VCFloat2: Floating-point error analysis in Coq

Andrew W. Appel[1] and Ariel Kellison[2]

[1] Princeton University
[2] Cornell University **DRAFT June 10, 2022**

Abstract. *[This draft describes a planned α-release of VCFloat 2.0]* VCFloat is a tool for automatic floating-point round-off analysis in Coq. It operates by computational reflection, using a proved-correct-in-Coq program on reified terms. It permits user annotations that improve the error bounds, but which (automatically) introduce extra verification conditions. It recognizes several special cases that tighten the error analysis.

VCFloat was published open-source several years ago by other authors and then abandoned. We have revived it and added several improvements: a new functional-modeling language shallow-embedded in Coq (for better interaction with analyses *other* than round-off error), a reifier for that language, and additional special cases for automatically improved error bounds. Also, VCFloat relies on Coq's Interval library for solving symbolic verification conditions. Before passing those VCs to Interval, they must be simplified. Unfortunately, Coq's standard field_simplify tactic can blow up the goal exponentially and Coq would run out of memory on some largish terms. We have implemented (and proved correct) a high-performance special-purpose simplifier to prepare VCs for Interval solving.

VCFloat is available at github.com/VeriNum/vcfloat.

1 Introduction

Several researchers have demonstrated, in different systems [16, 4, 13, 2] a successful approach to program verification:

1. Write a program (in a low-level language such as C or Haskell); write a functional model (typically in the ML-like calculus of your logic); write a high-level specification of what your program is supposed to accomplish. (These three tasks might be done in a different order!)
2. Prove that the program implements the functional model, using a program logic for your low-level language.
3. Prove that the functional model satisfies the high-level properties, using a general-purpose logic (and ideally, quite independent of any programming-language details);
4. and compose all these proofs together in a single machine-checked logical framework.

When the program implements a *numerical method*—does floating-point or fixed-point computations—then there are typically at least two levels of functional model: some ideal computation in the real numbers, and the corresponding floating-point computation. Now there are *three* parts to the overall correctness proof: prove that the real-valued functional model accurately approximates the high-level goal; prove that the floating-point model accurately approximates the real-valued model; and prove that the low-level program *exactly* implements the float-valued model. These three proofs are sometimes slightly entangled, in ways that we describe elsewhere [15].

We are particularly interested in composing all three of these proofs in the same logic, in the same proof-checking tool, so that there are no gaps in which misunderstandings might compromise soundness.

All these observations are not new, and several authors have published tools for floating-point roundoff analysis, floating-point interval arithmetic, real-number analysis in higher-order logics, and program logics for C and other low-level programming languages. See section 13.

In this paper we are concerned with three such preexisting tools:

Interval [7, §4.2] a Coq package that finds proofs for "interval goals:" each variable is bounded by its bounds, prove that some formula over those variables is bounded. The methods used include (repeated) bisection of intervals, Taylor series, automatic differentiation, and so on.

VCFloat [18], a Coq package that calculates the maximum round-off error of a floating-point formula, compared to the analogous real-valued formula; it reduces these proofs to interval goals, and then calls Interval to solve them.

VST [1], the Verified Software Toolchain for proving correctness of C programs.

VCFloat 1.0 was built in 2015 and described in 2016 by Ramananandro *et al.* [18]. In this paper we report on **VCFloat 2.0**, that is, several improvements to the VCFloat package:

1. VCFloat was previously unmaintained since 2015. It was based on obsolete versions of Coq, of CompCert, and of Coq's floating-point theory (named Flocq). We have brought it up to date and are maintaining it and documenting it.
2. VCFloat 1.0 assumed that the input expressions were C abstract-syntax trees as parsed by CompCert's front end. We think there's a better way than that, to think about functional models: we have added a new front end that can reify directly from functional models written in a general and natural style.
3. To support our natural-style functional models, we added new floating-point literal notations within Coq that work at any floating-point precision (single, double, half, quad, or arbitary user-specified).
4. VCFloat 1.0 recognized that floating-point multiplication by (nonnegative) powers of 2 is exact, with no roundoff error, provided it doesn't overflow. Floating-point division by powers of 2 (or multiplication by powers of 1/2) can also be exact or highly accurate, but the analysis is more complicated because of possible underflow. We have implemented this analysis.

5. VCFloat 1.0 generated interval goals that could blow up: field_simplify would generate an exponentially large term that caused Coq to run out of memory, and without field_simplify the Interval package would generate very loose bounds. We have written an efficient and accurate interval-goal optimizer in Coq's term language, proved correct.
6. Given the style of functional model that we have here developed, we show that with VST one can straightforwardly prove that C programs implement those functional models—with no changes or improvements required to VST. Furthermore, our style of functional model has no dependence on VST; it it quite general and could work with other program logics and proof styles.

2 Review of round-off error analysis

A floating-point number may be 0, *normal finite*, *subnormal finite*, $+\infty$, $-\infty$, or *NaN* (not a number). A *normal* number is representable as $\pm 1.dddddd \cdot 2^e$, where the *dddddd* represents a string of binary digits of length M (the mantissa size) and $e_{\min} \leq e \leq e_{\max}$ is the exponent. A *subnormal* number is representable as $\pm 0.dddddd \cdot 2^{e_{\min}}$, where there may be several leading zeros. A *finite* number is either ± 0, normal, or subnormal.

Every finite float x *exactly* represents some real number $\mathrm{R}(x)$. But the floating-point calculation $x+y$ or $x \cdot y$ is not (usually) exact: often $\mathrm{R}(x)+\mathrm{R}(y) \neq \mathrm{R}(x+y)$. We will often write $\hat{x}$ for $\mathrm{R}(x)$.

Real addition and multiplication are associative, $(\hat{x}+\hat{y})+\hat{z} = \hat{x}+(\hat{y}+\hat{x})$, but floating point operations are not, $(x+y)+z \neq x+(y+z)$ in many cases.

