Range Reduction Based on Pythagorean Triples for Trigonometric Function Evaluation

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

Range Reduction Based on Pythagorean Triples for Trigonometric Function Evaluation

Hugues de Lassus Saint-Geniès* David Defour† Guillaume Revy‡
Univ. Perpignan Via Domitia, DALI, F-66860, Perpignan, France
Univ. Montpellier II, LIRMM, UMR 5506, F-34095, Montpellier, France
CNRS, LIRMM, UMR 5506, F-34095, Montpellier, France
* hugues.de-lassus@univ-perp.fr † david.defour@univ-perp.fr ‡ guillaume.revy@univ-perp.fr

***Abstract*—Software evaluation of elementary functions usually requires three steps: a range reduction, a polynomial evaluation, and a reconstruction step. These evaluation schemes are designed to give the best performance for a given accuracy, which requires a fine control of errors. One of the main issues is to minimize the number of sources of error and/or their influence on the final result. The work presented in this article addresses this problem as it removes one source of error for the evaluation of trigonometric functions. We propose a method that eliminates rounding errors from tabulated values used in the second range reduction for the sine and cosine evaluation. When targeting correct rounding, we show that such tables are smaller, and make the reconstruction step cheaper than existing method. This approach relies on Pythagorean triples generators. Finally, we show how to generate tables indexed by up to 10 bits in a reasonable time and with little memory consumption.**

I. Introduction

The representation formats (binary16, binary32, and binary64) and the behavior of floating-point arithmetics available in computers are defined by the IEEE 754-2008 standard [1]. For the five basic operations ($+$, $-$, $\times$, $/$, and $\sqrt{}$), this standard requires the system to return the rounding of the exact result, according to one of the four rounding modes (to nearest, toward $-\infty$, toward $+\infty$, and toward 0). This property is called *correct rounding*, and it warranties the quality of the result. However, due to the Table Maker's Dilemma [2] and the difficulties to develop accurate and efficient evaluation schemes, correct rounding is only recommended for elementary functions in the IEEE 754-2008 standard.

The algorithms used for the evaluation of elementary functions, like sine, cosine, logarithm, and exponential, 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 accuracy, less than 30 bits [3], [4], [5]. The second category concerns algorithms that usually target single or double precision (24 or 53 bits) with an implementation on general processors that rely on the available hardware units [6], [7].

Implementations of those functions that target correct rounding [8] are usually divided into two or more phases. A *quick phase* based on a fast approximation which ensures a few extra bits than the targeted format. With this phase, rounding is possible most of the time at a reasonable cost. When rounding is not possible, a much slower but more *accurate phase* is used. The quick phase uses operations with a precision similar to the input and output precision, while the accurate phase is based on extended precision. For example, in the correctly rounded library CR-Libm, the quick phase for the sine and cosine function in double precision targets 66 bits while the accurate phase corresponds to 200 bits [6, § 7]. The accuracy used for the accurate phase is linked with the search for worst cases for the Table Makers Dilemma [9]. This corresponds to 142 bits for trigonometric functions in rounding to nearest in double precision over the range $[2^{-25}, (1 + 25317/2^{16})2^9]$.

Given a floating-point number x, the internal design of each phase is based on the following 4-step process:

1) A *first range reduction*, based on mathematical properties, narrows the domain of the function to a smaller one. For the sine function, this consists in using the following property:
$$\sin(x + k \cdot \pi/2) = f_k(x^*)$$
with
$$f_k(x^*) = \pm\sin(x^*) \quad \text{or} \quad f_k(x^*) = \pm\cos(x^*)$$
depending on $k \mod 4$. This step results in a reduced argument x^* in the range $[0, \frac{\pi}{4}]$.
2) A *second range reduction*, based on tabulated values, reduces further the range $[0, \frac{\pi}{4}]$. The argument x^* is split into two parts, x_h^* and x_l^*, such that:
$$x^* = x_h^* + x_l^* \quad \text{with} \quad |x_l^*| < 2^{-p}. \tag{1}$$
The term x_h^* corresponds to the p leading bits of x^*, which are used to address a table of $\lceil \pi/4 \times 2^p \rceil$ entries with precomputed values of $\sin_h = \sin(x_h^*)$ and $\cos_h = \cos(x_h^*)$. In the case of sine, we have
$$\sin(x^*) = \sin_h \cdot \cos(x_l^*) + \cos_h \cdot \sin(x_l^*).$$
3) Then, a *polynomial evaluation* is used to approximate the remaining terms over the restricted range. In the case of sine function, this corresponds to the evaluation of two polynomials of small degree that approximate $\sin(x_l^*)$ and $\cos(x_l^*)$ over the range $[0, 2^{-p}]$.
4) Finally, all terms computed during steps 2 and 3 are merged together during the *reconstruction step*.

