A Unified Coq Framework for Verifying C Programs with Floating-Point Computations

Tahina Ramananandro Paul Mountcastle Benoît Meister Richard Lethin

Reservoir Labs Inc., USA

lastname@reservoir.com

Abstract

We provide concrete evidence that floating-point computations in C programs can be verified in a homogeneous verification setting based on Coq only, by evaluating the practicality of the combination of the formal semantics of CompCert Clight and the Flocq formal specification of IEEE 754 floating-point arithmetic for the verification of properties of floating-point computations in C programs. To this end, we develop a framework to automatically compute real-number expressions of C floating-point computations with rounding error terms along with their correctness proofs. We apply our framework to the complete analysis of an energy-efficient C implementation of a radar image processing algorithm, for which we provide a certified bound on the total noise introduced by floating-point rounding errors and energy-efficient approximations of square root and sine.

Categories and Subject Descriptors G.1.0 [*Mathematics of Computing, Numerical Analysis, General*]: Computer arithmetic, Error analysis, Interval arithmetic; D.3.1 [*Software, Programming Languages, Formal Definitions and Theory*]: Semantics; D.2.4 [*Software, Software Engineering, Software/Program Verification*]: Correctness proofs, Formal methods

Keywords Formal Verification, Coq, Floating-point Computations, C.

1. Introduction

Numerical rounding errors can often have catastrophic effects, and throwing more bits at a problem is no guarantee that these numerical precision problems will be avoided. Conversely, significant performance savings can be achieved by reducing some precision and introducing some approximations, without necessarily introducing dramatic errors in the final results. To what extent can such error bounds be guaranteed on C programs with floating-point computations? How trustworthy can such guarantees be? The goal of our effort described in this paper is to create a context for provable error estimates at lower numerical precision, thereby saving power by possibly avoiding unnecessary computations, and to do so with certifiably lower risk to the mission due to precision failures.

CPP'16, January 18–19, 2016, St. Petersburg, FL, USA

http://dx.doi.org/10.1145/2854065.2854066

Properties (accuracy, stability, complexity, time, space and energy consumption, etc.) of floating-point computations have been an ongoing concern for industrial companies developing control software heavily relying on machine arithmetic. Since the inception of floating-point computations in computers, their study has led to the entire field of numerical analysis, at the intersection of computer science and mathematics. However, the desire to strengthen the trust in computer implementations of numerical programs has grown to the point that pen-and-paper proofs are no longer sufficient and computerized verification strategies become necessary.

In this paper, we combine both Coq specifications of floating-point arithmetic and C semantics in a unified Coq setting for the purpose of source-level verification of C programs performing floating-point computations, to provide stronger numerical guarantees based on the real-number semantics of floating-point numbers. The main goal of our approach is to show that it is possible to prove numerical properties of practical C programs in a verification setting whose trusted computing base only contains the faithfulness of formal mathematical Coq specifications of C and floating-point numbers, the soundness of Coq's underlying logic (the Calculus of Inductive Constructions [7]), and the implementation of the Coq proof checker.

Our approach relies on specifications of floating-point arithmetic and C semantics written in a mathematical language and thus meant to be more widely readable and understandable than the actual code of specific implementations of verification tools, especially if such tools are automated and highly optimized, such as Fluctuat [21, 38] which is written in C++. In particular, it is not easy to trust the fact that the results computed by those tools will reflect on the actual behavior of the C program with floating-point computations being verified. In this paper, we provide a verification approach that bridges such gap.

Contributions The contributions of our work, which we describe in this paper, are as follows:

- In Section 3, we clarify the interpretation of the semantics of C floating-point computations by defining a core floating-point calculus and proving its consistency with C implicit type conversions (casts and type promotions) in Coq against the formal semantics of a subset of CompCert C.
- Based on our core floating-point calculus, we develop VCFloat, a verification framework based on an automatic Coq tactic to reason about the real-number values of C floating-point computations, which we describe in Section 4.
- In Section 5, we demonstrate the practicality of VCFloat by applying it to the first complete analysis of a practical C program with floating-point computations: we introduce an energy-efficient C implementation of a radar image processing algorithm with energy-efficient approximations. We have computed

and proved a bound on the total image noise introduced by our C implementation, and we have mechanized the whole proof using Coq, with a Coq theorem statement about the actual behaviour of our C program.

We have carried out our proofs using the Coq proof assistant [16]. Our proofs are available on the Internet at `http://github.com/reservoirlabs`

2. Related Work

There already exist practical tools to carry on some form of verification of C programs computing floating-point values. Fluctuat [21, 38] is a closed-source commercial automatic static analyzer for the verification of floating-point properties of C programs, based on abstract interpretation [17]. Fluctuat is heavily used in industry [18], and it is implemented in C++. However, trusting the correctness of Fluctuat implies to trust the implementations of the static analysis algorithms and their optimizations, which can be very complex. This is why we rather advocate for the use of general-purpose verification tools in which floating-point arithmetic is formally specified in a readable mathematical specification language.

Mechanized Proofs of Floating-Point Properties with Coq To provide the highest possible level of trust in IEEE 754 floating-point computations independently of the particular implementation of the verification method or tool, it becomes necessary to specify IEEE 754 using a mathematical specification language in a proof assistant. Flocq [9] is such a comprehensive specification of IEEE 754 in the Coq proof assistant [7, 16], on which our work builds.

The Metalibm project [26] aims to build a certified mathematical library implemented in C with floating-point computations. Metalibm builds on Sollya [15], an open-source tool and environment for the development of "safe floating-point code." Sollya is targeted to help develop implementations of mathematical elementary transcendental functions such as trigonometry, exponential, etc. and automatically computes approximation and rounding error bounds. Some results produced by Sollya can be verified using the Gappa tool [29], which supports floating-point and fixed-point interval arithmetic and produces a proof certificate that can be checked [10] with Coq against Flocq. Gappa is also used in the CRlibm project [33], a certified mathematical library which comes with a mostly on-paper correctness proof with some parts checked using Gappa, and on which Metalibm is building. The certification effort of Metalibm and CRlibm only focuses on the correctness of the floating-point computations, rather than the verification of their embedding in C code, which we also address in our work.

Mechanized Proofs of Floating-Point Computations in C Programs Even though floating-point computations can be verified, such verification results must transport to the actual C program in which those computations will be implemented.

There have been very few successful attempts to verify C programs with floating-point computations with respect to a real-number specification. The first fully verified implementation of a numerical (floating-point) C program has been verified by Boldo et al. [11]: a C implementation of a numerical solver for a wave equation. They prove that the function computed by their program is a solution of the wave equation provided by the user within some error bounds. Their verification is based on Frama-C [5], an automatic static analyzer for C programs that generates verification conditions deemed enough to prove the functional correctness of the C code. Such verification conditions are checked using external verification tools such as Coq with Flocq, but also automatic SMT solvers such as Alt-Ergo, CVC3 and Z3. Allowing the user to choose their own combination of tools can make verification very practical (which explains why Frama-C is already used in industry), but it is a major drawback when it comes to assessing the mathematical soundness of such heterogeneous combination of verification tools. Moreover, users must trust the implementation of Frama-C, more precisely the fact that the verification conditions generated by Frama-C are enough to ensure functional correctness. By contrast, our approach advocates the use of a homogeneous combination of verification tools and proof libraries based on Coq only, and trusts as little implementation code as possible, replacing trusted, sometimes heavily optimized implementations of domain-specific verification tools with more trustworthy formal specifications in readable mathematical languages.

Mechanized Proofs on C Programs against a Formal Semantics To avoid trusting a particular implementation of a C program verifier, it is necessary to formalize the semantics of a suitable subset of C and to build verification tools that are certified against this formal semantics.

CompCert [27, 28] is one of the first realistic efforts to specify a subset of C in Coq for the purpose of formal verification. CompCert specifies several subsets of C, the largest being CompCert C, to build a verified realistic compiler down to x86, PowerPC and ARM assembly. Our work relies on the formal semantics of Clight [8], a subset of C specified by CompCert.

The formal semantics of various subsets of C in CompCert have allowed Appel et al. to develop Verifiable C [3], a subset of C equipped with a powerful program logic in Coq. Certified implementations in Verifiable C include the SHA-256 encryption algorithm [2] and OpenSSL HMAC [6]. However, Verifiable C's program logic provides no specific support for reasoning with floating-point numbers, so that examples of proofs of C programs with floating-point computations distributed with Verifiable C state no properties about their real-number semantics. The goal of our work is precisely to provide such specific floating-point support to the formal verification of C programs against a formal specification of Clight.

Another formal semantics of C not based on Coq is Ellison and Rosu's, who formalized a more comprehensive subset of C using the K verification framework [20], from which they derived a program verification tool based on model-checking. However, they have not used it to verify any program with floating-point computations, since their formal semantics does not fully specify IEEE 754 floating-point computations (they are using the *implementation* provided by K itself instead), which "is fine for interpretation and explicit state model checking, but not for deductive reasoning."

Combining Flocq with a Formal Semantics of C Coq actually allows solving the problems of trusting verifiers for floating-point computations in C programs, using a combination of Flocq with a formal semantics of a subset of C such as CompCert.

However, in practice, combining Flocq with CompCert was not first meant with source-level program verification in mind. Indeed, such a unified setting was first meant for compiler verification, namely formal verification of semantics-preserving optimizations of floating-point computations [12]. In their work, Boldo et al. focus on the semantics preservation of floating-point computations down to the bitwise representation of floating-point numbers, thus permitting some of those operations to be implemented using *integer* operators instead. Their representation preservation covers infinities and also NaN ("not a number") cases. In other words, their view of floating-point numbers is merely in terms of low-level raw bits rather than their high-level real-number meaning. So, whereas Boldo et al.'s work shows the practicality of the Flocq-CompCert combination on the compiler verification side, we show it on the source program verification side.

Jourdan et al. developed Verasco [24], a verified static analyzer for C programs. Verasco allows the user to annotate their program with assumptions and assertions in such a way that it can be, subsequently, automatically proven not to go wrong (no divide by zero, no mishandling of infinities or NaNs). This is the premise necessary for verified compilers such as CompCert to ensure that compilation preserves the semantics of the C program. Verasco is wholly proved in Coq and fully supports CompCert C#minor (a subset of C with side effect-free expressions and a weaker type system), including floating-point computations. However, although Verasco's floating-point analysis is also based on combining Flocq with a formal semantics of a subset of C, it does not support error analysis with respect to the corresponding real-number computations. Indeed, since such property is based on true real numbers, it cannot be even stated in Verasco's assertion language. In fact, any such property is not useful for Verasco's particular purpose of showing that a C#minor program cannot go wrong, so Verasco handles floating-point computations with a purely floating-point interval analyzer without the need for a real-number interpretation.

