Slimmer Formal Proofs for Mathematical Libraries

Paul Geneau de Lamarlière*†, Guillaume Melquiond†, and Florian Faissole*
* Mitsubishi Electric R&D Centre Europe, 35700 Rennes, France.
† Université Paris-Saclay, CNRS, ENS Paris-Saclay, Inria, LMF, 91190, Gif-sur-Yvette, France.

***Abstract*—Short of being able to exhaustively test all the inputs, writing a formal proof offers the highest possible confidence in the correctness of a mathematical library. This comes at a large cost though, since formal proofs require taking into account all the details, even the seemingly insignificant ones, which makes them tedious to write. This issue is compounded by the fact that the objects whose properties we need to verify (floating-point numbers) are not the ones we would like to reason about (real numbers and integers). This short paper explores some ways of reducing the overhead of formal proofs in the setting of mathematical libraries, so as to let the user focus on the details that really matter.**

***Index Terms*—Floating-point arithmetic; Mathematical libraries; Formal methods; Coq proof assistant**

I. Introduction

The floating-point functions offered by a mathematical library are often intricate pieces of code that strive for accuracy and speed, and their development is an error-prone process. For a univariate binary32 function, one could exhaustively test all the inputs to make sure that the function is correct, as done for the CORE-MATH library [1]. But for larger input domains, one has to resort to a mathematical proof. The highest confidence is then reached by writing a formal proof and having it mechanically checked [2, §13.2].

When verifying a floating-point function, it is often useful to interpret its code in different ways. First of all, the code can be interpreted as a computation over IEEE-754 floating-point numbers, including infinities and NaNs [2, §2.1]. Ideally, any property should be proven on this model as this is the way the function actually computes. This is the model followed by the SMTLIB standard and the SMT solvers that adhere to it [3]. But this SMT theory does not offer any way to relate finite floating-point numbers to the real numbers they represent. As a consequence, expressing the distance between a computed value and the real number it is meant to approximate is impossible, which considerably limits the usefulness of this model when verifying mathematical libraries.

Since the IEEE-754 standard mandates that floating-point operations should behave as if they were first computing an infinitely precise result and then rounded it to the target format, computed values can also be interpreted as rounded real numbers. This makes it possible to express the error between the computed value and some ideal result. So as to ease the use of known properties of floating-point arithmetic such as $|\circ(x) - x| \le 2^{-prec}|x|$, it is better to let exponents grow arbitrarily large. This is the main model used when doing pen-and-paper proofs due to its expressiveness, but it comes at the expense of not being faithful to the IEEE-754 standard.

In the context of an interactive theorem prover, we would like to interpret floating-point numbers as rounded real numbers, but a formal proof would only be complete once all the exceptional cases (*e.g.*, overflows) have been handled. Proving that a floating-point number is finite is something that falls right in the scope of SMT solvers, since it just involves its representation as a bit vector. But due to the intricacy of mathematical libraries, the solvers would not succeed in a timely fashion without some mathematical knowledge about the approximated functions, which in turn requires the ability to speak about real numbers and rounding errors.

The Flocq library for the Coq proof assistant offers both representations of floating-point numbers, as well as the theorems to switch from one to the other [4, §3]. This short paper explores how to make it easier to juggle between both representations when verifying a mathematical library. The methodology is then illustrated with the formal proof of two floating-point functions: exponential and logarithm. Both examples are available at https://gitlab.inria.fr/pgeneaud/examples.

II. Related Works

Several tools make it easier to go from the IEEE-754 model of floating-point arithmetic to one based on real numbers, during a formal proofs. For the HOL Light proof assistant, one can use FPTaylor [5], which represents a floating-point result as a symbolic affine form, that is, an expression $f(\vec{x}) + \Sigma_i f_i(\vec{x})\varepsilon_i + \mu$ where f and the f_is are real functions of the inputs $\vec{x}$, the ε_i variables represent all the potential rounding errors, and μ is a miscellaneous term that conflates all the second-order behaviors. In other words, the f_is show how the individual rounding errors are propagated at first order. Since the f_is are computed symbolically, this provides a rich description of the numerical behavior of the code.

The VCFloat2 tool plays a similar role for the Coq proof assistant [6]. This time, floating-point values are represented as rational functions in the inputs and the ε_is. These rational functions are first fully expanded and then all the monomials that are below a given threshold, *e.g.*, 2^{-30}, are conflated in a single miscellaneous term. This makes it possible to account for some second-order behaviors, contrarily to FPTaylor, but at the cost of an "exponential time".

