Formal Methods in Computer-Aided Design 2024

Formally Verified Rounding Errors of the Logarithm-Sum-Exponential Function

Paul Bonnot*, Benoît Boyer†, Florian Faissole†, Claude Marché* and Raphaël Rieu-Helft‡
*Université Paris-Saclay, CNRS, ENS Paris-Saclay, Inria, LMF, 91190, Gif-sur-Yvette, France
†Mitsubishi Electric R&D Centre Europe, Rennes, France
‡TrustInSoft, 75014 Paris, France

***Abstract*—We study the numerical accuracy of some specific computer program performing numerical computations. Such a numerical accuracy is expressed in terms of a bound on the difference between the floating-point computation and the corresponding rounding-free computation using mathematical real numbers. We do not only seek to discover such a bound "on paper" but we aim at obtaining computer-assisted formal proofs that this bound is correct for any possible inputs. The function we study comes from the domain of machine learning: a function computing the logarithm of the sum of exponentials of a sequence. The bound obtained is an original result, parameterized by the error bounds of the underlying implementations of the logarithm and exponential functions. The methodology we follow to conduct our formal proofs is also original, using a combination of the Why3 environment for deductive verification, an original modelling of floating-point computations using *unbounded* numbers, and the J^3 environment for proving properties on C source code.**

I. Introduction

Software is involved in many industrial systems nowadays. In cyber-physical systems in a broad sense, the software controlling a system must perform numerical computations, that are typically based on floating-point arithmetic. The floating-point representation of numbers, and the operations on them, are standardized by the IEEE-754 standard [43]. Despite of these rules, guessing the accuracy of a program that compounds thousands of elementary operations, without software assistance, becomes almost impossible. However, recent history has shown that underestimating these errors could have catastrophic consequences [57]. That explains the recent interest for the formal verification of floating-point properties of numerical programs in proof assistants [12], [55] or deductive verification platforms [14]. Formally proving the accuracy of floating-point computations is a complex topic addressed by different approaches in the scientific literature. Recent overviews of this topic can be found in the Handbook of Floating-Point Arithmetic [53], or surveys by Melquiond [50] and Boldo *et al.* [18].

In contrast with elementary operations, the accuracy of mathematical functions, say as provided by the `libm` library for C code, is not enforced by the standard and could depend on the target architecture and on a specific library. The same goes for more complex mathematical algorithms, for which the accuracy strongly depends on the accuracies of underlying `libm` functions. In addition, the proof makes sense only if the parameterized bound assumed on `libm` functions can be realized by some implementation. For these reasons, there was very little effort to formally verify programs that call mathematical functions implementations.

An example of algorithm making use of `libm` functions is the *Log-Sum-Exp* algorithm, abbreviated as LSE. It is typically used in machine-learning applications [24], [35], [52]. This algorithm is a smooth approximation of the $\max$ function, the smoothness property of inner functions being a pre-requisite for the efficiency of machine-learning algorithms. It applies to an n-dimensional vector $a = (a_1, \ldots, a_n)$ and computes the logarithm of the sum of exponentials over its a_i components:

$$\mathsf{LSE}(a) = \log\left(\sum_{1\leq i\leq n} \exp(a_i)\right)$$

This function is frequently used in implementations of statistical classifiers [42], [48] and satisfies the following property:

$$\max_{1\leq i\leq n}(a_i) \leq \mathsf{LSE}(a) \leq \max_{1\leq i\leq n}(a_i) + \log(n)$$

Blanchard *et al.* [10] showed that the rounding error of a floating-point implementation of the LSE function can be bounded by relatively tight values as long as no overflow or underflow occurs. They present a pen-and-paper proof taking advantage of both floating-point arithmetic specificities and mathematical properties of the $\log$ and $\exp$ functions. They assume that the implementations of these two functions are correctly rounded, that is, their relative errors are bounded by the ε unit round-off (see Section II). Therefore, they provide the best accuracy that can be achieved in the considered floating-point format.

Compared to Blanchard *et al.* [10], one of the contribution of this paper is to consider the possibility of using less accurate but more efficient implementations of $\exp$ and $\log$. Such implementations may be useful in contexts involving energy-saving small devices like IoT [38]. Another contribution is to give formal proofs of our results. Generally speaking, obtaining formal proofs on the accuracy of floating-point programs is not a simple task. One can start from a software environment for proving functional properties of programs, and augment it with a formalization of floating-point arithmetic, typically via a library built on top of a formalization of real numbers

[0]This research was supported partly by the bilateral contract ProofInUse-MERCE and partly by the Décysif project funded by the Île-de-France region and by the French government in the context of "Plan France 2030"

https://doi.org/10.34727/2024/isbn.978-3-85448-065-5_31

that provides a rounding function and its logical properties. It was done for example by Boldo and Filliâtre [17] using the Coq proof assistant, augmented with the Flocq library [22] for floating-point arithmetic, allowing to prove programs using the Coq general-purpose environment. These proofs typically require a large amount of manually written proof steps. To obtain a higher degree of automation, Ayad and Marché [5] proposed a setting making use of the Frama-C [46] environment for static analysis on C source code, with a dedicated library of ACSL [9] specifications that allows to discharge the proofs to various theorem provers, in particular SMT solvers. Boldo and Marché [19] presented an overview of what could be achieved on increasingly complex codes, using combination of automated solvers and the Coq proof assistant for the most complex proof obligations. With the addition, later on, of some built-in support for floating-point operations in SMT solvers, Fumex *et al.* [34] showed that this methodology can reach a fairly high amount of automation.

Initially we planned to follow the methodology above to prove a C code for LSE. Yet, to achieve the proofs in a reasonably simple manner, we achieved two important points that should be emphasized: first, the use of the intermediate language WhyML, and second the use *unbounded* floating-point numbers. The rest of this paper is organized as follows. In Section II we expose original results bounding the accuracy of LSE, parameterized by assumed accuracies of implementations of $\exp$ and $\log$. Unbounded Floating-Point numbers are detailed in Section II-B. We present our formalization in Section III, formalizing our results up to a formal proof of a C code computing LSE. The WhyML language is introduced in Section III. We discuss related work in Section IV and conclude in Section V with an overview of future work. Due to lack of space, we do not include pen-and-paper proofs here: these proofs can be found in an extended research report of ours [23], to which the reader should refer for any more technical details. The code formalizing our results is publicly available on the Toccata gallery [20], specifically at URL https://toccata.gitlabpages.inria.fr/toccata/gallery/lse.en.html.

II. Statement of Accuracy Results

A. Preliminaries on Floating-Point Arithmetic

The IEEE-754 standard [43] defines several formats of representation of floating-point numbers. A format is characterized by a precision p as well as upper and lower bounds $e_{\max}$ and $e_{\min}$ for the exponent. A floating-point number is either a value among $+\infty$, $-\infty$ and NaN or a value $\pm m \times 2^{e-p+1}$ where $m, e \in \mathbb{Z}$, $0 \le m \le 2^p - 1$ and $e_{\min} \le e \le e_{\max}$. The largest representable number in this format is $\mathsf{maxf} = (2 - 2^{-p-1}) \times 2^{e_{\max}}$, and the smallest positive representable number is $2^{e_{\min}-p+1}$.

In this paper we are not interested in a particular format since our proof methodology is independent of the format used, however only two formats are currently supported in our formal proofs: single format (32 bits) where $p = 24$, $e_{\max} = 127$ and $e_{\min} = -126$, and double format (64 bits) where $p = 53$, $e_{\max} = 1023$ and $e_{\min} = -1022$. For simplicity we focus on the double format.