By contrast with these works, our approach allows the formal verification of implementations of C programs against a formal semantics of C to prove functional correctness properties about the *real-number values* of the floating-point computations performed by such C programs, such as approximation or rounding error analysis.

3. Floating-Point Computations in C

Our setting is based on the CompCert Clight language, which allows us to design a faithful view of C floating-point expressions and their semantics, which we describe in this section.

3.1 The Source Language: CompCert Clight

We assume that our program is written in the CompCert Clight [8] subset of C where expressions have no side effects and each function call is isolated as a standalone statement. In this case, CompCert Clight expressions are pure and deterministic.

However, it may not be totally obvious to assess the trustworthiness of the formal semantics of CompCert Clight expressions with respect to the actual semantics specified by ANSI C. In fact, Boldo et al. [12, §3] describe a more comprehensive floating-point semantics for CompCert C, a nondeterministic subset of ANSI C that is much larger than Clight and that is actually the top-most source language of CompCert. Then, the trustworthiness of the CompCert C semantics of floating-point expressions is transported to Clight thanks to the fact that CompCert C is compiled to Clight in a provably semantics-preserving way as part of CompCert's frontend [27].

The basic principle of floating-point computations in C, as correctly specified by CompCert C and Clight, is that every binary operation is performed in the higher of the two precisions of its arguments, regardless of the precision actually expected when reusing the result in another expression. Consider for instance the following C code:

```
float x = ... ; float y = ... ;  double z = x + y;
```

Then, the sum `x+y` is first performed in single-precision (due to `x` and `y` being single-precision floating-point arguments) before being cast to `double` when stored to `z`. To enforce its computation in double-precision, the user would have to explicitly cast either of the two arguments to `double`, and then the other argument would be implicitly cast to `double`.

The formal semantics of Clight defines expression evaluation rules as a big-step semantics. CompCert Clight's expression evaluation rules are described in extenso in Blazy et al.'s paper [8, §2.2, §3.2] as well as in the Coq development of CompCert [27].

f	$\in$	$\mathbb{F}$	Floating-point literal
τ	$\in$	$\mathbb{N}\times\mathbb{N}$	Floating-point type
v	$\in$	V	Variable
t	$::=$		Floating-point computation
		(f,τ)	Typed constant
	$\mid$	(v,τ)	Typed variable
	$\mid$	$t\oplus t \mid t\ominus t \mid t\otimes t \mid t\oslash t$	Rounded binary operation
	$\mid$	$\sqrt{\!\!\bigcirc}\, t$	Rounded unary operation
	$\mid$	$-t \mid \lvert t\rvert$	Exact unary operation
	$\mid$	$[t]_\tau$	Type cast

Figure 1: VCFloat floating-point core calculus: syntax of terms

3.2 A Core Floating-Point Calculus for C

In this section, we describe our view of floating-point computations, and we prove that it is consistent with the semantics of CompCert Clight floating-point expressions.

Following the IEEE 754 Standard [1] as specified in Flocq [9], a *finite floating-point number* of precision $prec \in \mathbb{N}$ and exponent range $(emin, emax) \in \mathbb{Z}^2$ is a number that can be represented in one of the two following ways:

- either $(-1)^s \times 2^{-(prec-1)} \times (2^{prec-1}+m) \times 2^e$ with the sign bit $s \in \{0,1\}$, the significand $m \in \mathbb{N} \cap \left[0, 2^{prec-1}\right)$, and the exponent $e \in \mathbb{Z} \cap [emin, emax)$. In this case, the floating-point number is said to be *normal* (or *normalized*.)
- or $(-1)^s \times 2^{-(prec-1)} \times m \times 2^{emin}$, with the sign bit $s \in \{0,1\}$, the significand $m \in \mathbb{N} \cap [0, 2^{prec-1}]$, and the exponent equal to $emin$. In this case, the floating-point number is said to be *denormal* (or *denormalized*.)

These two cases can be merged into a unique case $(-1)^s \times m' \times 2^e$ with the sign bit $s \in \{0,1\}$, the mantissa $m' \in \mathbb{N} \cap [0, 2^{prec})$, and the exponent $e \in \mathbb{Z} \cap [emin - prec, emax - prec]$, with the boundary between normal and denormal numbers being at $m' = 2^{prec-1}$ and $e = emin$.

Our VCFloat framework supports *typed* floating-point computations. We describe the type of floating-point numbers of precision $prec$ and exponent range $(emin, emax)$ as the pair of integers $(prec, emax)$ with $emin = 3 - emax$. We assume that $2 \leq prec < emax$. For IEEE 754 floating-point numbers, the double-precision type is represented by $(53, 1024)$ whereas the single-precision type is represented by $(24, 128)$. Our implementation is actually generic in the type for future support of IEEE extended double precision (C `long double`) floating-point numbers, which can be easily supported by Flocq[1], but currently not by CompCert.

VCFloat automatically transforms every Clight floating-point expression into a term t of the grammar defined in Figure 1 (where we assume V is an infinite set of variables, and $\mathbb{F}$ represents the type of IEEE 754 floating-point literals as specified in Flocq). $\sqrt{\!\!\bigcirc}\, t$ represents the rounded square root. A floating-point term t is valid if, and only if, for all typed constants (f,τ) appearing in t, f is a floating-point number of type τ. From now on, we only consider valid floating-point terms.

Every floating-point term t has a type $\langle t\rangle$ defined in Figure 2. The type of a typed constant or variable is explicitly given. The type of a (rounded or exact) unary operation is the type of its operand, except for a cast to some type τ, which has type τ. For (rounded or exact) binary operations between two operands of respective types τ_1, τ_2, the type of the binary expression is the *least upper-bound*

[1] Flocq is even more generic, since it is also generic in the radix itself, to accomodate radix-10 floating-point numbers specified in IEEE 754:2008 and used in some financial contexts.

$$
\begin{array}{rlrl}
\langle (f,\tau) \rangle & = \tau & \langle (v,\tau) \rangle & = \tau \\
\langle t_1 \circledcirc t_2 \rangle & = \langle t_1 \rangle \sqcup \langle t_2 \rangle & (\circledcirc \in \{\oplus, \ominus, \otimes, \oslash\}) & \\
\langle \textcircled{$\surd$} t \rangle & = \langle t \rangle & \langle [t]_\tau \rangle & = \tau \\
\langle -t \rangle & = \langle t \rangle & \langle |t| \rangle & = \langle t \rangle
\end{array}
$$

Figure 2: Type of floating-point terms

$$
\begin{array}{rlrl}
\llbracket (f,\tau) \rrbracket_\rho & = f & \llbracket (v,\tau) \rrbracket_\rho & = \rho(v) \\
\llbracket t_1 \circledast t_2 \rrbracket_\rho & \multicolumn{3}{l}{= \circ_{\langle t_1 \rangle \sqcup \langle t_2 \rangle} (\circ_{\langle t_1 \rangle \sqcup \langle t_2 \rangle} \llbracket t_1 \rrbracket_\rho * \circ_{\langle t_1 \rangle \sqcup \langle t_2 \rangle} \llbracket t_2 \rrbracket_\rho)} \\
 & \multicolumn{3}{l}{(* \in \{+, -, \times, /\})} \\
\llbracket \textcircled{$\surd$} t \rrbracket_\rho & = \circ_{\langle t \rangle} \sqrt{\llbracket t \rrbracket_\rho} & \llbracket [t]_\tau \rrbracket_\rho & = \circ_\tau \llbracket t \rrbracket_\rho \\
\llbracket -t \rrbracket_\rho & = -\llbracket t \rrbracket_\rho & \llbracket |t| \rrbracket_\rho & = |\llbracket t \rrbracket_\rho|
\end{array}
$$

Figure 3: Real-number semantics of VCFloat floating-point terms

$\tau_1 \sqcup \tau_2$ defined as:

$$
\begin{aligned}
& (prec_1, emax_1) \sqcup (prec_2, emax_2) \\
= \; & (\max(prec_1, prec_2), \max(emax_1, emax_2))
\end{aligned}
$$

Given an environment $\rho : V \to \mathbb{F}$, the semantics $\llbracket t \rrbracket_\rho$ of a floating-point term t is a floating-point number of type $\langle t \rangle$ (if we assume that, for any typed variable (v, τ) appearing in t, $\rho(v)$ is a floating-point number of type τ), as defined in Figure 3.

If x is a real-number and τ is a floating-point type, then we use CompCert's choice of the rounding operator $\circ_\tau(x)$, namely the floating-point number of type τ nearest to x, with ties broken to choosing the floating-point number with the even significand.

As the general principle of the real-number semantics of IEEE floating-point computations, as specified by Flocq, rounded operations are first computed in real numbers then rounded to the destination type. For binary operators, both operands are first cast to the least upper-bound type prior to computing the operation, following the general principle of C floating-point computations. Such casts are actually innocuous for finite floating-point numbers (their real-number values are not changed) but we include them to preserve the C semantics of floating-point computations for all floats (including infinities and NaN) as specified by Flocq [9].

In fact, we define the semantics of our floating-point terms directly using the floating-point operators defined in Flocq; their real-number semantics explained in Figure 3 is actually based on theorems (already proven in Flocq) valid only if no overflow occurs.

Then, we have proved the following theorem:

Theorem 1. *Every CompCert Clight floating-point expression has the same semantics as its transformed floating-point term of VCFloat.*

This theorem is important in particular to show that the type casts introduced in the semantics of the floating-point terms of VCFloat are consistent with the implicit casts and type promotions in the semantics of Clight expressions.

In our Coq development, we have proved this theorem in terms of floating-point semantics, which means that the result also holds for infinities and NaNs.

4. VCFloat: From Floating-Point to Real-Number Expressions

On top of our core calculus of floating-point terms, which we proved consistent with the semantics of C expressions, we design and implement practical automation mechanisms to sidestep all floating-point-specific reasoning, which we describe in this section.

