Floats & Ropes: a case study for formal numerical program verification*

Sylvie Boldo

INRIA Saclay - Île-de-France, ProVal, Orsay, F-91893
LRI, Univ Paris-Sud, CNRS, Orsay, F-91405

Abstract. We present a case study of a formal verification of a numerical program that computes the discretization of a simple partial differential equation. Bounding the rounding error was tricky as the usual idea, that is to bound the absolute value of the error at each step, fails. Our idea is to find out a precise analytical expression that cancels with itself at the next step, and to formally prove the correctness of this approach.

Keywords: rounding error, cancellation, formal proof, scientific computation, discretization of a partial differential equation.

1 Introduction

Given a program using floating-point arithmetic, it is pretty hard to know the final rounding error of the result. We are interested in proving numerical analysis programs with a very high level of guarantee. We present here one of the simplest example of scientific computation.

The basic property is that each floating-point result is a correct rounding of the exact real value: using the default rounding mode, the result is the floating-point number that is closest to the real value. This property is defined in the IEEE-754 standard [1, 2] and all modern processors comply with it.

Nevertheless, even if each computation is correct, *i.e.* the best possible, there is no guarantee that the final result after many such computations is still accurate. There exist several methods for bounding the final error of a program, including forward analysis [3], backward analysis [4] and interval arithmetic [5]. These well-known methods may or may not give useful results. However, when they state a bad rounding error, it does not always imply the error is huge. It is a known fact that floating-point errors may cancel [3] but it is very difficult to handle. We use here a method that displays these cancellations and takes advantage of them. This idea of exhibiting floating-point errors cancellation has been used by Even, Seidel and Ferguson in [6]. This article's technique is also linked to static analysis [7], and provides more precision and more readability at the cost of less genericity. This technique is also linked to expansions [8] as the error is somewhat "computed" and used.

* This work was funded by the French national research organization (ANR), by the CerPAN (ANR-05-BLAN-0281-04) and F∮st projects (ANR-08-BLAN-0246-01).

To increase the trust in our results, we use deductive formal methods: we machine-check all proofs using the Coq proof checker [9]. We use a high-level formalization of floating-point numbers [10, 11]. We use the Why platform for verification of C programs, that includes the Caduceus tool [12, 13]. The Caduceus tool allows the user to precisely specify a C program. Each function is annotated with pre-conditions (what the function `requires` from the inputs) and post-conditions (what the function `ensures` at its end). The annotations and requirements (pointer dereferencing for example) are then transformed into proof obligations that have to be solved by proof assistants or decision procedures.

The Caduceus tool has floating-point annotations [14] that allow to specify numerical programs. More precisely, each floating-point number has a ghost value called exact which does not suffer from rounding. This real value is then computed with the same operations as the float value except that the ghost operation is exact. The macro `round_error`(f) is then used for $|f - \text{exact}(f)|$. More, each floating-point number has another ghost value called model that the user may set and which does not suffer from rounding. It is used to represent the ideal result of the function (computed with infinite sums, no discretization...).

Inside the annotations, all computations are exact. For example, this program takes x as input and multiplies it by 2.

```
/*@ requires |x| < 2^^(1022)
  @ ensures   \result=2*x
  @     && \round_error(\result)=2*\round_error(x) */

double multiply2(double x) { return 2*x; }
```

This function requires x to be small enough so that the multiplication does not overflow. It ensures that the result, denoted by the macro `\result` is equal to the *mathematical* multiplication of x by 2. This is correct as the radix is 2.More, the rounding error of the result is twice the rounding error of the input.

Section 2 describes the first example, which is a simplification of the numerical program. Section 3 tackles the discretization of the spread of acoustic waves on a rope. Section 4 gives conclusive remarks and perspectives.

2 First Example: Linear Recurrence of Order 2

2.1 The Problem

We first present a linear recurrence of order 2. For some initial values u_0 and u_1, we compute the following sequence:

$$u_{n+1} = 2 \times u_n - u_{n-1}.$$

This may seem a silly idea as this sequence can be solved. Indeed, we know that, mathematically, $u_n = u_0 + (u_1 - u_0) \times n$. Nevertheless, this example is representative of the analytical error idea and corresponds rather nicely to our real problem (Section 3 with $a = 0$).

To compute u_N, we assume that $(u_i)_{i \leq N}$ is bounded: for all $i \leq N$, $|u_i| \leq 1$. This is a very strong property that requires u_0 and u_1 to be small and close enough one to another. This noteworthy limitation still corresponds to similar properties in our real program.

