Generating formally certified bounds on values and round-off errors

Marc Daumas [a,1] Guillaume Melquiond [a]

[a] *LIP Computer Science Laboratory*
UMR 5668 CNRS–ENS Lyon–INRIA
Lyon, France

Abstract

We present a new tool that generates bounds on the values and the round-off errors of programs using floating point operations. The tool is based on forward error analysis and interval arithmetic. The novelty of our tool is that it produces a formal proof of the bounds that can be checked independently using an automatic proof checker such as Coq and a complete model of floating point arithmetic. For the first time ever, we can easily certify that simple numerical programs such as the ones usually found in real time applications do not overflow and that round-off errors are contained. Such level of quality should be compulsory on safety critical applications. As our tool is easy to handle, it could be used for many pieces of software.

Key words: Round-off error, Overflow, Formal proof, Certification, Safety critical.

1 Introduction

Deadly and disastrous failures [14,19,9] have shown that common professional practices do not guarantee quality of the produced software. Pen-and-paper hand-written formal proofs are not sufficient as errors are commonly printed [17,28] even in highly regarded publications by honored scientists in well reviewed and cited journals [26,10]. Results produced by interval libraries, including for example [23,15], are also not sufficient for safety critical applications. Some hand-written pen-and-paper proofs contain errors and there is no doubt that some highly regarded hand-written programs also contain errors

[1] This material is based on work partially supported by National Aeronautics and Space Administration, Langley Research Center under the Research Cooperative Agreement No. NCC-1-02043 awarded to the National Institute of Aerospace.

even if they have been well reviewed and produced by honored scientists. In both cases, deriving the conclusions is not sufficient and the process should be certified by an automatic proof checker.

Such certifications by automatic proof checkers have already been used to detect or prevent errors in algorithms and implementations [1]. Few results however are available for computer arithmetic [20,2] and even fewer results consider the effect of round-off errors in floating point arithmetic. Our contacts with leading teams at Intel and NASA LaRC allow us to safely believe that at the time we are reporting on this work, all developments that follow round-off errors are linked to one of the authors of this report [18,4]. As certifying formal proofs by hand can be extremely tedious and needs specialized training, only carefully chosen applications have been developed so far.

The purpose of our work is to safely and automatically qualify the behavior of algorithms concerning the range of all the floating point variables and all the round-off errors. Given a description of the application, our tool generates a formal proof of these arithmetic properties. As this work was intended for safety critical applications, the formal proof is such that it can then be automatically verified by a proof checker such as Coq [13] in order to ensure the application will not fail due to an unexpected numerical behavior.

There might be many reasons for a computing system to fail. The first one might be that some hardware, the operating system or the compiler is not working correctly. Another one might be that the user did not handle the software correctly or that the designer did not understand exactly the intended behavior of the system. Yet another one might be that one programmed structure is not initialized or used correctly, and so on. Teams around the world are developing tools and methodologies to prevent all these problems. They are not new to floating point arithmetic and our work does not target them.

Failures may also be a consequence of substituting a mathematical term by a finite number of operations when implementing it. For example when we truncate an infinite series, or when we replace the mathematical solution of an equation by a constant number of steps of Newton-Raphson iterations. Our tool is not intended to deal with these problems either. Its purpose is to bound the gap between the behavior of the given program executed on real numbers, with no round-off error, and the behavior of the same program executed on a computer, with round-off errors.

The article will first present the formal model used in our tool. Interval arithmetic is used to guarantee the range of the variables of an application, and round-off errors are dealt with forward error analysis methods. This model is adapted to the generation of formal proofs that can be verified by automatic proof checkers. This approach is then applied to our first application that is used to maintain safety distances between aircrafts.

2 Setting up a formal model adapted to automatic proof checkers

In the following, $x : \mathbb{R}$ denotes a value x known with an infinite precision. This is a theoretical number and an algorithm uses an approximated representable value $\tilde{x} : \mathbb{F}$, with $\mathbb{F}$ being the set of hardware-representable numbers. We do not mean that $\tilde{x}$ is the rounded value of x, we mean that $\tilde{x}$ is the value that is used and possibly computed by the program in place of x.

A closed mathematical function $f(x)$ is implemented by a program $\tilde{f}(\tilde{x})$ that takes into account the round-off errors caused by the computation. Discretization and truncation errors have to be handled separately. We use $\tilde{x}$ instead of x to recall the reader that a program always uses only representable inputs.

The tool has been designed to automatically generate a proof of the range of $\tilde{f}(\tilde{x})$, the difference between the exact value $f(x)$ and the computed value $\tilde{f}(\tilde{x})$, and so on, when some properties are known on the input variables. For example, under the hypothesis $\tilde{x} \in [1, 4]$, it is easy to prove that $\sqrt{\tilde{x}}$ lies in the range $[1, 2]$. The tool is here to help certifying proofs on complete applications.