III. Language of Expressions

A. Expression trees

Given an abstract arithmetic expression e, we denote $[\![e]\!]_{\mathsf{flt}}$ the floating-point number it computes, according to the IEEE-754 standard. By extension, when $[\![e]\!]_{\mathsf{flt}}$ is finite, it can also be seen as a real number. We denote $[\![e]\!]_{\mathsf{rnd}}$ the value obtained by performing every operation on real numbers and then rounding it to finite precision. We do not restrict the range of $[\![e]\!]_{\mathsf{rnd}}$ in any way, that is, $[\![e]\!]_{\mathsf{rnd}}$ might differ from $[\![e]\!]_{\mathsf{flt}}$ if any operation of the latter overflows. Finally, we might need to talk about the result $[\![e]\!]_{\mathsf{exa}}$ that would have been obtained with an infinitely precise arithmetic. Note that $[\![e]\!]_{\mathsf{exa}}$ is the same as the coefficient $f(\vec{x})$ of the affine form computed by FPTaylor.

As an illustration, assume that we want to prove that some code e approximates some ideal mathematical value E with a relative error at most ε. The specification looks like "$[\![e]\!]_{\mathsf{flt}}$ is finite and $|[\![e]\!]_{\mathsf{flt}}/E - 1| \le \varepsilon$". The first step is to prove that no exceptional behavior occurs during the evaluation of $[\![e]\!]_{\mathsf{flt}}$ and thus that it is sufficient to prove $|[\![e]\!]_{\mathsf{rnd}}/E-1| \le \varepsilon$. To verify the latter, one usually splits the inequality into a rounding error $|[\![e]\!]_{\mathsf{rnd}}/[\![e]\!]_{\mathsf{exa}} - 1| \le \varepsilon_1$ and a method error $|[\![e]\!]_{\mathsf{exa}}/E - 1| \le \varepsilon_2$ with $\varepsilon \ge \varepsilon_1 + \varepsilon_2 + \varepsilon_1\varepsilon_2$. When using the Coq proof assistant, the bound on the rounding error can be verified by the Gappa tool, while the method error can be tackled by the CoqInterval library [4, §4].

Unsurprisingly, the expression e is represented by the value of an inductive type in Coq, that is, an abstract syntax tree whose internal nodes are arithmetic operations. Floating-point addition, subtraction, multiplication, division, square root, and fused multiply-add are supported in rounding to nearest. Integer operations, as well as functions `nearbyint`, `trunc`, and `ldexp` are also supported, as they are commonly encountered during argument reduction and result reconstruction in mathematical libraries. Thanks to Coq's dependent type system, only well-formed expressions can be expressed; for example, one cannot pass a floating-point number to an integer division.

The interpretation $[\![e]\!]_{\mathsf{flt}}$ is defined using Flocq's formalization of the IEEE-754 standard [4, §3.4]. Contrarily to the arithmetic operations, the conversions to integer `nearbyint` and `trunc` were originally not supported by Flocq, so we have formalized them. The interpretation $[\![e]\!]_{\mathsf{rnd}}$ also relies on Flocq, but this time, it uses the formalization of the so-called FLT formats [4, §3.1]. These formats are defined as the sets $\{x \in \mathbb{R} \mid \exists m, e \in \mathbb{Z}, x = m \cdot 2^e \wedge |m| < 2^p \wedge e \ge e_{\min}\}$ with p the precision and $e_{\min}$ the minimal exponent (*e.g.*, $p = 53$ and $e_{\min} = -1074$ for binary64). As for the operations on real numbers used in $[\![e]\!]_{\mathsf{rnd}}$ and $[\![e]\!]_{\mathsf{exa}}$, they are defined in Coq's standard library.

Our expression trees offer two additional features. First, they support let-bindings, with binders represented by their de Bruijn indices. The primary use of these bindings is to explicitly express sharing between sub-expressions, as in most programming languages. But they will also play an important role when reasoning about expressions, as explained in Section IV-C.

The second feature is the availability of exact arithmetic operations. Indeed, when implementing accurate approximations of mathematical functions, one might arrange floating-point operations in a way such that their results are exactly representable. For example, the FastTwoSum operator is commonly used during the last step of a polynomial evaluation. It is thus important that the user can annotate the expression e so that $[\![e]\!]_{\mathsf{rnd}}$ is not polluted with spurious rounding operators.

