Computing hard-to-round cases of sin, cos, tan in double precision

Vincent Lefèvre
Inria, ENS de Lyon, CNRS,
Université Claude Bernard Lyon 1,
LIP, UMR5668, 69342, Lyon cedex 07, France

Tue Ly
Google
Cambridge, MA, US

Paul Zimmermann
Université de Lorraine
CNRS, Inria, LORIA
F-54000 Nancy, France

***Abstract*—This paper describes an exhaustive search algorithm to find the hardest-to-round cases of trigonometric functions (sin, cos, tan) in double precision (binary64). This algorithm reuses a clever reduction from the literature, but instead of using a sublinear search, it uses a brute force linear search. This algorithm was implemented using multi-threading and SIMD, and a full set of hard-to-round cases for the binary64 trigonometric functions was computed. As a consequence, the Table Maker's Dilemma is now fully solved for the most common univariate binary64 functions.**

***Index Terms*—IEEE 754, binary64 format, correct rounding, Table Maker's Dilemma, trigonometric function, floating-point arithmetic.**

I. Introduction

Since its first revision in 2008, the IEEE 754 standard [7] recommends correctly rounded mathematical functions. One of the main difficulties in designing efficient correctly rounded functions is the Table Maker's Dilemma (TMD). Solving the TMD requires to find, for a given function $f(x)$ and a given format, the hardest-to-round inputs. The TMD is easy to solve for univariate single precision (binary32) functions, since there are at most 2^{32} possible inputs that could be exhaustively checked rather quickly. However, this approach does not scale for univariate double precision (binary64) functions, where there are up to 2^{64} inputs to check. For binary64, the TMD was still unsolved before this work for trigonometric functions (sine, cosine, tangent). We fill this gap so that the TMD is now fully solved for univariate binary64 functions. As a consequence, there is no more obstacle to *require* (and not only recommend) correctly rounded mathematical functions in the next revision of IEEE 754 (due in 2029).

The contributions of this paper are the following: (a) we exhibit an efficient algorithm to perform an exhaustive search for hard-to-round cases of trigonometric functions; (b) we demonstrate that an efficient implementation of this algorithm using multi-threading and SIMD instructions can check a full binary64 binade in a few hours; (c) we computed a full set of hard-to-round cases for the binary64 sine, cosine and tangent functions, and exhibit the worst cases. With this new algorithm, trigonometric functions are not much harder to check than other functions. As a side result, the same kind of exhaustive search algorithm can be applied to other functions.

Notations: throughout the paper, we use h to denote the exponent of a binade $[2^{h-1}, 2^h)$. The notation $a \operatorname{cmod} b$ denotes a remainder with a centered modulus, i.e., $a \operatorname{cmod} b = r$ with $|r| \le b/2$, while $a \bmod b = r$ means the classical remainder $0 \le r < b$. We also assume the use of a computer with unsigned 64-bit arithmetic.

II. State of the art

We call an algorithm to solve the TMD *linear* or *exhaustive* (resp. *sublinear*) if it checks (resp. does not check) every input x individually. Sublinear algorithms to solve the TMD are Lefèvre's algorithm and the SLZ algorithm. Lefèvre's algorithm [9] reduces the search to finding integer points near a line; Lefèvre designed an efficient technique to solve that problem. The SLZ algorithm [12] is a generalization that uses a polynomial approximation of degree $d \ge 1$, whereas Lefèvre's algorithm uses degree 1. The TMD is reduced to a lattice reduction problem, for which efficient implementations of the LLL algorithm are used (see [12] for more details). The SLZ algorithm is implemented in the BaCSeL software tool [6]. BaCSeL was intensively used within the CORE-MATH project [11] to compute hard-to-round cases of binary64 functions.

Linear algorithms where each individual check calls a reference implementation with a larger precision (for example, GNU MPFR [3]) are too expensive for double precision: they take of the order of 10^4 cycles per input for the larger binades, see Appendix A. However, if a linear algorithm uses a few cycles per input x, it can be competitive with sublinear algorithms. This was demonstrated by de Dinechin, Muller, Pasca and Plesco in [2]. They addressed the binary64 exponential function, and estimated a total time of 125 hours for one binade with 100 cores at a frequency of 100 MHz. Unfortunately, the FPGA built for this paper never worked, thus no hard-to-round case was ever computed. They use the table-of-difference method, which was used earlier in Lefèvre's PhD thesis [9] (however, not for an exhaustive search).

Before this work, according to [1], hard-to-round cases for the binary64 trigonometric functions were known only for $|x| < 2^{11}$ for sin, cos, and for $|x| < 10.5\pi$ for tan. Since binary64 numbers can be as large as about 2^{1024}, the TMD was far from being solved for these functions.

We recall here the algorithm from [5]. Since $\cos$ is an even function, and $\sin$, $\tan$ are odd functions, we can restrict to positive inputs. Consider a binade $x \in [2^{h-1}, 2^h)$ in binary64 (all numbers considered here are in the normal range). Each floating-point number in that binade can be written $x = m \cdot u$, where $u = 2^{h-53}$ is the binade ulp (unit in last place), and m is an integer, $2^{52} \le m < 2^{53}$. The main idea of [5] is to deal with arithmetic progressions $x_j = x_0 + jqu$ such that qu is small modulo 2π, so that $\sin x_0$ and $\sin x_j$ are close. Then Lefèvre's algorithm or the SLZ algorithm can be applied. For example, for the binade $[2^{1023}, 2^{1024})$, we have $u = 2^{971}$, and we can use $q = 15106909301$, which yields $qu \approx 4.41 \cdot 10^{-13} \operatorname{cmod} (2\pi)$. We then have to search among q arithmetic progressions in that binade, using either Lefèvre's algorithm or SLZ, with each progression having about 300,000 numbers. One drawback of the implementation of that algorithm in BaCSeL is its large overhead for these relatively small arithmetic progressions.

The integer q is taken as a denominator of a convergent of $u/(2\pi)$. Here $q = 15106909301$ is the denominator of the 17th convergent of $u/(2\pi)$. For this q and the denominators of the previous and next convergents, Table I shows the value of $\tau = qu \operatorname{cmod} (2\pi)$ and the approximate number n of binary64 numbers distant by qu in the target binade. We see that the larger is q, the smaller is $|\tau|$, i.e., the most efficient will be Lefèvre's and the SLZ algorithms, but on the other side, n decreases, thus we can only check a few numbers per arithmetic progression.

TABLE I

DIFFERENT POSSIBLE VALUES OF q FOR THE BINADE $[2^{1023}, 2^{1024})$, WITH CORRESPONDING APPROXIMATE VALUES OF $\tau = qu \operatorname{cmod} (2\pi)$, AND $n \approx 2^{52}/q$.

q	3087468052	15106909301	14233796029594
τ	$-4.2 \cdot 10^{-10}$	$4.4 \cdot 10^{-13}$	$-7.6 \cdot 10^{-14}$
n	$1.5 \cdot 10^{6}$	300,000	320

A search for worst cases of sin for the binade $[2^{1023}, 2^{1024})$ was done in 2023 by the CORE-MATH project using the BaCSeL implementation of [5], with $q = 15106909301$, using degree-2 polynomials. This search was done on a cluster of Intel Xeon Gold 6130 processors, each one having 64 cores. This search took a total of 847 hours (real time), which corresponds to about 6 core-years, and found 1038 hard-to-round cases with at least 43 identical bits after the round bit. For all binades of the binary64 format, restricting to $h > 0$, the total time would thus be of the order of 6000 core-years. However, it might be that the binade $[2^{1023}, 2^{1024})$ is not representative of an "average binade" (our experiments tend to show that it is easier than an average binade).

The paper is organized as follows. In Section III we describe our new algorithm to search hard-to-round cases of the sine function. Then in Section IV we give experimental results of our implementation, and exhibit the worst cases for the sine and cosine functions. Then Section V deals with the tangent function, and we conclude in Section VI.

III. PROPOSED ALGORITHM

Our algorithm follows [5]. First, split the computation per binade. Then, for a given binade, first find an optimal integer q such that $qu \operatorname{cmod} (2\pi)$ is small, then split the search into q arithmetic progressions. This high-level algorithm is detailed in §III-A, while the algorithm dealing with a given arithmetic progression is detailed in §III-B and §III-C. Some improvements are described in §III-D to §III-I.

