Primitive Floats in Coq

Guillaume Bertholon
École Normale Supérieure, Paris, [École Normale Supérieure, 45 rue d'Ulm, 75230 Paris Cedex 05], France
guillaume.bertholon@ens.fr

Érik Martin-Dorel
Lab. IRIT, University of Toulouse, CNRS, [IRIT Université Paul Sabatier, 118 route de Narbonne, 31062 Toulouse Cedex 9], France
`https://www.irit.fr/~Erik.Martin-Dorel/`
erik.martin-dorel@irit.fr

Pierre Roux
ONERA, Toulouse, [ONERA, Centre de Toulouse, 2 Avenue Édouard Belin, 31055 Toulouse Cedex 4], France
`https://www.onera.fr/staff/pierre-roux`
pierre.roux@onera.fr

Abstract

Some mathematical proofs involve intensive computations, for instance: the four-color theorem, Hales' theorem on sphere packing (formerly known as the Kepler conjecture) or interval arithmetic. For numerical computations, floating-point arithmetic enjoys widespread usage thanks to its efficiency, despite the introduction of rounding errors.

Formal guarantees can be obtained on floating-point algorithms based on the IEEE 754 standard, which precisely specifies floating-point arithmetic and its rounding modes, and a proof assistant such as Coq, that enjoys efficient computation capabilities. Coq offers machine integers, however floating-point arithmetic still needed to be emulated using these integers.

A modified version of Coq is presented that enables using the machine floating-point operators. The main obstacles to such an implementation and its soundness are discussed. Benchmarks show potential performance gains of two orders of magnitude.

2012 ACM Subject Classification Theory of computation → Type theory; Mathematics of computing → Numerical analysis; General and reference → Performance

Keywords and phrases Coq formal proofs, floating-point arithmetic, reflexive tactics, Cholesky decomposition

Digital Object Identifier 10.4230/LIPIcs.ITP.2019.33

Supplement Material `https://github.com/coq/coq/pull/9867`

Acknowledgements The authors would like to thank Maxime Dénès and Guillaume Melquiond for helpful discussions.

1 Motivation

The proof of some mathematical facts can involve a numerical computation in such a way that trusting the proof requires trusting the numerical computation itself. Thus, being able to efficiently perform this kind of proofs inside a proof assistant eventually means that the tool must offer efficient numerical computation capabilities.

Floating-point arithmetic is widely used in particular for its efficiency thanks to its hardware implementation. Although it does not generally give exact results, introducing rounding errors, rigorous proofs can still be obtained by bounding the accumulated errors.

10th International Conference on Interactive Theorem Proving (ITP 2019).
Editors: John Harrison, John O'Leary, and Andrew Tolmach; Article No. 33; pp. 33:1–33:20
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

```
$R := 0$;
for $j$ from $1$ to $n$ do
  for $i$ from $1$ to $j-1$ do
    $R_{i,j} := \left(A_{i,j} - \Sigma_{k=1}^{i-1} R_{k,i} R_{k,j}\right) / R_{i,i}$;
  end for
  $R_{j,j} := \sqrt{M_{j,j} - \Sigma_{k=1}^{j-1} {R_{k,j}}^2}$;
end for
```

Figure 1 Cholesky decomposition: given $A \in \mathbb{R}^{n\times n}$, attempts to compute R such that $A = R^T R$.

There is thus a clear interest in providing an efficient and sound access to the processor floating-point operators inside a proof assistant such as Coq.

1.1 Proofs Involving Numerical Computations

We give below a few examples of proofs involving floating-point computations.

As a first example, consider the proof that a given real number $a \in \mathbb{R}$ is nonnegative. One can exhibit another real number r such that $a = r^2$ and apply a lemma stating that all squares of real numbers are nonnegative. Typically, one could use the square root $\sqrt{a}$.

A similar method can be applied to prove that a matrix $A \in \mathbb{R}^{n\times n}$ is positive semidefinite[1] as one can exhibit R such that[2] $A = R^T R$. Such a matrix can be computed using an algorithm called Cholesky decomposition, given in Figure 1. The algorithm succeeds, taking neither square roots of negative numbers nor divisions by zero, whenever A is positive definite[3].

When executed with floating-point arithmetic, the exact equality $A = R^T R$ is lost but it remains possible to bound the accumulated rounding errors in the Cholesky decomposition such that the following theorem holds under mild conditions.

▶ **Theorem 1** (Corollary 2.4 in [34]). *For $A \in R^{n\times n}$, defining $c := \frac{(n+1)\epsilon}{1-2(n+1)\epsilon}\,\mathrm{tr}(A) + 4n\,(2(n+1) + \max_i A_{i,i})\,\eta$, if the floating-point Cholesky decomposition succeeds on $A - c\,I$, then A is positive definite. ϵ and η are tiny constants given by the floating-point format used.*

A formal proof in Coq of this theorem can be found in a previous work [33]. Thus, an efficient implementation of floating-point arithmetic inside the proof assistant leads to efficient proofs of matrix positive definiteness. This can have multiple applications, such as proving that polynomials are nonnegative by expressing them as sums of squares [26] which can be used in a proof of the Kepler conjecture [24].

Interval arithmetic constitutes another example of proofs involving numerical computations. Sound enclosing intervals can be easily computed in floating-point arithmetic using directed roundings, towards ±∞ for lower or upper bounds. The Coq.Interval library [25] implements interval arithmetic and could benefit from efficient floating-point arithmetic.

More generally, there are many results on rigorous numerical methods [35] that could see efficient formal implementations provided efficient floating-point arithmetic is available inside proof assistants.

---

[1] A matrix $A \in \mathbb{R}^{n\times n}$ is said positive semidefinite when for all $x \in \mathbb{R}^n$, $x^T A\,x \geq 0$.

[2] Since, when $A = R^T R$, one gets $x^T A\,x = x^T \left(R^T R\right) x = (Rx)^T\,(Rx) = \|Rx\|^2 \geq 0$.

[3] A matrix $A \in \mathbb{R}^{n\times n}$ is said positive definite when for all $x \in \mathbb{R}^n \setminus \{0\}$, $x^T A\,x > 0$.

1.2 Objectives

The Coq proof assistant has built-in support for computation, which can be used within proofs, and recent progress have been done to provide efficient integer computation (relying on 63-bit machine integers).

The overall goal of this work is to implement efficient floating-point computation in Coq, relying directly on machine `binary64` floats, instead of emulating floats with pairs of integers. Experimentally, that latter emulation in Coq incurs a slowdown of about three orders of magnitude with respect to an equivalent implementation written in OCaml.

1.3 Outline

The article is organized as follows: Section 2 provides the background required to position our approach, from proof-by-reflection to the IEEE 754 standard for floating-point arithmetic to interval arithmetic formalized in Coq. Section 3 is devoted to the implementation itself, with a special focus on the interface that its exposes. Section 4 gathers a discussion on several design choices or technicalities that have been important to carry out the implementation and avoid some pitfalls. Section 5 provides benchmarks to evaluate the performance of the implementation. Section 6 finally gives concluding remarks and perspectives for future work.

2 Prerequisites and Related Works

In this section, we start by reviewing the two main features that underlie and motivate our work in the Coq proof assistant: Poincaré's principle and the availability of efficient reduction tactics (in Section 2.1). We then give an overview of all notions of floating-point arithmetic that appear necessary to make this paper self-contained (in Section 2.2). We finally summarize the features of two related Coq libraries that are either a prerequisite for our developments (in Section 2.3), or an important building block for a possible extension of this work (in Section 2.4).

