ERC Advanced Grant
Research Proposal (Part B2)

CORE-MATH: Ensuring Correctly Rounded Mathematical Functions

Principal Investigator (PI):	**Dr Paul Zimmermann**
PI's host institution:	**Inria, France**
Proposal full title:	**Ensuring Correctly Rounded Mathematical Functions**
Proposal short name:	**CORE-MATH**
Project duration:	**60 months**
Targeted Review Panel:	**PE6 (Computer Science and Informatics)**

In 1985, the IEEE 754 standard defined for the first time what the result of a computation on floating-point numbers should be. Today, any program using floating-point additions, subtractions, multiplications and divisions yields bit-to-bit identical results, whatever the hardware, compiler, or operating system. This is because IEEE 754 requires the best possible result for these operations, called **correct rounding**.

However, most scientific or industrial applications, like the Large Hadron Collider software, developed by thousands of physicists and engineers over two decades, or Karplus' equation $J(\phi) = A\cos^2\phi + B\cos\phi + C$ in nuclear magnetic resonance spectroscopy, also require the evaluation of various **mathematical functions**: sine, exponential, logarithm, etc. These functions are provided by a **mathematical library**, which does not always provide correct rounding. As a resulting effect, a program using such mathematical functions might yield **wrong results**, which could have disastrous consequences [25]. Moreover, these results might **differ** depending on the mathematical library, hardware, compiler or operating system.

We strongly believe it is the right time to fix that numerical reproducibility issue **once and for all**. CORE-MATH will provide new numerical algorithms to evaluate mathematical functions, which will always yield **correct rounding** (i.e., the best possible result) with speed comparable to the best libraries currently available. This will require **clever algorithms** to identify the most difficult cases for correct rounding, and **innovative research** in the field of the evaluation of mathematical functions.

Thanks to CORE-MATH, scientists, engineers, researchers will obtain **correct results** and thus **bit-to-bit reproducible results** for their numerical computations, with the same efficiency as with currently available mathematical libraries, or even more.

Section A describes the state-of-the-art and the main objectives of CORE-MATH. To reach these objectives, Section B details the methodology and organization of the three Research Tracks and of the Validation Track.

A State-of-the-Art and Objectives

A.1 Introduction

Before describing the state-of-the art, let us introduce the CORE-MATH scientific context. To compute with real numbers, two main schemes exist: the RealRAM model, and floating-point numbers. The RealRAM model uses as much memory as needed to represent exactly real numbers, in consequence it is time and memory expensive, thus unusable for large applications. Floating-point numbers use a fixed amount of memory, and have been standardized through IEEE 754. For efficiency, IEEE 754 defines some fixed-precision formats (single, double, and

quadruple precision). If arbitrary precision is needed, the GNU MPFR library is the current reference implementation.

Binary Floating-Point: the IEEE 754 Standard and Beyond. The IEEE 754 standard defines how binary and decimal floating-point arithmetic should be performed. Published in 1985, IEEE 754 was revised in 2008 and 2019 [11]. Scientific applications mainly use the binary formats, while the decimal formats are better suited for applications in finance. The standard defines three main binary formats for numerical computations: `binary32`, `binary64`, and `binary128`, with significands of 24 bits, 53 bits and 113 bits respectively. In this document, we focus on binary formats only:

IEEE 754 format	precision (bits)	$\|x\|_{\min}$	$\|x\|_{\max}$
`binary32` (single precision)	24	$1.4 \cdot 10^{-45}$	$3.4 \cdot 10^{38}$
`binary64` (double precision)	53	$4.9 \cdot 10^{-324}$	$1.8 \cdot 10^{308}$
`binary128` (quadruple precision)	113	$6.5 \cdot 10^{-4966}$	$1.2 \cdot 10^{4932}$

IEEE 754 requires *correct rounding* for the four arithmetic operations (addition, subtraction, multiplication, division), the fused multiply-add $\mathrm{FMA}(x, y, z) = xy + z$, and the square root. This means that for a given operation, say $x + t$, the implementation shall return the floating-point number y closest to the exact result according to the given rounding mode (to nearest, toward $-\infty$, toward zero, toward $+\infty$). Therefore, there is a *unique* possible answer y, which is called the *correct rounding* of $x+t$. For a mathematical function, e.g., exp, the correct rounding is the floating-point number y closest to the (infinite precision) exact value of $\exp(x)$. IEEE 754 *only recommends* (unlike for basic arithmetic operations) a set of correctly rounded mathematical functions.[1] Currently available mathematical libraries for the IEEE binary formats (for example GNU libc) do not provide correct rounding, and thus do not conform to IEEE 754 (see for example [27] for single precision).

Hardware and Software Support. Most current processors perform basic arithmetic operations $(+, -, \times, \div, \mathrm{FMA})$ in hardware (usually as micro-code for the division) for single and double precision (`binary32` and `binary64`), but not for quadruple precision, except the IBM Power9 processor, which also implements them in hardware. For quadruple precision, most compilers provide a data type (`__float128` in GCC) with basic arithmetic operations.

Mathematical functions are implemented in software, by a *mathematical library*. Current mathematical libraries do not guarantee correct rounding. Discrepancies from correct rounding range from one ulp (unit in the last place) to hundreds of thousands ulps, even in single precision [27]. For quadruple precision, mathematical functions are available for the `__float128` type since 2011 through GNU libc and/or the `libquadmath` library[2], which originates from the FDLIBM library developed by Sun Microsystems around 1993.

GNU MPFR. GNU MPFR (MPFR for short) is a C library performing arbitrary precision floating-point computations with correct rounding [5]. It thus extends IEEE 754 to arbitrary precision, with the main difference that correct rounding is guaranteed in MPFR not only for basic arithmetic operations, but also for all mathematical functions it implements. The development of MPFR has started in 1999, and it is continuously improved, mainly by the PI and Vincent Lefèvre. It is now a very mature library, which is available on all major operating systems (GNU/Linux, BSD, Windows, AIX, Solaris, MacOS), and is required to compile GCC and Gfortran. The original idea to create MPFR was due to the PI.