Satisfactory solutions exist to address the first range reduction, the generation and evaluation of precise and efficient polynomial evaluation schemes, 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 and more specifically for the sine and cosine functions. The proposed method relies on precomputed sine and cosine values with remarkable properties that simplify and accelerate the evaluation of these functions. These properties are threefold: (1) First, each value has to represent exactly both the sine and cosine of a given number, that is, without any rounding error. For this purpose, we use points that are rational numbers. (2) Second, each numerator of rational values should be exactly representable in a representation format available in hardware, that is, as an integer or a floating-point number. (3) Third, each of them has to share the same denominator. These three properties put all together lead to tabulated values for sine and cosine that are exact and exactly representable with rational values sharing a common denominator. Rational numbers that fulfill these three properties are linked to Pythagorean triples.

This article is organized as follows: Section II gives some background on the second range reduction step for the sine and cosine functions. Section III details properties of the proposed tables and shows how they remove two sources of error involved in this step. Section IV presents our approach to build tables of exact points for sine and cosine using Pythagorean triples. Then, Section V presents some experimental results which show that we can compute tables up to 10 bits of index in a reasonable execution time and memory consumption. Finally, some comparison results with other classical approaches are given in Section VI, before concluding in Section VII.

II. Preliminaries

The second range reduction is usually implemented using table lookup methods. This section presents three solutions that address this step. Note that in the sequel of this article, without loss of generality, we will consider the sine function evaluation, while cosine evaluation can be easily derived.

Here and hereafter we denote by $\widehat{x}$ the rounded value of x to a given precision and according to a given rounding mode. To characterize rounding error we use ϵ_{-p} that corresponds to a number y such that $|y| \leq 2^{-p}$. For example, in the case of rounding to the nearest in double precision, we have

$$\widehat{x} \leq x(1+\epsilon_{-54}).$$

A. Tang's Tables

Tang proposed a general method used to implement elementary functions that relies on hardware-tabulated values [10]. Given the reduced argument x^* as in (1), Tang's method uses the upper part x_h^* to address a table. It retrieves two values sin_h and cos_h which are good approximations of $\sin(x_h^*)$ and $\cos(x_h^*)$, respectively, rounded to the destination format. Hence, we have:

$$\sin_h = \sin(x_h^*) \cdot (1+\epsilon_{-54n})$$

and

$$\cos_h = \cos(x_h^*) \cdot (1+\epsilon_{-54n}),$$

where n is the number of floating-point numbers used to represent those quantities.

Then, these values are combined with the results of the evaluation of the two polynomials $S(x)$ and $C(x)$, defined as:

$$S(x) = \sin(x) - x \quad \text{and} \quad C(x) = \cos(x) - 1.$$

Finally, the value $\sin(x^*)$ is reconstructed as follows:

$$\begin{aligned}\sin(x^*) &= sin_h \cdot \cos(x_l^*) + cos_h \cdot \sin(x_l^*) \\ &= sin_h \cdot \big(1 + C(x_l^*)\big) + cos_h \cdot \big(x_l^* + S(x_l^*)\big).\end{aligned}$$

In order to reach a given accuracy, this method requires the use of a higher precision than the targeted one. This extra precision usually involves the use of error-free algorithms such as Fast2Sum or exact multiplication [11].

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 two parts x_h^* and x_l^*, the memory units can provide the two tabulated values sin_h and cos_h, while floating-point units (FPUs) evaluate the polynomials $C(x_l^*)$ and $S(x_l^*)$. As the degree of the polynomials decreases when the table size increases, the objective is to find parameters such that the polynomial evaluations take as long as memory accesses, in average [12].

B. Gal's Accurate Tables

In Tang's method, sin_h and cos_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 proposed a method to transfer some of the errors due to rounding over the reduced argument [13]. This consists in introducing a small "corrective" term on the values x_h^*, denoted by $corr$ here and hereafter. For each input entry x_h^* of the table, this term is carefully chosen to ensure that both $\sin(x_h^* + corr)$ and $\cos(x_h^* + corr)$ are very close to a floating-point number. In [14], Gal and Bachelis were able to find $corr$ such that

$$sin_h = \sin(x_h^* + corr) \cdot (1+\epsilon_{-(10+53n)})$$

and

$$cos_h = \cos(x_h^* + corr) \cdot (1+\epsilon_{-(10+53n)}).$$

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 input number x^*. However, this imposes to store the corrective term $corr$ along with the values sin_h and cos_h as the unevaluated sum of n floating-point numbers such that $10 + 53n$ is larger than the targeted accuracy. The value $\sin(x^*)$ is thus reconstructed as follows:

$$\sin(x^*) = sin_h \cdot \cos(x_l^* - corr) + cos_h \cdot \sin(x_l^* - corr).$$

Gal's solution requires an exhaustive search in order to determine a "good" corrective term for each entry x_h^*. The search space grows exponentially with the number of extra bits for sin_h and cos_h. Stehlé and Zimmermann proposed an improvement based on LLL algorithm [15] to speed up the search. They were able to increase the accuracy of Gal's table by 11 extra bits.

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

In 2012, Brisebarre, Ercegovac, and Muller proposed a new method for sine and cosine evaluation with a few table lookups and additions in hardware [16], [17]. Their approach consists in tabulating four values a, b, z, and $\hat{x}$, defined as:

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

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 among specific points with integer coordinates (a, b) called (M, p, k)-friendly points. These points are recoded using *canonical recoding* which minimizes the number of non-zero digits in table entries. More precisely, a and b must be positive integers lower than M, such 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 conditions make hardware multiplications by a, b, and z cheaper. Compared to other hardware evaluation schemes, this solution reduces by 50% the area on FPGAs for 24-bit accuracy.

