



HAL
open science

End-to-End Formal Verification of a Fast and Accurate Floating-Point Approximation

Florian Faissole, Paul Geneau de Lamarlière, Guillaume Melquiond

► **To cite this version:**

Florian Faissole, Paul Geneau de Lamarlière, Guillaume Melquiond. End-to-End Formal Verification of a Fast and Accurate Floating-Point Approximation. 2024. hal-04515714

HAL Id: hal-04515714

<https://hal.science/hal-04515714>

Preprint submitted on 21 Mar 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

1 End-to-End Formal Verification of a Fast and 2 Accurate Floating-Point Approximation

3 **Florian Faissole** ✉

4 Mitsubishi Electric R&D Centre Europe, 35700 Rennes, France

5 **Paul Geneau de Lamarlière** ✉

6 Mitsubishi Electric R&D Centre Europe, 35700 Rennes, France

7 Université Paris-Saclay, CNRS, ENS Paris-Saclay, Inria, LMF, 91190 Gif-sur-Yvette, France

8 **Guillaume Melquiond** ✉ 

9 Université Paris-Saclay, CNRS, ENS Paris-Saclay, Inria, LMF, 91190 Gif-sur-Yvette, France

10 — Abstract —

11 Designing an efficient yet accurate floating-point approximation of a mathematical function is an
12 intricate and error-prone process. This warrants the use of formal methods, especially formal proof,
13 to achieve some degree of confidence in the implementation. Unfortunately, the lack of automation
14 or its poor interplay with the more manual parts of the proof makes it way too costly in practice.
15 This article revisits the issue by proposing a methodology and some dedicated automation, and
16 applies them to the use case of a faithful *binary64* approximation of exponential. The peculiarity of
17 this use case is that the target of the formal verification is not a simple modeling of an external
18 code, it is an actual floating-point function defined in the logic of the Coq proof assistant, which is
19 thus usable inside proofs once its correctness has been fully verified. This function presents all the
20 attributes of a state-of-the-art implementation: bit-level manipulations, large tables of constants,
21 obscure floating-point transformations, exceptional values, etc. This function has been integrated
22 into the proof strategies of the CoqInterval library, bringing a 20× speedup with respect to the
23 previous implementation.

24 **2012 ACM Subject Classification** Software and its engineering → Formal software verification;
25 Theory of computation → Interactive proof systems; Theory of computation → Automated reasoning;
26 Mathematics of computing → Mathematical software performance; Mathematics of computing →
27 Interval arithmetic

28 **Keywords and phrases** Program verification, floating-point arithmetic, formal proof, automated
29 reasoning, mathematical library

30 **Digital Object Identifier** 10.4230/LIPIcs...

31 **1** Introduction

32 The CoqInterval library¹ provides a set of strategies for the Coq proof assistant that
33 automatize the formal verification of enclosures of real-valued expressions. It is based on a
34 formalization of rigorous polynomial approximations that are computed using an interval
35 arithmetic with floating-point bounds [9]. Originally, the floating-point computations were
36 performed one bit at a time in the logic of the Coq system. But now that hardware floating-
37 point computations can be performed inside Coq proofs, CoqInterval’s strategies can delegate
38 some computations to the floating-point unit of the processor, thus greatly speeding up the
39 proof checking [10]. This makes it possible to formally verify the following approximation of
40 Siegfried Rump’s integral, which is known to cause computer algebra systems to struggle

¹ <https://coqinterval.gitlabpages.inria.fr/>



© Author: Please provide a copyright holder;

licensed under Creative Commons License CC-BY 4.0

Leibniz International Proceedings in Informatics

LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

41 due to the large number of oscillations of the integrand, in a handful of seconds:

$$42 \quad \int_0^8 \sin(x + \exp x) dx = 0.3474 \pm 10^{-6}.$$

43 Among numerous other things, this proof requires the ability to compute a floating-point
 44 enclosure of the mathematical value $\exp x$ for x some floating-point number. Thanks to a
 45 suitable abstraction of floating-point arithmetic [10], CoqInterval uses the same algorithm
 46 for both floating-point numbers computed in hardware and floating-point numbers slowly
 47 emulated in the logic of Coq.² Having a single algorithm, and thus a single proof of correctness,
 48 for both implementations of floating-point arithmetic made the large formalization effort that
 49 went into adding hardware computations to CoqInterval much less tedious. The algorithm
 50 for computing an enclosure of $\exp x$ goes as follows. First, using the following mathematical
 51 identities, an argument reduction brings the input x into the interval $[-2^{-8}; 0]$:

$$52 \quad \begin{aligned} \exp x &= (\exp(-x))^{-1} \\ \exp x &= (\exp(x/2))^2 \end{aligned} \tag{1}$$

54 Second, the alternating series $\exp(-x) = \sum (-x)^n/n!$ is computed using interval arithmetic
 55 to a high enough order. Thanks to the use of interval arithmetic and an alternating series,
 56 the algorithm is guaranteed to compute an enclosure of the real number $\exp x$. Third, the
 57 argument reduction is inverted to reconstruct the final interval result.

58 By suitably choosing the order of truncation of the series, one can obtain arbitrarily
 59 tight enclosures of $\exp x$, assuming that the precision of the floating-point arithmetic used
 60 to compute the interval bounds can be made accordingly large. This property is invaluable
 61 when used in conjunction with the original multi-precision floating-point arithmetic of
 62 CoqInterval. But for hardware floating-point numbers and their fixed precision of 53 bits,
 63 the property is pointless. The inadequacies of the implementation of \exp then become
 64 prominent. First, Equation (1) means that computing an enclosure of $\exp x$ is not constant
 65 time, but proportional to the magnitude of x . Second, an alternating series is the worst way
 66 of approximating a value, as part of the computations performed at order i are immediately
 67 canceled by those at order $i + 1$ and thus have been performed in vain. Third, while interval
 68 arithmetic is correct by construction, hence very proof-friendly, it performs twice as many
 69 floating-point operations as needed.

70 Outside of a formal system, a state-of-the-art approximation of the exponential function
 71 using hardware floating-point numbers would be implemented along the following guidelines [12,
 72 §6.2]. It would first perform a constant-time argument reduction using the following
 73 mathematical identity:

$$74 \quad \exp x = \exp(x - k \cdot \ln 2) \cdot 2^k \quad \text{with } k = \lceil x/\ln 2 \rceil \in \mathbb{Z}.$$

75 Then, a low-degree polynomial approximation of \exp around 0 would be evaluated. Finally,
 76 the result reconstruction is trivial, as it is a multiplication by a power of two. The whole
 77 algorithm amounts to just a few tens of operations; it is thus extremely fast. The issue
 78 now is that one needs to formally verify the adequacy of the result, as it no longer derives
 79 from the use of interval arithmetic. In particular, if one wants to use such a state-of-the-art
 80 implementation of \exp in place of the one currently in CoqInterval, the proof effort needs to
 81 be sufficiently light so that it is worth replacing a feature that is already good enough for
 82 most use cases.

² This emulation is still useful for proofs that require more than the 53 bits of precision provided by the *binary64* format.

```

let iexp x =
  if x < -0x1.74385446d71c4p9 then (0., 0x1.p-1074) else
  if x > 0x1.62e42fefa39efp9 then (0x1.fffffffffff2ap1023, infinity) else
  let k' = x *. invLog2_64 +. 0x1.8p52 in
  let k = k' -. 0x1.8p52 in
  let t = (x -. k *. log2_64h) -. k *. log2_64l in
  let y = t *. (p1 +. t *. (p2 +. t *. (... + t *. p5))) in
  let ki = int_of_float k' - 0x18000000000000 in
  let p0 = cst.(ki land 63) in
  let lb = p0 +. (p0 *. y -. d) in
  let ub = p0 +. (p0 *. y +. d) in
  next_down (ldexp lb (ki asr 6)), next_up (ldexp ub (ki asr 6))

```

■ **Figure 1** Guaranteed approximation of exponential in OCaml. The output is a pair of floating-point numbers that enclose $\exp x$. Symbols `invLog2_64`, `log2_64h`, `log2_64l`, `d`, `cst`, `p1`, `p2`, etc., are predefined floating-point literals; in particular, `d` is about $3 \cdot 2^{-58}$.