A floating-point mantissa has a fixed number of bits, and the result of a floating point calculation may have more mantissa-bits than fit into the representation, so it will be rounded off. If $x+y$ is a *normal* number we know that $\mathrm{R}(x+y) = (\hat{x}+\hat{y})(1+\delta)$, for some δ such that $|\delta| \leq 2^{-M}$. That is, there is a *relative error bound*. If $x+y$ is *subnormal* then $\mathrm{R}(x+y) = \hat{x}+\hat{y}+\epsilon$, where $|\epsilon| \leq 2^{-E}$, where $E = e_{\max}+M-2$. For single precision $M = 24$ and $e_{\max} = 128$, so $|\delta| \leq 2^{-24}$ and $|\epsilon| \leq 2^{-150}$.

If we know $x+y$ will be finite but don't know whether it will be normal or subnormal, then $\mathrm{R}(x+y) = (\hat{x}+\hat{y})(1+\delta)+\epsilon$ where δ and ϵ are bounded as above. For other operations (subtraction, division) the analysis is similar: $\mathrm{R}(x \oplus y) = (\hat{x} \oplus \hat{y})(1+\delta)+\epsilon$, or without the ϵ or δ (respectively) if the result is known to be normal or subnormal.

There are some special cases: if x and y have the same binary order of magnitude (roughly speaking, $\frac{1}{2} \leq x/y < 2$) then $\mathrm{R}(x+y) = \hat{x}+\hat{y}$ exactly; this is called *Sterbenz subtraction.* If $y = 2^k$ for $0 \leq k$, $\mathrm{R}(x \cdot y) = \hat{x} \cdot \hat{y}$ exactly, so long as it doesn't overflow. If $y = 2^{-k}$, then $\mathrm{R}(x \cdot y) = \hat{x} \cdot \hat{y}$ exactly, so long as the result is normal; or $\mathrm{R}(x \cdot y) = \hat{x} \cdot \hat{y} + \epsilon$ if $x \cdot y$ is subnormal.

3 What VCFloat does

We will use this formula as a running example:

$$x + h \cdot (v + (h/2) \cdot (3 - x))$$

which arises in the simulation of a harmonic oscillator by leapfrog integration with time-step $h = 1/32$.

Assume $2 \leq x \leq 4$ and $-2 \leq v \leq +2$, and we compute this in single-precision floating point.

VCFloat's job is to soundly insert deltas and epsilons as described in Section 2, taking into account of all the special cases (known-normal numbers, known-subnormal numbers, multiplication by powers of two, Sterbenz subtraction, and so on).

We will illustrate the process from the user's point of view, using VCFloat2's new annotation system.

The user writes in Coq,

```
Definition h := (1/32)%F32.
Definition F(x: ftype Tsingle) : ftype Tsingle := Sterbenz(3.0−x)%F32.
Definition step (x v: ftype Tsingle) := (Norm(x + h∗(v+(h/2)∗F(x)))%F32.
```

Factoring out the constant h and function F is entirely discretionary, these could have been written in-line in the step function.

Our functional-modeling language is just Coq formulas and Coq functions over variables of type ftype(T), where T:type is any floating-point precision. (Don't confuse type, meaning a floating-point format, with Type, which is the notion of type in Coq's logic.) Tsingle and Tdouble are predefined, but the user can define any binary IEEE-754 floating-point format (e.g., half-precision or quad precision) and everything will work. So: Tsingle is a floating-point precision description, and the inhabitants of ftype Tsingle are single-precision floats. Formulas may mix different precisions with explicit casting.

The indicator %F32 means that the constants such as 1, 3.0, 38.571e-2 are to be parsed as single-precision (32-bit) floating-point, and the operator + means single-precision addition. We also permit %F64 and the user can easily define, parametrically, constant notations and operators for any desired precision.

VCFloat2 comes with these exciting new functions built in:

```
Definition Norm {A}(x: A) := A.
Definition Denorm {A}(x: A) := A.
Definition Sterbenz {A}(x: A) := A.
```

But wait, you say, these are hardly exciting at all: each one is just the identity function. Actually these are *annotations* for the analysis. Norm(x+y) suggests the sum $x + y$ is going to be a normal (not a subnormal) number. Denorm suggests a subnormal number. Sterbenz(x−y) suggests that x and y will have the same binary order of magnitude. These suggestions will have to be proved—but then they will lead to a better round-off error analysis.

Our running example has already been annotated with Norm and Sterbenz at specific places where it is expected that these conditions will be met.

In general, an error analysis is valid only for a particular range of input values: the assumptions $2 \leq x \leq 4$ and $-2 \leq y \leq +2$ are important. The user encodes these into a *boundsmap* that maps *identifiers* (denoted here as _x and _v) for the free variables of the expression (which are x and v in our running example) to user-specified lower and upper bounds:

```
Definition leapfrog_bmap_list : list varinfo :=
  [ Build_varinfo Tsingle _x 2 4 ; Build_varinfo Tsingle _v (−2) 2 ].
Definition leapfrog_bmap : boundsmap :=
    one line of standard boilerplate mentioning leapfrog_bmap_list.
Definition leapfrog_vmap (x v: ftype Tsingle) : valmap :=
    one line of standard boilerplate mentioning _x, x, _v, v.
```

The round-off error of the step function is the maximum difference, for all x and v in bounds, between the floating-point evaluation of step x v and the evaluation of the same formula on the real numbers. Suppose one wishes to prove that the round-off error will be less than about one in four million: $1/4,000,000$.

To prove this, the user invokes VCFloat2 to calculate an error analysis by first reifying the formula step:

```
Definition step' := (∗ Reification boilerplate ∗)
  ltac:(let e' := HO_reify_float_expr constr:([_x; _v]) step in exact e').

  Lemma prove_roundoff_bound_step:
  ∀x v : ftype Tsingle,
  prove_roundoff_bound leapfrog_bmap (leapfrog_vmap x v) step' (1 / 4000000).
Proof.
intros.
(∗A∗) prove_roundoff_bound.
− prove_rndval.
(∗B∗) all: interval.
− prove_roundoff_bound2.
(∗C∗) optimize_for_interval.
(∗D∗) interval.
Qed.
```

To illustrate what VCFloat2 does, and how it differs from VCFloat1, we will show the proof state at points A, B, C, D.

Point A. By now we have already reified the formula step into a deep-embedded tree step'. We use the same deep-embedded tree language as VCFloat1 (augmented for our new division-by-powers-of-2 operator), but unlike VCFloat1 we build those by reifying from formulas such as step, instead of from C programs; see section 5.