Given a floating-point term t, we first compute all constant subexpressions of t using Flocq and replace the corresponding subterms with their constant results.

$$
\begin{array}{rcll}
r & ::= & \mathsf{Unkn} \mid \mathsf{Norm} \mid \mathsf{Deno} & \text{Rounding annotation} \\
n & \in & \mathbb{Z} & \\
u & ::= & (f, \tau) & \text{Typed constant} \\
 & \mid & (x, \tau) & \text{Typed variable} \\
 & \mid & u \oplus^r u \mid u \ominus^r u & \text{Rounded binary operation} \\
 & \mid & u \otimes^r u \mid u \oslash^r u & \\
 & \mid & \textcircled{$\surd$}^r u & \text{Rounded unary operation} \\
 & \mid & -u \mid |u| & \text{Exact unary operation} \\
 & \mid & [u]^r_\tau & \text{Type cast} \\
 & \mid & u -^{\text{Sterbenz}} u & \text{Sterbenz exact subtraction} \\
 & \mid & u \times 2^n \mid 2^n \times u & \text{Exact multiply with power of 2}
\end{array}
$$

Figure 4: Syntax of VCFloat annotated floating-point terms

$$
\begin{array}{rlrl}
\llparenthesis (f,\tau) \rrparenthesis & = (f, \tau) & \llparenthesis (v,\tau) \rrparenthesis & = (v, \tau) \\
\llparenthesis u_1 \circledast^r u_2 \rrparenthesis & = \llparenthesis u_1 \rrparenthesis \circledast \llparenthesis u_2 \rrparenthesis & (\circledast \in \{\oplus, \ominus, \otimes, \oslash\}) & \\
\llparenthesis \textcircled{$\surd$}^r u \rrparenthesis & = \textcircled{$\surd$} \llparenthesis u \rrparenthesis & \llparenthesis [u]^r_\tau \rrparenthesis & = [\llparenthesis u \rrparenthesis]_\tau \\
\llparenthesis -u \rrparenthesis & = -\llparenthesis u \rrparenthesis & \llparenthesis |u| \rrparenthesis & = |\llparenthesis u \rrparenthesis| \\
\llparenthesis u_1 -^{\text{Sterbenz}} u_2 \rrparenthesis & = \llparenthesis u_1 \rrparenthesis \ominus \llparenthesis u_2 \rrparenthesis & & \\
\llparenthesis u \times 2^n \rrparenthesis & = \llparenthesis u \rrparenthesis \otimes (2^n, \langle \llparenthesis u \rrparenthesis \rangle) & & \\
\llparenthesis 2^n \times u \rrparenthesis & = (2^n, \langle \llparenthesis u \rrparenthesis \rangle) \otimes \llparenthesis u \rrparenthesis & &
\end{array}
$$

Figure 5: Erasure of annotated terms

Then, we have implemented an automatic tactic to make VCFloat annotate a term t with *rounding information*, as explained in this section. The goal of those rounding information annotations is to remove unnecessary rounding error terms from the real-number semantics of floating-point computations.

Annotated Terms Annotation yields an annotated term u of the extended grammar defined in Figure 4. Each rounded operator is annotated with either Norm (the result of the computation will be a normal floating-point number), Deno (the result of the computation will be a denormal floating-point number), or Unkn if unknown. Moreover, some rounded operators are annotated to be exact, such as multiplying and dividing by constants equal to a power of 2, or subtractions using Sterbenz's lemma (already proven in Flocq [9]):

Lemma 2 (Sterbenz [36])**.** *If a and b are two floating-point numbers such that $a/2 \leq b \leq a \times 2$, then $a - b$ is a floating-point number: no rounding needs to occur. (In other words, $a \ominus b = a - b$.)*

Then, we define two semantics for annotated terms: the floating-point semantics and the real-number semantics.

Annotated Terms: Floating-Point Semantics The floating-point semantics of an annotated term is the same as the semantics of the corresponding non-annotated term, with all rounding operations preserved. To this end, we define in Figure 5 the *erasure* $\llparenthesis u \rrparenthesis$ of an annotated term u as the corresponding term without annotations. Then, we define the floating-point semantics $\llbracket u \rrbracket_\rho$ of an annotated term u as $\llbracket u \rrbracket_\rho = \llbracket \llparenthesis u \rrparenthesis \rrbracket_\rho$, with all rounding operations preserved.

Similarly, we define its type $\langle u \rangle$ as the type of its erasure: $\langle u \rangle = \langle \llparenthesis u \rrparenthesis \rangle$.

Annotated Terms: Real-Number Semantics and Validity Conditions By contrast, the real-number semantics $\mathsf{R}(u, \rho)$ of an annotated term u is a real-number expression with fresh *existential* variables, and a validity condition. In Coq, we represent $\mathsf{R}(u, \rho)$ as a triple (x, W, P) where x is a real-number expression to represent the real-number value of u and P is a validity condition[2] such that their free variables all appear either in u or in the finite

[2] More precisely, in our Coq implementation, we represent x and P using *deep embedding*: x is an expression following a simple grammar of real number arithmetic, and P is represented as a list of elementary conditions, each of which is of the form yRz where y is a deeply-embedded expression following the same grammar as x, $R \in \{\leq, <\}$ and z is a real-number constant.

set $W \subseteq V \times \mathfrak{I}(\mathbb{R})$ of pairs of variables and real-number intervals, so that the following correctness theorem holds:

Theorem 3. *Let u be an annotated term and ρ be an environment such that, for every typed variable (v, τ) appearing in u, $\rho(v)$ is a finite floating-point value of type τ.*

Assume $\mathsf{R}(u, \rho) = (x, W, P)$. Then, if the validity condition P holds, then the real-number x is equal to the real-number value of the floating-point semantics $[\![u]\!]_\rho$. In other words, we have:

$$(\forall^W, P) \Rightarrow (\exists^W, [\![u]\!]_\rho = x)$$

where, if $W = \{(x_1, I_1), (x_2, I_2), \dots\}$, then we write $\exists^W, P$ for $\exists x_1 \in I_1, \exists x_2 \in I_2, \dots, P$, and similarly for $\forall$.

The generated conditions and rounding expressions are summarized in the table in Figures 6 and 7. The validity conditions enforce the following principles:

- operations must not overflow in the target type.[3]
- for rounded operators where the real-number result before rounding is r, the rounded term is of one of the following three forms, as formalized in Flocq [9]:
  - $r \times (1 + \delta)$ if annotated by Norm (i.e. the result is expected to be a normal number)
  - $r + \epsilon$ if annotated by Deno (i.e. the result is expected to be a denormal number)
  - $(r \times (1 + \delta)) + \epsilon$ if annotated by Unkn (i.e. we don't know)
- Sterbenz's condition must hold for Sterbenz's exact subtraction
- multiplying or dividing with a power of two must not gradually underflow (i.e. flush to a denormal number)
- cast to a greater type introduces no rounding error

The validity condition P is a sufficient condition for the soundness of the rounding error terms, in particular in the case where those can be optimized away. The validity condition for an expression accumulates the validity conditions of all of its subexpressions.

We write $W_1 \uplus W_2$ to denote disjoint union.[4]

Our Tactic We have implemented a Ltac tactic which, from a non-annotated floating-point term t to be evaluated in a given environment ρ, automatically generates an annotated term u, its real-number semantics x and a Coq proof term π such that $(\!|u|\!) = t$ and $\mathsf{R}(u, \rho) = (x, W, P)$ and π is a proof of $\forall^W, P$. Our tactic produces the proof term π by automatically checking the validity conditions defined in Figures 6 and 7 on the fly.

For each annotation of one operation, its subexpressions are first recursively annotated and their corresponding real-number expressions computed. Then, the validity condition for one possible annotation for the considered operation can be checked using the real-number expressions computed from the already annotated subexpressions.

For one operation, once its subexpressions have already been annotated:

- For a subtraction, Sterbenz's condition is checked first
- For a product or a quotient, constants are first checked to be equal to a power of 2

[3] Indeed, if any operation overflows, it is flushed to a floating-point infinity, which is not representable in Coq real numbers and cannot be reasoned upon. Our VCFloat framework only focuses on finite floating-point numbers, which can be associated to a meaningful real-number value.

[4] In our Coq formalism, we implemented R with an additional argument and an additional result to record the domain of variables that are already used, to ensure that such unions are always disjoint

$$
\begin{array}{lrcl}
 & \mathsf{R}(u_1 \circledast^{\mathsf{Unkn}} u_2, \rho) & = & (r, \\
 & & & W_1 \uplus W_2 \uplus \\
 & \text{(Rounding errors)} & & \{(\delta, [-2^D, 2^D])\} \uplus \\
 & & & \{(\epsilon, [-2^E, 2^E])\}, \\
 & & & P_1 \wedge P_2 \wedge \\
 & \text{(No overflow)} & & |r| < 2^{emax}) \\
\text{where} & \mathsf{R}(u_1, \rho) & = & (r_1, W_1, P_1) \\
\text{and} & \mathsf{R}(u_2, \rho) & = & (r_2, W_2, P_2) \\
\text{and} & r & = & (r_1 * r_2) \times (1 + \delta) + \epsilon \\
\text{and} & \langle u_1 \rangle \sqcup \langle u_2 \rangle & = & (prec, emax) \\
\text{and} & D & = & -prec \\
\text{and} & E & = & 2 - emax - prec \\
\\
 & \mathsf{R}(u_1 \circledast^{\mathsf{Norm}} u_2, \rho) & = & (r, \\
 & & & W_1 \uplus W_2 \uplus \\
 & \text{(Rounding error)} & & \{(\delta, [-2^{-prec}, 2^{-prec}])\}, \\
 & & & P_1 \wedge P_2 \wedge \\
 & \text{(No overflow)} & & |r| < 2^{emax} \wedge \\
 & \text{(Result will be normal)} & & |r_1 * r_2| \geq 2^{3 - emax}) \\
\text{where} & \mathsf{R}(u_1, \rho) & = & (r_1, W_1, P_1) \\
\text{and} & \mathsf{R}(u_2, \rho) & = & (r_2, W_2, P_2) \\
\text{and} & r & = & (r_1 * r_2) \times (1 + \delta) \\
\text{and} & \langle u_1 \rangle \sqcup \langle u_2 \rangle & = & (prec, emax) \\
\\
 & \mathsf{R}(u_1 \circledast^{\mathsf{Deno}} u_2, \rho) & = & (r, \\
 & \text{(Rounding error)} & & W_1 \uplus W_2 \uplus \{(\epsilon, [-2^E, 2^E])\}, \\
 & & & P_1 \wedge P_2 \wedge \\
 & \text{(No overflow)} & & |r| < 2^{emax}) \\
 & \text{(Result will be denormal)} & & |r_1 * r_2| < 2^{3 - emax}) \wedge \\
\text{where} & \mathsf{R}(u_1, \rho) & = & (r_1, W_1, P_1) \\
\text{and} & \mathsf{R}(u_2, \rho) & = & (r_2, W_2, P_2) \\
\text{and} & r & = & (r_1 * r_2) + \epsilon \\
\text{and} & \langle u_1 \rangle \sqcup \langle u_2 \rangle & = & (prec, emax) \\
\text{and} & E & = & 2 - emax - prec
\end{array}
$$