B. Correspondence theorem

As explained above, to verify the correctness of an expression e, we need to prove a property about the floating-point value $[\![e]\!]_{\mathsf{flt}}$, but this is usually too cumbersome and we would like instead to reason about the rounded real number $[\![e]\!]_{\mathsf{rnd}}$. We say that an expression e is well-behaved if $[\![e]\!]_{\mathsf{flt}}$ is a finite floating-point number and it represents the real number $[\![e]\!]_{\mathsf{rnd}}$. Thus, as long as we prove that e is well-behaved, we can reason over $[\![e]\!]_{\mathsf{rnd}}$.

We recursively define a logical predicate WB over expressions in such a way that, if $\mathrm{WB}(e)$ holds, then e is well-behaved. For example, the formula $\mathrm{WB}(u/v)$ for a floating-point division is

$$\mathrm{WB}(u) \wedge \mathrm{WB}(v) \wedge [\![v]\!]_{\mathsf{rnd}} \neq 0 \wedge |\circ([\![u]\!]_{\mathsf{rnd}}/[\![v]\!]_{\mathsf{rnd}})| \le \Omega$$

with Ω the value of the largest finite floating-point number. In other words, for the expression u/v to be well-behaved, it is sufficient that u and v are well-behaved, that v is non-zero, and that the division does not overflow.

For the other floating-point operations, WB is defined in a similar way. For exact operations, the formula contains an additional conjunct that states that the result is exactly representable, *e.g.*, $\circ([\![u]\!]_{\mathsf{rnd}} + [\![v]\!]_{\mathsf{rnd}}) = [\![u]\!]_{\mathsf{rnd}} + [\![v]\!]_{\mathsf{rnd}}$ for addition.

Theorem 1. *Given an expression e and some predicate G over the real numbers, we have*

$$\mathrm{WB}(e) \wedge G([\![e]\!]_{\mathsf{rnd}}) \Rightarrow [\![e]\!]_{\mathsf{flt}}\ \textit{finite} \wedge G([\![e]\!]_{\mathsf{flt}}).$$

In other words, to prove some property $G([\![e]\!]_{\mathsf{flt}})$ over floating-point numbers, *e.g.*, a bound on the distance to an ideal result, one just has to prove the same property over rounded real numbers $G([\![e]\!]_{\mathsf{rnd}})$, assuming $\mathrm{WB}(e)$ holds.

IV. Proof Tools

A. Polynomial approximations and rounding operators

The CoqInterval library is able to automatically verify bounds over expressions mixing arithmetic operations and some elementary functions (*e.g.*, $\exp$, $\ln$, $\cos$, $\sin$, $\tan$, $\arctan$). It does so by computing a reliable polynomial approximation of the expression, that is, a polynomial that approximates it and an interval that encloses the distance between this polynomial and the original expression [7]. This is especially effective when bounding the method error $[\![e]\!]_{\mathsf{exa}}/E - 1$. But since CoqInterval does not know about rounding operators, it is of little help to prove $\mathrm{WB}(e)$ as all the conjuncts of the form $|\circ([\![u]\!]_{\mathsf{rnd}} \diamond [\![v]\!]_{\mathsf{rnd}})| \le \Omega$ are thus out of its scope.

To make CoqInterval useful for our purpose, we have extended its set of supported operators with some rounding operators from the Flocq library. These rounding operators take a real number and return one of the closest numbers in an FLT format described by a precision p and a minimal exponent $e_{\min}$. The formalization was performed in two steps.

First, we have taught CoqInterval how to compute an enclosure of the rounding error $\circ(y) - y$ given an enclosure of y. If $|y| \leq 2^n$, then $|\circ(y)-y| \leq 2^{n'}$ with $n' = \max(n-p+1, e_{\min})$.

Second, we have extended it to reliable polynomial approximations. Consider an expression $f(x)$ and its approximation (P, Δ) over an interval X, that is, $\forall x \in X,\ f(x)-P(x) \in \Delta$. From P and Δ, we can find n such that $|f(x)| \leq 2^n$ for any $x \in X$. Thus, $(P, \Delta + [-2^{n'}, 2^{n'}])$ is an approximation of $\circ(f(x))$ with n' computed as above. The correctness of this polynomial approximation is a straightforward consequence of the following equality: $\circ(f(x)) - P(x) = (\circ(f(x)) - f(x)) + (f(x) - P(x))$.