We use the symbol rnd to denote the *rounding* of a real number to a floating-point number. The IEEE-754 standard defines several rounding modes. Here we consider only the mode *nearest-ties-to-even*: when a real number x lies within an interval $[x_1; x_2]$ of two consecutive floating-point numbers, then $\mathsf{rnd}(x)$ is either x_1 or x_2: the one of these which is closest to x, or in case x is exactly in the middle, the one among x_1 and x_2 whose mantissa is even. Also, when x is too large (larger than or equal to the middle of maxf and $2 \times 2^{e_{\max}}$), $\mathsf{rnd}(x)$ is $+\infty$.

We use the symbols $\oplus$, $\ominus$, $\otimes$, $\oslash$ to denote the basic operations of addition, subtraction, multiplication and division of floating-point numbers. As specified by IEEE-754, all these operations must use the best possible rounding, that is $x \oplus y = \mathsf{rnd}(x+y)$ and similarly for the three other operations.

The main property of the rounding function that we use in this paper is the following: for any real number x such that $|x| \le \mathsf{maxf}$, $\mathsf{rnd}(x)$ is finite and

$$|\mathsf{rnd}(x) - x| \le \varepsilon|x| + \eta \qquad (1)$$

where $\varepsilon = \frac{2^{-p}}{1+2^{-p}}$ and $\eta = 2^{e_{\min}-p}$. This property can be considered as well-known and folklore in the literature, see for example Jeannerod and Rump [44]. In seminal publications, such as the Handbook of Computer Arithmetic [54] or Higham's survey [41], the simpler term $\varepsilon = 2^{-p}$ is used instead of $\frac{2^{-p}}{1+2^{-p}}$, inducing a slightly larger bound. Jeannerod and Rump [44, Theorem 2.1] showed that the refined bound is actually optimal in the sense that there exist some inputs values and floating-point formats (with certain conditions) for which it is attained. In most cases, the precision gain obtained using this optimal bound instead of 2^{-p} is small. Anyway, the latter results and proofs simply use the symbol ε to denote either of the bounds.

As remarked by Jeannerod and Rump [44], Property (1) can be refined in the special case of addition because underflowing additions are exact:

$$|(x \oplus y) - (x + y)| \le \varepsilon|x + y| \qquad (2)$$

that is, the term η can be removed from Formula (1). Moreover, it should be noted that (see for example the Handbook [54])

$$|(x \oplus y) - (x + y)| \le |x| \qquad (3)$$

and symmetrically

$$|(x \oplus y) - (x + y)| \le |y| \qquad (4)$$

The combination of the formulas (2), (3) and (4) is used later on to obtain bounds on compound sums.

B. Unbounded Floating-Point Numbers

The notion of unbounded floating-point number is somewhat simple, and is in fact not original: it is commonly used in the literature on numerical programs [54] and also in advanced formalization such as Flocq [22]. Roughly speaking, unbounded floating-point number are very much like standard IEEE floating-point numbers, except that their exponent can be arbitrarily large: it is a value $\pm m \times 2^{e-p+1}$ where $0 \leq m \leq 2^p - 1$, $e_{\min} \leq e$ without upper bound on e. As a consequence, unlike standard floating-point numbers, the four basic operations on unbounded floating-point numbers *never overflow*. There is no need for special values for infinities to represent the result of unbounded floating-point operations. On the other hand, notice that unbounded floats include subnormal numbers. There is an injection from finite IEEE float numbers to unbounded float numbers. The properties (2), (3) and (4) indeed hold for unbounded floating-point numbers. This fact allows us to separate the proofs concerning functional behavior of numerical programs from the proof of absence of overflow: to prove a property on floating-numbers it suffices to prove the same on unbounded floats, and separately prove that each floating-point operation involved does not overflow nor produces NaN values.

C. Accuracy of Compound Summations

The compound sum of a vector $(a_1, \ldots, a_n)$ of floating-point numbers is the sum of all a_i. Defining it properly is more complex than the compound sum of real numbers because $\oplus$ is not associative. It is thus necessary to choose the order in which the additions are done. We make the choice to associate to the left, meaning that we define the compound sum from a_m (included) to a_k (excluded), denoted by $\bigoplus_{m \leq i < k} a_i$, by the following recursive equations.

$$\begin{aligned} \bigoplus_{m \leq i < k} a_i &= 0 \qquad \text{if } k \leq m \\ \bigoplus_{m \leq i < k} a_i &= \left(\bigoplus_{m \leq i < k-1} a_i \right) \oplus a_{k-1} \qquad \text{when } m < k \end{aligned}$$

Associating to the left is important because it impacts the final result of a sum. Yet the bounds we prove in the following are invariant by permutation of the element of the input vector. In other words, the same bounds could be proved when the sum is performed in any other order.

The following states a bound on compound sums, as a slight reformulation of a theorem by Jeannerod and Rump [44].

Theorem II.1 (Accuracy of compound sums)**.** *For any vector a of unbounded doubles, and any $m \leq n$:*

$$\left| \bigoplus_{m \leq i < n} a_i \; - \sum_{m \leq i < n} a_i \right| \leq (n - m - 1)\varepsilon \sum_{m \leq i < n} |a_i|$$

The proof of this theorem [23] is far for trivial and make a clever use of properties (2), (3) and (4). From the previous theorem, we deduce the following corollary, which is useful in the proofs we perform.

Corollary II.2 (Bound on sums)**.** *For any constant S and M_a, any vector a, any indices m and n such that $n - m \leq S$ and $|a_i| \leq M_a$ for any $m \leq i < n$ we have*

$$|\bigoplus_{m \leq i < n} a_i| \leq M_a \times S \times (1 + \varepsilon(S - 1))$$

D. Approximations of $\exp$ and $\log$

In implementations using floating-point numbers, not only the sum is subject to rounding, but also the computations of functions $\exp$ and $\log$. Here, we do not discuss any particular implementations of these two functions. Instead, we assume given implementations for them with given bounds in the rounding errors they perform. We do not want to rely, as Blanchard *et al.* [10] do, on perfectly rounded implementations of exponential and logarithm. Instead we assume we have implementations that are possibly less precise, the precision of them being specified as parameters.

Concerning exponential first, we assume given an implementation $\widehat{\exp}$ which satisfies the following property: for any real x such that $|x| \leq M_{\exp}$,

$$|\widehat{\exp}(x) - \exp(x)| \leq E_{\exp} \exp(x)$$

where $M_{\exp}$ and $E_{\exp}$ are two positive parameters. In the following we need to assume $E_{\exp} \leq 0.5$, a reasonable assumption, which in particular implies that $\widehat{\exp}(x)$ is always non-negative. Concerning logarithm we assume similarly an implementation satisfying the following property: for any real x such $0 < x \leq M_{\log}$

$$|\widehat{\log}(x) - \log(x)| \leq E_{\log} |\log(x)|$$

where $M_{\log}$ and $E_{\log}$ are positive parameters.

Notice that we do not claim that there exist implementations of approximations of exponential and logarithm, satisfying the properties above, for any value of the parameters $E_{\exp}$, $M_{\exp}$, $E_{\log}$ and $M_{\log}$. We just assume we are given some. Indeed it is known in the literature that such implementations exist for double precision, with a correct rounding, that is with $E_{\exp} = E_{\log} = \varepsilon$, $M_{\log} = \mathsf{maxf}$, and $M_{\exp}$ at most 708 (for larger values the exponential overflows): see for example Daramy *et al.* [29] and the implementations provided by the CORE-MATH project [56].