and similarly for rounded unary operators and cast to a non-greater type

Figure 6: Real-number semantics and validity conditions for floating-point annotated terms: rounded cases

$$
\begin{array}{lrcl}
 & \mathsf{R}((f, \tau), \rho) & = & (f, \varnothing, \mathsf{True}) \\
 & \mathsf{R}((v, \tau), \rho) & = & (\rho(v), \varnothing, \mathsf{True}) \\
\\
 & \mathsf{R}(-u, \rho) & = & (-x, W, P) \\
 & \mathsf{R}(|u|, \rho) & = & (|x|, W, P) \\
\text{where} & \mathsf{R}(u, \rho) & = & (x, W, P) \\
\\
 & \mathsf{R}([u]_{(prec', emax')}, \rho) & = & (x, W, P) \\
\text{where} & \mathsf{R}(u, \rho) & = & (x, W, P) \\
\text{and} & \langle u \rangle & = & (prec, emax) \\
\text{and} & prec & \leq & prec' \\
\text{and} & emax & \leq & emax' \\
\\
 & \mathsf{R}(2^n \times u, \rho) & = & (2^n \times x, W, P \wedge \\
 & \text{(No overflow)} & & |2^n \times x| < 2^{emax}) \\
\text{where} & \mathsf{R}(u, \rho) & = & (x, W, P) \\
\text{and} & \langle u \rangle & = & (prec, emax) \\
\text{and} & 0 & \leq & n \\
\\
 & \mathsf{R}(2^n \times u, \rho) & = & (2^n \times x, W, P \wedge \\
 & \text{(No gradual underflow)} & & 2^{2 - emax + n} \leq |x|) \\
\text{where} & \mathsf{R}(u, \rho) & = & (x, W, P) \\
\text{and} & \langle u \rangle & = & (prec, emax) \\
\text{and} & n & < & 0 \\
\\
 & \mathsf{R}(u_1 -^{\mathsf{Sterbenz}} u_2, \rho) & = & (r_1 - r_2, W_1 \uplus W_2, P \wedge \\
 & \text{(Sterbenz's condition)} & & r_2/2 \leq r_1 \leq r_2 \times 2) \\
\text{where} & \mathsf{R}(u_1, \rho) & = & (r_1, W_1, P_1) \\
\text{and} & \mathsf{R}(u_2, \rho) & = & (r_2, W_2, P_2)
\end{array}
$$

Figure 7: Real-number semantics and validity conditions for floating-point annotated terms: exact cases

- Otherwise, Norm, Deno, Unkn are tried in this order, stopping at the first success. If all fail, then it means that overflow cannot be ruled out, and so the overall tactic fails: no other annotations are tried for subexpressions.

Once an annotation is successfully found for one operation, it is no longer changed, so the total number of checks is linear in the number of operations (tree nodes) in the expression.

Our tactic automatically checks the corresponding validity conditions using Coq-Interval [30, 31], a Coq proof and tactic library for automatic interval arithmetic. The implementation of VCFloat with the connection to CompCert Clight totals less than 8,000 lines of Coq code (3,000 lines of specification and tactics and 5,000 lines of proof).

Examples Our tactic allows to automatically sort out all floating-point issues (namely the absence of overflow and the shape of rounding error terms). Consider the following C expressions:

- `2.0f * (float) x - 3.0`, where `x` is a double-precision floating-point number known to have a value between 1 and 2. Then, VCFloat first automatically converts this expression to the non-annotated core expression $(2_{(24,128)} \otimes [x]_{(24,128)}) \ominus 3_{(53,1024)}$. Then, $[x]_{(24,128)}$ is automatically annotated with Norm since x is large enough to be represented by a normal floating-point number. Then, the product $2_{(24,128)} \otimes [x]^{\mathsf{Norm}}_{(24,128)}$ is automatically annotated to become $2^1 \times [x]^{\mathsf{Norm}}_{(24,128)}$. Then, the subtraction $(2^1 \times [x]^{\mathsf{Norm}}_{24,128}) \ominus 3_{(53,1024)}$ is automatically annotated to become $(2^1 \times [x]^{\mathsf{Norm}}_{24,128}) -^{\mathsf{Sterbenz}} 3_{(53,1024)}$. Finally, VCFloat automatically computes the real-number expression for this fully annotated expression as $2 \times (x \times (1+\delta)) - 3$ where δ is a free variable representing some unknown real number in $[-2^{-24}, 2^{-24}]$.
- `DBL_MAX * (x + .5)`, where `x` is a double-precision floating-point number known to have a value greater than .5. Then, VCFloat first automatically converts this expression to the non-annotated core expression $((2^{1024})_{(53,1024)} \otimes (x \oplus .5_{(53,1024)}))$. Then, $x \oplus .5_{(53,1024)}$ is automatically annotated with Norm since $x + .5$ is large enough to be represented by a normal floating-point number. However, the product $((2^{1024})_{(53,1024)} \otimes (x \oplus^{\mathsf{Norm}} .5_{(53,1024)}))$ cannot be annotated since overflow cannot be ruled out. So, the tactic immediately and unrecoverably fails, without even trying Unkn for the subexpression $x \oplus .5_{(53,1024)}$.

Thanks to VCFloat, the user can directly reason on the real-number value of a Clight floating-point computation, as we illustrate with our complete program analysis example, which we describe in the next section.

5. Application: Certified Energy-Efficient Radar Image Processing

We apply our VCFloat framework to the complete verification of an energy-efficient C implementation of a radar image processing algorithm, namely Synthetic Aperture Radar [35] image backprojection [19]. Our high-level goal is to estimate certified error bounds introduced by floating-point computations, and to evaluate their variations when introducing approximations and reducing precision for some floating-point computations. Indeed, since image processing on embedded radar systems involves heavy numerical computations onboard energy-constrained platforms, we hereby want to show that practical energy-efficient optimizations in floating-point computations can be achieved with provably bounded noise, thus providing some strong formal guarantee on the quality of the synthesized radar image.

Require: $\mathsf{N\_PULSES} \in \mathbb{N}^{>0}$: number of pulses
Require: $\mathsf{N\_RANGE\_UPSAMPLED} \in \mathbb{N}^{>0}$: range of samples
Require: $\mathsf{BP\_NPIX\_X}, \mathsf{BP\_NPIX\_Y} \in \mathbb{N}^{>0}$: size of the image
Require: $\underline{data}[0..\mathsf{N\_PULSES}-1][0..\mathsf{N\_RANGE\_UPSAMPLED}-1] \in \mathbb{C}$: sensor data
Require: $\overrightarrow{platpos}[0..\mathsf{N\_PULSES}-1] \in \mathbb{R}^3$: position of the radar platform
Require: $z[0..\mathsf{N\_PULSES}-1][0..\mathsf{BP\_NPIX\_Y}-1][0..\mathsf{BP\_NPIX\_X}-1] \in \mathbb{R}$: measured heights
Require: $dxdy \in \mathbb{R}$: real distance between two pixels (m)
Require: $r_0 \in \mathbb{R}$: radial mean distance between plane and target (m)
Require: $dr \in \mathbb{R}$: range bin resolution (m)
Require: $ku \in \mathbb{R}$: carrier wavenumber
  for $y := 0$ **to** $\mathsf{BP\_NPIX\_Y}-1$ **do**
    $py := (y + \frac{1-\mathsf{BP\_NPIX\_Y}}{2}) \times dxdy$
    for $x := 0$ **to** $\mathsf{BP\_NPIX\_X}-1$ **do**
      $px := (x + \frac{1-\mathsf{BP\_NPIX\_X}}{2}) \times dxdy$
      $\underline{image}[y][x] := 0 \in \mathbb{C}$
      for $p := 0$ **to** $\mathsf{N\_PULSES}-1$ **do**
        $r := \|\overrightarrow{platpos}[p] - (px, py, z[p][y][x])\|$
        $bin := (r - r_0)/dr$
        $\underline{sample} := \mathsf{binSample}(\mathsf{N\_RANGE\_UPSAMPLED}, \underline{data}[p], bin)$
        $\underline{matchedFilter} := \exp(2\mathbf{i} \times ku \times r)$
        $\underline{image}[y][x] := \underline{image}[y][x] + \underline{sample} \times \underline{matchedFilter}$
      end for
    end for
  end for
  return $\underline{image}$

Figure 8: SAR backprojection, general algorithm

Require: $\mathsf{N\_RANGE\_UPSAMPLED} \in \mathbb{N}^{>0}$: range of samples
Require: $\underline{data}[0..\mathsf{N\_RANGE\_UPSAMPLED}-1] \in \mathbb{C}$: sensor data
Require: $bin \in \mathbb{R}$
  if $0 \le bin < \mathsf{N\_RANGE\_UPSAMPLED} - 1$ **then**
    $k := \lfloor bin \rfloor$
    $w := bin - k$
    return $(1-w) \times \underline{data}[k] + w \times \underline{data}[k+1]$
  else
    return 0
  end if

Figure 9: Bin sampling: linear interpolation

In classical radar imaging, the image resolution is limited by the aperture of the physical antenna. To relax this constraint, Synthetic Aperture Radar (SAR) allows simulating much larger apertures by embedding classical radar sensors onto a platform onboard a plane flying over the target zone to be imaged. Then, the radar periodically sends *pulse signals* down to the ground target and measures the amplitude and phase of each returned pulse signal, all along the flight path of the plane. The data collected from these multiple platform locations about the same target zone is then reprocessed by software to synthesize an image. Backprojection is such a software image reconstruction algorithm for SAR; the general algorithm for backprojection is described in Figure 8 (which also defines the notations of constants in this section.)

SAR image backprojection actually depends on a $\underline{\mathsf{binSample}}$ function, which interpolates a *sample* from the discrete measured sensor data. In our case here, we choose *linear interpolation* (see Figure 9).