A. Splitting the search into arithmetic progressions

Assume that $qu = \tau \operatorname{cmod} (2\pi)$, with τ small (we assume $\tau > 0$ for simplicity here). The Taylor approximation of $\sin(x_0 + jqu)$ with explicit remainder is:

$$\sin(x_0 + j\tau) = \sin x_0 + j\tau \cos x_0 - \frac{1}{2}(j\tau)^2 \sin x_0 - \frac{1}{6}(j\tau)^3 \cos x_0 + \frac{1}{24}(j\tau)^4 \sin \xi \quad (1)$$

with $\xi \in [x_0, x_0 + j\tau]$, thus since $|\sin \xi| \le 1$:

$$|\sin(x_0 + j\tau) - (a + jb + j^2 c + j^3 d)| \le e, \quad (2)$$

where $a \approx \sin x_0$, $b \approx \tau \cos x_0$, $c \approx -\frac{\tau^2}{2} \sin x_0$, $d \approx -\frac{\tau^3}{6} \cos x_0$, and e accounts for the error term $\frac{(j\tau)^4}{24} \sin \xi$ and for the rounding errors in a, b, c, d.

This leads to Algorithm SearchAll.

Algorithm 1 (SearchAll)

Input: a binade $[2^{h-1}, 2^h)$, an integer m
Output: all binary64 numbers $x \in [2^{h-1}, 2^h)$ such that $\sin x$ has at least m identical bits after the round bit

let $u = 2^{h-53}$
find the smallest denominator q in the convergents of $u/(2\pi)$ such that $(n\tau)^4/24 \le 2^{-74}$, with $\tau = qu \operatorname{cmod} (2\pi)$, and $n = \lceil 2^{52}/q \rceil$
for i from 0 to $q - 1$
  let $x_0 = 2^{h-1} + iu$ and $v = \frac{1}{2}\operatorname{ulp}(\sin x_0)$
  compute a such that $|a - \sin x_0| < 2^{-65} v$
  compute b such that $|b - \tau \cos x_0| < 2^{-65} v$
  compute c such that $|c + \frac{\tau^2}{2} \sin x_0| < 2^{-65} v$
  compute d such that $|d + \frac{\tau^3}{6} \cos x_0| < 2^{-65} v$
  call SearchOne$(x_0, q, n, a, b, c, d, m)$

For each binade $[2^{h-1}, 2^h)$, $11 \le h \le 1024$, we computed the smallest value of q that satisfies the condition $(n\tau)^4/24 \le 2^{-74}$ (this condition is explained in the proof of Theorem 1). We started at 2^{10} to get a partial overlap with the values computed by CORE-MATH, which will enable us to check if we get the same worst cases. We find that the average of q is $\approx 1.4 \cdot 10^{11}$, and that of n is about 87,000.

B. Searching in an arithmetic progression

The search in a given arithmetic progression x_0, $x_0 + qu$, $x_0 + 2qu$, … is done using Algorithm SearchOne. This algorithm is split in two parts. Lines 2-10 form the initialization, while lines 11-15 are the critical loop, which is using the table-of-differences method.

More precisely, lines 2-6 initialize the 64-bit integer variables A, B, C, D, E that will be used in the table-of-differences method, line 7 shifts A by E so that we can check $0 \le A \le 2E$ instead of $-E \le A \le E$ (interpreting A as a signed integer), and lines 8-10 initialize the table-of-differences variables.

Algorithm 2 (SearchOne)

Input: a binary64 number x_0, two integers q and n, floating-point numbers a, b, c, d, and an integer m

Assume: $|a - \sin x_0| < 2^{-65}v$, $|b - \tau \cos x_0| < 2^{-65}v$, $|c + \frac{\tau^2}{2}\sin x_0| < 2^{-65}v$, $|d + \frac{\tau^3}{6}\cos x_0| < 2^{-65}v$, with $\tau = qu \text{ cmod } (2\pi)$, $u = \mathrm{ulp}(x_0)$, and $v = \frac{1}{2}\mathrm{ulp}(\sin x_0)$

Output: all binary64 numbers $x_0 + jqu$, $0 \le j < n$, such that $\sin x$ has at least m identical bits after the round bit

1: check all $\sin(x_0 + jqu)$ do lie in the same binade
2: $A \leftarrow \mathrm{round}(2^{64} \cdot a/v) \bmod 2^{64}$
3: $B \leftarrow \mathrm{round}(2^{64} \cdot b/v) \bmod 2^{64}$
4: $C \leftarrow \mathrm{round}(2^{64} \cdot c/v) \bmod 2^{64}$
5: $D \leftarrow \mathrm{round}(2^{64} \cdot d/v) \bmod 2^{64}$
6: $E \leftarrow 2^{44} + n^3 + n^2 + n + 1 + 2^{64-m}$
7: $A \leftarrow A + E \bmod 2^{64}$
8: $B \leftarrow B + C + D \bmod 2^{64}$
9: $C \leftarrow 2C + 6D \bmod 2^{64}$
10: $D \leftarrow 6D \bmod 2^{64}$
11: for $j = 0$ to $n - 1$
12:     if $A \le 2E$ then Check$(x_0 + jqu)$
13:     $A \leftarrow A + B \bmod 2^{64}$
14:     $B \leftarrow B + C \bmod 2^{64}$
15:     $C \leftarrow C + D \bmod 2^{64}$

In Algorithm SearchOne, $\mathrm{round}(\cdot)$ rounds to a nearest integer (with any tie-breaking rule). The Check routine at line 12 is specified in Appendix A.

We first check in Algorithm SearchOne that all values of $\sin(x_0 + jqu)$ lie in the same binade. This check can be done by checking that a and $a + nb$ (also taking into account rounding errors) lie in the same binade. Since nb is small, this check almost always succeeds. When this check fails, if the progression length n is larger than a certain threshold (determined empirically), we use dichotomy by calling SearchOne$(x_0, q, \lfloor n/2 \rfloor, a, b, c, d, m)$ and SearchOne$(x_0 + \lfloor n/2 \rfloor qu, q, \lceil n/2 \rceil, a', b', c', d', m)$, where a', b', c', d' are recomputed following steps 5-8 in Algorithm SearchAll corresponding to $x_0 + \lfloor n/2 \rfloor qu$. Otherwise we use a naive algorithm when n is sufficiently small.

In Algorithm SearchOne, most of the time is spent in the loop at lines 11-15, with the test $A \le 2E$ being false (in real code, the value $2E$ should be precomputed).

For values of q which are small, we get a large value of n. For example, for $h = 783$, we get $q = 4211615663$ and $n = 1069329$. Then the term n^3 in E will be very large, and the test $A \le 2E$ will often succeed. To mitigate this, we cut the interval for j into smaller pieces, and perform the initialization of Algorithm SearchOne for each one.

In the C language, the operation $A \leftarrow A + B \bmod 2^{64}$ is simply written $A = A + B$: this is exactly the semantics of the addition of 64-bit unsigned variables.

The term E = $2^{44} + n^3 + n^2 + n + 1 + 2^{64-m}$ is explained in the proof of Theorem 1.

Since the average of n is about 87,000 from §III-A, the loop at lines 11-15 of Algorithm SearchOne will run over 87,000 values on average. Since each loop performs three additions and a test (which should not cost much if the branch prediction correctly predicts it to false), the critical loop of Algorithm SearchOne will cost a few hundreds of thousands cycles (cumulated on all n values of j). Therefore, to get an efficient search, we need that the computation of a, b, c, d in Algorithm SearchAll and the initialization of A, B, C, D, E in Algorithm SearchOne cost at most a few hundreds of thousands cycles too. This is detailed in the rest of this section.