[1] Together with Jean-Michel Muller, the PI already argued in 2005 for correct rounding of mathematical functions during the first revision of IEEE 754 [26].

[2] The `libquadmath` code is automatically extracted from GNU libc.

A.2 State-of-the-Art

This section reviews the state-of-the-art of research relevant to CORE-MATH, for the search of Hard-to-Round cases (§A.2.1), and the numerical evaluation of mathematical functions (§A.2.2).

A.2.1 Search for HR-cases.

The search for HR-cases (Hard-to-Round cases) is a crucial step to design efficient algorithms that guarantee correct rounding. Indeed, in Ziv's onion-peeling strategy (which is explained in §A.2.2 below), the last step should always return the correct rounding, and the knowledge of HR-cases determines the minimum accuracy of this last step, i.e., its efficiency. Thus, apart from exact cases, one needs to determine the smallest distance between $f(x)$ and a floating-number in the target precision p (or $p+1$ for rounding to nearest). This is known as the Table Maker's Dilemma [18]. For some algebraic functions, the HR-cases are known [12], but in general one has to resort to exhaustive search to find them. The first non-trivial algorithm for this task is due to Lefèvre, and the best algorithm currently known is the SLZ algorithm.

Lefèvre's Algorithm. Lefèvre's algorithm [14] was the first non-trivial algorithm to search HR-cases of mathematical functions. It is based on the "three-distance theorem" in a circle. This algorithm uses a first-order approximation of the function f: with target precision p, it splits the interval to check into subranges of roughly $p^{1/3}$ consecutive floating-point values if f is smooth enough, thus giving a complexity of roughly $p^{2/3}$ to check a whole *binade*[3].

The SLZ Algorithm. SLZ [21, 22] is a clever algorithm using modern lattice reduction techniques (due to Coppersmith) to find HR-cases of mathematical functions. It is in fact a family of algorithms, with parameters the degree d of the approximation polynomial of the function, and another parameter called α, the case $d = \alpha = 1$ corresponding to Lefèvre's algorithm. Let f be a mathematical function, p the target precision in bits, $N = 2^p$, M an integer, and assume one searches all integers $|t| \le T$ such that

$$|Nf(t/N) \bmod 1| < 1/M, \tag{1}$$

where $u \bmod v$ denotes a centered modulus with value in $[-v/2, v/2]$. In a nutshell, SLZ first computes a degree-d approximation polynomial $P(t)$ of $Nf(t/N)$, then constructs polynomials $Q_{i,j} = M^{\alpha-j}(T\tau)^i(P(\tau)+y)^j$, reduces a matrix made from the $Q_{i,j}$ using the Lenstra-Lenstra-Lovász (LLL) algorithm, and if two reduced polynomials $q_1(\tau, y)$ and $q_2(\tau, y)$ are found with small enough coefficients, then for any integer t satisfying Eq. (1), $\tau = t/T$ will be a root of the resultant $\mathrm{Res}_y(q_1, q_2)$, which has integer coefficients.

For univariate functions, the asymptotic complexity of the SLZ algorithm, when the parameter α goes to infinity, is $N^{4/7}$ for $d = 2$, and $N^{1/\sqrt{d+1}}$ for $d \ge 3$ [20]. This yields $N^{0.5}$ for $d = 3$, $N^{0.45}$ for $d = 4$. When the degree d increases, SLZ allows one to consider larger ranges T, but the size of the matrix becomes larger, and the LLL reduction becomes more expensive. A reference implementation of SLZ is available in the BaCSeL software tool, written by Hanrot, Lefèvre, Stehlé and the PI. In practice, the SLZ algorithm starts to outperforms Lefèvre's algorithm for double precision, and gives a large speedup for quadruple precision (see Table 1).

In [20, Section 1.6], Stehlé extended SLZ to bivariate functions: with parameters $d = \alpha = 2$, the asymptotic complexity would be $N^{10/7} \sim N^{1.429}$, where N is the number of possible inputs for each variable. For single precision ($N = 2^{32}$), this corresponds to a complexity of about 2^{46}, and 2^{91} for double precision. However, these algorithms for bivariate functions have not been implemented and used for a real search, thus these figures should be handled with care.

[3] A binade is the set of all binary floating-point numbers between two consecutive powers of two.

format	N	M	T	est. time
`binary64`	2^{53}	2^{53}	2^{20}	1.1 days
`binary128`	2^{113}	2^{113}	2^{44}	420 Myears

Table 1: Best parameters for the SLZ algorithm and estimated time to check a binade of $N/2$ values for the 2^x function, using the BaCSeL tool on an Intel i5-4590 at 3.3 GHz (using one core).

A.2.2 Numerical Evaluation of Mathematical Functions

Once the HR-cases are known for a given function, it is possible to design an efficient correct rounding algorithm. The main ingredients are Ziv's strategy, argument reduction and reconstruction, and efficient polynomial evaluation.

Ziv's strategy. Ziv's strategy (also called onion-peeling strategy) consists in evaluating approximations of $f(x)$ with increasing working precisions $p_1 < p_2 < \cdots$ (see [30]). At step i with precision p_i, one gets an approximation y_i with an error bound ε_i: $f(x) \in [y_i - \varepsilon_i, y_i + \varepsilon_i]$. If both $y_i - \varepsilon_i$ and $y_i + \varepsilon_i$ round to the same number y in the target precision, then by monotonicity of the rounding function, y is the correct rounding of $f(x)$. Otherwise, one continues with a larger precision p_{i+1}.

Ziv's strategy will loop if $f(x)$ is exactly representable in the target precision p ($p+1$ for rounding to nearest). It is thus mandatory to know these "exact" cases and be able to efficiently detect them. Fortunately for most functions the exact cases are quite rare, for example for the exponential function there is only $e^0 = 1$. A tricky case is the power function x^y, which admits plenty of exact cases, for example $x = 625$ and $y = 3/4$ yield $x^y = 125$.