The goal of our verification project is to compute upper bounds on the *signal-noise ratio* (SNR) of the synthetic image $\underline{image}$ computed by SAR backprojection with respect to a *gold-standard* image $\underline{image_0}$: $SNR := \frac{\|\underline{image_0}\|^2}{\|\underline{image} - \underline{image_0}\|^2}$, where $\|M\|$ is the 2-norm of matrix M, defined by $\|M\|^2 = \sum_x \sum_y |M[x][y]|^2$. A human-readable value of SNR is expressed in dB (i.e., $10 \log SNR$).

To estimate the noise actually introduced by floating-point computation roundings and approximations, we assume that the gold-standard image is computed with the real number algorithm of SAR backprojection with linear interpolation[5], and the actual image is computed with our implementation in floating-point number with approximations for square root and sine.

Since we have no information on the sensor data or platform position, we have no information on the "signal" (i.e., the numerator) part of the SNR, so we are interested in an upper bound on the denominator. It is straightforward to see that, if for all pixels, $|\Re(\underline{image}[y][x] - \underline{image_0}[y][x])| \leq \varepsilon$ and $|\Im(\underline{image}[y][x] - \underline{image_0}[y][x])| \leq \varepsilon$, then:

$$\|\underline{image} - \underline{image_0}\|^2 \leq 2 \times \mathsf{BP\_NPIX\_X} \times \mathsf{BP\_NPIX\_Y} \times \varepsilon^2$$

since $|\underline{z}|^2 = |\Re\underline{z}|^2 + |\Im\underline{z}|^2$. So it is enough to compute an absolute error bound ε on the computation of the real or the imaginary part of all pulse contributions for one pixel image.

In this paper, we assume that the input data are exact, and we are only interested in the implementation error. Indeed, propagation of input data errors can be analyzed directly on the real algorithm, independently of any implementation. In particular for absolute error, when SAR backprojection is used in a larger context where input data is computed by another algorithm introducing some error, we simply add both the absolute propagation error and the absolute implementation error for SAR backprojection.

Overview of our Implementation and Proofs We implement SAR backprojection with linear interpolation in the CompCert Clight subset of C. In Figure 8, bin is computed in double-precision floating-point numbers through an approximate, adaptive computation for the norm, which we study in detail in Section 5.1. Then, we perform linear interpolation (Section 5.2) in single-precision floating-point numbers. We compute the complex exponential using approximate sine and cosine, which we study in detail in Section 5.3. Then, the contribution of one pulse is obtained by their product, which introduces several further rounding errors, which our VCFloat framework handles automatically. Finally, we compute the final sum of all pulse contributions for one pixel using naive summation, which we study in detail in Section 5.4. We summarize our overall bound on the noise introduced by our C floating-point implementation in Section 5.5. We finally comment on the energy savings of our code in Section 5.6.

Our implementation totals more than 120 lines of C code. We apply VCFloat to compute and prove error bounds using Coq 8.5beta2 and the trunk version of Coq-Interval [31].

Beyond numerical bounds, we assume nothing more on the input data — in particular, no statistical argument. The reason is that we want to construct proofs of *worst-case* error bounds. Also, Coq and most other proof assistants provide poor support for statistical or probabilistic reasoning, if any at all. Moreover, it is known [32] that floating-point rounding errors are not random (i.e., floating-point computations over uniformly distributed floating-point values do not yield uniformly distributed floating-point errors), which makes statistical reasoning about rounding errors and their propagation even harder, even on paper.

5.1 Approximate Norm

To save energy while preserving the quality of the synthesized radar image, we introduce a tradeoff between the amount of computations performed by the program and the resulting precision. To this end, we compute the norm using a Taylor approximation for square root. In this subsection, we are studying the total absolute error bound introduced by our implementation of the square root for the norm. This error contains three sources of error:

- Propagation of the computation errors introduced in the computation of the squared norm
- Method error introduced by the Taylor approximation
- Rounding errors introduced by the actual floating-point computations

Error Propagation. Coq and its standard library allow us to easily provide a formal proof of $\sqrt{x'} - \sqrt{x} = (x' - x)/(\sqrt{x'} + \sqrt{x})$. Using this rewriting, since we already know a bound on $x' - x$ (which is the error to propagate) and individual bounds on x and x', we avoid correlation problems and thus we can directly use Coq-Interval [31] on the rewritten expression to derive a bound on the propagation of the argument error in the square root.

Method Error. Instead of always computing the square root using the standard square root function specified by IEEE 754 and implemented by the standard mathematical C library, we approximate the square root with a second-order Taylor x-polynomial S around some x_0 defined as $S = \sqrt{x_0} + \frac{x-x_0}{2\sqrt{x_0}} - \frac{(x-x_0)^2}{8(\sqrt{x_0})^3}$ where $\sqrt{x_0}$ was previously computed and memorized.

Based on the univariate Taylor theorem with mean value formalized in CoqApprox [14], we know that $\sqrt{x} - S = \frac{(x-x_0)^3}{16(\sqrt{\xi})^5}$ for some $\xi \in [x_0, x] \cup [x, x_0]$, which allows us to prove the following method error bound:

Lemma 4. *On the half-line of positive numbers, consider the disc of some radius* $\mathsf{TAU\_S2} > 0$ *centered on some point* $x_0 > \mathsf{TAU\_S2}$. *For any* $x > 0$ *within this disc (i.e.* $|x - x_0| \leq \mathsf{TAU\_S2}$*), we have* $|\sqrt{x} - S| \leq \frac{(\mathsf{TAU\_S2})^3}{16 \times (x_0 - \mathsf{TAU\_S2})^{5/2}}$

Rounding Errors and C Implementation We now compute and certify an absolute rounding error bound in the evaluation of our C implementation of S for $|x - x_0| \leq \mathsf{TAU\_S2}$, assuming that $\sqrt{x_0}$ is not computed exactly but accurately rounded in double-precision floating-point numbers like other arithmetic operations in the polynomial evaluation.

To minimize the number of computations and thus both energy consumption and the number of potential sources of rounding errors, we compute S as follows:

$$\begin{array}{llll} n_0 & = \frac{1}{4x_0} & n & = n_0 \times (x - x_0) \\ u & = \sqrt{x_0} \times (2 \times n) & S & = \sqrt{x_0} + u \times (1 - n) \end{array}$$

Our VCFloat framework applied to the implementation of this algorithm automatically highlights the following rounding errors:

- VCFloat determines that $4x_0$ is computed exactly (since 4 is a power of 2). So the only rounding error introduced in the computation of n_0 is the one introduced for $1/(4x_0)$.
- From the bounds on x and x_0, VCFloat determines that the hypotheses of Sterbenz's theorem are satisfied, so $(x - x_0)$ introduces no rounding error. Thus, the computation of n only introduces one new rounding error, in addition to the one in n_0.
- Since multiplying by 2 never introduces rounding error, the computation of u introduces only one further rounding error.
- Finally, three further rounding errors are introduced by the computation of S.

We implement approximate square root in an *adaptive* fashion: the value of x_0 is not determined upfront but may change during the computation. The code of our Clight implementation is shown in Figure 10. We choose to *dynamically* adapt the computation of x_0: we assume that at any point, the correctly rounded value s_0 of $\sqrt{x_0}$ is available, as well as the computed value of $n_0(x_0)$, and we use them with our Taylor approximation for all x until $|x - x_0| \geq \mathsf{TAU\_S2}$. In that case, we replace x_0 with x and compute

[5] We assume that linear interpolation is our reference algorithm for interpolation.

```c
#define TAU_S2 25000

double x0 = 1.0;       /* dummy starting value */
double s0 = 1.0;       /* sqrt(x0) correctly rounded */
double n0 = 1.0 / 4.0; /* computed value of n0(x0) */

void adaptive_sqrt (double* res, double x) {
     const double d = x - x0;
     if ((d < - TAU_S2) || (d > TAU_S2)) {
        const double s = sqrt(x);
        x0 = x;
        s0 = s;
        n0 = 1.0 / (4.0 * x);
        *res = s;
     } else {
        const double s = s0;
        const double n = n0 * d;
        const double u = s * (2 * n);
        *res = s + u * (1 - n);
     }
}
```

Figure 10: Clight implementation of adaptive square root

its corresponding correctly rounded square root s_0 and $n_0(x_0)$ and use those values for further norm computations. In other words, x_0 serves as a *pivot*.

Our adaptive implementation allows reducing the number of accurate square root computations (and thus save energy) while staying within the total implementation error bound determined by the Taylor approximation of square root within a disc of constant radius, by contrast to Tang et al. [34], who use the Taylor approximation everywhere and thus dramatically reduce the overall precision of their norm computation.

5.2 Linear Interpolation

The next step in the SAR backprojection algorithm is to obtain the measured signal corresponding to the current pixel and pulse. However, the signal is measured on a discrete, regularly-spaced set of antennas, represented in software as an integer-indexed array. To obtain the signal, we would need to index the array using the range bin value corresponding to each pixel and pulse. Since this value is not necessarily an integer, we need to interpolate the signal from the next higher and lower integer values. In this work, we use a linear interpolation method.

The computation of the range bin introduces some error. Such error may introduce a non-continuous error in the computation of the integer k used to index the data array (see Figure 9). So in this case, the overall error on sampling will include not only rounding errors, but also some errors due to the "wrong" choice of the data array cell. In this section, we are interested in the latter error.

On the one hand, we need to know a bound on $|\underline{data}[p][i+1] - \underline{data}[p][i]|$. More precisely, we formally proved the following lemma:

Lemma 5. *Let bin' be the computed value of the ideal range bin. Assume that the guard test succeeds for both bin' and bin (in particular, this is true if both values are between 0 and $\mathsf{N\_RANGE\_UPSAMPLED} - 1$).*

Assume that the absolute error between bin and bin' is δ, and that, for any $i \in [0, \mathsf{N\_RANGE\_UPSAMPLED}-1] \cap \mathbb{N}$ and for any pulse p, we have the following "smoothness" condition on the data: $|\Re(\underline{data}[p][i+1]) - \Re(\underline{data}[p][i])| \leq \frac{\epsilon}{2\delta\sqrt{2}}$ Then the absolute error propagates to the real parts of the ideal *values of the linear interpolation as $\epsilon/\sqrt{2}$.*

Furthermore, if the angle is fixed, and if the absolute error on bin does not change the branches taken by the guard test, then absolute error ϵ is introduced in the real part of the ideal value of the contribution of one pulse for one pixel.

The same also holds for the imaginary parts, mutatis mutandis.

Lemma 5 assumes that the angle is already fixed; however, errors introduced in the computation of r also propagate to the angle, and both propagated errors must be added together.

