# A new open-source SIMD vector libm fully implemented with high-level scalar C

Christoph Lauter

# A new Open-Source SIMD Vector Libm Fully Implemented With High-Level Scalar C

Christoph Lauter
Sorbonne Universités, UPMC Univ Paris 06,
UMR 7606, LIP6, F-75005, Paris, France
Email: christoph.lauter@lip6.fr

***Abstract*—Systems support mathematical functions like $\exp$, $\sin$, $\cos$ through mathematical libraries (libm). With increasingly parallel hardware, scalar libm functions do not suffice; implementations that work on vectors in an element-by-element (SIMD) fashion are required. Only few Open-Source implementations of vector libms exist. They are mostly written in assembly, which hinders portability and maintenance. As they depend on the scalar system libm for special case handling, the existing vector libms may induce a source of non-reproducibility.**

**We present an Open-Source vector libm implemented with high-level scalar C that a modern compiler can translate to SIMD code. The error of all functions does not exceed $8\,\mathrm{ulp}$, while a performance gain of up to $278.5\%$ is obtained. Our library is fully free-standing, i.e. it does not depend on any other system library.**

## I. Introduction

Operation Systems provide support for mathematical functions such as $\exp$, $\sin$, $\cos$, $x^y$ through mathematical libraries, commonly called libm. As hardware becomes more and more parallel, in particular through integration of hardware for floating-point [1] SIMD instructions, a need for vector libms arises. These parallelized libms work on vectors, applying the respective mathematical functions in an element-by-element (SIMD-like) manner.

The vector lengths vary: systems may choose to provide only functions for fixed length vectors, typically for the vectors sizes supported by the underlying SIMD hardware, or they may provide APIs for arbitrary vector lengths, cutting the vectors into chunks then mapped onto the SIMD units.

A first advantage of such vectorized libms is to give the user an interface to leverage the full potential of their SIMD hardware. Second, modern compilers such as gcc[1] have support for auto-vectorization, where scalar loops are transformed into SIMD operations and calls to SIMD-like libm functions. Vectorized libms are hence required as support libraries of modern compilers.

Vectorized libms are no new idea. Several such libms already exist: Intel's SVML[2], AMD's libm[3], GNU glibc/gcc recent addition libmvec[4] or CERN's VDT [2]. However, all these existing libraries have certain shortcomings: some of them are proprietary, which does not allow them to be used on a wider range of systems. Others are written in assembly, which also hinders portability. The only library source code is available for, CERN's VDT, is a mere adaptation of an existing scalar libm, with no optimization of the underlying algorithms to a SIMD environment.

In this article, we present a new Open-Source vector libm that was developed to fill the gap left by the existing libraries. It was designed with the following goals in mind: first, our library is written in plain C, making it fully portable and easier to maintain and inspect. However, the C source is written in a manner that allows a modern compiler such as gcc to auto-vectorize the library code to any SIMD hardware unit with reasonable floating-point support. Second, our library uses newly optimized polynomial approximations [3], chosen with parallelization in mind. Finally, the library is completely freestanding, i.e. it does not require a classical scalar libm to handle corner cases of the functions, like libmvec does for example. This further improves code maintainability and prevents certain issues of floating-point non-reproducibility.

This article is organized as follows: in Section II, we analyze properties of existing vector math libraries. In Section III, we give a short overview on the principal techniques used to implement a mathematical function, such as $\exp(x)$, in floating-point arithmetic. In Section IV, we describe the auto-vectorization capabilities of modern C compilers and identify certain issues arising with classical math function implementation techniques with respect to auto-vectorization. In Section V, we describe our approach to implementing vectorized math functions, before presenting experimental results in Section VI. We conclude with an outlook in Section VII.

## II. Existing Vector Math Libraries

As already mentionned, several vectorized libms provided by different parties exist: Intel's SVML and VML libraries, AMDs libm, GNU's libmvec and CERN's VDT. None of these existing libraries fully satisfies all users' needs, in particular in the Open-Source world.

First of all, the libraries provided by Intel and AMD are closed-source and particularly optimized for each vendor's hardware. Alhough libmvec, recently added to the GNU glibc to support auto-vectorization in gcc, is Open-Source software, it is not fully portable: as a matter of fact, the library is provided as an assembly source for x86-64 systems only. Porting it to other hardware, such as ARM systems is next

[1] https://gcc.gnu.org/
[2] https://software.intel.com/en-us/node/583201
[3] http://developer.amd.com/tools-and-sdks/archive/libm/
[4] https://sourceware.org/glibc/wiki/libmvec

to impossible. Maintenance and code-inspection (e.g. to derive bounds on maximum approximation error) are hindered by the assembly nature of libmvec.