If the HR-cases are not known, the only way to guarantee correct rounding is to implement Ziv's strategy with an unbounded number of steps, thus with unbounded working precision p_i. However, embarking an arithmetic with unbounded precision would be highly inefficient. This is why HR-cases are needed, or at least a tight bound for the maximal required precision. Ziv's original implementation uses a 3-step strategy for `binary64`: a first step using double precision, a second step using double-double arithmetic, and a final one using 32 digits in base 2^{24}, thus a total of 768 bits. For `binary64`, 768 bits is very likely large enough to guarantee correct rounding, but no proof was given by Ziv.

For a fixed target precision like in the CORE-MATH objectives, a close to optimal strategy is to use two precisions: a *fast path* with precision p_1 that will be able to return a correctly rounded value in almost all cases, and an *accurate path* with precision p_2 large enough to always return the correctly rounded value, as detailed in Figure 1. If t_1 (resp. t_2) is the average time

fast path	call a routine f_1, using a working precision $p_1 > p$, yielding an approximation y_1 such that $\lvert y_1 - f(x)\rvert < \varepsilon_1$;
rounding test	if $\mathrm{round}_p(y_1 - \varepsilon_1) = \mathrm{round}_p(y_1 + \varepsilon_1)$, return that number;
accurate path	otherwise call a routine f_2, using a working precision $p_2 > p_1$, yielding an approximation y_2, and return $\mathrm{round}_p(y_2)$.

Figure 1: The correct rounding algorithm for a univariate function $f(x)$ and target precision p.

of f_1 (resp. f_2), and ξ the probability that f_1 is not able to round correctly, the average time is:

$$t = t_1 + \xi t_2. \tag{2}$$

Once the HR-cases are known, we know the minimal precision p_2 required for f_2, and we can design such a function with the smallest t_2. Then different implementations of f_1 can be

tried, with different values of t_1 and ξ, keeping the one giving the smallest average time t in Eq. (2). This approach has been successfully used by Godunov for the `binary64` exponential function [7].

Argument Reduction and Reconstruction. When a function $f(x)$ satisfies some mathematical properties, for example $\sin(x + 2\pi) = \sin(x)$, one can use these properties to reduce the evaluation to a small interval, usually around zero or one. This technique is called *argument reduction*. One distinguishes between *additive argument reduction*, like in $\sin(x + 2\pi) = \sin(x)$, and *multiplicative argument reduction*, like in $\log(2x) = \log(x) + \log(2)$. The typical workflow is thus the following: (i) reduce the input x to a reduced argument x', (ii) approximate $f(x')$, and (iii) recover $f(x)$ from $f(x')$. Step (iii) is called *argument reconstruction*, it can be trivial like in $\sin(x + 2\pi) = \sin(x)$.

Algorithms and Arithmetic. For the argument reduction/reconstruction and for the approximation of $f(x')$ (see above), different algorithms and arithmetic implementations are possible.

Once the argument has been reduced to a small interval $x' \in [a, b]$, a good polynomial approximation of the function f over $[a, b]$ is chosen. The state-of-the-art tool to choose this polynomial is the Sollya program [4], which in addition allows one to give constraints on the polynomial coefficients, so that they fit into the desired format. The range $[a, b]$ can also be split further into smaller sub-intervals, on which a polynomial of smaller degree can be used. The "middle" point of each sub-interval can be chosen according to Gal's "accurate table method", so that the constant coefficient of the polynomial yields some extra accuracy [6]. Very recently, Gustafson proposed a completely new approach [9], which however seems to be usable only for single precision.

Finally, the choice of the arithmetic implementation is crucial. Here three cases are distinguished according to the target precision. If the target precision is `binary32`, one can use `binary64` for the working precision of the fast path routine: it will yield $53 - 24 = 29$ extra bits of accuracy, and `binary64` is very efficient since implemented in hardware. For `binary64` target precision, one could use double extended variables, with a significand of 64 bits, but this format is available on some processors only. A better solution is to use a 64-bit integer type [13]. This approach has also demonstrated its efficiency for `binary128`, with multi-word integer types [29], where a speedup of more than 10 was obtained for the quadruple precision exponential function. Table 2 shows the maximal error in units in last place for some mathematical libraries in single precision, and the number of computer cycles needed by the reference GNU libc implementation (it is free, widely available, highly optimized, well tested, and contains benchmark utilities to measure average latency).

Summary. The main weakness of IEEE 754 is that correctly rounded mathematical functions are not mandatory. This has too major consequences: different mathematical libraries might give inaccurate results (and indeed they do [27]), and scientific computations are not bit-to-bit reproducible, as soon as they involve mathematical functions. When IEEE 754 was first published in 1985, it was too early to standardize mathematical functions. Nowadays, a lot of progress has been made by several researchers in the field, particularly by the PI and his co-authors; however, the issue of correctly rounded mathematical functions is far from being solved, since major algorithmic obstructions remain.

A.3 Grand Challenge and Scientific Objectives

It would be rather easy to provide correct rounding with an average factor of two slowdown with respect to current mathematical libraries (which do not yield correct rounding). However,

	GNU libc	Intel Math Library	AMD libm	Newlib	OpenLibm	Musl
asin	0.898	0.528	0.861	0.926	0.743	0.743
exp2	0.502	0.519	1.00	1.02	0.501	0.502
log2	0.752	0.508	0.586	1.65	0.865	0.752
sqrt	**0.500**	**0.500**	**0.500**	**0.500**	**0.500**	**0.500**

function	binary32	binary64	binary128
sin	71/79	306/299	3060/3361
exp	47/43	45/46	3546/3342
pow	82/76	115/108	9412/9027

Table 2: Top: maximal error in units in last place for some mathematical libraries in single precision [27]. Bottom: latency (in clock cycles) for some GNU libc 2.31 functions on an Intel Core i7-8750H (left) and an AMD Ryzen 5-2400G (right), for random inputs in $[-10, 10]$ for exp, in $[0, 10]^2$ for pow, and in $[2^{e-1}, 2^e]$ for sin, with $e = 128, 1024, 16384$ for `binary32`, `binary64` and `binary128` respectively.

to definitively convince users and members of the next IEEE 754 revision committee to adopt correctly rounded mathematical functions, we strongly believe in the following Grand Challenge:

Design correct rounding algorithms for mathematical functions, and corresponding IEEE 754 conforming implementations for single, double, and quadruple precision, with better efficiency than current non-conforming mathematical libraries.