83 Figure 1 shows the implementation we have devised, represented as an OCaml function
 84 for readability. (Its translation to Coq’s λ -calculus is straightforward.) Given a finite floating-
 85 point number x , the code computes a pair of floating-point numbers that enclose $\exp x$. In
 86 particular, it uses the functions `next_down` and `next_up` to compute the predecessor and
 87 successor of a floating-point number.

88 While the code seems to contain useless, if not adverse, floating-point computations, this is
 89 not the case. For example, `k` looks like it could be directly computed as `x *. invLog2_64` by
 90 canceling `0x1.8p52 -. 0x1.8p52`. This optimization would completely break the function,
 91 causing it to no longer approximate the exponential, not even roughly. Similarly, `t` should not
 92 be rewritten as `x -. k *. (log2_64h +. log2_64l)`, and `d` should not be moved outside of
 93 the parentheses in the computations of `lb` and `ub`. So, not only does floating-point arithmetic
 94 not respect the usual algebraic laws of associativity and distributivity, floating-point experts
 95 actively rely on the lack of these laws to compute more accurate approximations.

96 As for the accuracy of the code, one can get an intuitive feel of it by considering the
 97 distance between both components of the returned pair. Ideally, the distance should be one
 98 unit in the last place (*ulp*), as the components should be consecutive floating-point numbers
 99 for $x \neq 0$. This property, called *correct rounding* [12, §12.3], is still an open research question
 100 for floating-point formats larger than *binary32* and completely out of reach of a formal proof,
 101 as of today. So, the best we can hope to achieve is a distance of up to two ulps, that is,
 102 one component is optimal, while the other is off-by-one. If the constant `d` was zero, this
 103 would be the case. As it is not quite zero here, when $\exp x$ is close to the midpoint between
 104 two consecutive floating-point numbers, the distance might end up being three ulps. The
 105 proportion of inputs that cause a 3-ulp interval output is $d \cdot 2^{51} \simeq 1/40$.

106 There have been several attempts at formally verifying this kind of state-of-the-art
 107 implementation using the Coq proof assistant, but they all have suffered from various
 108 shortcomings. It might have been that the floating-point arithmetic was modeled without any
 109 exceptional value [3, §6.2.3]. Indeed, when a computer-assisted proof is meant to complement
 110 a pen-and-paper proof, it is acceptable that it only focuses on the most intricate parts of the
 111 proof, which the absence of exceptional behavior is hardly ever. But, since this idealized
 112 arithmetic does not match the behavior of hardware floating-point numbers, it cannot be
 113 used here. Some later attempt solved the issue of the exceptional values [6], but it was still

114 targeting the verification of some code meant to run outside of Coq and thus did not need
 115 to cover all of its facets. For example, it was ignoring the first and last few steps of the
 116 algorithm, so as to focus on the important part. It was also assuming the existence of some
 117 higher-level functions like `nearbyint`, which are either missing or too slow in our setting.
 118 Moreover, it was ignoring the issue of accuracy on subnormal outputs, where it is ill-defined,
 119 but that we cannot just disregard. Finally, the code was not as intricate as the one presented
 120 in Figure 1: no array accesses, no mixed integer/floating-point operations, etc.

121 The novelty of the work presented in this article is thus the full verification of a state-of-
 122 the-art floating-point implementation of a mathematical function. This verification really
 123 covers all the facets, since the algorithm is not just modeled, but it is an actual code that
 124 will effectively be run when checking subsequent Coq proofs, so absolutely no shortcuts can
 125 be taken. This article also presents all the automated strategies that were added to make
 126 this verification as painless as possible. Indeed, the exponential function is just the first step;
 127 the goal is to optimize all the mathematical functions of CoqInterval using the presented
 128 methodology.

129 Section 2 reminds both the arithmetic language and the notion of well-behaved expression
 130 that were introduced in a previous work [6]. Section 3 explains how some strategies of
 131 CoqInterval have been improved to automatically verify properties involving tight bounds
 132 on rounding errors. Section 4 details the new features added to the arithmetic language
 133 and associated tools to tackle the algorithm of Figure 1: hardware floating-point numbers,
 134 conversions, macro-operations, array accesses, etc. Section 5 describes the methodology
 135 used to formally verify the correctness of exponential, as well as some unusual properties of
 136 floating-point arithmetic we ended up with. Section 6 explains how this work relates to some
 137 other works. Finally, Section 7 concludes with some benchmarks and some perspectives.

138 2 Preliminaries

139 This work is partly built on top of a framework for modeling floating-point expressions [6].
 140 In particular, that framework provides some facilities to automate the proof of the absence of
 141 exceptional behaviors, thus making it possible for the user to focus on a modeling of floating-
 142 point expressions as real numbers. This section reminds the features of that framework that
 143 are the most relevant to the presented work. Section 2.1 focuses on the expressions and their
 144 various interpretations, while Section 2.2 shows how one can jump between interpretations
 145 to ease the proof process.

146 In the following, the unary operator $\circ(\cdot)$ designates a rounding operator from \mathbb{R} to \mathbb{R} ; it
 147 returns the real number the nearest to the input that fits in the target floating-point format
 148 (with unlimited range) [3, §3.2.2]. This theoretical operator is at the core of the IEEE-754
 149 standard for floating-point arithmetic.

150 2.1 Arithmetic expressions

151 An arithmetic expression e is represented as the value of an inductive type corresponding to
 152 a typed abstract syntax tree, namely an expression tree [6]. The nodes of an expression tree
 153 correspond to arithmetic expressions, including floating-point operations, integer operations,
 154 and some functions such as `nearbyint`.

155 The expression e can then be interpreted in several ways, two of which are relevant
 156 here. First, it can be interpreted as the floating-point number $\llbracket e \rrbracket_{\text{ft}}$ that would be obtained
 157 according to the IEEE-754 standard. Second, e can be interpreted as the value $\llbracket e \rrbracket_{\text{rnd}}$ obtained
 158 by performing all the operations on real numbers and rounding their results. For example, in

159 the case of the floating-point addition, we have $\llbracket u + v \rrbracket_{\text{rnd}} = \circ(\llbracket u \rrbracket_{\text{rnd}} + \llbracket v \rrbracket_{\text{rnd}})$. In the case of
 160 integer operations, $\llbracket e \rrbracket_{\text{flt}}$ performs computations modulo a power of two, while $\llbracket e \rrbracket_{\text{rnd}}$ performs
 161 operations on unbounded integers, *e.g.*, $\llbracket u + v \rrbracket_{\text{rnd}} = \llbracket u \rrbracket_{\text{rnd}} + \llbracket v \rrbracket_{\text{rnd}}$. The first interpretation
 162 corresponds to the value actually computed by an implementation, and therefore the value
 163 on which we need to prove a correctness theorem. The second interpretation, however, is the
 164 one that is the more amenable to formal reasoning, as it is not susceptible to exceptional
 165 behaviors such as overflows.

166 There are two features of expression trees that are of interest to us. The first one is the
 167 support for let-binding operators, with binders represented by their de Bruijn indices, to
 168 express sharing between sub-expressions and to guide proofs. The second one is the availability
 169 of exact arithmetic operations, as they are commonly encountered in implementation of
 170 mathematical functions. As far as $\llbracket \cdot \rrbracket_{\text{flt}}$ is concerned, there is no difference in interpretation
 171 between exact and inexact operations over floating-point numbers; they are performed as
 172 mandated by the IEEE-754 standard. For $\llbracket \cdot \rrbracket_{\text{rnd}}$, exact arithmetic operations, however, are
 173 not rounded, which makes formal proofs, both manual and automatic, much simpler. This
 174 rises the concern of whether such a proof about $\llbracket e \rrbracket_{\text{rnd}}$ is meaningful, which Theorem 1 below
 175 will tackle.