We use this C program that we will later annotate:

```
double comput_seq(double u0, double u1, int N) {
  int i;
  double uprev, ucur, tmp;
  uprev=u0; ucur =u1;

  for (i=2; i<=N; i++) {
    tmp   = 2*ucur-uprev;
    uprev = ucur;
    ucur  = tmp;
  }
  return ucur;
}
```

As the exact value of $|u_n|$ is bounded by 1, if the errors of u_{n-1} and u_{n-2} are not too big, then the error in the subtraction is less than 2^{-53}. A natural error analysis gives: let $E_i = u_i - \text{exact}(u_i)$, then $|E_{i+1}| \leq 2 \times |E_i| + |E_{i-1}| + 2^{-53}$. Assuming u_0 and u_1 are error-free, this gives us that $|E_N|$ is roughly equal to $2^N \times 2^{-53}$. This error is very pessimistic and should be improved upon.

2.2 The Analytical Error

The idea is that the error should be signed. Taking its absolute value can only lead to an exponential error. By keeping its sign, we have that:

$$E_{i+1} = 2 \times E_i - E_{i-1} + \varepsilon_{i+1} \quad \text{with} \quad |\varepsilon_{i+1}| \leq 2^{-53}.$$

The key point is that we have now a subtraction between $2 \times E_i$ and E_{i-1}. At each step, the error will only be added a small value, therefore E_{i-1} is close to E_i, so that $2 \times E_i - E_{i-1} \approx E_i$. This allows us to get rid of the exponential in the error bound. We now assume that u_0 and u_1 are not exact anymore. Of course, the initial errors must not be too big: we assume the computed u_i do not exceed 2. More precisely, the annotations of the C function are given in Figure 1.

Theorem 1. *If the pre-conditions of Figure 1 are satisfied, then the post-conditions of Figure 1 hold.*

Proof. This proof deeply relies on the definition of the predicate *mkp* which is a loop invariant inductively proved correct at each iteration update. The idea is to keep track of both the exact value and the exact floating-point error of u_p and u_c in order to bound the final error.

The predicate *mkp* is a property linking the inputs u_0 and u_1 and the state of the program: the number of iterations n and the current $u_n = \texttt{ucur} = u_c$ and

```
/*@ requires 2 <= N <= 2^^25-1 &&
  @       \round_error(u0) + \round_error(u1) <= 1./(6*N) &&
  @       \forall int k; 0 <= k <= N =>
  @              |\exact(u0)+k*(\exact(u1)-\exact(u0))| <= 1
  @ ensures
  @    \exact(\result)==\exact(u0)+N*(\exact(u1)-\exact(u0))
  @     && \round_error(\result) <= N*(N+1)/2.*2^^(-53)
  @                 + N*(\round_error(u0)+\round_error(u1))
  @*/

double comput_seq(double u0, double u1, int N) { ...
```

Fig. 1. Annotated C program for computing the linear recurrence of order 2

$u_{n-1} = \texttt{uprev} = u_p$. For a given float f, let us denote by $\delta(f) = f - \text{exact}(f)$. We have $\texttt{round_error}(f) = |\delta(f)|$. We define the predicate *mkp* by

$$\begin{array}{l} mkp(u_0, u_1, u_c, u_p, n) = \exists \varepsilon : \mathbb{N} \to \mathbb{R}, \\ \quad \forall i \in \mathbb{N}, i \le n \Rightarrow |\varepsilon_i| \le 2^{-53} \\ \quad \wedge\ \delta(u_p) = \sum_{i=0}^{n-1} (n-i)\ \varepsilon_i + (1-n)\ \delta(u_0) + n\ \delta(u_1) \\ \quad \wedge\ \delta(u_c) = \sum_{i=0}^{n} (n+1-i)\ \varepsilon_i + (-n)\ \delta(u_0) + (n+1)\ \delta(u_1) \end{array}.$$

As soon as *mkp* is defined, the proof is rather easy. The exact value of u_i is computed by recurrence. The *mkp* property is proved the same way. At stage i of the iteration, we define a new ε_i which is the signed rounding error committed during this iteration. As the multiplication is exact, $\varepsilon_i = u_i - (2 \times u_{i-1} - u_{i-2})$. This value can easily be bounded as this is the error of one single subtraction such that the result is smaller than 2. Therefore $|\varepsilon_i| \le 2^{-53}$ and

$$|E_N| \le \frac{N(N+1)}{2} 2^{-53} + N \times (|\delta(u_0)| + |\delta(u_1)|).$$