2.1 Proof by Reflection and Efficient Numerical Computation

In the family of formal proof assistants, the underlying logic of several systems—including Agda, Coq, Lego, and Nuprl [2]—provides a notion of definitional equality that allows one to automatically prove some equalities by a mere computation. This feature is called *Poincaré's principle* in reference to Poincaré's statement that "a reasoning proving that $2 + 2 = 4$ is not a proof in the strict sense, it is a verification" [32, chap. I]. Based upon this principle, the so-called *proof by reflection* methodology has been developed to take advantage of the computational capabilities of the provers and build efficient (semi)-decision procedures [7]: this approach has been successfully applied to various application domains, such as: graph theory, with the formal verification of the four-color theorem in Coq by Gonthier and Werner [14], discrete geometry, with the formal proof of the Kepler conjecture developed in the Flyspeck project [17], Boolean satisfiability, with the verification of SAT traces in Coq [1], satisfiability modulo theories, with the development of the SMTCoq library [13], or global optimization, with the development of the ValidSDP library [26].

To be able to address the verification of increasingly complex proofs relying on this approach, works have been carried out to increase the computational performance of proof assistants, relying on two complementary approaches: (i) implement alternative evaluation engines, such as evaluators based on compilation to bytecode or native code, and (ii) optimized data structures that might be based on machine values and hardware operators.

For example, the Isabelle proof assistant provides (i) several evaluators that can be used within proofs, and allows one to generate Standard ML, OCaml, Haskell, or Scala code, then (ii) libraries of fast machine words (for fixed size or unspecified size) have been developed while ensuring compatibility with all Isabelle's target languages and evaluators [23].

In this work, we specifically focus on the Coq proof assistant which offers in particular (i) the reduction tactics `vm_compute`, involving bytecode compilation and evaluation by a virtual machine [15] and `native_compute`, involving code generation and native OCaml compilation [3], as well as (ii) *machine integers*, upon which the `Bignums` library for multiple-precision arithmetic has been developed [16].

Regarding machine integers in Coq, the original implementation by Spiwack [1, 39] was based on the so-called *retro-knowledge* approach, which consisted in developing a reference implementation of 31-bit integer operators in Coq (using lists of bits), then optimizing their evaluation in `vm_compute` (and later `native_compute`) by replacing the considered Coq operator on-the-fly with the corresponding hardware operator. The implicit assumption here is that both implementations match. This implementation has been recently replaced with so-called *primitive integers*[4] [12]: this approach required adding a representation of 63-bit machine integers in the kernel, and has the two-fold benefit of offering efficient operators for all reduction strategies with a compact representation of integers, and making explicit the axioms that specify the primitive operators.

The overall aim of this work is to provide a similar facility for floating-point arithmetic, to be able to compute with *primitive floating-point numbers* in Coq, instead of emulating floating-point numbers with pairs of integers.

A facility to compute with floating-point numbers for prototyping purposes is available in the PVS proof assistant thanks to the PVSio package [31] but to the best of our knowledge, no proof assistant currently provides support for machine floating-point computations in the scope of proof by reflection.

2.2 Floating-point Arithmetic

This section reviews the main concepts of floating-point arithmetic used in the remainder of this paper. The reader interested in more details could find them in reference books [30].

Computing in floating-point arithmetic amounts to performing calculations in what is often called scientific notation with one digit before the dot, a fixed number of digits following it and a power of ten specifying the position of the dot, hence the name *floating-point* arithmetic. When results do not fit in the required precision, they have to be rounded, e.g., with a precision of five digits, $1.234 \cdot 10^2 + 5.678 \cdot 10^{-1} = 1.240 \cdot 10^2$.

2.2.1 IEEE 754 Standard

Implementations of floating-point arithmetic in hardware nowadays adhere to the IEEE 754 standard [19]. This standard prescribes sets of floating-point numbers, mostly as subsets of the real numbers field $\mathbb{R}$, binary representations for them, rounding modes and basic arithmetic operators $+$, $-$, $\times$, $\div$ and $\sqrt{\cdot}$ defined as functions giving the same result as the operator in the real field composed with a rounding.

A floating-point format $\mathbb{F}$ is a subset of $\mathbb{R}$ such that $x \in \mathbb{F}$ when

$$x = m\beta^e \tag{1}$$

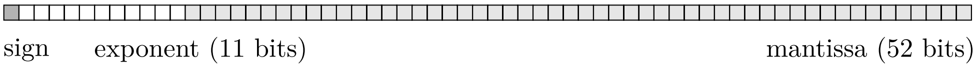
---

[4] See the pull request `https://github.com/coq/coq/pull/6914`.

for some $m, e \in \mathbb{Z}$, $|m| < \beta^p$ and $e_{\min} \le e \le e_{\max} - p$. The integer m is called the *mantissa* of x and e its *exponent*[5]. The constants β and p are called respectively the *radix* and *precision* of the format $\mathbb{F}$ while the constants $e_{\min}$ and $e_{\max}$ define the exponent range of $\mathbb{F}$. Some floating-point values can have multiple representations, e.g., $1230 \cdot 10^2 = 123 \cdot 10^3$. To get a canonical representation, $|m| \ge \beta^{p-1}$ is enforced as soon as $|x| \ge \beta^{p-1+e_{\min}}$. In other words, all the space allowed by the precision is used for the mantissa. Mantissas smaller than β^{p-1} are only used for tiny values x such that $\beta^{e_{\min}} \le |x| < \beta^{p-1+e_{\min}}$, called *denormalized numbers*. Finally, 0 can get a canonical representation by arbitrary choosing an exponent.

2.2.1.1 Binary64 Format

The IEEE 754 standard defines multiple formats in radix $\beta = 2$ and $\beta = 10$ and various precisions. In the remaining of this paper, `binary64` will be the only format considered[6]. This is a binary format, i.e. $\beta = 2$, offering a precision of $p = 53$ bits and its minimal and maximal exponents are respectively $e_{\min} = -1074$ and $e_{\max} = 1024$. As its name suggests, this format enjoys a binary representation on 64 bits as follows:



The exponent is encoded on 11 bits while the mantissa is encoded as its sign and its absolute value on 52 bits[7]. One can notice that, out the of the 2048 values enabled by the 11 bits of exponent, two are unused when encoding exponents in the range $[e_{\min}, e_{\max} - p] = [-1074, 971]$. One is used for denormalized numbers, and 0 when the mantissa is 0, the other for special values NaN, and infinities when the mantissa is 0.

The two infinities $-\infty$ and $+\infty$ are used to represent values that are too large to fit in the range of representable numbers. Similarly, it is worth noting that due to the sign bit, there are actually two representations of 0, namely -0 and $+0$. The standard states that these two values should behave as if they were equal for comparison operators $=, <$ and $\le$. However, they can be distinguished since $1 \div (+0)$ returns $+\infty$ whereas $1 \div (-0)$ returns $-\infty$. Finally, NaN stands for "Not a Number" and is used when a computation does not have any mathematical meaning, e.g., $0 \div 0$ or $\sqrt{-2}$. NaNs propagate, i.e., any operator on a NaN returns a NaN. Moreover, comparison with a NaN always returns false, in particular both $x < y$ and $x \ge y$ are false when x is a NaN, as well as[8] $x = x$. Thanks to the mantissa and sign bits, there are actually $2^{53} - 2$ different NaN values. These payloads can be used to keep track of which error created the special value but they are only partially specified by the standard and are in practice hardware dependent.

2.2.1.2 Precise Specification of Rounding Modes

From a formal point of view, a key definition introduced by the IEEE 754 standard is the notion of rounding. For a given floating-point format $\mathbb{F}$, a rounding is an increasing function $\bigcirc : \mathbb{R} \to \mathbb{F} \cup \{\pm\infty\}$ whose restriction to $\mathbb{F}$ is identity, that is:

$$\begin{cases} \forall x, y \in \mathbb{R}, & x \le y \implies \bigcirc(x) \le \bigcirc(y) \\ \forall x \in \mathbb{R}, & x \in \mathbb{F} \implies \bigcirc(x) = x. \end{cases}$$

---

[5] More precisely called *quantum exponent* [30, p. 14].