Said otherwise, our Grand Challenge is to have all entries **0.500** in the top part of Table 2, which is the optimal maximal error in terms of units in last place for rounding to nearest, like in the sqrt row, while having better timings than in the bottom part of Table 2.

The scientific challenges to be solved depending on the format (single, double, quadruple), the Grand Challenge naturally splits into three Research Tracks: RT-1 for single precision (§A.3.1), RT-2 for double precision (§A.3.2), and RT-3 for quadruple precision (§A.3.3).

For each Research Track, some hard scientific challenges are identified, and corresponding success criteria are given. The Validation Track will ensure these scientific results will be made available to the scientific and research community.

A.3.1 Research Track 1: Single Precision

The IEEE 754 `binary32` format can represent numbers as small as $x_{\min} \approx 1.4 \cdot 10^{-45}$ (in absolute value), and as large as $x_{\max} \approx 3.4 \cdot 10^{38}$. This format being encoded on 32 bits, there are at most 2^{32} possible inputs. Searching all HR-cases for an univariate function is straightforward, even with a naive algorithm comparing each value to the one obtained with MPFR (which explains why nobody did bother publishing them). However, bivariate functions, for example the power function x^y, the hypot function $\sqrt{x^2 + y^2}$, or the atan2 function $\arctan(y/x)$, remain out of reach for an exhaustive HR-case search. The objective of Research Track 1 is to solve the Table Maker's Dilemma for these bivariate functions, and to provide corresponding correct rounding algorithms:

- **Track RT1-a: Search HR-cases for `binary32` bivariate functions**
- **Track RT1-b: Design efficient correct rounding algorithms for `binary32` bivariate functions**

Criterion of Success for Research Track 1: new algorithms and a reference IEEE 754-conforming implementation for the power function in single precision within 50

cycles on average (compared to 76-82 cycles for the current non-conforming GNU libc implementation)[4].

A.3.2 Research Track 2: Double Precision

For univariate functions in double precision, many HR-cases are known, thanks to the work of Lefèvre and Muller [15, 17]. One exception is the case of periodic functions. For example, the HR-cases of $\sin(x)$ are known only for $|x| \leq 12867/4096 \approx 3.1413$, the worst case in that range being (where the right-hand side is in binary)

$$\sin(8980155785351021 \cdot 2^{-54}) = 0.011110100110010101000001110011000011000100011010010101\underbrace{111...111}_{66}000...$$

HR-cases for larger absolute values (up to the largest `binary64` number $x \approx 1.8 \cdot 10^{308}$) are still unknown. In [10], a new algorithm was proposed for periodic functions, with an estimate of about 4 core-years for the $[2^{1023}, 2^{1024}]$ binade. This gives about 4000 core-years to find all HR-cases of $\sin(x)$ for the whole `binary64` format. One objective of this research track is to find new algorithms that will reduce that search time by a factor of 10 to make it feasible:

- **Track RT2-a: Search HR-cases for `binary64` periodic functions**
- **Track RT2-b: Design efficient correct rounding algorithms for `binary64` periodic functions**

Criterion of Success for Research Track 2: new algorithms and a reference IEEE 754-conforming implementation for the sine function within 100 cycles on average for the whole double precision exponent range (compared to about 300 cycles for the current non-conforming GNU libc implementation).

A.3.3 Research Track 3: Quadruple Precision

Quadruple precision (`binary128`) is the wider IEEE 754 format, which can represent up to 2^{128} different values, ranging from $6.5 \cdot 10^{-4966}$ to $1.2 \cdot 10^{4932}$ in absolute value. With the current state-of-the-art algorithms, it would take of the order of 420M core-years to find HR-cases for one binade (see Table 1), and the `binary128` format corresponds to about 2^{15} binades. No HR-cases are known (except for the square root [12]). CORE-MATH will provide major breakthroughs in two directions:

- **Track RT3-a: Search HR-cases for `binary128` functions**
- **Track RT3-b: Design efficient correct rounding algorithms for `binary128` functions**

For the design of efficient algorithms, an important difference with single and double precision is that basic arithmetic for quadruple precision (addition, subtraction, multiplication, division) is usually implemented in software, and it thus slow (with the already-mentioned exception of the IBM Power9 processor, see §A.1).

Criterion of Success for Research Track 3: new algorithms and a reference IEEE 754-conforming implementation for the exponential function in quadruple precision within 200 cycles on average (compared to 3300-3500 cycles for the current non-conforming GNU libc implementation).

[4]While CORE-MATH will target **all functions** of Annex F from the C language standard, within each research track we identify hard problems that remain currently unsolved, and give corresponding success criteria.

A.3.4 Validation Track

Using the results of RT-1, RT-2, and RT-3, the Validation Track will provide correct rounding implementations for the `binary32`, `binary64`, and `binary128` formats respectively, for all rounding modes. It will address all 28 functions of Annex F of the C language standard: trigonometric functions (acos, asin, atan, atan2, cos, sin, tan), hyperbolic functions (acosh, asinh, atanh, cosh, sinh, tanh), exponential and logarithmic functions (exp, exp2, expm1, log, log10, log1p, log2), power-like functions (cbrt, hypot, pow, sqrt), error and gamma functions (erf, erfc, lgamma, tgamma), together with the new functions planned in the C2X standard [3].

Criterion of Success for the Validation Track: ensure the correct rounding functions designed within CORE-MATH are integrated into at least one of the current mathematical libraries: GNU libc, Intel Math Library, etc.

B Methodology

We explain now how we will achieve our Grand Challenge, and which research plan we will set up for each research track. Some methodology of CORE-MATH is common to all three research tracks, in particular the dependency to the IEEE 754 rounding modes. The correct rounding algorithms designed within CORE-MATH will depend on the rounding mode in the very last step only: first the inputs will be converted to some internal representation, then all computations will be done within that internal representation (where the current rounding mode will have no effect), and the final approximation will be correctly rounded according to the current rounding mode.