176 Let us illustrate these two features on the example of the argument reduction of the Cody-
 177 Waite exponential [4], variants of which are still widely used in modern implementations,
 178 including in the code shown in Figure 1:

```
179   k ← nearbyint(x · C),
180   t ← x - k · c1 - k · c2,
```

182 with $c_1 + c_2 \simeq 1/C$ and $c_2 \ll c_1$. Below is the formulation of this argument reduction as an
 183 expression tree.

```
184   Let (NearbyInt (Op MUL (Var 0) (BinFl C)))
185   (Op SUB
186     (OpExact SUB (Var 1) (OpExact MUL (Var 0) (BinFl c1)))
187     (Op MUL (Var 0) (BinFl c2)))
```

190 Notice that both floating-point operations in $x - k \cdot c_1$ are annotated as exact operations
 191 by using the `OpExact` constructor. All the other operations are marked as potentially inexact
 192 (`Op` constructor). This gives the following value for $\llbracket t \rrbracket_{\text{rnd}}$:

193 $\circ(x - k \cdot c_1 - \circ(k \cdot c_2))$ with $k = \lfloor \circ(x \cdot C) \rfloor$.

194 2.2 Relation between interpretations

195 As mentioned earlier, a correctness statement is about $\llbracket e \rrbracket_{\text{flt}}$, while a user only wants to have to
 196 deal with $\llbracket e \rrbracket_{\text{rnd}}$, as it is free of exceptional behaviors and contains fewer rounding operations.
 197 In order to bridge the gap between both interpretations, a predicate `WB` (for *well-behaved*)
 198 is defined recursively over expressions. For example, the proposition `WB(Op DIV u v)` is
 199 defined as

200 $\text{WB}(u) \wedge \text{WB}(v) \wedge \llbracket v \rrbracket_{\text{rnd}} \neq 0 \wedge |\circ(\llbracket u \rrbracket_{\text{rnd}} / \llbracket v \rrbracket_{\text{rnd}})| \leq \Omega$

201 with Ω the value of the largest finite floating-point number. In other words, for the floating-
 202 point division u/v to be well-behaved, it is sufficient that u and v are well-behaved, that the
 203 interpretation of v as a real number is non-zero, and that the division over real numbers,

204 once rounded, does not overflow the floating-point format. The predicate WB is defined
 205 in a similar way for the other inexact operations over floating-point numbers. For exact
 206 operations, the formula contains an additional conjunct that states that the result is exactly
 207 representable. For example, the proposition $\text{WB}(\text{OpExact ADD } u \ v)$ is defined as

$$208 \quad \text{WB}(u) \wedge \text{WB}(v) \wedge \circ(\llbracket u \rrbracket_{\text{rnd}} + \llbracket v \rrbracket_{\text{rnd}}) = \llbracket u \rrbracket_{\text{rnd}} + \llbracket v \rrbracket_{\text{rnd}} \wedge |\llbracket u \rrbracket_{\text{rnd}} + \llbracket v \rrbracket_{\text{rnd}}| \leq \Omega.$$

209 The key result is that, if an expression e is well-behaved, then $\llbracket e \rrbracket_{\text{flt}}$ is a finite floating-point
 210 number and it represents the real number $\llbracket e \rrbracket_{\text{rnd}}$. This is expressed by the following theorem:

211 ► **Theorem 1.** *Given an expression e , $\text{WB}(e) \Rightarrow \llbracket e \rrbracket_{\text{flt}} \text{ finite} \wedge \llbracket e \rrbracket_{\text{flt}} = \llbracket e \rrbracket_{\text{rnd}}$.*

212 When applying Theorem 1, the user is left with a subgoal $\text{WB}(e)$, which is painful to
 213 prove by hand. So, to ease the proof process, the framework proposes a proof strategy called
 214 `simplify_wb`, which tackles this subgoal by applying a procedure similar to CoqInterval’s
 215 `interval` strategy to every conjunct of $\text{WB}(e)$ individually. In practice, one can expect all
 216 the conjuncts related to the absence of exceptional behaviors to be automatically proved.
 217 Conjuncts related to exact operations, *e.g.*, $\circ(\llbracket u \rrbracket_{\text{rnd}} + \llbracket v \rrbracket_{\text{rnd}}) = \llbracket u \rrbracket_{\text{rnd}} + \llbracket v \rrbracket_{\text{rnd}}$, are however
 218 out of the scope of CoqInterval. So, the user will have to prove them either manually or
 219 using a dedicated tool like Gappa [3, §4.3].

220 Note that, in order for `simplify_wb` to make use of the `interval` strategy of CoqInterval,
 221 the latter had to be enhanced with some support for rounding operators, as they appear in
 222 almost all the conjuncts of $\text{WB}(e)$. This support was based on the so-called *standard model*
 223 of floating-point arithmetic. (Section 3.2 will propose a better approach.) For instance, given
 224 an enclosure $u \in [\underline{u}; \bar{u}]$, CoqInterval would compute an enclosure of $\circ(u)$ as follows, assuming
 225 a *binary64* format:

$$226 \quad \circ(u) \in [\underline{u} - \varepsilon; \bar{u} + \varepsilon] \quad \text{with } \varepsilon = \max(2^{-1075}, -2^{-53} \cdot \underline{u}, +2^{-53} \cdot \bar{u}). \quad (2)$$

227 **3 Automated tools and rounding errors**

228 Since support for rounding operators had been added to CoqInterval to suit `simplify_wb` [6],
 229 it seemed like it could be used for more than just automatically proving some conjuncts of
 230 $\text{WB}(e)$. In particular, it should be possible to automatically prove a bound on the error
 231 between a rounded expression and the ideal expression it supposedly approximates. Consider
 232 the code of Figure 1. An intermediate property that is needed to prove its correctness is the
 233 following one:

$$234 \quad \forall t \in \mathbb{R}, |t| \leq 355 \cdot 2^{-16} \Rightarrow |1 + y - \exp t| \leq 2^{-58} \quad \text{with } y = \circ(t \cdot \circ(p_1 + \circ(t \cdot \circ(p_2 + \dots)))). \quad (3)$$

235 Prior to this work, the methodology to prove the bound would have been to decompose
 236 the expression $1 + y - \exp t$ into two parts $e_1 + e_2$ [3, §6.2.3]: $e_1 = y - t \cdot (p_1 + t \cdot (p_2 + \dots))$
 237 and $e_2 = (1 + t \cdot (p_1 + t \cdot (p_2 + \dots))) - \exp t$. Expression e_2 (no rounding operator) would then
 238 be bounded using the rigorous polynomial approximations of CoqInterval, while expression e_1
 239 (no exponential) would be bounded using the Gappa tool [3, §4.3]. This would give a proof
 240 of the expected bound 2^{-58} . Our goal is to perform this proof directly using CoqInterval,
 241 without any need for such algebraic manipulations nor the use of an external tool.

242 Unfortunately, several issues arise when using the `interval` strategy on Equation (3).
 243 First of all, both sides of the subtraction are strongly correlated by definition, since the
 244 left-hand side was chosen among the best possible floating-point approximations of the
 245 right-hand side $\exp t$. This means that naive interval arithmetic, as used in Equation (2) to

246 define the enclosure of a rounding operator, will cause an overestimation of the final enclosure
 247 that is so large that it becomes useless for proving anything interesting. On this example,
 248 the strategy would only be able to prove that the error is bounded by 10^{-2} , very far from the
 249 expected bound of 2^{-58} . So, the first step is to define rigorous polynomial approximations
 250 for rounding operators (Section 3.1).