III. A Table of Exact Points

Tang's tables store tabulated value of equally spaced distinguished points rounded to the destination format. This rounding error is problematic when seeking an accurate evaluation scheme since at least three double precision floating-point numbers are required. We have seen in Section II how Gal, Stehlé, and Brisebarre improved the accuracy of those tabulated values. This consists in finding input numbers whose images by the function are close to machine representable numbers. This artificially increases the accuracy of stored values by a few extra bits, but those values still embed some errors.

The proposed improvement is based on the following fact: After the first range reduction, the reduced number x^* has to be considered as an irrational number, and therefore it has to be rounded to a given precision. We look for almost regularly spaced points whose images by the considered function will be exactly representable. The table will store exact values for $\sin_h$ and $\cos_h$ in a machine friendly format, plus a truncated corrective term to apply to the reduced argument as in Gal's method. This way, the error will be concentrated in the reduced number used in the polynomial evaluation x_l^*.

We will now describe what good points for this method are and how the evaluation scheme can benefit of such points. For this purpose, let x be the input floating-point number and x^* the reduced argument after the first range reduction such that

$$x^* = x - k \cdot \frac{\pi}{2}, \quad \text{with} \quad k \in \mathbb{N}^*.$$

As $\pi/2$ is an irrational number, x^* is irrational as well, and it has to be rounded to some precision j such that

$$\widehat{x^*} = x^* \cdot (1 + \epsilon_{-j}).$$

We should mention that $\widehat{x^*}$ is generally represented as the unevaluated sum of two or more floating-point numbers to reach an accuracy of j bits. As seen in Section I, the second range reduction splits $\widehat{x^*}$ into two parts, x_h^* and x_l^*, such that

$$\widehat{x^*} = x_h^* + x_l^* \quad \text{and} \quad |x_l^*| < 2^{-p}.$$

The first p bits of x_h^* are used to address the table T made of $\lceil \pi/4 \times 2^p \rceil$ entries. This table has to store values such that sin_h and cos_h can be recovered without any rounding error. This means that the following properties hold:

1) The value sin_h and cos_h have to be rational numbers such that
$$sin_h = \frac{s_n}{s_d} \text{ and } cos_h = \frac{c_n}{c_d}, \text{ with } s_n, s_d, c_n, c_d \in \mathbb{N}^*.$$
2) In order to make the reconstruction step easier, the denominators s_d and c_d have to be equal, that is, $s_d = c_d$.
3) In order to avoid unnecessary division during reconstruction, the common denominator has to be equal to the same k for every input number x_h^*. This value k can be seen as the lowest common multiple (LCM) of the denominator of all entries.
4) In order to minimize the table size, stored values s_n and c_n have to be representable as a machine word.

With numbers satisfying those properties, the reconstruction step corresponds to:

$$\sin(x^*) = s_n \cdot P_{cos}(x_l^* - corr) + c_n \cdot P_{sin}(x_l^* - corr),$$

where

- $P_{cos}(x)$ and $P_{sin}(x)$ are polynomial approximations, defined as follows for $x \in [0, 2^{-p}]$:
$$P_{cos} = \frac{\cos(x)}{k} \quad \text{and} \quad P_{sin} = \frac{\sin(x)}{k},$$
- The tabulated values s_n, c_n, and $\widehat{corr}$ are such that
$$x_h^* = \arcsin\left(\frac{s_n}{k}\right) - corr = \arccos\left(\frac{c_n}{k}\right) - corr$$
with $\widehat{corr} = corr \cdot (1 + \epsilon_{-53n})$ corresponding to an approximation of the corrective term stored as the unevaluated sum of n floating-point numbers,
- And s_n and c_n are stored exactly in a machine word (i.e double precision floating-point number).

As we can observe, we impose sin_h and cos_h to be rational numbers. This should requires to store both numerator and denominator, and to perform two divisions in the reconstruction step. However, we avoid the need to store the denominator and perform the associated division by integrating it in the polynomial approximation. This eliminates the cost of the division, and the error coming from this operation.

IV. Building the Table of Exact Points

The proposed table used to perform the second range reduction brings several benefits over existing solutions. However, building such a table of exact points is not trivial and it involves results from *Pythagorean triples*. These objects are described in the first part of this section. Then, we present a method to efficiently build tables of exact points for the sine and cosine functions.