CERN's VDT library [2] is Open-Source and fully written in high-level languages (C and C++). It hence portable across systems and allows for code maintenance and inspection. However, VDT is not a full redevelopment: for its core, i.e. the polynomial and rational approximations of the mathematical functions it implements, it is based on the legacy Cephes library[5]. This means that its developers could not optimize these approximations (in terms of degree etc.) with respect to vectorization. Finally VDT has a C++ interface, which makes it unsuitable for certain C-based projects and hinders its use as a support library for a C compiler.

All existing vector libraries share one common issue: they all depend on a scalar libm to implement certain special cases of certain functions. For example, most vector libraries implement the trigonometric functions such as $\sin(x)$ only in a limited domain around zero. For larger input values, they call respective function in the scalar math library. While this approach is reasonable with respect to performance, as large input values to trigonometric functions are rare in well-written codes, reproducibility issues may arise for input values around the limit when the vector library switches from internal handling to calling the scalar library: the code-base and hence floating-point properties (such as maximum error) for both implementations of the same mathematical functions are not the same. Reproducibility issues, such as monotonicity problems, may hence arise. These issues are even more widespread for non-freestanding vector libms that choose to call the scalar math library for all elements of a vector if at least one of the vector elements requires special case handling.

## III. Implementation of Mathematical Functions

The implementation of a mathematical function, such as $\exp(x), \sin(x)$ or $\operatorname{atan}(x)$, in the floating-point environment provided by IEEE754-2008 has been extensively studied and described in the literature [4], [5], [6], [7], [8], [9], [10]. Classically, a mathematical function on a floating-point argument is computed in four or five steps.

In the first step, the inputs are filtered for special values such as infinities or Not-A-Number data, as well as for numeric input values that lie outside of the function's definition domain or surely provoke overflow or underflow. These special inputs are handled appropriately.

All unhandled numeric floating-point inputs proceed to the second step, called argument reduction step. In this step, algebraic properties of the implemented function, such as $e^{a+b} = e^a \cdot e^b$ or $\sin(x + 2k\pi) = \sin(x)$, $k \in \mathbb{Z}$, are used to reduce the domain the function needs to be actually computed on to some small interval, mostly around zero. In cases when the function does not allow for algebraic reduction of its input domain or when the domain obtained after algebraic argument reduction is still to large, the domain is simply split into subdomains. Such subdomain splitting gets reflected in the function's implementation by a sequence of memory reads and conditional branches.

In the third step, called polynomial approximation step, the function's value gets approximated by evaluation of an approximation polynomial. In some cases, a rational approximation may also be used. This part of the code is fully straight-line but may be optimized in terms of degree of the polynomial used and of amount of memory to be read in for the polynomial's coefficients [3].

In a fourth and final step, the function's value on its original argument is recovered by "inverting" the effects of the argument reduction step. This final code sequence commonly is pretty short.

In cases when addressing a slightly larger (about $8$ to $24$ kbytes) amount of memory with indirect addressing is possible, an additional step, executed in parallel with the polynomial approximation step, called table lookup step, may be used [4]. The use of a lookup table with precomputed values allows the reduced argument's domain to be yet smaller, which decreases the degree of the approximation polynomial and hence evaluation latency.

## IV. Leveraging Compiler Support for Autovectorization

In Section II, we have expressed our concern with the use of assembly in different vector libms on the one hand. On the other hand, if we wish to increase portability with the use of a high-level language to implement a vector libm, we need to find a way to express the SIMD parallelism in that high-level language. A good compiler can then perform SIMD instruction selection, hence may avoid us the use of assembly.

With modern C compilers, such as gcc, the SIMD parallelism actually does not need to be expressed explicitly: when enabling so called auto-vectorization, the compiler is able to map scalar code sequences that get executed repeatedly with in a loop onto SIMD instructions. As a matter of course, both the loop with its length and memory access pattern and the scalar code inside that loop need to satisfy certain conditions in order to allow the compiler to perform the auto-vectorization.

In our use-case, the implementation of mathematical functions, the loop around the scalar floating-point code is of statically known length (most commonly a small multiple of the underlying hardware's native SIMD vector length). The access pattern to the input and output vectors is strictly linear. Satisfying the compiler's constraints is easy toward this respect.