There is left to guarantee that $|u_i| \le 2$, we first know, according to our assumptions, that $|\text{exact}(u_i)| \le 1$. So there is left to prove that $|E_i| \le 1$. And the analytical expression of the error shows that the floating-point error is smaller than $i(i+1) \times 2^{-54} + i\,(|\delta(u_0)| + |\delta(u_1)|)$. We therefore need to bound N by about 2^{25} so that $i(i+1)2^{-54}$ is bounded enough. Moreover, we have to bound $|\delta(u_0)| + |\delta(u_1)| = \text{round_error}(u_0) + \text{round_error}(u_1)$ by $1/(6N)$. These values are sufficient to guarantee that the error is bounded: $|E_i| \le 1$. Note also that the rounding error bound could be improved to $O\left(N2^{-53}\right)$. □

3 Second Example: Rope

3.1 The Problem

The piece of code that is studied here is extracted from a numerical code by F. Clément about acoustic waves [15]: given a rope attached at its two ends, we

create a wave by applying a force (initializations). The rope then undulates, depending on some mathematical equations that can be discretized and computed.

The mathematical point of view is that we are looking for the solution u from $\mathbb{R}^2$ to $\mathbb{R}$ of the differential equation, knowing initial values of u and its derivative for $t = 0$:

$$\frac{\partial^2 u(x,t)}{\partial t^2} - c^2 \frac{\partial^2 u(x,t)}{\partial x^2} = 0.$$

The solution of the partial differential equation is approximated by the following piece of code that gives the position of the rope as times increases:

```
for (k=1; k<nk; k++) {
  p[0][k+1] = 0.;
  for (i=1; i<ni; i++) {
    dp = p[i+1][k] - 2.*p[i][k] + p[i-1][k];
    p[i][k+1] = 2.*p[i][k] - p[i][k-1] + a*dp;
   }
  p[ni][k+1] = 0.;
}
```

This is the main iteration of the program. Before that, $p[\dots][0]$ and $p[\dots][1]$ are set but we are interested in this main loop. The value `a` is a parameter computed previously. It is assumed that $0 < \mathtt{a} \lessapprox 1$. Typically, `a` can be the rounding of 0.9 or 0.99. At a given abscissa i and a given time k, the value `p[i][k]` $= p_i^k$ is the position of the rope. The value i is bounded by the ends of the rope 0 and n_i. We compute the position of the rope between the initial time 0 and a maximum time n_k.

We assume that (p_i^k) is bounded. The reason is that (p_i^k) represents values that are supposed to be smaller than 1 (as the rope cannot fly away). These model p_i^k cannot be computed as they would need infinite sums or absence of discretizations. Nevertheless, the $\text{exact}(p_i^k)$ are near these model values so we may assume they are smaller that 1.5. We assume n_k is small enough to guarantee that the floating-point values $|p_i^k|$ are smaller than 2. As the error will be proved proportional to k^2, this roughly corresponds to $n_k \le 2^{22}$.

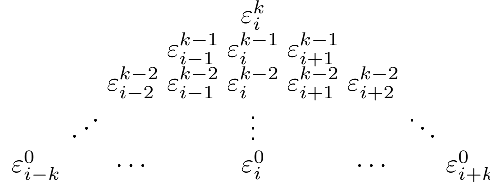
3.2 The Pyramids

Let ε_i^k be the (signed) floating-point error made when computing p_i^k. From the preceding program, we set

$$\varepsilon_i^{k+1} = p_i^{k+1} - (2p_i^k - p_i^{k-1} + \text{exact}(\mathtt{a}) \times (p_{i+1}^k - 2p_i^k + p_{i-1}^k)).$$

As the $|p_i^k|$ are assumed to be smaller than 2, this value can be bounded. To prove that, we use simple interval arithmetic: the idea is to bound each step of the proof. We formally prove that $|\varepsilon_i^{k+1}| \le 85 \times 2^{-52}$ for a reasonable error bound for `a`, that is to say $|\mathtt{a} - \text{exact}(\mathtt{a})| \le 2^{-49}$.

Given the definition of p_i^k and the preceding discussion on the dependencies, the floating-point error $E_i^k = p_i^k - \text{exact}(p_i^k)$ depends and only depends on the following ε_j^l:



This unfortunately means both that the error is at least proportional to k^2, and that the analytical error is a pyramidal double summation.