The analysis is done on a low-level description of the application on which arithmetic properties have to be proved. When two variables are summed, our tools does not simply do an addition, it generates proofs about the implemented addition that a computer will effectively execute, for example a floating-point addition on simple precision numbers with respect to the IEEE-754 standard and its default rounding mode as defined in our formal specification. The description of the program must carry enough information in order for the proofs to relate to the real program as implemented and executed.

The tool also takes into account some properties (range, absolute error, relative error) on the input variables. Given the same kind of properties on the output variables, the tool tries to generate a script of proof that states and verifies these properties from the hypotheses on the input variables. Section 2.1 describes one of the mechanisms involved in proving range properties. Section 2.2 is devoted to the proofs of errors, both absolute and relative.

A proof checker like Coq does not work on the actual proof but on a script of tactics that describes each necessary logic operation to validate the conclusion from the premises. The proof is generated by the automatic proof checker by playing the script. Coq kernel stores the proof as a lambda-term following Curry-Howard isomorphism. The lambda-term cannot be read by human beings but it can be double checked by tools independent from Coq kernel and

Coq development teams. As it stands now, none of the files manipulated by Coq stores the proof in a human readable format. This is the reason why we always refer to the proof script that is played on the automatic proof checker to print on screen the steps of the proof.

2.1 Bounds on the computed values

Variables may be bounded into explicitly computed ranges certified by an automatic proof checker. For example, a responsible programmer should ascertain that inputs of all square root operations cannot be negative and that his program never divides by zero. Range checking also lets us verify that no result will overflow as floating point algorithms that are not specially built to deal with them will often lead to unanticipated results if one or more operations overflow. Worse, an overflowing operation on non saturated two's complement fixed point arithmetic will return a wrapped value with the wrong sign.

Ranges are stored as intervals using pairs of numbers: a lower bound and an upper bound. Interval representations are not optimal. It is however sufficient in most cases and as a trade-off, we avoid manipulating complex structures. Indeed, formally checking a proof is a computation intensive operation and using simpler properties (a pair of numbers instead of the whole description of a domain) makes verifications easier for the proof checker.

What is interesting for a proof is the fundamental property of interval arithmetic: if the arguments of an expression are replaced by enclosing intervals, the expression evaluation will lead to an interval that contains the result. Such an inclusion property is exactly what is needed for a proof: given the ranges of the inputs, interval arithmetic will give the range of the output – or more generally an overestimation of this range, but it still is a compatible answer since it contains all possible outputs.

Our tool may have handled domains on variables x but our past experiences on programs lead us to rather bound computed values. The upper and lower bounds will be used in pre- and post-conditions of implemented functions. When a C program contains an assertion `(0. <= x)`, `x` refers to the value actually in memory or in a register; consequently the assertion does not apply to x but to the computed value $\tilde{x}$. When a variable must be proved not to underflow or overflow, we mean to check the property on $\tilde{x}$, and so on. Most of the properties proved at this stage require knowing the domain of $\tilde{x}$; the domain of x may never be needed though it can be estimated with the error bound defined Section 2.2.

Our educated decision to bound the approximated variable rather than the exact quantities has another important consequence: there is no loss if the lower and upper bounds of $\tilde{x}$ are in $\mathbb{F}$ as the approximated value $\tilde{x}$ is an element of $\mathbb{F}$ by definition and we do not consider intervals of $\mathbb{R}$ but intervals of $\mathbb{F}$. The consequence on formal proof checking is substantial, bounds on the exact values could need arbitrarily wide expressions to exactly represent real numbers. On the other hand, bounds on $\tilde{x}$ are rationals with a very limited precision (for example IEEE standard single or double precision numbers).

Common interval arithmetic considers convex subsets of $\mathbb{R}$. Consequently, the implementation of the addition operator for a traditional interval arithmetic library is:

$$[\tilde{a}, \tilde{b}] + [\tilde{c}, \tilde{d}] = [\bigtriangledown(\tilde{a} + \tilde{c}), \bigtriangleup(\tilde{b} + \tilde{d})].$$

Here $\bigtriangledown$ and $\bigtriangleup$ denote that operations should be rounded toward $-\infty$ and $+\infty$ respectively. The bounds may not carry enough precision for $\tilde{a} + \tilde{c}$ and $\tilde{b} + \tilde{d}$ to be exactly stored. It is the reason why the two operations are rounded so that the resulting interval encloses all possible values.