The constraints on the scalar code inside the loop are more restricting for our purpose. We were unable to find any (formal) specification of the actual constraints required by an autovectorizing C compiler such as gcc but we were able, by experiment, to guess the following constraints:

- The scalar code inside the loop needs to be free of conditional branches. Conditional expression evaluation (thru the C operator `cond ? a : b`) is possible in very easy cases: both expressions are integer constants. This constraint typically has the effect that argument reduction by domain splitting cannot be used for the implementation of vector math functions.

[5]http://www.netlib.org/cephes/

- All memory accesses need to be either to a constant address (e.g. to load literal constants) or to consecutive addresses, linearly depending on the loop variable. Hence it is of course possible to read a function's argument in a vector and to write the function's value back to an output vector. The constraint however prohibits the use of lookup tables for argument reduction, as the index to the lookup table does not merely depend on the loop variable but on the floating-point values given in the argument vector.

We are going to explain the next Section which approach we propose for the implementation of mathematical functions whilst still satisfying these constraints on the code, set out by the capabilities of auto-vectorizing compilers.

## V. A Freestanding Vector Math Library

Our goal is to propose a freestanding vector math library, fully written in C, with code that can be auto-vectorized with a modern compiler, such as gcc. We strive at providing the most common functions, namely $\exp, \log, \sin, \cos, \tan, \mathrm{asin}, \mathrm{acos}, \mathrm{atan}, \sqrt[3]{}$ in both the IEEE754-2008 binary32 and binary64 format. In this paper, we concentrate on the IEEE754-2008 binary64 format, leaving out the binary32 format. We do this for two reasons: one an auto-vectorizable algorithm is known for a function in the binary64 format, transcribing it to the binary32 format is a pure matter of software development. Second, as we are going to explain in more detail below, a scalar implementation of each function is required as a fallback in certain cases. While this doubles development work for the binary64 format, performance is not of an issue for that fallback. The binary32 format can hence reuse the same fallback functions, converting hence and forth from binary32 to binary64.

### A. A Bird's Eyes View on the Proposed Algorithms

The algorithms in our proposed vector libm keep the principal structure of an implementation of a mathematical function, as described in Section III: special case handling, argument reduction, polynomial approximation and reconstruction. As set out in Section IV, no conditional branches are possible inside the loop to be vectorized. We therefore cut the loop on the vector into two loops.

First, a loop passes over all vector elements and determines if the corresponding inputs lie inside the main definition domain of the function. This test is implemented using integer operations that work on the memory representation of IEEE754-2008 binary floating-point numbers. The outputs of each vector element's boolean test result are considered integer numbers that are summed. Autovectorizing compilers are able to translate this construction into a vector reduction operation. If at least one of the elements lies outside the main definition domain of the function, a fallback function is called (see Subsection V-B for details).

If all vector inputs lie inside the main definition domain of the function, a second loop is started that will compute the mathematical function in a SIMD-parallel manner. Manual analysis of the generated assembly shows that no memory accesses to the input vector are needed for this loop, as the vector elements have already been loaded for the first, domain-check loop. Inside the function computation loop an argument reduction will first be performed. That argument reduction is always based on an algebraic property of the implemented function, at least for the $9$ functions currently implemented. No subdomain splitting is performed, no tables are used. We are going to give more details in Subsection V-C.

Polynomial approximation follows argument reduction. Its SIMD auto-vectorization is trivial to achieve. The degrees of the used polynomials are pretty high, as the use of no tables for argument reduction requires the polynomials to be accurate on larger domains. For certain functions, in particular the inverse trigonometric functions ($\mathrm{asin}, \mathrm{acos}$ and $\mathrm{atan}$) that have either infinite derivatives at one point or derivatives that tend toward zero, we use more than one approximation polynomial. Their results are then combined in the last, reconstruction step. We have optimized all our polynomials coefficients using the techniques described in [3]. None of our polynomials uses coefficients with higher precision than the working precision, whilst other library tend to use floating-point expansions at least for the coefficients of lower degree [8], [10].

Function reconstruction typically requires no branching nor table access and is hence also easy to auto-vectorize. We always restrict the main path of the function to a domain where the function's value surely does not underflow. This way, we avoid any handling of subnormals on the main path, which is to be autovectorized.