This is quite harder than the previous expression, as a double summation is more difficult to handle using a proof assistant. Unfortunately again, the error is not the sum of all the ε_j^l of the pyramid. They have to be multiplied by a well-chosen constant depending on their place in the pyramid. This constant is far from trivial and is mathematically defined in the next subsection.

3.3 The α Sequence

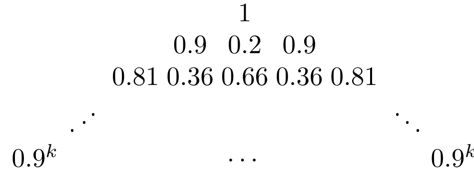
Let us introduce $\alpha : (\mathbb{Z} \times \mathbb{Z}) \to \mathbb{R}$ defined by

$$\alpha_0^0 = 1 \qquad \forall i \neq 0,\ \alpha_i^0 = 0$$

$$\alpha_{-1}^1 = \alpha_1^1 = \breve{a} \qquad \alpha_0^1 = 2(1-\breve{a}) \qquad \forall i \notin \{-1,0,1\},\ \alpha_i^1 = 0$$

$$\alpha_i^k = \breve{a} \times (\alpha_{i-1}^{k-1} + \alpha_{i+1}^{k-1}) + 2(1-\breve{a}) \times \alpha_i^{k-1} - \alpha_i^{k-2}$$

where $\breve{a}$ is the exact value of the floating-point value a of the preceding subsection (so $\breve{a}$ is typically 0.9 or 0.99). The non-zero terms fit in a pyramid looking like this for $\breve{a} = 0.9$, where lines are indexed by k and columns by i:



It is easy to prove that $\alpha_i^k = \alpha_{-i}^k$, that $\alpha_k^k = \breve{a}^k$ and that if $i < -k$ or $i > k$, then $\alpha_i^k = 0$. More interesting, the sum "by line" has a surprisingly simple expression:

Lemma 1.

$$\sum_{i=-\infty}^{+\infty} \alpha_i^k = k + 1.$$

Proof. We have:

$$\sum_{i=-\infty}^{+\infty} \alpha_i^{k+1} = 2\breve{a} \sum_{i=-\infty}^{+\infty} \alpha_i^k + 2(1-\breve{a}) \sum_{i=-\infty}^{+\infty} \alpha_i^k - \sum_{i=-\infty}^{+\infty} \alpha_i^{k-1} = 2 \sum_{i=-\infty}^{+\infty} \alpha_i^k - \sum_{i=-\infty}^{+\infty} \alpha_i^{k-1}.$$

The sum by line verifies the linear recurrence of Section 2. As $\sum_{i=-\infty}^{+\infty} \alpha_i^0 = 1$ and $\sum_{i=-\infty}^{+\infty} \alpha_i^1 = 2$, we have $\displaystyle\sum_{i=-\infty}^{+\infty} \alpha_i^k = k+1$. □

Lemma 2. $\alpha_i^k \geq 0$

Proof. The demonstration was found out by M. Kauers and V. Pillwein.

If we denote by P the Jacobi polynomial, we have

$$\alpha_n^j = \sum_{k=j}^{n} \binom{2k}{j+k} \binom{n+k+1}{2k+1} (-1)^{j+k} a^k = a^j \sum_{k=0}^{n-j} P_k^{(2j,0)}(1-2a)$$

Now the conjecture follows directly from the inequality of Askey and Gasper [16, 17], which asserts that $\sum_{k=0}^{n} P_k^{(r,0)}(x) > 0$ for $r > -1$ and $-1 < x \leq 1$ (see Thm 7.4.2 in The Red Book [18]). □

This assertion is not formally proved as it involves both many complex computations and very high level mathematics. Moreover, this lemma can be ignored at the price of a less tight bound (see Section 3.7).

3.4 The Analytical Error

Now, we claim that we can express the exact floating-point error of p_i^k in an analytical way:

Theorem 2.

$$E_i^k = p_i^k - exact(p_i^k) = \sum_{l=0}^{k} \sum_{j=-l}^{l} \alpha_j^l \ \varepsilon_{i+j}^{k-l}$$

Proof. Given the correct expression of E_i^k, the proof is a piece of cake. The analytical expression exactly fits the computation of p_i^{k+1} and the sequence α_i^k is defined so that they cancel at the right time.