In our cases, such rounding considerations are not necessary: the hardware-implemented addition $\boxplus$ is considered as a new exact operation with respect to $\mathbb{F}$ and not as the approximation of the addition on the field of the real numbers. That means that

$$\text{if} \quad \tilde{x} \in [\tilde{a}, \tilde{b}] \quad \text{and} \quad \tilde{y} \in [\tilde{c}, \tilde{d}] \quad \text{then} \quad \tilde{x} \boxplus \tilde{y} \in [\tilde{a} \boxplus \tilde{c}, \tilde{b} \boxplus \tilde{d}]$$

Indeed, the values $\tilde{a} \boxplus \tilde{c}$ and $\tilde{b} \boxplus \tilde{d}$ already lie in $\mathbb{F}$, there is no need for an additional rounding. Consequently, since the addition $\boxplus$ is a monotone function of $\mathbb{F}^2 \to \mathbb{F}$ whatever the rounding mode, then $\tilde{a} \boxplus \tilde{c}$ and $\tilde{b} \boxplus \tilde{d}$ are the bounds.

$$[\tilde{a}, \tilde{b}] \boxplus [\tilde{c}, \tilde{d}] = [\tilde{a} \boxplus \tilde{c}, \tilde{b} \boxplus \tilde{d}].$$

There is no need to take instead possibly different and non-optimal values $\bigtriangledown(\tilde{a} + \tilde{c})$ and $\bigtriangleup(\tilde{b} + \tilde{d})$.

Consequently, the most commonly used interval arithmetic libraries were not appropriate for our project. We chose the Boost parameterizable interval arithmetic library [5]. This is a C++ library that allows us to specify the types and the arithmetic to be used instead of being limited to arithmetic on $\mathbb{R}$.

2.2 Bounds on the errors

The other arithmetic property that our tool proves is the difference between the exact result and the computed value. Finite precision of numeric computations is likely to induce a difference between the expected result and the computed value. Such differences will progressively build up and it is necessary for the user to certify that the computed result is meaningful.

Bounds on values and errors are quite similar and we could have decided to handle them at the same time. Each variable would have been represented by four numbers (two ranges) and there would have been new operators inspired by interval arithmetic that would compute on these quartets. Such a representation would have given shorter proof scripts. However there are situations where such a simplicity would be restrictive.

For example, the error range for the expression $x/\sqrt{1+x^2}$ would be easily computed by this method, but the range of the result would not. Indeed the result is in the interval $[-1,1]$ for any x [16], and this tight interval may be a precondition for another part of the algorithm. Yet, interval arithmetic is usually not able to yield such a small interval: multiple occurrences of the same variable in the expression leads to an overestimation of the result usually called decorrelation.

Interval arithmetic is unable to keep track of the relations between sub-expressions and $x/\sqrt{1+x^2}$ is seen as if it was $x/\sqrt{1+y^2}$, with y having the same domain as x but not necessarily the same value. Consequently, in this example, if x is in the interval $[0,2]$, $\sqrt{1+x^2}$ is in the interval $[1,s_5]$ with s_5 the representable value the closest from $\sqrt{5}$. Standard interval arithmetic will then answer that the whole expression is in $[0,2] \div [1,s_5] = [0,2]$. This interval is no subset of the range $[-1,1]$ another part of the algorithm could expect. Standard interval arithmetic cannot be used to validate the tight range of this expression and human interaction is needed instead.

Consequently, it may be preferable for proofs on bounds and errors to be separated so that if our tool is not able to prove one of them it may still be used for the other one, limiting human interactions as much as possible. Point 2.2.1 recalls the standard definition of the absolute error in forward analysis. Point 2.2.2 elaborates on the two alternate definitions for the relative error.

2.2.1 Absolute error

The most commonly encountered kind of error representations are absolute and relative errors [7]. Absolute error is useful with fixed point arithmetic since the magnitude of the rounding error does not depend on the magnitude of the computed value. It may also be useful with floating point arithmetic when a subtraction cancels, when the exponents of the results stay close or with subnormal numbers (including zero). Relative errors can be used in all the other cases.

Absolute error η_x is attached to variable x by equality

$$x = \tilde{x} + \eta_x.$$

We will not define the behavior of all the implemented operators as forward error analysis is very common. As an example, here is the formula giving the absolute error for addition. η_0 is the rounding error between the exact result of $\tilde{x} + \tilde{y}$ and its computed value $\tilde{x} \boxplus \tilde{y}$. The absolute error is the difference between the exact value $z = x + y$ and the computed value $\tilde{z} = \tilde{x} \boxplus \tilde{y}$.

$$\begin{aligned} z - \tilde{z} &= (x + y) - (\tilde{x} \boxplus \tilde{y}) \\ &= (x + y) - (\tilde{x} + \tilde{y} - \eta_0) \\ &= (x - \tilde{x}) + (y - \tilde{y}) + \eta_0 \\ &= \eta_x + \eta_y + \eta_0. \end{aligned}$$

We extend the definition to interval bounds. Intervals A_x and A_y are attached to variables x and y and respectively enclose η_x and η_y. Interval A_0 encloses all possible values for η_0. This interval is given by the properties of the underlying arithmetic. For fixed point arithmetic, it is a constant interval. For floating point arithmetic, it depends on the value of $\tilde{z}$. Our tool can also detect Sterbenz's conditions and automatically proves that A_0 is equal to zero if the hypotheses of the theorem are verified for all $\tilde{x}$ and $\tilde{y}$ [27,3].

$$\eta_z \in A_z \subseteq A_x + A_y + A_0$$

Bounds on values of Section 2.1 were not real numbers and the precision of interval arithmetic was very limited. It is not the case for error intervals and we used common interval arithmetic with rounded overestimations.