For each research track, we detail the CORE-MATH methodology on the example function given in the corresponding criterion of success. As explained above, the other functions of Annex F of the C standard (see §A.3.4) will be considered too, but each example function represents well the main difficulties that will be encountered.

B.1 Research Track 1: Single Precision

Track RT1-a: Search HR-cases for `binary32` bivariate functions

Let us detail the CORE-MATH methodology on the power function x^y. A good news is that not all 2^{64} pairs of inputs yield a result in the `binary32` exponent range. Indeed, for $x = 17$ and $y = 42$, x^y overflows, thus there is no need to consider that pair for HR-cases. For the power function, the number of positive inputs such that x^y does not underflow or overflow is about 2^{61}. However, this number is still huge.

Some preliminary experiments with the bivariate SLZ algorithm in the SageMath computer algebra system [23] yield the following optimal settings for the x^y function, degree $d = 3$ and parameter $\alpha = 2$, where each call of the SLZ algorithm (see §A.2.1) deals with a rectangle of $2T$ consecutive floating-point values for x, and $2U$ for y:

format	T	U	estimated time
`binary32`	2^6	2^6	400 000 years
`binary64`	2^{14}	2^{14}	10^{17} years
`binary128`	2^{31}	2^{31}	10^{44} years

Apart from the fact that `binary64` and `binary128` are out of reach, the interesting figure is $T = U = 2^6$ for `binary32`. This means that each run of the algorithm deals with a square containing $2T = 128$ consecutive `binary32` x-values, and $2U = 128$ consecutive `binary32` y-values, thus a total of 16384 pairs (x, y). This corresponds to about 90 milliseconds per square

of 16384 pairs for the SageMath toy implementation. We propose the following alternate algorithm:

- compute an order-1 expansion $f(x,y) \approx a+bt+cu$ around x_0, y_0, with a, b, c floating-point values, and t, u integers, $|t| < T$, $|u| < U$, assuming $1/2 \le |f(x,y)| < 1$;
- deduce $a' = \mathrm{frac}(2^p a)$, $b' = \mathrm{frac}(2^p b)$, $c' = \mathrm{frac}(2^p c)$, corresponding to the least significant bits, with p the target precision;
- now one wants to find integers t, u such that $a' + b't + c'u \bmod 1$ is small, where a', b', c' are real numbers in $[0,1)$, and cmod denotes the centered modulus.

A classical approach for the last step is the following: let a'' be the integer closest to $2^{64}a'$, and similarly for b'' and c'', then one is looking for $a'' + b''t + c''u \bmod 2^{64}$ small, say less than some bound d in absolute value. Another classical approach (already used for example in Lefèvre's algorithm [14]) is to compute instead $a''+b''t+c''u+d \bmod 2^{64}$ (now with the classical modulus, giving a number in $[0, 2^{64}-1]$) and check whether it is smaller than $2d$. This could be done at the speed of one operation every clock cycle. On a 3 GHz processor, one should be able to check $3 \cdot 10^9$ `binary32` pairs (x,y) per second, and thus checking all the $\approx 2^{61}$ cases of x^y that do not yield underflow or overflow (see above) would take a few core-years, which becomes feasible.

Track RT1-b: Design correct rounding algorithms for `binary32` bivariate functions

The objective of this research track is to provide efficient correct rounding algorithms for mathematical functions in single precision, unlike current mathematical libraries [27]. The 28 functions of Annex F of the C language standard, detailed in §A.3.4, will be addressed.

The challenging functions will be the bivariate ones (atan2, pow, hypot), since on the one hand the HR-cases will be harder to compute (cf Track RT1-a), and on the other hand they are likely to require more accuracy for the accurate path. Indeed, for a format on p bits, the HR-cases for univariate functions are expected to require about $2p$ bits of accuracy, and those for bivariate functions are expected to require about $3p$ bits of accuracy, thus about 96 bits here. Instead of using Ziv's original strategy (see Figure 1), we propose to have an accurate path with a working precision of 64 bits, and to add a third "very accurate" path with a working precision sufficient to round correctly all HR-cases. This will require to implement an efficient arithmetic with about 96 bits of accuracy. Here the experience of the PI with MPFR will be extremely valuable [16].

B.2 Research Track 2: Double Precision

Track RT2-a: Search HR-cases for `binary64` periodic functions

Research Track RT2-a will first re-evaluate the estimate of 4000 core-years to find all HR-cases of $\sin(x)$ for `binary64` (see §A.3.2). A first research direction will be to investigate whether the algorithm from [10] can be parallelized. For the $[2^{1023}, 2^{1024}]$ binade, where two consecutive floating-point numbers are distant from $\mu = 2^{971}$, this algorithm considers arithmetic progressions of numbers distant of $q\mu$ from each other, where $q = 15\,106\,909\,301$ is chosen such that $q\mu$ is very small modulo 2π, here $q\mu \bmod (2\pi) \approx 4.41 \cdot 10^{-13}$. Then the original SLZ algorithm is used on each arithmetic progression (or Lefèvre's algorithm, which strangely was not even tried in [10]).

Within CORE-MATH, we will search for new ideas making obsolete the state-of-the-art algorithm from [10]. On the algorithmic side, one such idea to be experimented is the following. The algorithm from [10] deals independently with every binade. However, binades could be

grouped together, for example if we group the $[2^{1022}, 2^{1023}]$ and $[2^{1023}, 2^{1024}]$ binades together, we have to consider inputs of the form $m \cdot 2^{970}$ for an integer m in the range $[2^{52}, 2^{54}]$. Since the algorithm from [10] is sublinear in the input size N, one can expect a smaller asymptotic complexity.

