Formally Verified Bounds on Rounding Errors in Concrete Implementations of Logarithm-Sum-Exponential Functions

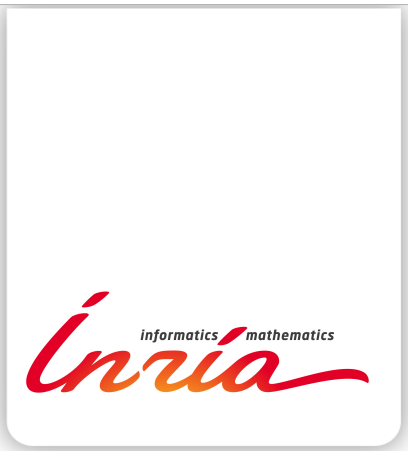
Paul Bonnot, Benoît Boyer, Florian Faissole, Claude Marché, Raphaël Rieu-Helft

Formally Verified Bounds on Rounding Errors in Concrete Implementations of Logarithm-Sum-Exponential Functions

Paul Bonnot, Benoît Boyer, Florian Faissole, Claude Marché, Raphaël Rieu-Helft

RESEARCH REPORT
N° 9531
December 2023
Project-Team Toccata

ISSN 0249-6399 ISRN INRIA/RR--9531--FR+ENG



Formally Verified Bounds on Rounding Errors in Concrete Implementations of Logarithm-Sum-Exponential Functions*

Paul Bonnot†, Benoît Boyer‡, Florian Faissole‡, Claude Marché†,

Raphaël Rieu-Helft§

Project-Team Toccata

Research Report n° 9531 — December 2023 — 50 pages

Abstract: We study the numerical accuracy of some specific computer programs performing numerical computations. Such a numerical accuracy is expressed in terms of bounds on the difference between the floating-point computations and the corresponding rounding-free computation using mathematical real numbers. We do not only seek to discover such bounds "on paper" but we aim at obtaining computer-assisted formal proofs that these bounds are correct for any possible inputs. The functions we study come from the domain of machine learning: first the function computing the logarithm of the sum of exponentials of a sequence, second an algorithm on which is based the so-called mutual information function from probability theory. The final bounds obtained are parameterised by the error bounds of the underlying implementations of the logarithm and exponential functions. We propose an original methodology to conduct our formal proofs, using a combination of the Why3 environment for deductive verification, a original modelling of floating-point computations using *unbounded* numbers, and finally the recently available J^3 prototype for proving properties directly on C source code.

Key-words: Formal specification, Deductive verification, Why3 environment for deductive verification, C program involving floating-point computations.

* This work has been partially supported by the bilateral contract ProofInUse-MERCE between Inria team Toccata and Mitsubishi Electric R&D Centre Europe, Rennes ; the bilateral contract ProofInUse-TrustInSoft between Inria team Toccata and the TrustInSoft company, Paris ; and by the ANR convention « SIF TrustInSoft TOCCATA A1 » for « préservation de l'emploi R&D ».

† 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

RESEARCH CENTRE
SACLAY – ÎLE-DE-FRANCE

1 rue Honoré d'Estienne d'Orves
Bâtiment Alan Turing
Campus de l'École Polytechnique
91120 Palaiseau

Bornes formellement vérifiées sur les erreurs d'arrondi dans les implémentations concrètes de fonctions logarithme-somme-exponentielles

Résumé : Nous étudions la précision numérique de certains programmes effectuant des calculs numériques. Une telle précision numérique s'exprime en termes de bornes sur la différence entre les calculs à virgule flottante et le calcul sans arrondi similaire utilisant les nombres réels mathématiques. Nous ne cherchons pas seulement à découvrir de telles bornes « sur le papier », mais nous visons à obtenir des preuves formelles, c'est-à-dire assistées par ordinateur, que ces bornes sont correctes pour toutes les entrées possibles des programmes. Les fonctions que nous étudions proviennent du domaine de l'apprentissage automatique : en premier lieu la fonction calculant le logarithme de la somme des exponentielles d'une séquence, en second lieu un algorithme sur lequel se base ce qu'on appelle la fonction d'information mutuelle issue de la théorie des probabilités. Les bornes que l'on propose sont paramétriques en fonction des bornes d'erreur des implémentations sous-jacentes du logarithme et de l'exponentielle. Nous proposons une méthodologie originale pour conduire nos preuves formelles, en utilisant un combinaison de l'environnement Why3 pour la vérification déductive, une modélisation originale des calculs à virgule flottante utilisant des nombres *non bornés*, et enfin le prototype J^3 récent pour prouver les propriétés directement sur le code source C.

Mots-clés : Spécification formelle, preuve de programmes, environnement Why3 pour la vérification déductive, programmes C en virgule flottante.

Ces recherches ont été partiellement financées par le contrat bilatéral ProofInUse-MERCE entre l'équipe Inria Toccata et Mitsubishi Electric R&D Centre Europe, à Rennes ; par le contrat bilatéral ProofInUse-TrustInSoft entre l'équipe Toccata et la PME TrustInSoft, à Paris ; et par la convention ANR « SIF TrustInSoft TOCCATA A1 » pour la préservation de l'emploi R&D.

Contents

List of Figures

1 Introduction

Software is involved in many industrial systems nowadays. In cyber-physical systems in a broad sense, the software that controls a system must perform numerical computations typically based on floating-point arithmetic. The floating-point representation of numbers, and the operations on them, are standardised by the IEEE-754 standard [38]. The *Handbook of Floating-Point Arithmetic* [48] presents a wide overview of the topic of computer arithmetic.

In this report we are interested in studying the numerical accuracy of some specific programs performing numerical computations. Such a numerical accuracy is expressed in terms of bounds on the difference between the floating-point computations and the corresponding rounding-free computation using mathematical real numbers. We do not only seek to discover such bounds "on paper" but we aim at formally proving (in the sense: with a computer-assisted proof) that these bounds are valid for any possible inputs. 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 Melquiond's Habilitation dissertation [45] and a survey by Boldo *et al.* [15].

1.1 The Numerical Programs Considered

The first numerical program we target is the *log-sum-exp* function, abbreviated as LSE. 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:

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

This function is frequently used in machine learning applications [20, 32, 47], for example statistical classifiers [37, 42], in which the function computing the maximum of a set of values is required. However, algorithms may behave wrongly with functions having non-differentiable points, legitimating the use of a smooth approximation of the maximum function. LSE is such a smooth approximation satisfying the following property

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

Blanchard *et al.* [9] showed that the rounding errors 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 2.1), therefore, they provide the best accuracy that can be achieved in the considered floating-point format. However, one may want to use less accurate but more efficient implementations, therefore, to determine bounds that are parameterised by the accuracies of exp and log.

An example in which a variant of the LSE function is used is the computation of *mutual information*, a key concept in probability theory. It provides a measure of the mutual dependence of two random variables, that is, it estimates the amount of information (for example a number of representation bits) that can be deduced on a given random variable by observing information on the other one. A typical application of mutual information is signal processing making use of error correcting code responsible for reducing the amount of errors due to potential noise in the analysed communications [23]. Our case study is a simplified expression of one of the base algorithm used in the computation of a discrete input computation of mutual information. As it corresponds to the summation of LSE functions applied to some compound vector, let us abbreviate it as SLSE and define it as follows (ρ being a real value, that we consider such that $0 \le \rho \le 1$ for simplification but that could take larger values in concrete applications).

```
/*@ requires \abs(x) <= 1.0;
  @ ensures \abs(\result - \exp(x)) <= 0x1p-4;
  @*/
double my_exp(double x) {
  /*@ assert \abs(0.9890365552 + 1.130258690*x +
    @             0.5540440796*x*x - \exp(x)) <= 0x0.FFFFp-4;
    @*/
  return 0.9890365552 + 1.130258690 * x + 0.5540440796 * x * x;
}
```

Figure 1: Approximation of the exponential by a polynomial of degree 2, on the interval $[-1;1]$. The chosen polynomial is obtained by the Remez method, computing the polynomial closest to exp for the maximum norm. The precondition, given by keyword `requires`, states that the argument is assumed to be in the expected interval. The post-condition (keyword `ensures`) states that the difference between the result and the true exponential is smaller or equal to 2^{-4}. The assertion in the body of the function is there to help the proof (see the main text). It is important to notice that the symbols + and $*$ in the code denote floating-point operations, whereas the same symbols in the assertion and the post-condition are operations on real numbers.

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

This formula exhibits the fact that the main block of the expression is a sum of logarithms of sums of exponentials, therefore, a sum of evaluations of the LSE function (modulo the use of logarithm of base 2 instead of the natural one).

1.2 Overview of our Proof Methodology

Obtaining formal proofs on the accuracy of floating-point programs is not a simple task. Generally speaking, one can start from a software environment for proving functional properties of programs, and augment it with a formalisation of floating-point arithmetic, typically via a library built on top of a formalisation of real numbers that provides a rounding function and its logical properties. It was done for example using the Coq proof assistant, augmented with the Flocq library [18, 19] for floating-point arithmetic, allowing to prove programs using the Coq general-purpose environment, as demonstrated by Boldo and Filliâtre in 2007 [14]. In such a context, the program must be written using the Coq vernacular language, and the proofs typically require a large amount of manually written proof steps.

To perform verification on codes written in the C language, and to obtain a higher degree of automation, Ayad and Marché [5] proposed a setting making use of the Frama-C environment for static analysis on C source code, with a dedicated library of ACSL specifications that allows to discharge the proofs to various theorem provers, in particular SMT solvers. Boldo and Marché [16] 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, this methodology can reach a fairly high amount of automation [31].

Let's illustrate the underlying methodology on the example given on Figure 1 proposing a rough approximation of the exponential function on the interval $[-1;1]$ (inspired by [5, Example 1]). The accuracy of that computation is expressed by the post-condition saying that the difference with the ideal real computation of exponential is smaller or equal to $\frac{1}{16}$. Formally proving that the program satisfies this

specification is not a simple task. Yet, the intermediate assertion added in the code allows to split this task into two parts: the assertion is a pure mathematical fact (not involving any floating-point computation) and thus concerns the so-called *method error*; whereas the proof of the post-condition can be performed by using the assertion as an hypothesis. The first task can thus be proved using pure mathematical reasoning, whereas the second part can be performed using reasoning on floating-point rounding, without needing any knowledge of the exponential function. Concretely, the proof of the assertion was performed by Ayad and Marché using the Coq-interval tactic of Coq [43], and the proof of the post-condition was achieved using the Gappa solver which possesses dedicated support for floating-point rounding [19].

Our proof methodology is based on the methodology illustrated by the former example. Yet, to achieve the proofs of our case studies in a reasonable simple manner, we discovered two important points that should be emphasised: first, the use of the intermediate language WhyML, and second the use *unbounded* floating-point numbers.

The WhyML language. It is the language of Why3 [10], a general-purpose environment for deductive verification. It is indeed used as an intermediate tool by Frama-C for C code, as it was done for the example in Figure 1, even if the user doesn't need to be aware of. In fact, Why3 is also used by other front-ends, for instance by Spark for Ada code [44]. 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 like the assertion of the example above, including dReal [33] and Metitarski [2]. 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 [30] and Alt-Ergo-FPA [22]. 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.

Unbounded floating-point numbers. The second important element in our work is the use of *unbounded* floating-point numbers. The goal is to separate the proofs concerning functional behaviour of numerical programs from the proof of absence of overflow. Mixing the two, as it was done in the example above becomes quickly a problem for larger programs, making the proofs overly complex. 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 [48] and also in advanced formalisation such as Flocq [18, 19]. Roughly speaking, unbounded floating-point number are very much like standard IEEE floating-point numbers, except that their exponent can be arbitrarily large. 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. There is an injection from finite IEEE float numbers to unbounded float numbers, therefore to use the proofs made on unbounded float numbers from C code it suffices to prove that the floating-point operations within the C code don't overflow nor produce NaN values.

Our methodology thus makes use of unbounded floating-point numbers and heavily relies on WhyML. It will be introduced in detail in Section 2. We still aim to achieve proofs on concrete C code. For that, we use the environment TIS-kernel, a fork of Frama-C, and its J3 plug-in for deductive verification, which is a prototype 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.

1.3 Outline of the Report and Contributions

In this document, we report on proofs of bounds on accuracy of functions LSE and SLSE. As a preliminary, we need to formalise and prove auxiliary results on iterated sums of numbers. As suggested above,

it appears that the best methodology was to formalise and prove these preliminary results directly using the intermediate language WhyML. We then provide a WhyML implementation of LSE, annotated with a post-condition stating our intended result on the accuracy of that function. Only in a second step we consider C implementations of the sum and the LSE function. For the SLSE function we only present a proof on paper, leaving formalisation for future work.

This report is structured as follows. Section 2 analyses the accuracy of iterated additions of numbers. The analysis is first stated informally, and then formalised in WhyML. In this section we introduce our modelling of unbounded floating-point numbers, and we illustrate how they simplify the proof of the accuracy of computation of sums. Even if the use of unbounded floating-point numbers allows us to ignore overflow, this section also analyses the required assumption needed to show the absence of overflow in a real C code. Section 3 focuses on the LSE function. After some discussion on the assumed accuracies of the underlying functions for computing exponential and logarithm, a general error bound on the LSE function is provided, first informally and then with a formal proof. As for compound sums, the formal proof is mainly done in WhyML. Only at a final step we consider the C code, in which we present the ACSL annotations expressing the expected results, and how the code is formally proved conforming to these specifications. Section 4 focuses on the SLSE function, following the same methodology as for LSE but without doing the formal proof. The concluding Section 5 discusses related work and future work, including a discussion on how we plan the formalise the proof of the SLSE function with a more automated methodology.

The full source code of our formalisation in WhyML and in C is available on the "Toccata" web gallery of verified programs [17].

2 Analysis and Error Bounds on Addition of Vectors

This section is focused on the analysis of rounding errors when computing the floating-point sum of a vector of numbers. We first introduce preliminaries and notations on floating-point numbers in Section 2.1. In Section 2.2, we introduce the unbounded floating-point numbers and their formalisation in WhyML. Section 2.3 then focuses on compound sums and presents a first main result on rounding errors, with a paper proof. Section 2.4 present the formalisation of sums in WhyML, and the formal proof of the same result. Finally in Section 2.5, we illustrate how this formal proof can be used to perform a formal proof of a simple C program for computing the sum of elements of an array. On this C code, we prove results on both the absence of overflow and on the bounds on the accumulated rounding error.