2.2.2 Relative error

We only consider relative error when x and $\tilde{x}$ have the same sign. The most common representation of this error is $\tilde{x} = x \times (1 + \epsilon_1)$ but another possible one is $x = \tilde{x} \times (1 + \epsilon_2)$ [12].

In order to select one of these two representations, let us consider the case where $\tilde{x}$ is one representable number closest to x (there is usually only one $\tilde{x}$ value unless x is a mid-point between two representable values). Bounding ϵ_1 and ϵ_2 for this particular case is important because it will be used after each operation to incorporate the additional rounding error. Provided we are sufficiently far from the underflow threshold,

$$\begin{aligned} -\frac{u}{2} \le -\frac{u}{2+u} &\le \epsilon_1 \le \frac{u}{2+u} \le \frac{u}{2} \quad \text{and} \\ -\frac{u}{2} \le -\frac{u}{2(1+u)} &\le \epsilon_2 \qquad \le \qquad \frac{u}{2} \end{aligned}$$

where u is the difference between 1 and the next representable floating point number. These bounds are a consequence of the nearness of x and $\tilde{x}$ ($\tilde{x}^-$ and $\tilde{x}^+$ are the floating point numbers respectively just below and just above $\tilde{x}$):

$$\frac{\tilde{x}^- - \tilde{x}}{2} \leq x - \tilde{x} \leq \frac{\tilde{x}^+ - \tilde{x}}{2}$$

Now if $\tilde{x}$ can also be the smallest positive normalized number, the inequalities become

$$\begin{array}{ccccccccc} -\frac{u}{2} & \leq & -\frac{u}{2+u} & \leq \epsilon_1 \leq & \frac{u}{2-u} & \not\leq & \frac{u}{2} & \quad \text{and} \\ -\frac{u}{2} & & \leq & \epsilon_2 & \leq & & \frac{u}{2} & . \end{array}$$

As a conclusion using $x = \tilde{x} \times (1 + \epsilon_x)$ with $\epsilon_x = \epsilon_2 \in [-u/2, u/2]$ gives a range valid on all normalized numbers and consequently only the exponent of $\tilde{x}$ needs to be checked. Once again, we provide only one example as forward error analysis is very common. For the multiplication $z = x \times y$, if $\tilde{z}$ is always normalized, the relative error is:

$$\begin{aligned} \frac{z}{\tilde{z}} &= \frac{x \times y}{\tilde{x} \boxtimes \tilde{y}} \\ &= \frac{x \times y}{\tilde{x} \times \tilde{y}}(1 + \epsilon_0) \\ &= (1 + \epsilon_x)(1 + \epsilon_y)(1 + \epsilon_0). \end{aligned}$$

Again, we attach intervals R_x and R_y to respective variables x and y enclosing ϵ_x and ϵ_y. Let interval R_0 enclose the value of the relative error ϵ_0. Its range depends on the properties of the arithmetic and we have seen that in the common cases of floating point arithmetic rounding to nearest, it is $[-u/2, u/2]$ and finally

$$\epsilon_z \in R_z \subseteq (1 + R_x)(1 + R_y)(1 + R_0) - 1.$$

2.3 Formal proof checking

Our tool generates a Coq script of the proof. Such a proof script does not live by itself: it only describes what elementary facts should be applied, it does not provide the proofs of these theorems. Indeed these facts are provided by various libraries: theorems on real numbers arithmetic (the Coq standard library), specification and basic properties of floating point arithmetic [6]. It also depends on a library about interval arithmetic and its floating point extensions we developed so that each generated proof could be formally verified. For example, in order to use interval arithmetic for the addition, this composition property is needed:

$$\tilde{a} \boxplus \tilde{c} \leq \tilde{b} \boxplus \tilde{d} \quad \text{if} \quad \tilde{a} \leq \tilde{b} \quad \text{and} \quad \tilde{c} \leq \tilde{d}.$$

Non trivial theorems like Sterbenz's one are also stated [3], extended to floating point intervals in our library, and used by our tool.