We prove the expression of E_i^k by induction on k. We assume the initializations fulfill this requirement by choosing wisely ε_i^0 and ε_i^1 so that this expression is correct for $k=0$ and $k=1$. We now assume the expression is correct for $k-1$ and k and we prove it for $k+1$:

$$\begin{aligned}
E_i^{k+1} &= p_i^{k+1} - \text{exact}(p_i^{k+1}) \\
&= p_i^{k+1} - (2p_i^k - p_i^{k-1} + \check{a} \times (p_{i+1}^k - 2p_i^k + p_{i-1}^k)) \\
&\quad +2(1-\check{a})(p_i^k - \text{exact}(p_i^k)) \\
&\quad +\check{a}(p_{i+1}^k - \text{exact}(p_{i+1}^k) + p_{i-1}^k - \text{exact}(p_{i-1}^k)) \\
&\quad -(p_i^{k-1} - \text{exact}(p_i^{k-1})) \\
&= \varepsilon_i^{k+1} + 2(1-\check{a})E_i^k + \check{a}(E_{i+1}^k + E_{i-1}^k) - E_i^{k-1}
\end{aligned}$$

After a lot of tiring but stupid computation (mostly summation games), we have the equality:

$$E_i^{k+1} = \varepsilon_i^{k+1} + 2(1-\check{a})\varepsilon_i^k + \check{a}\varepsilon_{i+1}^k + \check{a}\varepsilon_{i-1}^k + \sum_{l=0}^{k-1}\Bigg(2(1-\check{a})\alpha_{-l-1}^{l+1}\varepsilon_{i-l-1}^{k-1-l}$$

$$+2(1-\check{a})\alpha_{l+1}^{l+1}\varepsilon_{i+l+1}^{k-1-l} + \check{a}\alpha_l^{l+1}\ \varepsilon_{i+l+1}^{k-1-l} + \check{a}\alpha_{l+1}^{l+1}\ \varepsilon_{i+l+2}^{k-1-l}$$

$$+\check{a}\alpha_{-l}^{l+1}\ \varepsilon_{i-l-1}^{k-1-l} + \check{a}\alpha_{-l-1}^{l+1}\ \varepsilon_{i-l-2}^{k-1-l} + \sum_{j=-l}^{l}\alpha_j^{l+2}\varepsilon_{i+j}^{k-1-l}\Bigg)$$

We also go the other way:

$$\sum_{l=0}^{k+1}\sum_{j=-l}^{l}\alpha_j^l\ \varepsilon_{i+j}^{k+1-l} = \quad \varepsilon_i^{k+1} + 2(1-\check{a})\varepsilon_i^k + \check{a}\varepsilon_{i+1}^k + \check{a}\varepsilon_{i-1}^k$$

$$+\sum_{l=0}^{k-1}\Bigg(\alpha_{-l-2}^{l+2}\ \varepsilon_{i-l-2}^{k-1-l} + \alpha_{-l-1}^{l+2}\ \varepsilon_{i-l-1}^{k-1-l} + \alpha_{l+1}^{l+2}\ \varepsilon_{i+l+1}^{k-1-l}$$

$$+\alpha_{l+2}^{l+2}\ \varepsilon_{i+l+2}^{k-1-l} + \sum_{j=-l}^{l}\alpha_j^{l+2}\ \varepsilon_{i+j}^{k-1-l}\Bigg)$$

Let us compute $\Delta = E_i^{k+1} - \sum_{l=0}^{k+1}\sum_{j=-l}^{l}\alpha_j^l\ \varepsilon_{i+j}^{k+1-l}$ in order to prove this value is 0. We use the facts that $\alpha_i^i = \check{a}^i$ and that $\alpha_{-l-1}^l = \alpha_{l+1}^l = 0$ and $\alpha_{-l-2}^{l+1} = \alpha_{l+2}^{l+1} = 0$.