251 This is not sufficient though, as the strategy would only succeed in proving a bound of
 252 about $1.5 \cdot 2^{-58}$, which is already quite good, but not sufficient to prove the correctness of
 253 the whole code. This overestimation is a consequence of the simplicity of the standard model
 254 of floating-point arithmetic, as it often causes bounds on rounding errors to be overestimated.
 255 So, the second step is to prove tighter bounds (Section 3.2). In the context of `simplify_wb`,
 256 both issues were insignificant, since the goal was to automatically prove that some expression
 257 $\llbracket e \rrbracket_{\text{rnd}}$ is bounded by $\Omega \simeq 2^{1024}$, which is usually several hundreds of orders of magnitude
 258 larger than e .

259 3.1 Rigorous polynomial approximations

260 The correlation issue of naive interval arithmetic is well-known, and it is independent of
 261 rounding errors. In fact, even the interval evaluation of $(a + x) \cdot (b - x)$ would suffer from
 262 it, as $a + x$ and $b - x$ vary in opposite directions with respect to x . A first solution to this
 263 issue is to split the domain of x into smaller sub-intervals and to take the union of the
 264 enclosures of the whole expression on all these sub-intervals. This approach is very simple
 265 proof-wise, but it scales poorly computation-wise, so it should only be used as a last resort.
 266 A second solution is to compute enclosures whose bounds symbolically depend on x rather
 267 than being just numerical values. This approach scales better, but it requires a much larger
 268 formalization effort.

269 CoqInterval provides both approaches [9]. In particular, the second approach is implemented
 270 using rigorous polynomial approximations. Instead of just computing a single interval $[\underline{e}; \bar{e}]$
 271 that encloses an expression $e(x)$ for any $x \in X$, it computes a polynomial P and an interval Δ
 272 such that, for any $x \in X$, we have $e(x) - P(x) \in \Delta$, which we denote by $e \in (P, \Delta)_X$. Those
 273 polynomial enclosures can then be composed. For example, if we have $f \in (P_f, \Delta_f)_X$ and
 274 $g \in (P_g, \Delta_g)_X$, we also have $f + g \in (P_f + P_g, \Delta_f + \Delta_g)_X$. This makes it possible to compute
 275 the polynomial enclosure of an arbitrary expression, by induction on its structure.

276 Therefore, to benefit from the rigorous polynomial approximations of CoqInterval, we
 277 need to be able to compute a polynomial enclosure of $\circ(u)$, given an enclosure $u \in (P, \Delta)_X$.
 278 To do so, we rewrite $\circ(u(x))$ into the sum $[\circ(u(x)) - u(x)] + u(x)$. For the left-hand side,
 279 we use the degree-0 enclosure $\circ(u) - u \in (0, [-\varepsilon; \varepsilon])_X$ with ε computed as in Equation (2).
 280 Then, by adding the original enclosure $(P, \Delta)_X$, we get a polynomial enclosure of $\circ(u)$.

281 This change to CoqInterval was straightforward, but it has shifted the perspective on
 282 rounding operators in the library. Indeed, the original implementation, which was designed
 283 for `simplify_wb`, computed an enclosure of $\circ(u)$ given an enclosure of u . Then, the user
 284 could ask for an enclosure of $\circ(u) - u$, which would be correct but overestimated. The
 285 new implementation computes a tight enclosure of $\circ(u) - u$ from an enclosure of u , from
 286 which it derives an enclosure of $\circ(u)$. This change has been propagated up to the surface
 287 language, that is, CoqInterval now recognizes the expression $\circ(u) - u$ as an atomic error for
 288 an expression u rather than a subtraction between two sub-expressions involving u .

289 3.2 Tighter error bounds

290 By adding support for rounding operators, CoqInterval is now able to automatically prove
 291 Equation (3), but only if the rightmost bound is changed to $1.5 \cdot 2^{-58}$. It fails for any tighter
 292 bound, especially for 2^{-58} , which we need to prove the correctness of the implementation of
 293 Figure 1. As mentioned earlier, the issue comes from the simplicity of the standard model of
 294 floating-point arithmetic, which states that the absolute error between $\circ(u)$ and u is bounded
 295 by $2^{-53} \cdot |u|$, assuming that u is in the normal range. While this is sensibly true for values
 296 of u slightly larger than a power of two, this is off by a factor two for values of u that are
 297 slightly smaller than a power of two. A better model of floating-point errors is to bound the
 298 absolute error between $\circ(u)$ and u by $\frac{1}{2}\text{ulp}(u)$. In other words, given an enclosure $u \in [\underline{u}; \bar{u}]$,
 299 we have the following enclosure of the absolute error:

$$300 \quad \circ(u) - u \in \left[-\frac{\varepsilon}{2}; \frac{\varepsilon}{2}\right] \quad \text{with } \varepsilon = \text{ulp}(\max(-\underline{u}, \bar{u})).$$

301 This has required us to implement in CoqInterval an overestimation of ulp and to prove
 302 that it was sound with respect to the non-computational definition found in the Flocq
 303 library. Not only is this new enclosure tighter, but it separates the concerns about the
 304 target format and the rounding direction. Regarding the target format, one just has to chose
 305 the corresponding definition for ulp. As a consequence, CoqInterval now supports not only
 306 the floating-point formats of Flocq, but also its fixed-point formats. As for the rounding
 307 direction, it is a matter of choosing the enclosing interval: $[-\frac{\varepsilon}{2}; \frac{\varepsilon}{2}]$ for rounding to nearest,
 308 $[0; \varepsilon]$ for rounding toward $+\infty$, and so on.

309 Thanks to these improvements, the `interval` strategy can now directly prove Equation (3).
 310 This proof only takes a tenth of a second using degree-10 polynomials (default degree for
 311 `interval`). Note that the use of polynomial approximations, rather than the use of more
 312 naive variants of interval arithmetic, is critical for this proof, as can be experienced by
 313 reducing the degree. With degree 3, it takes about one second; with degree 2, it takes about
 314 one minute; and with degree 1, it does not seem to terminate.

315 4 New features

316 Since the goal of our work is to formally prove the function shown in Figure 1, we need several
 317 new features that were missing from the earlier work on the Cody-Waite algorithm [6]. First
 318 of all, since our implementation relies on hardware support for both integer and floating-point
 319 numbers, we need an interpretation of the expressions from Section 2 into the corresponding
 320 types (Section 4.1). Since the implementation also uses an array of pre-calculated values to
 321 reduce the degree of the polynomial approximation, expressions have been extended with
 322 support for array accesses (Section 4.2). Finally, as we mentioned earlier, the algorithm takes
 323 advantage of the inaccuracies and “flaws” of floating-point arithmetic to implement optimized
 324 versions of `nearbyint` and `int_of_float`. Hence, to ease the proof of this algorithm, we
 325 have added support for these optimized operations (Section 4.3).

326 4.1 Hardware operations

327 Similarly to $\llbracket e \rrbracket_{\text{fit}}$, we would like to define another interpretation $\llbracket e \rrbracket_{\text{prim}}$ which represents
 328 the computation of e using the hardware types provided by Coq’s standard library. The
 329 `PrimFloat` module offers support for hardware floating-point numbers [10], while the
 330 `PrimInt63` module offers support for OCaml’s 63-bit integers [5]. Both modules provide
 331 constants, basic operations ($+$, $-$, \times , $/$, etc.), comparisons ($=$, $<$, \leq), conversions, and some

332 miscellaneous functions (*e.g.*, floating-point predecessor and successor functions). They also
 333 provide axiomatized specifications for these hardware operations.

334 Since hardware floating-point numbers are just an instance of Flocq’s generic floating-point
 335 numbers, we have derived the following variant of Theorem 1:

336 ► **Theorem 2.** *Given an expression e , $\text{WB}(e) \Rightarrow \llbracket e \rrbracket_{\text{prim}} \text{ finite} \wedge \llbracket e \rrbracket_{\text{prim}} = \llbracket e \rrbracket_{\text{rnd}}$.*

337 There are two things to note about the definition of $\llbracket e \rrbracket_{\text{prim}}$. First, not all operations can
 338 be performed directly on hardware types. Fused multiply-add (FMA), for example, is not
 339 yet provided by the `PrimFloat` module. Therefore, to complete the definition of $\llbracket e \rrbracket_{\text{prim}}$, we
 340 emulate these missing operations using the Flocq library (*i.e.*, convert the operands to the
 341 formalized Flocq types, compute the result using Flocq’s operations, and convert it back to
 342 the hardware type).