On the other hand, our tool is not a part of the Coq program and it can also be used with other provers such as HOL Light [11] or PVS [24] with the ProofLite[2] package that supports batch proving and proof scripting in PVS. To preserve the independence of our tool, we do not use any technique specific to one prover such as tactics and automatic proof developments. The tool only requires the prover to provide basic integer computations and results comparisons. If the prover is able to decide on its own that an expression like $5 \times 3 - 4 \geq 2 + 8$ is true, then it can be used by our tool. Adapting our tool to another prover will only be a matter of syntax as soon as libraries for computer arithmetic are available.

We benefit from the fact that our tool is not a part of any prover and consequently that it does not need to be proved. For example, when proving the property $\tilde{a} \leq \tilde{b} \boxplus \tilde{c}$, the tool acts as an oracle and computes beforehand the sum $\tilde{d} = \tilde{b} \boxplus \tilde{c}$ and provides this number $\tilde{d}$ to the proof checker. Consequently, the prover does not have to explicitly compute $\tilde{b} \boxplus \tilde{c}$ in order to prove the property, it only has to verify that $\tilde{d}$ as provided by the tool is the closest floating point number to $\tilde{b} + \tilde{c}$ (with respect to the rounding mode of $\boxplus$), and then that $\tilde{a} \leq \tilde{d}$. Both operations can be done by the existing Coq floating point library. Using the tool as an oracle avoids having to implement and prove in the prover all the floating point operations with their various rounding.

This solution works fine for the addition and the multiplication as such operations are internal on dyadic numbers and they are provided by the library. Dyadic numbers are pairs (m, e) of $\mathbb{Z}^2$ interpreted as $m \times 2^e$. Radix 2 floating point numbers are dyadic numbers and our library relies on these. However mathematical division and square root of dyadic numbers do not generally produce dyadic numbers. Consequently these operators on real numbers cannot be directly used on dyadic numbers and the proofs cannot rely on them. Fortunately these operations can be reverted when rounding to nearest. For example,

$$\tilde{c} = \circ(\tilde{a} \div \tilde{b}) \quad \text{iff} \quad \tilde{b} \times \tilde{c} \ \text{ is closer from } \ \tilde{a} \ \text{ than } \ \tilde{b} \times \tilde{c}^- \ \text{ and } \ \tilde{b} \times \tilde{c}^+$$

where $\tilde{c}^-$ and $\tilde{c}^+$ are the floating point numbers respectively just below and just above $\tilde{c}$.

[2] `http://research.nianet.org/~munoz/ProofLite/`.

Error bounds are also represented on dyadic numbers from the library. Our decision certainly reduces precision, but the situation is sensible in regard to common practices and it is also fully parameterizable. Moreover, we decided to further limit precision of the error bounds. Since our tool is much faster than any automatic proof checker, it tries to simplify the numbers used in the proof as much as possible. Without any simplification, the length necessary to represent the error bounds grows continually. So does the time needed to verify each statement of the script as the proof goes on.

The simplification is done by replacing $m \times 2^e$ by $\lceil m \cdot 2^{-k} \rceil \times 2^{e+k}$ for a given k. This new bound is worse, but computing with it is faster since the constant $\lceil m \cdot 2^{-k} \rceil$ has a shorter representation. In order to benefit from this property, our tool enlarges all the error ranges as long as the final ranges remain smaller than the ones targeted by the user. Since the interesting part of an error is generally its magnitude and not its exact value, it is an efficient way to reduce the script size and its certification time. For example, formally checking the simplified proof was four times faster on the application of Section 3.

3 Maintaining safety distances between aircrafts

There is a lot of work done on shifting responsibility for aircraft safety distances from air traffic controllers to pilots themselves. This involves providing pilots with conflict detection systems. One algorithm [22] works in an Euclidean three-dimensional neighborhood and a translation is necessary from geodetic coordinates as given by positioning systems [21] to Euclidean coordinates.

Airplanes are protected by a cylinder (typically 5 miles diameter and 1000 feet height) and a conflict is defined as the overlapping of two protected zones. The above mentioned work has certified that it is equivalent to detect conflicts relative to the intruder. In this case, the intruder has a protected zone of radius 5 miles and height of 2000 feet. The ownship, the airplane running the conflict detection system, is regarded as a point. A conflict is then defined as the incursion of the ownship in the protected zone of the intruder. The purpose of conflict detection is to detect such intrusions ahead of time and to propose escape and recovery trajectories.

A lot of approximations are necessary to solve this problem: position and speed of the planes are not exactly known, the use of an Euclidean neighborhood instead of geodetic coordinates creates modeling errors, translation itself incurs modeling and round-off errors. All these errors need to be taken into account so that the safety of the systems is guaranteed. The cylinder is easily inflated so that fewer and fewer trajectories may cross the initial safety cylinders. Yet, it is life critical to guarantee that the cylinder has been sufficiently inflated to remove any incorrect behavior.