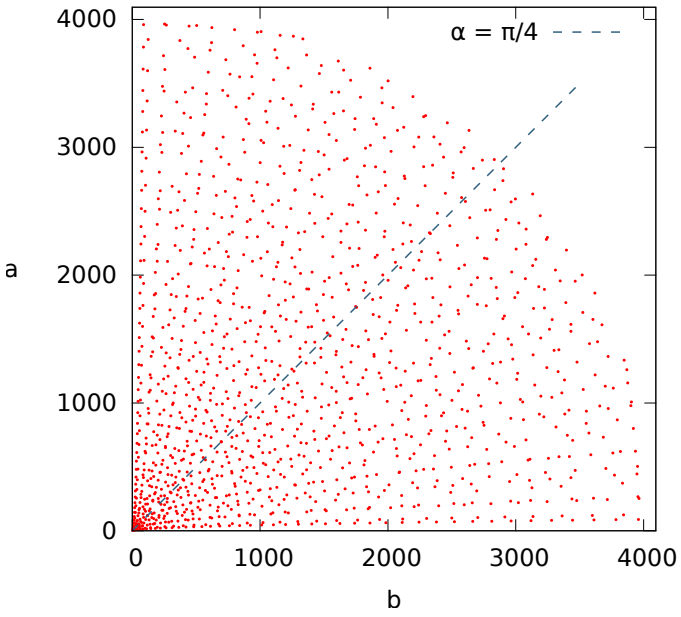


Figure 1. Primitive Pythagorean triples whose hypotenuse c is less than 2^{12}.

A. Pythagorean Triples

Pythagorean triples are mathematical objects that have been widely studied in number theory. By definition a triple (a, b, c) of integers is a *Pythagorean triple* if and only if:

$$a^2 + b^2 = c^2, \quad \text{with} \quad a, b, c \in \mathbb{N}^*.$$

It follows from the Pythagorean theorem that such a triple corresponds to the lengths of a right triangle edges. Sine and cosine of right triangle are defined as quotients of these lengths. Hence, given any Pythagorean triple (a, b, c), there exists an angle $\theta \in]0, \pi/2[$ such that:

$$\sin(\theta) = \frac{a}{c} \quad \text{and} \quad \cos(\theta) = \frac{b}{c}.$$

A Pythagorean triple for which the fractions a/c and b/c are irreducible is called a *primitive Pythagorean triple* (PPT). This notion is essential since a PPT and its multiples refer to *similar* triangles, and thus are associated with the *same* angle θ. For example, the well known PPT $(3, 4, 5)$ and all its multiples are associated to the angle $\theta = \arcsin(3/5) \approx 0.6435$ rad. Recall that we are looking for rational numbers which approximate the sine and cosine for *various* angles between 0 and $\pi/4$. Therefore we will focus only on primitive Pythagorean triples.

B. Construction of Subsets of Primitive Pythagorean Triples

The set of primitive Pythagorean triples is unbounded. Figure 1 shows all PPTs whose hypotenuse c is such that $c \leq 2^{12}$. We can observe that they cover a wide range of angles over $]0, \pi/2[$, and that this set is finite.

In this article, we use the Barning-Hall tree [18] to build sets of PPTs that exploit its ternary-tree structure [19]. From a given PPT (a, b, c), represented as a column vector, three new PPTs are computed by multiplying the former with the following three constants matrices:

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

It is proven that all PPTs can be generated from the root $(3, 4, 5)$ with increasing hypotenuse lengths. For every generated PPT (a, b, c), we also consider its *symmetric* PPT (b, a, c), as it corresponds to a different angle in $]0, \pi/2[$. For instance, the three children of the triple $(3, 4, 5)$ using the Barning-Hall tree are:

$$(5, 12, 13), \quad (15, 8, 17), \quad \text{and} \quad (21, 20, 29),$$

and their symmetric counterparts are

$$(12, 5, 13), \quad (8, 15, 17), \quad \text{and} \quad (20, 21, 29).$$

C. Selection of Primitive Pythagorean Triples

Recall that the table is addressed by x_h^*, the p leading bits of the reduced argument x^*. Then for each entry i of the table, we want to select only one primitive Pythagorean triple such that its corresponding angle θ is as close as possible to x_h^*, where:

$$\theta = \arcsin(a/c) \quad \text{and} \quad x_h^* = i \cdot 2^{-p}.$$

This means that for each table entry, we have to select the triple that minimizes the corrective term $corr$ defined by:

$$corr = x_h^* - \arcsin(a/c).$$

Then, once the terms a, b, c, and $corr$ are computed, a naive solution would consist in storing exactly a, b, c in double precision, and an approximation of $corr$ on as many bits as x_l^* and computed in multiple precision. As presented in Section III, instead of doing this, the solution we propose consists in storing two integers A and B of the form

$$A = \frac{a}{c} \cdot k \quad \text{and} \quad B = \frac{b}{c} \cdot k$$

where $k \in \mathbb{N}^*$ is the same for all table entries. In this article, in order to minimize the error coming from the multiplication, we are looking for table entries such that A and B are the smallest. This consists in looking for a value k that is small or even better the smallest.

Depending on the number of bits p for x_h^*, recall that we have a table with $\lceil \pi/4 \times 2^p \rceil$ entries. This corresponds to 101 entries for $p = 7$. For each considered entry, we need to have at least one PPT in order to find a set of A's and B's. We have reported in Figure 2(a) the number of PPTs per entry for $p = 7$, such that the generated hypotenuses were less than 2^{18}. This corresponds to the minimum number of bits for the hypotenuse so that we have at least one PPT per entry. On this figure, we observe that we have a mean number of 413 PPTs per entry, and a standard deviation of 33. We are looking for the lowest common multiple (LCM) k of one PPT hypotenuse per table entry. This means that we have to compute the LCM for the set made of 101 elements which can take 413 different values. This corresponds to approximately 413^{101} combinations. This number is definitely too large to be tested.