In the worst-case scenario, assuming only a factor of 4 will be gained, it will decrease the cost of the search of HR-cases for $\sin(x)$ from 4000 years to about 1000 years. This is tractable with university-level resources (using if needed the Grid5000 platform [8] or the PRACE Research Infrastructure [19]). Thus Track RT2-a should be able to determine and publish HR-cases of $\sin(x)$ for the whole `binary64` format, as well as of the other considered functions.

Track RT2-b: Design correct rounding algorithms for `binary64` periodic functions

We will provide correctly-rounded algorithms for `binary64` using Ziv's strategy with two steps: a fast path using 64-bit integer arithmetic, and an accurate path using 128-bit integer arithmetic, assuming it is enough for the HR-cases obtained by Track RT2-a. Indeed, since the advent of 64-bit processors, the clever use of integer operations can be faster than using hardware floating-point operations to implement mathematical functions [13, 29].

Efficient arithmetic will also be needed for the argument reduction step. In the case of $\sin(x)$ for x large, one first needs to compute $k = \lfloor x/(2\pi) \rceil$, then compute the reduced argument $x' = x - 2k\pi$, before approximating $\sin(x')$. For the largest possible x, the integer k will have up to 1022 bits. We will also try the following research direction: since every `binary64` number can be written $x = m \cdot 2^e$ for integers m and e, the idea is to precompute $\tau \approx 2^e \bmod (2\pi)$, then $x' \approx m\tau \bmod (2\pi)$. Using statistical considerations, an accuracy of about 128 bits should be enough for τ, therefore the argument reduction will require a smaller arithmetic and be faster, at the expense of more memory to store the table of the precomputed τ values. Track RT2-b will compare all these strategies and keep the best one.

B.3 Research Track 3: Quadruple Precision

Track RT3-a: Search HR-cases for `binary128` functions

The current cost estimates for the search of `binary128` HR-cases are huge: 420M core-years for one binade of the 2^x function (Table 1). A first research direction will be to revisit this estimate, using algorithmic and implementation ideas, as already detailed in Track RT2-a. On the other side, since the parameters are larger, this opens more room for new ideas.

In case the revised estimate is still too large, it will not be possible to actually *compute* the HR-cases for the target function. We will then implement the fallback solution of determining an *upper bound* for the required working precision. Indeed, the cost of the SLZ algorithm decreases when the number of sought identical bits after the rounding bit increases, i.e., when the parameter M increases in Equation (1). In his PhD thesis, Torres has shown that with $M = 2^{10p}$, degree $d = 45$, and parameter $\alpha = 10$, the cost of checking one quadruple-precision binade for the exponential function decreases to 66 core-years [24, Section 3.9.6.2]. The search then becomes tractable, very likely it will find no HR-case, nevertheless it will prove that a working precision of $113 + 10 \cdot 113 = 1243$ bits will be enough, i.e., about 20 words of 64 bits. Another research direction will be to re-evaluate this estimate, and similarly for smaller values of M (2^{9p}, 2^{8p}, ...), in order to determine the smallest value of M for which the HR-cases search (or more precisely bounding the HR-cases) is feasible. Indeed, for such large values of M, most of the time is spent in the LLL reduction, and one will search for a special-purpose reduction algorithm along the lines of [1].

In any case, Track RT3-a will provide for every function a bound $p_{\max}$, certifying that no solution to Equation (1) exists for $M = 2^{p_{\max}}$. This bound will be used in Track RT3-b.

Track RT3-b: Design correct rounding algorithms for `binary128` functions

We will provide correctly-rounded algorithms for `binary128` using Ziv's strategy with two or three steps: a fast path using 128-bit integer arithmetic, a second path using 256-bit integer arithmetic, and if needed a third path using larger integer arithmetic, using the bound $p_{\max}$ provided by Track RT3-a.

A preliminary study performed by the PI has shown that using integer-only arithmetic to implement quadruple precision mathematical functions can yield a speedup of 27% over GNU libc on platforms which support `binary128` in hardware, and a factor of more than 10 on platforms without such hardware support [28]. These figures are very preliminary and are likely to be improved by CORE-MATH. This will be the main direction followed by Track RT3-b.

For each of the two or three steps of Ziv's strategy (128 bits, 256 bits, and up to about 1200 bits depending on the results of Track RT3-a), we will design efficient algorithms for the argument reduction and reconstruction, and the evaluation of the approximation polynomial itself. The `binary128` instances of these algorithms will be automatically generated using the Meta-MPFR generator (see below).

B.4 Validation Track

The Validation Track will take care of efficiently implementing the algorithms designed in RT-1, RT-2, and RT-3. For this purpose, a meta-generator of efficient code (called Meta-MPFR) will be designed and tuned for the scientific objectives of CORE-MATH. This will greatly help disseminate and integrate the scientific results of CORE-MATH.

Track VT-a: Meta-MPFR

Meta-MPFR will be meta-generator, written in a high-level language like Python. It will generate efficient code for the C language with rigorous error bounds, that will be used to efficiently implement the algorithms designed in RT1-b, RT2-b, RT3-b. Meta-MPFR will have two layers:

- a lower layer providing low-level functions, in particular addition, subtraction and multiplication;
- a higher layer providing high-level functions, for example argument reduction or reconstruction, evaluation of an approximation polynomial.

The higher layer will be interfaced with the Sollya tool [4] to automatically compute approximation polynomials. The programs generated by Meta-MPFR will manipulate fixed-precision floating-point numbers stored on several computer words, using only integer operations (as in MPFR). Meta-MPFR will take as input the target precision, the bit size of the target processor (32 or 64), and other parameters like the maximal absolute value that can arise during the computations, the hardware configuration of the target processor (for example presence of a fused multiply-add operation, size of caches). Note that for a given step (fast or accurate path), the working precision will be determined by one of Tracks RT1-a, RT2-a, or RT3-a, then will be the same for all routines needed for that step, and can thus be implicit (contrary to MPFR where each floating-point variable stores its own precision).