As mentioned, there already is a published algorithm that works in an Euclidean three-dimensional neighborhood. We will describe here a new algorithm responsible of the translation part and use it as an example of a safety critical application that can be formally proved thanks to our tool.

Algorithm 1 Translation from geodetic coordinates to an Euclidean neighborhood

---

Input: ϕ_1, ϕ_2, λ_1, λ_2, v_{N1}, v_{N2}, v_{E1}, v_{E2}

$$
\begin{aligned}
\phi_0 &\leftarrow (\phi_1 + \phi_2)/2 \\
r_{p0} &\leftarrow R_p(\phi_0) \\
s_{N1} &\leftarrow v_{N1}/R_m(\phi_1) \\
s_{N2} &\leftarrow v_{N2}/R_m(\phi_2) \\
s_x &\leftarrow (\lambda_1 - \lambda_2) * r_{p0} \\
s_y &\leftarrow (\phi_1 - \phi_2) * R_m(\phi_0) \\
v_{x1} &\leftarrow v_{E1} * r_{p0}/R_p(\phi_1 + s_{N1} * t_r) \\
v_{x2} &\leftarrow v_{E2} * r_{p0}/R_p(\phi_2 + s_{N2} * t_r)
\end{aligned}
$$

The inputs of the Algorithm 1 are the longitudes $\lambda_{\{1,2\}}$ and the geodetic latitudes $\phi_{\{1,2\}}$ of planes 1 and 2, and their speeds toward north $v_{N\{1,2\}}$, toward east $v_{E\{1,2\}}$. The outputs are the relative coordinates and the speed of both planes in an Euclidean neighborhood. t_r is a constant. R_m and R_p are Earth radii along a meridian and along a parallel for a given latitude:

$$
\begin{aligned}
R_p(\phi) &= \frac{a}{1 + (1 - f)^2 \tan^2 \phi} \\
R_m(\phi) &= \frac{a(1 - e^2)}{(1 - e^2 \sin^2 \phi)^{3/2}}
\end{aligned}
$$

Constants a, e and f are described by WGS84[3] and are also used to define the geodetic coordinates in the positioning systems.

These functions use trigonometric functions that must be approximated on computers. Therefore, we decided to approximate directly R_m and R_p. This adds more errors as the algorithm uses polynomials $\hat{R}_m$ and $\hat{R}_p$ rather than R_m and R_p. The program for $\hat{R}_p$ is given in Algorithm 2. $\hat{R}_m$ is omitted but it is simple implementation of Horner's rule. We introduce an uncertainty on the first coefficient of each polynomial to account for the difference between the exact function and the approximated polynomial in our tool. This difference

[3] http://www.wgs84.com/.

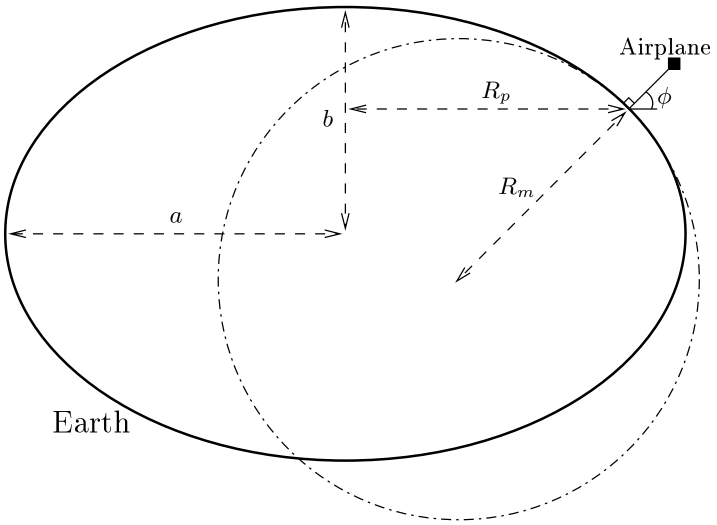

Figure 1. Geodetic latitude and radii in WGS84 system

Algorithm 2 Definition of $\hat{R}_p(\phi)$ approximating $R_p(\phi)$

Input: ϕ {all the dyadic numbers are floating point numbers}

$x \leftarrow 511225 \times 2^{-18} - \phi^2$
$y \leftarrow 4439091 \cdot 2^{-2} + x \times (9023647 \cdot 2^{-2} + x \times ($
$\quad 13868737 \cdot 2^{-6} + x \times (13233647 \cdot 2^{-11} + x \times ($
$\quad -1898597 \cdot 2^{-14} + x \times (-6661427 \cdot 2^{-17})))))$

will soon be formally certified with PVS. By adding this uncertainty on the constants as an hypothesis, the tool is able to prove a global error encompassing the truncation error caused by the use of polynomial approximations.

The accuracy of the results depends on the precision of the arithmetic. For this example, we used underlying IEEE-754 simple precision floating point arithmetic with subnormal numbers and default rounding mode (rounding to nearest with even tie-breaking). The speed corresponds to an airliner and the latitudes range from 30°N to 40°N.