We can observe on Figure 2(a) that the number of PPTs for the table entry 0, that is, near the angle 0, is equal

to 106 whereas the mean is around 413. We have reported in Figure 2(b), the number of PPTs per entry for $p = 7$, such that the generated hypotenuses were less than 2^{14}. This corresponds to the minimum number of bits for the hypotenuse so that we have at least one PPT per entry, but when the first entry from the table is excluded from the search space. On this figure, we observe that we have a mean number of 26 PPTs per entry and a standard deviation of 4.6. Therefore, if the first entry is excluded, the search space is reduced to 26^{100} combinations to test. There are still too many combinations to compute an LCM in a reasonable time. To further reduce the space search, the LCM will not be computed but rather be looked for among generated hypotenuses.

Nevertheless, it is still possible to exclude the first entry by selecting the *degenerated* PPT $(0, 1, 1)$ for this entry. This PPT corresponds to the angle $\theta = 0°$ exactly with a corrective term $corr = 0$. The advantage of selecting this PPT is twofold: (1) First, the search space is reduced. (2) Second, its hypotenuse is equal to 1 which will not impact the search of LCM and the corrective term is exact.

V. Implementation and Numeric Results

In this section we present how we implemented the proposed method to generate tables of exact points that fulfill the properties listed in Section III. We have designed two solutions to look for a small common multiple k. The first solution is based on an *exhaustive* approach 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.

A. Exhaustive Search

1) Algorithm: As seen in Section IV-C, we need to look for a small common multiple k of a combination of one PPT per table entry. We designed a C++ program that takes as input the number p of bits used to index the table and look for the minimal number n of bits used to represent the hypotenuse that corresponds to $k \cdot c$. The search is exhaustive amongst the generated hypotenuses, which warranties that if a common multiple is found, it is the lowest. We initialize n to 4 and we build the program around the following three steps:

1) Generate all PPTs (a, b, c) such that $c \leq 2^n$.
2) Search for the LCM k among all generated hypotenuses c.
3) If such a k is found, build tabulated values $(A, B, corr)$ for every entry. Otherwise, set $n = n + 1$ and go back to step 1.

Values $(A, B, corr)$ are computed using the PPTs (a, b, c) such that k is a multiple of c. In case several PPTs per entry fulfill this property, the one for which $\arcsin(a/c)$ is the closest to x_h^* is selected as this minimizes the error on x_l^*.

2) Numeric Results: Table I shows the results we obtained on an Intel(R) Xeon(R) CPU E5-2650 v2 @ 2.6 GHz (32 cores) with 125 GB of RAM running on GNU/Linux. It describes for the number p of bits that is targeted, the value of k_{min} that was found followed by the number n of bits used to represent k_{min}, the time necessary to find this value, and the numbers of considered PPTs and hypotenuses.

Table I. Exhaustive Search Results.

p	k_{min}	n	time (s)	Triples	Hypotenuses
3	425	9	$\ll 1$	86	66
4	5525	13	$\ll 1$	1404	889
5	160 225	18	0.2	42 328	24 228
6	1 698 385	21	7	335 344	179 632
7	6 569 225	23	31	1 347 953	686 701
8	$> 2^{27}$	> 27	> 6700?	$> 21\,407\,992$	$> 10\,144\,723$

Table II. Values $(A, B, corr)$ Computed for $p = 5$.

Input index	A	B	$corr$
0	0	160225	+0x0.0000000000000p+0
1	5772	160121	+0x1.3966f27bfc9f0p-8
2	11385	159820	+0x1.1a56677e409a8p-7
3	15225	159500	+0x1.73400355ad200p-10
4	19775	159000	-0x1.4b6e4ab5496c0p-10
5	24505	158340	-0x1.62b596fa18d80p-9
6	30044	157383	+0x1.27ac440de0a80p-10
7	33820	156615	-0x1.8df1bd239dc40p-8
8	39440	155295	-0x1.522b2a9e84900p-10
9	44863	153816	+0x1.4d7623b1c6e80p-9
10	50375	152100	+0x1.e0220454e0100p-8
11	56257	150024	+0x1.ebd078b04dc80p-7
12	58305	149240	-0x1.4eccbaa9e7000p-9
13	63504	147103	+0x1.4f7dfcae44600p-10
14	67860	145145	-0x1.53f734851f800p-13
15	73593	142324	+0x1.158024b185460p-7
16	75400	141375	-0x1.49140da6fe460p-7
17	81345	138040	+0x1.48c11cc509200p-10
18	85255	135660	-0x1.75ed3145ed600p-10
19	90596	132153	+0x1.d824b6b5b8300p-8
20	93247	130296	-0x1.f7e24ff929500p-9
21	97760	126945	-0x1.5696f82020000p-17
22	101500	123975	-0x1.7caa112f28800p-10
23	105465	120620	-0x1.29d074350d800p-12
24	110500	116025	+0x1.68ddad9df7000p-7
25	111969	114608	-0x1.eb6d097519180p-8

We observe that it is possible to find k_{min} and to build tables indexed by up to $p = 7$ bits in a reasonable amount of time. However, during our test we noticed that the heap memory was heavily solicited and that it was not possible to go beyond 8 bits.