[6] It is the usual implementation of the type `double` in the C language.

[7] It actually fits in 53 bits but, except for denormalized numbers, the most significant one is always 1 and doesn't need to be explicitly encoded.

[8] This is a simple way to test for NaN as otherwise $x = x$ is always true.

The IEEE 754-2008 standard [19] defines five standard rounding modes:
toward $-\infty$: $\mathrm{RD}(x)$ is the largest floating-point number $\leq x$;
toward $+\infty$: $\mathrm{RU}(x)$ is the smallest floating-point number $\geq x$;
toward zero: $\mathrm{RZ}(x)$ is equal to $\mathrm{RD}(x)$ if $x \geq 0$, and to $\mathrm{RU}(x)$ if $x \leq 0$;
to nearest even: $\mathrm{RNE}(x)$ is the floating-point number closest to x.
In case of a tie: the one with an even mantissa;
to nearest away from zero: $\mathrm{RNA}(x)$ is the floating-point number closest to x.
In case of a tie: the one with the largest mantissa in absolute value.

In this work, we will only rely on the RNE rounding, which is the default rounding mode in most floating-point programming environments. See Section 4.1 for a more in depth discussion of this point.

Then, all floating-point operators are required to be correctly rounded, that is to say, they should behave as if they were computed with an infinitely precise mantissa, then rounded according to the specified rounding mode. To be more precise, for a given floating-point format $\mathbb{F}$, operator $* : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$, and rounding mode $\bigcirc : \mathbb{R} \to \mathbb{F}$, a correctly-rounded implementation $\circledast$ of $*$ should verify:

$$\forall x, y \in \mathbb{F}, \quad x \circledast y = \bigcirc(x * y).$$

The benefits of this definition are two-fold:

- all floating-point operators that are correctly-rounded (the 2008 revision of the standard requiring this for $+$, $-$, $\times$, $\div$, $\sqrt{\cdot}$) are fully-specified, which straightforwardly ensures the reproducibility of the results;
- it allows one to devise floating-point algorithms that directly rely upon this specification, as exemplified in the upcoming Section 2.2.2.

2.2.2 Error Free Transformations

Noticing that the rounding error of a floating-point addition is itself a floating-point number, algorithms such as Fast2Sum [11] and 2Sum [21, 28] can compute that exact error, taking advantage of correct rounding.

These two "compensated summation algorithms" fall into the larger class of error-free transformations [22, 37] which constitute an essential building block in the development of extended precision floating-point algorithms.

2.2.3 Standard Model

Although precise specifications are known for roundings, hence for basic arithmetic operators, a simpler model is commonly used to prove compound bounds of rounding errors on larger expressions [18]. Despite being weaker, this model is more amenable to algebraic proofs, whether pen and paper or mechanized. Called standard model of floating-point arithmetic, it states the following main properties in the absence of overflow[9]

$$\forall x, y \in \mathbb{F}, \quad \exists \delta, \quad |\delta| \leq \epsilon \ \wedge\ \bigcirc(x + y) = (1 + \delta)(x + y) \tag{2}$$

$$\forall x, y \in \mathbb{F}, \quad \exists \delta, \varphi, \quad |\delta| \leq \epsilon \ \wedge\ |\varphi| \leq \eta \ \wedge\ \bigcirc(x \times y) = (1 + \delta)(x \times y) + \varphi \tag{3}$$

where ϵ and η are tiny constants depending on the floating-point format[10]. As a recent example, the following result is proved in a slightly refined standard model [20].

---

[9] Overflow can often be handled separately.
[10] For `binary64` and $\bigcirc$ a rounding to nearest, $\epsilon = 2^{-53}$ and $\eta = 2^{-1075}$.

▶ **Theorem 2** (Theorem 4.1 in [20]). *For $x \in \mathbb{F}^n$, denoting $\hat{s}$ the sum $\sum_{i=1}^n x_i$ computed with floating-point arithmetic in any order*[11]*, assuming no overflow occurs, it satisfies*

$$\left|\hat{s} - \sum_{i=1}^{n} x_i\right| \leq \frac{(n-1)\epsilon}{1+\epsilon} \left(\sum_{i=1}^{n} |x_i|\right).$$

Coq proofs of such results can be performed, and are at the core of the proof of Theorem 1 [33].

2.3 The Flocq Library

Flocq [5, 6] is a Coq library offering a very generic formalization of floating-point arithmetic. Radix and precision can be fully parameterized and floating-point values are defined, similarly to (1), as a subset of the real numbers $\mathbb{R}$ provided in the Coq standard library [27, Chapter 1].

More specifically, multiple models are available:

- With an unbounded exponent range, i.e., without underflow nor overflow. Although unrealistic, this model is attractive for its simplicity and commonly used for error bounds [18].
- With an exponent range only lower bounded, i.e., with underflow but without overflow. This may still seem unrealistic but overflows can often be studied separately which usually proves much harder for underflows [33].
- A binary model of the `binary32` and `binary64` formats defined in the IEEE 754 standard, with underflows, overflows to infinities, signed zeros and NaNs with payloads. This model is used in the verified C compiler CompCert [4].

Along with these models and links between them, the library contains many classical results about roundings, about some error-free transformations as presented in Section 2.2.2, and basic properties of the standard model described in Section 2.2.3.

The library is mainly developed by Sylvie Boldo and Guillaume Melquiond and is available at URL `http://flocq.gforge.inria.fr/`.

2.4 The Coq.Interval Library

Another Coq library could benefit from efficient floating-point arithmetic: Coq.Interval [25], which offers a modular formalization of interval arithmetic. First, module types (a.k.a. signatures) are defined for floating-point and interval operators. Then, several implementations of the floating-point signature are provided, relying on the Flocq library and specifically its model with unbounded exponent range. A generic implementation is provided, as well as a specialized implementation assuming radix 2 and representing mantissa and exponent as pairs of integers from `Bignums`. Next, a parameterized module implements interval operators where intervals are pairs of floating-point numbers, and related computations are performed using directed roundings, towards $-\infty$ or $+\infty$. Elementary functions such as `exp`, `ln` or `atan` are provided among these interval operators, but correct rounding is not guaranteed (namely, the computed intervals can be overestimated, albeit the containment property always holds and has been formally proved). Finally, tactics `interval` (decision procedure) and `interval_intro` (for forward reasoning) are provided to automatically and formally prove inequalities on real-valued expressions.

The library is mainly developed by Guillaume Melquiond and is available at URL `http://coq-interval.gforge.inria.fr`.

---

[11] Floating-point addition is not associative.

3 Contributions

In order to provide access to efficient floating-point arithmetic inside proofs, the following steps have been performed:

1. Define a minimal working interface for the IEEE 754 `binary64` format. See Section 3.1.
2. Devise a specification of this interface that enables using `binary64` computations in proofs. This specification should be compatible with Flocq, so that all previously proved results, both in Flocq and based upon it, can be straightforwardly reused, using a simple compatibility layer. Details are in Section 3.2.
3. Implement the chosen interface in Coq's various computation mechanisms, i.e., `compute`, `vm_compute` and `native_compute` at the OCaml and C levels. A brief summary of the implementation is given in Section 3.3 and salient points are discussed in Section 4.
4. Assess the performance by running some benchmarks. Results are given in Section 5.

3.1 Interface

In our modified version of Coq, after typing

```
Require Import Floats.
```

the user gets access to the following interface[12]:

```
Parameter float : Set.
```

A type for primitive floating-point values. Inside the kernel, this is mapped to the `float` type of OCaml[13] that matches `binary64`.

```
Parameters add sub mul div : float -> float -> float.
Parameters sqrt opp abs : float -> float.
```

The basic arithmetic operators $+$, $-$, $\times$, $\div$, $\sqrt{\cdot}$, opposite and absolute value.

```
Variant float_comparison : Set := FEq | FLt | FGt | FNotComparable.
Parameter compare : float -> float -> float_comparison.
```

A comparison function that behaves as specified by the IEEE 754 standard. In particular $+0$ and -0 are considered equal and NaNs are not comparable to any value, hence the `FNotComparable` answer.