2.1 Preliminaries and Notations on Floating-Point Arithmetic

The IEEE-754 standard [38] defines several formats of representation of floating-point numbers. In this report we use only the 64-bit binary format (corresponding to the type `double` in C). In this format, a 64-bit word is decomposed into three parts: 1 bit for the sign, 11 bits for the exponent and 52 bits for the mantissa. Each of the 64-bit values represents either a real number or a special value among $+\infty$, $-\infty$ and NaN. The largest representable number is this format is $\mathsf{maxdouble} = (2-2^{-52}) \times 2^{1023}$, and the smallest positive representable number is 2^{-1074}.

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. In this report 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 $\mathsf{maxdouble}$ and 2^{1024}), $\mathsf{rnd}(x)$ 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.

When working with the Why3 environment, one can make use of floating-point operations via the standard library, which provides a module for floating-point numbers that is faithful to the IEEE-754 standard. Type declarations for floating-point numbers are provided. This includes the type named `double` for the 64-bit format. The same library provides operations including rnd (denoted `round` in that library), and also operation $\oplus$ (denoted as `.+` in that library), $\ominus$ etc.

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

$$|\mathsf{rnd}(x) - x| \leq \varepsilon|x| + \eta \tag{1}$$

where $\varepsilon = \frac{2^{-53}}{1+2^{-53}}$ and $\eta = 2^{-1075}$. This property can be considered as well-known and folklore in the literature, see for example Jeannerod and Rump [39]. In seminal publications, such as the Handbook of Computer Arithmetic [48] or Higham's survey [36], the simpler term $\varepsilon = 2^{-53}$ is used instead of $\frac{2^{-53}}{1+2^{-53}}$, inducing a slightly larger bound. Jeannerod and Rump [39, 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^{-53} 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 [39], Property 1 can be refined in the special case of addition because underflowing additions are exact:

$$|(x \oplus y) - (x+y)| \leq \varepsilon|x+y| \tag{2}$$

that is, the term η can be removed from formula 1. Moreover, it should be noted that (see for example the Handbook of Floating-Point Arithmetic [48])

$$|(x \oplus y) - (x+y)| \leq |x| \tag{3}$$

and symmetrically

$$|(x \oplus y) - (x+y)| \leq |y| \tag{4}$$

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

2.2 Unbounded Floating-Point Numbers

There is an annoying aspect when trying to use the properties 2, 3 and 4 when working on the true IEEE-754 double-precision type. These properties hold only if the addition does not overflow, otherwise the result of $x \oplus y$ is infinite and the formulas do not make any sense. In the literature on numerical programs it is folklore to abstract away from the true IEEE-754 types by considering a form of unbounded floating-point numbers. These are similar to the IEEE-754 ones except that their exponent part is not bounded. Equivalently, such an unbounded floating-point number can be represented by a pair of two integers (m,e), denoting the real number $m \times 2^e$, where the mantissa m satisfies $|m| < 2^{53}$ and the exponent e satisfies $e \geq -1074$. With this representation, there is no overflow anymore. The properties 2, 3 and 4 indeed hold for unbounded floating-point numbers.

Since this notion of unbounded floating-point number greatly simplifies the statements and the proofs of the results, we want to make use of it in our formal development using Why3. That's why we introduced a new theory for these. The theory is given on figures 2, 3 and 4. 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

```
module UDouble

  use real.RealInfix
  use real.Abs
  use ieee_float.RoundingMode

  (**  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

  axiom uround_exact: forall m:mode, x: udouble. uround m (to_real x) = x

  constant uzero:udouble
  axiom to_real_uzero : to_real uzero = 0.0

  constant utwo:udouble
  axiom to_real_utwo : to_real utwo = 2.0
```

Figure 2: Theory of unbounded doubles in WhyML: conversion to real numbers and rounding.

```
  (** {2 operations in RNE mode} *)

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

  val uadd (x y:udouble) : udouble
    ensures { result = uadd x y }

  function usub (x y:udouble) : udouble = uround RNE (to_real x -. to_real y)

  val usub (x y:udouble) : udouble
    ensures { result = usub x y }

  function umul (x y:udouble) : udouble = uround RNE (to_real x *. to_real y)

  val umul (x y:udouble) : udouble
    ensures { result = umul x y }

  function udiv (x y:udouble) : udouble = uround RNE (to_real x /. to_real y)

  val udiv (x y:udouble) : udouble
    requires { y <> uzero }
    ensures { result = udiv x y }

  function uminus (x:udouble) : udouble = uround RNE (-. (to_real x))

  val uminus (x:udouble) : udouble
    ensures { result = uminus x }
```

Figure 3: Theory of unbounded doubles in WhyML: basic operations.

```
(** {2 Lemmas on rounding} *)

  constant eps:real = 0x1p-53 /. (1. +. 0x1p-53)
  constant eta:real = 0x1p-1075

(** addition *)

  lemma 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

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

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

Figure 4: Theory of unbounded doubles in WhyML: accuracy properties (excerpt).

abstractly. The basic operations are defined as the rounding of the real operations. The properties 2, 3 and 4 are stated as lemmas in the theory. To provide guarantees that this theory is consistent, the Why3 environment offers the mechanism of *realisation*, constructing a refinement of that theory inside a proof assistant. For out theory of unbounded floating-point numbers, it is indeed straightforward to provide a Coq realisation built upon the Flocq library.

2.3 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 in this report to associate to the left, meaning that we define the compound sum from a_m (included) to a_k (excluded), denoted by $\bigoplus_{m\le i<k} a_i$, by the following recursive equations.

$$\bigoplus_{m\le i<k} a_i \;=\; 0 \qquad \text{if } k \le m$$

$$\bigoplus_{m\le i<k} a_i \;=\; \left(\bigoplus_{m\le i<k-1} a_i\right) \oplus a_{k-1} \qquad \text{when } m < k$$

Associating to the left is important because it impacts the final result of a sum. [1]

The following theorem states a bound on compound sums. It is a slight reformulation of a theorem by Jeannerod and Rump [39].

Theorem 2.1 (Accuracy of compound sums)**.** *For any vector a of unbounded doubles, and any $m \le n$:*

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

[1] 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.

Proof. The proof is done by induction on n. The base case $n = m$ is trivially true since both terms are zero, and the case $n = m+1$ is also trivial since the sums are both equal to a_m, without any rounding involved. For the inductive case, we assume $n > m+1$ and we assume the property holds for m and $n-1$. We first prove the additional property

$$\left|\mathsf{rnd}\left(\bigoplus_{m\le i<n-1} a_i \ + a_{n-1}\right) - \left(\bigoplus_{m\le i<n-1} a_i \ + a_{n-1}\right)\right| \le \varepsilon\left(\sum_{m\le i<n-1} |a_i| \ + (n-m-1)|a_{n-1}|\right) \tag{5}$$

by distinguishing two cases:

- If $|a_{n-1}| \le \varepsilon \sum_{m\le i<n-1} |a_i|$: by Property (4) we have

  $$\left|\mathsf{rnd}\left(\bigoplus_{m\le i<n-1} a_i \ + a_{n-1}\right) - \left(\bigoplus_{m\le i<n-1} a_i \ + a_{n-1}\right)\right| \le |a_{n-1}| \le \varepsilon \sum_{m\le i<n-1} |a_i|$$

  which proves Formula (5).

- Otherwise, that is when $\varepsilon \sum_{m\le i<n-1} |a_i| < |a_{n-1}|$, we use Property (2) to state that

  $$\left|\mathsf{rnd}\left(\bigoplus_{m\le i<n-1} a_i \ + a_{n-1}\right) - \left(\bigoplus_{m\le i<n-1} a_i \ + a_{n-1}\right)\right| \le \varepsilon \left|\bigoplus_{m\le i<n-1} a_i \ + a_{n-1}\right|$$

  with

  $$\begin{aligned}
  &\left|\bigoplus_{m\le i<n-1} a_i \ + a_{n-1}\right| \\
  &= \left|\bigoplus_{m\le i<n-1} a_i \ - \sum_{m\le i<n-1} a_i \ + \sum_{m\le i<n-1} a_i \ + a_{n-1}\right| \\
  &\le \left|\bigoplus_{m\le i<n-1} a_i \ - \sum_{m\le i<n-1} a_i\right| + \sum_{m\le i<n-1} |a_i| \ + |a_{n-1}| && \text{by triangular inequality} \\
  &\le (n-m-2)\varepsilon \sum_{m\le i<n-1} |a_i| \ + \sum_{m\le i<n-1} |a_i| \ + |a_{n-1}| && \text{by induction hypothesis} \\
  &\le (n-m-2)|a_{n-1}| + \sum_{m\le i<n-1} |a_i| \ + |a_{n-1}| && \text{by the case assumption, and } n-m-2 \ge 0 \\
  &= \sum_{m\le i<n-1} |a_i| \ + (n-m-1)|a_{n-1}|
  \end{aligned}$$

  which proves Formula (5).

We have now proved formula 5. We obtain the final result as follows:

$$\begin{aligned}
&\left|\bigoplus_{m\le i<n} a_i \ - \sum_{m\le i<n} a_i\right| \\
&= \left|\bigoplus_{m\le i<n} a_i \ - \left(\bigoplus_{m\le i<n-1} a_i \ + a_{n-1}\right) + \left(\bigoplus_{m\le i<n-1} a_i \ + a_{n-1}\right) - \sum_{m\le i<n} a_i\right| \\
&\le \left|\mathsf{rnd}\left(\bigoplus_{m\le i<n-1} a_i \ + a_{n-1}\right) - \left(\bigoplus_{m\le i<n-1} a_i \ + a_{n-1}\right)\right| + \left|\left(\bigoplus_{m\le i<n-1} a_i \ + a_{n-1}\right) - \sum_{m\le i<n} a_i\right|
\end{aligned}$$

```
let rec ghost function u_sum (a: int → udouble) (m n: int) : udouble
  variant { n - m }
= if n <= m then uzero else uadd (u_sum a m (n-1)) (a (n-1))
```

Figure 5: WhyML definition of compound sums of unbounded doubles

```
function real_fun (a:int → udouble) : int → real = fun i → to_real (a i)
function abs_real_fun (a:int → udouble) : int → real = fun i → abs (to_real (a i))

let rec lemma u_sum_accuracy (a:int → udouble) (m n:int)
  requires { m <= n }
  variant { n - m }
  ensures { abs (to_real (u_sum a m n) -. RealSum.sum (real_fun a) m n) <=.
              from_int (n-m-1) *. eps *. RealSum.sum (abs_real_fun a) m n  }
```

Figure 6: WhyML statement corresponding to Theorem 2.1.

On the left part we use Formula (5) and on the right part, the term a_{n-1} cancels so we can apply the induction hypothesis, hence

$$\begin{aligned}
&\left| \bigoplus_{m\le i<n} a_i \;-\; \sum_{m\le i<n} a_i \right| \\
&\le \varepsilon \left(\sum_{m\le i<n-1} |a_i| \;+(n-m-1)|a_{n-1}| \right) + (n-m-2)\varepsilon \sum_{m\le i<n-1} |a_i| \\
&= \varepsilon \left((n-m-1) \sum_{m\le i<n-1} |a_i| \;+(n-m-1)|a_{n-1}| \right) \\
&= \varepsilon(n-m-1) \sum_{m\le i<n} |a_i|
\end{aligned}$$

□

2.4 Formalising the Sum Accuracy Theorem in WhyML

We have to start by formally defining the compound operator $\bigoplus$ in WhyML. Given that its mathematical definition above is recursive, the idiomatic way to define it in WhyML is defining this function as a *ghost* recursive program that we prove terminating, as shown in Figure 5. Roughly speaking, a ghost program is a definition which is intended to be used in formal specifications, but not in concrete executable code. Note that the function here is defined on our new type `udouble` for unbounded doubles. The vector is provided as a function `a` from integers to floating-point numbers. We make use here of the higher-order features provided by Why3.

The WhyML statement corresponding to Theorem 2.1 is given by the lemma function `u_sum_accuracy` of Figure 6. Note how the pre-condition and the post-condition correspond respectively to the hypothesis and the statement of the theorem.

The body of this function constitutes the proof of the postcondition. The fact that the lemma function is recursive allows to prove it by induction. The body is shown in Figure 7. The structure of the code