On the other hand, the test that guards against out-of-bound array accesses may also suffer from the error introduced in the computation of bin. So we need to account for the cases where the branches followed are not the same for the ideal value of bin as for its computed value. In other words, we need *boundary conditions* to formally prove the following:

Lemma 6. *Under the conditions of Lemma 5 (except that the test may have different behaviors in the ideal computation than in the floating-point computation), assume furthermore that we have the following boundary conditions, for any $\eta \in [0, \delta]$:*

$$|(1-\eta) \times \Re(\underline{data}[p][1]) + \eta \times \Re(\underline{data}[p][0])| \leq \epsilon/\sqrt{2}$$

$$|(1-\eta) \times \Re(\underline{data}[p][\mathsf{N\_RANGE\_UPSAMPLED} - 1]) + \eta \times \Re(\underline{data}[p][\mathsf{N\_RANGE\_UPSAMPLED} - 2])| \leq \epsilon/\sqrt{2}$$

Then the absolute error propagates to the real parts of the ideal *values of the contribution of one pulse for one pixel as ϵ. The same also holds for the imaginary parts, mutatis mutandis.*

Those formal standalone proofs are needed by the overall proof. Verification tools that specialize in floating-point computations such as Gappa [29] cannot handle those theorems. So, it is necessary to be able to integrate the formal proofs of these error propagation results to the computations in interval arithmetic and the rest of the proofs, which once more advocates for our approach based on a unified Coq-only verification framework.

5.3 Approximate Sine

In this section, we discuss the derivation of polynomial approximations to $\sin\,\theta$ and $\cos\,\theta$ that minimize the maximum error over the entire domain $\mathbb{R}$. Because of this minimax property, these polynomials converge uniformly with small error in relatively few terms. This makes them good candidates for establishing tight theoretical error bounds on polynomial approximations.

Argument Reduction Accurate implementations of argument reduction for trigonometric functions have been proved in Gappa [29] in the context of the development of the CRlibm [33] correctly rounded mathematical library. However, such implementations can be energy-costly. Since we are ready to trade some accuracy for energy-efficiency, as long as the accuracy loss can be provably bounded, we can focus on certifying the error of a naive implementation of argument reduction.

Our C implementation for argument reduction is presented in Figure 11. Our VCFloat framework addresses most of its floating-point steps automatically except calls to ISO C99 `copysign` which is not part of CompCert's supported built-in floating-point operations[6]. The purpose of using `copysign` is to manipulate the sign bit of a floating-point number while avoiding tests and cache misses in the C implementation. In particular, for our purposes, we have manually proved that the sign bit of a product is still meaningful if the product overflows. So, through our proof using VCFloat and Coq-Interval [31], we found that our naive argument reduction introduces absolute error at most $3567 \cdot 2^{-31}$ for initial argument of magnitude less than 2^{30}, due to both rounding errors and approximations of π.

Polynomial Approximation We now concentrate on finding the polynomial of order N that minimizes the maximum deviation

[6] So we specified it using Flocq [9]. We claim that it should be possible to prove the correctness of a platform-specific implementation of `copysign` in the flavour of Boldo et al.'s bitwise semantics preservation proofs for optimizations of floating-point computations [12].

```
#define BP_PI_2_SINGLE 0xc90fdb.0p-23f

/* Input: arg, where -PI <= arg <= PI (maybe with some error)
   Output: *sine = sin(arg); *cosine = cos(arg) */
inline void res_sin_cos_pol
(float *sine, float *cosine, float arg) {
    const float y  = /* y in [-PI/2,PI/2] */
          BP_PI_2_SINGLE - fabsf(arg);
    const float ay = /* ay in [0,PI/2] */
          fabsf(y);
    const float z  = BP_PI_2_SINGLE - ay;
    *cosine = copysignf(res_pol(ay), y);
    *sine = copysignf(res_pol(fabsf(z)), z*arg);
}

#define BP_PI_DOUBLE 0x1921fb54442d18.0p-51
#define BP_2_PI_DOUBLE 0x1921fb54442d18.0p-50
#define BP_INV_2_PI_DOUBLE 0x145f306dc9c883.0p-55

/* Input: arg
   Output: *sine = sin(arg); *cosine = cos(arg) */
inline void res_sin_cos(float *sine, float *cosine, double arg) {
    const double red    =  /* red in [0, 2*PI] */
          arg - BP_2_PI_DOUBLE * floor(BP_INV_2_PI_DOUBLE * arg);
    const float  pi_red = /* pi_red in [-PI, PI] */
          (float) (BP_PI_DOUBLE - red);
    res_sin_cos_pol(sine, cosine, pi_red);
    *cosine = -*cosine;
}
```

Figure 11: Argument reduction for sine/cosine

from sin θ for arguments in the restricted domain $[0, \pi/2]$, while satisfying boundary conditions on the value and the first derivative at the two endpoints. This ensures an approximation that is everywhere smooth and continuous, can never go outside the valid range $(-1, 1)$, and (with the additional error introduced by argument reduction) is as accurate for any argument in $\mathbb{R}$ as the worst case error in the domain $[0, \pi/2]$. This error is to be minimized by choice of fit coefficients.

We will denote our polynomial approximation of order N by $s(x|N, c)$, where x denotes an argument in the restricted domain, N denotes the order of the polynomial, and c denotes the vector $[c(0), c(1), c(2) \dots c(N)]$ of polynomial coefficients, so that we have $s\left(x|N, c\right) = \sum_{n=0}^{N} c\left(n\right) x^n$, leading to the following identity for its derivative: $s'\left(x|N, c\right) = \sum_{n=1}^{N} c\left(n\right)\, n\, x^{n-1}$.

Setting the value to zero and the slope to 1 at $x = 0$, and the value to 1 and the slope to zero at $x = \pi/2$ provides four linear equations of constraint in the $N + 1$ coefficients $c(n)$.

The four equations of constraint in N unknown polynomial coefficients can be written as $Ac = b$, where c is a column vector containing the polynomial coefficients, $b = \left[\begin{array}{cccc} 0 & 1 & 1 & 0 \end{array}\right]^\top$ and A is the $4 \times N$ matrix

$$A = \left[\begin{array}{cccccc} 1 & 0 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & 0 & \dots & 0 \\ 1 & \frac{\pi}{2} & \left(\frac{\pi}{2}\right)^2 & \left(\frac{\pi}{2}\right)^3 & \dots & \left(\frac{\pi}{2}\right)^{N-1} \\ 0 & 1 & 2\left(\frac{\pi}{2}\right) & 3\left(\frac{\pi}{2}\right)^2 & \dots & (N-1)\left(\frac{\pi}{2}\right)^{N-2} \end{array}\right] \quad (1)$$

The constraints ensure that $c(0) = 0$ and $c(1) = 1$, so that the number of nontrivial coefficients is really $N - 2$. The order 3 polynomial ($N = 4$) is of special interest because the polynomial is completely determined by the equations of constraint, since A is square.

In order to obtain polynomial fits of higher order, it is necessary to perform an optimization, since the system of linear equations is underdetermined if $N > 4$. For purposes of establishing precision bounds, we minimize the maximum deviation of the fit polynomial from the true sin function subject to the four linear equations of constraint.

$$\text{minimize } \left\{\max_x \left|\sin\left(x\right) - s\left(x\left|N, c\right.\right)\right|\right\} \text{ subject to } Ac = b.$$

This problem qualifies as a convex optimization problem, since the functional to be minimized is convex in the unknowns c and the equations of constraint are linear in the components of c [13]. Optimizations of this kind can be solved efficiently using modern convex solvers. For purposes of this work, we used the MATLAB routine SeDuMi [37] within the convex optimization framework CVX [22] to perform the needed optimizations.

The obtained polynomial for $N = 7$, with absolute bound error $1.10186E - 06$, and coefficients rounded to single-precision, is:

$$\begin{array}{rl} P_6(x) = ((((((& -16392343/17179869184 \\ ) \times x & + 21895786/2147483648 \\ ) \times x & + -15145514/8589934592 \\ ) \times x & + -22264200/134217728 \\ ) \times x & + 17533087/137438953472 \\ ) \times x & + 1 \\ ) \times x & \end{array} \quad (2)$$

Proof of C Implementation We efficiently derived polynomial approximations of sine through convex optimization. However, to derive such approximations, we used MATLAB-based tools relying on floating-point computations. This means that all computations performed in MATLAB, and in particular the coefficients of the polynomial approximations and their computed error bounds, bear some rounding errors. Moreover, although MATLAB claims to follow the IEEE 754 standard for floating-point computations, its rounding mode is not clearly specified and could very well conflict with the rounding mode used by the client C code. Those two sources of inaccuracies thus weaken the confidence in the error bounds computed using the method that we described in the previous subsection. So, they have to be further backed by a formal method, which we describe here. Moreover, the error bound that we obtained there is only the approximation method error, and ignores the rounding errors introduced by implementing the evaluation of the polynomial approximation with floating-point numbers, so we have to formally study them as well.

We integrated these polynomial approximations into our verification by computing and proving the correctness of the absolute error bounds for sine computed using the 6-order approximation P_6 of sine obtained in Equation 2, and evaluated in single-precision floating-point. These absolute error bounds take into account both method error (replacing sine with P_6) and rounding errors (evaluating P_6 in single-precision floating-point).

For our proof, we considered the argument range $[0, 8/5]$ which is a large superset of $[0, \pi/2]$. Actually, this argument range is a superset of $[0, 3373262642 \cdot 2^{-31}]$ which is the range of the reduced argument that we computed before.

Coq-Interval [31] supports interval arithmetic with a small set of analytic transcendental functions, including sine. So, we can use Coq-Interval to compute and prove a bound of $|P_6(x) - \sin x|$. We find that the absolute method error introduced is at most $2388 \cdot 2^{-31} \simeq 1.112 \cdot 10^{-6}$, similar to the bound empirically found in Equation 2.

Finally, our VCFloat framework allows us to find that our C implementation in single precision introduces absolute rounding error at most $1235 \cdot 2^{-31}$. This means that the bound on the error introduced by the core computation is of similar amount of magnitude to the error bound for argument reduction.