343 Second, we have made our integers 32-bit wide, so that we can use Coq’s 63-bit hardware
 344 integers to compute $\llbracket e \rrbracket_{\text{prim}}$ while maintaining our ability to export verified algorithms as C
 345 programs. For 32-bit integer expressions, $\text{WB}(e)$ hence requires $\llbracket e \rrbracket_{\text{rnd}}$ to remain inside the
 346 $[-2^{31}; 2^{31} - 1]$ range.

347 4.2 Array accesses

348 The implementation of Figure 1 starts with an argument reduction that is very similar to
 349 Cody & Waite’s, but based on a slightly different identity:

$$350 \quad \exp(x) = \exp\left(x - k \cdot \frac{\ln 2}{64}\right) \cdot 2^{k/64} \quad \text{with} \quad k = \left\lfloor x \cdot \frac{64}{\ln 2} \right\rfloor. \quad (4)$$

351 This makes the reduced argument much smaller, but it also means that the reconstruction
 352 is not a simple multiplication by an integer power of 2 anymore. To multiply by $2^{k/64}$ for
 353 some integer k , we first compute the Euclidean division of k by 64, in other words find k_q
 354 and k_r such that $k = k_q \cdot 64 + k_r$ and $0 \leq k_r \leq 63$. Since there are only finitely many different
 355 values of k_r , we pre-compute a correct rounding of $2^{k_r/64}$ for each value of k_r and store the
 356 results in a table `cst`. Therefore, to multiply by $2^{k/64}$, we first multiply by `cst`. $[k_r]$ and
 357 then by 2^{k_q} (see Figure 1).

358 We have defined `cst` using the Coq standard library `PArray`, which provides persistent
 359 arrays [5]. To ease proofs, we have added a constructor `ArrayAcc` to the type of expressions
 360 to represent accesses to tables of constants. The constructor takes as argument an array a of
 361 hardware floating-point numbers and an integer expression i and is interpreted as follows:

$$362 \quad \llbracket \text{ArrayAcc } a \ i \rrbracket_{\text{prim}} := a. \llbracket i \rrbracket_{\text{prim}}.$$

363 For an access to be well-behaved, we need the index to be well-behaved and smaller than
 364 the length of the array, and all the entries of the array to be finite floating-point numbers.

365 4.3 Macro-operations

366 As we can see in Equation (4), the argument reduction also requires the `nearbyint` function.
 367 This poses a problem as the latter is not provided by the `PrimFloat` module. Since we only
 368 need to compute the exponential on inputs in the $[-746; 710]$ range, we can use the following
 369 trick³ to compute the integer part:

$$370 \quad \lceil f \rceil = \circ(\circ(f + 1.5 \cdot 2^{52}) - 1.5 \cdot 2^{52}).$$

³ If $|f| \leq 2^{51}$ then $f + 1.5 \cdot 2^{52}$ is between 2^{52} and 2^{53} with $\text{ulp}(2^{52}) = 1$, which means the result of the addition is rounded to the nearest integer.

XX:10 End-to-End Formal Verification of a Fast and Accurate Floating-Point Approximation

371 Using the language of abstract expressions, we could simply represent this sequence of
372 operations as `Op SUB (Op ADD t (BinF1 0x1.8p52)) (BinF1 0x1.8p52)`. However, that
373 would not be very helpful in proofs because it leaves us the tedious work of showing that those
374 operations behave as `nearbyint`. Instead, we want to treat those operations as if they were
375 one single `nearbyint` operation. For this, we define a new constructor `FastNearbyint` in
376 the language whose interpretation as a floating-point expression is the sequence of operations
377 above, but whose interpretation as a rounded expression is the integer part:

$$\begin{aligned} \llbracket \text{FastNearbyint } e \rrbracket_{\text{flt/prim}} &:= \llbracket e \rrbracket_{\text{flt/prim}} \oplus 0\text{x}1.8\text{p}52 \ominus 0\text{x}1.8\text{p}52, \\ \llbracket \text{FastNearbyint } e \rrbracket_{\text{rnd}} &:= \lceil \llbracket e \rrbracket_{\text{rnd}} \rceil. \end{aligned}$$

379 Since these interpretations are no longer in one-to-one correspondence, proving Theorems 1
380 and 2 for these constructors required significantly more work on our part. This, however,
381 saves the user from having to do the work themselves. Note that the macro-operation only
382 works for inputs $|t| \leq 2^{51}$, so $\text{WB}(\text{FastNearbyint } e)$ must contain a conjunct $|\llbracket e \rrbracket_{\text{rnd}}| \leq 2^{51}$
383 for the theorems to hold.

384 The macro-operation we have just defined computes the integer part as a floating-point
385 number, but the algorithm in Figure 1 also needs it as an integer. Hence, we define another
386 constructor `FastNearbyintToInt` which extracts the mantissa⁴ after adding $1.5 \cdot 2^{52}$:

$$\lceil f \rceil_{\mathbb{Z}} = \text{mantissa}(\circ(f + 0\text{x}1.8\text{p}52)) - 3 \cdot 2^{51}.$$

5 Application

389 We now have all the ingredients to state and prove the correctness of the algorithm shown in
390 Figure 1. To simplify notation in this section, whenever a floating-point value `f` is finite (*i.e.*,
391 neither $\pm\infty$ nor NaN), we denote by f the real value it represents.

392 We state the correctness of the algorithm as follows, with `x` the input, and with `flb` and
393 `fub` respectively the lower and upper bounds of the output:

394 ► **Theorem 3.** *If `x` is finite, then $\text{flb} \leq \exp x \leq \text{fub}$.*

395 Proof of this theorem for large positive and negative values of `x` is straightforward,
396 as those are the cases where the exponential either overflows or degenerates to 0. This
397 section presents the methodology we have followed for proving the correctness theorem for
398 $x \in [-745.13; 709.78]$.

399 For a given input `x`, to find an enclosure of $\exp x$, our algorithm performs only one
400 approximation `y`, but then subtracts (resp. adds) an error term `d` to find the lower (resp.
401 upper) bound of the enclosure. Correctness of the algorithm therefore relies on whether `d` is
402 big enough to cancel out the inaccuracy of the approximation. For this reason, an essential
403 step in our proof is to find some ε such that any choice of d with $|d| > \varepsilon$ makes $\circ(\circ(p_0 \cdot y) + d)$
404 an upper bound of $\exp(x - k \ln 2/64) - 2^{k_r/64}$ (and similarly for the lower bound). The value
405 we have experimentally found for ε is characterized by the following lemma:

406 ► **Lemma 4.** *Let `d` a finite floating-point number such that $|d| \leq 2^{-52}$. Then*

$$\left| 2^{k_r/64} + \circ(\circ(p_0 \cdot y) + d) - (\exp(x - k_q \ln 2) + d) \right| < \varepsilon \simeq 3 \cdot 2^{-58}.$$

⁴ For hardware numbers, we have implemented it as `normfr_mantissa (fst (frshiftexp f))`.

408 Since the proof of this lemma is rather intricate, we will just illustrate our methodology
 409 on the piece of code that performs the argument reduction (Section 5.1). We will also show
 410 part of the proof for the reconstruction as it involves some unusual facts about floating-point
 411 arithmetic (Section 5.2).