$$\begin{aligned}
\Delta = \sum_{l=0}^{k-1}\big(&\varepsilon_{i+l+2}^{k-1-l}(\check{a}\alpha_{l+1}^{l+1} - \alpha_{l+2}^{l+2}) + \varepsilon_{i-l-2}^{k-1-l}(\check{a}\alpha_{-l-1}^{l+1} - \alpha_{-l-2}^{l+2})\\
&+\varepsilon_{i-l-1}^{k-1-l}(2(1-\check{a})\alpha_{-l-1}^{l+1} + \check{a}\alpha_{-l}^{l+1} - \alpha_{-l-1}^{l+2})\\
&+\varepsilon_{i+l+1}^{k-1-l}(2(1-\check{a})\alpha_{l+1}^{l+1} + \check{a}\alpha_l^{l+1} - \alpha_{l+1}^{l+2})\big)\\
= \sum_{l=0}^{k-1}&\varepsilon_{i-l-1}^{k-1-l}(2(1-\check{a})\alpha_{-l-1}^{l+1} + \check{a}\alpha_{-l}^{l+1}\\
&-(2(1-\check{a})\alpha_{-l-1}^{l+1} + \check{a}\alpha_{-l}^{l+1} + \check{a}\alpha_{-l-2}^{l+1} - \alpha_{-l-1}^l))\\
&+\varepsilon_{i+l+1}^{k-1-l}(2(1-\check{a})\alpha_{l+1}^{l+1} + \check{a}\alpha_l^{l+1}\\
&-(2(1-\check{a})\alpha_{l+1}^{l+1} + \check{a}\alpha_l^{l+1} + \check{a}\alpha_{l+2}^{l+1} - \alpha_{l+1}^l))\\
= \sum_{l=0}^{k-1}&\big(\varepsilon_{i-l-1}^{k-1-l}(-\check{a}\alpha_{-l-2}^{l+1} + \alpha_{-l-1}^l)) + \varepsilon_{i+l+1}^{k-1-l}(-\check{a}\alpha_{l+2}^{l+1} + \alpha_{l+1}^l))\big) = 0
\end{aligned}$$

This rather complicated expression is proved correct. We can express the precise floating-point error with this double summation. □

3.5 ε Tossing

The previous proof assumes that the double summation is correct for all (i', k') such that $k' < k$. This would be correct if there was an infinite set of i where p_i^k is computed. The i such that p_i^k is computed are the integers between 0 and n_i. At the ends of the range, p_i^k is exact because set to 0.

This is extremely unpleasant because this means our fine recurrence fails because of these extrema. The reason is that $E_0^k = 0$, so E_0^k is *a priori* not equal to the expected double summation, except if we define ε out of the $[0; n_i]$ range in order to ensure that $E_0^k = \sum_{l=0}^{k} \sum_{j=-l}^{l} \alpha_j^l \; \varepsilon_j^{k-l} = 0$ and $E_{n_i}^k = p_{n_i}^k - \text{exact}(p_{n_i}^k) = \sum_{l=0}^{k} \sum_{j=-l}^{l} \alpha_j^l \; \varepsilon_{n_i+j}^{k-l} = 0$.

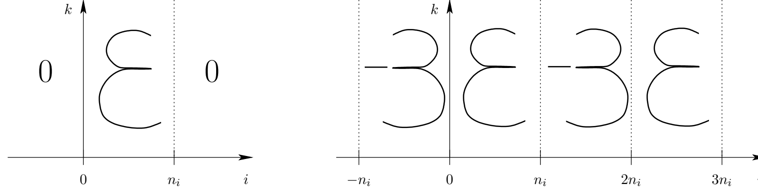


Fig. 2. Initial ε and tossed ε

We defined ε_i^k in the preceding Section for all k and for $0 < i < n_i$ as in Figure 2. We define $\varepsilon_0^k = 0$. We then define ε_i^k for all i, k in the following way:

- If $i \geq 0$, then
  - if $(i \div n_i) \mod 2 = 0$, then $\varepsilon_i^k = \varepsilon_{i \mod n_i}^k$,
  - else $\varepsilon_i^k = -\varepsilon_{(n_i - i) \mod n_i}^k$,
- else $\varepsilon_i^k = -\varepsilon_{-i}^k$.

On the ranges $[k \times n_i; (k+1) \times n_i]$, either we have ε (for even k), or we have a negated mirrored ε (for odd k). Figure 2 illustrates the way ε is defined on the whole range. This weird definition allows us to guarantee that the double summation is exactly zero for $i = 0$ and $i = n_i$. Indeed

$$E_0^k = \sum_{l=0}^{k} \sum_{j=-l}^{l} \alpha_j^l \; \varepsilon_j^{k-l} = \sum_{l=0}^{k} \left(\sum_{j=-l}^{-1} \alpha_j^l \; \varepsilon_j^{k-l} + \alpha_0^l \; \varepsilon_0^{k-l} + \sum_{j=1}^{l} \alpha_j^l \; \varepsilon_j^{k-l} \right)$$
$$= \sum_{l=0}^{k} \left(\sum_{j=1}^{l} -\alpha_j^l \; \varepsilon_j^{k-l} + 0 + \sum_{j=1}^{l} \alpha_j^l \; \varepsilon_j^{k-l} \right) = 0$$

This holds by symmetry of α and by antisymmetry of ε. The same kind of proof holds for $E_{n_i}^k = 0$.