5.4 Absolute Error of Approximate Sum Computation

To compute the value of one pixel image, we need to sum all the contributions of all pulses for this pixel. To save on energy and the number of floating-point operations, we choose to stick to naive summation, instead of adopting Kahan's summation [25], known to significantly improve the rounding error by a running compensation. Rounding errors for a wide range of floating-point summation algorithms are studied by Higham [23], but this survey is based on a naive rounding model which neglects gradual underflow. In the following, we describe our formal proof of a worst-case error bound for naive summation with both rounding errors in the presence of gradual underflow and propagation of summand errors.

Let $q \in \mathbb{R}^{\mathbb{N}}$ be a sequence of ideal real values, and $Q_{n+1} = \sum_{i=0}^{n} q_i$ be their ideal sum ($Q_0 = 0$). Let $\tilde{q}$ be the sequence of approximate values actually computed for each term of q. Then, the approximate sum $\tilde{Q}$ actually computed with further rounding errors δ and ϵ introduced at each step has the following shape: $\tilde{Q}_0 = 0$ and $\tilde{Q}_{n+1} = (\tilde{Q}_n + \tilde{q}_n)(1 + \delta_n) + \epsilon_n$.

We want to find an absolute error bound for the computation of Q, i.e., bound $|\tilde{Q}_n - Q_n|$.

Assume that $|q| \leq B_q$; $|\tilde{q} - q| \leq B$; $|\delta| \leq B_\delta$, and $|\epsilon| \leq B_\epsilon$. Then we easy prove that $|\tilde{Q}_{n+1} - Q_{n+1}| \leq |\tilde{Q}_n - Q_n|M + nL + K$ where $M = 1 + B$; $L = B_q + B$; and $K = L + B_\delta M + B_\epsilon$.

Thus, we have: $|\tilde{Q}_n - \tilde{Q}_n| \leq D_n$ where D is the recursive real sequence defined as $D_0 = 0$ and $D_{n+1} = D_n M + nL + K$, for which we prove that $D_n = \frac{1-M^n}{1-M}\left(K - \frac{L}{1-M}\right) + \frac{nL}{1-M}$, assuming $M \neq 1$ and $0^0 = 1$. Although we found this formula through a pen-and-paper proof sketch based on formal derivation of polynomials, our actual Coq proof does not need such an argument and works out directly by induction on n using basic real field algebra.

It is interesting to know that as long as the bounds of the terms of the sum are uniform, the error bound on the sum does not depend on the order in which the terms are summed. However, our result assumes that the sum is computed linearly: it cannot apply to cases where partial sums are first computed and then summed up. In particular, our result does not apply to divide-and-conquer or other parallel summing strategies. For our implementation of SAR backprojection, this is not the case: whereas the contributions of any two different pixels can be computed in parallel since they are independent of each other, we have chosen to sequentially compute and sum the contributions of each pulse for one pixel.

5.5 Summary and Interpretation of Error Bounds

We first performed our proof in the case where all steps of the sum are computed in single precision. Given the low quality of the obtained bound, we decided to tackle the case where all steps of the sum are computed in double precision, but the final result is cast back to single precision. We further extended our proofs by replacing all or part of our approximations with standard mathematical functions assumed to be correctly rounded, and by studying the impact of the precision choice for the computation of the linear interpolation. The table in Figure 12 summarizes the absolute implementation error bound ε on the real or imaginary part of all pulse contributions for one pixel, and the upper bound on the denominator of the SNR ($\sum_{y,x} |\underline{image}[y][x] - \underline{image_0}[y][x]|^2$), computed as $D = 2 \times \mathsf{BP\_NPIX\_X} \times \mathsf{BP\_NPIX\_Y} \times \varepsilon^2$. The bounds on the input data are taken from the PERFECT suite [4] for three image sizes (512×512, 1024×1024 and 2048×2048 pixels).

Our results show that if the steps of the sum are computed in single precision, then the accumulation of rounding errors introduced by the summation actually overwhelms all other implementation errors, and goes well beyond acceptable losses. However, if the steps of the sum are computed in double precision, the worst case for large images with approximations enabled introduces losses up to 44 dB, which is nearly acceptable in practice (if the PERFECT data suite arbitrarily sets the ideal SNR to 140 dB and SNR ranges around 100 dB and beyond are considered acceptable).

The implementation error grows with the size of the image and the number of pulses. This may seem surprising since images of bigger size with more pulses are supposed to reduce the error in practice, but error growth is actually due to the accumulation of implementation and rounding errors in the final sum. Thus in practice, the input data seems to have statistical properties prone to lower or cancel the error. We are not relying on any such statistical assumptions in our verification work, which is focused on a certified proof for worst-case error bounds. From the radar image processing point of view, it means that we do not assume anything on the behavior of the adversary with respect to terrain camouflage or signal scrambling. A more thorough analysis taking into account such statistical arguments might reveal potential weaknesses in the implementation, which an adversary might exploit to worsen the quality of the synthesized image.

5.6 Performance Measurements

To clearly assess the gain introduced by approximations, we measured the energy and power performance improvements introduced by our C implementation of SAR backprojection.

We conducted performance tests on a Intel SandyBridge machine with 4 overclocked processors with 8 physical cores (i.e., 16 logical cores) each. We measure time and energy consumption for both the naive (accurate square root and sine) implementation and our approximate implementation, each both sequentially and in a parallel setting using OpenMP where we only use one processor and 8 cores of this processor. Unsound floating-point optimizations such as associativity reorderings have been disabled. The results are summarized in Figure 13. We take our raw input data from the PERFECT data suite [4], which arbitrarily sets the ideal SNR to 140 dB. We show that our approximations cut energy consumption by nearly one half. Our results show that energy gains are mainly due to the speedup obtained by our approximate algorithms. This is fairly understandable since on SandyBridge machines, most floating-point operations consume the same amount of power.

We also performed time measurements on a Intel Haswell platform, shown in Figure 14. However, hardware floating-point counters are disabled on Haswell platforms. So, assuming that floating-point operations consume similar amounts of power (similar to SandyBridge machines), we can deduce energy consumption trends from time. For parallel executions, we show that our approximations cut time consumption by 9% to 18%.

6. Limitations and Future Work

Our VCFloat framework is specialized in real numbers with rounding error terms, so it is targeted to the verification of numerical C programs. However, it may not be totally suitable to the verification of implementations of elementary functions such as CRlibm [33], which may require reasoning about the actual significands and exponents of floating-point numbers.

The main limitation of VCFloat that we encountered during our proof of SAR backprojection is that we have not investigated the interplay between floating-point numbers and integers. In particular, linear interpolation needs to cast a floating-point number to an integer to index an array, which we currently address using a manual proof, after using VCFloat for the purely floating-point part. Conversely, integers would deserve to be more automatically handled by VCFloat as well, so we plan to take advantage of the fact that casting an integer of magnitude less than 2^{24} (resp. 2^{53}) to a single-precision (resp. double-precision) floating-point number does not modify its real-number value.

Also, on the practicality side, although we have drastically reduced the size of floating-point-related proof scripts, our total overall proof of our C implementation of SAR backprojection is still

Norm	Interpol.	Sine/cosine	Final sum	Small		Medium		Large	
				ε	D	ε	D	ε	D
Double	Double	Double	Double then Single	$1.006 \cdot 10^{-3}$	0.5299	$1.011 \cdot 10^{-3}$	2.144	$1.022 \cdot 10^{-3}$	8.761
Double	Single	Double then Single	Double then Single	$1.839 \cdot 10^{-3}$	1.772	$3.696 \cdot 10^{-3}$	28.64	$7.391 \cdot 10^{-3}$	458.3
Adaptive	Double	Double	Double then Single	$5.245 \cdot 10^{-3}$	14.42	$1.050 \cdot 10^{-2}$	230.8	$2.199 \cdot 10^{-2}$	4054
Adaptive	Single	Double then Single	Double then Single	$6.515 \cdot 10^{-3}$	22.25	$1.350 \cdot 10^{-2}$	381.9	$2.782 \cdot 10^{-2}$	6492
Double	Single	Approximate	Double then Single	$9.244 \cdot 10^{-3}$	44.81	$1.750 \cdot 10^{-2}$	641.6	$3.399 \cdot 10^{-2}$	9687
Adaptive	**Single**	**Approximate**	**Double then Single**	$1.320 \cdot 10^{-2}$	91.22	$2.679 \cdot 10^{-2}$	1505	$5.441 \cdot 10^{-2}$	24840
Double	Single	Double then Single	Single	$6.422 \cdot 10^{-2}$	2162	$2.545 \cdot 10^{-1}$	135800	1.012	8585000
Adaptive	Single	Approximate	Single	$7.551 \cdot 10^{-2}$	2988	$2.776 \cdot 10^{-1}$	161600	1.060	9408000

Figure 12: Certified bounds on pixel contribution error (ε) and total image noise (D) for different implementations of SAR backprojection with linear interpolation. **In bold**, our implementation considered in this paper.

Algorithm	Size	Setting	SNR	Time (s)	GFlOp/core	MFlOp/s	Total energy (J)	Total power (W)
Original	Large	Sequential	138.3	1614	1110	688	46800	29
Optimized	Large	Sequential	114.9	841	596	710	24400	29
Original	Large	Parallel	138.3	301	139	743	14300	48
Optimized	Large	Parallel	114.9	137	75	809	7150	52
Original	Medium	Sequential	138.9	190	141	742	5520	29
Optimized	Medium	Sequential	117	92	74	805	2680	29
Original	Medium	Parallel	138.9	31	17	765	1540	50
Optimized	Medium	Parallel	117	15	9	914	742	50

Figure 13: Energy consumption for SAR backprojection implementations on a SandyBridge machine.

Image size	Sequential Original	Parallel Original	Parallel Optimized
Small	13.41	2.29	1.88
Medium	140.06	20.46	17.7
Large	1301.41	183.98	166.2

Figure 14: Time consumption (in seconds) for SAR backprojection implementations on a Haswell machine.

more than 12,000 lines long[7] mainly due to C language constructs.[8] We have not focused on such constructs, since they can be mostly addressed by existing Coq program logics for C such as Verifiable C [3], which our proofs are not using. Thus, to further shorten our proofs, we plan to integrate VCFloat into Verifiable C, combined with more advanced rewriting automation for shared floating-point computations evaluated in separate C variables and reused further down in the program.

Finally, most proof checking time[9] is spent in interval arithmetic computations with Coq-Interval and the validation of their subsequently produced proof terms. Our SAR proof can be considered as one of the first practical "stress tests" for Coq-Interval, which could be useful to detect potential sources of improvement in Coq-Interval to make it scale to realistic intensive interval computations.