Point B. By this point, VCFloat has calculated several verification conditions. In this case there are 5 of them. Number 3 of 5, for example, arises from the claim that $(x + h \cdot v)$ is a normal number. Here is this verification condition (cleaned up a bit):

$$\frac{\begin{array}{lll} -2 \le \hat{v} \le 2 & 2 \le \hat{x} \le 4 & |\delta_1| \le 2^{-24} \\ |\epsilon_0| \le 2^{-149} & |\epsilon_1| \le 2^{-150} & |\epsilon_3| \le 2^{-149} \end{array}}{2^{-126} \le |\hat{x} + (\frac{1}{32}(v + (\frac{1}{32}(3 - \hat{x}) + \epsilon_1))(1 + \delta_1) + \epsilon_3) + \epsilon_0|}$$

This is a proof that $x + h(v + (h/2)(3 - x))$, after floating-point rounding, is not closer to 0 than 2^{-126}: it is not subnormal. The line all: interval (at Point B in the proof) indicates that all these goals are easily dispatched by the interval tactic [7, §4.2].

Point C. By this point VCFloat has soundly inserted the appropriate deltas and epsilons using the improved analysis of VCFloat2 as described in Section 7, and now one must prove that the resulting difference formula is within the error bound:

$$\frac{\begin{array}{lll} -2 \le \hat{v} \le 2 & 2 \le \hat{x} \le 4 & \\ |\delta_1| \le 2^{-24} & |\delta_2| \le 2^{-24} & \\ |\epsilon_0| \le 2^{-149} & |\epsilon_1| \le 2^{-150} & |\epsilon_3| \le 2^{-149} \end{array}}{\begin{array}{l} |(x + (\frac{1}{32}((v + (1/64(3 - x) + \epsilon_1))(1 + \delta_1) + \epsilon_3) + \epsilon_0))(1 + \delta_2) \\ \quad - \quad (x + \frac{1}{32}(v + (\frac{1}{32}/2)(3 - x)))| \qquad \le \quad \frac{1}{4{,}000{,}000} \end{array}}$$

In general this is a difficult formula to analyze, because of nested epsilons and deltas. If we give this directly to the interval package, it will derive a very weak bound, much worse than the desired one. The usual solution is to use Coq's field_simplify tactic first, but as we explain in Section 9, that can explode into a multinomial with exponentially many terms.

Point D. Instead we use our new special-purpose simplifier, which (efficiently and soundly) transforms the goal below the line into the proof goal at Point D:

$$|\,\delta_2 x + \frac{1}{32}\delta_1 v + \frac{1}{32}\delta_2 v| \ \le \ \frac{1}{4{,}000{,}000} - 4.07453 \cdot 10^{-10}$$

This goal is easily solved by the interval tactic.

4 Floating point types, operations, and notation

VCFloat defines a floating point type as,

```
Record type: Type := {fprec: Z; femax: Z; prec_range: 1 < fprec < femax}.
Definition Tsingle := TYPE 24 128   ltac:(simpl;lia).
Definition Tdouble := TYPE 53 1024  ltac:(simpl;lia).
```

(some details elided). Through the magic of dependent types, this record-type enforces that the precision (number of mantissa bits) must be less than the maximum exponent. We predefine two precisions Tsingle and Tdouble, but the user can easily add more. The `ltac:(simpl;lia)` finds a proof that $1 < 24 < 128$ or $1 < 53 < 1024$, et cetera.

For the underlying semantics of floating-point numbers and operations, we rely on Coq's Flocq floating-point library [6, 7]. Among those operators is, *e.g.,* subtraction:

```
Bminus (prec: Z) (emax: Z): prec>0 → prec<emax → mode →
      binary_float prec emax → binary_float prec emax → binary_float prec emax.
```

which operates on binary IEEE-754 floating-point numbers. The rounding `mode` may be round-to-nearest-even, round-toward-zero, round-down, round-up, etc. The remaining arguments, and the result, are two binary floats of the same precision.

VCFloat encapsulates the precision and emax into a `type`, and provides these wrappers for binary floats and their operations:

```
Definition ftype ty := binary_float (fprec ty) (femax ty).
Definition BINOP op ty :=
     op (fprec ty) (femax ty) (fprec_gt_0 ty) (fprec_lt_emax ty) mode_NE.
Definition BPLUS := BINOP Bplus.
Definition BMINUS := BINOP Bminus.
  (* et cetera *)
```

So therefore, `BPLUS Tsingle` has type `ftype Tsingle → ftype Tsingle → ftype Tsingle`; it is the single-precision floating-point add operator.

Many users would rather write `x+y` than `BPLUS Tsingle x y`, so we provide Coq *notations*:

```
Notation "x + y" := (BPLUS Tsingle x y) (at level 50, left associativity) : float32_scope.
Notation "x + y" := (BPLUS Tdouble x y) (at level 50, left associativity) : float64_scope.
```

so for example `((1/2) * (h * h))%F32` is an expression using `BMULT Tsingle` and with the constants 1 and 2 parsed as single-precision floating-point numbers.

Coq has 64-bit floats built-in, with notation parsers and printers for 1.36e+7, but we want to parse and pretty-print floats in any precision, so we implemented an entire scientific-notation parser and pretty-printer in Coq's Gallina language, and plugged it in using Coq's `Number Notation` mechanism.

Not a Number. Unfortunately, the IEEE-754 standard did not fully standardize the treatment of not-a-number. That is, if x or y is a Nan with a particular payload, then $x + y$ is a Nan whose payload is machine-architecture-dependent. VCFloat treats this accurately with a type-class `Nans` giving the architecture-specific details. Fortunately, all of VCFloat's analysis is *parametric* over `Nans`. The basic reason is that we prove, generally, that Nans do not arise—so it does not matter *which* Nans do not arise. Many of VCFloat's (and Flocq's) functions and theorems have an implicit `Nans` parameter; in the discussion above, we omitted them.