This proof trick is here only to pretend that E_0 and E_{n_i} are equal to 0. They do not imply anything on the values of the E_i between these bounds.

3.6 Formal Proof

All the proofs described have been done and machined-checked using Coq. This allows us to formally verify the annotations of the loop invariant and the final error bound. The unproved assumptions are solved using axioms. The difficulty did not lie in the floating-point part. The only floating-point proof is the 85×2^{-52} one which is basic interval arithmetic. The hard part (apart from finding out the analytical expression) is handling the double summation expressions. We handle expressions such as $\sum_{l=1}^{k} \sum_{j=-l+1}^{l+1} \alpha_{j-1}^{l} \, \varepsilon_{i+j}^{k-l}$ with the following expression:

```
(sum_f_z (fun l : Z => sum_f_z (fun j : Z
   => alpha a (j - 1) (Zabs_nat l) * eps (i + j) (k - l))
    (- l + 1) (l + 1)) 1 k)%R.
```

This is rather cumbersome to handle, even if it is just following the pen and paper proof. There is indeed nothing very technical or tricky in this formal proof, except that the loop invariant must be defined using higher order logic.

All the annotated programs and the corresponding Coq developments described in this article are available at `http://www.lri.fr/~sboldo/gallery.html`.

3.7 Final error

The analytical expression of the error is in itself interesting (it may lead to know which error is dominating the others). Nevertheless, the main interest is to get a not-too-overestimated final bound for the rounding error.

Theorem 3.

$$\left|\mathbf{E_i^k}\right| = \left|\mathbf{p_i^k} - \mathbf{exact}\left(\mathbf{p_i^k}\right)\right| \leq \mathbf{85} \times \mathbf{2^{-53}} \times (\mathbf{k+1}) \times (\mathbf{k+2})$$

Proof. Let us bound the rounding error of p_i^k, that is $|E_i^k| = \left|\sum_{l=0}^{k} \sum_{j=-l}^{l} \alpha_j^l \; \varepsilon_{i+j}^{k-l}\right|$. We know that for all j and l, $|\varepsilon_j^l| \leq 85 \times 2^{-52}$ and that $\sum_{i=-\infty}^{+\infty} \alpha_i^l = l+1$ by Lemma 1. As the α_i^k are nonnegative, then the error is easily bounded by $85 \times 2^{-52} \times \sum_{l=0}^{k} l+1$. □

As the proof of the non-negativity of the α_i^k is nontrivial and not formally proved, the validity of this result may be put in question. Nevertheless, the α_i^k are the discretization of the differential equation with different initializations (and no rounding error). Therefore, as the $|p_i^k|$ are bounded, the $|\alpha_i^k|$ are bounded the same way. Therefore we may assume that $|\alpha_i^k| \leq 1.5$, and this permits to prove a bound on $|E_i^k|$:

$$|E_i^k| = \left| \sum_{l=0}^{k} \sum_{j=-l}^{l} \alpha_j^l \varepsilon_{i+j}^{k-l} \right| \le \sum_{l=0}^{k} \sum_{j=-l}^{l} |\alpha_j^l \varepsilon_{i+j}^{k-l}| \le 85 \times 2^{-52} \times \sum_{l=0}^{k} \sum_{j=-l}^{l} |\alpha_j^l|$$
$$\le 85 \times 2^{-52} \times \frac{3}{2} \sum_{l=0}^{k} \sum_{j=-l}^{l} 1 = 255 \times 2^{-53} \sum_{l=0}^{k} 2\,l + 1 < 2^{-45} \times (k+1)^2$$

This is a slightly worse bound than the previous one, but it does not rely on an result external from the study of the partial differential equation.

4 Conclusion

In order to prove the program entirely, the only difficulty left is the boundedness of the discretized solution of the partial differential equation: we assumed that $|p_i^k| \le 2$. As we bound the floating-point error, if $n_k \le 2^{22}$ or so, it is indeed enough to bound $|\mathrm{exact}(p_i^k)|$ by 1.5. This last fact is due to the consistency of the numerical scheme. This is a part of the global proof of the program, that demonstrates that this programs computes an approximation of the spread of acoustic waves on a rope. This mostly tackles Coq formalization of mathematical knowledge and requires many new definitions and lemmas about scalar product, symmetrical operators, Taylor series, $f = O(g)$, O of functions of two variables. . . and is therefore out of the scope of this paper.