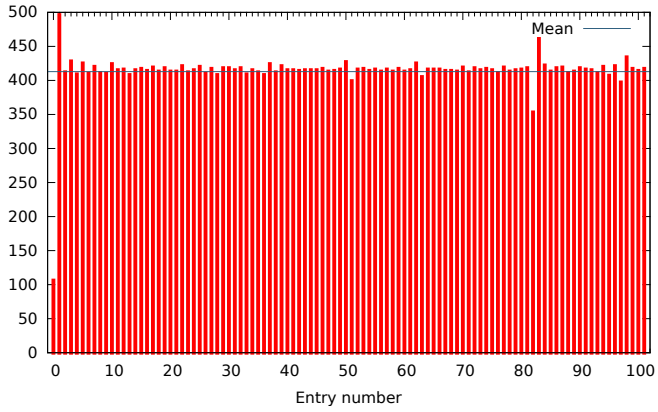
Table II describes the table of exact points for $p = 5$, where $k = 160\,225$ and the absolute value of the corrective term is at most $135188896813938 \times 2^{-53}$, that is, ≈ 0.0150090 for input index $i = 11$.

B. Heuristic Search

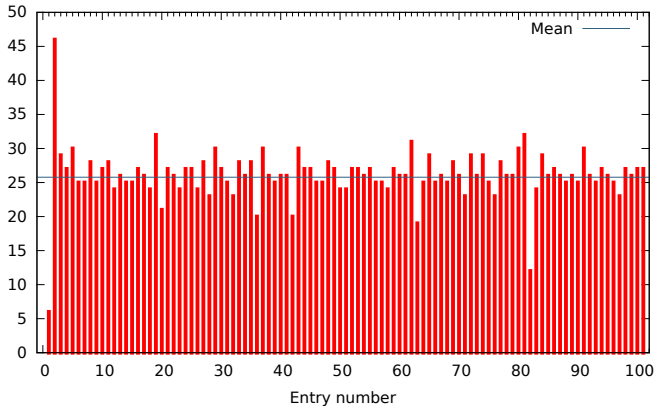
To build tables indexed by a larger number of bits, it is therefore mandatory to use another solution. In order to reduce the search space, we have developed a heuristic that selects "good" hypotenuses and rejects others during the PPT generation phase.

We collected information related to the decomposition in prime factors of each k_{min} found using the exhaustive search. Such a decomposition is given in Table III. These factorizations show that every k in the table is a composite number divisible by relatively *small* primes. Furthermore all those *small* primes are of the form $4n + 1$, better known as *Pythagorean primes* [20].

Therefore, the heuristic we propose follows a simple rule: only store primitive Pythagorean triples whose hypotenuse is



(a) With at least one PPT per entry.



(b) When entry 0 is omitted.

Figure 2. Minimum number of PPTs per entry for $p = 7$.

Table III. PRIME FACTORIZATION OF FOUND COMMON MULTIPLES.

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	$5^2 \cdot 13 \cdot 17 \cdot 29 \cdot 41$

of the form:

$$c = \prod_i p_i^{r_i} \text{ with } p_i \in \mathcal{P} \text{ and } \begin{cases} r_i \in \{0,1\} & \text{if } p_i \neq 5 \\ r_i \in \mathbb{N}^* & \text{else} \end{cases}, \quad (2)$$

where $\mathcal{P}$ is the set of Pythagorean primes lower than or equal to 73:

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

Results, timings, and numbers of considered triples and hypotenuses for this heuristic are given in Table IV. We can observe that this algorithm considers a number of hypotenuses several order of magnitude lower than the exhaustive search solution. This reduces the amount of necessary memory and execution time. For instance, for $p = 7$, only 3308 triples are stored, compared to the 1 347 953 triples for the exhaustive algorithm. In this case, the execution time was reduced from 31 seconds to 0.4 seconds.

Table IV. HEURISTIC SEARCH RESULTS.

p	n	k_{min}	time (s)	Triples	Hypotenuses
3	9	425	$\ll 1$	41	8
4	13	5525	$\ll 1$	210	17
5	18	160 225	$\ll 1$	995	40
6	21	1 698 385	0.1	2171	66
7	23	6 569 225	0.4	3452	69
8	29	314 201 225	9.5	10 467	100
9	34	12 882 250 225	294	20 311	109
10	39	279 827 610 985	9393	33 056	110

With this heuristic, the bottleneck is no longer memory but the selection of PPTs during their generation. Indeed, checking if a given hypotenuse satisfies (2) requires checking if it is multiple of Pythagorean prime numbers, which is an expensive test.