Notice also that we assume only some relative error ($E_{\exp}$ and $E_{\log}$) but no absolute error. For exponential, this is not needed because for an argument at least -708 the result is never a sub-normal. For a completely different reason, the logarithm do not need to return any sub-normal either, because the logarithm of the floating-point successor of 1 is around 2^{-52}, larger than a sub-normal too (and similar for the predecessor).

E. Accuracy of LSE

Our main result concerning the accuracy of the computation of $\widehat{\mathsf{LSE}}$ is given by Theorem II.5 below. To prove this theorem we need to establish first a few auxiliary lemmas. In these lemmas, we consider arbitrary positive constants A and B.

The first lemma is in fact a generalization of Theorem II.1 on the accuracy of compounds sums, when the input vector is itself subject to errors.

Lemma II.3 (Accuracy of sums, generalized)**.** *Given any vectors a and $\widehat{a}$ such that for all i, $|\widehat{a}_i - a_i| \leq A|a_i| + B$ we have:*

$$\left| \bigoplus_{0\leq i<n} \widehat{a}_i - \sum_{0\leq i<n} a_i \right| \leq (A + (n-1)\varepsilon(1+A)) \sum_{m\leq i<n} |a_i| + Bn\,(1 + (n-1)\varepsilon)$$

The next lemma is necessary to propagate errors bounds through the mathematical log.

Lemma II.4 (Error propagation for mathematical logarithm)**.** *For any positive real numbers x and $\widehat{x}$ such that $|\widehat{x} - x| \leq Ax$, with $A < 1$ we have:*

$$|\log \widehat{x} - \log x| \leq -\log(1-A)$$

These results are combined to get the final result we target, as follows.

Theorem II.5 (Accuracy of LSE)**.** *For any $n \geq 1$, and no larger than 2^{51}, any vector a of size n such that for all i, $|a_i| \leq M_a$ for some $M_a \leq M_{\exp}$, and assuming that*

$$\exp(M_a)(1 + E_{\exp})n(1 + \varepsilon(n-1)) \leq M_{\log} \qquad (5)$$

we have

$$\left|\widehat{\mathsf{LSE}}(a) - \mathsf{LSE}(a)\right| \leq E_{\log}|\mathsf{LSE}(a)| - \log\left(1 - (E_{\exp} + (n-1)\varepsilon(1+E_{\exp}))\right)(1 + E_{\log})$$

The hypothesis (5) above is required to call the $\widehat{\log}$ function on the proper interval of definition. A bound on the size of the input is needed to apply Lemma II.4 with $A = E_{\exp} + (n-1)\varepsilon(1+E_{\exp})$: to show that A is smaller than 1, together with the hypothesis $E_{\exp} \leq 0.5$, the bound 2^{51} on n suffices.

F. Discussion on the Variations of the Bound on Accuracy

The error bound of $\widehat{\mathsf{LSE}}$ has two parts :

- A relative part, which is $E_{\log}$
- A constant part :

  $$-\log\left(1 - (E_{\exp} + (n-1)\varepsilon(1+E_{\exp}))\right)(1 + E_{\log})$$

  We note that $-\log(1-x) < 2x$ for $x \leq \frac{1}{2}$. We can therefore bound the constant error by

  $$2 \times (E_{\exp} + (n-1)\varepsilon(1+E_{\exp}))\,(1 + E_{\log})$$

  The factors that dominate the constant bound are $E_{\exp}$ and $\varepsilon \times (n-1)$.

In Section IV, we compare this bound with the one proposed by Blanchard *et al.* [10].

The error bound grows linearly with $E_{\log}$, $E_{\exp}$ and n. Since it is possible to choose an implementation of $\widehat{\log}$ and $\widehat{\exp}$ with specific bounds, having the error bound of $\widehat{\mathsf{LSE}}$ depending on $E_{\exp}$ and $E_{\log}$ is useful in order to control the error.

To give some instances of the obtained bound, we can choose specific values for the parameters $E_{\log}$, $E_{\exp}$, $M_{\exp}$, $M_{\log}$, M_a and n. Let us assume reasonable bounds in practice on n and M_a that is $2^{10} = 1024$ and $M_a = 25$.

- Let us assume first we have some correctly rounded implementations of exponential and logarithm, that is $E_{\exp} = 2^{-53}$ and $E_{\log} = 2^{-53}$. Then, to ensure that hypothesis (5) of Theorem II.5 holds, it suffices to have $M_{\log}$ larger than

  $$\begin{aligned}&\exp(M_a)(1+E_{\exp})n(1+\varepsilon(n-1))\\ &\leq \exp(25)(1+2^{-53})2^{10}(1+2^{-53}\times 1023)\\ &\leq 7.38\times 10^{13}\end{aligned}$$

  Assuming thus that the implementation of $\widehat{\log}$ is correctly rounded on the domain given by the bound $M_{\log}$ above, the relative error on LSE is $E_{\log} = 2^{-53}$ and the absolute error is bounded by

  $$\begin{aligned}&2\times(E_{\exp} + (n-1)\varepsilon(1+E_{\exp}))\,(1+E_{\log})\\ &\leq 2\times\left(2^{-53} + 1023\times 2^{-53}(1+2^{53})\right)(1+2^{-53})\\ &\leq 2.28\times 10^{-13}\end{aligned}$$

- Let us assume less precise implementations of exponential and logarithm with $E_{\exp} = 2^{-40}$ and $E_{\log} = 2^{-36}$. These are some bounds for efficient implementations of exponential and logarithm that empirically seemed sufficiently accurate and energy-saving for industrial applications like IoT systems or deep-learning frameworks [38]. Then, to ensure that hypothesis (5) of Theorem II.5 holds, it suffices to have $M_{\log}$ larger than

  $$\begin{aligned}&\exp(M_a)(1+E_{\exp})n(1+\varepsilon(n-1))\\ &\leq \exp(25)(1+2^{-40})2^{10}(1+2^{-53}\times 1023)\\ &\leq 7.38\times 10^{13}\end{aligned}$$

  that is roughly the same bound as above with correct rounding on $\widehat{\exp}$ and $\widehat{\log}$. In other words, the required bound on the input domain of logarithm depends mostly on the bound on inputs and the number of elements in the input sequence. The relative error on the computation of LSE is now $E_{\log} = 2^{-36}$ and the absolute error is bounded by

  $$\begin{aligned}&2\times(E_{\exp} + (n-1)\varepsilon(1+E_{\exp}))\,(1+E_{\log})\\ &\leq 2\times\left(2^{-40} + 1023\times 2^{-53}(1+2^{53})\right)(1+2^{-36})\\ &\leq 2.05\times 10^{-12}\end{aligned}$$

  This absolute error is roughly twice the absolute error of the case with correct rounded implementations of exp and log.