Track VT-b: Dissemination and Integration

Track VT-b will take care of the dissemination of the CORE-MATH results toward the scientific community, and its full integration into existing mathematical libraries. For each function of

Annex F from the C language standard, a complete implementation with correct rounding will be published for public review. To assess the correctness of these reference implementations, a "CORE-MATH bugs bounty program" will be launched, with amounts of 1024 euros (single precision bounty), 2048 euros (double precision bounty), and 4096 euros (quadruple precision bounty) for the first individual to find a case that is not correctly rounded[5]. Apart from attracting public media, this will provide an excellent review of the work done in CORE-MATH. These implementations will be integrated into at least one of the main mathematical libraries (GNU libc for example) and thus available for every engineer, scientist or researcher.

B.5 CORE-MATH Roadmap

Table 3 summarizes the distribution of the work over the 5 years of CORE-MATH, according to the dependencies and relationships between the different research tracks. The first priorities for the HR-cases tracks (RT1-a, RT2-a, RT3-a) will be to compute upper bounds for the precision of the corresponding accurate paths, that will be needed by tracks RT1-b, RT2-b, RT3-b. Since Meta-MPFR is independent from the other tracks, its development can start at the beginning of CORE-MATH.

	Year 1	Year 2	Year 3	Year 4	Year 5
RT1	RT1-a (`binary32` HR-cases)				
RT1		RT1-b (`binary32` correct rounding)			
RT2		RT2-a (`binary64` HR-cases)			
RT2			RT2-b (`binary64` correct rounding)		
RT3			RT3-a (`binary128` HR-cases)		
RT3				RT3-b (`binary128` correct rounding)	
VT	VT-a (Meta-MPFR)		VT-b (dissemination and integration)		
		Workshop 1		Workshop 2	

Table 3: The CORE-MATH Roadmap (timeline in years).

Three PhD students will be hired to work on the HR-cases search (research tracks RT1-a, RT2-a, RT3-a), while three postdoctoral researchers will be hired to work on the efficient correct rounding algorithms (research tracks RT1-b, RT2-b, RT3-b). A confirmed researcher with 4-8 years of experience will be hired to work on the Validation Track.

B.6 High Risk, High Gain

The outcome of CORE-MATH will be new algorithms providing correct rounding for the three binary IEEE 754 formats, and the corresponding reference implementations. This will only be possible if we manage to do major algorithmic breakthroughs in the search for HR-cases, and in the accurate evaluation of mathematical functions.

High Risk. For Research Track 1, we are confident we will be able to determine the hard-to-round cases for x^y in the `binary32` format, by inventing an algorithm similar to SLZ for bivariate functions, if needed with the help of parallel computations.

For Research Track 2, saving a factor of 10 over the state-of-the-art search for HR-cases [10] entails a high risk. If we only save a smaller factor, for example 3, the total time will still be reachable using distributed computations, for which the PI has a very solid experience [2].

[5]The corresponding amounts will be paid by the Host Institution (Inria).

The risk for Research Track 3 is very high. Indeed, the current estimation of 420 Myears for the HR-cases search is huge (for just one binade), and here a factor of about one million should be saved to make it feasible. If the algorithmic and implementation improvements are not sufficient, another research direction would be to add a second rounding test in Figure 1 after the call to f_2, to check whether $\mathrm{round}_p(y_2-\varepsilon_2) = \mathrm{round}_p(y_2+\varepsilon_2)$, where ε_2 is the maximal error for y_2. If that is not the case, a third function f_3 will be called with a larger precision $p_3 > p_2$. Since the cost of determining HR-cases with the SLZ algorithm decreases with the precision, p_3 will be chosen such that this search becomes possible.

The Validation Track will consolidate the results obtained by Research Tracks 1-3. The main risk for this track is that the output of CORE-MATH will not be adopted by the scientific community. To mitigate this risk, contributions will be made to the main mathematical libraries used in scientific applications very early during CORE-MATH. (In addition, the PI is member of the IEEE 754 discussion list since 2001, and of the C Floating-Point group since early 2020.)

High Gain. Computer science achievements made IEEE 754 the most famous and successful industrial standard. However, it did not settle the case of correct rounding for mathematical functions, which has produced many incorrect results since 1985, and is still preventing bit-to-bit reproducibility of numerical computations. CORE-MATH will push the next revision of IEEE 754 to require correct rounding for mathematical functions. The timeline is perfect, since the next revision is due in 2029. CORE-MATH will open new possibilities for engineers and scientists from all domains. Firstly, scientific applications will yield the *best possible result* and become *bit-to-bit reproducible*, across hardware processors, compilers, operating systems. Secondly, since the roundoff error for every mathematical function will be bounded, it will become possible to compute rigorous error bounds for a whole computation. In particular, it will be possible to perform rigorous interval arithmetic with mathematical functions, and thus obtain a correct containment interval for the result of a whole computation. Last but not least, CORE-MATH will provide new algorithms for quadruple precision which will not only yield correct rounding, but also provide more than a tenfold speedup with respect to the best publicly available libraries. This will make quadruple precision really accessible for applications requiring it. In summary, CORE-MATH will allow to compute just right, and still fast!

The economic impact of CORE-MATH will be multiple. On the one hand, the cost of developing numerical applications will decrease, since it will no longer be required to test them on every different combination of hardware, compiler, operating system (or worse to tweak them so that the test suites run) and we will get for free *forward reproducibility*, i.e., a program written at year Y will still yield the same result at year $Y + 10$. This is not the case currently, since any tiny change in the mathematical library (either improving or degrading the accuracy) might change the final result. We also expect it will enable to join or share the development efforts of the different mathematical libraries currently available. Finally, *vendor lock-in* will no longer be possible, where the library designed by a vendor calls non-optimal routines on hardware from a different vendor.

B.7 Conclusion and Perspectives

As for the perspectives, one expectation is that CORE-MATH will motivate other researchers and numerical analysts to promote correct rounding for other domains of computation or other operations. For example, despite computations with (floating-point) complex numbers are standardized in the C language, there is currently no requirement for correct rounding at all, even when multiplying or dividing two complex numbers!