This modification of CoqInterval is sufficient to make it automatically prove most of the formula $\mathrm{WB}(e)$ in practice. Preconditions of exact operations (*e.g.*, $\circ([\![u]\!]_{\mathsf{rnd}} + [\![v]\!]_{\mathsf{rnd}}) = [\![u]\!]_{\mathsf{rnd}} + [\![v]\!]_{\mathsf{rnd}}$) are still out of the scope of CoqInterval and would have to be proved using a different approach, for example using Gappa.

Since CoqInterval is now able to verify bounds on expressions involving rounding operators, it could also be used to directly verify a bound on an absolute error $[\![e]\!]_{\mathsf{rnd}} - E$ rather than just $[\![e]\!]_{\mathsf{exa}} - E$. But since the modeling of rounding errors is a bit naive, one should not expect the bounds on $[\![e]\!]_{\mathsf{rnd}} - [\![e]\!]_{\mathsf{exa}}$ to be as tight as those found by Gappa. This is especially true when considering relative errors instead. CoqInterval is not meant to compete with FPTaylor either, since the latter is able to notice first-order error compensations.

B. *From* $[\![e]\!]_{\mathsf{flt}}$ *to* $[\![e]\!]_{\mathsf{rnd}}$

A proof assistant like Coq is primarily used for backward reasoning, that is, the user works on the conclusion of a theorem (the "goal") and simplifies it until it can be trivially deduced from the hypotheses. Theorem 1 is the primary way to get rid of the occurrences of $[\![e]\!]_{\mathsf{flt}}$ from the goal and to replace them with $[\![e]\!]_{\mathsf{rnd}}$. We provide two proof strategies for that purpose. The first one automatically applies Theorem 1 to the current goal. Indeed, the goal might not have quite the correct form to apply the theorem, so the strategy takes care of preprocessing it, thus avoiding some cumbersome manipulations from the user.

The second proof strategy, `simplify_wb`, is much more important. It takes care of the hypothesis $\mathrm{WB}(e)$ that results from the application of Theorem 1. In essence, it just applies CoqInterval to each conjunct individually and removes those that were automatically proved. The user could perform this process by hand, but it would not scale. Indeed, the formula $\mathrm{WB}(e)$ grows quadratically in the size of e, as there is a lot of redundancy between the various conjuncts. Consider $\mathrm{WB}(u+v)$; it requires computing polynomial approximations of $[\![u]\!]_{\mathsf{rnd}}$ (resp. $[\![v]\!]_{\mathsf{rnd}}$) to prove $\mathrm{WB}(u)$ (resp. $\mathrm{WB}(v)$), but it also requires these polynomial approximations to deduce the one for $\circ([\![u]\!]_{\mathsf{rnd}} + [\![v]\!]_{\mathsf{rnd}})$. It is thus important to share as much work as possible, hence the dedicated strategy.

C. *Structuring proofs*

When verifying some goal $G([\![e]\!]_{\mathsf{flt}})$ or $G([\![e]\!]_{\mathsf{rnd}})$, one often needs to first prove some property of some sub-expression. For example, knowing that an argument reduction has indeed produced a reduced argument might help in proving a bound on the rounding error of the subsequent polynomial evaluation. In other words, we want to be able to assert properties about sub-expressions. Any proof assistant provides such a critical feature, but it is cumbersome for our use case, as one needs to explicitly write the possibly large sub-expression.

So, we have decided to turn let-bindings into implicit assertion points, that is, any asserted predicate will necessarily be applied to the bound expression of the topmost let-binding. The rationale is that, if a sub-expression is not used at least twice, then it is presumably of little interest on its own. More concretely, if the current goal is $G([\![\mathsf{let}\ t = e_1\ \mathsf{in}\ e_2]\!]_{\mathsf{rnd}})$ and the user asserts some predicate H, two subgoals will be generated: $H([\![e_1]\!]_{\mathsf{rnd}})$ and $\forall t \in \mathbb{R},\ t \in \mathrm{FLT} \wedge H(t) \Rightarrow G([\![e_2]\!]_{\mathsf{rnd}})$.

For example, in the case of an argument reduction, the user just has to write a predicate $H(t)$ that states that t (not $[\![e_1]\!]_{\mathsf{rnd}}$) is in a small interval and is close to the ideal reduced argument. This example will be detailed below.