```
1  if n = m then (* base case, trivial *) return ();
2  if n = m+1 then (* base case, almost trivial *)
3     begin
4     assert { u_sum a m n = uadd (u_sum a m m) (a m)
5              = uadd uzero (a m) = a m };
6     return ()
7     end;
8  (* induction *)
9  u_sum_accuracy a m (n-1);  (** recursive call to obtain induction hypothesis *)
10 let ghost exact_sum = RealSum.sum (real_fun a) m (n - 1) in  (* sum in real numbers *)
11 let ghost abs_sum = RealSum.sum (abs_real_fun a) m (n - 1) in (* sum of absolute values *)
12 begin ensures { abs exact_sum <=. abs_sum }
13   RealSum.sum_abs (real_fun a) (abs_real_fun a) m (n-1);
14 end;
15 let ghost psum : udouble = u_sum a m (n - 1) in (* the previous sum, in udouble *)
16 let ghost res : udouble = uadd psum (a (n - 1)) in (* the final sum, in udouble *)
17 let ghost s : real = to_real psum in  (* the previous sum, in real *)
18 let ghost anm1 = to_real (a (n-1)) in (* the last element as a real *)
19 let ghost delta = to_real res -. (s +. anm1) in (* the difference to bound *)
20 begin (* additional property (5) *)
21   ensures { abs delta <=. eps *. (abs_sum +. from_int (n-m-1) *. abs anm1) }
22   if abs anm1 <=. eps *. abs_sum then (** case 1 of the proof, trivial *) ()
23   else begin
24        (** case 2 of the proof *)
25        (* we know eps *. abs_sum <. abs anm1 *)
26        assert { abs (s +. anm1) <=. abs (s -. exact_sum) +. abs abs_sum +. abs anm1 };
27        assert { from_int (n-m-2) *. (eps *. abs_sum) <=. from_int (n-m-2) *. abs anm1 }
28        end
29 end;
30 assert { abs (to_real (u_sum a m n) -. RealSum.sum (real_fun a) m n)
31          <=. abs delta +. abs (s +. anm1 -. RealSum.sum (real_fun a) m n) }
```

Figure 7: WhyML body of `sum_double_accuracy`, corresponding to the proof of Theorem 2.1.

follows quite closely the structure of the paper proof of Theorem 2.1 of Section 2.3.

The WhyML source code with all the components above can then be fed to Why3, and provers executed on the generated VCs. Figure 8 summarises the results of provers on the VCs for the WhyML lemma `u_sum_accuracy` which corresponds to Theorem 2.1. The results '(5.00s)' mean that the prover reached the time limit given of 5 seconds. The results '(1000M)' mean that the prover reached the memory limit of 1 Gbytes. We run each of the five provers we had, to have a rough idea of their respective capabilities. Some proofs are done by all provers, some other only by a part of them. For one proof, namely the assertion on line 26 of Figure 7, only the FPA variant of Alt-Ergo was able to prove the goal. The difficulty resides on the conversion between integers and real numbers, which is likely to be poorly supported by other provers.

Proof obligations	**Alt-Ergo 2.4.3**	**Alt-Ergo 2.4.3 (FPA)**	**CVC4 1.8**	**CVC5 1.0.5**	**Z3 4.12.2**
`VC for u_sum_accuracy`					
transformation `split_vc`					
postcondition	0.05	0.08	0.09	0.06	(5.00s)
assertion	(5.00s)	(5.00s)	(5.00s)	0.65	(1000M)
postcondition	0.03	0.03	0.05	0.06	(1000M)
variant decrease	0.03	0.04	0.05	0.04	0.03
precondition	0.03	0.05	0.05	0.05	0.04
precondition	0.03	0.05	0.07	0.07	0.04
precondition	0.03	0.06	0.08	0.12	(5.00s)
postcondition	0.04	0.04	0.06	0.06	0.05
assertion	0.11	0.14	0.07	0.07	(5.00s)
assertion	(5.00s)	0.43	(5.00s)	(5.00s)	(5.00s)
postcondition	0.12	0.12	(5.00s)	(5.00s)	(5.00s)
assertion	0.04	0.04	0.11	0.15	(5.00s)
postcondition	0.07	0.07	(5.00s)	(5.00s)	(5.00s)

Figure 8: Provers results on VCs for Lemma 2.1 on the error bound on a compound sum of doubles.

```
1  /*@ axiomatic UDouble __attribute__ ((j3_theory ("udouble.UDouble"))) {
2    @
3    @   // logic type for unbounded floats in double format
4    @   type udouble __attribute__((j3_symbol("UDouble.udouble")));
5    @
6    @   // injection of unbounded floats into real numbers
7    @   logic real to_real(udouble x) __attribute__((j3_symbol("UDouble.to_real")));
8    @
9    @   // the epsilon constant, 2^{-53} / (1 + 2^{-53})
10   @   logic real eps __attribute__((j3_symbol("UDouble.eps")));
11   @
12   @ }
13   @*/
```

Figure 9: ACSL axiomatic for unbounded doubles, linked to the WhyML theory

2.5 Proving a C program Computing the Sum of an Array

Ultimately, we want to conduct formal proofs on concrete implementations of our functions under study, as code written in the C language. In this section we expose how we do it on a C implementation of the computation of the sum of elements of an array of floating-point numbers represented in strict IEEE double-precision format, that is the type `double` in C. Unlike the code in WhyML, we are not working with unbounded numbers here, we must stick to the regular IEEE floating-point numbers. It means that we do not only aim at proving a bound on the accuracy of the computation, but also at proving absence of overflow.

Regarding the accuracy property, a first question is how we formally specify the expected property. For that, we use the existing ACSL language introduced in Section 1.2, which already provides the notion of mathematical real numbers. The way to proceed with such a proof is representative of the new methodology we propose: regarding the proof of accuracy, we are going to reuse the former proof done in WhyML, in particular by introducing a notion of unbounded numbers in the specifications of our code. So a first step is to have a way to talk about unbounded doubles in ACSL annotations. For such a purpose, ACSL has features for introducing extra logic types, functions and predicates. These can be explicitly defined, or just declared via a notion of *axiomatisation*. This is the technique we use to introduce unbounded doubles.

2.5.1 ACSL Specification for Unbounded Doubles

Figure 9 displays how we declare the type of unbounded doubles in ACSL. As a matter of fact, we need very few elements here: the declaration of the logic type `udouble` on line 4, and the constant `eps` on line 7. We use a specific feature of J^3 here, that is to link these declarations with already existing definitions from Section 2.2. Technically this works using an attribute `j3_symbol`.

The next step is to provide a way to inject the regular type `double` of the C code to unbounded doubles. This is done with the axiomatic `J3UDouble`, given in Figure 10. We declare, on line 6, the function `to_udouble` that converts a C double into an unbounded double. This axiomatic relies on an extra WhyML theory displayed in Figure 11 where `to_udouble` is defined on line 4. The module `cfloat.Safe64` imported on line 2 in this theory is part of the default J^3 modeling. The type `Double.t` is the type used for representing double floats of the C code. It comes with a type invariant stating that all floats represented by this type are finite. The only way to build a float of this type is to use operations that don't cause overflows. Therefore, the basic arithmetic operations on this type come with a precondition stating that they don't cause an overflow.

```
/*@ axiomatic J3UDouble __attribute__ ((j3_theory ("j3bridge.UDouble"))) {
  @
  @   // conversion of a J3 safe double to an unbounded double
  @   logic udouble to_udouble(double z)
  @           __attribute__((j3_symbol("J3UDouble.to_udouble")));
  @ }
  @*/
```

Figure 10: ACSL axiomatic for linking unbounded doubles with C doubles.

```
module UDouble
  use export udouble.UDouble
  use cfloat.Safe64 as Double

  function to_udouble (x:Double.t) : udouble
  axiom to_real_of_to_udouble : forall x:Double.t. Double.to_real x = to_real (to_udouble x)

  axiom to_uzero : to_udouble Double.zeroF = uzero

  axiom tou_add :
    forall x y r: Double.t.
      Double.safe64_add_post x y r → to_udouble r = uadd (to_udouble x) (to_udouble y)
end
```

Figure 11: WhyML theory providing a bridge between ACSL symbols and WhyML definitions for unbounded doubles.

The axiomatic `J3Sum` of Figure 12 is there to provide the definitions of the sum of `double` with result as an unbounded double (so without any possible overflow), the exact sum as a real number, and the sum of absolute values. This axiomatic is linked to the bridge shown on Figure 13. In this axiomatic, a current technical constraint of the J^3 tool is that the parameters of these predicates must be declared as array of a fixed size. Here we more or less arbitrarily set the maximal size as 1024. See later in Section 2.6 about changing this value.

Altogether, the definitions described above are sufficient to specify the C code.

2.5.2 Accuracy of a C program for Summation

Figure 14 displays a C program for computing the sum of elements in an array of doubles. It is annotated with an ACSL specifications which relies on all the elements above. Notice that the line 3, introducing the constant `MAX_VALUE`, the line 6, bounding the elements of array `x`, and line 22, providing an extra assertion, are only added for proving the absence of overflow, discussed below.

The important parts to emphasise here are

- the line 8 which gives a post-condition that states that the program computes the same value as the WhyML definition of compound sums of unbounded numbers;
- the lines 9-10 that give a post-condition stating the main result theorem;
- the line 17 which states as an invariant that the variable `s` stores the sum of numbers added so far;

Proving the invariant is a simple task, resulting from the definition of the compound sum. Proving the first post-condition is then a simple consequence of the loop invariant. Finally, the proof of the second

```
#define MAX_SIZE 1024

/*@ axiomatic J3Sum __attribute__ ((j3_theory ("j3bridge.Sum"))) {
  @
  @   // sum of doubles, result as an unbounded double
  @   // usum_double(a,n,m) = a[n] + ... + a[m-1]
  @   //   where + is the rounding sum of udoubles
  @   logic udouble usum_double(double a[MAX_SIZE], integer n, integer m)
  @           __attribute__((j3_symbol("J3Sum.usum_double")));
  @
  @   // sum of doubles as a real number, without rounding
  @   // real_sum(a,n,m) = a[n] + ... + a[m-1]
  @   //   where + is the mathematical addition of real numbers
  @   logic real real_sum(double a[MAX_SIZE], integer n, integer m)
  @           __attribute__((j3_symbol("J3Sum.real_sum")));
  @
  @   // sum of absolute values, as a real number, without rounding
  @   // real_sum_of_abs(a,n,m) = |a[n]| + ... + |a[m-1]|
  @   //   where + is the mathematical addition of real numbers
  @   logic real real_sum_of_abs(double a[MAX_SIZE], integer n, integer m)
  @           __attribute__((j3_symbol("J3Sum.real_sum_of_abs")));
  @
  @ }
  @*/
```

Figure 12: ACSL axiomatic for declarations of predicates on sums.

```
module Sum
  use cfloat.Safe64 as Double
  use UDouble
  use sum.Sum as USum

  (* 'ufun a' lifts function 'a', from int to double, to a function from int to udouble *)
  function ufun [@inline_trivial] (a:int → Double.t) : int → udouble =
    fun x → to_udouble (a x)

  (* 'usum_double a m n' is the sum, with rounding, of 'a m',..,'a (n-1)',
     as an unbounded double *)
  function usum_double [@inline_trivial] (a:int → Double.t) (m n:int) : udouble =
    USum.u_sum (ufun a) m n

  use sum_real.Sum as RSum

  (* 'real_sum a m n' is the mathematical sum of 'a m',..,'a (n-1)' *)
  function real_sum [@inline_trivial] (a:int → Double.t) (m n:int) : real =
     RSum.sum (USum.real_fun (ufun a)) m n

  (* 'real_sum_of_abs a m n' is the mathematical sum of '|a m|',..,'|a (n-1)|' *)
  function real_sum_of_abs [@inline_trivial] (a:int → Double.t) (m n:int) : real =
     RSum.sum (USum.abs_real_fun (ufun a)) m n
```

Figure 13: WhyML theory providing a bridge between ACSL symbols and WhyML definitions for sums.

post-condition is made automatically because a similar statement is proved in WhyML as the lemma `u_sum_accuracy` (Figure 6). As one can see, all the methodology introduced before allow to have a

```
1  double a[MAX_SIZE];
2
3  #define MAX_VALUE 0x1p1012
4
5  /*@ requires 0 < size <= MAX_SIZE;
6    @ requires \forall integer i; 0 <= i < size ==> \abs(a[i]) <= MAX_VALUE;
7    @ requires \initialized (&a[0..]);
8    @ ensures to_udouble(\result) == usum_double(a, 0, size);
9    @ ensures \abs(\result - real_sum(a,0,size))
10   @           <=  (size-1) * eps * real_sum_of_abs(a,0,size);
11   @*/
12 double sum(size_t size) {
13   int i;
14   double s = 0.0;
15
16   /*@ loop invariant 0 <= i <= size;
17     @ loop invariant to_udouble(s) == usum_double(a, 0, i);
18     @ loop assigns i, s;
19     @ loop variant size - i;
20     @*/
21   for (i = 0; i < size; i++) {
22     //@ assert usum_double_bound(to_udouble(s), MAX_VALUE, MAX_SIZE);
23     s += a[i];
24   }
25   return s;
26 }
```

Figure 14: C code for computing a sum, annotated with ACSL specifications

```
1    @   // predicate expressing a constant bound, defined in WhyML as:
2    @   //   \abs(s) <= max * size * (1. + eps * (size - 1));
3    @   predicate usum_double_bound(udouble s, real max, integer size)
4    @           __attribute__((j3_symbol("J3Sum.usum_double_bound")));
```

Figure 15: ACSL axiomatic for absence of overflows

simple proof of the C code itself.

2.5.3 Proving absence of overflow

Yet, for the C code we have the additional burden to show the absence of overflow, which in that particular case amounts to prove that the addition performed on line 23 of Figure 14 never overflows. To achieve this, we need some extra assumptions of the values stored in the input which cannot be too large: smaller to a given bound `MAX_VALUE` that we fixed at 2^{1012} here, for a reason that will be clear below. This is the purpose of lines 3 and 6. The line 22 is an assertion that aims at providing a bound on `s`, deduced from the loop invariant itself. This assertion relies on an additional predicate defined in Figure 15 as a synonym for a WhyML predicate defined in Figure 16, on which we prove a lemma which formalises the following.

Lemma 2.2 (Bound on sums)**.** *For any constant S and M_a, any vector a, any indices m and n such that*

- $n - m \leq S$
- $|a_i| \leq M_a$ *for any* $m \leq i < n$

```
1  (* A lemma for proving absence of overflow *)
2
3  use int.Int
4  use real.RealInfix
5  use real.Abs
6  use real.FromInt
7  use sum.Bound
8
9  predicate usum_double_bound (s:udouble) (max:real) (size:int)
10   = abs (to_real s) <=. (max *. from_int size) *. (1. +. eps *. from_int (size - 1))
11
12 let lemma usum_double_constant_bound (a:int → Double.t) (m n:int) (max:real) (size:int)
13   requires { 0 <= n - m <= size }
14   requires { 0. <=. max }
15   requires { forall i. m <= i < n → abs (Double.to_real (a i)) <=. max }
16   ensures { usum_double_bound (usum_double a m n) max size }
17 = u_sum_constant_bounds max (ufun a) size m n
18
19 let lemma usum_double_constant_bound2 (a:int → udouble) (m n:int) (max:real) (size:int)
20   requires { 0 <= n - m <= size }
21   requires { 0. <=. max }
22   requires { forall i. m <= i < n → abs (UDouble.to_real (a i)) <=. max }
23   ensures { usum_double_bound (USum.u_sum a m n) max size }
24 = u_sum_constant_bounds max a size m n
25
26 let lemma usum_double_constant_bound3 (s:udouble) (a:int → udouble) (m n:int) (max:real) (size:int)
27   requires { 0 <= n - m <= size }
28   requires { 0. <=. max }
29   requires { forall i. m <= i < n → abs (UDouble.to_real (a i)) <=. max }
30   requires { s = USum.u_sum a m n }
31   ensures { usum_double_bound s max size }
32 = u_sum_constant_bounds max a size m n
```

Figure 16: WhyML predicate and lemma for absence of overflows

we have

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

Proof. The proof is done as a corollary of Theorem 2.1, and can be seen in WhyML on Figure 17. □

A consequence of that lemma is that the sum computed at line 23 in the program of Figure 14 must be smaller that

$$2^{1012} \times 2^{10} \times (1+2^{-53} \times 1023)$$

which is smaller than 2^{1023}, thus representable in the 64-bit floating-point format. See Section 2.6 for a discussion about the chosen bounds.

2.5.4 Proof results

Figure 18 presents the prover results on the proof of the extra lemma bonding sums, for proving absence of overflow. Figure 19 presents the prover results on the proof of the C code for addition. Most proofs can be done by any provers, a few proofs are done only by Alt-Ergo, and in particular with its FPA variant. For the assertion for absence of overflow itself, in the addition of the C code, it is likely that the Gappa solver could prove it as well.