412 5.1 Illustration of the methodology

413 Among other facts about the argument reduction, we need to prove that the computation of
 414 `ki` causes no exceptional behaviors and that it is indeed an integer part equal to k despite
 415 its convoluted code. To do so in the Coq proof, we have defined an abstract expression `ki'`
 416 whose interpretation in the hardware numbers—namely, $\llbracket \text{ki}' \rrbracket_{\text{prim}}$ —is the value stored in `ki`,
 417 and whose interpretation in the rounded real numbers—namely, $\llbracket \text{ki}' \rrbracket_{\text{rnd}}$ —is k . Then we can
 418 use Theorem 2 to transform a goal about `ki` into a goal about k .

419 In practice, we not only want to transform the goal, but we also want to assert some
 420 property on the transformed subexpression. Hence, we have implemented a strategy
 421 `assert_float` which takes as argument a predicate Q , looks for an expression of the form
 422 $\llbracket e \rrbracket_{\text{prim}}$, and applies the following corollary of Theorem 2:

$$423 \quad \text{WB}(e) \implies Q(\llbracket e \rrbracket_{\text{rnd}}) \implies (\forall x, x = \llbracket e \rrbracket_{\text{rnd}} \wedge Q(x) \implies G(x)) \implies G(\llbracket e \rrbracket_{\text{prim}}).$$

424 The strategy also invokes `simplify_wb` to discharge as many conjuncts of `WB(e)` as possible.
 425 When using this strategy, the Coq proof usually looks as follows:

```
426 set (ki' := FastNearbyintToInt (Op MUL (Var 0) InvLog2_64)).
427 change (normfr_mantissa _ - _)
428 with (evalPrim ki' [:x:]). (* Var 0 is mapped to x *)
429 assert_float (fun ki => -68736 <= ki <= 65536).
430 { ... proof of the assertion ... }
431
432
```

433 We have used the `assert_float` strategy 8 times in the proof. Here is another example
 434 of its usage to state the main property about the reduced argument `t`, which contains exact
 435 operations just like in the original Cody-Waite algorithm (see Section 2.1):

```
436 set (t' := Op SUB (OpExact SUB (Var 1) ...) ...).
437 change (x - _ - _) with (evalPrim t' [:k, x:])).
438 assert_float (fun t => abs t <= 355 / 65536
439               /\ abs (t - (x - k * ln 2) <= 65537 * pow2 (-77)).
440
441
```

442 Contrary to the other uses of `assert_float` in the proof, `simplify_wb` is not able to
 443 completely discharge the subgoal `WB(t)`. So we have to manually prove the remaining
 444 conjuncts, which are the proof obligations of exact operations:

$$445 \quad \circ(k \cdot c_1) = k \cdot c_1 \quad \wedge \quad \circ(x - k \cdot c_1) = x - k \cdot c_1.$$

446 The proof of both equalities relies on bit-counting reasoning, which Gappa is specifically
 447 designed for [3, §4.3.4]. But since we do not want to introduce a dependency over Gappa in
 448 CoqInterval, we have performed this reasoning by hand.

449 5.2 Correctness of reconstruction

450 At this point in the proof, thanks to Lemma 4, we know $2^{k_r/64} + y_\ell \leq \exp x \cdot 2^{-k_q} \leq 2^{k_r/64} + y_u$,
 451 with y_ℓ and y_u some intermediate floating-point results. To complete the proof, we need to

452 deduce the following enclosure:

$$453 \quad flb = \text{pred}(\circ(\circ(p_0 + y_\ell) \cdot 2^{k_q})) \leq \exp x \leq \text{succ}(\circ(\circ(p_0 + y_u) \cdot 2^{k_q})) = fub.$$

454 To do so, we want to factor out the multiplication by 2^{k_q} , but the possibility that the
 455 result might fall into the subnormal range makes this factorization impossible. So we have
 456 proved the following lemma:

457 ► **Lemma 5.** *Let y be a binary64 floating-point number greater than 2^{-1021} . Then, for any*
 458 *integer k , $\text{pred}(\circ(y \cdot 2^k)) \leq \text{pred}(y) \cdot 2^k$ and $\text{succ}(\circ(y \cdot 2^k)) \geq \text{succ}(y) \cdot 2^k$.*

459 By transitivity, we are thus left to prove the following inequalities:

$$460 \quad \text{pred}(\circ(p_0 + y_\ell)) \leq 2^{k_r/64} + y_\ell \quad \wedge \quad 2^{k_r/64} + y_u \leq \text{succ}(\circ(p_0 + y_u)).$$

461 These inequalities hold because the predecessor and successor functions are enough to
 462 compensate both the error between p_0 and $2^{k_r/64}$ and the rounding error of the final addition.
 463 Indeed, there are two cases, depending on the value of k_r :

464 ■ If $k_r = 0$, then $p_0 = 1$. So, the first inequality reduces to $\text{pred}(\circ(1 + y_\ell)) \leq 1 + y_\ell$, which
 465 is a general property of the predecessor function. Proof of the upper bound is similar.

466 ■ If $k_r \neq 0$, then we have $1.01 < p_0 < 1.99$. Therefore, $1.001 < \circ(p_0 + y_\ell) < 1.999$ and
 467 hence $\text{pred}(\circ(p_0 + y_\ell)) = \circ(p_0 + y_\ell) - 2^{-52}$. Since p_0 is a correct rounding of $2^{k_r/64}$, we
 468 have $|p_0 - 2^{k_r/64}| \leq 2^{-53}$. Similarly, $|\circ(p_0 + y_\ell) - (p_0 + y_\ell)| \leq 2^{-53}$. As a consequence,

$$469 \quad \begin{aligned} \text{pred}(\circ(p_0 + y_\ell)) &= 2^{k_r/64} + y_\ell + (p_0 - 2^{k_r/64}) + (\circ(p_0 + y_\ell) - (p_0 + y_\ell)) - 2^{-52} \\ &\leq 2^{k_r/64} + y_\ell + 2^{-53} + 2^{-53} - 2^{-52} = 2^{k_r/64} + y_\ell \end{aligned}$$

470 which completes the proof for the lower bound. Proof of the upper bound is similar.

471 **6 Related works**

472 While there had been some earlier works to formalize hardware arithmetic operators [15],
 473 formal verification of mathematical libraries really started with the impressive work by John
 474 Harrison. Among other things, he used the HOL Light system to prove the correctness of
 475 the IA-64 implementation of sin and cos for floating-point inputs smaller than 2^{63} . This
 476 implementation relies on 80-bit floating-point arithmetic and is accurate to 0.574 ulp [7]. It
 477 uses an intricate argument reduction: first a pre-reduction, followed by a 3-term Cody-Waite
 478 reduction, resulting in a double-*binary80* reduced argument. Then a degree-17 polynomial is
 479 evaluated, followed by a simple reconstruction of the result so as to take the lower part of the
 480 reduced argument into account. During both the argument reduction and the reconstruction,
 481 several floating-operations are actually exact and need to be considered as such, in order to
 482 be able to prove anything interesting about the result. Our methodology could be used to
 483 automate various parts of this proof, but the representation of the reduced argument as a
 484 non-evaluated sum of two floating-point numbers would presumably warrant adding a few
 485 more macro-operations to our expression language, *e.g.*, a FastTwoSum operator [13, §1.3].

486 A more recent work is the large verification using the Coq proof assistant of the power
 487 function of the CORE-MATH library by Laurence Rideau and Laurent Théry [8]. This
 488 includes the correctness of an exponential function whose implementation shares some
 489 similarities with ours, but it is a lot more subtle, since both input and output are double-
 490 *binary64* numbers. Their formalization, however, ignores the issue of exceptional behaviors

491 and just assumes that numbers can be arbitrarily large, as is traditionally the case in
492 pen-and-paper proofs. Again, our methodology could help transition to a complete proof,
493 especially since they are already making heavy use of CoqInterval.