5 Reification

VCFloat's inner workings are explained in Sections 3 and 4 of Ramananandro *et al.* [18]. First the term is reified into a syntactic core language [18, Fig. 1]:

```
Inductive rounded_binop: Type := PLUS | MINUS | MULT | DIV.
Inductive rounding_knowledge: Type := Normal | Denormal.
Inductive binop: Type :=
| Rounded2 (op: rounded_binop) (knowl: option rounding_knowledge)
| SterbenzMinus
| PlusZero (minus: bool) (zero_left: bool).

Inductive rounded_unop: Type := SQRT | InvShift (pow: positive) (ltr: bool).
Inductive exact_unop: Type := Abs | Opp | Shift (pow: N) (ltr: bool).

Inductive unop: Type :=
| Rounded1 (op: rounded_unop) (knowl: option rounding_knowledge)
| Exact1 (o: exact_unop)
| CastTo (ty: type) (knowl: option rounding_knowledge).

Inductive expr: Type :=
| Const (ty: type) (f: ftype ty)
| Var (ty: type) (i: V)
| Binop (b: binop) (e1 e2: expr)
| Unop (u: unop) (e1: expr).
```

The only thing new here is InvShift (see §7).

VCFloat1 translated C statements into expr terms by first parsing the C using CompCert's front end, then translating CompCert Clight ASTs into exprs.

Even though we too are interested in proving the correctness of C programs, we don't take that approach. We reify from a *functional model* (such as the step function shown earlier) written directly in Coq. This is for several reasons:

- The functional model is an important artifact in its own right. It will be the subject of significant analysis, not only for floating-point roundoff but for the function it calculates on the real numbers. We don't *only* want to prove that the C program accurately approximates some real-valued discrete algorithm, we want to prove that the real-valued algorithm accurately approximates the high-level goal, some real-valued function or relation. For that, we want a stable, cleanly written, human-readable functional model, not something automatically reified from a C program.
- The user might not be programming in C.
- Our functional modeling language (and VCFloat) can work at any floating-point precision, but CompCert C only defines 32-bit single precision and 64-bit double precision.

Our reifier is a program written in Coq's Ltac tactic-programming language. It is almost entirely conventional, so we won't show it here. But we remark on a few clauses:

```
Ltac reify_float_expr E :=
 match E with
 | BMINUS _ ?a ?b ⇒ let a' := reify_float_expr a in let b' := reify_float_expr b
                     in constr:(@Binop ident (Rounded2 MINUS None) a' b')
 | Norm (BMINUS _ ?a ?b) ⇒ let a' := reify_float_expr a in let b' := reify_float_expr b
                     in constr:(@Binop ident (Rounded2 MINUS (Some Normal)) a' b')
 | Denorm (BMINUS _ ?a ?b) ⇒ let a' := reify_float_expr a in let b' := reify_float_expr b
                     in constr:(@Binop ident (Rounded2 MINUS (Some Denormal)) a' b')
 | Sterbenz (BMINUS _ ?a ?b) ⇒ let a' := reify_float_expr a in let b' := reify_float_expr b
                     in constr:(@Binop ident SterbenzMinus a' b')
 | . . .
```

These four clauses reify differently annotated subtractions. Since Norm, Denorm, and Sterbenz are all identity functions, a program in Gallina (Coq logic's core calculus) could not distinguish them. But the tactic language can. In the Binop tree-node that it builds, there is different "rounding knowledge" encoded into the syntax tree.

6 The core of VCFloat

VCFloat's core algorithm is called rndval_with_cond: "compute rounded value with verification conditions." We have not modified it except to add analysis for division by powers of two. In addition, we repackaged into a more user-friendly form, wrapping it with appropriate corollaries and adding proof-automation tactics to help discharge the verification conditions.

rndval_with_cond: expr → rexpr ∗ shiftmap ∗ list (environ → Prop).

Suppose the expression e is $(x + \frac{1}{32}v + \frac{1}{2}\frac{1}{32}\frac{1}{32}(3 - x))$, reified into the expr syntax. Then rndval_with_cond(e) is (r, m, vcs), where r is a (reified) expression containing epsilons and deltas indexed by natural numbers, such as appears in the left-hand-side below the line at Point C (although there it appears in its reflected, not reified, form).

The result m is a map from those natural numbers to bounds-descriptors, sufficient to describe the bounds for δ_i and ϵ_j above the line at Point C. The result vcs is a list of verification conditions, such as the one that appears (reflected, above the line) at Point B.

VCFloat's core soundness theorem, rndval_with_cond_correct, is a machine-checked proof in Coq. It is presented as Theorem 3 by Ramananandro *et al.* [18]. Basically, it says that

- for any environment ρ mapping reified variables (such as _x and _v in our example) to floating-point numbers,
- if each of the verifications vcs is true in ρ,
- then there exists an error-map σ from $\delta\epsilon$ indexes (natural numbers) to $\mathbb{R}$,
- such that every δ and ϵ (interpreted in σ) respects the bound in m,
- and the floating-point evaluation (in ρ) of e is finite (not an infinity or NaN),

– and the real-number interpretation of r (using ρ and σ) is exactly equal to the floating-point evaluation of e (in ρ).

Our prove_roundoff_bound tactic (at Point A) applies this rndval_with_cond_correct as part of an end-to-end machine-checked proof in Coq about the floating-point round-off error of the given formula.

7 Optimizations and Inverse shifts

VCFloat1 included some transformations on (reified) terms that improve the analysis. For example, multiplication by a power of 2 is exact in floating-point arithmetic, if the result stays finite. VCFloat's reified trees represent $64.0 \cdot x$ as Binop (Rounded2 MULT) 64 x, and represent $2^6 \cdot x$ as Unop (Exact1 (Shift 6)) x. Both of these "reflect" back to the same floating-point formula, but they are treated differently in the analysis: MULT introduces δ and ϵ, but Shift does not.

VCFloat1 included other such optimizing transformations[3] such as constant-folding. All these transformations came with proofs of correctness: that is, the (reflected) optimized program computes the same as the original.

VCFloat1 did not have a well-defined procedure for incorporating such optimizations (and their proofs) into the rndval_with_cond analysis. We do that as part of the prove_roundoff_bound tactic executed at Point A.

InvShift. We also implemented another optimizing transformation: divide by powers of 2 (or multiplication by powers of 1/2).