V. Examples

A. *Cody & Waite's exponential*

The first example we consider is the Cody-Waite implementation of the exponential, which is one of the earliest and is notable for its clever argument reduction [8]. Despite its age, variants of this reduction can still be encountered in modern implementations since it is cheap yet accurate [2, §10.2]. Given an input x, it computes the reduced argument t using the following floating-point operations (see [4, §6.2.2] for the actual constants C_i):

$$\begin{aligned} k &\leftarrow \lfloor x \cdot C_1 \rceil, \\ t &\leftarrow x - k \cdot C_2 - k \cdot C_3. \end{aligned}$$

This code exhibits several interesting features. First, it uses the `nearbyint` function (denoted $\lfloor\cdot\rceil$) and thus involves integers. Second, the two operations in the subexpression $x - k \cdot C_2$ are performed exactly (*i.e.*, no rounding error), which is crucial to guarantee that t is close enough to $x - k \ln 2$. By flagging them as exact in the expression tree, the expression $[\![t]\!]_{\mathsf{rnd}}$ contains fewer rounding operators, which makes it easier to manipulate for the user. Third, the exponential is approximated using a rational fraction rather than a polynomial, which introduces another potential source of exceptional behaviors.

The preexisting Coq proof of this algorithm only considered rounded real numbers [4, §6.2]. Our goal is thus to make it support IEEE-754 floating-point numbers instead, with minimal effort. More precisely, we consider an expression `cwexp` that implements the exponential, except for the final multiplication by 2^{k+1}, and we want to prove the following, for $-746 \leq x \leq 710$ and $k = \lfloor \circ(x \cdot C_1) \rceil$:

$$[\![\mathtt{cwexp}(x)]\!]_{\mathsf{flt}}\ \text{finite} \wedge \left| \frac{2^{k+1} [\![\mathtt{cwexp}(x)]\!]_{\mathsf{flt}}}{e^x} - 1 \right| \leq 2^{-51}.$$

We perform a few manipulations and apply Theorem 1, which yields the goal

$$\mathrm{WB}(\texttt{cwexp}(x)) \wedge \left| \frac{2[\![\texttt{cwexp}(x)]\!]_{\mathsf{rnd}}}{e^{x-k\ln 2}} - 1 \right| \le 2^{-51},$$

where $\mathrm{WB}(\texttt{cwexp}(x))$ contains not only proof obligations for the absence of overflow in all the subexpressions, but also a proof obligation that the denominator of the rational fraction is non-zero, and that the two operations in $x - k \cdot C_2$ are exact. Applying the `simplify_wb` strategy would automatically prove and thus remove all the conjuncts of $\mathrm{WB}(\texttt{cwexp}(x))$ besides the proof obligations for both exact operations. We would then be left with a goal that is logically equivalent to the original theorem statement that used rounded real numbers.

That goal, however, is hardly readable, due to the sheer size of the expressions. So, to make the proof process a bit more palatable, it is better to first separate the argument reduction from the fraction evaluation. Since we turned let-bindings into implicit assertion points, we can simply assert the following property, as was already done in the original Coq proof:

$$|[\![t]\!]_{\mathsf{rnd}}| \le 355 \cdot 2^{-10} \wedge |[\![t]\!]_{\mathsf{rnd}} - (x - k\ln 2)| \le 65537 \cdot 2^{-71}.$$

This extra step is not strictly necessary, since we could have just reused the original proof here. But if we were to start from scratch, the ability to easily state this assertion would have been a great help in proving the theorem.

B. CORE-MATH's logarithm

We have also exercised our approach on a state-of-the-art implementation: the logarithm from the CORE-MATH library [1]. Contrarily to the exponential, the argument reduction is much more straightforward, since it is mostly a matter of decomposing x into $t \cdot 2^e$ with $t \in [\sqrt{2}/2, \sqrt{2}]$. The code further reduces it to $z = t \cdot r_i - 1$ using a tabulated value r_i such that $z \simeq 0$. The interesting part comes afterwards, as the code relies on the following approximation:

$$\ln(t) \simeq P(z) + z + C_{i,1} + C_{i,2}$$

where P is a degree-6 polynomial and where $C_{i,1}$ and $C_{i,2}$ are two tabulated values whose sum approximates $-\ln r_i$.