A few functions are then given to examine or craft precise floating-point values by translating them from or to primitive integers.

```
Variant float_class : Set :=
  | PNormal | NNormal | PSubn | NSubn | PZero | NZero | PInf | NInf | NaN.
Parameter classify : float -> float_class.
```

A function testing whether a given value is a NaN, an infinity (`NInf` and `PInf` for $-\infty$ and $+\infty$ respectively), -0 (`NZero`), $+0$ (`PZero`), a denormalized value (`NSubn` and `PSubn`) or a regular one (`NNormal` and `PNormal`).

---

[12] Defined in file `theories/Floats/PrimFloat.v` in the implementation.
[13] The implementation language of Coq.

```
Definition shift := 2101%int63.  (* = 2 × emax + prec *)
Parameter frshiftexp : float -> float * Int63.int.
```

`frshiftexp` f returns a pair (m, e) such that[14] $|m| \in [0.5, 1)$ and $f = m \times 2^{e-\texttt{shift}}$. Primitive integers are unsigned so `shift` is used to ensure that e is nonnegative.

```
Parameter ldshiftexp : float -> Int63.int -> float.
```

`ldshiftexp` f e returns $f \times 2^{e-\texttt{shift}}$. This is the reverse of `frshiftexp` and it is exact except when underflow or overflow occurs, in which case the result is rounded using RNE.

```
Parameter normfr_mantissa : float -> Int63.int.
```

When f, typically obtained from `frshiftexp`, satisfies $|f| \in [0.5, 1)$, `normfr_mantissa` f returns the primitive integer $|f| \times 2^p$, that is the integer encoding the mantissa of f.

```
Parameter of_int63 : Int63.int -> float.
```

Converts a primitive integer to a floating-point value. Since primitive integers are unsigned 63-bit integers, they do not all fit into the 53-bit mantissas of the `binary64` format. Values that do not fit are rounded using RNE.

Finally, two functions compute the successor and predecessor of a floating-point value. They can be used to implement interval arithmetic for instance.

```
Parameters next_up, next_down : float -> float.
```

Equipped with this interface, the Coq user can now perform floating-point computations using the processor operators and any of the evaluation mechanisms provided by Coq.

```
Coq < Require Import Floats. Open Scope float_scope.
Coq < Eval compute in 1 + 0.5.
     = 1.5 : float
Coq < Eval vm_compute in 1 / -0.
     = neg_infinity : float
Coq < Eval native_compute in 0 / 0.
     = nan : float
```

3.2 Specification

Although floating-point computations are possible, they remain entirely useless in proofs at this point, since there is no specification of their behavior. We thus need a Coq specification of floating-point arithmetic.

First of all, the set of floating-point values itself has to be specified[15].

```
Variant spec_float :=
  | S754_zero (sign : bool)  (* true for -0, false for +0 *)
  | S754_infinity (sign : bool)
  | S754_nan
  | S754_finite (sign : bool) (mantissa : positive) (exponent : Z).
```

---

[14] When f is finite and non zero, otherwise $(m, e) = (f, 0)$.

[15] See file `theories/Floats/SpecFloat.v` in the implementation.

This is similar to the `full_float` type in the `IEEE754.Binary` module of the Flocq library except for one point: the sign and payload of NaNs are not modeled here. It is also worth noting that this models much more values than the `binary64` format[16] since no bounds on mantissas nor exponents are enforced. This makes for a simple specification.

Then, each of the above operators must be specified on this `spec_float` type. This specification is mostly borrowed[17] from the `IEEE754.Binary` module of the Flocq library and totals 398 lines in our implementation[18]. We thus only detail the multiplication operator. We first need to define a few characteristics of the `binary64` format as seen in Section 2.2.1.1

```
Definition prec := 53%Z.
Definition emax := 1024%Z.
Definition emin := (3 - emax - prec)%Z. (* = -1074 *)
Definition fexp e := Z.max (e - prec) emin.
```

When $|x| \in \left[2^{e-1}, 2^{e}\right)$, then `fexp` e is the exponent used to encode x in the `binary64` format.

As seen in Section 2.2.1.2, the floating point multiplication is defined by $x \otimes y = \bigcirc(x \times y)$. When $x = m_x 2^{e_x}$ and $y = m_y 2^{e_y}$, then $x \times y = (m_x \times m_y)\, 2^{e_x + e_y}$ and the rounding operator $\bigcirc$ has to remove the extra bits in the mantissa to make this value fit in the format. To this end, we first abstract the bits to remove as two booleans, the *rounding* bit remembers the first forgotten bit whereas the *sticky* bit is `true` when any of the remaining forgotten bits is 1 and `false` when they are all 0. The function `shr_1` then shifts a mantissa one bit to the right, updating the rounding and sticky bits accordingly

```
Record shr_record := { shr_m : Z ; shr_r : bool ; shr_s : bool }.
Definition shr_1 mrs :=
  let s := orb (shr_r mrs) (shr_s mrs) in match shr_m mrs with
  | Z0 (* 0 *) => Build_shr_record Z0 false s
  | Zpos xH (* 1 *) => Build_shr_record Z0 true s
  | Zpos (xO p) (* 2p *) => Build_shr_record (Zpos p) false s
  | Zpos (xI p) (* 2p+1 *) => Build_shr_record (Zpos p) true s
  | ... (* same for Zneg _ *) end.
```

Eventually, `shr` can iterate n shifts and `shr_fexp` removes the required number of bits using the above function `fexp` (`Zdigits2 m` is the number of bits of `m`)

```
Definition shr mrs e n := match n with
  | Zpos p => (iter_pos shr_1 p mrs, (e + n)%Z) | _ => (mrs, e) end.
Definition shr_fexp m e :=
  shr (Build_shr_record m false false) e (fexp (Zdigits2 m + e) - e).
```

It now remains to round the mantissa according to the values of the rounding and sticky bits

```
Definition round_nearest_even mrs := match mrs with
  | Build_shr_record mx false _ => mx
  | Build_shr_record mx true false => if Z.even mx then mx else (mx + 1)%Z
  | Build_shr_record mx true true => (mx + 1)%Z end.
```

---

[16] `spec_float` gathers an infinite number of values, whereas `binary64` only contains finitely many values.

[17] Except for the specifications of `frexp`, `ldexp`, `normfr_mantissa`, `succ` and `pred` which were not yet present in Flocq and which we took the opportunity to add `https://gitlab.inria.fr/flocq/flocq/merge_requests/3`.

[18] See file `theories/Floats/SpecFloat.v` in the implementation.