Notice that if the size of the sequence is significantly larger than the assumed bound 1024, then the required value for $M_{\log}$ gets significantly larger, indeed it increases roughly linearly with this size.

```
(**  The type of unbounded floats in "double" format *)
type udouble

(** injection of udouble to real numbers *)
function to_real udouble : real

(** The rounding function *)
function uround mode real : udouble

constant eps:real = 0x1p-53 / (1.0 + 0x1p-53)
constant eta:real = 0x1p-1075

axiom uround_rne: forall x:real.
  abs (uround RNE x - to_real x) <= eps * abs x + eta

(** addition *)
function uadd (x y:udouble) : udouble =
  uround RNE (to_real x + to_real y)

(** properties (2), (3) and (4) *)
axiom add_rounding : forall x y:udouble.
  abs (to_real (uadd x y) - (to_real x + to_real y))
    <= abs (to_real x + to_real y) * eps

axiom add_bound_left: forall x y:udouble.
  abs (to_real (uadd x y) - (to_real x + to_real y))
    <= abs (to_real x)

axiom add_bound_right: forall x y:udouble.
  abs (to_real (uadd x y) - (to_real x + to_real y))
    <= abs (to_real y)
```

Fig. 1. Theory of unbounded doubles in WhyML (excerpt): conversion to real numbers, rounding, addition and its properties.

```
constant exp_max_value :real  (* constant M_exp *)
axiom exp_max_value_spec: 0.0 < exp_max_value

constant exp_error:real  (* constant E_exp *)
axiom exp_error_bound : 0.0 < exp_error <= 0.5

function u_exp (x:udouble) : udouble  (* function \widehat{exp} *)
axiom u_exp_spec : forall x:udouble.
  abs (to_real x) <= exp_max_value →
    abs (to_real (u_exp x) - exp (to_real x))
      <= exp (to_real x) * exp_error
```

Fig. 2. Declaration of $\widehat{\exp}$ in WhyML, with assumed accuracy.

```
1  let lemma lse_accuracy (a:int → udouble) (size:int) (max_a:real)
2    requires { 1 <= size }
3    requires { from_int (size - 1) <= 0x1p51 }
4    requires {
5      forall i. 0 <= i < size →
6        abs (to_real (a i)) <= max_a <= exp_max_value }
7    requires {
8      exp max_a * (1.0 + exp_error) *
9        from_int size * (1.0 + eps * from_int (size - 1))
10     <= log_max_value }
11   ensures {
12     let err = exp_error + eps * from_int (size - 1)
13               * (1.0 + exp_error) in
14     abs (to_real (u_lse a size) - lse_exact a size) <=
15       log_error * abs (lse_exact a size)
16       - log (1.0 - err) * (1.0 + log_error) }
```

Fig. 3. Statement of Theorem II.5 in WhyML.

III. Formalization of the Accuracy Result

In this section we show how we formalized the statement of Theorem II.5. We first summarize the methodology we followed. We then consider successively the formalization of LSE accuracy theorem in WhyML (Section III-B) and then a proof of a corresponding C code (Section III-C).

Our methodology makes use of the unbounded floating-point numbers introduced in Section II-B and heavily relies on WhyML. WhyML is the language of Why3 [11], a general-purpose environment for deductive verification. Why3 is used as an intermediate tool by several front-ends including Frama-C [46] for C code and by Spark for Ada code [49]. The Why3 environment allows the user to access a large set of different provers, including Coq and Gappa. Moreover, nowadays there are alternatives to the use of Coq for proving pure mathematical facts, including dReal [36] and Metitarski [1]. Concerning the reasoning on floating-point computation, Why3 gives access to SMT solver which support the SMT-LIB floating-point theory, such as CVC4 [7], cvc5 [6], Z3 [33] and Alt-Ergo-FPA [26]. But WhyML also proposes to the user a large set of techniques and tools to achieve complex proofs, for example via the use of *lemma functions*, which are, roughly speaking, a way to construct a proof by writing a program.

A. WhyML Formalization of Unbounded Floating-Point Numbers

A starting point of our formalization was to design a new WhyML theory for unbounded floats. An excerpt of that theory is given in Figure 1. In this theory, the type `udouble` is abstract, and only assumed to be given a function `to_real` which returns the real number represented by any `udouble`. The rounding function `uround` that rounds any real number to a `udouble` is also declared abstractly. The basic operations are defined as the rounding of the real operations. The properties (2), (3) and (4) are stated as axioms in the theory. To provide guarantees that this theory is consistent with IEEE floats, it is *realized* using Coq and its Flocq library.

B. Accuracy of LSE proved in WhyML

To start with, we need to declare the approximations of exp and log that we consider. The declaration of $\widehat{\exp}$ is shown in Figure 2 and the one of $\widehat{\log}$ is similar. These declarations are axiomatic so as to make them parametric in the values of $M_{\exp}$, $E_{\exp}$ and such. The definitions of `u_sum` for $\bigoplus$ and `u_lse` for $\widehat{\mathsf{LSE}}$ follows naturally.

The WhyML statement corresponding to Theorem II.5 is given in Figure 3. Preconditions on lines 2–3 express that the size of the array is between 1 and 2^{51}. The precondition on lines 4–6 expresses the bound on the array elements, and the precondition on lines 7–10 expresses Hypothesis (5). the post-condition on lines 11–16 expresses the bound on accuracy given by Theorem II.5. The text of WhyML proof of Theorem II.5 is given by the body of the `lse_accuracy` lemma function, displayed in Figure 4. It more or less follows the paper proof detailed in our report [23]. Notice on lines 3–12 the invocation of Lemma II.3, and on lines 36–42 the invocation of Lemma II.4.

```
1  let ghost s : udouble = u_sum (u_exp_fun a) 0 size in
2  let ghost sum_exps : real = sum (exp_fun a) 0 size in
3  begin
4    (* statement corresponding to Lemma (II.3) *)
5    ensures {
6      abs ((to_real s) - sum_exps) <=
7        sum_exps * (exp_error +
8          eps * from_int (size - 1) * (1.0 + exp_error)) }
9    (* invocation of Lemma (II.3) proved earlier *)
10   u_sum_accuracy_combine_pos
11     exp_error 0.0 (exp_fun a) (u_exp_fun a) 0 size;
12 end;
13 begin
14   ensures { sum_exps > 0.0 }
15   sum_strictly_pos (exp_fun a) 0 size;
16 end;
17 begin (* required domain for calling u_log *)
18   ensures { 0.0 < to_real s <= log_max_value }
19   assert { forall i. 0 <= i < size ->
20     0.0 <= to_real (u_exp (a i)) <=
21       exp max_a * (1.0 + exp_error)
22     by abs (to_real (u_exp (a i)) - exp (to_real (a i)))
23          <= exp (to_real (a i)) * exp_error
24     so to_real (u_exp (a i)) <=
25       exp (to_real (a i)) * (1.0 + exp_error) };
26   (* invocation of Corollary (II.2) *)
27   u_sum_constant_bounds (exp max_a *
28     (1.0 + exp_error)) (u_exp_fun a) size 0 size;
29 end;
30 let ghost r : udouble = u_log s in
31 assert { r = u_lse a size };
32 let ghost err : real = exp_error +
33        eps * from_int (size - 1) * (1.0 + exp_error)
34 in
35 assert { err < 1.0 };
36 begin
37   ensures {
38     abs (log (to_real s) - log sum_exps) <=
39       - log (1.0 - err) }
40   (* invocation of Lemma (II.4) on log *)
41   log_combine_err sum_exps (to_real s) err 0.0;
42 end;
43 assert {
44   (log_error + 1.0) *
45     (abs (log (to_real s) - log sum_exps)) <=
46     - log (1.0 - err) * (log_error + 1.0)
47   by (log_error + 1.0 >= 0.0) };
48 assert {
49   abs (to_real r - lse_exact a size)
50   <= abs (to_real r - log (to_real s)) +
51       abs (log (to_real s) - log sum_exps)
52   <= (log_error + 1.0) *
53         (abs (log (to_real s) - log sum_exps))
54         + log_error * abs (lse_exact a size)
55   <= log_error * abs (lse_exact a size)
56         - log (1.0 - err) * (log_error + 1.0) }
```

Fig. 4. Proof of Theorem II.5 in WhyML.

To proceed with the proof, we ask Why3 to generate a set of verification conditions (VCs for short). On this lemma function, Why3 generates 30 VCs. All of them except one are proved by the Alt-Ergo SMT solver within a 5 seconds time limit. The only remaining one corresponds to the formula `0 < to_real s` on line 18 of Figure 4, which can be proved instead using the FPA variant of Alt-Ergo [26]. See our report [23] for more technical details on the proofs.