When $x \cdot 2^{-k}$ is a normal number, then $\mathrm{R}(x \cdot 2^{-k}) = \mathrm{R}(x) \cdot 2^{-k}$ exactly. When $x \cdot 2^{-k}$ is subnormal, then $\mathrm{R}(x \cdot 2^{-k}) = \mathrm{R}(x) \cdot 2^{-k} + \epsilon$, for $|\epsilon| \leq 2 \cdot 2^{-E}$.

We have added to VCFloat's reified tree language an InvShift operator. As an example, our optimizer replaces Binop (Rounded2 DIV) x 64 with Unop (Rounded1 (InvShift 6)) x so that rndval_with_cond introduces the appropriate ϵ with its bound.[4]

8 Coq Interval library

The Interval package [7, §4.2] is a procedure in Coq for proving goals of the form,

$$\frac{\begin{array}{ccccc} l_1 & \leq & x_1 & \leq & h_1 \\ l_2 & \leq & x_2 & \leq & h_2 \\ \vdots & & \vdots & & \vdots \\ l_n & \leq & x_n & \leq & h_n \end{array}}{l_0 \leq E \leq h_0}$$

[3] Again, the point is to optimize the analysis, not optimize the program that runs. The choice of what program to actually run is specified by the user, in writing the functional model.

[4] It is worth reexamining the constraint, $|\epsilon| \leq 2 \cdot 2^{-E}$. Perhaps we can prove that $|\epsilon| \leq 2^{-E}$, which would be slightly better.

where all the l_i and h_i are constants, the x_i are real-valued variables, and E is a real-valued expression over the variables. Any of the inequalities may be strict ($<$) rather than non-strict ($\leq$), some of the inequalities may be missing, there may be several redundant constraints over any given x_i, and any of the inequalities may be expressed as $|x_i| \leq h_i$. The l_0 and h_0 may be left unspecified, in which case Interval reports the best bounds that it can prove.

Interval uses floating-point interval arithmetic, being careful with floating-point rounding modes (round down on one side, up on the other). But that alone would provide very weak bounds, so Interval also uses bisection of intervals, Taylor series, and automatic differentiation.

At Point C, just after prove_roundoff_bound2, the proof goal is in the form accepted by the Interval package. Unfortunately, Interval doesn't do a very good job on that goal. The method of Taylor expansions would probably work quite well, but Interval uses only univariate Taylor series, whereas we have two variables x and v. A multivariate-Taylor package such as FPTaylor [19] would compute a good bound for this goal, but FPTaylor does not connect to Coq as smoothly as does Interval, which (like VCFloat) is implemented entirely within Coq.

The Interval mode that can work on our problem is (repeated) bisection of the interval. But in practice, all those nested expressions with deltas and epsilons are obstacles to good approximations. In particular, at Point C there is the formula $(x + \ldots)(1 + \delta_2) - (x + \ldots)$, and subtracting $x - x$ is a disaster for interval arithmetic.

We found that it helps to use Coq's field_simplify tactic before calling Interval; this would turn the (Point C) goal $l_0 \leq E \leq h_0$ into,

$$\begin{aligned}|\,(&-x\delta_1\delta_2 - x\delta_1 + 2047x\delta_2 + 64v\delta_1\delta_2 + 64v\delta_1 + 64v\delta_2\\ &+ 64\epsilon_1\delta_1\delta_2 + 64\epsilon_1\delta_1 + 64\epsilon_1\delta_2 + 64\epsilon_1 + 3\delta_1\delta_2 + 3\delta_1\\ &+ 64\epsilon_3\delta_2 + 64\epsilon_3 + 2048\epsilon_0\delta_2 + 2048\epsilon_0 + 3\delta_2)/2048| \qquad \leq \qquad \tfrac{1}{4{,}000{,}000}\end{aligned}$$

The two terms x and $-x$ have been *symbolically* canceled. With this goal, Interval computes an excellent bound.

But clearly, repeatedly applying the distributive law to this multinomial has caused an exponential blow-up in the number of terms. For the small expression in our running example, "exponential" means 17 terms, which is not such a problem. But when we apply VCFloat to more substantial examples, Coq runs out of memory.

9 Simplifying Interval Goals

As mentioned in Section 8, VCFloat could make good use of a package in Coq that uses multivariate Taylor expansions (or perhaps multivariate automatic differentiation). Here we describe an alternate approach, which has worked well for ODEs that can be described as multinomials. We expand the multinomial into sum-of-products form, then soundly and efficiently cancel terms while reducing the exponential blow-up in the number of terms.

A real-life numerical analyst might perhaps discard the higher-order terms, those in which more than one δ or ϵ are multiplied together. Another real-life alternative is to ignore the issue of underflow (denormalized numbers), in which case all the ϵ are discarded. Both of those alternatives work well most of the time. But neither one is *sound;* and we want proofs of our bounds!

Therefore, we implemented (and proved correct in Coq) an algorithm to efficiently and soundly simplify Interval goals:

Step 1: Apply *limited ring simplification*: the distributive law (but not the associative law[5]), and multiplication by 1 and by 0, division by 1, multiplication of constants together, limited simplification of rational constants.
Step 2: Discard *and bound the total of* insignificant terms.
Step 3: Cancel terms using an efficent balanced-binary-tree data structure.

The entire algorithm is implemented in Gallina (Coq logic's functional programming language), which Coq can compile directly to byte-code or machine-code for execution. Computing in Gallina is *much* more efficient than manipulating proof goals in "native" Coq; the latter builds lengthy proof terms that record the history of manipulations, the former is proved correct once and for all.

A program in Gallina must be applied to a *reified* term, not to a "native" proof goal. For this component, we chose to use the reified-tree syntax from the Interval package, rather than VCFloat's own. This is because (1) our interval-goal simplifier can be used by *any* user of the Interval package, not only in connection with VCFloat; and (2) we have no need of the "rounding knowledge" and other special features of VCFloat's tree syntax. Not only do we use Interval's tree language, we use its reifier tactic as well.

Step 1 of the algorithm doesn't need much explanation.

Step 2, soundly discarding insignificant terms, works as follows. One might think, "let's discard any higher-order term, i.e., containing the product of two or more δ and ϵ." But some of those terms might be multiplied by very large coefficients or user-variables; and on the other hand, some terms containing only a single δ or ϵ might be multiplied by tiny numbers and therefore be insignificant.