494 Regarding the use of hardware floating-point numbers in the Coq proof assistant, Érik
495 Martin-Dorel and Pierre Roux have implemented and verified a checker for semi-definite
496 positive matrices [10]. The algorithm performs a Choleski decomposition using floating-point
497 arithmetic on a slightly perturbed input matrix. The correctness theorem states, that if
498 this decomposition succeeds, then a Choleski decomposition using exact arithmetic would
499 have succeeded on the original input matrix, which guarantees that it was indeed semi-
500 definite positive. The perturbation, and hence the correctness proof, depends on the ability
501 to compute a bound on the rounding error of the floating-point decomposition [14]. Our
502 approach would have been of little help for that use case, as the algorithm and the error
503 bound highly depend on the dimension of the matrix.

504 Regarding the automation of proofs of bounds on rounding errors in a proof assistant, one
505 can cite the FPTaylor tool, which can generate proofs for HOL Light [16]. Given a floating-
506 point expression, it computes an affine form that encloses it, using elementary rounding errors
507 as variables of the affine form. The strength of that tool is that the coefficients of the affine
508 form are kept as symbolic expressions rather than intervals. This approach separates the
509 concerns between the global optimizer used for computing enclosures and the formalization
510 of affine forms for floating-point arithmetic. Indeed, enclosures of the symbolic expressions
511 are only needed when they occur in terms of order 2 or more, as these terms cannot be
512 represented as part of the affine form. Therefore, the verification of these enclosures can be
513 done in a rather naive way, since they are only used for higher-order error term and thus do
514 not have to be tight. It should be noted that FPTaylor supports both the standard model
515 of floating-point arithmetic and a tighter model (see Section 3.2), but it can only generate
516 proofs for the former. Moreover, the global optimizer used to compute the enclosures in
517 FPTaylor is not the same as the one used to verify them in HOL Light, which might cause
518 difficulties if the latter procedure is not strong enough or too slow.

519 A similar tool is PRECiSA, which targets the PVS proof assistant [11]. As with FPTaylor,
520 errors are kept as symbolic expressions, and a global optimizer is used to compute their
521 enclosures. Higher-order error terms, however, are not eliminated as computations progress,
522 which might cause some performance issues compared to FPTaylor. PRECiSA, however,
523 uses a tight formal model of floating-point errors, and the tool can detect exact subtractions
524 (Sterbenz' lemma). Moreover, it supports conditional expressions, including the cases where
525 rounding causes a different branch to be taken.

526 Finally, one should mention the VCFloat2 tool, which targets the Coq proof assistant [1].
527 As with the previous two tools, the error is kept as a symbolic expression. Before being fed
528 to a global optimizer (namely CoqInterval), this expression is first simplified by expanding it
529 through distributivity and discarding the sub-expressions that cancel. This expansion might
530 cause some performance issues, due to combinatorial explosion. A user-provided threshold is
531 used to further discard negligible terms, at the expense of a potentially worse error bound. It
532 can also use a technique similar to Gappa to reduce the correlation between sub-expressions,
533 and thus improve the tightness of the computed enclosures. The tool uses the standard
534 model of floating-point errors, but the user can annotate operations that are supposed to
535 be exact and the tool will verify that the conditions hold (Sterbenz' lemma). Moreover,
536 the tool supports user-defined operations, which means that it can easily be extended with
537 double-word arithmetic, as long as the user has formalized it beforehand.

538 **7 Conclusion**

539 In this article, we have presented a floating-point approximation of the exponential function,
 540 its mechanized proof of correctness, and the tools we have developed to ease the verification
 541 work. One peculiarity of this work is that the verified code is not just modeled using the
 542 Coq proof assistant, it can actually run in the logic of the system and therefore be used to
 543 perform proofs by computations. Indeed, the correctness theorem tells how the result of
 544 the approximation can be used as a lower/upper bound of the mathematical exponential.
 545 The specification of the code of Figure 1 and its correctness proof take about 600 lines of
 546 Coq script;⁵ Lemma 5 is about 130 lines; extending the proof of Theorem 1 to support
 547 macro-operations and arrays takes about 500 lines; the tighter bounds on rounding errors
 548 take about 200 lines. This work was integrated in release 4.10.0 of CoqInterval.

549 **7.1 Integration to CoqInterval and performances**

550 As explained in the introduction, the CoqInterval library provides an interval extension
 551 of exponential that can use the floating-point unit of the processor to speed up proof
 552 checking [10]. Its implementation, however, is based on a truncated power series, which
 553 is effective but rather naive, compared with the implementations that can be found in
 554 mathematical libraries targeting hardware floating-point formats. We have thus plugged our
 555 verified implementation in place of the original one. Consider the following Coq script.

```
556 Goal forall x, 10 <= x <= 11 -> Rabs (exp x - exp x) <= 0x1p-6.  
557 Proof. intros x Hx. interval with (i_bisect x, i_depth 30). Qed.  
558
```

560 It states that, for any real number x between 10 and 11, the difference between $\exp x$
 561 (mathematical exponential) and itself is less than 2^{-6} . From a mathematical point of view,
 562 this statement is useless, since it could be trivially proved by rewriting $\exp x - \exp x$ to zero,
 563 but it is a good way to exercise the computations performed by CoqInterval. Indeed, the way
 564 the `interval` strategy is invoked, it will not try to use anything fancier than naive interval
 565 arithmetic. As a consequence, because $\exp x$ is strongly correlated with itself, the formal
 566 proof generated by the tactic ends up considering around 6.7 million sub-intervals of the
 567 input enclosure $x \in [10; 11]$ (and as many interval evaluations of exponential).

568 Using the original implementation of exponential, this computationally intensive proof
 569 takes about 160 seconds to be checked by the Coq proof assistant on an Intel 13th-generation
 570 4GHz processor. With the implementation verified in this work, the proof is checked in less
 571 than 8 seconds. Taking the average of three runs, the speedup is $20.5\times$. Since the argument
 572 reduction of the original implementation is more costly the further away from zero the input
 573 is, the speedup can grow even larger, up to $24\times$.

574 **7.2 Real-life performances**

575 Being able to perform about one million faithful interval evaluations of exponential per second
 576 inside the logic of Coq is impressive, but it is nowhere near the actual throughput of the
 577 floating-point unit of the processor. Indeed, disregarding any concern about the guaranteed
 578 accuracy of a mathematical library, one should expect a state-of-the-art implementation

⁵ https://gitlab.inria.fr/coqinterval/interval/-/blob/master/src/Interval/Float_full_primfloat.v

```

let fexp x =
  if x < -0x1.74385446d71c4p9 then 0. else
  if x > 0x1.62e42fefa39efp9 then infinity else
  if x <> x then nan else
  ...
  ldexp (p0 +. p0 *. y) (ki asr 6)

```

■ **Figure 2** Floating-point exponential in OCaml.

579 to take 25–50 cycles to compute two floating-point approximations of exponential⁶ (and
 580 thus one interval enclosure), so about 100× faster than what we currently achieve in the
 581 logic of Coq. There are several reasons for the remaining gap. First of all, the code of our
 582 implementation is not directly run by the processor, but interpreted by a virtual machine.
 583 Second, this bytecode interpreter boxes floating-point numbers, and thus performs a large
 584 amount of memory allocations. Third, while our code only performs computations on values,
 585 the interpreter still needs to account for the possibility of open terms (*e.g.*, free variables)
 586 appearing as operands to the floating-point computations.

587 The first issue can be worked around by using the `native_compute` machinery of the Coq
 588 system, which compiles the code using the OCaml compiler and then executes it directly [2].
 589 This machinery also partially avoids the second issue, since the compiler can optimize away
 590 the boxing of some intermediate floating-point results. But the third issue is still present
 591 and makes it hard to avoid pessimization in the generated code. As a consequence, this only
 592 improves proof checking by a factor 3× to 4× for the longer proofs.