C. Proving a C code implementing LSE

We aim to achieve proofs on concrete C code. For that, we use the environment TIS-kernel, a fork of Frama-C, and its J^3 plug-in for deductive verification, which is a prototype

```
/*@ requires 0 < size <= 1024 && max_a <= exp_max_value;
  @ requires \initialized (&a[0..size-1]);
  @ requires
  @   \forall integer i;
  @     0 <= i < size ==> \abs(a[i]) <= max_a;
  @ // the hypothesis (5) of Theorem II.5
  @ requires
  @   \exp(max_a) * (1.0 + exp_error) * size *
  @           (1.0 + (eps * (size - 1))) <= log_max_value;
  @ // additional requirements to prevent
  @ // overflow on addition
  @ requires max_a <= 701.0;
  @ requires log_max_value <= 0x1p1023;
  @ // result is equal to the WhyML def of LSE on udouble
  @ ensures to_udouble(\result) == u_lse(a, size);
  @ // the accuracy property
  @ ensures \abs(\result - lse_exact(a, size)) <=
  @   log_error * \abs(lse_exact(a,size))
  @   - \log(1 - (exp_error + eps * (size - 1) *
  @             (1 + exp_error))) * (1 + log_error);
  @*/
double log_sum_exp(size_t size) {
 int i;
 double s = 0.0;
 /*@ loop invariant 0 <= i <= size;
   @ loop invariant // to prove the first post-condition
   @    to_udouble(s) == u_sum_of_u_exp(a, 0, i);
   @ // for proving s is the domain of the log
   @ loop invariant (i == 0 ? s == 0.0 : 0.0 < s);
   @ loop invariant
   @   \forall integer j; 0 <= j < i ==>
   @     \abs(to_real(u_exp(to_udouble(a[j])))) <=
   @       \exp(max_a) * (1.0 + exp_error) ;
   @ loop assigns i, s;
   @ loop variant (size - i);
   @*/
 for (i = 0; i < size; i++) {
  /*@ assert 0.0 <= to_real(u_exp(to_udouble(a[i]))) ;
    @ assert to_real(to_udouble(a[i])) <= max_a ;
    @ assert
    @   \exp(to_real(to_udouble(a[i]))) <= \exp(max_a) ;
    @ assert
    @   \abs(to_real(u_exp(to_udouble(a[i])))
    @          - \exp(to_real(to_udouble(a[i]))))
    @      <= \exp(max_a) * exp_error ;
    @ assert
    @   to_real(u_exp(to_udouble(a[i])))
    @     <= \exp(max_a) * (1.0+exp_error) ;
    @ assert   // invocation of Corollary (II.2)
    @   usum_double_bound(u_sum_of_u_exp(a, 0, i),
    @     \exp(max_a) * (1.0 + exp_error), size);
    @*/
  s += exp_approx(a[i]);
 }
 /*@ assert // another invocation of Corollary (II.2)
   @  usum_double_bound(to_udouble(s),
   @    \exp(max_a) * (1.0 + exp_error), size);
   @*/
 return log_approx(s);
}
```

Fig. 5. C code for computing LSE, annotated with ACSL specifications.

under development. Alternatively, there should be no technical difficulty to achieve the proofs of our C code using the regular Frama-C environment and its Wp plug-in for deductive verification.

Our C code computing the LSE function is given on Figure 5. In a first step, let's ignore the potential floating-point overflow, and focus on proving the accuracy property. To specify the intended behavior and its properties, we build a bridge to WhyML definitions (see our report [23] for technical details), so that for example we can use the WhyML definitions

```
/*@ requires \abs(x) <= exp_max_value;
  @ ensures to_udouble(\result) == u_exp(to_udouble(x));
  @ assigns \nothing;
  @*/
extern double exp_approx(double x);

/*@ requires 0 < x <= log_max_value;
  @ ensures to_udouble(\result) == u_log(to_udouble(x));
  @ assigns \nothing;
  @*/
extern double log_approx(double x);
```

Fig. 6. External C functions for $\widehat{\exp}$ and $\widehat{\log}$, specified in ACSL.

of `u_exp` and `u_log` in the ACSL annotations. It provides in particular a function `to_udouble` that promotes a regular C double to an unbounded double. We declare and specify the auxiliary C functions for computing approximations of $\exp$ and $\log$, as shown on Figure 6.

The first post-condition, on lines 14–15 of Figure 5, thus expresses that the result of the C function is equal to the LSE function defined in WhyML. The second post-condition, on lines 16–20 of Figure 5, expresses the expected bounding property, as stated by Theorem II.5. The second post-condition is going to be proved easily from the first one, and the accuracy result on `u_lse` already proved in WhyML. The precondition on lines 3–5 is required to allow calling the exponential inside its correct domain. The precondition on lines 6–9 expresses the required hypothesis 5 of Theorem II.5.

The first post-condition on lines 14–15 is an easy consequence of the definition of the LSE function, and the loop invariant given on lines 26–27. The post-condition on lines 16–20 is proved by invoking the proof of the same statement already done in WhyML. Together with the simple loop invariants on lines 25 and 29, all the VCs are proved, in particular the expected post-conditions, except two of them. The first unproved VC is related to line 59 where it is required to show that `s` fits in the expected domain of the approximated logarithm. The second unproved VC is related to line 53 where it is requires to show the absence of floating-point overflow when performing addition.

To prove the VC on line 59 and thus prove that the sum `s` on line 59 fits in the expected range of the log, we need to state the additional loop invariants on lines 29 and 30–33 to bound the sum. To prove that these invariants hold, we need again to invoke Corollary II.2. Achieving this proof is a bit involved, requiring all the extra intermediate assertions on lines 38–51.

The last VC remaining to prove is the absence of numerical overflow when computing the addition on line 53. It is indeed expected since the C code operates on true IEEE floating-point numbers and not the unbounded ones. To achieve this, the given pre-conditions are not enough, we need to assume extra bounds on the inputs. So far we assumed the input numbers smaller than $M_{\exp}$, for which no upper bound is assumed so far. Yet, we add the exponentials of these numbers, and summing up to say 1024 of these numbers, we can indeed have an overflow. A tighter bound must be assumed. We assume here, on lines 12–13 of Figure 5, that M_a is smaller than 701 and $M_{\log}$ is smaller than 2^{1023}. With these extra assumptions, and thanks to the already stated and proved loop invariants on lines 29 and 30–33, the VC is proved.

In all, 56 VCs are generated. 49 of them are proved by Alt-Ergo, within a 5 seconds time limit. For the rest, we tried CVC4 and cvc5, which are able to solve 6 VCs, and the last one remaining is proved by the FPA variant of Alt-Ergo.

IV. Related work

As far as we know, the only contribution focusing on rounding errors of LSE-based algorithms is the work of Blanchard *et al.* [10]. They bound the rounding errors of the LSE function and its gradient, namely the softmax function. In this work, the authors assume that the exponential and logarithm functions are implemented with correct rounding, i.e., $E_{\exp} = E_{\log} = \varepsilon$. In contrast, we provide a proof which is parameterized with arbitrary error bounds for the called functions. It means that we can rely on any implementations of these functions without invalidating the bounds. Going back to Theorem II.5, if we take $E_{\exp} = \varepsilon$, we get a bound in which the relative error term is $E_{\log} = \varepsilon$ and the constant term is $-\log(1-(\varepsilon+(n-1)\varepsilon(1+\varepsilon)))\,(1+\varepsilon)$, that is, about $-\log(1-n\varepsilon)+O(u^2)$. Blanchard *et al.*'s relative error term is identical. Their constant term is about $(n+1)\varepsilon+O(u^2)$, which is actually very slightly tighter than ours, but of comparable order of magnitude. Yet, a strength of our accuracy result is that is parametric in the accuracies of $\exp$ and $\log$, instead of assuming ideally precise implementations. Moreover, another main strength compared to this work is the fact that we made formal proofs.