We shall signal a particularity of our implementation of the cubic root function $\sqrt[3]{}$. We implement this function as $\sqrt[3]{x} = 2^{\log_8(x)}$ with additional handling for the sign of $x$. The particularity is that we use the sequence of argument reduction, polynomial approximation and reconstruction twice: once for $z = \log_8 x$, once for $2^z$.

### B. Special Case Handling

As mentionned, we call a fallback function to handle special and other particular cases, such as subnormal inputs, whenever at least one element in the given input vector lies outside the main definition domain of the function. This general approach gives convincing results in terms of average performance but requires two parameters to be determined: first, what is the main definition domain of a given function, and, second, which SIMD vector length is appropriate for our approach.

The first parameter is pretty easy to determine: for most functions, it is given by their mathematical definition domain and the properties of the floating-point format –with its exponent width defined by IEEE754-2008. Only for some functions, typically the trigonometric functions $\sin, \cos$ and $\tan$, a choice must be made by design. Essentially, their argument reduction $r = x - k \cdot \pi$ with $k$ the integer nearest to $x/\pi$ becomes increasingly harder for $x$ becoming larger in terms of required working precision. Specialized argument reduction schemes, such as Payne-Hanek argument reduction [11], exist for large input values $x$. Their SIMD parallelization is hard to achieve. In contrast, large input values to trigonometric functions are not quite common in scientific codes. It is hence reasonable to handle only a certain main definition domain

around zero in a SIMD vectorized manner and to handle all other (larger) inputs only in the scalar fallback code. We chose to handle inputs $x$ smaller in magnitude than $\left(2^{14}-1\right)\cdot\pi$ on the main path and to perform Payne-Hanek argument reduction only in the scalar fallback function. This choice is motivated by error analysis but stays a somewhat arbitrary design decision.

The second parameter to determine with respect to our approach to special case handling is the vector length. Our approach is to call a fallback function as soon as at least one of the input vector elements does not lie in the main evaluation domain. That fallback code is not SIMD-parallelized and hence significally slower. The average performance of our vector functions is hence determined by two parameters: the probability $p$ to encounter a out-of-main-domain input (amongst a supposedly large number of inputs) and, of course, the vector length $n$. With $n$ increasing, the probability for the main-domain SIMD-parallelized code to be used becomes lower, typically $(1-p)^n$. Further, when the fallback function gets called, with increasing vector length $n$, its evaluation time increases. In contrast, when no fallback is necessary, the SIMD-parallelized function typically gets more efficient with increasing vector length, as most compilers are able to interleave several SIMD evaluations, to make them run simultaneously. As the probability $p$ of out-of-domain inputs is typically unknown, it is hard to answer that second question of determining the vector length $n$. We performed some experiments with some assumptions on the distribution of floating-point numbers, in particular equidistribution of the exponents. We could determine that a vector length of $n=8$ seems to be most efficient on average. Anyway, as our code is written in a high-level language, adapting it to other vector lengths is pretty easy.

### C. Avoiding the Use of Lookup Tables

The constraint that code to be auto-vectorized must not use any indirect memory accesses where the addresses depend on the input vector implies that we may not use any lookup tables in our implementations. This has several consequences:

Most approaches for argument reduction found in the literature for the implementation of mathematical functions do use tables [4], [7]. We have to revise the argument reduction techniques. For certain functions, this may involve nothing but dropping the table lookup step, increasing the interval the reduced argument may fall in and hence increasing the degree of the approximation polynomial. The functions $\exp(x)$ and $\log(x)$ are examples for this situation.

For other functions, like $\cos(x)$, not having access to lookup tables –combined with the required avoiding of conditional branches– requires overcoming issues of computing and writing evaluation code for polynomials approximating a function that has a zero elsewhere than at zero. However, approaches for this issue do exist [12].

Finally, some other functions, like $\operatorname{asin}(x), \operatorname{acos}(x)$ or $\operatorname{atan}(x)$, have derivatives that become infinite at some point or that tend towards zero for the input argument going to infinity. In this cases, using polynomials for approximation is most inappropriate. Classical argument reduction techniques for these functions with lookup tables try to hide the singularities of the derivatives in the tables. For our work, we had to find other argument reduction schemes. The guiding idea is that an accurate evaluation of the inverses of these tricky functions is easy to obtain with polynomial approximation and evaluation. Argument reduction may hence proceed as follows: first evaluate a low-degree polynomial approximation of the direct function, yielding a pretty inaccurate result. Then compute an accurate approximation of the inverse function at the point found by the inaccurate approximation. Last compute a reduced argument out of the original evaluation point and the result of the inverse function evaluation. Finally accurately evaluate the function of the reduced argument, which will be bounded in magnitude by some upper bound related to the accuracy of the initial inaccurate approximation.