593 To get a better feel of the actual performances of our implementation, we can instead
 594 implement the function directly in OCaml, as shown in Figure 2. This is roughly the same
 595 code as Figure 1, except that the original last three lines, which were computing an enclosure
 596 of $\exp x$, have been replaced by a single floating-point value: `ldexp (p0 +. p0 *. y) (ki`
 597 `asr 6)`. Accordingly, the first few lines return a single value for the exceptional cases. The
 598 code is run on about $1.5 \cdot 10^9$ inputs uniformly distributed among those that lead to a finite
 599 output. Compiling the code with OCaml 5.1.1, we get that the floating-point exponential
 600 from the GNU C Library is about 1.45× faster than our implementation.

601 Even if the GNU C Library has been heavily tuned, this is still a rather large gap. Part
 602 of the reason is its use of the FMA operation. This ternary operation computes $\circ(x \cdot y + z)$
 603 at once, which halves the number of operations performed during the argument reduction
 604 and the polynomial evaluation. Modifying our code accordingly, this reduces its slowdown
 605 to 1.32×. When translating the code to C and compiling it with GCC, the slowdown is
 606 brought down to 1.24×. Obviously, using FMAs in place of multiplications and additions
 607 invalidates the correctness proof, since they do not compute the same values (notice the lack
 608 of rounding operator around the product). Fortunately, the proof can be easily adapted.
 609 Indeed, exact operations during the argument reduction are still exact when performed with
 610 an FMA, and having a more accurate polynomial evaluation only makes the proof simpler.
 611 Note that, while our framework supports reasoning about the FMA operation, it is not one
 612 of the native floating-point operations provided by the Coq system, so it cannot be used to
 613 speed up the implementation of CoqInterval. One would instead have to use larger tables, as
 614 does the GNU C Library, so as to reduce the degree of the polynomial approximation.

⁶ <https://core-math.gitlabpages.inria.fr/>

615 **7.3 Future works**

616 First, it should be noted that, while the GNU C Library does not implement correct rounding
 617 either, it is nonetheless slightly more accurate than our implementation. In about 20% of
 618 cases, the code of Figure 2 returns a floating-point result that is off by one, while for the
 619 GNU C Library, the probability is 10^{-5} . In the context of CoqInterval, this hardly matters,
 620 since we want to compute an enclosure of the mathematical result rather than the nearest
 621 floating-point number. But for a mathematical library, people might prefer a code that is
 622 experimentally a bit more accurate to a code whose correctness has been formally verified.
 623 Most of the inaccuracy comes from the factor `p0`. There are two ways to improve it, both of
 624 which require adding a new table along `cst`. In the first approach, the new table contains
 625 the error on `p0`, which can then be reintroduced in the computation. In the second approach,
 626 the new table tells how to shift the input, such that the error on `p0` becomes negligible.

627 A natural extension of this work is to convert all the other mathematical functions
 628 of CoqInterval to use some state-of-the-art implementation when hardware floating-point
 629 numbers are used as interval bounds. For functions such as `log` and `arctan`, our approach
 630 should work without difficulty, as they are quite similar to `exp`. For trigonometric functions
 631 such as `sin` and `cos`, the situation is slightly different. First of all, they are not monotone, so
 632 considering the lower and upper bounds of the input interval separately might be counter-
 633 productive; it might be better to perform a simultaneous argument reduction on both bounds.
 634 Second, the Cody-Waite approach to argument reduction does not scale well to extremely
 635 large inputs, while some other algorithms for argument reduction take advantage of the
 636 periodicity of the trigonometric functions [12, §11.4].

637 — **References** —

- 638 **1** Andrew Appel and Ariel Kellison. VCFLOAT2: Floating-point error analysis in Coq. In *13th*
 639 *ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 14–29,
 640 London, United Kingdom, 2024. doi:10.1145/3636501.3636953.
- 641 **2** Mathieu Boespflug, Maxime Dénès, and Benjamin Grégoire. Full reduction at full throttle.
 642 In *1st International Conference on Certified Programs and Proofs*, pages 362–377, Kenting,
 643 Taiwan, December 2011. doi:10.1007/978-3-642-25379-9_26.
- 644 **3** Sylvie Boldo and Guillaume Melquiond. *Computer Arithmetic and Formal Proofs*. ISTE Press
 645 – Elsevier, 2017.
- 646 **4** William J. Cody, Jr. and William Waite. *Software Manual for the Elementary Functions*.
 647 Prentice-Hall, Englewood Cliffs, NJ, 1980.
- 648 **5** Maxime Dénès. Towards primitive data types for Coq 63-bits integers and persistent arrays.
 649 In *5th Coq Workshop*, Rennes, France, July 2013. URL: [https://coq.inria.fr/files/coq5_](https://coq.inria.fr/files/coq5_submission_2.pdf)
 650 [submission_2.pdf](https://coq.inria.fr/files/coq5_submission_2.pdf).
- 651 **6** Paul Geneau de Lamarlière, Guillaume Melquiond, and Florian Faissole. Slimmer formal
 652 proofs for mathematical libraries. In Theo Drane and Anastasia Volkova, editors, *30th IEEE*
 653 *International Symposium on Computer Arithmetic*, Portland, OR, USA, September 2023.
- 654 **7** John Harrison. Formal verification of floating point trigonometric functions. In Warren A.
 655 Hunt and Steven D. Johnson, editors, *3rd International Conference on Formal Methods in*
 656 *Computer-Aided Design*, volume 1954 of *Lecture Notes in Computer Science*, pages 217–233,
 657 2000.
- 658 **8** Tom Hubrecht, Claude-Pierre Jeannerod, Paul Zimmermann, Laurence Rideau, and Laurent
 659 Théry. Towards a correctly-rounded and fast power function in binary64 arithmetic. 2024.
 660 URL: <https://inria.hal.science/hal-04159652>.

- 661 9 Érik Martin-Dorel and Guillaume Melquiond. Proving tight bounds on univariate expressions
662 with elementary functions in Coq. *Journal of Automated Reasoning*, 57(3):187–217, 2016.
663 doi:10.1007/s10817-015-9350-4.
- 664 10 Érik Martin-Dorel, Guillaume Melquiond, and Pierre Roux. Enabling floating-point arithmetic
665 in the Coq proof assistant. *Journal of Automated Reasoning*, 67, 2023. doi:10.1007/
666 s10817-023-09679-x.
- 667 11 Mariano Moscato, Laura Titolo, Aaron Dutle, and César A. Muñoz. Automatic estimation of
668 verified floating-point round-off errors via static analysis. In Stefano Tonetta, Erwin Schoitsch,
669 and Friedemann Bitsch, editors, *36th International Conference on Computer Safety, Reliability,
670 and Security*, volume 10488 of *Lecture Notes in Computer Science*, pages 213–229, Trento,
671 Italy, 2017. doi:10.1007/978-3-319-66266-4_14.
- 672 12 Jean-Michel Muller. *Elementary Functions, Algorithms and Implementation*. Birkhäuser,
673 Boston, MA, 3rd edition, 2016. doi:10.1007/978-1-4899-7983-4.
- 674 13 Jean-Michel Muller, Nicolas Brunie, Florent de Dinechin, Claude-Pierre Jeannerod, Mioara
675 Joldes, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, and Serge Torres. *Handbook
676 of Floating-Point Arithmetic*. Birkhäuser Basel, 2nd edition, 2018. doi:10.1007/
677 978-3-319-76526-6.
- 678 14 Pierre Roux. Formal proofs of rounding error bounds – with application to an automatic
679 positive definiteness check. *Journal of Automated Reasoning*, 57(2):135–156, 2016. doi:
680 10.1007/s10817-015-9339-z.
- 681 15 David M. Russinoff. *Formal Verification of Floating-Point Hardware Design*. Springer Cham,
682 2nd edition, 2022. doi:10.1007/978-3-030-87181-9.
- 683 16 Alexey Solovyev, Marek S. Baranowski, Ian Briggs, Charles Jacobsen, Zvonimir Rakamarić,
684 and Ganesh Gopalakrishnan. Rigorous estimation of floating-point round-off errors with
685 symbolic Taylor expansions. *ACM Transactions on Programming Languages and Systems*,
686 41(1), December 2018. doi:10.1145/3230733.