We will take advantage of the fact that *bounds are known for every variable*, both the original variables (x, v) and the δ and ϵ variables. So in a sum-of-products expression (resulting from limited ring simplification), we can bound every term. (Terms containing functions that we cannot bound in closed form, we handle as described below.)

The user supplies a *cutoff* such as 2^{-30} or 2^{-60} or whatever is appropriate. We preprocess all of the (already reified) bounds hypotheses (for variables $x, v, \delta_1, \epsilon_2$, etc.) into a single absolute-value bound for each variable: $|x_i| \leq h_i$.

Then for each term $x_i x_j x_k$ we can look up the (reified) bound hypotheses to find a bound $h_i h_j h_k$ on the absolute value of the term. We accumulate all those

[5] We avoid the associative law because we don't need it—the next parts of the algorithm can handle unflattened trees of multiplications or of additions—and we want to avoid any unnecessary rewriting of large formulas.

absolute values together to produce the transformed goal,

$$\begin{aligned}&|x + x\delta_2 + \tfrac{1}{32}v + \tfrac{1}{32}v\delta_1 + \tfrac{3}{2048} - \tfrac{1}{2048}x + \tfrac{1}{32}v\delta_2 \\ &\quad - \quad (x + \tfrac{1}{32}v + \tfrac{3}{2048} - \tfrac{1}{2048}x)| \quad \leq \quad \tfrac{1}{4{,}000{,}000} - 4.0745386 \cdot 10^{-10}.\end{aligned}$$

Here, the term $4.0745386 \cdot 10^{-10}$ is the sum of bounds of the deleted terms: this transformation is sound, and proved sound, in the sense that this goal implies the original goal.

The algorithm for deleting insignificant terms might encounter some terms for which the bounds are not known, or that it cannot analyze because they are not simply products of variables and constants. Such *"residual"* terms it leaves unchanged.

At this point one notices that some terms cancel. The field_simplify tactic could cancel these terms, but we want to do it more efficiently—and we want to do it in the already-reified trees, without first reflecting back into Coq (as would be necessary if we used field_simplify).

Step 3 cancels terms, efficiently. That is, we have a tree of additions of terms. Each term is *either* a product of constants and variables, *or* contains other operators; in the latter case we leave that term alone (as a *residual term*) and don't attempt to cancel it. An example of a (potentially) cancelable term is, $c_1x_1\delta_1\epsilon_2x_1c_2x_2$, where c_1, c_2 are rational constants, and $x_1, x_2, \delta_1, \epsilon_2$ are variables.

In our reified tree syntax, all variables are represented by natural numbers. So we can represent any product of (nonnegative integer) powers of these variables $x_0^{k_0}x_1^{k_1}x_2^{k_2}$ by a list of natural numbers $[k_0, k_1, k_2]$. The product of all the constants can be represented as a canonical-form rational number. For efficiency, we factor out all the powers of 2 from the numerator and denominator into a separate factor 2^e, where e may be positive, negative, or zero.

That is, the canonical form of a (nonresidual) term is, $\overrightarrow{k} \cdot (\pm n)/d \cdot 2^e$, where where $\overrightarrow{k}$ is a list of natural numbers representing the polynomial $x_0^{k_0}x_1^{k_1}x_2^{k_2}\ldots$, n is an odd integer (or zero), d is an odd positive number, $\gcd(n, d) = 1$, and e is an integer.

We maintain a balanced binary search tree indexed by keys $\overrightarrow{k}$. At each key $\overrightarrow{k}$ we have a list of coefficients, each of the form $(\pm n)/d \cdot 2^e$. In walking the expression-tree, whenever we find $\overrightarrow{k} \cdot (\pm n)/d \cdot 2^e$ we look up k, and traverse the list: if we find the *negation* of $(\pm n)/d \cdot 2^e$ we delete it from the list, otherwise we cons $(\pm n)/d \cdot 2^e$ to the front of the list; then reinsert at key k. Meanwhile, we replace that term in the expression-tree with 0.

What remains is:

– an expression-tree with residual terms that could not be represented in canonical form, and zeros where terms have been removed from the tree and added to the key-value map;
– a key-value map: for each key $k_0k_1 \ldots k_n$ a list of coeefficients each of the form $\frac{n}{d} \cdot 2^e$.

After the key-value map is built, for each k mapped to a list of coefficients, we add all the coefficients together, as follows: we normalize all the elements to have the same exponent e (so that it is no longer true that every n and d is odd), then add all the rational numbers, to collapse the list into a single coefficient.

Then we convert each key-value pair back into an expression $x_0^{k_0} x_1^{k_1} x_2^{k_2} \ldots x_n^{k_n} (\pm n)/d \cdot 2^e$, and build a final expression tree with our their sum added to the residual terms. The result is shown at Point D.

Generally speaking, Step 1 of the algorithm (distributive law) takes exponential time. Step 2 takes linear time and space (in the exponentially sized term produced by step 1). Step 3 takes $N \log N$ time and linear space (and tends to reduce the term back to small size).[6]

We tested this algorithm on a large expression that resulted from calculating position-change and momentum-change of a harmonic oscillator and then taking the sum of squares of position and momentum.

– The original expression-tree (Point A) had 242 nodes (constants, variables, operators).
– After step 1 (distributive law) there were 612,284 nodes, or 31,759 multinomial terms.
– After step 2 (delete insignificant terms) there were 1456 nodes, 244 terms.
– After step 3 (cancel) there were 219 nodes, 24 terms.

On this large expression, the entire algorithm runs in Coq in 2 to 5 seconds.[7]

A more efficient algorithm. It should be possible to cancel terms *interleaved* with the distributive law, so that the exponential blow-up in step 1 never occurs. To do so, one would first walk over the tree bounding the absolute value of every subexpression (using the bounds hypotheses). A second pass would walk the tree, such that in the context $A * B$, while processing B one would adjust the cutoff by the bound for A. We may implement this algorithm in the future. For now, the algorithm we have seems fast enough.

Floating-point interval arithmetic. In this section we have described analyses on real-valued formulas that contain integer and floating-point constants. Since one cannot efficiently compute on the real numbers, we perform our analyses in floating-point. But we want all our analyses to be sound even in the presence of round-off error.