```
let ghost u_sum_constant_bounds (max:real) (a:int → udouble) (size m n:int)
  requires { 0 <= n - m <= size }
  requires { 0. <=. max }
  requires { forall i. m <= i < n → abs (to_real (a i)) <=. max }
  ensures {
    abs (to_real (u_sum a m n))
    <=. (max *. from_int size) *. (1. +. eps *. from_int (size - 1))
  }
=
sum_bounds (-. max) max (real_fun a) m n;
   (* real sum is between -(n-m)*max and (n-m)*max *)
sum_bounds 0. max (abs_real_fun a) m n;
   (* real sum of abs values is between 0 and (n-m)*max *)
u_sum_accuracy a m n;
   (* call the known theorem on sum accuracy *)
assert {
  n-m > 0 →
  (max *. from_int size) *. (-.1. -. eps *. from_int (size - 1))
  <=. to_real (u_sum a m n) <=.
  (max *. from_int size) *. (1. +. eps *. from_int (size - 1))
  by
    abs (to_real (u_sum a m n) -. sum (real_fun a) m n) <=.
      (eps *. from_int (n-m-1)) *. (from_int size *. max)
  so
    abs (to_real (u_sum a m n) -. sum (real_fun a) m n) <=.
      max *. from_int size *. (eps *. from_int (size - 1))
};
assert {
  abs (to_real (u_sum a m n)) <=.
  (max *. from_int size) *. (1. +. eps *. from_int (size - 1))
}
```

Figure 17: Proof in WhyML for Lemma 2.2.

From now on, we will not run all the provers on each goal anymore, we report only of successful proofs.

2.6 Discussion about the bounds

In the axiomatic defined in Figure 12, we arbitrarily fixed a maximal bound $S = 1024 = 2^{10}$ on the size of the arrays involved. To prove the absence of overflow in the computation of the array sum in C, we also set a bound $M_a = 2^{1012}$ on the values inside the given array. As shown by the Lemma 2.2, the ultimate constraint is that the bound $M_a \times S \times (1+\varepsilon(S-1))$ is smaller than the maximal floating-point number in 64-bit format, that is a bit less than 2^{1024}. Since ε itself is a bit less than 2^{-53}, the coefficient $(1+\varepsilon(S-1))$ will never be larger than 2 for reasonable array sizes. So, roughly speaking, if one want to multiply the size of the array by some amount, then the bounds on the values in the array must be divided by the same amount.

Proof obligations	**Alt-Ergo 2.4.3**	**Alt-Ergo 2.4.3 (FPA)**	**CVC4 1.8**	**CVC5 1.0.5**	**Z3 4.12.2**
`VC for u_sum_constant_bounds`					
transformation `split_vc`					
precondition	0.04	0.06	0.15	0.10	(5.00s)
precondition	0.06	0.08	0.11	0.13	(5.00s)
precondition	0.02	0.06	0.06	0.07	0.03
assertion					
transformation `split_vc`					
assertion	(5.00s)	0.26	(5.00s)	(5.00s)	(5.00s)
VC for u_sum_constant_bounds	(5.00s)	0.28	(5.00s)	(5.00s)	(5.00s)
VC for u_sum_constant_bounds	0.08	0.20	(5.00s)	(5.00s)	(5.00s)
VC for u_sum_constant_bounds	0.06	0.07	(5.00s)	(5.00s)	(5.00s)
assertion	0.12	0.24	(5.00s)	(5.00s)	(5.00s)
postcondition	0.03	0.05	0.06	0.08	0.03

Figure 18: Prover results on proof of Lemma 2.2.

Proof obligations	Alt-Ergo 2.4.3	Alt-Ergo 2.4.3 (FPA)	CVC4 1.8	CVC5 1.0.5	Z3 4.12.2
`VC for sum`					
transformation `split_vc`					
precondition	0.06	0.07	0.09	0.12	0.02
loop allocates nothing (loop invariant init)	0.11	0.12	0.12	0.18	0.03
loop invariant init	0.06	0.08	0.15	0.18	0.06
loop invariant init	0.20	0.33	0.71	0.77	0.08
loop assigns memory (loop invariant init)	0.12	0.12	0.09	0.13	0.02
loop assigns (loop invariant init)	0.07	0.09	0.12	0.26	0.02
generated: var s initialized (loop invariant init)	0.07	0.07	0.09	0.13	0.03
generated: var i initialized (loop invariant init)	0.06	0.07	0.15	0.10	0.02
initialized (precondition)	0.06	0.08	0.11	0.11	0.02
assertion	0.27	0.37	(5.00s)	(5.00s)	(1000M)
initialized (precondition)	0.07	0.07	0.23	0.13	0.02
initialized (precondition)	0.07	0.08	0.13	0.18	0.02
index in array bounds	0.08	0.08	0.08	0.15	0.03
index in array bounds	0.13	0.16	0.36	0.31	0.06
initialized (precondition)	0.12	0.14	0.73	0.84	0.10
initialized (precondition)	0.07	0.08	0.21	0.22	0.02
floating-point overflow	(5.00s)	1.12	(5.00s)	(5.00s)	(1000M)
initialized (precondition)	0.06	0.08	0.12	0.15	0.03
integer underflow	0.06	0.09	0.27	0.42	0.21
integer overflow	0.07	0.08	0.28	0.29	0.35
loop variant decrease	0.07	0.09	0.45	0.33	0.04
loop allocates nothing (loop invariant preservation)	0.08	0.10	0.09	0.10	0.02
loop invariant preservation	0.18	0.21	0.49	0.35	0.49
loop invariant preservation	0.34	0.40	(5.00s)	(5.00s)	(1000M)
loop assigns memory (loop invariant preservation)	0.09	0.13	0.09	0.12	0.04
loop assigns (loop invariant preservation)	0.08	0.12	0.09	0.10	0.03
generated: var s initialized (loop invariant preservation)	0.08	0.16	0.09	0.12	0.04
generated: var i initialized (loop invariant preservation)	0.07	0.11	0.11	0.16	0.02
initialized (precondition)	0.07	0.08	0.12	0.09	0.04
generated: allocates nothing (postcondition)	0.07	0.07	0.18	0.09	0.04
postcondition	0.28	0.18	0.32	0.50	(1000M)
postcondition	0.27	0.19	0.89	0.70	(1000M)

Figure 19: Prover results on VCs for C code computing the compound sum of doubles.

3 Analysis of the Log-Sum-Exp Function

The purpose of this section is now to analyse programs (in WhyML first, then in C) for computing floating-point approximations of the function

$$\mathsf{LSE}(a) = \log\left(\sum_{0\leq i<n} \exp(a_i)\right)$$

In implementations using floating-point numbers, not only the sum is subject to rounding, but also the computations of functions exp and log. In this report, 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: this is presented in Section 3.1. In Section 3.2 we analyse the accuracy of the LSE computation on paper, again using unbounded floats. We discuss about how the obtained bound varies depending on the input bounds in Section 3.3. We formalise the result in WhyML, in Section 3.4, and then on a C code in Section 3.5.

3.1 Approximations of Logarithm and Exponential

As already mentioned in the introduction, we don't want to rely, as Blanchard et al. do [9], on perfectly rounded implementations of exponential and logarithm. Instead we assume we have implementations that are possibly less precise (such as the one given as toy example in Figure 1), 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. 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 pretend 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 litterature that such implementations exist for double precision, with a correct rounding [26] (that is with $E_{\exp} = E_{\log} = \varepsilon$ and $M_{\exp} = M_{\log} =$ *maxdouble*), such as the ones provided by the CORE-MATH project [49]. Yet, to perform the analysis below, we will need some properties, in particular that the approximated exponential is always positive. This would be a consequence of the properties above if we assume $E_{\exp} < 1$. Indeed we need a bit more than that, we will assume $E_{\exp} \leq 0.5$ for the proof of Theorem 3.4. We also assume $E_{\log} \leq 1$ in the following. Moreover, a reasonable assumption is to assume $M_{\exp} \leq 708$: with this assumption, the result of exponential will never be a subnormal number.

Figure 20 displays how these two assumed functions are declared in WhyML, working on unbounded doubles.

3.2 Mathematical Analysis of Accuracy of Computation of LSE

The floating-point approximations of LSE is given by the following formula

$$\widehat{\mathsf{LSE}}(a) = \widehat{\log}\left(\bigoplus_{0\leq i<n} \widehat{\exp}(a_i)\right)$$

```
constant exp_max_value :real
axiom exp_max_value_spec: 0.0 <. exp_max_value

(** parameter for the error on exponential *)
constant exp_error:real
axiom exp_error_bound : 0. <. exp_error <=. 0.5

(** the assumed exponential function on unbounded doubles *)
function u_exp (x:udouble) : udouble
axiom u_exp_spec :
  forall x:udouble [to_real (u_exp x)].
    abs (to_real x) <=. exp_max_value ->
    abs (to_real (u_exp x) -. exp (to_real x)) <=. exp (to_real x) *. exp_error

lemma u_exp_pos:
  forall x:udouble [to_real (u_exp x)].
    abs (to_real x) <=. exp_max_value -> 0.0 <. to_real (u_exp x)

constant log_max_value :real
axiom log_max_value_spec: 0.0 <. log_max_value

(** parameter for the error on logarithm *)
constant log_error:real
axiom log_error_bound : 0. <. log_error <=. 1.