C. Computation of a, b, c, d in Algorithm SearchAll

We propose the following algorithm to efficiently compute floating-point approximations $a \approx \sin x_0$, $b \approx \tau \cos x_0$, $c \approx -\frac{\tau^2}{2}\sin x_0$ and $d \approx -\frac{\tau^3}{6}\cos x_0$, for $x_0 = 2^{h-1} + iu$, $0 \le i < q$. Note that since τ only depends on q, it can be computed once for all for a given value of q, whereas $\sin x_0$ and $\cos x_0$ depend on the given arithmetic progression. The computation of a, b, c, d thus mainly consists in approximating $\sin x_0$ and $\cos x_0$.

First, since $u = 2^{h-53}$, we can write $x_0 = ju$ with $2^{52} \le j < 2^{53}$. If we approximate $\sin u$ and $\cos u$ with sufficient precision, we can approximate $\sin(ju)$ and $\cos(ju)$ in a few cycles using binary exponentiation (see Algorithm 3).

Algorithm 3 (ComputeAB)

Input: an integer $j > 0$, two numbers $s \approx \sin u$ and $c \approx \cos u$

Output: approximations $S \approx \sin(ju)$ and $C \approx \cos(ju)$

1: write $j = 2^k + b_1 2^{k-1} + \cdots + b_{k-1}2^1 + b_k 2^0$ ▷ binary decomposition
2: $(S, C) \leftarrow (s, c)$
3: for i from 1 to k do
4:     $(S, C) \leftarrow (2SC, C^2 - S^2)$
5:     if $b_i = 1$ then
6:         $(S, C) \leftarrow (sC + cS, cC - sS)$
7: return (S, C)

From the values S, C computed by Algorithm ComputeAB, we deduce $a = S$, $b = \tau C$, $c = -\frac{\tau^2}{2}S$ and $d = -\frac{\tau^3}{6}C$ for Algorithm SearchAll, where $-\frac{\tau^2}{2}$ and $-\frac{\tau^3}{6}$ can be precomputed.

Note: Algorithm ComputeAB assumes that all values s, c, S, C have absolute value at most 1 (this should be enforced by the implementation).

Lemma 1. *Assume the values s, c, S, C in Algorithm 3 are fixed-point numbers rounded to 2^{-p}. If $\max(|s - \sin u|, |c - \cos u|) \le 2^{-p}$, we have at the end:*

$$\max(|S - \sin(ju)|, |C - \cos(ju)|) < 8^k \cdot 2^{-p+1}.$$

Proof. Let S_0 be the initial value of S, S'_i be the value of the variable S after step 4 of loop i, S_i be the value of the variable S at the end of loop i, and similarly for C_0, C'_i, and C_i. Also denote j_i the number formed by the $i + 1$ upper bits from j, and write $\max(|S_i - \sin(j_i u)|, |C_i - \cos(j_i u)|) \leq \varepsilon_i 2^{-p}$ with $\varepsilon_0 = 1$. By induction, the product $2SC$ yields:

$$\begin{aligned} & |2S_{i-1}C_{i-1} - \sin(2j_{i-1}u)| \\ = \quad & 2|S_{i-1}C_{i-1} - \sin(j_{i-1}u)\cos(j_{i-1}u)| \\ \leq \quad & 2|S_{i-1} - \sin(j_{i-1}u)||C_{i-1}| \\ + \quad & 2|\sin(j_{i-1}u)||C_{i-1} - \cos(j_{i-1}u)| \leq 4\varepsilon_{i-1}2^{-p}. \end{aligned}$$

If the product $2SC$ is first computed exactly, then rounded to 2^{-p}, the total rounding error for S'_i after step 4 is thus:

$$|S'_i - \sin(2j_{i-1}u)| \leq (4\varepsilon_{i-1} + 1)2^{-p}.$$

Similarly for $C^2 - S^2$:

$$\begin{aligned} & |(C^2_{i-1} - S^2_{i-1}) - (\cos^2(j_{i-1}u) - \sin^2(j_{i-1}u))| \\ \leq \quad & |C^2_{i-1} - \cos^2(j_{i-1}u)| + |S^2_{i-1} - \sin^2(j_{i-1}u)| \\ \leq \quad & |C_{i-1} - \cos(j_{i-1}u)||C_{i-1}| \\ + \quad & |\cos(j_{i-1}u)||C_{i-1} - \cos(j_{i-1}u)| \\ + \quad & |S_{i-1} - \sin(j_{i-1}u)||S_{i-1}| \\ + \quad & |\sin(j_{i-1}u)||S_{i-1} - \sin(j_{i-1}u)| \leq 4\varepsilon_{i-1}2^{-p}. \end{aligned}$$

If both products C^2_{i-1} and S^2_{i-1} are computed exactly, then rounded to 2^{-p}, and then subtracted (exactly), the total error for C'_i after step 4 is thus:

$$|C'_i - \cos(2j_{i-1}u)| \leq (4\varepsilon_{i-1} + 2)2^{-p}.$$

When $b_i = 0$, we thus have $\varepsilon_i \leq 4\varepsilon_{i-1} + 2$.

When $b_i = 1$, we get further errors. We then have $j_i = 2j_{i-1} + 1$, thus for $sC + cS$:

$$\begin{aligned} & |(sC'_i + cS'_i) - \sin(j_i u)| \\ \leq \quad & |sC'_i - \sin u \cos(2j_{i-1}u)| + |cS'_i - \cos u \sin(2j_{i-1}u)| \\ \leq \quad & |s - \sin u||C'_i| + |\sin u||C'_i - \cos(2j_{i-1}u)| \\ + \quad & |c - \cos u||S'_i| + |\cos u||S'_i - \sin(2j_{i-1}u)| \\ \leq \quad & (2\varepsilon_0 + 4\varepsilon_{i-1} + 2 + 4\varepsilon_{i-1} + 1)2^{-p}. \end{aligned}$$

If both products sC and cS are computed exactly, then rounded to 2^{-p}, and added (exactly), the total error for S_i at the end of loop i is thus:

$$|S_i - \sin(j_i u)| \leq (8\varepsilon_{i-1} + 2\varepsilon_0 + 5)2^{-p}.$$

For $cC - sS$, we get the same bound, assuming we compute the products cC and sS exactly, round them to 2^{-p}, and subtract them (exactly). Using $\varepsilon_0 = 1$, we get the recurrence $\varepsilon_i \leq 8\varepsilon_{i-1} + 7$. This yields $\varepsilon_i + 1 \leq 8(\varepsilon_{i-1} + 1)$, thus by recurrence $\varepsilon_k + 1 \leq 8^k(\varepsilon_0 + 1) = 2 \cdot 8^k$. □

Since in our case $j < 2^{53}$, we have $k \leq 52$ in Algorithm ComputeAB, thus the factor $2 \cdot 8^k$ means we lose at most 157 bits. On the other hand, we know (cf [10], section 10.2.2) that for binary64 inputs, a reduced argument by $C = \pi/2$ will never be of absolute value less than $2^{-60.89}$, moreover the reduction by 2π cannot yield smaller values than the reduction by $\pi/2$. Since we want a and b to be approximations with error less than $2^{-65}v$, and since $v \geq \frac{1}{2}\mathrm{ulp}(\sin 2^{-60.89}) = 2^{-114}$, this means the fixed-point precision of 2^{-p} in Algorithm ComputeAB should satisfy $p \geq 65 + 114 + 157 = 336$. Thus on a 64-bit computer, fixed-point numbers encoded on 6 words will suffice.