Our work can also be compared to other contributions targeting the end-to-end formal proof of numerical software written in C. For instance, Appel and Bertot [3] have combined the Verifiable Software Toolchain (VST) [2], [4], Flocq [21], [22] and Gappa [30] to formally-verify an example of square root implementation using the Newton method. VST ensures the correctness of the C code, while Flocq and Gappa are used as backend tools to check numerical accuracy facts. Kellison *et al.* [45] propose a Coq formal proofs library, called *LAProof*, for rounding error analysis of basic linear algebra operations, e.g. inner product or matrix-matrix multiplication. As an application example, the authors prove a C program computing a sparse matrix-vector multiplication using the VST [2] approach and the LAProof library.

Boldo *et al.* [13], [15] formalized a numerical integration scheme for a wave partial differential equation in Coq. They not only formally proved a bound on the mathematical errors [15], but also a bound on the rounding errors [13]. These works have been used as a basis for the formal verification of a wave equation resolution C program [16], based on the the Jessie plug-in of Frama-C. The most complex proof obligations related to numerical errors are discharged to Coq.

Becker *et al.* [8] developed a CakeML extension for optimizing floating-point arithmetic in Standard ML. Their approach relies on an end-to-end soundness proof linking a real-number specification of the initial code with the code obtained after optimization. The approach is entirely automated (code and proof generation) and targets the optimization of floating-point kernels, which are essentially blocks of floating-point computations free of control flow instructions. The roundoff errors are obtained and proved by using the prover FloVer (interval-based prover in HOL4).

V. Conclusion and future work

We presented bounds on the accuracy of an implementation of the LSE function. The resulting expressions for the bounds are parametric in the precision of the underlying implementations of $\exp$ and $\log$, and also parameterized by bounds on the size of the argument vectors, and bounds on the values of the vectors components. The given bounds are proved on paper and then in WhyML, using Why3 constructs such as lemma function to provide proofs that follow more or less the paper proofs. We also proved some C implementation by reusing the results proved in WhyML. The proofs are made simpler by using a theory of unbounded floating-point numbers, allowing us to separate the reasoning on accuracy from the reasoning on absence of overflow.

a) Future work.: The bounds exhibited by Blanchard *et al.* [10] show that naive implementations of these functions are relatively well-behaved regarding numerical accuracy, but prone to spurious overflow. The authors then study an alternative implementation to bypass this issue. The principle is to find the maximal value $\alpha = \max(x_i)$ among the components of the input vector and to rewrite the LSE expression as follows:

$$\mathsf{LSE}(x) = \alpha + \log\left(\sum_{1\leq i\leq n} \exp(x_i - \alpha)\right)$$

As for all i, $x_i - \alpha \leq 0$, the exponential takes a reasonable value whatever is the magnitude of the components of x, therefore, the risk of overflow is limited. They also prove that the accuracy of this alternative evaluation is not only as good as with the standard evaluation, but even slightly better. In practice, most applications using the LSE function rely on the shifted version. The proof of the error bounds associated to this alternative evaluation which is presented by Blanchard *et al.* uses rather sophisticated mathematical arguments, *e.g.* Taylor series expansions of the $\log(1+x)$ quantity. Providing a formal proof of this result could be an interesting perspective.

In the presented work, some parts of the proof are performed with a high level of automation. Making formal verification processes automatic and push-button has a strong impact on their industrial applicability. While the studied example is rather intricate and would be difficult to fully automate, there are plenty of applicative source code in which simple combinations of mathematical functions calls appear. For instance, we could try to apply our methodology on benchmarks from the FPBench [27] or COPRIN projects [51]. Most examples from these benchmarks are loop-free which strongly eases the verification process. We could try to handle these examples in a fully automatic way, providing bounds depending on error bounds for the mathematical functions implementations appearing in the source code.

For now, we assume error bounds on the implementations of the mathematical functions $\log$ and $\exp$. Our formally proved bounds apply only when implementations satisfying the assumptions are provided. It is known in the literature that correctly-rounded implementations of these functions can be achieved, *e.g.* in the recent CORE-MATH library [56]. To complement our work, we envision the formal verification of the errors induced by such implementations. In the late 20th century, Harrison [39], [40] formally verified implementations of specific floating-point exponential and trigonometric functions implementations in HOL, but the proofs were *ad hoc*, low-level and far from automatic. More recently, the Gappa tool [30] has been partly used to bound rounding errors of floating-point implementations of functions from the CR-LIBM library [28], [32], [31]. However, proofs are not fully automatic and are devoted to specific implementations. In addition, these works only focus on rounding errors, without taking the mathematical approximation errors into account. Geneau de Lamarlière *et al.* [37] provide a methodology and tooling to ease the formal proofs of low-level floating-point components with a minimal user effort. Their approach relies on a framework for modeling and reasoning on floating-point expressions with some facilities, without neglecting potential exceptional behaviors. For that purpose, they offer tools in the Coq proof assistant to automate the proof of the absence of exceptional behaviors, so that the user can reason on real numbers representation of floating-point expressions. This work is typically applied on mathematical functions implementations, *e.g.* $\exp$ and $\log$. Combining our approach with their methodology would be valuable to complete the toolchain. A longer-term goal could be the development of a formally verified implementation synthesis tool, in the spirit of the Metalibm tools [47], [25]. Metalibm already enables the generation of Gappa scripts certifying the synthesized implementations, but this feature is limited to some pieces of code.

Our work is not limited to the LSE function. We already applied our methodology to others, including the following extension related to computing mutual information:

$$\mathsf{SLSE}(a) = \sum_{0\leq i<n} \log_2\left(\sum_{0\leq j<n} \exp\left(-\frac{(a_i+\rho-a_j)^2}{2}\right)\right)$$

Some bounds on accuracy are already obtained on paper [23]. Yet, we did not yet satisfactorily achieve a formal proof. We identified remaining issues in our proof methodology, that deserve future work. In particular, the formal proofs that we already made on SLSE require a large amount of manual steps, so it is desirable to automate the process. We currently plan to automate the application of so-called "forward propagation lemmas", which are properties similar to our Lemma II.4 for logarithm, but applied to additions, multiplications, exponential. We believe that automating the application of such lemmas would naturally be reused for proving any code proceeding by composing numerical functions and operations. Another concern is related to the methodogy to deal directly with C code. Our current methodology is far from being usable by non-expert users. This is illustrated for example by the numerous assertions that we had to add in our C code for LSE, on lines 38–51 of Figure 5. There are constructs available in WhyML that would be nice to have at the C level: we think in particular, on one hand, about arbitrary lambda-expressions, and on the other hand the ability to call ghost functions. In fact, ghost functions are in principle present in ACSL [9], but they are limited to ghost C programs, whereas we would need to have ghost *logic* functions that would accept *logic types* are parameters: these include real numbers, unbounded floats, functions (lambda-expressions), etc.

References

[1] Behzad Akbarpour and Lawrence C. Paulson. Metitarski: An automatic theorem prover for real-valued special functions. *Journal of Automated Reasoning*, 44(3):175–205, 2010. Tool page at http://www.cl.cam.ac.uk/~lp15/papers/Arith/. doi:10.1007/s10817-009-9149-2.