Our tool produced a 1650-line long script without human interaction. The generated proof guarantees that the absolute error on the euclidean positions s_x and s_y is bounded by half a meter, and the error on the speeds along the x-axis v_{x1} and v_{x2} of the two planes is bounded by 5×10^{-4} m/s. By combining these two errors, we conclude that the safety cylinder radius should be increased by 2.60 m to ensure the results of the algorithm stay valid during a 300-second window.

Coq script 1 is the last theorem as it appears in the Coq proof script, and below stands its translation when replacing the variables by their definition.

Coq script 1 E_195: The last lemma guaranteed by Coq

```
Lemma E_195: ... -> _B_16 -> ... -> (Eint_error Ce_195 V_vx_2 R_vx_2).
```

Lemma E_195: ... -> $R_{p3} = 13233647 \times 2^{-11}$ -> ... -> $|v_{x2} - \tilde{v}_{x2}| \leq 2^{-11}$.

The ellipses hide the 34 other hypotheses of the theorem. The one shown here is the hypothesis `_B_16` giving the value of the fourth constant used in the computation of R_p (Algorithm 2). The conclusion of the theorem says the absolute error between the theoretical value v_{x2} and the computed value $\tilde{v}_{x2}$ is bounded by 2^{-11} (almost 5×10^{-4} m/s).

The 1650 lines of the script can be partitioned to 850 lines of definitions and 800 lines of theorems and proofs. Only these 800 last lines are important with respect to computer arithmetic. The huge number of definitions is caused by the need to describe each operation and each number. For example, when proving a variable is in a given interval, the lower and upper bounds need to be defined separately first.

On the test-machine, a 2.6 GHz Pentium 4 computer, the script is verified by Coq in 45 seconds. The verification is a lot slower than the generation itself (a split second). This gap is to due to the way the script is verified: Coq has to check that any computation involved in the proof is indeed correct. Coq does not rely on the same interval arithmetic library than the tool, it redoes all the operations by itself, computing values one bit at a time (since it is the way integer arithmetic is proved in Coq standard library).

4 Conclusion and Perspectives

The tool that we have just presented is very limited. Yet it is complete enough to deal with our airplane safety distance example. The tool generates a script that can be formally verified by Coq and that proves bounds on the absolute error of the outputs. It can also easily generate a script for the case where the inputs are only known with a given limited accuracy.

Such scripts are self-contained: there is no need to formally prove the tool and the underlying interval arithmetic library in order to ensure the validity of the generated proofs. Even if there were bugs in the tool, they would not impact the validity of the results: once the libraries of lemmas and the proof scripts have been checked by the automatic prover, the results are formally guaranteed.

We are starting to offer generation of script from an Internet browser. Web pages at the following address use our tool to generate bounds and round-off errors of evaluation of polynomials following Horner's rule.

http://lipforge.ens-lyon.fr/www/gappa/horner.html

As our Web interface grows, we will offer more and more proof schemes available just at a few clicks.

Adding functionalities to our tool takes time. We have to state and prove the supporting theorems in our library. Today, our tool is limited to floating point arithmetic (rounded to nearest) and absolute error bounds. More functionalities will be incorporated in the future such as fixed point arithmetic and additional arithmetic operations. We may also take into account some DSP specific implementation of floating point arithmetic. Yet it is always possible to state quickly a few unvalidated axioms to incorporate one specific function for setting up a demonstration.

The results presented here are very different from what could have been done with a certified software using interval arithmetic. Our tool generates a proof of each theorem it creates. All proofs and theorems can later be used by Coq proof checker to validate new properties. A certified software for interval arithmetic would only return guaranteed intervals. It would not be possible to add anything not available when the software was certified. With our tool, the user may decide to prove any theorem and can use it as well as all the generated theorems and proofs.

Another perspective is a connection to the Why tool [8]. Why treats C and ML programs and generates proof obligations for automatic proof checkers. As no tool was available for floating point arithmetic, variables were treated as real numbers with no rounding error. We have started working with Why developers to extend it with the existing floating point arithmetic.

Finally, our tool only performs simple forward error analysis. It may be linked with other tools such as Fluctuat [25] that is able to automatically propose contracting intervals for loops. Only the certification step has to be validated by an automatic proof checker for the result to be fully guaranteed. One may use any tool to provide useful oracles as long as oracles are finally validated by the proofs produced by our tool.

5 Acknowledgment

The authors would like to thank César Muñoz from the National Institute of Aerospace and Víctor Carreño from the NASA Langley Research Center, who raised our interest on the critical application used as an example, for their guidance and discussions on aeronautical matters.