(** the assumed logarithm function on unbounded doubles *)
function u_log (x:udouble) : udouble
axiom u_log_spec  :
  forall x:udouble [to_real (u_log x)]. 0.0 <. to_real x <=. log_max_value ->
  abs (to_real (u_log x) -. log (to_real x)) <=. abs (log (to_real x)) *. log_error
```

Figure 20: WhyML formalisation of approximations of exp and log

naturally involving the rounded compound sum. Our main result concerning the accuracy of the computation of $\widehat{\mathsf{LSE}}$ is given by Theorem 3.4 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 a simple composition of errors on a sum.

Lemma 3.1 (Composition of errors in a sum)**.** *Given any vectors a and $\widehat{a}$ such that for all i, $|\widehat{a}_i - a_i| \leq A|a_i| + B$ we have:*

$$\left| \sum_{0\leq i<n} \widehat{a}_i \;\; - \sum_{0\leq i<n} a_i \right| \leq A \sum_{0\leq i<n} |a_i| \;\; + nB$$

Proof. Straightforward by induction on n. □

The second lemma is in fact a generalisation of Theorem 2.1 on the accuracy of compounds sums, when the input vector is itself subject to errors.

Lemma 3.2 (Accuracy of sums, generalised)**.** *Given any vectors a and $\widehat{a}$ such that for all i, $|\widehat{a}_i - a_i| \leq A|a_i| + B$ we have:*

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

Proof. We show how to prove the upper bound, the method is the same for the lower bound. By Theorem 2.1 on accuracy of compound sums, we have:

$$\bigoplus_{0\le i<n} \widehat{a_i} \le \sum_{0\le i<n} \widehat{a_i} \ +(n-1)\varepsilon \sum_{0\le i<n} |\widehat{a_i}|$$

By Lemma 3.1 we have :

$$\sum_{0\le i<n} \widehat{a_i} \le \sum_{0\le i<n} a_i + A \sum_{0\le i<n} |a_i| \ + nB$$

and

$$\sum_{0\le i<n} |\widehat{a_i}| \le \sum_{0\le i<n} |a_i| + A \sum_{0\le i<n} |a_i| \ + nB$$

Therefore:

$$\begin{aligned}\bigoplus_{0\le i<n} \widehat{a_i} &\le \sum_{0\le i<n} a_i \ + A \sum_{0\le i<n} |a_i| \ + nB + (n-1)\varepsilon \left(\sum_{0\le i<n} |a_i| \ + A \sum_{0\le i<n} |a_i| \ + nB \right) \\ &\le \sum_{0\le i<n} a_i \ + (A + (n-1)\varepsilon(1+A)) \sum_{m\le i<n} |a_i| \ + Bn(1+(n-1)\varepsilon)\end{aligned}$$

□

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

Lemma 3.3 (Error propagation for mathematical logarithm)**.** *For any positive real numbers x and $\widehat{x}$ such that $|\widehat{x}-x| \le Ax+B$, with $B < x(1-A)$ we have:*

$$|\log\widehat{x} - \log x| \le -\log(1-(A+B/x))$$

Proof. Pose $E = A + B/x$. Since $B < x(1-A)$ we have $E < 1$. We have $x(1-E) \le \widehat{x} \le x(1+E)$, so

$$\log(x(1-E)) = \log(x) + \log(1-E) \le \log(\widehat{x}) \le \log(x(1+E)) = \log(x) + \log(1+E)$$

Since $0 \le E < 1$, we have $\log(1+E) \le -\log(1-E)$, therefore:

$$\log(x) + \log(1-E) \le \log(\widehat{x}) \le \log(x) - \log(1-E)$$

□

Theorem 3.4 (Accuracy of LSE)**.** *For any $n \ge 1$, and no larger than 2^{51}, any vector a of size n such that for all i, $|a_i| \le M_{\exp}$, and assuming that*

$$\exp(M_{\exp})(1+E_{\exp})n(1+\varepsilon(n-1)) \le M_{\log} \tag{6}$$

we have

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

Proof. We apply Lemma 3.2 with $\widehat{\exp}(a)$ for $\widehat{a}$, $\exp(a)$ for a, $E_{\exp}$ for A and 0 for B. The hypothesis of Lemma 3.2 is satisfied due to the hypothesis $|a_i| \le M_{\exp}$ of the theorem, that is needed to call $\widehat{\exp}$ on each a_i. Therefore

$$\left| \bigoplus_{0\le i<n} \widehat{\exp}(a_i) - \sum_{0\le i<n} \exp(a_i) \right| \le \left(E_{\exp} + (n-1)\varepsilon(1+E_{\exp})\right) \sum_{0\le i<n} \exp(a_i)$$

We then use Lemma 3.3 with $\bigoplus_{0\le i<n} \widehat{\exp}(a_i)$ for $\widehat{x}$ and $\sum_{0\le i<n} \exp(a_i)$ for x, $B=0$ and $A = E_{\exp} + (n-1)\varepsilon(1+E_{\exp})$. To satisfy hypothesis $B < x(1-A)$ of Lemma 3.3 we need the sum of exponentials to be positive, which is trivial, and $A<1$, which is true since $E_{\exp}$ is assumed not larger than 0.5, hence

$$A = E_{\exp} + (n-1)\varepsilon(1+E_{\exp}) \le 0.5 + (2^{51}-1)2^{-53} \times 1.5 < 1$$

By applying the Lemma 3.3, we get

$$\left|\log\left(\bigoplus_{0\le i<n} \widehat{\exp}(a_i)\right) - \log\left(\sum_{0\le i<n} \exp(a_i)\right)\right| \le -\log\left(1-\left(E_{\exp}+(n-1)\varepsilon(1+E_{\exp})\right)\right)$$

that is

$$\left|\log\left(\bigoplus_{0\le i<n} \widehat{\exp}(a_i)\right) - \mathsf{LSE}(a)\right| \le -\log\left(1-\left(E_{\exp}+(n-1)\varepsilon(1+E_{\exp})\right)\right)$$

Therefore

$$\begin{aligned}
&\left|\widehat{\mathsf{LSE}}(a) - \mathsf{LSE}(a)\right| \\
&\le \left|\widehat{\mathsf{LSE}}(a) - \log\left(\bigoplus_{0\le i<n} \widehat{\exp}(a_i)\right)\right| + \left|\log\left(\bigoplus_{0\le i<n} \widehat{\exp}(a_i)\right) - \mathsf{LSE}(a)\right| \\
&= \left|\widehat{\log}\left(\bigoplus_{0\le i<n} \widehat{\exp}(a_i)\right) - \log\left(\bigoplus_{0\le i<n} \widehat{\exp}(a_i)\right)\right| + \left|\log\left(\bigoplus_{0\le i<n} \widehat{\exp}(a_i)\right) - \mathsf{LSE}(a)\right|
\end{aligned}$$

At this point we want to apply the assumption of the accuracy of $\widehat{\log}$. For that, we need to show that the argument $\bigoplus_{0\le i<n} \widehat{\exp}(a_i)$ is in the assumed domain, that is, is positive and no larger than $M_{\log}$. It is positive as a sum of positive numbers, and according to Lemma 2.2 it is bounded by $M_a \times n(1+\varepsilon(n-1))$ where M_a is a bound for all $\widehat{\exp}(a_i)$. Such a bound can be obtained from the assumption on the accuracy of $\widehat{\exp}$ giving

$$\widehat{\exp}(a_i) \le \exp(a_i)(1+E_{\exp}) \le \exp(M_{\exp})(1+E_{\exp})$$

Our sum is thus bounded by $\exp(M_{\exp})(1+E_{\exp})n(1+\varepsilon(n-1))$ which is smaller or equal to $M_{\log}$ using the hypothesis (6). We can continue our proof with

$$\begin{aligned}
&\left|\widehat{\mathsf{LSE}}(a) - \mathsf{LSE}(a)\right| \\
&\le E_{\log} \times \left|\log\left(\bigoplus_{0\le i<n} \widehat{\exp}(a_i)\right)\right| + \left|\log\left(\bigoplus_{0\le i<n} \widehat{\exp}(a_i)\right) - \mathsf{LSE}(a)\right| \\
&\le (E_{\log}+1) \times \left|\log\left(\bigoplus_{0\le i<n} \widehat{\exp}(a_i)\right) - \mathsf{LSE}(a)\right| + E_{\log} \times |\mathsf{LSE}(a)| \\
&\le E_{\log}|\mathsf{LSE}(a)| - \log\left(1-\left(E_{\exp}+(n-1)\varepsilon(1+E_{\exp})\right)\right)(1+E_{\log})
\end{aligned}$$

which is the expected bound. □

3.3 Discussion on the Variations of the Bound on Accuracy

As seen in the proof above, the hypothesis (6) of Theorem 3.4 is required to call the $\widehat{\log}$ function on the proper interval of definition.

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

- A relative part, which is $E_{\log}$
- A constant part :
$$-\log\left(1-\left(E_{\exp}+(n-1)\varepsilon(1+E_{\exp})\right)\right)(1+E_{\log})$$
We note that $-\log(1-x) < 2x$ for $x \leq \frac{1}{2}$. We can therefore bound the constant error by
$$2\times\left(E_{\exp}+(n-1)\varepsilon(1+E_{\exp})\right)(1+E_{\log})$$
The factors that dominate the constant bound are $E_{\exp}$ and $\varepsilon\times(n-1)$.

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 $E_{\log}$, $E_{\exp}$, $M_{\exp}$, $M_{\log}$ and n. As in Section 2.6, let us assume a reasonable bound in practice on n that is 1024. Similarly, let us assume the input numbers are smaller than 25, so that we can choose $M_{\exp} = 25$.

- 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 (6) of Theorem 3.4, it suffices to have $M_{\log}$ larger than
$$\begin{array}{rcl}\exp(M_{\exp})(1+E_{\exp})n(1+\varepsilon(n-1)) & \leq & \exp(25)(1+2^{-53})2^{10}(1+2^{-53}\times 1023)\\ & \leq & 7.38\times 10^{13}\end{array}$$
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
$$\begin{array}{rcl}2\times\left(E_{\exp}+(n-1)\varepsilon(1+E_{\exp})\right)(1+E_{\log}) & \leq & 2\times\left(2^{-53}+1023\times 2^{-53}(1+2^{53})\right)(1+2^{-53})\\ & \leq & 2.28\times 10^{-13}\end{array}$$
- 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 for industrial needs of some Mitsubishi Electric application. Then, to ensure that hypothesis (6) of Theorem 3.4, it suffices the have $M_{\log}$ larger than
$$\begin{array}{rcl}\exp(M_{\exp})(1+E_{\exp})n(1+\varepsilon(n-1)) & \leq & \exp(25)(1+2^{-40})2^{10}(1+2^{-53}\times 1023)\\ & \leq & 7.38\times 10^{13}\end{array}$$
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
$$\begin{array}{rcl}2\times\left(E_{\exp}+(n-1)\varepsilon(1+E_{\exp})\right)(1+E_{\log}) & \leq & 2\times\left(2^{-40}+1023\times 2^{-53}(1+2^{53})\right)(1+2^{-36})\\ & \leq & 2.05\times 10^{-12}\end{array}$$
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.

```
let lemma lse_accuracy (a:int → udouble) (size:int)
  requires { 1 <= size }
  requires { from_int (size - 1) <=. 0x1p51 }
  requires { forall i. 0 <= i < size → abs (to_real (a i)) <=. exp_max_value }
  requires { exp exp_max_value *. (1.0 +. exp_error) *. from_int size *. (1.0 +. eps *. from_int (size -
   1))
            <=. log_max_value }
  ensures {
    let err = exp_error +. eps *. from_int (size - 1) *. (1. +. exp_error) in
    abs (to_real (u_lse a size) -. lse_exact a size) <=.
    log_error *. abs (lse_exact a size) -. log (1. -. err) *. (1. +. log_error)
  }
```

Figure 21: Statement of Theorem 3.4 in WhyML.

3.4 Formalisation and Proofs in WhyML

The WhyML statement corresponding to Theorem 3.4 is given in Figure 21. The WhyML proof of Theorem 3.4 is given by the body of the `lse_accuracy` lemma function, displayed in Figure 22. It more or less follows the paper proof above. Figure 23 summarises the results of provers on the VCs for the WhyML lemma function `lse_accuracy`. The Alt-Ergo prover, and its FPA variant, is enough to prove all the generated VCs.

```
let ghost s = USum.u_sum (u_exp_fun a) 0 size in
let ghost sum_exps = SumReal.sum (exp_fun a) 0 size in
begin (* statement corresponding to invocation of Lemma 3.2 *)
  ensures {
    abs ((to_real s) -. sum_exps) <=.
      sum_exps *. (exp_error +. eps *. from_int (size - 1) *. (1. +. exp_error))
  }
  Combine.u_sum_accuracy_combine_pos (* this is Lemma 3.2 *)
    exp_error 0.0 (exp_fun a) (u_exp_fun a) 0 size;
end;
begin
  ensures { sum_exps >. 0.0 }
  SumReal.sum_strictly_pos (exp_fun a) 0 size;
end;
begin (* required domain for u_log *)
  ensures { 0. <. (to_real s) <=. log_max_value }
  (* assert { s = USum.u_sum (u_exp_fun a) 0 size}; *)
  assert { forall i. 0 <= i < size →
    0.0 <=. to_real (u_exp (a i)) <=. exp exp_max_value *. (1.0 +. exp_error)
    by abs (to_real (u_exp (a i)) -. exp (to_real (a i))) <=. exp (to_real (a i)) *. exp_error
    so to_real (u_exp (a i)) <=. exp (to_real (a i)) *. (1.0 +. exp_error) };
  B.u_sum_constant_bounds (exp exp_max_value *. (1.0 +. exp_error)) (u_exp_fun a) size 0 size;
end;
let ghost r = u_log s in
assert { r = u_lse a size };
let ghost err = exp_error +. eps *. from_int (size - 1) *. (1. +. exp_error) in
assert { err <. 1.0 };
begin
  ensures { abs (log (to_real s) -. log sum_exps) <=. -. log (1. -. err) }
  log_combine_err sum_exps (to_real s) err 0.; (* invocation of Lemma 3.3 *)
end;
assert { (log_error +. 1.0) *. (abs (log (to_real s) -. log sum_exps)) <=. -. log (1. -. err) *.
   (log_error +. 1.0)
         by (log_error +. 1.0 >=. 0.0)
};
assert {
  abs (to_real r -. lse_exact a size)
  <=. abs (to_real r -. log (to_real s)) +.
      abs (log (to_real s) -. log sum_exps)
  <=. (log_error +. 1.0) *. (abs (log (to_real s) -. log sum_exps))
      +. log_error *. abs (lse_exact a size)
  <=. log_error *. abs (lse_exact a size) -. log (1. -. err) *. (log_error +. 1.0)

  }
```

Figure 22: Proof of Theorem 3.4 in WhyML.

Proof obligations	**Alt-Ergo 2.4.3**	**Alt-Ergo 2.4.3 (FPA)**
`VC for lse_accuracy`		
transformation `split_vc`		
precondition	2.45	
precondition	0.07	
precondition	0.02	
precondition	0.03	
precondition	0.02	
postcondition	0.04	
precondition	0.04	
precondition	0.18	
postcondition	0.03	
assertion		
transformation `split_vc`		
assertion	0.05	
VC for lse_accuracy	0.07	
VC for lse_accuracy	0.06	
VC for lse_accuracy	0.99	
precondition	0.02	
precondition	0.06	
precondition	0.34	
postcondition		
transformation `split_vc`		
postcondition		0.05
postcondition	0.03	
assertion	0.03	
assertion	0.02	
precondition	0.02	
precondition	0.19	
precondition	0.03	
postcondition	0.02	
assertion		
transformation `split_vc`		
assertion	0.02	
VC for lse_accuracy	0.93	
assertion	2.54	
postcondition	0.02	

Figure 23: Provers results on VCs for Theorem 3.4 on accuracy theorem of the LSE function.

```
/*@ requires 0 < size <= MAX_SIZE;
  @ requires \initialized (&a[0..size-1]);
  @ // to fit exp_approx pre-condition
  @ requires \forall integer i; 0 <= i < size ==> \abs(a[i]) <= exp_max_value;
  @ // additional requirement to prevent overflow on addition
  @ requires exp_max_value <= 701.0;
  @ // the hypothesis (6) of Theorem 3.4
  @ requires \exp(exp_max_value) * (1.0 + exp_error) * size * (1.0 + (eps * (size - 1)))
             <= log_max_value;
  @ // additional requirement to prevent overflow on addition
  @ requires log_max_value <= 0x1p1023;   // close to max double
  @ // the 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;
    @ // to prove the first post-condition
    @ loop invariant to_udouble(s) == u_sum_of_u_exp(a, 0, i);
    @ // for absence of overflow on the sum
    @ loop invariant \forall integer j; 0 <= j < i ==>
    @   \abs(to_real(u_exp(to_udouble(a[j])))) <= \exp(exp_max_value) * (1.0 + exp_error) ;
    @ // for calling log:
    @ loop invariant (i == 0 ? s == 0.0 : 0.0 < s);
    @ 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])) <= exp_max_value ;
    //@ assert \exp(to_real(to_udouble(a[i]))) <= \exp(exp_max_value) ;
    /*@ assert \abs(to_real(u_exp(to_udouble(a[i])))) - \exp(to_real(to_udouble(a[i]))))
      @        <= \exp(exp_max_value) * exp_error ;
      @*/
    //@ assert to_real(u_exp(to_udouble(a[i]))) <= \exp(exp_max_value) * (1.0+exp_error) ;
    //@ assert usum_double_bound(u_sum_of_u_exp(a, 0, i), \exp(exp_max_value) * (1.0 + exp_error), size);
    s += exp_approx(a[i]);
  }

  //@ assert usum_double_bound(to_udouble(s), \exp(exp_max_value) * (1.0 + exp_error), size);
  return log_approx(s);
}
```

Figure 24: C code for computing LSE.

3.5 Analysis of a C Code for LSE

Our C code computing the LSE function is given on Figure 24. In a first step, as we did for the summation function, 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, as shown