[2] Andrew Appel. Verified software toolchain. In *European Symposium on Programming*, volume 6602 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2011. Tool page at https://vst.cs.princeton.edu/. doi:10.5555/1987211.1987212.

[3] Andrew Appel and Yves Bertot. C-language floating-point proofs layered with VST and Flocq. *Journal of Formalized Reasoning*, 13(1):1–16, 2020. URL: https://inria.hal.science/hal-03130704/.

[4] Andrew Appel and Ariel Kellison. Vcfloat2: Floating-point error analysis in coq. In *Proceedings of the 13th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 14–29, 2024.

[5] Ali Ayad and Claude Marché. Multi-prover verification of floating-point programs. In Jürgen Giesl and Reiner Hähnle, editors, *Fifth International Joint Conference on Automated Reasoning*, volume 6173 of *Lecture Notes in Artificial Intelligence*, pages 127–141, Edinburgh, Scotland, July 2010. Springer. URL: http://hal.inria.fr/inria-00534333.

[6] Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. cvc5: A versatile and industrial-strength SMT solver. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 13243 of *Lecture Notes in Computer Science*, pages 415–442. Springer, 2022. doi:10.1007/978-3-030-99524-9_24.

[7] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *Computer Aided Verification*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, 2011. http://cvc4.cs.stanford.edu/web/. doi:10.1007/978-3-642-22110-1_14.

[8] Heiko Becker, Robert Rabe, Eva Darulova, Magnus O. Myreen, Zachary Tatlock, Ramana Kumar, Yong Kiam Tan, and Anthony C. J. Fox. Verified compilation and optimization of floating-point programs in cakeml. In Karim Ali and Jan Vitek, editors, *36th European Conference on Object-Oriented Programming, ECOOP 2022, June 6-10, 2022, Berlin, Germany*, volume 222 of *LIPIcs*, pages 1:1–1:28. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICS.ECOOP.2022.1.

[9] Allan Blanchard, Claude Marché, and Virgile Prevosto. *Guide to Software Verification with Frama-C — Core Components, Usages, and Applications*, chapter Formally Expressing what a Program Should Do: the ACSL Language. Springer-Verlag, 2024. URL: https://inria.hal.science/hal-04265707.

[10] Pierre Blanchard, Desmond J Higham, and Nicholas J Higham. Accurately computing the log-sum-exp and softmax functions. *IMA Journal of Numerical Analysis*, 41(4):2311–2330, 2021. doi:10.1093/imanum/draa038.

[11] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Let's verify this with Why3. *International Journal on Software Tools for Technology Transfer (STTT)*, 17(6):709–727, 2015. See also http://toccata.gitlabpages.inria.fr/toccata/gallery/fm2012comp.en.html. URL: http://hal.inria.fr/hal-00967132/en, doi:10.1007/s10009-014-0314-5.

[12] Sylvie Boldo. Floats & Ropes: a case study for formal numerical program verification. In *36th International Colloquium on Automata, Languages and Programming*, volume 5556 of *Lecture Notes in Computer Science - ARCoSS*, pages 91–102, Rhodos, Greece, July 2009. Springer. doi:10.1007/978-3-642-02930-1_8.

[13] Sylvie Boldo. Floats and ropes: A case study for formal numerical program verification. In *36th International Colloquium on Automata, Languages and Programming*, volume 5556 of *Lecture Notes in Computer Science*, pages 91–102. Springer, 2009. doi:10.1007/978-3-642-02930-1_8.

[14] Sylvie Boldo. *Deductive formal verification: how to make your floating-point programs behave*. PhD thesis, Université Paris-Sud, 2014.

[15] Sylvie Boldo, François Clément, Jean-Christophe Filliâtre, Micaela Mayero, Guillaume Melquiond, and Pierre Weis. Formal proof of a wave equation resolution scheme: the method error. In *Interactive Theorem Proving*, volume 6172 of *Lecture Notes in Computer Science*, pages 147–162. Springer, 2010. URL: http://hal.inria.fr/inria-00450789/en, doi:10.1007/978-3-642-14052-5_12/.

[16] Sylvie Boldo, François Clément, Jean-Christophe Filliâtre, Micaela Mayero, Guillaume Melquiond, and Pierre Weis. Wave equation numerical resolution: a comprehensive mechanized proof of a C program. *Journal of Automated Reasoning*, 50(4):423–456, April 2013. URL: http://hal.inria.fr/hal-00649240/en/, doi:10.1007/s10817-012-9255-4.

[17] Sylvie Boldo and Jean-Christophe Filliâtre. Formal verification of floating-point programs. In *18th IEEE International Symposium on Computer Arithmetic*, pages 187–194, 2007. URL: https://usr.lmf.cnrs.fr/~jcf/publis/caduceus-floats.pdf, doi:10.1109/ARITH.2007.20.

[18] Sylvie Boldo, Claude-Pierre Jeannerod, Guillaume Melquiond, and Jean-Michel Muller. Floating-point arithmetic. *Acta Numerica*, 32:203–290, 2023. URL: https://hal.science/hal-04095151, doi:10.1017/S0962492922000101.

[19] Sylvie Boldo and Claude Marché. Formal verification of numerical programs: from C annotated programs to mechanical proofs. *Mathematics in Computer Science*, 5:377–393, 2011. URL: http://hal.inria.fr/hal-00777605, doi:10.1007/s11786-011-0099-9.

[20] Sylvie Boldo and Claude Marché. Toccata gallery of verified programs, section "floating-point computations". https://toccata.gitlabpages.inria.fr/toccata/gallery/fp.en.html, 2023.

[21] Sylvie Boldo and Guillaume Melquiond. *Computer Arithmetic and Formal Proofs: Verifying Floating-point Algorithms with the Coq System*. ISTE Press - Elsevier, December 2017. URL: https://hal.inria.fr/hal-01632617.

[22] Sylvie Boldo and Guillaume Melquiond. Some formal tools for computer arithmetic: Flocq and Gappa. In Mioara Joldes and Fabrizio Lamberti, editors, *28th IEEE International Symposium on Computer Arithmetic*, 2021. URL: https://hal.inria.fr/hal-03233227.

[23] Paul Bonnot, Benoît Boyer, Florian Faissole, Claude Marché, and Raphaël Rieu-Helft. Formally verified bounds on rounding errors in concrete implementations of logarithm-sum-exponential functions. Research Report 9531, Inria, 2023. URL: https://inria.hal.science/hal-04343157.

[24] Sven Brüggemann and Corrado Possieri. On the use of difference of log-sum-exp neural networks to solve data-driven model predictive control tracking problems. *IEEE Control Systems Letters*, 5(4):1267–1272, 2020. doi:10.1109/LCSYS.2020.3032083.

[25] Nicolas Brunie, Christoph Lauter, and Guillaume Revy. Precision adaptation for fast and accurate polynomial evaluation generation. In *30th International Conference on Application-specific Systems, Architectures and Processors*, volume 2160-052X, pages 41–41. IEEE, 2019. doi:10.1109/ASAP.2019.00-32.

[26] Sylvain Conchon, Mohamed Iguernlala, Kailiang Ji, Guillaume Melquiond, and Clément Fumex. A three-tier strategy for reasoning about floating-point numbers in SMT. In *Computer Aided Verification*, volume 10427 of *Lecture Notes in Computer Science*, pages 419–435, 2017. URL: https://hal.inria.fr/hal-01522770, doi:10.1007/978-3-319-63390-9_22.

[27] Nasrine Damouche, Matthieu Martel, Pavel Panchekha, Chen Qiu, Alexander Sanchez-Stern, and Zachary Tatlock. Toward a standard

benchmark format and suite for floating-point analysis. In *Numerical Software Verification*, pages 63–77. Springer, 2017. doi:10.1007/978-3-319-54292-8_6.