Finally, the rounding function first shifts the mantissa, rounds it, shifts the result one bit to the right in case the rounding added an extra bit and handles potential overflows

```
Definition binary_round_aux sx mx ex :=
  let '(mrs', e') := shr_fexp mx ex in
  let '(mrs'', e'') := shr_fexp (round_nearest_even mrs') e'
  in match shr_m mrs'' with Z0 => S754_zero sx | Zneg _ => S754_nan
  | Zpos m => if Zle_bool e'' (emax - prec) then S754_finite sx m e''
    else S754_infinity sx end.
```

Thus, it remains to the multiplication to handle all particular cases

```
Definition SFmul x y := match x, y with
  | S754_nan, _ | _, S754_nan => S754_nan
  | S754_infinity sx, S754_infinity sy => S754_infinity (xorb sx sy)
  | S754_infinity sx, S754_finite sy _ _ => S754_infinity (xorb sx sy)
  | S754_finite sx _ _, S754_infinity sy => S754_infinity (xorb sx sy)
  | S754_infinity _, S754_zero _ => S754_nan
  | S754_zero _, S754_infinity _ => S754_nan
  | S754_finite sx _ _, S754_zero sy => S754_zero (xorb sx sy)
  | S754_zero sx, S754_finite sy _ _ => S754_zero (xorb sx sy)
  | S754_zero sx, S754_zero sy => S754_zero (xorb sx sy)
  | S754_finite sx mx ex, S754_finite sy my ey =>
    binary_round_aux (xorb sx sy) (Zpos (mx * my)) (ex + ey) end.
```

In addition to the usual operators, two functions are defined going back and forth from primitive floats to specification floats.

```
Definition Prim2SF : float -> spec_float.
Definition SF2Prim : spec_float -> float.
```

Finally, one needs to establish a link between the primitive operators and the specification. This is done by adding axioms to the system.[19] First, to specify the two functions `Prim2SF` and `SF2Prim` above, one needs to characterize those values of type `spec_float` that actually represent a `binary64` floating-point number, i.e., values with appropriately bounded mantissa and exponent.

```
Definition canonical_mantissa m e := Zeq_bool (fexp (Zdigits2 m + e)) e.
Definition bounded m e :=
  andb (canonical_mantissa m e) (Zle_bool e (emax - prec)).
Definition valid_binary x := match x with
  | SF754_finite _ m e => bounded m e | _ => true end.
```

Again, this code comes from the Flocq library [5]. So equipped, the following three axioms can be stated:

```
Axiom Prim2SF_valid : forall x, valid_binary (Prim2SF x) = true.
Axiom SF2Prim_Prim2SF : forall x, SF2Prim (Prim2SF x) = x.
Axiom Prim2SF_SF2Prim :
  forall x, valid_binary x = true -> Prim2SF (SF2Prim x) = x.
```

---

[19] See file `theories/Floats/FloatAxioms.v` in the implementation.

These properties allow one to prove that both `Prim2SF` and `SF2Prim` are injective and thereby form a bijection between primitive floats and the subset of valid specification floats.

```
Theorem Prim2SF_inj : forall x y, Prim2SF x = Prim2SF y -> x = y.
Theorem SF2Prim_inj : forall x y, SF2Prim x = SF2Prim y ->
  valid_binary x = true -> valid_binary y = true -> x = y.
```

Thus, all of the fifteen operators given in Section 3.1 are linked to their specification by an axiom such as, for the multiplication:

```
Axiom mul_spec :
  forall x y, Prim2SF (x * y)%float = SFmul (Prim2SF x) (Prim2SF y).
```

Since the specification is almost identical to the `IEEE754.Binary` module of Flocq, a link with Flocq is straightforwardly built[20], establishing a bridge towards real numbers and giving access to all the results already proved in the library. This plays a key role in enabling actual proofs using primitive floating-point computations. Moreover, this enables to gain additional confidence in the above non trivial specification, since Flocq contains correctness theorems basically stating for instance[21] that, except when overflow occurs, `SFmul x y` is indeed the rounding of the real number $\mathtt{x} \times \mathtt{y}$.

3.3 Implementation

The implementation was submitted to be integrated in Coq through the GitHub pull request `https://github.com/coq/coq/pull/9867`.

Below is an overview of the size of the development at the time of writing, summarized by sub-components (over the ≈ 3.7 kLoC added).

- OCaml and C: 1815 LoC
  (floats ↪ kernel : 1070) (`vm_compute` support: 255) (`native_compute` support: 355) (parsing and pretty-printing: 85) (Coq checker: 50)
- Coq specifications: 620 LoC [mostly borrowed from Flocq]
- Coq proofs: 340 LoC
- Tests: 800 LoC
- Sphinx documentation: 115 LoC

This implementation required the addition of some code in the kernel of Coq. Most of it only consists in wrapping the floating-point operators into the different evaluation mechanisms of Coq and its core, actually dealing with floating-point arithmetic, can be found in the files `kernel/float64.ml`, `kernel/byterun/coq_interp.c` and `kernel/byterun/coq_float64.h`. Most operators are implemented in C, as required by the `vm_compute` mechanism, and boil down to calls to the appropriate functions of the C standard library. Thus, no involved algorithmic happens in this added code itself.

4 Discussion

4.1 Rounding Modes

We implement only one of the five rounding modes defined in the IEEE 754-2008 standard, namely rounding to nearest even (RNE). We argue here that implementing other rounding

---

[20] See `https://gitlab.inria.fr/flocq/flocq/merge_requests/6`.
[21] See theorem `Bmult_correct` in module `Flocq.IEEE754.Binary`.

modes would not only easily be seriously harmful in terms of performance, notwithstanding the potential threat to soundness of the implementation, but also not very useful.

Unfortunately on most common processors, operators with different rounding modes are not implemented using different opcodes but a status flag. Once the flag is set to a particular rounding mode, all subsequent computations are performed with this rounding mode. Changing the rounding mode is then costly as it requires flushing pipelines.

Interval arithmetic constitutes the main use of rounding modes other than RNE we can foresee in a proof assistant. A common solution to the aforementioned performance issue is to set the rounding mode once to $+\infty$ (RU), used to compute upper bounds, and emulate rounding toward $-\infty$ (RD), used to compute lower bounds, by relying on properties like[22] $\mathrm{RD}(x+y) = -\mathrm{RU}((-x)+(-y))$. Although a monadic interface could be a reasonable implementation, this remains an imperative programming feature and doesn't integrate well within the functional paradigm offered by Coq. Moreover, if no particular care is taken to avoid or disable them, wild compiler optimizations—assuming that only RNE is used—could easily break the previous property, thus ruining the soundness of the whole approach.

However, interval arithmetic doesn't require precise directed roundings but only over- and under-approximations thereof. We thus offer the `next_up` and `next_down` functions, computing the successor and predecessor of a floating-point value. Together with rounding to nearest operators, they satisfy the following property, ensuring soundness of interval arithmetic while providing a reasonably precise approximation of directed roundings:

$$\forall x \in \mathbb{R}, \quad \mathrm{RU}(x) \le \texttt{next\_up}(\mathrm{RNE}(x))$$
$$\forall x \in \mathbb{R}, \quad \texttt{next\_down}(\mathrm{RNE}(x)) \le \mathrm{RD}(x).$$

4.2 Parsing and Pretty-Printing

Parsing and pretty-printing floating-point values is a non trivial question. We expect the following main property: printing a floating-point value and then reparsing the output of the printing function should give the initial value, i.e., $\mathit{parse} \circ \mathit{print}$ should be the identity over `binary64`. It is worth noting that this necessarily implies the injectivity of the printing function. However, we don't require the parsing function to be injective, i.e., we do accept that multiple strings are parsed as the same floating-point value.

A simple solution would be to print an exact hexadecimal representation of the floating-point values, with a binary exponent, e.g., "0xcp-3". This fulfills the above requirement. Unfortunately, this is not very user-friendly. A decimal output would be much more human readable, e.g., "1.5" instead of "0xcp-3".

It is known that printing `binary64` values using at least 17 significant digits and implementing parsing as a rounding to nearest guarantees the above requirements [30, Table 2.3, p. 44]. This is thus the adopted solution. The current version of Coq only offers support for parsing and printing integer constants, so we extended this support[23] to decimal constants using the ubiquitous format ⟨*integer_part*⟩.⟨*fractional_part*⟩e⟨*decimal_exponent*⟩, e.g., "1.23e-4".

4.3 Soundness

During our development, we identified three main potential threats to soundness:

---

[22] The opposite $x \mapsto -x$ being exact in floating-point arithmetic (the sign bit is simply flipped).

[23] See the pull request `https://github.com/coq/coq/pull/8764`.

Specification Issues due to a mismatch w.r.t. the implementation would break the soundness. We hope that taking in extenso our specification from the Flocq library, resulting from a few decades of experience in the field and proving links with other models, mitigates this risk. Moreover, such an error in the specification can only be harmful when the corresponding axiom is used. It is worth noting that all the axioms used in a proved theorem explicitly appear in the result of the Coq command `Print Assumptions`.