```
1  /*@ // An axiomatic for floating-point exponential and logarithm
2    @ // the specifications are given in exp_log.mlw, module ExpLogApprox
3    @
4    @ axiomatic ExpLogApprox __attribute__ ((j3_theory ("exp_log.ExpLogApprox"))) {
5    @
6    @ // Validity range on exponential, assuming 0.0 < exp_max_value
7    @ logic real exp_max_value __attribute__((j3_symbol("ExpLogApprox.exp_max_value")));
8    @
9    @ // Error parameter on exponential, assuming 0. <=. exp_error <=. 0.5
10   @ logic real exp_error __attribute__((j3_symbol("ExpLogApprox.exp_error")));
11   @
12   @ // Error parameter on logarithm, assuming 0. <=. log_error <=. 1.
13   @ logic real log_error __attribute__((j3_symbol("ExpLogApprox.log_error")));
14   @
15   @ // the approximate exponential, on unbounded doubles
16   @ // spec: for all x, |u_exp(x) - exp(x)| <= exp(x) * exp_error
17   @ logic udouble u_exp(udouble x) __attribute__((j3_symbol("ExpLogApprox.u_exp")));
18   @
19   @ // Validity range on logarithm, assuming 0.0 < log_max_value
20   @ logic real log_max_value __attribute__((j3_symbol("ExpLogApprox.log_max_value")));
21   @
22   @ // the approximate logarithm, on unbounded doubles
23   @ // spec: for all x, 0 < x ==> |u_log(x) - log(x)| <= exp(x) * exp_error
24   @ logic udouble u_log(udouble x) __attribute__((j3_symbol("ExpLogApprox.u_log")));
25   @ }
26   @*/
27
28 /*@ // An axiomatic for LSE
29   @
30   @ axiomatic LSE __attribute__ ((j3_theory ("j3bridge.LSE"))) {
31   @
32   @ logic udouble u_sum_of_u_exp(double a[MAX_SIZE], integer m, integer n)
33   @         __attribute__((j3_symbol("LSE.u_sum_of_u_exp")));
34   @
35   @ logic udouble u_lse(double a[MAX_SIZE], integer size)
36   @         __attribute__((j3_symbol("LSE.u_lse")));
37   @
38   @ logic real lse_exact(double a[MAX_SIZE], integer size)
39   @        __attribute__((j3_symbol("LSE.lse_exact")));
40   @
41   @ // predicate no_overflow_pre(double s, double a[MAX_SIZE], integer size_ub, integer m, integer n)
42   @ //             __attribute__((j3_symbol("J3Axiomatic.no_overflow_pre")));
43   @
44   @ // predicate no_overflow_post(double s, integer size_ub)
45   @ //             __attribute__((j3_symbol("J3Axiomatic.no_overflow_post")));
46   @ }
47   @*/
```

Figure 25: Bridge with WhyML definitions, dedicated to LSE.

on Figure 25. We declare and specify the auxiliary C functions for computing approximatations of exp and log, as shown on Figure 26.

The post-condition on lines 14-15 of Figure 24 thus expresses that the result of the C function is equal to the LSE function defined in WhyML. The post-condition on lines 16-20 of Figure 24 expresses the expected bounding property, as stated by Theorem 3.4. The pre-condition on lines 7-8 is required to allow calling the exponential inside its correct domain. The pre-condition on lines 11-12 expresses the