There is a well-understood solution to this problem: compute in floating-point interval arithmetic, using operators that carefully round down and round

[6] However, since variables (x, δ, ϵ) are represented by natural numbers with unary representation, all these numbers must be multiplied by the number of variables. It would be possible to reduce this to a factor logarithmic in the number of variables by using a binary representation.

[7] 1.55 to 5.7 seconds using `vm_compute` on an Intel Core i7 laptop in Coq 8.15.0. Would probably be faster using `native_compute`. The variation in times may be explained by whether a major-generation garbage collection occurs. Maybe.

up, respectively, for the bottom and top of the interval. Coq's Interval package already does this for its own (proved correct) analyses. We use Interval's library of floating-point interval arithmetic for the algorithms described in section 9.

10 Soundness theorem for simplification

We use the Interval package's reify function to turn the user's functional model into a tree-term e of type expr. As usual in Coq, reify is written as a tactic program in the **Ltac** language, and we validate each reification by reflecting.

Then, once a (small, floating-point) cutoff value is chosen, we can apply the following function:

```
Definition simplify_and_prune hyps e cutoff :=
(* Step 1 *) let e1 := ring_simp false 100 e in
(* Step 2 *) let '(e2,slop) := prune (map b_hyps hyps) e1 cutoff in
(* Step 3 *) let e3 := cancel_terms e2 in
                 (e3, slop).
```

The function simplify_and_prune embodies the three-part algorithm described in Section 9. It is proved correct in Coq.

Before stating the correctness theorem, we must define the notion of equivalently evaluating expressions:

```
Inductive expr := ... (* from the Interval package *)
Definition environment := list ℝ. (* map variables, represented as ℕ, to values *)
Definition eval : expr → environment → ℝ := ... (* from the Interval package *)
Definition expr_equiv (a b: expr) : Prop := ∀env, eval a env = eval b env.
Infix "=="    :=    expr_equiv (at level 70, no associativity).
```

The correctness theorems for ring_simp and cancel_terms is that they exactly preserve evaluation, in any environment. (The enable parameter controls whether the associative law is to be used.)

```
Lemma ring_simp_correct: ∀enable n e, ring_simp enable n e == e.

Lemma cancel_terms_correct: ∀e, cancel_terms e == e.
```

The specification of prune is a bit more complicated, and therefore so is the spec of simplify_and_prune:

```
Lemma simplify_and_prune_correct:
 ∀hyps e cutoff e₁ s,
   simplify_and_prune hyps e cutoff = (e₁, s) →
   F.real s = true →
   ∀(vars: list ℝ) (r : ℝ),
     length hyps = length vars →
     eval_hyps hyps vars (Rabs (eval e₁ vars) ≤ r − R(s)) →
       eval_hyps hyps vars (Rabs (eval e vars) ≤ r).
```

It says, suppose you wish to prove that, with bounds-hypotheses hyps and variables vars that satisfy those hypothesis, that $|e| \leq r$. And suppose that simplify_and_prune gives you a simplified expression e_1, and claims that the total of all deleted terms is bounded by s. Then indeed, it will suffice to prove that $|e_1| \leq r - s$.

If you specify a cutoff of 10^{-8} for example, and there are $\leq 10^{10}$ terms to delete, then it is *inconceivable* that s will overflow, since 10^{10-8} is representable in double-precision floating-point. But just in case, the hypothesis $\mathsf{F.real}\ s = \mathsf{true}$ tests for overflow in s.

11 Proving the C program implements the model

The Verified Software Toolchain is a Coq library for proving the correctness of C programs with respect to specifications written in Coq's logic. Its specification language is higher-order separation logic with propositions that can use all of Coq's logic. Therefore its specification language and proof system is much more expressive than such systems as Dafny, Verifast, Frama-C. Furthermore, because it is embedded in Coq, one can compose, *all in Coq*, a VST proof that a C program correctly implements a functional model, with a Coq proof that a functional model correctly implements some high-level specification.

And it happens that VST includes a full treatment of C language floating point, using the Flocq model of the IEEE-754 standard. In VST one can (fairly easily) prove that a C program implements a low-level functional model of a floating-point computation. But VST provides no help in reasoning *about* the functional model. That is what VCFloat can do, with our improvements.

In particular, proving that a C function correctly implements a formula such as our step example is very easy, and fully automatic. This is described (for a different example) by Appel and Bertot [3].

12 Open Source

VCFloat1 was developed in 2015 by Tahina Ramananandro, Paul Mountcastle, Benoît Meister, and Richard Lethin, all researchers at Reservoir Labs, funded by a research contract from DARPA, the Defense Advanced Research Projects Agency. Starting in 2005, DARPA encouraged (and sometimes required) the products of its sponsored research to be open-sourced[8], and so VCFloat is open-source. Between 2015 and 2021 the github repository sat unmaintained and untouched.

In 2021, when we were looking for Coq tools to reason about Flocq floating point, a Google Scholar search led us to the 2016 paper [18], which led us to (and explained) the Reservoir github repository. Reservoir Labs, which was recently acquired by Qualcomm and is now a branch of Qualcomm research, does not

[8] ACERT program, Jonathan M. Smith, program manager, https://www.federalgrants.com/Adaptive-Cognition-Enhanced-Radio-Teams-ACERT-1286.html

intend to further maintain this product. We are now the keepers and maintainers of VCFloat, with the blessing of its original authors.[9]

This can be counted as a success of DARPA's open-source policy, a success of Reservoir's cooperation with the letter and spirit of that policy, and a success of the system of academic publication that led to the 2016 paper, without which we would never have found or understood the github repo.

13 Related work

In the table below we list some related tools. Some of these are proved correct *and* embedded within the language and logic of a proof assistant. We are particularly interested in such tools because they are easier to connect, both on their input side and on their output side, to larger verified analyses. Some tools are not verified but can *validate*—generate proof witness checkable by a proof assistant. Others are neither verified nor validating.