However, for a real number t, we have $|\sin t| + |\cos t| \leq \sqrt{2}$. Thus in the bound for $|2S_{i-1}C_{i-1} - \sin(2j_{i-1}u)|$, the term $|C_{i-1}| + |\sin(j_{i-1}u)|$ is bounded by $\sqrt{2} + \varepsilon_{i-1}2^{-p}$ instead of 2:

$$|S'_i - \sin(2j_{i-1}u)| \leq (2\sqrt{2}\varepsilon_{i-1} + 1)2^{-p} + 2\varepsilon^2_{i-1}2^{-2p}.$$

The same argument yields:

$$|C'_i - \cos(2j_{i-1}u)| \leq (2\sqrt{2}\varepsilon_{i-1} + 2)2^{-p} + 2\varepsilon^2_{i-1}2^{-2p},$$

and:

$$|S_i - \sin(j_i u)| \leq (4\varepsilon_{i-1} + 2\varepsilon_0 + 2\sqrt{2} + 2)2^{-p} + 2\sqrt{2}\varepsilon^2_{i-1}2^{-2p},$$

and similarly for $|C_i - \cos(j_i u)|$. Assuming $\varepsilon_0 \leq 1/2$, which can be obtained if we round $\sin u$ and $\cos u$ to nearest, this yields the recurrence $\varepsilon_i \leq 4\varepsilon_{i-1} + 6 + 2\sqrt{2}\varepsilon^2_{i-1}2^{-p}$, where we used $2\varepsilon_0 + 2\sqrt{2} + 2 \leq 6$. Assuming $2\sqrt{2}\varepsilon^2_{i-1} \leq 2^p$ (which we will check afterwards), this yields $\varepsilon_i \leq 4\varepsilon_{i-1} + 7$, thus $\varepsilon_i + 7/3 \leq 4(\varepsilon_{i-1} + 7/3) \leq 4^i(\varepsilon_0 + 7/3) \leq 17/6 \cdot 4^i$, thus $\varepsilon_k \leq 4^{k+1}$. The factor 4^{k+1} means we lose at most 106 bits for $k \leq 52$. Whence the fixed-point precision 2^{-p} in Algorithm ComputeAB should satisfy $p \geq 65 + 114 + 106 = 285$. Thus on a 64-bit computer, fixed-point numbers with 5 words will suffice. We now check the hypothesis $2\sqrt{2}\varepsilon^2_{i-1} \leq 2^p$. Assuming $\varepsilon_{i-1} \leq 4^i$ by induction, it follows $2\sqrt{2}\varepsilon^2_{i-1} \leq 4 \cdot 4^{2i} \leq 2^{106}$ (since $i \leq k \leq 52$), which is clearly less than 2^{285}.

To compute the values of a, b, c, d in Algorithm SearchAll, we implemented a fixed-point arithmetic using P words of 64 bits (with $P = 5$ or $P = 6$, see below), to represent numbers $|x| < 1$, with x multiple of 2^{-64P}. This implementation is based on the `mpn` layer from GNU MP [4]. Each operation (addition, subtraction, multiplication) is performed with an error less than 2^{-64P}.

Theorem 1. *Algorithm SearchAll is correct, i.e., it outputs all binary64 numbers $x \in [2^{h-1}, 2^h)$ such that $\sin(x)$ has at least m identical bits after the round bit.*

Proof. Since SearchAll calls SearchOne for q arithmetic progressions covering the binade $[2^{h-1}, 2^h)$, it suffices to prove the correctness of SearchOne.

In SearchOne, A approximates $2^{64}\sin(x_0)/v \bmod 2^{64}$, recalling $v = \frac{1}{2}\mathrm{ulp}(\sin x_0)$. For two real values X and Y, we consider here $|(X - Y) \operatorname{cmod} 2^{64}|$ which is the absolute value of the centered modulus. More precisely:

$$\left| \left[A - 2^{64} \cdot \frac{\sin(x_0)}{v} \right] \operatorname{cmod} 2^{64} \right| \leq$$

$$\left| \left[A - 2^{64} \cdot \frac{a}{v} \right] \operatorname{cmod} 2^{64} \right| + \frac{2^{64}}{v} \cdot |a - \sin x_0| \leq \frac{1}{2} + \frac{1}{2} \leq 1.$$

Similarly, we get:

$$\left|\left[B - 2^{64}\cdot\frac{\tau\cos(x_0)}{v}\right] \operatorname{cmod} 2^{64}\right| \le 1,$$
$$\left|\left[C - 2^{64}\cdot\frac{(-\tau^2)\sin(x_0)}{2v}\right] \operatorname{cmod} 2^{64}\right| \le 1,$$
$$\left|\left[D - 2^{64}\cdot\frac{(-\tau^3)\cos(x_0)}{6v}\right] \operatorname{cmod} 2^{64}\right| \le 1.$$

It follows from Eq. (1), using the fact that $\sin\xi$ lies in the same binade as $\sin(x_0)$ (which follows from the check at line 1 of Algorithm SearchOne), thus $|\sin\xi| \le 2^{54}v$:

$$\begin{aligned}
&\left|\left[\frac{2^{64}}{v}\sin(x_0+j\tau) - (A+Bj+Cj^2+Dj^3)\right] \operatorname{cmod} 2^{64}\right| \\
&\le 1 + j + j^2 + j^3 + \frac{2^{64}}{v}\cdot\frac{(j\tau)^4}{24}\cdot 2^{54}v \\
&\le 1 + n + n^2 + n^3 + 2^{44},
\end{aligned}$$

where we used $j \le n$ and $(n\tau)^4/24 \le 2^{-74}$.

Now assume $\sin(x_0+j\tau)$ has at least m identical bits after the round bit. Let z be the rounding to nearest of $\sin(x_0 + j\tau)$ with 54 bits of precision, and $Z = 2^{64}z/v$, which is an integer multiple of 2^{64}. Then $|\sin(x_0+j\tau) - z| \le 2^{-m}v$, thus $|\frac{2^{64}}{v}\sin(x_0+j\tau) - Z| \le 2^{64-m}$ and:

$$\left|(A+Bj+Cj^2+Dj^3) \operatorname{cmod} 2^{64}\right| \le E,$$

with $E = 1+n+n^2+n^3+2^{44}+2^{64-m}$. With $A' = A+E \bmod 2^{64}$, this implies:

$$(A'+Bj+Cj^2+Dj^3) \bmod 2^{64} \le 2E,$$

which is exactly the condition used in SearchOne to call Check. □

D. Batch computation

To mitigate the cost of Algorithm ComputeAB, which in our case for $k = 52$ performs 52 "doublings" of sin/cos values, and on average 26 "multiplications", thus a total of 78 sin/cos operations, we can approximate $\sin(ju)$ and $\cos(ju)$ for k consecutive j-values $j = j_0, j_0 + 1, \ldots, j_0+k-1$ as follows. First approximate $\sin(j_0u)$ and $\cos(j_0u)$ using Algorithm ComputeAB, then use the relations $\sin((j+1)u) = \sin u\cos(ju) + \cos u\sin(ju)$ and $\cos((j+1)u) = \cos u\cos(ju) - \sin u\sin(ju)$. For $0 \le i < k$, let $j_i = j_0 + i$, let S_i, C_i denote the approximations of $\sin(j_iu), \cos(j_iu)$, and let τ_i such that $\max(|S_i - \sin(j_iu)|, |C_i - \cos(j_iu)|) \le \tau_i 2^{-p}$.

If $|s - \sin u|, |c - \cos u| \le 2^{-p}$:

$$\begin{aligned}
&|(sC_i + cS_i) - \sin((j_i+1)u)| \\
\le\ & |sC_i - \sin u\cos(j_iu)| + |cS_i - \cos u\sin(j_iu)| \\
\le\ & |s - \sin u||C_i| + |\sin u||C_i - \cos(j_iu)| \\
+\ & |c - \cos u||S_i| + |\cos u||S_i - \sin(j_iu)| \\
\le\ & 2^{-p}(|C_i| + |S_i|) + \tau_i 2^{-p}(|\sin u| + |\cos u|) \\
\le\ & 2^{-p}(\sqrt{2} + \tau_i 2^{-p+1}) + \tau_i 2^{-p}\sqrt{2} \\
\le\ & 2^{-p}\sqrt{2} + (1 + 2^{-p+\frac{1}{2}})\tau_i 2^{-p}\sqrt{2},
\end{aligned}$$

where we used $|C_i| + |S_i| \le |\cos(j_iu)| + |\sin(j_iu)| + |C_i - \cos(j_iu)| + |S_i - \sin(j_iu)| \le \sqrt{2} + 2\tau_i 2^{-p}$. If both products sC_i and cS_i are computed exactly, then rounded to 2^{-p}, and added (exactly), this yields an additional error of $2\cdot 2^{-p}$. Similarly, we have the same bound for approximating $\cos(j_{i+1}u)$ with $C_{i+1} \approx cC_i - sS_i$. We thus get the recurrence $\tau_{i+1} \le 2 + \sqrt{2} + \sqrt{2}\tau_i(1 + 2^{-p+\frac{1}{2}})$, which admits as solution

$$\tau_i \le 2^{\frac{i}{2}}(1 + 2^{-p+\frac{1}{2}})^i(\tau_0 + 4 + 3\sqrt{2}) - 4 - 3\sqrt{2}.$$

Thus we lose about half a bit per iteration, since the term $(1+2^{-p+\frac{1}{2}})^i$ remains near 1. If we process batches of $k = 128$ values, we trade the $128\cdot 78 = 9984$ sin/cos operations on $P = 5$ words (without batch computation) for 78 sin/cos operations for the first value of j, plus one sin/cos operation for the remaining values of j, thus a total of 205 sin/cos operations on $P = 6$ words, thus a speedup of about 34 in the initialization, assuming the sin/cos operations are in $O(P^2)$.

