic, numerical algorithms, computer architecture, and microarchitecture for high performance computing and specifically GPUs.

Guillaume Revy received the MS degree in Computer Science from the École Normale Supérieure de Lyon in 2006 and the PhD degree in Computer Science from the Université de Lyon - École Normale Supérieure de Lyon in 2009. After being a postdoctoral fellow in the ParLab (Parallel Computing Laboratory) at the University of California at Berkeley, he is now Associate Professor at the Université de Perpignan Via Domitia (Perpignan, France) and member of the DALI project-team, joint project-team of Université de Perpignan Via Domitia and LIRMM laboratory (UM, CNRS: UMR 5506). His research interests include computer arithmetic, automated code synthesis and certification, and automatic debugging of floating-point applications.