References

[1] Bi, J., Coron, J., Faugère, J., Nguyen, P. Q., Renault, G., and Zeitoun, R. Rounding and chaining LLL: finding faster small roots of univariate polynomial congruences. In *Public-Key Cryptography - PKC 2014 - 17th International Conference on Practice and Theory in Public-Key Cryptography, Buenos Aires, Argentina, March 26-28, 2014. Proceedings* (2014), H. Krawczyk, Ed., vol. 8383 of *Lecture Notes in Computer Science*, Springer, pp. 185–202.

[2] Boudot, F., Gaudry, P., Guillevic, A., Heninger, N., Thomé, E., and Zimmermann, P. Comparing the difficulty of factorization and discrete logarithm: a 240-digit experiment. In *Proceedings of Advances in Cryptology (CRYPTO)* (2020), D. Micciancio and T. Ristenpart, Eds., vol. 12171 of *Lecture Notes in Computer Science*, pp. 62–91.

[3] C Floating-point Group. IEC 60559 math functions for C2X. `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2373.pdf`, 2019.

[4] Chevillard, S., Joldes, M. M., and Lauter, C. Sollya: an environment for the development of numerical codes. In *Third International Congress on Mathematical Software - ICMS 2010* (Kobe, Japan, 2010), K. Fukuda, J. van der Hoeven, M. Joswig, and N. Takayama, Eds., vol. 6327 of *Lecture Notes in Computer Science*, Springer, pp. 28 – 31.

[5] 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).

[6] Gal, S. 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* (1985), W. L. Miranker and R. A. Toupin, Eds., vol. 235 of *Lecture Notes in Computer Science*, Springer, pp. 1–16.

[7] Godunov, A. Algorithms for calculating correctly rounded exponential function in double-precision arithmetic. *IEEE Transactions on Computers 69*, 9 (2020), 1388–1400.

[8] The Grid'5000 testbed for parallel and distributed computing. `https://www.grid5000.fr`.

[9] Gustafson, J. L. The Minefield method: A uniformly fast solution to the Table-Maker's Dilemma. `https://bit.ly/2ZP4kHj`, 2020.

[10] 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)* (Montpellier, France, 2007), P. Kornerup and J.-M. Muller, Eds., IEEE Computer Society Press, Los Alamitos, CA, pp. 133–140.

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

[12] Lang, T., and Muller, J.-M. Bounds on runs of zeros and ones for algebraic functions. In *Proceedings of the 15th IEEE Symposium on Computer Arithmetic* (2001), IEEE Computer Society, pp. 13–20.

[13] Le Maire, J., Brunie, N., de Dinechin, F., and Muller, J.-M. Computing floating-point logarithms with fixed-point operations. In *23rd IEEE Symposium on Computer Arithmetic* (Santa Clara, United States, 2016), P. Montuschi, M. J. Schulte, J. Hormigo, S. F. Oberman, and N. Revol, Eds., IEEE, pp. 156–163.

[14] Lefèvre, V. New Results on the Distance Between a Segment and Z^2. Application to the Exact Rounding. In *17th IEEE Symposium on Computer Arithmetic - Arith'17* (Cape Cod, MA, United States, 2005), P. Montuschi and E. Schwarz, Eds., IEEE Computer Society, pp. 68–75.

[15] Lefèvre, V., and Muller, J.-M. Worst Cases for Correct Rounding of the Elementary Functions in Double Precision. In *15th IEEE Symposium on Computer Arithmetic - ARITH 2001* (Vail, Colorado, 2001), N. Burgess and L. Ciminiera, Eds., pp. 111–118.

[16] Lefèvre, V., and Zimmermann, P. Optimized binary64 and binary128 arithmetic with GNU MPFR. In *24th IEEE Symposium on Computer Arithmetic, ARITH 2017, London, United Kingdom, July 24-26, 2017* (2017), N. Burgess, J. D. Bruguera, and F. de Dinechin, Eds., pp. 18–26.

[17] Lefèvre, V. Hardest-to-round cases – part 2. `http://tamadiwiki.ens-lyon.fr/tamadiwiki/images/c/c1/Lefevre2013.pdf`, 2013. Slides presented at the final TaMaDi meeting. 30 pages.

[18] Lefèvre, V., Muller, J.-M., and Tisserand, A. Toward correctly rounded transcendentals. *IEEE Transactions on Computers 47* (1998), 1235 – 1243.

[19] Partnership for Advance Computing in Europe. `https://prace-ri.eu`.

[20] Stehlé, D. *Algorithmique de la réduction de réseaux et application à la recherche de pires cas pour l'arrondi de fonctions mathématiques.* Theses, Université Henri Poincaré - Nancy I, 2005.

[21] Stehlé, D. On the Randomness of Bits Generated by Sufficiently Smooth Functions. In *Seventh Algorithmic Number Theory Symposium - ANTS 2006* (Berlin, Germany, 2006), F. Hess, S. Pauli, and M. Pohst, Eds., vol. 4076 of *Lecture Notes in Computer Science*, Springer, pp. 257–274.

[22] 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.

[23] The Sage Developers. *SageMath, the Sage Mathematics Software System (Version 9.1)*, 2020. `https://www.sagemath.org`.

[24] Torres, S. *Tools for the Design of Reliable and Efficient Functions Evaluation Libraries.* PhD thesis, Université de Lyon, 2016.

[25] Vuik, K. Some disasters caused by numerical errors. `http://ta.twi.tudelft.nl/users/vuik/wi211/disasters.html`.

[26] Zimmermann, P. Why transcendentals and arbitrary precision? Invited talk at the IEEE 754 revision committee, Sun Menlo Park, 2005.

[27] Zimmermann, P. Accuracy of mathematical functions in single precision. `https://members.loria.fr/PZimmermann/papers/accuracy.pdf`, 2020.

[28] Zimmermann, P. Faster expf128. `https://sourceware.org/pipermail/libc-alpha/2020-June/115229.html`, 2020.

[29] Zimmermann, P. How slow is quadruple precision? Invited talk at the ICERM workshop on Variable Precision in Mathematical and Scientific Computing, Providence, 2020. `https://icerm.brown.edu/events/htw-20-vp/`.

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