### D. Example: Implementation of the Logarithm Function

Let us now detail the implementation of the logarithm function as an example for the functions in our vector libm. The function $\log x$ is defined for all inputs $x>0$. On output, it may not produce any underflow nor overflow [7].

We remove the handling of subnormal input from the main path: the test loop checking whether special case handling is required tests whether all IEEE754-2008 binary64 inputs $x$ are between $2^{-1021}$ and $2^{1023}\cdot\left(2-2^{-52}\right)$ (bounds included). Since we implement this test with two integer comparisons on the IEEE754-2008 memory representation, the same test eliminates all zeros, infinities and Not-A-Numbers as well.

All inputs $x$ in that range are then split into an exponent $E$, stored on an integer variable, and a significand $m$, stored on an IEEE754-2008 binary64 variable, such that $x=2^E\cdot m$ and $0.75\leq m<1.5$. The uncommon range for the significand $m$ between $0.75$ and $1.5$ requires extra work but allows a catastrophic cancellation to be avoided [7]. It is possible to perform that split of the input $x$ into the exponent $E$ and the significand $m$ without using branches, merely with integer operations on the IEEE754-2008 memory representation, even when targeting the uncommon significand range between $0.75$ and $1.5$.

A constant value of 1 is then subtracted from the significand $m$, yielding $r=m-1$. This subtraction does not provoke any rounding, as per Sterbenz' lemma. The argument of the logarithm function is hence reduced as follows:

$$\log x = E\,\log 2 + \log\left(1+r\right),$$

where $r$ varies in the interval $r\in[-0.25;0.5]$.

On this domain, the function $\log\left(1+r\right)$ can be replaced by a polynomial $p(r)$ of degree 20 to achieve the required accuracy for IEEE754-2008 binary64.

Reconstruction involves converting the exponent $E$, originally represented on an integer variable, to a IEEE754-2008 variable and multiplying it by the constant $\log 2$. The conversion is performed using the appropriate machine instruction, available in most SIMD instruction sets. It is expressed as a cast in C and runs in parallel with polynomial evaluation. The constant $\log 2$ is stored with some extra precision as an

| **Function** | **Vector libm** max error in ulps | **System libm** cycles per element | **Vector libm** cycles per element | **Speed-up** |
|---|---|---|---|---|
| exp | 2.6695 | 40.2041 | 10.6208 | 278.5% |
| log | 1.0825 | 84.7105 | 31.6007 | 168% |
| sin | 2.3300 | 263.1053 | 176.7726 | 48.8% |
| cos | 2.4221 | 251.0564 | 204.5421 | 22.7% |
| tan | 3.0352 | 280.7870 | 234.8114 | 19.5% |
| asin | 4.9627 | 24.3864 | 14.1950 | 71.7% |
| acos | 2.4031 | 23.2520 | 15.7615 | 47.5% |
| atan | 2.1013 | 16.7001 | 8.7906 | 89.9% |
| cbrt | 1.1638 | 48.9236 | 28.2182 | 73.3% |

Table I
MEASURED MAXIMUM ERROR OF VECTOR LIBM AND MEASURED PERFORMANCE OF SYSTEM AND VECTOR LIBM

unevaluated sum of two IEEE754-2008 numbers with some zero trailing bits in their significand to make the multiplication by the exponent $E$ exact.

## VI. Experimental results

The vector libm we propose was developed targeting a maximum error of the functions of at most $8\,\mathrm{ulp}$ with respect to the corresponding precision of the function. While this target means that our functions are clearly less accurate than standard scalar libms, which commonly provide correct rounding [1], [8], [13], it well corresponds to the accuracy requirements of High-Performance Computing e.g. for Physics [2]. In any case, providing correct rounding for vectorized transcendental functions is extremely hard, due to the divergence of the execution paths in the different SIMD slots created by the rounding test [13]. Non-exhaustive measurements show that our library fulfills that accuracy target. Most functions actually have a maximum error below $5\,\mathrm{ulp}$ (see Table I).