[28] Catherine Daramy, David Defour, Florent de Dinechin, and Jean-Michel Muller. Cr-libm: a correctly rounded elementary function library. In *Advanced Signal Processing Algorithms, Architectures, and Implementations XIII*, volume 5205, pages 458–464. SPIE, 2003.

[29] Catherine Daramy-Loirat, David Defour, Florent de Dinechin, Matthieu Gallet, Nicolas Gast, Christoph Lauter, and Jean-Michel Muller. CR-LIBM: a library of correctly rounded elementary functions in double-precision. Research report, LIP, 2006. URL: https://ens-lyon.hal.science/ensl-01529804.

[30] Marc Daumas and Guillaume Melquiond. Certification of bounds on expressions involving rounded operators. *ACM Transactions on Mathematical Software (TOMS)*, 37(1):1–20, 2010.

[31] Florent De Dinechin, Christoph Lauter, and Guillaume Melquiond. Certifying the floating-point implementation of an elementary function using gappa. *IEEE Transactions on Computers*, 60(2):242–253, 2010.

[32] Florent De Dinechin, Christoph Quirin Lauter, and Guillaume Melquiond. Assisted verification of elementary functions using gappa. In *Proceedings of the 2006 ACM symposium on Applied computing*, pages 1318–1322, 2006.

[33] Leonardo de Moura and Nikolaj Bjørner. Z3, an efficient SMT solver. In *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008. doi:10.1007/978-3-540-78800-3_24.

[34] Clément Fumex, Claude Marché, and Yannick Moy. Automating the verification of floating-point programs. In Andrei Paskevich and Thomas Wies, editors, *Verified Software: Theories, Tools, and Experiments. Revised Selected Papers Presented at the 9th International Conference VSTTE*, number 10712 in Lecture Notes in Computer Science, Heidelberg, Germany, December 2017. Springer. URL: https://hal.inria.fr/hal-01534533/.

[35] Bolin Gao and Lacra Pavel. On the properties of the softmax function with application in game theory and reinforcement learning. *arXiv preprint arXiv:1704.00805*, 2017.

[36] Sicun Gao, Jeremy Avigad, and Edmund M. Clarke. δ-complete decision procedures for satisfiability over the reals. In Bernhard Gramlich, Dale Miller, and Uli Sattler, editors, *Automated Reasoning*, pages 286–300. Springer, 2012. doi:978-3-642-31365-3_23.

[37] Paul Geneau de Lamarlière, Guillaume Melquiond, and Florian Faissole. Slimmer formal proofs for mathematical libraries. In Theo Drane and Anastasia Volkova, editors, *Proceedings of the 30th IEEE International Symposium on Computer Arithmetic*, Portland, OR, USA, September 2023.

[38] Cédric Gernigon, Silviu-Ioan Filip, Olivier Sentieys, Clément Coggiola, and Mickaël Bruno. Low-precision floating-point for efficient on-board deep neural network processing, 2023. arXiv:2311.11172.

[39] John Harrison. Floating point verification in hol light: the exponential function. In *International Conference on Algebraic Methodology and Software Technology*, pages 246–260. Springer, 1997.

[40] John Harrison. Formal verification of floating point trigonometric functions. In *International conference on formal methods in computer-aided design*, pages 254–270. Springer, 2000.

[41] Nicholas J Higham. *Accuracy and stability of numerical algorithms*. SIAM, 2002. doi:10.1137/1.9780898718027.

[42] Taocheng Hu and Jinhui Yu. LogSumExp for unlabeled data processing. In *15th International Conference on Software Engineering Research, Management and Applications*, pages 63–69. IEEE, 2017. doi:10.1109/SERA.2017.7965708.

[43] IEEE standard for floating-point arithmetic, 2008. https://dx.doi.org/10.1109/IEEESTD.2008.4610935. doi:10.1109/IEEESTD.2008.4610935.

[44] Claude-Pierre Jeannerod and Siegfried M. Rump. On relative errors of floating-point operations: optimal bounds and applications. *Mathematics of Computation*, 87:803–819, 2018. URL: https://hal.inria.fr/hal-00934443, doi:10.1090/mcom/3234.

[45] Ariel E. Kellison, Andrew W. Appel, Mohit Tekriwal, and David S. Bindel. LAProof: A library of formal proofs of accuracy and correctness for linear algebra programs. In *Proceedings of the 30th IEEE International Symposium on Computer Arithhmetic (ARITH)*, September 2023. To appear.

[46] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c: A software analysis perspective. *Formal Aspects of Computing*, 27(3):573–609, May 2015. doi:10.1007/s00165-014-0326-7.

[47] Olga Kupriianova and Christoph Lauter. Metalibm: A mathematical functions code generator. In *International Congress on Mathematical Software*, volume 8592 of *Lecture Notes in Computer Science*, pages 713–717. Springer, 2014. doi:10.1007/978-3-662-44199-2_106.

[48] Radek Mackowiak, Lynton Ardizzone, Ullrich Kothe, and Carsten Rother. Generative classifiers as a basis for trustworthy image classification. In *Computer Vision and Pattern Recognition*, pages 2971–2981, 2021. doi:10.1109/CVPR46437.2021.00299.

[49] John W. McCormick and Peter C. Chapin. *Building High Integrity Applications with SPARK*. Cambridge University Press, 2015. doi:10.1017/CBO9781139629294.

[50] Guillaume Melquiond. *Formal Verification for Numerical Computations, and the Other Way Around*. Habilitation à diriger des recherches, Université Paris Sud, April 2019. URL: https://tel.archives-ouvertes.fr/tel-02194683.

[51] Jean-Pierre Merlet. *Parallel Robots*, chapter Structural synthesis and architectures, pages 19–94. Springer, 2006. doi:10.1007/1-4020-4133-0_2.

[52] Taiki Miyagawa and Akinori F Ebihara. The power of log-sum-exp: Sequential density ratio matrix estimation for speed-accuracy optimization. In *International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 7792–7804, 2021. URL: https://proceedings.mlr.press/v139/miyagawa21a.html.

[53] Jean-Michel Muller, Nicolas Brisebarre, Florent De Dinechin, Claude-Pierre Jeannerod, Vincent Lefevre, Guillaume Melquiond, Nathalie Revol, Damien Stehlé, Serge Torres, et al. *Handbook of floating-point arithmetic*. Springer, 2018.

[54] Jean-Michel Muller, Nicolas Brunie, Florent de Dinechin, Claude-Pierre Jeannerod, Mioara Joldes, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, and Serge Torres. *Handbook of Floating-point Arithmetic (2nd edition)*. Birkhäuser Basel, July 2018. URL: https://hal.inria.fr/hal-01766584, doi:10.1007/978-3-319-76526-6.

[55] Pierre Roux. Formal Proofs of Rounding Error Bounds - With Application to an Automatic Positive Definiteness Check. *Journal of Automated Reasoning*, 57(2):135–156, 2016. doi:10.1007/s10817-015-9339-z.

[56] Alexei Sibidanov, Paul Zimmermann, and Stéphane Glondu. The CORE-MATH project. In *29th IEEE Symposium on Computer Arithmetic*, pages 26–34, 2022. URL: https://inria.hal.science/hal-03721525, doi:10.1109/ARITH54963.2022.00014.

[57] US Government Accountability Office. Defense patriot missile: Software problem led to system failure at dhahran, saudi arabia. *US Government Accountability Office Reports, rapport no. GAO/IMTEC-92-26*, 1992.