References

[1] W. Richards Adrion, Martha A. Branstad, and John C. Cherniavsky. Validation, verification and testing of computer software. *ACM Computing Surveys*, 14(2):159–192, 1982.

[2] Yves Bertot, Nicolas Magaud, and Paul Zimmermann. A proof of GMP square root. *Journal of Automated Reasoning*, 29(3-4):225–252, 2002.

[3] Sylvie Boldo and Marc Daumas. Properties of two's complement floating point notations. *International Journal on Software Tools for Technology Transfer*, 5(2-3):237–246, 2004.

[4] Sylvie Boldo and Marc Daumas. A simple test qualifying the accuracy of Horner's rule for polynomials. *Numerical Algorithms*, 2004.

[5] Hervé Brönnimann, Guillaume Melquiond, and Sylvain Pion. The Boost interval arithmetic library. In *Real Numbers and Computers*, pages 65–80, Lyon, France, 2003.

[6] Marc Daumas, Laurence Rideau, and Laurent Théry. A generic library of floating-point numbers and its application to exact computing. In *14th International Conference on Theorem Proving in Higher Order Logics*, pages 169–184, Edinburgh, Scotland, 2001.

[7] James W. Demmel. *Applied Numerical Linear Algebra*. SIAM, 1997.

[8] Jean-Christophe Filliâtre. Why: a multi-language multi-prover verification tool. Research Report 1366, Université Paris Sud, 2003.

[9] Debbie Gage and John McCormick. We did nothing wrong. *Baseline*, 1(28):32–58, 2004.

[10] John Harrison. Floating point verification in HOL light: the exponential function. Technical Report 428, University of Cambridge Computer Laboratory, 1997.

[11] John Harrison. *The HOL Light manual*, 2000. Version 1.1.

[12] Nicholas J. Higham. *Accuracy and stability of numerical algorithms*. SIAM, 1996.

[13] Gérard Huet, Gilles Kahn, and Christine Paulin-Mohring. The Coq proof assistant: a tutorial: version 7.2. Technical Report 256, Institut National de Recherche en Informatique et en Automatique, Le Chesnay, France, 2002.

[14] Information Management and Technology Division. Patriot missile defense: software problem led to system failure at Dhahran, Saudi Arabia. Report B-247094, United States General Accounting Office, 1992.

[15] Luc Jaulin, Michel Kieffer, Olivier Didrit, and Eric Walter. *Applied interval analysis*. Springer, 2001.

[16] William Kahan. Lecture notes on the status of the IEEE standard 754 for binary floating point arithmetic. Published on the net, 1996.

[17] Leslie Lamport and P. Michael Melliar-Smith. Synchronizing clocks in the presence of faults. *Journal of the ACM*, 32(1):52–78, 1985.

[18] Ren-Cang Li, Sylvie Boldo, and Marc Daumas. Theorems on efficient argument reductions. In Jean-Claude Bajard and Michael Schulte, editors, *Proceedings of the 16th Symposium on Computer Arithmetic*, pages 129–136, Santiago de Compostela, Spain, 2003.

[19] Jacques-Louis Lions et al. Ariane 5 flight 501 failure report by the inquiry board. Technical report, European Space Agency, Paris, France, 1996.

[20] Paul S. Miner. *Hardware verification using coinductive assertions.* PhD thesis, Indiana University, Bloomington, Indiana, 1998.

[21] Chris Moody and Gary Furr. Automatic dependent surveillance broadcast. Section 2.2, Draft 12 UAT-WP-12-07, Radio Technical Commission for Aeronautics, 2002.

[22] César Muñoz, Víctor Carreño, Gilles Dowek, and Ricky Butler. Formal verification of conflict detection algorithms. *International Journal on Software Tools for Technology Transfer*, 4(3):371–380, 2003.

[23] Arnold Neumaier. *Interval methods for systems of equations.* Cambridge University Press, 1990.

[24] Sam Owre, John M. Rushby, , and Natarajan Shankar. PVS: a prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction*, pages 748–752, Saratoga, New-York, 1992. Springer-Verlag.

[25] Sylvie Putot, Eric Goubault, and Matthieu Martel. Static analysis based validation of floating point computations. In *Novel Approaches to Verification*, volume 2991 of *Lecture Notes in Computer Science*, pages 306–313, Dagstuhl, Germany, 2004.

[26] John Rushby and Friedrich von Henke. Formal verification of algorithms for critical systems. In *Proceedings of the Conference on Software for Critical Systems*, pages 1–15, New Orleans, Louisiana, 1991.

[27] Pat H. Sterbenz. *Floating point computation.* Prentice Hall, 1974.

[28] Ping Tak Peter Tang. Table-lookup algorithms for elementary functions and their error analysis. In Peter Kornerup and David Matula, editors, *Proceedings of the 10th Symposium on Computer Arithmetic*, pages 232–236, Grenoble, France, 1991.