This technique of the analytical error and precise floating-point error cancellation coming with its formal proof is new. The reason is that it requires very generic specifications as the loop invariant needs to be logically defined: it states there exists a function ε that has such and such property. And Caduceus allows us to express such a high-level property on a C program. We then use Coq as a back-end to formally check the specifications. This Caduceus genericity is an advantage compared to automatic methods that cannot express our loop invariant.

We have shown how the analytical error technique increases the quality of the final floating-point error on two examples. Instead of the exponential error we obtain by the usual methods, we get a quadratic error: this is indeed a terrific improvement. But the price is rather high as the user has to find out the exact expression of the analytical error before proving it. The analytical expression of the second example took us a few months to be worked out. We did not find any method to infer the analytical error from the program. We plan to develop methods to find out automatically analytical expressions or hints towards analytical expressions in order to spread the use of this technique.

This technique cannot provide an error bound to all floating-point programs as it requires a readable expression for the analytical error simple enough to handle in proofs. We also intend to study the class of programs to which this technique applies.

Acknowledgements

We thank the participants of the CerPAN project, namely M. Mayero, J.-C. Filliâtre and especially F. Clément for providing the C program, documentation and many explanations. We also thank B. Salvy, M. Kauers and V. Pillwein for their help (and proof!) on the non-negativity of the α_i^k.

References

1. Stevenson, D., et al.: A proposed standard for binary floating point arithmetic. IEEE Computer **14**(3) (1981) 51–62
2. IEEE: IEEE Standard for Floating-Point Arithmetic. IEEE Std. 754-2008 (August 2008) 1–58
3. Higham, N.J.: Accuracy and stability of numerical algorithms. SIAM (2002) Second Edition.
4. Wilkinson, J.H.: Rounding Errors in Algebraic Processes. Prentice-Hall, Upper Saddle River, NJ 07458, USA (1963)
5. Rump, S.M.: Fast and parallel interval arithmetic. BIT Numerical Mathematics **39**(3) (1999) 534–554
6. Even, G., Seidel, P.M., Ferguson, W.E.: A parametric error analysis of Goldschmidt's division algorithm. arith **00** (2003) 165
7. Goubault, E., Putot, S.: Static analysis of numerical algorithms. In Yi, K., ed.: SAS. Volume 4134 of LNCS., Springer (2006) 18–34
8. Dekker, T.J.: A floating point technique for extending the available precision. Numerische Mathematik **18**(3) (1971) 224–242
9. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. Springer Verlag (2004)
10. Daumas, M., Rideau, L., Théry, L.: A generic library of floating-point numbers and its application to exact computing. In: 14th International Conference on Theorem Proving in Higher Order Logics, Edinburgh, Scotland (2001) 169–184
11. Boldo, S.: Preuves formelles en arithmétiques à virgule flottante. PhD thesis, École Normale Supérieure de Lyon (November 2004)
12. Filliâtre, J.C., Marché, C.: Multi-Prover Verification of C Programs. In: Sixth International Conference on Formal Engineering Methods (ICFEM). Volume 3308 of LNCS., Seattle, Springer-Verlag (November 2004) 15–29
13. Filliâtre, J.C., Marché, C.: The Why/Krakatoa/Caduceus platform for deductive program verification. In Damm, W., Hermanns, H., eds.: 19th International Conference on Computer Aided Verification. LNCS, Berlin, Germany, Springer (2007)
14. Boldo, S., Filliâtre, J.C.: Formal verification of floating-point programs. In Kornerup, P., Muller, J.M., eds.: Proceedings of the 18th IEEE Symposium on Computer Arithmetic, Montpellier, France (June 2007) 187–194
15. Bécache, E.: Étude de schémas numériques pour la résolution de l'équation des ondes. ENSTA (September 2003)
16. Askey, R., Gasper, G.: Certain rational functions whose power series have positive coefficients. The American Mathematical Monthly **79** (1972) 327–341
17. Gasper, G.: Positive sums of the classical orthogonal polynomials. SIAM Journal on Mathematical Analysis **8**(3) (1977) 423–447
18. Andrews, G.E., Askey, R., Roy, R.: Special functions. Cambridge University Press, Cambridge (1999)