Tool	Year	**Purpose**	**Impl.**	Ver/Val?
Harrison [14]	1999	Specification of floating point	HOL	Verified
Flocq [12, 6]	2001-11	Specification of floating point	Coq	Verified
Interval [11]	2005	Symbolic interval analysis	Coq	Verified
Gappa [5]	2009	Floating-point round-off analysis	C++	Validating
VCFloat [18]	2016	Floating-point round-off analysis	Coq	Verified
FPBench [9]	2016	Benchmark suite	n/a	
PRECiSA [17]	2017	Floating-point round-off analysis	Haskell,C	Validating
FPTaylor [19]	2018	Floating-point round-off analysis	OCaml	Validating
Satire [10]	2020	Floating-point round-off analysis	Python	no
various [8]		Precision tuning		seems no

We are particularly interested in tools that are fully integrated into a proof assistant for a higher-order logic. As part of an integrated error analysis and correctness verification of a numerical program, floating-point round-off is only one component. We not only want to *export* proofs from the tool into the proof assistant, but we need to *import* problem specifications from (the language of) the proof assistant into the tool. VCFloat serves very well because it is fully integrated into Coq.

14 Future work

VCFloat could be further improved:

- Improving the Interval package to handle multivariate Taylor analysis would give VCFloat power comparable to FPTaylor [19].
- Allowing input varibles to come with their own error bounds (instead of being assumed exact) would give VCFloat modularity comparable to Satire [10].
- It might be possible to tighten the bound on InvShift; see footnote 4.

[9] Not that we needed their blessing, it's open-source after all!

15 Conclusion

Floating-point round-off analysis should be done as part of a larger numerical analysis that treats algorithm correctness, discretization analysis, and low-level program correctness, *all in the same general-purpose logic* and with end-to-end composable, machine-checked proofs. In such a context, it's important to have a language for writing floating-point functional models that clearly and simply relate to real-valued functional models. The individual analysis tools, of which round-off error is just one example, should be able to take their inputs and their outputs in the same logic as the rest of the framework.

VCFloat2, as we have adapted and extended it from VCFloat1, is quite useful in a toolchain for such analyses. Below it, the VST tool reasons about C program correctness and connects to functional models of the kind we have described here. Above VCFloat, tools for reasoning *with proofs in Coq* about real-valued functional models are mostly yet to be developed, and are an exciting area for future research.

References

1. Andrew W. Appel. Verified software toolchain. In Gilles Barthe, editor, *ESOP'11: European Symposium on Programming*, volume 6602 of *LNCS*, pages 1–17. Springer, 2011.
2. Andrew W. Appel. Modular verification for computer security. In *CSF 2016: 29th IEEE Computer Security Foundations Symposium*, pages 1–8, June 2016.
3. Andrew W. Appel and Yves Bertot. C-language floating-point proofs layered with VST and Flocq. *Journal of Formalized Reasoning*, 13(1):1–16, December 2020.
4. Lennart Beringer, Adam Petcher, Katherine Q. Ye, and Andrew W. Appel. Verified correctness and security of OpenSSL HMAC. In *24th USENIX Security Symposium*, pages 207–221. USENIX Assocation, August 2015.
5. Sylvie Boldo, Jean-Christophe Filliâtre, and Guillaume Melquiond. Combining Coq and Gappa for certifying floating-point programs. In *International Conference on Intelligent Computer Mathematics*, pages 59–74. Springer, 2009.
6. Sylvie Boldo and Guillaume Melquiond. Flocq: A unified library for proving floating-point algorithms in coq. In *2011 IEEE 20th Symposium on Computer Arithmetic*, pages 243–252. IEEE, 2011.
7. Sylvie Boldo and Guillaume Melquiond. *Computer Arithmetic and Formal Proofs: Verifying Floating-point Algorithms with the Coq System.* Elsevier, 2017.
8. Wei-Fan Chiang, Mark Baranowski, Ian Briggs, Alexey Solovyev, Ganesh Gopalakrishnan, and Zvonimir Rakamarić. Rigorous floating-point mixed-precision tuning. In *POPL'17: 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, page 300–315, New York, NY, USA, 2017. Association for Computing Machinery.
9. Nasrine Damouche, Matthieu Martel, Pavel Panchekha, Jason Qiu, Alex Sanchez-Stern, and Zachary Tatlock. Toward a standard benchmark format and suite for floating-point analysis. In *Numerical Software Verification (NSV'16)*, July 2016.
10. Arnab Das, Ian Briggs, Ganesh Gopalakrishnan, Sriram Krishnamoorthy, and Pavel Panchekha. Scalable yet rigorous floating-point error analysis. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2020.

11. M. Daumas, G. Melquiond, and C. Munoz. Guaranteed proofs using interval arithmetic. In *17th IEEE Symposium on Computer Arithmetic (ARITH'05)*, pages 188–195, 2005.
12. Marc Daumas, Laurence Rideau, , and Laurent Théry. A generic library of floating-point numbers and its application to exact computing. In *TPHOLS'01: 14th International Conference on Theorem Proving in Higher Order Logics*, page 169–184, 2001.
13. Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. Deep specifications and certified abstraction layers. In Sriram K. Rajamani and David Walker, editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015*, pages 595–608. ACM, 2015.
14. John Harrison. A machine-checked theory of floating point arithmetic. In *Theorem Proving in Higher Order Logics: 12th International Conference, TPHOLs'99*, volume 1690 of *LNCS*, pages 113–130. Springer-Verlag, 1999.
15. Ariel Kellison and Andrew W. Appel. That other paper, 2022. in preparation.
16. Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, pages 207–220. ACM, 2009.
17. Mariano M. Moscato, Laura Titolo, Aaron Dutle, and César A. Muñoz. Automatic estimation of verified floating-point round-off errors via static analysis. In *Computer Safety, Reliability, and Security - 36th International Conference, SAFECOMP'17*, pages 213–229, 2017.
18. Tahina Ramananandro, Paul Mountcastle, Benoît Meister, and Richard Lethin. A unified Coq framework for verifying C programs with floating-point computations. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs*, CPP 2016, page 15–26, New York, NY, USA, 2016. Association for Computing Machinery.
19. Alexey Solovyev, Marek S Baranowski, Ian Briggs, Charles Jacobsen, Zvonimir Rakamarić, and Ganesh Gopalakrishnan. Rigorous estimation of floating-point round-off errors with symbolic Taylor expansions. *ACM Transactions on Programming Languages and Systems*, 41(1):1–39, 2018.