Table II shows the speedup obtained using batch computations. We see that we save from 18% to 36% for "average" h-values, and up to a factor of 50 for $h = 11$, which corresponds to very small arithmetic progressions ($n = 163$).

TABLE II
WALL-CLOCK TIME (IN SECONDS) WITHOUT AND WITH BATCH COMPUTATION (FOR BATCHES OF $k = 128$ VALUES) FOR A SAMPLE OF $1/s$ OF THE INPUTS PER BINADE, ON AN AMD EPYC 9754 PROCESSOR WITH 512 CORES AND GCC 10.2.1.

h	11	86	449	461	801	920
s	10^5	1000	1000	1000	1000	1000
no batch	43.9s	12.3s	13.4s	12.8s	14.2s	12.1s
batch	0.9s	9.5s	9.0s	9.4s	9.1s	9.8s

E. Tuning the values of q and n

For some binades, the value of $n \approx 2^{52}/q$ might be large. For example, for $h = 783$, we get $q = 4211615663$ and $n = 1069329$. In this case, the term n^3 in the error bound E (line 6 of Algorithm SearchOne) is about 2^{60}. This means the test $A \le 2E$ will succeed with probability about 12.5%, and one will perform many expensive calls to Check. To mitigate this, we cap the value of n so that the n^3 error term does not exceed the 2^{44} term, for example, $n \le n_{\max} := 26007$. The test $(n\tau)^4/24 \le 2^{-74}$ at line 2 of Algorithm SearchAll is done using $n = \min(\lceil 2^{52}/q\rceil, n_{\max})$; thus a smaller value of q might pass this test. If the length $\lceil 2^{52}/q\rceil$ of the arithmetic progressions exceeds $n_{\max}$, we cut it into smaller arithmetic progressions of length not exceeding $n_{\max}$. We call this process "tuning the values of q and n".

If the arithmetic progression is cut into chunks starting at x_0, $x_1, x_2, \ldots$, with $x_{i+1} = x_i + n_{\max}qu$, we can reuse the "batch computation" idea from §III-D to deduce $\sin(x_{i+1}), \cos(x_{i+1})$ from $\sin(x_i), \cos(x_i)$, after having precomputed $\sin(n_{\max}qu)$, $\cos(n_{\max}qu)$.

The initialization cost for a given binade is inversely proportional to n. Capping n to $n_{\max}$ increases this cost, but limiting the error bound E has a much greater benefit. Conversely, when the non-tuned n is less than $n_{\max}$, tuning may allow one to increase n to $n_{\max}$ (due to a smaller q), thus reduce the initialization cost.

The tuned values are identical to the non-tuned values for 82 binades (about 8%), and differ for 922 binades (92%).

Table III shows the obtained speedup, without batch computation (batch computation makes it more difficult to select a sample of 0.1%). We see the speedup is larger when the non-tuned value of $2^{52}/q$ is small, in which case the initialization overhead is large.

TABLE III
TOTAL LENGTH $2^{52}/q$ OF THE ARITHMETIC PROGRESSIONS WITH THE NON-TUNED AND TUNED VERSIONS, AND CORRESPONDING TIMINGS, FOR A SAMPLE OF 0.1% OF THE INPUTS FOR EACH BINADE.

h	482	689
$2^{52}/q$ (non-tuned)	28731	4605
$2^{52}/q$ (tuned)	4565096037	1537558
speedup	2.05	13.8

F. Small values of h

With the tuning described in §III-E, we get $q = 1$ for $11 \leq h \leq 20$. This means that we have only one arithmetic progression to deal with. For example, for $h = 20$, we have $\tau = 2^{-33} \approx 1.2 \cdot 2^{-10}$. With arithmetic progressions capped to a length $n = 26007$, we have $(n\tau)^4/24 \approx 2^{-77.9}$ (and we get smaller values for smaller h). On a multi-core computer, we thus have to split the unique arithmetic progression into smaller chunks. We have implemented this, and on a 64-core AMD EPYC 7282, the estimated time for the full binade $[2^{10}, 2^{11})$ decreases to about 25.5 hours (wall-clock time).

G. Parallelism

To fully utilize modern CPUs, we implemented the algorithm using a hybrid approach that combines multi-threading with OpenMP for coarse-grained parallelism and SIMD intrinsics for fine-grained parallelism.

Our multi-threading strategy adapts to the structure of the search space, which changes with the arithmetic progression step q:

- When $q = 1$ (small h): The binade consists of a single, massive arithmetic progression. We partition this progression evenly into M contiguous intervals (where M is the number of threads) and assign one interval to each thread.
- When $q > 1$ (large h): The binade is covered by q distinct, interleaved arithmetic progressions with offsets $0, \ldots, q-1$. We parallelize the outer loop over these offsets, assigning batches of distinct progressions to different threads.

For vectorization, we targeted the x86-64 architecture, supporting SSE4.2, AVX2, and AVX512 instruction sets. These extensions allow us to process $W = 2$, $W = 4$, or $W = 8$ 64-bit integers per instruction, respectively. We developed a hybrid strategy to handle the table-of-differences recurrence in lines 12-15 in Algorithm SearchOne. Our primary method, used for the vast majority of the search space, is an *inter-progression* strategy. We group W independent arithmetic progressions into a single SIMD vector, where lane k manages the k-th progression. This allows the recurrence relations to remain independent across lanes, enabling the use of efficient 64-bit vector additions with minimal overhead.

However, grouping progressions requires them to have identical lengths to avoid lane divergence. To mitigate workload imbalance, we implemented a dual-queue strategy. A FullQueue collects standard, full-length progressions (constrained by $n_{\max}$), which are processed in batches of W with 100% lane utilization. A TailQueue collects irregular or truncated progressions.

When grouping is not possible—such as for the TailQueue or when a progression has been split by the dichotomy step—we fall back to a secondary, *intra-progression* strategy which aims to vectorize the main loop of the SearchOne algorithm directly. Here, we process a single progression by computing blocks of W steps in parallel. After initializing the SIMD vectors with the first W steps of the sequence, we advance these vectors by W steps at a time, using the W-step relation derived from lines 13-15 in Algorithm SearchOne:

$$\begin{aligned} A_{i+W} &= A_i + WB_i + \binom{W}{2} C_i + \binom{W}{3} D, \\ B_{i+W} &= B_i + WC_i + \binom{W}{2} D, \\ C_{i+W} &= C_i + WD. \end{aligned}$$

This approach requires 64-bit integer multiplication SIMD instructions to update the vectors, so we only implemented it for the AVX512 instruction set.

H. Reduction to degree 2

Experiments show that for most sequences, the term D of degree 3 in the loop of Algorithm SearchOne, interpreted as a signed integer, is small enough so that it can be ignored for a significant number of iterations while keeping the error bound E small enough, thus saving an addition for each iteration. So the interval of size n can be split into subintervals of size v, where v can be determined dynamically from the value of D; the last subinterval may be truncated. In each subinterval, the iteration is the same as in Algorithm SearchOne, except that $C \leftarrow C{+}D \bmod 2^{64}$ is not done. After each subinterval, we need to update the values of A, B and C so that they retrieve their values as if D were not ignored. Said otherwise, the original polynomial of degree 3 is approximated by a polynomial of degree 2 in each subinterval. This is an application of the hierarchical approximation of a polynomial by polynomials of smaller degrees, described in [9].