As of writing this article, our tool allows us to prove a correctness property on the polynomial evaluation $P(z)$, which uses fused multiply-add operations. More precisely, assuming that $z = t \cdot r_i - 1$ is close enough to 0, we want to prove

$$[\![P(z)]\!]_{\mathsf{flt}} \text{ finite} \wedge |[\![P(z)]\!]_{\mathsf{flt}} - (\ln(1+z) - z)| \le 2^{-68.72}.$$

Applying Theorem 1 yields a subgoal $\mathrm{WB}(P(z))$ which consists only of proof obligations of the absence of overflow. These obligations are automatically proved by the strategy `simplify_wb`. The remaining goal

$$|[\![P(z)]\!]_{\mathsf{rnd}} - (\ln(1+z) - z)| \le 2^{-68.72}$$

is proved in three lines of Coq, using Gappa and CoqInterval.

VI. Conclusion

We have presented a way to express the kind of floating-point expressions that appear in the code of mathematical libraries and a few proof strategies to manipulate them. The few examples we have experimented with illustrate that it is effective at tackling the uninteresting but required details of formal proofs. In particular, in the case of Cody & Waite's exponential, a Coq proof already existed but it assumed exceptional behaviors to be irrelevant and represented floating-point values as real numbers. Thanks to our approach, handling actual floating-point numbers instead almost came for free.

This is an early work and the language of expressions is still lacking several important features. First of all, concrete support for arrays should be added, as they are commonly encountered in argument reduction, *e.g.*, for CORE-MATH's logarithm. Another missing piece is control flow. Indeed, while the core of most mathematical functions is branch-free for performance reasons, there might be some conditional execution at the entrance and exit of functions to handle exceptional cases.

The language should later be extended with some macro-operations. For example, a "FastTwoSum" computation is currently expressed as three separate floating-point additions and subtractions. It would be more expressive to have a single macro-operation that receives two numbers x and y and returns two numbers s and e, with a precondition $|[\![x]\!]_{\mathsf{rnd}}| \ge |[\![y]\!]_{\mathsf{rnd}}|$ in WB. The user would then be allowed to use the equality $[\![x]\!]_{\mathsf{rnd}} + [\![y]\!]_{\mathsf{rnd}} = [\![s]\!]_{\mathsf{rnd}} + [\![e]\!]_{\mathsf{rnd}}$ for free in the subsequent proof, without having to deal with rounding.

Acknowledgments

This work was supported by the NuSCAP (ANR-20-CE48-0014) project of the French national research agency (ANR).

References

[1] A. Sibidanov, P. Zimmermann, and S. Glondu, "The CORE-MATH project," in *29th IEEE Symposium on Computer Arithmetic*, Sep. 2022.

[2] J.-M. Muller, N. Brunie, F. de Dinechin, C.-P. Jeannerod, M. Joldes, V. Lefèvre, G. Melquiond, N. Revol, and S. Torres, *Handbook of Floating-Point Arithmetic*, 2nd ed. Birkhäuser Basel, 2018.

[3] M. Brain, C. Tinelli, P. Rümmer, and T. Wahl, "An automatable formal semantics for IEEE-754 floating-point arithmetic," in *22nd IEEE Symposium on Computer Arithmetic*, J.-M. Muller, A. Tisserand, and J. Villalba, Eds., Lyon, France, Jun. 2015, pp. 160–167.

[4] S. Boldo and G. Melquiond, *Computer Arithmetic and Formal Proofs*. ISTE Press – Elsevier, 2017.

[5] A. Solovyev, M. S. Baranowski, I. Briggs, C. Jacobsen, Z. Rakamaric, and G. Gopalakrishnan, "Rigorous estimation of floating-point round-off errors with symbolic Taylor expansions," *ACM Transactions on Programming Language Systems*, vol. 41, no. 1, pp. 2:1–2:39, 2019.

[6] A. W. Appel and A. E. Kellison, "VCFloat2: Floating-point error analysis in Coq," Tech. Rep., 2022. [Online]. Available: https://github.com/VeriNum/vcfloat/blob/master/doc/vcfloat2.pdf

[7] É. Martin-Dorel and G. Melquiond, "Proving tight bounds on univariate expressions with elementary functions in Coq," *Journal of Automated Reasoning*, vol. 57, no. 3, pp. 187–217, 2016.

[8] W. J. Cody, Jr. and W. Waite, *Software Manual for the Elementary Functions*. Englewood Cliffs, NJ: Prentice-Hall, 1980.