Incompatible Implementations in different evaluation mechanisms (`compute`, `vm_compute` or `native_compute`) or even on different machines could lead to a proof of `False` by evaluating a same term to different results. For instance, the payload of NaNs is not fully specified by the IEEE 754 standard and different hardwares can produce different NaNs for a same computation. That's why we chose to consider all NaNs as equal and not distinguish them. Thus incompatible implementations at the bit level remain compatible at the logical level. Double roundings due to the x87 on old 32 bits architectures [29] could also be harmful. The OCaml[24] compiler systematically relies on it, forcing us to implement all floating-point operators in C and to use the appropriate compiler flags. A runtime test[25] is eventually added to prevent Coq from running in case of miscompilation. Another extreme example of implementation discrepancy would be a hardware bug such as the one encountered in the division of the early Pentium processors.

Incorrect Convertibility Test that distinguish two values that shouldn't or vice versa is also a threat. For instance, implementing this test using the equality test on floating-point values (as defined in the IEEE 754 standard) would be wrong as it equates -0 and $+0$ which should be distinguished since $1 \div (-0) = -\infty \neq 1 \div (+0) = +\infty$. Fortunately enough, this keeps a very simple implementation, with the following OCaml code:

```
let equal f1 f2 =
  let is_nan f = f <> f in
  match classify_float f1 with
  | FP_normal | FP_subnormal | FP_infinite -> f1 = f2
  | FP_nan -> is_nan f2 | FP_zero -> f1 = f2 && 1. /. f1 = 1. /. f2
```

A few other, more minor, points appeared during the development. Among them, the fact that primitive integers in Coq are unsigned did require some care[26]. Finally, the way OCaml optimizes arrays[27] of floating-point values[28] did cause a few nasty bugs, although it is unlikely that such bugs could lead to a proof of `False` as they often yield a mere segmentation fault.

5 Benchmarks

The overall objective of this work is to increase the performance of reflexive tactics involving floating-point arithmetic in Coq. Thus we first measure the performance gain on such a tactic, then evaluate it on its individual floating-point operators. We first present the reference problems under study (Section 5.1), then recap the hardware and software setup for these benchmarks (Section 5.2), and finally give the experimental results (Section 5.3).

---

[24] The implementation language of Coq.
[25] See file `kernel/float64.ml` in the implementation.
[26] We indeed fixed a few soundness bugs in primitive integers, pertaining with unsigned integers, before they were merged in Coq master development branch (`https://github.com/coq/coq/pull/6914`).
[27] Arrays are used to communicate environments between the OCaml implementation of the kernel and the C implementation of the `vm_compute` virtual machine.
[28] This causes other issues in OCaml itself and seems to be a hot topic currently in the OCaml community [9].

5.1 Reference Test-suite

We developed a reflexive tactic `posdef_check`, performing some matrix positive definiteness check along the lines of Theorem 1 introduced in Section 1.1. Its implementation was adapted by reusing building blocks from our previous work on the `validsdp` tactic for multivariate polynomial positivity [26].

This tactic is available in four flavors using `vm_compute` or `native_compute` and emulated floats or primitive floats. Emulated floats are a state of the art implementation of floating point arithmetic, based on primitive integers, from the Coq.Interval library whereas primitive floats are our new implementation.

Regarding the test-suite, we generated a set of random positive definite matrices (after fixing a given seed to make the random data reproducible) of size 50×50 up to 400×400.

We perform two kinds of benchmarks on this test-suite: the overall speedup between the versions of `posdef_check` using emulated vs. primitive floats; and the individual speedup in floating-point operators involved in this tactic.

5.2 Hardware/Software Setup

The formalization of the `posdef_check` tactic relies on a large set of dependencies that takes around one hour to compile. For greater convenience, we devised some Docker images containing the benchmark environment, based on Debian Stretch, opam 2 (the OCaml package manager) and OCaml 4.07.0+flambda. The source code of all benchmarks as well as guidelines to install Docker and run the benchmarks are gathered on GitHub at this URL: `https://github.com/validsdp/benchs-primitive-floats/tree/1.0`

The use of Docker (a so-called *OS-level virtualization system*) for these benchmarks yields a number of interesting features, beyond the facility to download and run a pre-built image on different machines: it runs containers in an isolated environment from the host machine, it ensures portability (across OSes such as GNU/Linux, macOS and Windows) and reproducibility, while being more lightweight than traditional virtual machines (VMs).

The experimental results of the upcoming Section 5.3 have been obtained using a Debian GNU/Linux workstation based on a Intel Core i7-7700 CPU clocked at 3.60 GHz, with 16 GB of RAM. All benchmarks have been executed sequentially (namely, without the `-j` option of `make`), with a total elapsed time of about 3h35', using the following image: `"docker pull registry.gitlab.com/erikmd/docker-coq-primitive-floats/master_compiler-edge:9_coq-2ac1f46532264bacf2b1d8f5b6ee3659fe0cde67"`.

5.3 Experimental Results

We first measure the execution time of the whole tactic on the test-suite and compare it between emulated floats and primitive floats. The results are displayed in Table 1 for `vm_compute` and `native_compute`. Each timing is measured 5 times. The tables indicate the corresponding average and relative error among the 5 samples.

One can notice that the obtained speedups are far from the three order of magnitudes separating emulated floats from equivalent OCaml implementations. From the above results, it appears that arithmetic operators constitute most of the computation time with emulated floats (at least 95% with `vm_compute`) but nothing tells us this is still the case with primitive floats. In fact, with primitive floats, most of the computation time is dedicated to list

Table 1 Proof time for the reflexive tactic posdef_check.

Source	`vm_compute` Emulated	`vm_compute` Primitive	Diff.	`native_compute` Emulated	`native_compute` Primitive	Diff.
mat050	0.16s ±2.0%	0.01s ±0.0%	20x	0.05s ±4.0%	0.02s ±5.1%	3x
mat100	1.16s ±1.3%	0.06s ±5.8%	21x	0.28s ±2.5%	0.03s ±2.5%	9x
mat150	3.61s ±1.2%	0.18s ±2.2%	21x	0.75s ±3.0%	0.08s ±3.5%	9x
mat200	8.68s ±0.2%	0.41s ±1.0%	21x	1.71s ±1.0%	0.18s ±3.4%	10x
mat250	17.14s ±1.3%	0.80s ±0.3%	21x	3.34s ±1.4%	0.33s ±2.1%	10x
mat300	30.01s ±1.2%	1.37s ±0.7%	22x	5.77s ±2.4%	0.56s ±1.0%	11x
mat350	48.31s ±1.3%	2.15s ±0.1%	23x	9.09s ±3.0%	0.81s ±1.2%	11x
mat400	70.19s ±1.4%	3.18s ±0.5%	22x	13.56s ±4.0%	1.12s ±0.7%	12x

manipulating functions as our matrices are implemented using lists[29] [8]. Thus, we would like to get an idea of the time actually devoted to floating-point arithmetic in the total proof time of our reflexive tactic. We use the following simple methodology: replace each arithmetic operator with a version, uselessly, performing the computation twice[30], then subtract the execution time of the original program ("Op" in the tables) to the one of this modified program ("Op×2" in the tables). The obtained time ("Op time" in the tables) corresponds to the time devoted to the considered arithmetic operator. Note that the redundant computations involved in the modified program ("Op×2") could not be implemented with a mere additional let-in such as `...let m1 := mul a b in let m2 := mul a b in m2` because the virtual machine and the OCaml native compiler would optimize away the unused local definition; but doing so and adding an extra function call `...in select m1 m2` with `Definition select (a b : F.type) := a.` made it possible to use this doubling trick. The results are given in Table 2 for `vm_compute` and Table 3 for `native_compute`, in each case both on addition and multiplication[31]. Again, each timing is measured 5 times. It is worth noting that those last results should be taken more as coarse orders of magnitude than precise results. In particular, due to the overhead stemming from the duplication itself of the operators[32], the speedups are—maybe seriously—underapproximated. Actual speedups could thus be higher than the ones suggested here.