7. Conclusion

To the best of our knowledge, our certified C implementation of SAR backprojection is the first example of a realistic C program with floating-point computations proven correct using a unified verification setting based on Coq only, against a specification involving error estimates in real-number values. By virtue of its small trusted computing base containing only the faithfulness of the formal specifications of C and floating-point arithmetic, the soundness of Coq's underlying logic, and the correctness of the Coq proof checker, our work shows that it is possible to gain an unprecedented level of confidence in verified source C programs with floating-point computations. We claim that our work, once combined with Verifiable C [3], will broaden the verifiable features of C towards more complete programs. Overall, our work validates the approach of specifying extensive formal semantics for C including all of its "dark corners" such as floating-point computations for the purpose of source-level verification, thus confirming the validity of the following motto:

"always look on the dark side of C"

Acknowledgments

We thank Xavier Leroy and Jacques-Henri Jourdan for our insightful discussions, and our anonymous reviewers for their many valuable comments and questions. This work is sponsored in part by DARPA MTO as part of the Power Efficiency Revolution for Embedded Computing Technologies (PERFECT) program (issued by DARPA/CMO under Contract No: HR0011-12-C-0123). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressly or implied, of the DARPA or the U.S. Government. Distribution Statement "A" (Approved for Public Release, Distribution Unlimited.) Technologies described in section 5 are patent-pending.

References

[1] IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008*, pages 1–70, Aug 2008. .

[2] A. W. Appel. Verification of a Cryptographic Primitive: SHA-256. *ACM Trans. Program. Lang. Syst.*, 37(2):7:1–7:31, Apr. 2015. ISSN 0164-0925. . URL `http://doi.acm.org/10.1145/2701415`.

[3] A. W. Appel, R. Dockins, A. Hobor, L. Beringer, J. Dodds, G. Stewart, S. Blazy, and X. Leroy. *Program Logics for Certified Compilers*. Cambridge University Press, New York, NY, USA, 2014. ISBN 110704801X, 9781107048010.

[7] 5,000 lines of specification and 7,000 lines of proof, excluding VCFloat, compared to an initial proof without VCFloat of more than 26,000 lines long dealing with floating-point calculi for only one pulse contribution without any connection to the C code

[8] Most reasoning at the level of real numbers can be done separately from the verification of the C implementation, and totals less than 2000 lines.

[9] about 1 hour in total on a 4-core 2.10 GHz Intel Core i7 laptop with 8 Gb RAM, including 20 minutes for the approximate sine alone

[4] K. Barker, T. Benson, D. Campbell, D. Ediger, R. Gioiosa, A. Hoisie, D. Kerbyson, J. Manzano, A. Marquez, L. Song, N. Tallent, and A. Tumeo. *PERFECT (Power Efficiency Revolution For Embedded Computing Technologies) Benchmark Suite Manual*. Pacific Northwest National Laboratory and Georgia Tech Research Institute, December 2013. `http://hpc.pnnl.gov/projects/PERFECT/`.

[5] P. Baudin, F. Bobot, R. Bonichon, L. Correnson, P. Cuoq, Z. Dargaye, J.-C. Filliâtre, P. Hermann, F. Kirchner, M. Lemerre., C. Marché, B. Monate, Y. Moy, A. Pacalet, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C. `http://frama-c.com`, 2007–2015.

[6] L. Beringer, A. Petcher, K. Q. Ye, and A. W. Appel. Verified Correctness and Security of OpenSSL HMAC. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 207–221, Washington, D.C., Aug. 2015. USENIX Association. ISBN 978-1-931971-232. URL `https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/beringer`.

[7] Y. Bertot, P. Castéran, G. Huet, and C. Paulin-Mohring. *Interactive theorem proving and program development : Coq'Art : the calculus of inductive constructions*. Texts in theoretical computer science. Springer, Berlin, New York, 2004. ISBN 978-3-540-20854-9. URL `http://opac.inria.fr/record=b1101046`. Données complémentaires http://coq.inria.fr.

[8] S. Blazy and X. Leroy. Mechanized semantics for the Clight subset of the C language. *Journal of Automated Reasoning*, 43(3):263–288, 2009.

[9] S. Boldo and G. Melquiond. Flocq: A unified library for proving floating-point algorithms in Coq. In *Computer Arithmetic (ARITH), 2011 20th IEEE Symposium on*, pages 243–252, July 2011. .

[10] S. Boldo, J.-C. Filliâtre, and G. Melquiond. Combining Coq and Gappa for certifying floating-point programs. In J. Carette, L. Dixon, C. Sacerdoti Coen, and S. M. Watt, editors, *Intelligent Computer Mathematics*, volume 5625 of *Lecture Notes in Computer Science*, pages 59–74. Springer Berlin Heidelberg, 2009. ISBN 978-3-642-02613-3. . URL `http://dx.doi.org/10.1007/978-3-642-02614-0_10`.

[11] S. Boldo, F. Clément, J.-C. Filliâtre, M. Mayero, G. Melquiond, and P. Weis. Wave equation numerical resolution: A comprehensive mechanical proof of a C program. *Journal of Automated Reasoning*, 50 (4):423–456, 2013.

[12] S. Boldo, J.-H. Jourdan, X. Leroy, and G. Melquiond. Verified compilation of floating-point computations. *Journal of Automated Reasoning*, 54(2):135–163, 2015.

[13] S. P. Boyd and L. Vandenberghe. *Convex Optimization*. Cambridge University Press, Cambridge, 2004.

[14] N. Brisebarre, M. Jolde, E. Martin-Dorel, M. Mayero, J.-M. Muller, I. Paca, L. Rideau, and L. Théry. Rigorous polynomial approximation using Taylor models in Coq. In A. Goodloe and S. Person, editors, *NASA Formal Methods*, volume 7226 of *Lecture Notes in Computer Science*, pages 85–99. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-28890-6.

[15] S. Chevillard, M. Joldeş, and C. Lauter. Sollya: An environment for the development of numerical codes. In K. Fukuda, J. van der Hoeven, M. Joswig, and N. Takayama, editors, *Mathematical Software - ICMS 2010*, volume 6327 of *Lecture Notes in Computer Science*, pages 28–31, Heidelberg, Germany, September 2010. Springer.

[16] T. Coq development team. The Coq proof assistant. `http://coq.inria.fr`, 1984–2015.

[17] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *ACM Symposium on Principles of Programming Languages*, pages 238–252, Jan. 1977.

[18] D. Delmas, E. Goubault, S. Putot, J. Souyris, K. Tekkal, and F. Védrine. Towards an industrial use of FLUCTUAT on safety-critical avionics software. In *Proceedings of the 14th International Workshop on Formal Methods for Industrial Critical Systems*, FMICS '09, pages 53–69, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-04569-1. . URL `http://dx.doi.org/10.1007/978-3-642-04570-7_6`.

[19] M. D. Desai and W. K. Jenkins. Convolution backprojection image reconstruction for spotlight mode synthetic aperture radar. *Image Processing, IEEE Transactions on*, 1(4):505–517, Oct 1992. ISSN 1057-7149. .

[20] C. Ellison and G. Rosu. An executable formal semantics of C with applications. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 533–544, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1083-3. . URL `http://doi.acm.org/10.1145/2103656.2103719`.

[21] E. Goubault and S. Putot. Static analysis of numerical algorithms. In *Proceedings of the 13th International Conference on Static Analysis*, SAS'06, pages 18–34, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 3-540-37756-5, 978-3-540-37756-6. . URL `http://dx.doi.org/10.1007/11823230_3`.

[22] M. Grant and S. Boyd. CVX: Matlab software for disciplined convex programming, version 2.0 beta. `http://cvxr.com/cvx`, Sept. 2013.

[23] N. J. Higham. The accuracy of floating point summation. *SIAM J. Sci. Comput*, 14:783–799, 1993.

[24] J.-H. Jourdan, V. Laporte, S. Blazy, X. Leroy, and D. Pichardie. A formally-verified C static analyzer. In *42nd symposium Principles of Programming Languages*, pages 247–259. ACM Press, 2015.

[25] W. Kahan. Pracniques: Further remarks on reducing truncation errors. *Commun. ACM*, 8(1):40–, Jan. 1965. ISSN 0001-0782. . URL `http://doi.acm.org/10.1145/363707.363723`.

[26] O. Kupriianova and C. Lauter. Metalibm. `http://lipforge.ens-lyon.fr/www/metalibm/`, 2013.

[27] X. Leroy. Compcert. `http://compcert.inria.fr`, 2005–2015.

[28] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[29] G. Melquiond. *De l'arithmétique d'intervalles à la certification de programmes*. PhD thesis, École Normale Supérieure de Lyon, Lyon, France, 2006. URL `http://www.lri.fr/~melquion/doc/06-these.pdf`.

[30] G. Melquiond. Proving bounds on real-valued functions with computations. In A. Armando, P. Baumgartner, and G. Dowek, editors, *Automated Reasoning*, volume 5195 of *Lecture Notes in Computer Science*, pages 2–17. Springer Berlin Heidelberg, 2008. ISBN 978-3-540-71069-1. . URL `http://dx.doi.org/10.1007/978-3-540-71070-7_2`.

[31] G. Melquiond. Coq-interval. `http://coq-interval.gforge.inria.fr/`, 2008–2015.

[32] J.-M. Muller. *Elementary Functions: Algorithms and Implementation*. Birkhäuser, 1997.

[33] J. M. Muller, F. D. Dinechin, et al. CRlibm: Correctly Rounded mathematical library. `http://lipforge.ens-lyon.fr/www/crlibm/`, 2005–2010.

[34] J. Park, P. T. P. Tang, M. Smelyanskiy, D. Kim, and T. Benson. Efficient backprojection-based synthetic aperture radar computation with many-core processors. In *Proceedings of Supercomputing '12*, 2012.

[35] M. Soumekh. *Synthetic aperture radar signal processing*. New York: Wiley, 1999.

[36] P. H. Sterbenz. *Floating-point computation*. Englewood Cliffs ; London : Prentice-Hall, 1973. ISBN 0133224953.

[37] J. F. Sturm. Using SeDuMi 1.02, a MATLAB toolbox for optimization over symmetric cones. *Optimization Methods and Software*, 11–12:625–653, 1999. Version 1.3 available from `http://coral.ie.lehigh.edu/~newsedumi/?page_id=20`.

[38] F. Vedrine, E. Goubault, and S. Putot. FLUCTUAT. `http://www.lix.polytechnique.fr/Labo/Sylvie.Putot/fluctuat.html`, 2001–2015.