VI. COMPARISONS WITH OTHER METHODS

We have presented a range reduction based on exact points and how to efficiently build those points. In order to compare this solution with the other solutions presented in Section II, we consider a two-phase evaluation scheme of the sine function that targets correct rounding in double precision. The quick and the accurate phases target a relative error less than 2^{-66} and 2^{-150}, respectively. We choose $p = 10$ which corresponds to $\lceil \pi/4 \times 2^{10} \rceil = 805$ entries in each considered table.

In order to ease comparisons, we are only considering the number of memory accesses required by the second range reduction and the number of floating-point operations involved in the reconstruction step. 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]. An expansion of size n consists in storing a given number as the unevaluated sum of n floating-point numbers [21]. We will use Table V extracted from [22] as the reference cost for those algorithms when no fma (fused multiply-add) is available. The notation E_n stands for expansion of size n, such that, with this notation, E_1 represents a regular floating-point number.

Table V. COST OF ADDITION AND MULTIPLICATION OF EXPANSIONS.

Operation	Number of operation
$E2 = E_2 + E_2$	12
$E3 = E_3 + E_3$	27
$E2 = E_1 \times E_2$	20
$E3 = E_1 \times E_3$	47
$E2 = E_1 \times E_1$	17
$E2 = E_2 \times E_2$	26
$E3 = E_3 \times E_3$	107

A. Tang's Solution

In order to reach an accuracy of 66 bits, Tang's solution requires accessing to tabulated values sin_h and cos_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 (MA), 2 multiplications $E_2 \times E_2$,

and 1 addition $E_2 + E_2$, that is, 64 floating-point operations (FLOP).

In case the quick phase failed to return correct rounding, the accurate phase is launched. This requires accessing to 2 extra tabulated values to represent sin_h and cos_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: 241 floating-point operations, and a computed table of 38 640 bytes.

B. Gal's Solution

Using Gal's method, the corrective term allows to achieve around 66 bits of accuracy, and Stehlé's improvement allows to reach 74 bits. In both cases, only one double precision number is required for sin_h, cos_h, and the corrective term. Again, these values need to be multiplied by the two results of the polynomial evaluations that can be considered stored as expansions of size 2. Thus the quick phase requires $2+1$ double precision memory accesses, 1 addition 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 53 floating-point operations.

To reach the 150-bit accuracy required by the accurate phase, it is necessary to get 2 extra floating-point numbers for each tabulated values. The corrective term is then integrated in the final result using addition between expansions of size 3. The rest of the operations need to be done using size-3 expansions. The total cost for the accurate phase becomes 6 extra memory accesses, 2 multiplications $E_3 \times E_3$ and 2 additions $E_3 + E_3$, that is, 9 memory accesses, 268 floating-point operations, for a 57 960-byte computed table.

C. The Exact Points Solution

With our solution, as shown in Table IV, at most 39 bits are required to store A and B, that is, only one floating-point number per entry. Hence, the cost of the quick phase is the same as Gal's approach in Section VI-B.

However, for the accurate phase, values sin_h and cos_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 2 extra floating-point numbers to reach the 150 bits of accuracy. The corrective term is integrated in the final result using an addition with expansion of size 3. Multiplications correspond to $E_3 = E_1 \times E_3$ as the results of the polynomial evaluations are stored as expansions of size 3. The final addition is done using $E_3 = E_3 + E_3$ operation. That is, the total cost of this step becomes 3 memory accesses and 148 floating-point operations, for a computed table of 32 200 bytes.

D. Comparison Results

Table VI synthesizes the estimated costs for those three range reduction algorithms based on tabulated values. This table reports the number of floating-point operations for the quick and accurate phases, together with the size in bytes of the computed table.

Table VI. COMPARISONS BETWEEN THREE TABLE-BASED RANGE REDUCTIONS, FOR $p = 10$. THE NUMBER OF MEMORY ACCESSES (MA) AND THE NUMBER OF FLOATING POINT OPERATIONS (FLOP) ARE REPORTED.

Solution	Quick phase	Accurate phase	Table size (bytes)
Tang	4 MA + 64 FLOP	6 MA + 241 FLOP	38640
Gal	**3 MA + 53 FLOP**	9 MA + 268 FLOP	57960
Proposed	**3 MA + 53 FLOP**	**5 MA + 148 FLOP**	**32200**

First, we notice from this table that the proposed table-based range reduction requires less memory per table entry than others tested solutions. Tang's method needs 48 bytes per entry and Gal's method, 72 bytes per entry, while 40 bytes are enough for the table with exact points. This is an improvement in memory usage of $\approx 17\%$ and $\approx 45\%$ over Tang's and Gal's methods, respectively.

Second, regarding the number of operations for both the quick and accurate phases, we observe that our solution provides similar performance to Gal's solution for the quick phase. For the accurate phase, we observe an improvement in favor of our approach of $\approx 39\%$ and 45% over Tang and Gal, respectively.

Third, we notice that the proposed solution reduces the number of memory accesses. The quick phase requires 3 accesses, while Tang's approach uses 4 accesses, that is, an improvement of 25%. The benefit is even more significant for the accurate phase, as the number of accesses is reduced from 9 to 4 compared to Gal's approach. This is an improvement of $\approx 45\%$.