6 Conclusion and Future Work

We developed a theory of floating-point arithmetic for the Coq proof assistant, composed of primitive implementation of basic arithmetic operators ($+$, $-$, $\times$, $\div$, $\sqrt{\cdot}$), using the processor floating-point operators in rounding-to-nearest even, as well as successor and predecessor operators that can be used to approximate directed roundings to $-\infty$ or $+\infty$. This implementation is axiomatized under the assumption that the processor complies with the IEEE 754 standard for floating-point arithmetic. Particular care has been taken to make the implementation compatible across the different reduction engines of Coq, and across different hardware, thereby avoiding soundness issues that could be caused, for example, by the semantics of NaN payloads that is under-specified in the IEEE 754 standard.

---

[29] This could be improved using primitive "persistent arrays" once they will be integrated in Coq [1].

[30] Or thousand times for primitive floats to avoid getting a result of the same order of magnitude than the variability of computation times.

[31] Additions and multiplications constitute the vast majority of the arithmetic computations performed in a Cholesky decomposition, as seen in Figure 1.

[32] Like expensive function calls.

Table 2 Computation time for individual operators with `vm_compute`.

Op Source	Emulated floats CPU times (Op×2−Op)	Op	Primitive floats CPU times (Op×1001−Op)	Op	Diff.
add					
mat200	10.78±0.9% − 8.38±2.8%	2.40s	15.72±0.5% − 0.45±1.1%	0.02s	157x
mat250	21.46±1.7% − 16.41±1.5%	5.06s	30.62±0.6% − 0.82±0.6%	0.03s	170x
mat300	37.43±1.4% − 28.63±1.4%	8.80s	53.12±2.4% − 1.40±0.5%	0.05s	170x
mat350	59.42±0.8% − 45.95±2.9%	13.48s	84.19±0.8% − 2.19±0.5%	0.08s	164x
mat400	87.78±0.9% − 66.17±1.7%	21.61s	127.56±8.5% − 3.21±0.3%	0.12s	174x
mul					
mat200	12.21±1.4% − 8.38±2.8%	3.83s	16.10±3.0% − 0.45±1.1%	0.02s	245x
mat250	24.52±1.4% − 16.41±1.5%	8.11s	31.12±3.7% − 0.82±0.6%	0.03s	268x
mat300	42.84±1.7% − 28.63±1.4%	14.21s	53.25±0.8% − 1.40±0.5%	0.05s	274x
mat350	68.23±1.5% − 45.95±2.9%	22.28s	84.33±0.7% − 2.19±0.5%	0.08s	271x
mat400	99.72±1.5% − 66.17±1.7%	33.55s	125.74±0.8% − 3.21±0.3%	0.12s	274x

Table 3 Computation time for individual operators with `native_compute`.

Op Source	Emulated floats CPU times (Op×2−Op)	Op	Primitive floats CPU times (Op×1001−Op)	Op	Diff.
add					
mat200	2.24±1.4% − 1.78±1.7%	0.46s	17.68±1.4% − 0.22±0.9%	0.02s	27x
mat250	4.49±4.2% − 3.41±3.1%	1.08s	34.29±0.7% − 0.37±1.5%	0.03s	32x
mat300	7.25±1.2% − 5.83±4.6%	1.42s	59.57±2.5% − 0.55±0.9%	0.06s	24x
mat350	11.66±3.8% − 9.28±3.5%	2.39s	93.82±1.1% − 0.82±0.8%	0.09s	26x
mat400	17.07±2.9% − 13.14±0.9%	3.93s	141.97±2.6% − 1.18±0.9%	0.14s	28x
mul					
mat200	2.48±1.5% − 1.78±1.7%	0.70s	17.81±1.1% − 0.22±0.9%	0.02s	40x
mat250	4.82±2.4% − 3.41±3.1%	1.41s	35.14±2.1% − 0.37±1.5%	0.04s	41x
mat300	8.41±2.4% − 5.83±4.6%	2.59s	60.66±2.2% − 0.55±0.9%	0.06s	43x
mat350	13.21±2.4% − 9.28±3.5%	3.94s	97.25±1.0% − 0.82±0.8%	0.10s	41x
mat400	19.27±1.5% − 13.14±0.9%	6.13s	138.61±2.3% − 1.18±0.9%	0.14s	45x

We evaluated the performance on an implementation—carried out in Gallina, the input language of Coq—of a Cholesky decomposition that underlies a reflexive tactic for matrix positive definiteness, and the experimental results indicate a speedup of two orders of magnitude for arithmetic operators using `vm_compute`. This is consistent with the performance factor of about three orders of magnitude observed between floating-point arithmetic emulated using primitive integers in Coq and equivalent implementations written in OCaml.

Now that primitive floats are available in a proof assistant, multiple future works can be envisioned. The most obvious one would be to adapt the Coq.Interval library to take advantage of primitive floats. Still in this direction, it is known that the successor and predecessor functions, used to approximate directed roundings, can be efficiently implemented using only arithmetic operators [36, 38]. Such an implementation could enable to remove these functions from the trusted code base. It would also be interesting to look at more elaborate elementary functions such as exp or arctan, relying for example on the CR-libm implementation [10]. Finally, in an attempt to improve confidence in the consistency between specification and implementation, and while waiting for a fully formally specified hardware interface, it is worth noting that this consistency is amenable to some intensive automatic testing, although exhaustive testing is out of reach for even unary operators on `binary64`.

References

1 Michaël Armand, Benjamin Grégoire, Arnaud Spiwack, and Laurent Théry. Extending coq with imperative features and its application to SAT verification. In Matt Kaufmann and Lawrence C. Paulson, editors, *Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*, volume 6172 of *Lecture Notes in Computer Science*, pages 83–98. Springer, 2010. `doi:10.1007/978-3-642-14052-5_8`.

2 Henk Barendregt and Herman Geuvers. Proof-assistants using dependent type systems. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning (in 2 volumes)*, pages 1149–1238. Elsevier and MIT Press, 2001.

3 Mathieu Boespflug, Maxime Dénès, and Benjamin Grégoire. Full Reduction at Full Throttle. In Jean-Pierre Jouannaud and Zhong Shao, editors, *Certified Programs and Proofs - First International Conference, CPP 2011, Kenting, Taiwan, December 7-9, 2011. Proceedings*, volume 7086 of *Lecture Notes in Computer Science*, pages 362–377. Springer, 2011. `doi:10.1007/978-3-642-25379-9_26`.

4 Sylvie Boldo, Jacques-Henri Jourdan, Xavier Leroy, and Guillaume Melquiond. A formally-verified C compiler supporting floating-point arithmetic. In Alberto Nannarelli, Peter-Michael Seidel, and Ping Tak Peter Tang, editors, *21st IEEE Symposium on Computer Arithmetic, ARITH 2013, Austin, TX, USA, April 7-10, 2013*, pages 107–115. IEEE Computer Society, 2013. URL: `http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=6545866`, `doi:10.1109/ARITH.2013.30`.

5 Sylvie Boldo and Guillaume Melquiond. Flocq: A unified library for proving floating-point algorithms in coq. In Elisardo Antelo, David Hough, and Paolo Ienne, editors, *20th IEEE Symposium on Computer Arithmetic, ARITH 2011, Tübingen, Germany, 25-27 July 2011*, pages 243–252. IEEE Computer Society, 2011. URL: `http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=5991607`, `doi:10.1109/ARITH.2011.40`.