```
1  /*@ requires \abs(x) <= exp_max_value;
2    @ ensures to_udouble(\result) == u_exp(to_udouble(x));
3    @ assigns \nothing;
4    @*/
5  extern double exp_approx(double x);
6
7  /*@ requires 0 < x <= log_max_value;
8    @ ensures to_udouble(\result) == u_log(to_udouble(x));
9    @ assigns \nothing;
10   @*/
11 extern double log_approx(double x);
```

Figure 26: External C functions for exp and log, specified in ACSL.

required hypothesis 6 of Theorem 3.4.

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 27-28. The post-condition on lines 16-20 is proved using the proof of the same statement already done in WhyML.

To prove that the sum `s` on line 46 fits in the expected range of the log, we do as we did for the WhyML code by invoking Lemma 2.2, which required a few intermediate assertions on lines 38-45.

Proving absence of numerical overflow Unlike for the WhyML code, and similarly to the C code for compound sums, we additionally have to prove the absence of overflow, when computing the addition on line 68. 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}, which is assumed smaller than 708. Yet, we adding 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 7-8 of Figure 24, that M_{exp} is smaller than 701. With this extra assumption, we can prove the loop invariant on lines 30-31, also using the assertions on lines 38-43. Then, to prove the absence of overflow on the sum, we can use the same technique as we did in Section 2.2 for the summation, and invoke Lemma 2.2. This is the purpose of the assertion on line 45.

We also have to prove that the resulting sum fits in the domain of $\widehat{\log}$, for that we first have to prove that the sum is positive, which is using the loop invariant on line 32-33, and that it smaller than $M_{\log}$, which is done by the assertion on line 49 combined with hypothesis 6 stated on lines 9-10 and the pre-condition on $M_{\log}$ given lines 12-13.

There are 49 VCs generated from the C code implementing LSE. Most of them are discharged using Alt-Ergo. For the assertion just before the `return`, we neeed to make a manual application of a bounding lemma, no solver being able to discover the proper instance automatically. There are also 2 VCs that are discharged by Alt-Ergo FPA.

4 Analysis of the SLSE function

The goal of this section is to analyse the accuracy of a floating-point implementation of the function

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

that we introduced in Section 1.1. We remind that the parameter ρ has any value between 0 and 1, included.

As we did in Section 3.1 for functions exp and log, we assume given an implementation of the base 2 logarithm, with the following property: for all x such that $0 < x \le M_{\log_2}$,

$$|\widehat{\log_2}(x) - \log_2(x)| \le E_{\log_2} |\log_2(x)|$$

where the parameter $M_{\log_2}$ is positive and the parameter $E_{\log_2}$ is positive and smaller than or equal to 1.

For readability, let us denote

$$g(x,y) = \exp\left(-\frac{(x+\rho-y)^2}{2}\right)$$

and its floating-point approximation

$$\hat{g}(x,y) = \widehat{\exp}(\ominus((((x\oplus\rho)\ominus y)\otimes((x\oplus\rho)\ominus y))\oslash 2))$$

Notice that trivially $0 < g(x,y) \le 1$ for any x,y.

The floating-point approximation of the SLSE function is as follows.

$$\widehat{\mathsf{SLSE}}(a) = \bigoplus_{0\le i<n} \widehat{\log_2}\left(\bigoplus_{0\le j<n} \hat{g}(a_i,a_j)\right)$$

To obtain a bound on the accuracy of $\widehat{\mathsf{SLSE}}$, we try to proceed in a more or less systematic way by stating lemmas of propagation of errors.

4.1 Propagation lemmas for addition and multiplication

Lemma 4.1 (Propagation of errors in a sum)**.** *Assuming $|\hat{x}-x| \le A\tilde{x}$ with $|x| \le \tilde{x}$ and $|\hat{y}-y| \le B\tilde{y}$ with $|y| \le \tilde{y}$, for any non-negative parameters A,B, we have*

$$|(\hat{x}\oplus\hat{y}) - (x+y)| \le (A+B+\varepsilon)(\tilde{x}+\tilde{y})$$

Proof. We first prove the following inequality:

$$|(\hat{x}\oplus\hat{y}) - (\hat{x}+\hat{y})| \le (\varepsilon+B)\tilde{x} + (\varepsilon+A)\tilde{y} \tag{7}$$

We distinguish three cases:

- if $\tilde{x} \le \varepsilon\tilde{y}$ then

$$\begin{aligned} |(\hat{x}\oplus\hat{y}) - (\hat{x}+\hat{y})| &\le |\hat{x}| && \text{(by Property (3))}\\ &\le A\tilde{x} + |x| \\ &\le A\tilde{x} + \tilde{x} \\ &\le A\varepsilon\tilde{y} + \varepsilon\tilde{y} \\ &\le (\varepsilon+B)\tilde{x} + (\varepsilon+A)\tilde{y} && \text{(because } \varepsilon \le 1\text{)} \end{aligned}$$

  which proves (7).

- if $\tilde{y} \leq \varepsilon\tilde{x}$, by using Property (4) and the same reasoning as above.
- otherwise, we have $\varepsilon\tilde{x} \leq \tilde{y}$ and $\varepsilon\tilde{y} \leq \tilde{y}$. We have

$$\begin{aligned}
|(\hat{x}\oplus\hat{y}) - (\hat{x}+\hat{y})| &\leq \varepsilon|\hat{x}+\hat{y}| && \text{(by Property (2))}\\
&\leq \varepsilon|(\hat{x}-x+x)+(\hat{y}-y+y)|\\
&\leq \varepsilon(|\hat{x}-x|+|x|+|\hat{y}-y|+|y|) && \text{(by triangular inequality)}\\
&\leq \varepsilon(A\tilde{x}+\tilde{x}+B\tilde{y}+\tilde{y})\\
&\leq A\tilde{y}+B\tilde{x}+\varepsilon(\tilde{x}+\tilde{y})\\
&\leq (\varepsilon+B)\tilde{x}+(\varepsilon+A)\tilde{y}
\end{aligned}$$

We now use Property (7) to prove our lemma.

$$\begin{aligned}
\hat{x}\oplus\hat{y} &\leq (\hat{x}+\hat{y})+(\varepsilon+B)\tilde{x}+(\varepsilon+A)\tilde{y}\\
&\leq x+A\tilde{x}+y+B\tilde{y}+(\varepsilon+B)\tilde{x}+(\varepsilon+A)\tilde{y}\\
&= (x+y)+(A+B+\varepsilon)(\tilde{x}+\tilde{y})
\end{aligned}$$

and similarly

$$\begin{aligned}
\hat{x}\oplus\hat{y} &\geq (\hat{x}+\hat{y})-(\varepsilon+B)\tilde{x}-(\varepsilon+A)\tilde{y}\\
&\geq x-A\tilde{x}+y-B\tilde{y}-(\varepsilon+B)\tilde{x}-(\varepsilon+A)\tilde{y}\\
&= (x+y)-(A+B+\varepsilon)(\tilde{x}+\tilde{y})
\end{aligned}$$

□

We devise a similar propagation lemma for multiplication, but first need to state and prove an auxiliary lemma.

Lemma 4.2. *Assuming $|\hat{x}-x| \leq A\tilde{x}$ with $|x| \leq \tilde{x}$ and $0 \leq A$, for any real z we have*

$$|\hat{x}z - xz| \leq A\tilde{x}|z|$$

Proof.

$$|\hat{x}z - xz| = |(\hat{x}-x)z| = |(\hat{x}-x)| \times |z| \leq A\tilde{x}|z|$$

□

Lemma 4.3 (Propagation lemma for multiplication)**.** *Assuming $|\hat{x}-x| \leq A\tilde{x}$ with $|x| \leq \tilde{x}$ and $|\hat{y}-y| \leq B|x|$ with $|y| \leq \tilde{y}$, for any non-negative parameters A, B, we have*

$$|(\hat{x}\otimes\hat{y}) - xy| \leq (\varepsilon + (A+B+AB)(1+\varepsilon))\tilde{x}\tilde{y} + \eta$$

Proof. We have

$$\begin{aligned}
\hat{x}\hat{y} &\leq x\hat{y} + A\tilde{x}|\hat{y}| && \text{(by Lemma 4.2 with } z = \hat{y})\\
&\leq xy + B|x|\tilde{y} + A\tilde{x}|\hat{y}| && \text{(by Lemma 4.2 with } z = x)\\
&\leq xy + B\tilde{x}\tilde{y} + A\tilde{x}|\hat{y}|
\end{aligned}$$

We can use Lemma 4.2 one last time with $z = \tilde{x}$ to obtain

$$\tilde{x}\hat{y} \leq \tilde{x}y + B\tilde{x}\tilde{y}$$

hence

$$\tilde{x}|\hat{y}| \leq \tilde{x}|y| + B\tilde{x}\tilde{y} \leq (1+B)\tilde{x}\tilde{y}$$

Combining it with inequality above, we get

$$\hat{x}\hat{y} \leq xy + (A+B+AB)\tilde{x}\tilde{y} \tag{8}$$

Finally, we have

$$\begin{aligned}
\hat{x} \otimes \hat{y} - xy &= (\hat{x} \otimes \hat{y} - \hat{x}\hat{y}) + (\hat{x}\hat{y} - xy) \\
&\leq (\varepsilon|\hat{x}\hat{y}| + \eta) + (\hat{x}\hat{y} - xy) && \text{(by Property (1))} \\
&\leq (\varepsilon|\hat{x}\hat{y}| + \eta) + (A+B+AB)\tilde{x}\tilde{y} && \text{(by (8))}
\end{aligned}$$

and since $|\hat{x}| \leq (1+A)\tilde{x}$ and $|\hat{x}| \leq (1+B)\tilde{y}$ we get

$$\begin{aligned}
\hat{x} \otimes \hat{y} - xy &\leq \varepsilon(1+A)(1+B)\tilde{x}\tilde{y} + \eta + (A+B+AB)\tilde{x}\tilde{y} \\
&= (\varepsilon + (1+\varepsilon)(A+B+AB))\tilde{x}\tilde{y} + \eta
\end{aligned}$$

We proceed similarly to prove the lower bound for $\hat{x} \otimes \hat{y} - xy$. □

4.2 Propagation Lemmas for Exponential and Logarithm

We have now error propagation lemmas for addition and multiplication, we continue by stating propagation lemmas for the exponential and the logarithm functions.

Lemma 4.4 (Propagation lemma for exponential)**.** *Assuming $|\hat{x} - x| \leq A\tilde{x} + C$ with $|x| \leq \tilde{x}$ and $|\hat{x}| \leq M_{\exp}$, for any non-negative parameters A, C, we have*

$$|\widehat{\exp}(\hat{x}) - \exp(x)| \leq \exp(x)(E_{\exp} + (\exp(A\tilde{x} + C) - 1)(1 + E_{\exp})))$$

Proof. We assume

$$\hat{x} \leq x + A\tilde{x} + C$$

hence, since exp is monotonic,

$$\exp(\hat{x}) \leq \exp(x + A\tilde{x} + C) = \exp(x)\exp(A\tilde{x} + C)$$

therefore

$$\exp(\hat{x}) \leq \exp(x) + \exp(x)(\exp(A\tilde{x} + C) - 1)$$

We have $\widehat{\exp}(\hat{x}) \leq \exp(\hat{x})(1 + E_{\exp})$ so

$$\widehat{\exp}(\hat{x}) \leq (\exp(x) + \exp(x)(\exp(A\tilde{x} + C) - 1))(1 + E_{\exp})$$

hence

$$\widehat{\exp}(\hat{x}) - \exp(x) \leq \exp(x)(E_{\exp} + (\exp(A\tilde{x} + C) - 1)(1 + E_{\exp})))$$

Similarly, for the lower bound:

$$\hat{x} \geq x - A\tilde{x} - C$$

therefore

$$\exp(\hat{x}) \geq \exp(x - A\tilde{x} - C) = \exp(x)\exp(-A\tilde{x} - C)$$

thus

$$\exp(\hat{x}) - \exp(x) \geq \exp(x)(\exp(-A\tilde{x} - C) - 1)$$

Furthermore, it is a known mathematical fact[2] that for any y, $\exp(y)+\exp(-y)\geq 2$, therefore

$$\exp(A\tilde{x}+C)+\exp(-A\tilde{x}-C)\geq 2$$

hence

$$-\exp(A\tilde{x}+C)+1\leq \exp(-A\tilde{x}-C)-1$$

so

$$\exp(x)(-\exp(A\tilde{x}+C)+1)\leq \exp(x)(\exp(-A\tilde{x}-C)-1)$$

therefore

$$-\exp(x)(\exp(A\tilde{x}+C)-1)\leq \exp(x)(\exp(-A\tilde{x}-C)-1)$$

thus

$$\exp(\hat{x})\geq \exp(x)-\exp(x)(\exp(A\tilde{x}+C)-1)$$

We have $\widehat{\exp}(\hat{x})\geq \exp(\hat{x})(1-E_{\exp})$ so

$$\widehat{\exp}(\hat{x})\geq (\exp(x)+\exp(x)(\exp(A\tilde{x}+C)-1))(1+E_{\exp})$$

hence

$$\widehat{\exp}(\hat{x})-\exp(x)\leq \exp(x)(E_{\exp}+(\exp(A\tilde{x}+C)-1)(1+E_{\exp})))$$

□

Lemma 4.5 (Propagation lemma for $\log_2$)**.** *For any positive real numbers x and $\widehat{x}$ such that $|\widehat{x}-x|\leq Ax+B$ with $B<x(1-A)$, and $|\hat{x}|\leq M_{\log_2}$, we have:*

$$\left|\widehat{\log_2}(\widehat{x})-\log_2(x)\right|\leq E_{\log_2}|\log_2(x)|-\log_2(1-(A+B/x))(1+E_{\log_2})$$

Proof. We have $\log_2(x)=\frac{\log(x)}{\log(2)}$, therefore by applying Lemma 3.3 we have

$$|\log_2(\widehat{x})-\log_2(x)|\leq -\log_2(1-(A+B/x))$$

We have $|\widehat{\log_2}(\widehat{x})-\log_2(\widehat{x})|\leq E_{\log_2}|\log_2(\widehat{x})|$, therefore, we can apply the triangular inequality to get

$$\begin{aligned}\widehat{\log_2}(\widehat{x})-\log_2(x)&\leq E_{\log_2}|\log_2(\widehat{x})|+|\log_2(\widehat{x})-\log_2(x)|\\&\leq E_{\log_2}|\log_2(x)|+(1+E_{\log_2})|\log_2(\widehat{x})-\log_2(x)|\\&\leq E_{\log_2}|\log_2(x)|-(1+E_{\log_2})\log_2(1-(A+B/x))\end{aligned}$$

and similarly for the lower bound. □

4.3 Accuracy of SLSE function

To achieve our main result on the accuracy of the SLSE function, we need a few hypotheses on the inputs. For the input vector a, we need to assumed the components bounded by some bound M_a. The constraint on this bound essentially comes from the fact that we need to call the exponential function on arguments which must be assumed smaller than $M_{\exp}$. Here is the assumption we pose:

$$(2M_a+1)^2(\frac{1}{2}+A_\varepsilon)+\frac{3}{2}\eta\leq M_{\exp} \tag{9}$$

[2] $\exp(y)-2+\exp(-y)=\exp(y)(1-2\exp(-y)+(\exp(-y))^2)=\exp(y)(1-\exp(-y))^2\geq 0$

where

$$A_\varepsilon = \frac{\varepsilon}{2}(1+4(1+\varepsilon)^2)$$

We are now ready to state and prove, as a first step, a result on the accuracy of the g auxiliary function, which as a reminder is defined as :

$$g(x,y) = \exp\left(-\frac{(x+\rho-y)^2}{2}\right)$$

Lemma 4.6 (Accuracy of the g function)**.** *Assuming x and y satisfying $|x| \leq M_a$ and $|y| \leq M_a$, we have*

$$|\hat{g}(x,y) - g(x,y)| \leq g(x,y)\left(E_{\exp} + \left(\exp\left(A_\varepsilon\left(|x|+\rho+|y|\right)^2 + \frac{3}{2}\eta\right) - 1\right)\left(1+E_{\exp}\right)\right)$$

Proof. Let $f(x,y) = x+\rho-y$ and $\hat{f}(x,y) = (x \oplus \rho) \ominus y$. From Property (2) we have

$$|(x \oplus \rho) - (x+\rho)| \leq \varepsilon|x+\rho|$$

hence from Lemma 4.1, with $\tilde{x} = |x+\rho|$, $\tilde{y} = |y|$, $A = \varepsilon$, $B = C = D = 0$, we have

$$|((x \oplus \rho) \ominus y) - ((x+\rho)-y)| \leq 2\varepsilon(|x+\rho| + |y|)$$

then, since $|(x+\rho)-y| \leq |x|+\rho+|y|$, using Lemma 4.3, with $\tilde{x} = \tilde{y} = |x|+\rho+|y|$, $A = B = 2\varepsilon$ we get

$$\begin{aligned}|(\hat{f}(x,y) \otimes \hat{f}(x,y)) - f(x,y)^2| &\leq (\varepsilon + (2\varepsilon + 2\varepsilon + 4\varepsilon^2)(1+\varepsilon))(|x|+\rho+|y|)^2 + \eta \\ &\leq 2A_\varepsilon(|x|+\rho+|y|)^2 + \eta\end{aligned}$$

Dividing by 2 is an exact operation except for de-normalised numbers for which a constant error of η can occur. Taking the opposite is an exact operation, therefore

$$|\ominus((\hat{f}(x,y) \otimes \hat{f}(x,y)) \oslash 2) - (-f(x,y)^2/2)| \leq A_\varepsilon\left(|x|+\rho+|y|\right)^2 + \frac{\eta}{2} + \eta$$

We need now to call the exponential function on the floating-point argument so we need to show that its absolute value is bounded by $M_{\exp}$. We have

$$\begin{aligned}|\ominus((\hat{f}(x,y) \otimes \hat{f}(x,y)) \oslash 2| &\leq |f(x,y)^2/2)| + A_\varepsilon\left(|x|+\rho+|y|\right)^2 + \frac{3}{2}\eta \\ &\leq \left(|x|+\rho+|y|\right)^2/2 + A_\varepsilon\left(|x|+\rho+|y|\right)^2 + \frac{\eta}{2} + \eta \\ &= \left(|x|+\rho+|y|\right)^2\left(\frac{1}{2} + A_\varepsilon\right) + \frac{3}{2}\eta \\ &\leq (2M_a+1)^2(\frac{1}{2}+A_\varepsilon) + \frac{3}{2}\eta \\ &\leq M_{\exp} \qquad \text{(by Hypothesis 9)}\end{aligned}$$

thus justifying the choice of Hypothesis 9. We can now apply Lemma 4.4 for error propagation by the exponential with $\tilde{x} = \left(|x|+\rho+|y|\right)^2$, $A = A_\varepsilon$ and $C = \frac{3}{2}\eta$ to get the expected result

$$|\hat{g}(x,y) - g(x,y)| \leq g(x,y)\left(E_{\exp} + \left(\exp\left(A_\varepsilon\left(|x|+\rho+|y|\right)^2 + \frac{3}{2}\eta\right) - 1\right)\left(1+E_{\exp}\right)\right)$$

□

To ease the reading of the remaining, let us introduce the constant

$$G = E_{\exp} + \left(\exp\left(A_\varepsilon\,(2M_a+1)^2 + \frac{3}{2}\eta\right) - 1\right)\left(1+E_{\exp}\right)$$

We state two auxiliary lemmas about G.

Lemma 4.7 (bound on G)**.** *We have the constant bound* $G \leq \frac{9}{16}$

Proof. From Hypothesis 9 we have

$$(2M_a+1)^2 \leq \frac{M_{\exp} - \frac{3}{2}\eta}{\frac{1}{2}+A_\varepsilon} \leq 2M_{\exp}$$

hence

$$G \leq E_{\exp} + \left(\exp\left(2A_\varepsilon M_{\exp}\right) - 1\right)\left(1+E_{\exp}\right)$$

We have

$$A_\varepsilon = \frac{\varepsilon}{2}(1+4(1+\varepsilon)^2) \leq 2^{-54}\times 6$$

and we assumed $M_{\exp}$ no larger than 708 so

$$2A_\varepsilon M_{\exp} \leq 2^{-53}\times 4248 \leq 2^{-53}\times 2^{12} = 2^{-41}$$

hence

$$\exp\left(2A_\varepsilon M_{\exp}\right) \leq 1 + 2^{-40}$$

hence, since we assumed $E_{\exp} \leq \frac{1}{2}$,

$$G \leq E_{\exp} + 2^{-40}(1+E_{\exp}) \leq \frac{1}{2} + 2^{-40}\frac{3}{2} \leq \frac{9}{16}$$

□

Lemma 4.8 (constant bounds on function g)**.** *for any x and y such that $|x| \leq M_a$ and $|y| \leq M_a$, with $0 \leq \rho \leq 1$,*

$$0 < \hat{g}(x,y) \leq 1+G$$

Proof. By definition of G and Lemma 4.6 we have

$$|\hat{g}(x,y) - g(x,y)| \leq G g(x,y)$$

hence

$$\hat{g}(x,y) \geq (1-G)g(x,y) > 0$$

and

$$\hat{g}(x,y) \leq (1+G)g(x,y) \leq 1+G$$

□

Our main theorem for the accuracy of the floating point implementation of SLSE is then as follows. To ease the reading of the formula, we introduce the term

$$|\mathsf{SLSE}|(a) = \sum_{0\leq i<n}\left|\log_2\left(\sum_{0\leq j<n} g(a_i,a_j)\right)\right|$$

which is actually a formula similar to $\mathsf{SLSE}(a)$ but with taking the absolute values for the elements of the outer sum.

Theorem 4.9 (Accuracy of SLSE)**.** *For any n such that* $0 < n \leq 2^{51}$*, any* ρ *such that* $0 \leq \rho \leq 1$ *and any vector a such that for all* a_i*,* $|a_i| \leq M_a$*, assuming Hypothesis 9, and finally assuming*

$$n\left(1+\left(G+(n-1)\varepsilon(1+G)\right)\right) \leq M_{\log_2} \tag{10}$$

we have

$$\begin{aligned}\left|\widehat{SLSE}(a) \;\; - SLSE(a)\right| \leq \left(E_{\log_2} + (n-1)\varepsilon(1+E_{\log_2})\right)|SLSE|(a)\\ -\log_2\left(1-\left(G+(n-1)\varepsilon\left(1+G\right)\right)\right)\left(1+E_{\log_2}\right)n\left(1+(n-1)\varepsilon\right)\end{aligned}$$

Proof. For any i and j such that $0 < i < n$ and $0 < j < n$ we invoke Lemma 4.6 to get

$$|\hat{g}(a_i,a_j) - g(a_i,a_j)| \leq g(a_i,a_j)\left(E_{\exp} + \left(\exp\left(A_\varepsilon\left(|a_i|+\rho+|a_j|\right)^2 + \frac{3}{2}\eta\right) - 1\right)\left(1+E_{\exp}\right)\right)$$