It can be proved by induction that in Algorithm SearchOne, we have after j iterations:

$$\begin{aligned} &A_j = A_0 + j\,B_0 + (j(j-1)/2)\,C_0 + (j(j-1)(j-2)/6)\,D \\ &B_j = B_0 + j\,C_0 + (j(j-1)/2)\,D \\ &C_j = C_0 + j\,D. \end{aligned}$$

So the update of the coefficients after v iterations will consist in doing

$$A \leftarrow A + (v(v-1)(v-2)/6)\, D \bmod 2^{64}$$
$$B \leftarrow B + (v(v-1)/2)\, D \bmod 2^{64}$$
$$C \leftarrow C + v\, D \bmod 2^{64}$$

where the factors $v(v-1)(v-2)/6\, D$, $v(v-1)/2\, D$ and $v\, D$ can be computed before the loops.

If A'_j denotes the value of A after j iterations when ignoring D, then $A'_j = A_j - (j(j-1)(j-2)/6)\, D$. So, by ignoring D, we introduce an error on A that is bounded by $((v-1)(v-2)(v-3)/6)\, |D|$ since j goes from 0 to $v-1$. This additional term needs to be added to E in line 6 of Algorithm SearchOne.

Ignoring D is interesting as long as the time saved by avoiding an addition per iteration is larger than the additional time due to the increased value of E and the update of the coefficients after each subinterval. The additional probability of failure is $((v-1)(v-2)(v-3)/3)\, |D|/2^{64}$, and after each subinterval, we need 3 additions. The choice of the value of v has an impact only on the time taken by these additional failures and the time taken by updating the subintervals (that is, the time taken by the external loop except the time spent in the internal loop).

I. Further possible improvements

We list here further possible improvements to be explored:

- Since $\sin(x + \pi) = -\sin(x)$, and x is a hard-to-round case for the sine function iff $-x$ is, in the computation of q, instead of requiring $qu \operatorname{cmod} (2\pi)$ to be small, we can require $qu \operatorname{cmod} \pi$ to be small. For $11 \le h \le 1024$, targeting $(n\tau)^4/24 \le 2^{-74}$, this yields a value of $2^{52}/q_{\text{avg}}$ which is larger by about 55%; this decreases the initialization cost.
- Since $\sin(x + \pi/2) = \cos(x)$, we can push the above idea even further, by requiring $qu \operatorname{cmod} (\pi/2)$ small. Then $\sin(x_0 + j\pi/2)$ will yield a hard-to-round case of sin if j is even, and of cos if j is odd, and similarly for $\cos(x_0 + j\pi/2)$. With this variant, hard-to-round cases of sin and cos should be computed simultaneously, with the same value of q. For $11 \le h \le 1024$, targeting $(n\tau)^4/24 \le 2^{-74}$, this yields a value of $2^{52}/q_{\text{avg}}$ which is larger by about 136% than with $qu \operatorname{cmod} (2\pi)$.

IV. Hard-to-round cases of sin and cos

We implemented the above algorithms in the C language, using the optimizations from §III-D, §III-E, §III-F, and §III-G (but without the one from §III-H which we found after the search was done), and tested it on various binades on an AMD EPYC 9754 processor with 256 cores (512 with hyperthreading) and gcc 10.2.1. We ran the algorithms with and without hyperthreading: we save about 30% with hyperthreading. We also tried with AVX2 instead of AVX512 since for some applications, AVX2 is faster. In our case, AVX512 is about 24% faster on a small experiment.

TABLE IV
Wall-clock time (in seconds) of our implementation, for a sample of 0.1% of the inputs for each binade.

h	36	86	449	461	801
n	62504	86544	73414	80691	67776
time	13.2s	12.7s	15.7s	13.4s	16.7s

We see on Table IV that the average time per binade is less than 14 seconds, for a sample of 0.1% of the inputs. This yields a wall-clock time of less than 4 hours for a full binade. Since we have access to up to 16 such computers, a full search for the sine function takes less than two weeks.

We ran our algorithm for all 1014 binades of $[2^{10}, 2^{1024})$, searching for hard-to-round cases with at least 43 identical bits after the round bit. We found a total of 1,048,756 hard-to-round cases, i.e., an average of 1034 per binade. We notice that many hard-to-round cases (in particular the worst ones) correspond to $\sin x$ being extremely close to ± 1 (see Table V). This can be explained as follows. For a given binade $[2^{h-1}, 2^h)$, consider its ulp $u = 2^{h-53}$, and compute the continued fraction expansion of $u/(\pi/2)$ [8]. Take the largest convergent p/q whose numerator p is odd and denominator q is representable in binary64. It is well known that $|u/(\pi/2) - p/q| < 1/q^2$, thus $|qu - p(\pi/2)| < \pi/(2q)$. This implies that $\sin x$ is near ± 1 for $x = qu$. Among all binades in $[2^{10}, 2^{1024})$, we find 535 values of q such that $q \ge 2^{52}$. Hence it is not surprising that this yields cases with 63 identical bits or more after the round bit.

TABLE V
Hardest-to-round cases of the binary64 sine for $x \ge 2^{10}$, where m is the number of identical bits of $\sin x$ after the round bit. All of them occur in directed rounding modes (D).

x	m	mode	$\sin x \approx$
`0x1.e009c53148be1p+991`	64	D	1.0
`0x1.cfe482285f8edp+860`	63	D	-1.0
`0x1.6ac5b262ca1ffp+849`	68	D	1.0
`0x1.db41f3cb71d7bp+680`	63	D	1.0
`0x1.4c96c11134d36p+577`	63	D	-1.0
`0x1.e7e44a78ac18cp+197`	63	D	-1.0
`0x1.230280c47f5c1p+136`	63	D	0.270
`0x1.504cac51f1eafp+131`	64	D	-1.0
`0x1.b951f1572eba5p+23`	65	D	-1.0

Remark: Toru Koizumi found three hard-to-round cases with at least 43 identical bits after the round bit which were missing from the CORE-MATH `sin.wc` file, namely `0x1.4f1d73be27a31p+1`, `0x1.821adbddf43c8p+1`, and `0x1.2d58d2e060d7dp+3`. These missing inputs were due to a wrong use of BaCSeL by the third author during his search for $|x| < 2^{11}$. We checked these inputs are found by our new algorithm (and with BaCSeL when used correctly).

The same algorithm readily extends to the cosine function. Equation (2) becomes:

$$|\cos(x_0 + j\tau) - (a + jb + j^2c + j^3d)| \le e, \qquad (3)$$

with $a = \cos x_0$, $b = -\tau \sin x_0$, $c = -\frac{\tau^2}{2}\cos x_0$, $d = \frac{\tau^3}{6}\sin x_0$, and $e = (n\tau)^4/24$, where n is a bound on j. We leave details to the reader.

Like for the sine function, we ran our algorithms for all 1014 binades of $[2^{10}, 2^{1024})$, searching for hard-to-round cases with at least 43 identical bits after the round bit. We found a total of 1,049,705 hard-to-round cases, i.e., an average of 1035 per binade. As with sin, many hard-to-round cases correspond to $\cos x$ being extremely close to ± 1 (see Table VI), and the explanation is similar, except that p needs to be even. Among all binades in $[2^{10}, 2^{1024})$, we find 223 values of q such that $q \geq 2^{52}$. This is less than half the number of values for sin. This can be explained by the fact that since p is even, q must be odd. In particular, this eliminates all candidate values of q that are larger than or equal to 2^{53} and representable in binary64, as these values are even. So, in the table of hard-to-round cases, m starts at 62 instead of 63.