Concerning performance, the speed-up (in terms of cycles per element computed) provided by our library, compiled with gcc 4.9.2, compared with respect to a scalar libm, for instance the GNU glibc 2.19-18+deb8u4 running Debian 3.16.0-4-amd64 on a x86-64 Intel i7-5500U at 2.4GHz, is pretty convincing: for certain functions, such as $\exp(x)$ we obtain a speed-up of up to $278.5\%$. For certain functions, like $\tan(x)$, which we optimized less, we obtain a speed-up of at least $19.5\%$. See Table I for details. Additional experiments showed that about half of the speed-up is due to a better utilization of the hardware (SIMD processing, decreasing function call overhead etc.); the other half stemming from optimization of the approximation polynomials.

## VII. Conclusion and Future Work

We presented a vectorized mathematical library (libm) that is fully implemented in modern C, allowing for good code maintenance and portability. The library is Open-Source, which is not commonly the case for its competitors. It provides reasonable speed-up with respect to a scalar libm, attaining up to $278.5\%$ of speed-up per vector element. It is fully free-standing as it does not require any scalar libm as a support library. The maximum error of the functions provides stays well below $8\,\mathrm{ulp}$, corresponding to accuracy requirements of High Performance Computing.

As it stands now, the library is not fully completed with respect to the functions provided by a classical scalar libm. We currently concentrated our efforts on the IEEE754-2008 binary64 format. We still need to derive an implementation of the functions for IEEE754-2008 binary32 format from the algorithms we proposed. Research and design work is also ongoing for certain functions that are hard to implement in a vectorized manner, such as the power function $x^y$, the bivariate arctangent $\mathrm{atan}\left(\frac{y}{x}\right)$ or the Gauß error function $\mathrm{erf}(x)$.

## References


[1] IEEE Computer Society, *IEEE Standard for Floating-Point Arithmetic*. IEEE Standard 754-2008, Aug. 2008.

[2] D. Piparo, V. Innocente, and T. Hauth, "Speeding up hep experiment software with a library of fast and auto-vectorisable mathematical functions," *Journal of Physics: Conference Series*, vol. 513, no. 5, p. 052027, 2014. [Online]. Available: http://stacks.iop.org/1742-6596/513/i=5/a=052027

[3] N. Brisebarre and S. Chevillard, "Efficient polynomial $L^\infty$-approximations," in *18th IEEE SYMPOSIUM on Computer Arithmetic*, P. Kornerup and J.-M. Muller, Eds. Los Alamitos, CA: IEEE Computer Society, June 2007, pp. 169–176.

[4] P.-T. P. Tang, "Table-driven implementation of the exponential function in ieee floating-point arithmetic," *ACM Trans. Math. Softw.*, vol. 15, no. 2, pp. 144–157, Jun. 1989. [Online]. Available: http://doi.acm.org/10.1145/63522.214389

[5] S. Gal and B. Bachelis, "An accurate elementary mathematical library for the IEEE floating point standard," *ACM Transactions on Mathematical Software*, vol. 17, no. 1, pp. 26–45, March 1991.

[6] M. Cornea, J. Harrison, and P. T. P. Tang, *Scientific Computing on Itanium-Based Systems*. Intel Press, 2002.

[7] F. de Dinechin, C. Q. Lauter, and J.-M. Muller, "Fast and correctly rounded logarithms in double-precision," *RAIRO, Theoretical Informatics and Applications*, vol. 41, pp. 85–102, 2007.

[8] C. Lauter, "Arrondi correct de fonctions mathématiques, fonctions univariées et bivariées, certification et automatisation," Ph.D. dissertation, École Normale Supérieure de Lyon, Lyon, 2008.

[9] J.-M. Muller, N. Brisebarre, F. de Dinechin, C.-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, and S. Torres, *Handbook of Floating-Point Arithmetic*. Boston: Birkhäuser, 2010.

[10] N. Brunie, F. de Dinechin, O. Kupriianova, and C. Lauter, "Code generators for mathematical functions," in *22nd IEEE Symposium on Computer Arithmetic*, 2015, pp. 66–73.

[11] M. Payne and R. Hanek, "Radian reduction for trigonometric functions," *SIGNUM Newsletter*, vol. 18, pp. 19–24, 1983.

[12] C. Lauter and M. Mezzarobba, "Semi-automatic floating-point implementation of special functions," in *Proceedings of the 22nd IEEE Symposium on Computer Arithmetic*, 2015.

[13] F. de Dinechin, C. Lauter, J.-M. Muller, and S. Torres, "On Ziv's rounding test," *ACM Trans. Math. Softw.*, vol. 39, no. 4, pp. 25:1–25:19, Jul. 2013.