Note that the assumption of Lemma 4.6 is met because of the upper bound M_a on each $|a_i|$. Moreover, since $\rho \leq 1$ the factor on the right can be bounded by G. Therefore,

$$|\hat{g}(a_i,a_j) - g(a_i,a_j)| \leq g(a_i,a_j)G$$

so that we can invoke Lemma 3.2 with $A = G$ and $B = 0$, to get

$$\left|\bigoplus_{0\leq j<n} \widehat{g}(a_i,a_j) \;\; - \sum_{0\leq j<n} g(a_i,a_j)\right| \leq \left(G+(n-1)\varepsilon(1+G)\right)\sum_{0\leq j<n} g(a_i,a_j) \tag{11}$$

We would now like to invoke Lemma 4.5 with

$$\hat{x} = \bigoplus_{0\leq j<n} \widehat{g}(a_j,a_i),\; x = \sum_{0\leq j<n} g(a_j,a_i),\; A = G+(n-1)\varepsilon(1+G), B = 0$$

We need to prove that $\hat{x}$ is positive and smaller than $M_{\log_2}$, and to prove the hypothesis $B < x(1-A)$, which here is equivalent to say that x is positive (which is trivially true) and that $G+(n-1)\varepsilon(1+G) < 1$. From Lemma 4.8 we know than $\hat{x}$ is positive, and together with Property (11) we have

$$\begin{aligned}\hat{x} &\leq \left(1+\left(G+(n-1)\varepsilon(1+G)\right)\right)\sum_{0\leq j<n} g(a_i,a_j)\\ &\leq n\left(1+\left(G+(n-1)\varepsilon(1+G)\right)\right)\\ &\leq M_{\log_2} && \text{(from hypothesis)}\end{aligned}$$

Also, from Lemma 4.7 and since we assumed $n \leq 2^{51}$,

$$G+(n-1)\varepsilon(1+G) \leq \frac{9}{16} + 2^{51}\times 2^{-53}\frac{25}{16} = \frac{9}{16}+\frac{25}{64} = \frac{61}{64} < 1$$

That's it, we can apply Lemma 4.5 and get

$$\begin{aligned}&\left|\widehat{\log_2}\left(\bigoplus_{0\leq j<n}\widehat{g}(a_i,a_j)\right) - \log_2\left(\sum_{0\leq j<n} g(a_i,a_j)\right)\right|\\ &\quad\leq E_{\log_2}\left|\log_2\left(\sum_{0\leq j<n} g(a_i,a_j)\right)\right| - \log_2\left(1-\left(G+(n-1)\varepsilon\left(1+G\right)\right)\right)\left(1+E_{\log_2}\right)\end{aligned}$$

Finally, we can apply Lemma 3.2 with

$$\hat{a}_i = \widehat{\log_2}\left(\bigoplus_{0\le j<n} \widehat{g}(a_i,a_j)\right),\ a_i = \log_2\left(\sum_{0\le i<n} g(a_i,a_j)\right),\ A = E_{\log_2}$$

and

$$B = -\log_2\left(1-(G+(n-1)\varepsilon\,(1+G))\right)\left(1+E_{\log_2}\right)$$

to get

$$\begin{aligned}
&\left|\bigoplus_{0\le i<n} \widehat{\log_2}\left(\bigoplus_{0\le j<n} \widehat{g}(a_i,a_j)\right) \ - \sum_{0\le i<n} \log_2\left(\sum_{0\le j<n} g(a_i,a_j)\right)\right| \\
&\quad\le \left(E_{\log_2}+(n-1)\varepsilon(1+E_{\log_2})\right) \sum_{0\le i<n}\left|\log_2\left(\sum_{0\le j<n} g(a_i,a_j)\right)\right| \\
&\qquad -\log_2\left(1-(G+(n-1)\varepsilon\,(1+G))\right)\left(1+E_{\log_2}\right) n\,(1+(n-1)\varepsilon)
\end{aligned}$$

□

4.4 Discussion on the Variations of the Bound on Accuracy of SLSE

As we did in Section 3.3 for the LSE function, let us analyse how the bound given by Theorem 4.9 varies depending of the various parameters involved. That error bound comes in two parts:

- A (more or less) relative part, which is $E_{\log_2}+(n-1)\varepsilon(1+E_{\log_2})$. This is not properly speaking a relative error because it is a coefficient of $|\mathsf{SLSE}|(a)$ and not $|\mathsf{SLSE}(a)|$. Indeed, since the argument passed to $\log_2$ can be smaller or larger than 1, there is no way to avoid taking the absolute value of the $\log_2$. Unlike for the LSE function, this relative error depends on the size n of the input vector, whereas for LSE it was only depending on the relative error $E_{\log}$ of the log. That relative error coefficient is approximately $E_{\log_2}+n\varepsilon$, and grows linearly in $E_{\log_2}$ and n, but none of the two terms necessarily dominates the other.
- An absolute part, which is

  $$-\log_2\left(1-(G+(n-1)\varepsilon\,(1+G))\right)\left(1+E_{\log_2}\right) n\,(1+(n-1)\varepsilon)$$

  and can be approximated to

  $$2\,(G+(n-1)\varepsilon\,(1+G))\times 2n\times 2 \le 8n\,(G+2n\varepsilon)$$

  so the bound essentially depends linearly on the size of the vector. The factor of linearity is $8G$ so for the absolute error to remain small, the constant G should be kept small, meaning that $E_{\exp}$ should be kept small. See below for examples of values that can be chosen.

To give concrete values for the bound, let us assume, as in Section 3.3, a reasonable bound in practice on n that is 1024. We assume the input numbers are smaller than $M_a = 3$ in absolute value, so that $|a_i+\rho-a_j| \le 7$ and then the argument of the exponential is between $-\frac{49}{2}$ and 0, thus fitting with the assumption $M_{\exp} \le 25$ taken in Section 3.3. Indeed, with such values the hypothesis 9 requires $M_{\exp}$ to satisfy

$$(2M_a+1)^2(\frac{1}{2}+A_\varepsilon) \le M_{\exp} - \frac{3}{2}\eta$$

that is

$$M_{\exp} \ge \frac{3}{2}\eta + 49(\frac{1}{2}+A_\varepsilon)$$

- Let us assume first we have some implementations of exponential and logarithm with correct rounding, that is $E_{\exp} = 2^{-53}$ and $E_{\log} = 2^{-53}$. With these values we have

$$A_\varepsilon = \frac{\varepsilon}{2}(1+4(1+\varepsilon)^2) \leq 3\times 2^{-53}$$

the constant G satisfy

$$\begin{aligned} G &= E_{\exp} + \left(\exp\left(A_\varepsilon\,(2M_a+1)^2 + \frac{3}{2}\eta\right) - 1\right)\left(1+E_{\exp}\right) \\ &\leq 2^{-53} + \left(\exp\left(49\times 3\times 2^{-53} + \frac{3}{2}2^{-1074}\right) - 1\right)\left(1+2^{-53}\right) \\ &\leq 2^{-53}(1+2\times 49\times 3) \\ &\leq 2^{-44} \end{aligned}$$

To ensure hypothesis 10 of Theorem 4.9 holds, it suffices to have $M_{\log_2}$ larger than

$$\begin{array}{rcl} n(1+G+(n-1)\varepsilon(1+G)) & \leq & 2^{10}(1+2^{-44}) + 2^{10}\times 2^{-53}(1+2^{-44}) \\ & \leq & 1025 \end{array}$$

indeed nothing surprising here, the sum taken as argument of $\widehat{\log_2}$ is a sum of at most 1024 numbers between 0 and 1. Thus, assuming that the implementation of $\widehat{\log_2}$ is correctly rounded on the domain given by the bound $M_{\log_2}$ above, the relative error on SLSE is bounded by

$$E_{\log_2} + (n-1)\varepsilon(1+E_{\log_2}) \leq 2^{10}\times 2\varepsilon = 2^{-43} = 1.14\times 10^{-13}$$

and the absolute error is bounded by

$$8n\,(G+2n\varepsilon) \leq 2^{13}(2^{-44}+2^{-42}) = 2.33\times 10^{-9}$$

- Let us assume less precise implementations of exponential and logarithm with $E_{\exp} = 2^{-40}$ and $E_{\log_2} = 2^{-36}$. With these values the constant G satisfy

$$\begin{aligned} G &= E_{\exp} + \left(\exp\left(A_\varepsilon\,(2M_a+1)^2 + \frac{3}{2}\eta\right) - 1\right)\left(1+E_{\exp}\right) \\ &\leq 2^{-40} + \left(\exp\left(49\times 3\times 2^{-53} + \frac{3}{2}2^{-1074}\right) - 1\right)\left(1+2^{-40}\right) \\ &\leq 2^{-40}(1+2\times 49\times 3) \\ &\leq 2^{-31} \end{aligned}$$

To ensure hypothesis 10 of Theorem 4.9 holds, it suffices to have $M_{\log_2}$ larger than

$$\begin{array}{rcl} n(1+G+(n-1)\varepsilon(1+G)) & \leq & 2^{10}(1+2^{-31}) + 2^{10}\times 2^{-53}(1+2^{-31}) \\ & \leq & 1025 \end{array}$$

it should not surprising again. The relative error on SLSE is bounded by

$$E_{\log_2} + (n-1)\varepsilon(1+E_{\log_2}) \leq 2^{10}\times 2\varepsilon = 1.14\times 10^{-13}$$

so essentially the same as for correctly rounded exponential and logarithm, and the absolute error is bounded by

$$8n\,(G+2n\varepsilon) \leq 2^{13}(2^{-31}+2^{-42}) = 3.82\times 10^{-6}$$

which is this time very significantly larger than with the correctly rounded implementation of exponential.

A general conclusion is that the dominant part of the accuracy of SLSE is coming from the accuracy of the implementation of the exponential. It is also more or less linear in the size of the input vector.

5 Discussions, Related Work and Future Work

We presented bounds on accuracy of implementations of LSE ans SLSE functions. The resulting expressions for the bounds are thus parametric in the precision of the underlying implementations of exp, log and log, and also parameterised 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, for LSE, in WhyML, using Why3 constructs such as lemma function to provide proofs that follow more or less the paper proofs. The bound for LSE is then also proved on some C implementation, the proof reusing the results already 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.

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.* [9]. They bound the rounding errors of the LSE function and its gradient, namely the softmax function. The exhibited bounds 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:

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

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 contrast, we only focus on the standard implementation. Nonetheless, our main strength compared to this work is the fact that we give a formal proof of the error bound.

To our knowledge, the mutual information algorithm has never been formally-verified. Regarding similar applications, Affeldt and Garrigue [1] have proposed a formalisation of the theory of linear error-correcting codes in the Coq proof assistant, with an application to Hamming codes and the verification of an algorithm called the sum-product algorithm, which is used for decoding LDPC codes. However, the studied algorithms do not make use of mathematical functions and the authors do not deal with rounding errors.

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 [4] have combined various tools, namely the Verifiable Software Toolchain (VST) [3], Flocq [18, 19] and Gappa [27], to perform such a formal verification process. They applied their methodology to an example of square root implementation using the Newton method. The approach is modular and the different tools are used in different layers of the verification. VST (which has been proven correct inside Coq) ensures the correctness of the C code. Flocq and Gappa are used as backend tools to check numerical accuracy facts.

Kellison *et al.* [40] 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. These proofs can typically be connected to BLAS (Basic Linear Algebra Subprograms) low-level routines implementations, so that LAProof can serve as a proof layer between low-level BLAS implementations and numerical

programs verification. As an application example, the authors prove a C program computing a sparse matrix-vector multiplication using the VST [3] approach and the LAProof library.

Boldo *et al.* [11, 12] formalised a numerical integration scheme for a wave partial differential equation in Coq. They not only formally proved a bound on the mathematical errors [12], but also a bound on the rounding errors [11]. These works have been used as a basis for the formal verification of a wave equation resolution C program [13]. For that purpose, they have used the Jessie plug-in of Frama-C, that discharged some of the proof obligations to various automated provers, e.g. Gappa or Alt-Ergo. More complex proof obligations, mainly related to numerical errors, are discharged to Coq.

Becker *et al.* [8] developed a CakeML extension for optimising floating-point arithmetic in Standard ML. Such optimisations generally do not preserve the IEEE-754 semantics, which avoids a proof restricted to the arithmetic expression under optimisation. Instead, their approach relies on an end-to-end soundness proof linking a real-number specification of the initial code with the code obtained after optimisation. The approach, entirely automated (code and proof generation), targets the optimisation 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).

Future work. In practice, most applications using the LSE function rely on the shifted version presented by Blanchard *et al.* [9]. 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 for our work.

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 [24] or COPRIN projects [46]. 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 [49]. To complement our work, we envision the formal verification of the errors induced by such implementations. In the late 20^{th} century, Harrison [34, 35] 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 [27] has been partly used to bound rounding errors of floating-point implementations of functions from the CR-LIBM library [25, 29, 28]. 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. Providing a methodology and tooling to ease the formal proofs of low-level floating-point components with a minimal user effort would be valuable, especially to bound the total errors of functions implementations. A longer-term goal could be the development of a formally verified implementation synthesis tool, in the spirit of the Metalibm tools [41, 21]. Metalibm already enables the generation of Gappa scripts certifying the synthesised implementations, but this feature is limited to some pieces of code.

Regarding the proof methodology, there are remaining issues that would deserve future work. The theory of unbounded floats that we designed is a good candidate to be integrated in the standard library of Why3. Yet, the proofs still require a large amount of manual steps, it is desirable to automate the

process. We are currently planing to automatise the application of "forward propagation lemmas" that were becoming visibly pervasive in the proof of SLSE, and should be naturally used for proving any code proceeding by composing numerical functions and operations. Concerning the proof of the C code, the current methodology is far from being usable by non-expert users. 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, 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] Reynald Affeldt and Jacques Garrigue. Formalization of error-correcting codes: From Hamming to modern coding theory. In *Interactive Theorem Proving*, volume 9236 of *Lecture Notes in Computer Science*, pages 17–33. Springer, 2015. doi:10.1007/978-3-319-22102-1_2.

[2] 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.

[3] 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.

[4] 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/.

[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. URL: https://doi.org/10.4230/LIPIcs.ECOOP.2022.1, doi:10.4230/LIPICS.ECOOP.2022.1.

[9] 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.

[10] 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.lri.fr/gallery/fm2012comp.en.html. URL: http://hal.inria.fr/hal-00967132/en, doi:10.1007/s10009-014-0314-5.

[11] 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.

[12] 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/.

[13] 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.

[14] 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: http://www.lri.fr/~filliatr/ftp/publis/caduceus-floats.pdf, doi:10.1109/ARITH.2007.20.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] Vincent Corlay and Nicolas Gresset. A simple sign-bit probabilistic shaping scheme. *IEEE communications letters*, 26(4):763–767, 2022. doi:10.1109/LCOMM.2022.3144610.

[24] 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.

[25] 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.

[26] 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`.

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

[28] 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.

[29] 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.

[30] 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`.

[31] 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/`.

[32] 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.

[33] 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`.

[34] 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.

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

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

[37] 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`.

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

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] Érik Martin-Dorel and Guillaume Melquiond. Proving tight bounds on univariate expressions with elementary functions in Coq. *Journal of Automated Reasoning*, 2016. URL: https://hal.inria.fr/hal-01086460, doi:10.1007/s10817-015-9350-4.

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

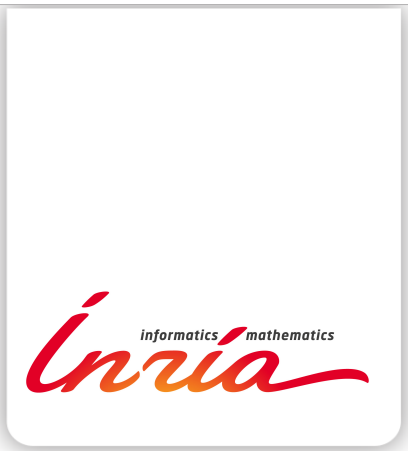
[45] Guillaume Melquiond. *Formal Verification for Numerical Computations, and the Other Way Around*. Habilitation à diriger des recherches, Université Paris Sud, April 2019. URL: https://tel.archives-ouvertes.fr/tel-02194683.

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

[47] 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.

[48] 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.

[49] 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.



RESEARCH CENTRE
SACLAY – ÎLE-DE-FRANCE

1 rue Honoré d'Estienne d'Orves
Bâtiment Alan Turing
Campus de l'École Polytechnique
91120 Palaiseau

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399