TABLE VI
HARDEST-TO-ROUND CASES OF THE BINARY64 COSINE FOR $x \geq 2^{10}$, WHERE m IS THE NUMBER OF IDENTICAL BITS OF $\cos x$ AFTER THE ROUND BIT. ONE OF THEM OCCURS IN ROUNDING TO NEAREST (N), THE OTHER ONES IN DIRECTED ROUNDING MODES (D).

x	m	mode	$\cos x \approx$
`0x1.5afb7107105d9p+1006`	62	D	0.918
`0x1.e009c53148be1p+992`	62	D	−1.0
`0x1.b7fe89bf86037p+917`	62	N	−0.101
`0x1.6ac5b262ca1ffp+852`	62	D	1.0
`0x1.6ac5b262ca1ffp+851`	64	D	1.0
`0x1.6ac5b262ca1ffp+850`	66	D	−1.0
`0x1.1fa76750679fcp+285`	62	D	0.225
`0x1.504cac51f1eafp+132`	62	D	−1.0
`0x1.b951f1572eba5p+24`	63	D	−1.0

V. HARD-TO-ROUND CASES OF TAN

Extending our algorithm to the tangent function is not straightforward. Equation (2) becomes:

$$|\tan(x_0 + j\tau) - (a + jb + j^2c + j^3d)| \leq e, \qquad (4)$$

with $a = \tan x_0$, $b = \tau(a^2 + 1)$, $c = \tau^2(a^3 + a)$, $d = \tau^3/3 \cdot (3a^4 + 4a^2 + 1)$, and $e = (n\tau)^4/3 \cdot |3t^5 + 5t^3 + 2t|$, with $t = \tan \xi$, $\xi \in (x_0, x_0 + n\tau)$, and n is a bound on j. We have $|e/a| = (n\tau)^4/3 \cdot |3t^5 + 5t^3 + 2t|/|a|$. Assume that $\tan \xi \approx \tan x_0 = a$, then $|e/a| \approx f(a) \cdot (n\tau)^4/24$, where $f(u) := 24u^4 + 40u^2 + 16$. The function f varies a lot in magnitude: If we sample 1000 random values of x_0 in the binade $[2^{1023}, 2^{1024})$, the values of $f(\tan(x_0))$ vary from about 16 to more than 2^{53}. Therefore, there is a trade-off between the choice of q, which should not be too large, and the value of e, which should be small enough to avoid calling the expensive Check routine too often. This makes it difficult to compute a priori bounds. Apart from that, the coefficients a, b, c, d can be computed with Algorithm ComputeAB: first compute approximations s, c of $\sin x_0$ and $\cos x_0$, then deduce $a \approx s/c$, and compute b, c, d from a and τ. One extra difficulty is that we do not know a bound on a, b, c, d in absolute value, thus one has to deal with floating-point numbers instead of fixed-point numbers. Thus the P-word fixed-point arithmetic described in §III-C has to be replaced by a P-word floating-point arithmetic, for which we use MPFR.

To turn this reasoning into a rigorous algorithm we need to:

- rigorously bound the error e in Eq. (4). First ensure that $\tan x$ is monotonic on $(x_0, x_0 + n\tau)$ and lies in the same binade (if not, split the interval in two and recurse), then we know $|e| \leq (n\tau)^4/3 \cdot (3u^5 + 6u^3 + 2u)$ where $u = \max(|\tan x_0|, |\tan(x_0 + n\tau)|)$;
- bound the total error when approximating $\sin x_0$ and $\cos x_0$ by s and c, and deduce error bounds on the approximations a, b, c, d. These error bounds should be less than $2^{-65}v$ with $v = \frac{1}{2}\mathrm{ulp}(\tan x_0)$ as in Algorithm SearchAll. If this is not the case, split the interval in two or increase the working precision;
- in Algorithm SearchOne, deduce the 64-bit integers A, B, C, D, and E for the error bound, where the term 2^{44} in E is replaced by a bound on $2^{64}|e|/v$;
- run the critical loop of Algorithm SearchOne as for sin and cos.

Table VII shows that if we aim at a probability of 2^{-16} of calling the Check routine (which is expensive), we should use parameters such that $2(n\tau)^4/3 \approx 2^{-120}$. We see that when 2^k decreases, the average value of n also decreases, thus there is a trade-off between the cost of Check and the cost of the initialization for each arithmetic progression. With our program, we find the optimal is to require $2(n\tau)^4/3 \leq 2^{-95}$, which yields in practice $2(n\tau)^4/3 \approx 2^{-105.4}$ (there is a gap when going from one convergent of $u/(2\pi)$ to the next one, thus one cannot have exactly 2^{-95}), then we get a probability $2^{-12.9}$ of calling Check, which is coherent with the value of $p \approx 2^{-12.63}$ for $k = -105$ in Table VII.

TABLE VII
FOR A TARGET $2(n\tau)^4/3 \leq 2^k$, PROBABILITY p OF CALLING CHECK WITH RESPECT TO k (USING 10^7 RANDOM CALLS IN THE BINADE $h = 1024$), AND AVERAGE VALUE OF n FOR $21 \leq h \leq 1024$.

k	−90	−95	−100	−105	−110	−115
$\log_2 p$	−8.83	−10.06	−11.37	−12.63	−13.91	−15.03
avg n	17504	11173	7220	4347	2802	1831

We ran our algorithms for all 1020 binades of $[2^4, 2^{1024})$, searching for hard-to-round cases with at least 43 identical bits after the round bit. Indeed, the search with BaCSeL from the CORE-MATH project was done up to 10.5π, which lies in the binade $[2^5, 2^6)$, and we wanted to have a full binade where we could check the search with BaCSeL and our new algorithm give the same results. In the binade $[2^4, 2^5)$, BaCSeL did find 1034 hard-to-round cases, while our search finds 1035 hard-to-round cases. It appears BaCSeL missed `0x1.f671de534c343p+4`, with 44 identical bits after the round bit. We found a total of 1,045,244 hard-to-round cases, i.e., an average of 1025 per binade.

VI. CONCLUSION

In this paper, we propose a new algorithm to find hard-to-round cases of trigonometric functions. One of our con-

TABLE VIII

HARDEST-TO-ROUND CASES OF THE BINARY64 TANGENT FOR $x \geq 2^4$, WHERE m IS THE NUMBER OF IDENTICAL BITS OF $\tan x$ AFTER THE ROUND BIT. ONE OF THEM OCCURS IN ROUNDING TO NEAREST (N), THE OTHER ONES IN DIRECTED ROUNDING MODES (D).

x	m	mode	$\tan x \approx$
`0x1.20e3e80d2b617p+990`	61	D	-1.15
`0x1.94bb90326441ap+953`	61	D	-0.32
`0x1.52042b55571c6p+952`	60	D	-1.83
`0x1.fe6e530194af6p+681`	62	D	5.00
`0x1.8b4c4b528e351p+578`	60	N	-1.59
`0x1.57237795e9208p+324`	61	D	0.68

tributions is to drastically reduce the initialization cost. As a consequence, the obtained algorithm has asymptotic complexity $O(2^p)$ for a p-bit format, but with a very small constant (a few cycles per input).

We efficiently implemented this algorithm on a multi-core computer and using SIMD instructions, so that checking a full binade for the sine and cosine functions in double precision takes less than 4 hours on a 512-core node. As a side result, this demonstrates that an exhaustive check is now doable for double precision.

This enabled us to compute a full set of hard-to-round cases in double precision for sin, cos and tan, which were the only common univariate binary64 functions for which hard-to-round cases were unknown. The full set of hard-to-round cases is available in the CORE-MATH test suite (files `sin.wc`, `cos.wc`, `tan.wc`). We hope this work will help requiring correct rounding of univariate binary64 functions in the next revision of IEEE 754.

Acknowledgements.

The authors thank Bogdan Pasca for suggesting trying without hyperthreading, David Defour for suggesting trying with AVX2 instead of AVX512, and Stefanos Kourtis for his comments on an earlier version of this paper. The full search for hard-to-round cases was possible thanks to the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several universities as well as other organizations (see https://www.grid5000.fr/).

REFERENCES

[1] BRISEBARRE, N., HANROT, G., MULLER, J.-M., AND ZIMMERMANN, P. Correctly rounded evaluation of a function: why, how, and at what cost? *ACM Computing Surveys 58*, 1 (2026).

[2] DE DINECHIN, F., MULLER, J., PASCA, B., AND PLESCO, A. An FPGA architecture for solving the Table Maker's Dilemma. In *22nd IEEE International Conference on Application-specific Systems, Architectures and Processors, ASAP 2011* (2011), J. R. Cavallaro, M. D. Ercegovac, F. Hannig, P. Ienne, E. E. S. Jr., and A. F. Tenca, Eds., pp. 187–194.