6 Sylvie Boldo and Guillaume Melquiond. *Computer Arithmetic and Formal Proofs: Verifying Floating-point Algorithms with the Coq System.* Elsevier, 2017.

7 Samuel Boutin. Using reflection to build efficient and certified decision procedures. In Martín Abadi and Takayasu Ito, editors, *Theoretical Aspects of Computer Software, Third International Symposium, TACS '97, Sendai, Japan, September 23-26, 1997, Proceedings*, volume 1281 of *Lecture Notes in Computer Science*, pages 515–529. Springer, 1997. `doi:10.1007/BFb0014565`.

8 Cyril Cohen, Maxime Dénès, and Anders Mörtberg. Refinements for free! In Georges Gonthier and Michael Norrish, editors, *Certified Programs and Proofs*, volume 8307 of *Lecture Notes in Computer Science*, pages 147–162. Springer, 2013. URL: `http://dx.doi.org/10.1007/978-3-319-03545-1_10`.

9 Simon Colin, Rodolphe Lepigre, and Gabriel Scherer. Unboxing mutually recursive type definitions in ocaml. In *Proceedings of JFLA, Les Rousses, France, 30th January to 2nd February 2019.*, 2019.

10 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. Technical report, LIP, ENS Lyon, 2006. URL: `https://hal-ens-lyon.archives-ouvertes.fr/ensl-01529804/`.

11 T. J. Dekker. A floating-point technique for extending the available precision. *Numerische Mathematik*, 18(3):224–242, 1971.

12 Maxime Dénès. Towards primitive data types for COQ 63-bits integers and persistent arrays. In *Coq-5, the Coq Workshop 2013*, Rennes, France, July 2013. Extended abstract. URL: `https://coq.inria.fr/files/coq5_submission_2.pdf`.

13 Burak Ekici, Alain Mebsout, Cesare Tinelli, Chantal Keller, Guy Katz, Andrew Reynolds, and Clark W. Barrett. Smtcoq: A plug-in for integrating SMT solvers into Coq. In Rupak Majumdar and Viktor Kuncak, editors, *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*, volume

10427 of *Lecture Notes in Computer Science*, pages 126–133. Springer, 2017. `doi:10.1007/978-3-319-63390-9_7`.

14 Georges Gonthier. Formal Proof—The Four-Color Theorem. *Notices of the American Mathematical Society*, 55(11):1382–1393, 2008. URL: `http://www.ams.org/notices/200811/tx081101382p.pdf`.

15 Benjamin Grégoire and Xavier Leroy. A compiled implementation of strong reduction. In Mitchell Wand and Simon L. Peyton Jones, editors, *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02), Pittsburgh, Pennsylvania, USA, October 4-6, 2002.*, pages 235–246. ACM, 2002. `doi:10.1145/581478.581501`.

16 Benjamin Grégoire and Laurent Théry. A Purely Functional Library for Modular Arithmetic and Its Application to Certifying Large Prime Numbers. In Ulrich Furbach and Natarajan Shankar, editors, *IJCAR*, volume 4130 of *Lecture Notes in Computer Science*, pages 423–437. Springer, 2006. `doi:10.1007/11814771_36`.

17 Thomas Hales, Mark Adams, Gertrud Bauer, Tat Dat Dang, John Harrison, Le Truong Hoang, Cezary Kaliszyk, Victor Magron, Sean Mclaughlin, Tat Thang Nguyen, and et al. a formal proof of the Kepler conjecture. *Forum of Mathematics, Pi*, 5:e2, 2017. 29 pages. `doi:10.1017/fmp.2017.1`.

18 Nicholas Higham. *Accuracy and Stability of Numerical Algorithms.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1996.

19 IEEE Computer Society. IEEE Standard for Floating-Point Arithmetic. *IEEE Standard 754-2008*, 2008.

20 Claude-Pierre Jeannerod and Siegfried M. Rump. On relative errors of floating-point operations: Optimal bounds and applications. *Math. Comput.*, 87(310):803–819, 2018. `doi:10.1090/mcom/3234`.

21 D. Knuth. *The Art of Computer Programming*, volume 2. Addison Wesley, Reading, MA, 1973.

22 Peter Kornerup, Vincent Lefèvre, Nicolas Louvet, and Jean-Michel Muller. On the computation of correctly rounded sums. *IEEE Trans. Computers*, 61(3):289–298, 2012. `doi:10.1109/TC.2011.27`.

23 Andreas Lochbihler. Fast Machine Words in Isabelle/HOL. In Jeremy Avigad and Assia Mahboubi, editors, *Interactive Theorem Proving - 9th International Conference, ITP 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings*, volume 10895 of *Lecture Notes in Computer Science*, pages 388–410. Springer, 2018. `doi:10.1007/978-3-319-94821-8_23`.

24 Victor Magron, Xavier Allamigeon, Stéphane Gaubert, and Benjamin Werner. Formal proofs for nonlinear optimization. *Journal of Formalized Reasoning*, 8(1):1–24, 2015. URL: `http://jfr.unibo.it/article/view/4319`.

25 Érik Martin-Dorel and Guillaume Melquiond. Proving tight bounds on univariate expressions with elementary functions in Coq. *Journal of Automated Reasoning*, 57(3):187–217, October 2016. `doi:10.1007/s10817-015-9350-4`.

26 Érik Martin-Dorel and Pierre Roux. A reflexive tactic for polynomial positivity using numerical solvers and floating-point computations. In Yves Bertot and Viktor Vafeiadis, editors, *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs, CPP 2017, Paris, France, January 16-17, 2017*, pages 90–99. ACM, 2017. `doi:10.1145/3018610.3018622`.

27 Micaela Mayero. *Formalisation et automatisation de preuves en analyses réelle et numérique.* PhD thesis, Université Paris VI, décembre 2001.

28 O. Møller. Quasi double-precision in floating-point addition. *BIT*, 5:37–50, 1965.

29 David Monniaux. The pitfalls of verifying floating-point computations. *ACM Trans. Program. Lang. Syst.*, 30(3):12:1–12:41, 2008. URL: `https://doi.org/10.1145/1353445.1353446`, `doi:10.1145/1353445.1353446`.

30 Jean-Michel Muller, Nicolas Brisebarre, Florent de Dinechin, Claude-Pierre Jeannerod, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, Damien Stehlé, and Serge Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser, 2010. `doi:10.1007/978-0-8176-4705-6`.

31 César Muñoz. Rapid prototyping in PVS. Technical Report CR-2003-212418, NASA, 2003.

32 Henri Poincaré. *La science et l'hypothèse*. Flammarion, Paris, 1902.

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

34 Siegfried M. Rump. Verification of positive definiteness. *BIT Numerical Mathematics*, 46:433–452, 2006.

35 Siegfried M. Rump. Verification methods: Rigorous results using floating-point arithmetic. *Acta Numerica*, 19:287–449, 2010.

36 Siegfried M Rump, Takeshi Ogita, Yusuke Morikura, and Shin'ichi Oishi. Interval arithmetic with fixed rounding mode. *Nonlinear Theory and Its Applications, IEICE*, 7(3):362–373, 2016.

37 Siegfried M. Rump, Takeshi Ogita, and Shin'ichi Oishi. Accurate floating-point summation part I: faithful rounding. *SIAM J. Scientific Computing*, 31(1):189–224, 2008. `doi:10.1137/050645671`.

38 Siegfried M. Rump, Paul Zimmermann, Sylvie Boldo, and Guillaume Melquiond. Computing predecessor and successor in rounding to nearest. *BIT Numerical Mathematics*, 49(2):419–431, Jun 2009. `doi:10.1007/s10543-009-0218-z`.

39 Arnaud Spiwack. *Verified Computing in Homological Algebra. (Calculs vérifiés en algèbre homologique)*. PhD thesis, École Polytechnique, Palaiseau, France, 2011. URL: `https://tel.archives-ouvertes.fr/pastel-00605836`.