VII. CONCLUSIONS AND PERSPECTIVES

In this article, we have presented a new approach to address the second range reduction of the evaluation process of elementary functions based on tabulated values. It relies on Pythagorean triples, which allow to simplify and accelerate the evaluation of these functions. Compared to other solutions, the table of *Exact Points* eliminates the rounding error on certain tabulated values, and transfers this error in the remaining reduced argument. We have focused the method on sine and cosine functions as it is the most difficult. However, the concept remains valid for other functions. For those functions and thanks to the proposed method, it is possible to reduce up to 45% the table sizes of, the number of memory accesses and the number of floating-point operations involved in the reconstruction process.

Our further research direction is threefold: First, it would be interesting to plug the computed table into a full sine and cosine implementation to observe its actual impact. This could be done within the CR-Libm project. Second, as we have seen in Section V-B, relevant hypotenuses are factors of small Pythagorean primes. Following this, it would be interesting to characterize "good" hypotenuses, so that, instead of generating a huge set of triples and then choosing the relevant ones, they could be computed directly possibly using Bresenham's algorithm. Doing this way, it would speed up the table computation process. Third, we have focused on looking for the lowest common multiple 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. For software solution, it would be more interesting to look for tabulated values that could fit in a given format (i.e double precision floating-point number) but that would minimize the corrective terms. This may result in corrective terms that could be representable as expansions of size strictly less than n, and thus save some extra memory accesses and associated floating-point operations.

ACKNOWLEDGMENT

This work was supported by the ANR MetaLibm project (*Ingénierie Numérique et Sécurité 2013*, ANR-13-INSE-0007).

REFERENCES

[1] "IEEE Standard for Floating-Point Arithmetic," *IEEE Std 754-2008*, pp. 1–70, Aug 2008.

[2] V. Lefèvre, J.-M. Muller, and A. Tisserand, "The table maker's dilemma," Ecole normale supérieure de Lyon, Lyon, Tech. Rep., 1998.

[3] D. DasSarma and D. W. Matula, "Faithful bipartite ROM reciprocal tables," in *Proc. 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 Trans. Math. Software*, vol. 17, pp. 410–423, 1991.

[8] J.-M. Muller, *Elementary functions - algorithms and implementation (2. ed.)*. Birkhäuser, 2006.

[9] V. Lefevre, "Hardest-to-round cases," ENS Lyon, Tech. Rep., 2010.

[10] P. T. P. Tang, "Table-lookup algorithms for elementary functions and their error analysis," in *Proceedings: 10th IEEE Symposium on Computer Arithmetic: June 26–28, 1991, Grenoble, France*, P. Kornerup and D. W. Matula, Eds. pub-IEEE:adr: IEEE Computer Society Press, 1991, pp. 232–236. [Online]. Available: http://www.acsel-lab.com/arithmetic/arith10/papers/ARITH10_Tang.pdf

[11] D. E. Knuth, *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997.

[12] D. Defour, "Cache-optimised methods for the evaluation of elementary functions," 2002.

[13] S. Gal, "Computing elementary functions: a new approach for achieving high accuracy and good performance," in *Accurate Scientific Computations: Symposium, Bad Neuenahr, FRG, March 12–14, 1985: Proceedings*, ser. Lecture Notes in Computer Science, W. L. Miranker and R. A. Toupin, Eds., vol. 235. pub-SV:adr: Springer-Verlag, 1985, pp. 1–16.

[14] S. Gal and B. Bachelis, "An accurate elementary mathematical library for the IEEE floating point standard," *ACM Trans. Math. Software*, vol. 17, pp. 26–45, 1991.

[15] D. Stehlé and P. Zimmermann, "Gal's accurate tables method revisited," in *IEEE Symposium on Computer Arithmetic*. IEEE Computer Society, 2005, pp. 257–264. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/ARITH.2005.24

[16] 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. [Online]. Available: http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=6341240

[17] D. Wang, J.-M. Muller, N. Brisebarre, and M. D. Ercegovac, "(M, p, k)-friendly points: A table-based method to evaluate trigonometric function," *IEEE Trans. on Circuits and Systems*, vol. 61-II, no. 9, pp. 711–715, 2014. [Online]. Available: http://dx.doi.org/10.1109/TCSII.2014.2331094

[18] 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.

[19] H. L. Price, "The pythagorean tree: A new species," Oct. 24 2008. [Online]. Available: http://arxiv.org/abs/0809.4324

[20] A. Fässler, "Multiple Pythagorean number triples," *American Mathematical Monthly*, vol. 98, no. 6, pp. 505–517, Jun./Jul. 1991.

[21] J. R. Shewchuk, "Adaptive precision floating-point arithmetic and fast robust geometric predicates," *Discrete & Computational Geometry*, vol. 18, pp. 305–363, 1997.

[22] C. Q. Lauter, "Basic building blocks for a triple-double intermediate format," Sep. 2005. [Online]. Available: http://hal.inria.fr/inria-00070314/en/;http://hal.ccsd.cnrs.fr/docs/00/07/03/14/PDF/RR-5702.pdf