[3] FOUSSE, L., HANROT, G., LEFÈVRE, V., PÉLISSIER, P., AND ZIMMERMANN, P. MPFR: A multiple-precision binary floating-point library with correct rounding. *ACM Trans. Math. Softw. 33*, 2 (2007), article 13.

[4] GRANLUND, T., AND THE GMP DEVELOPMENT TEAM. *GNU MP: The GNU Multiple Precision Arithmetic Library*, 6.3.0 ed., 2023.

[5] HANROT, G., LEFÈVRE, V., STEHLÉ, D., AND ZIMMERMANN, P. Worst cases of a periodic function for large arguments. In *Proceedings of the 18th IEEE Symposium on Computer Arithmetic (ARITH'18)* (2007), IEEE Computer Society Press, pp. 133–140.

[6] HANROT, G., LEFÈVRE, V., STEHLÉ, D., AND ZIMMERMANN, P. BaCSeL version 4.0. https://gitlab.inria.fr/zimmerma/bacsel.

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

[8] KAHAN, W., AND MCDONALD, S. Nearpi, a C program to exhibit large floating-point numbers very close to integer multiples of $\pi/2$. https://people.eecs.berkeley.edu/~wkahan/testpi/nearpi.c, 1984.

[9] LEFÈVRE, V. *Moyens arithmétiques pour un calcul fiable*. Thèse de doctorat, École Normale Supérieure de Lyon, 2000.

[10] MULLER, J.-M., BRUNIE, N., DE DINECHIN, F., JEANNEROD, C.-P., JOLDES, M., LEFÈVRE, V., MELQUIOND, G., REVOL, N., AND TORRES, S. *Handbook of Floating-Point Arithmetic (2nd edition)*. Birkhäuser Basel, 2018.

[11] SIBIDANOV, A., ZIMMERMANN, P., AND GLONDU, S. The CORE-MATH project. In *ARITH 2022 - 29th IEEE Symposium on Computer Arithmetic* (virtual, France, 2022).

[12] STEHLÉ, D., LEFÈVRE, V., AND ZIMMERMANN, P. Searching worst cases of a one-variable function using lattice reduction. *IEEE Transactions on Computers 54*, 3 (2005), 340–346.

APPENDIX

A. Checking candidates

We specify the Check routine used in line 12 of Algorithm SearchOne. At line 1, $\mathrm{RN}_{53+m}(\sin x)$ rounds $\sin x$ to nearest

Algorithm 4 (Check)

Input: a binary64 number x, an integer m

Output: output x if $\sin x$ has at least m identical bits after the round bit

1: $y \leftarrow \mathrm{RN}_{53+m}(\sin x)$

2: $z \leftarrow \mathrm{RN}_{54}(y)$

3: if $z = y$ output x

to a precision of $53+m$ bits, and at line 2, $\mathrm{RN}_{54}(y)$ rounds y to nearest at a precision of 54 bits. These computations might be done for example with MPFR. On a Intel Xeon Silver 4214, one call to check in the binade $[2^{1023}, 2^{1024})$ takes about 12,000 cycles with MPFR.

Lemma 2. *Algorithm Check is correct: if $\sin x$ has at least m identical bits after the round bit, then x is output.*

Proof. Assume $\sin x$ has at least m identical bits after the round bit. Then $\sin x$ is written in binary (modulo sign and exponent) $\underbrace{bbb...bbb}_{54}\underbrace{000...000}_{m}ccc...$ or $\underbrace{bbb...bbb}_{54}\underbrace{111...111}_{m}ccc...$ where $bbb...bbb$ represents the upper 54 bits and $ccc...$ the trailing bits. In the first case, $\sin x$ is rounded at line 1 to $y = y_0 := bbb...bbb$, and in the second case, it is rounded to the 54-bit number adjacent to y_0 away from zero. In both cases, y is representable on 54 bits, thus $z = y$, and x is output. □

Conversely, when x is output by Check, $\sin x$ is written in binary (modulo sign and exponent) into one of the two above forms, thus it has at least m identical bits after the round bit (in both cases, we cannot have $ccc... = 0$, since $\sin x$ cannot be exact for any non-zero floating-point number x).

B. Checking correctly rounded libraries

Once we know the hard-to-round inputs for trigonometric functions, we can test libraries claiming correct rounding.

`MathLib/Libultim`. There is no more any official repository for MathLib. We tried both the version integrated into GNU libc 2.27, and the unofficial repository at https://github.com/dreal-deps/mathlib. MathLib only claims correct rounding for rounding to nearest. The version in GNU libc 2.27 passes all tests for this rounding mode. However, the unofficial repository fails for two inputs for $\sin(x)$, namely $x = \pm$`0x1.022da8407edfp-2`. For $\cos(x)$, we get 54 failures with the unofficial repository, some of which are probably related to a bug[1] reported in 2002. For example, for $x =$ `0x1.b4287fe717028p-1`, instead of `0x1.5130dee743525p-1`, we get `0x1.81498bdcaa1e8p-1`, which is the correctly rounded *sine* of x. For $\tan(x)$, we get no error with the unofficial repository.

`Libmcr`. There is no more any official repository for Libmcr. We used the unofficial repository at https://github.com/simonbyrne/libmcr, which we modified to force the macro `__LITTLE_ENDIAN` on x86-64 under Linux. Libmcr only supports rounding to nearest. For $\sin(x)$, we get 314 failures with the CORE-MATH test suite; for example, for $x =$ `0x1.17fa49f5bf327p+41`, we get `0x1.adfaf34d5560ep-6` instead of `0x1.adfaf34d5560fp-6` (similar error for $-x$). This is the smallest x in absolute value for which we found a failure. For $\cos(x)$, we get 232 failures; for example, for $x =$ `0x1.567fb180b78bap+42`, we get `0x1.94ca60f882064p-5` instead of `0x1.94ca60f882065p-5`. For $\tan(x)$, we get 810 failures; for example, for $x =$ `0x1.42ec31a947798p+42`, we get `0x1.bdcdf4988244fp-5` instead of `0x1.bdcdf4988245p-5`. All these failures for these three functions occur on inputs for which the unrounded result has at least 43 identical bits after the round bit.

`CRLIBM`. There is no more any official repository for CRLIBM. We used the unofficial repository at https://github.com/taschini/crlibm. CRLIBM supports all rounding modes, with a different function for each rounding mode: `sin_rn` for rounding $\sin(x)$ to nearest, `sin_rz` for rounding toward zero, `sin_ru` for rounding up, and `sin_rd` for rounding down. The CRLIBM documentation says we have to call `crlibm_init()` before any call to the CRLIBM functions. This does two things: (a) it sets the rounding precision to double instead of double-extended which is the default under Linux for the x87 co-processor; (b) it sets the rounding mode to nearest-even. On modern 64-bit processors, (a) is no longer needed because double arithmetic is done using SSE2. And for (b), it seems `crlibm_init()` only touches the x87 co-processor, which is not used. In fact, the rounding mode from `fenv.h` has to be reset to nearest-even before any call to a CRLIBM function like `sin_rz`. After doing that, all tests do pass with sin, cos, tan, with all four C23 rounding modes.

`CORE-MATH`. All hard-to-round inputs of sin, cos, tan pass with CORE-MATH, with all four C23 rounding modes, and with and without FMA contraction. The only change that was done by the CORE-MATH developers is the following: there was an extra rounding test after the accurate path, which gave an error if correct rounding could not be decided. Since all hard-to-round inputs are correctly rounded, this extra test could be removed.

`LLVM libc`. As for CORE-MATH, all hard-to-round inputs of sin, cos, tan pass with LLVM libc, with all four C23 rounding modes.

Except Libmcr, it is astonishing that among all libraries which claimed correct rounding before hard-to-round cases were known (MathLib, CRLIBM, CORE-MATH, LLVM libc), all are indeed correctly rounded on these numbers, for all three functions (sin, cos, tan), and all supported rounding modes.

[1]https://www.vinc17.net/research/testlibm/bug153548.en.html