Computers and Mathematics with Applications 68 (2014) 325–352

Contents lists available at ScienceDirect

Computers and Mathematics with Applications

journal homepage: www.elsevier.com/locate/camwa

# Trusting computations: A mechanized proof from partial differential equations to actual program☆

Sylvie Boldo [a,b], François Clément [c,*], Jean-Christophe Filliâtre [b,a], Micaela Mayero [d], Guillaume Melquiond [a,b], Pierre Weis [c]

[a] Inria, Inria Saclay – Île-de-France, Palaiseau cedex, F-91120, France
[b] LRI, UMR 8623, Université Paris-Sud, CNRS, Orsay cedex, F-91405, France
[c] Inria, Inria Paris – Rocquencourt, Le Chesnay cedex, F-78153, France
[d] LIPN, UMR 7030, Université de Paris-Nord, CNRS, Villetaneuse, F-93430, France

ARTICLE INFO

*Article history:*
Received 14 May 2013
Received in revised form 28 May 2014
Accepted 1 June 2014
Available online 21 June 2014

*Keywords:*
Acoustic wave equation
Formal proof of numerical program
Convergence of numerical scheme
Rounding error analysis

ABSTRACT

Computer programs may go wrong due to exceptional behaviors, out-of-bound array accesses, or simply coding errors. Thus, they cannot be blindly trusted. Scientific computing programs make no exception in that respect, and even bring specific accuracy issues due to their massive use of floating-point computations. Yet, it is uncommon to guarantee their correctness. Indeed, we had to extend existing methods and tools for proving the correct behavior of programs to verify an existing numerical analysis program. This C program implements the second-order centered finite difference explicit scheme for solving the 1D wave equation. In fact, we have gone much further as we have mechanically verified the convergence of the numerical scheme in order to get a complete formal proof covering all aspects from partial differential equations to actual numerical results. To the best of our knowledge, this is the first time such a comprehensive proof is achieved.



## 1. Introduction

Given an appropriate set of mathematical equations (such as ODEs or PDEs) modeling a physical event, the usual simulation process consists of two stages. First, the continuous equations are approximated by a set of discrete equations, called the numerical scheme, which is then proved to be convergent. Second, the set of discrete equations is implemented as a computer program, which is eventually run to perform simulations.

The modeling of critical systems requires correctness of the modeling programs in the sense that there is no runtime error and the computed value is an accurate solution to the mathematical equations. The correctness of the program relies on the correctness of the two stages. Note that we do not consider here the issue of adequacy of the mathematical equations with the physical phenomenon of interest. We take the differential equation as a starting point in the simulation process. Usually, the discretization stage is justified by a pen-and-paper proof of the convergence of the selected scheme, while, following [1], the implementation stage is ratified by both code verification and solution verification. Code verification (checking for bugs)

☆ This research was supported by the ANR projects CerPAN (ANR-05-BLAN-0281-04) and Ffast (ANR-08-BLAN-0246-01).
* Corresponding author.
*E-mail addresses:* Sylvie.Boldo@inria.fr (S. Boldo), Francois.Clement@inria.fr (F. Clément), Jean-Christophe.Filliatre@inria.fr (J.-C. Filliâtre), Micaela.Mayero@lipn.univ-paris13.fr (M. Mayero), Guillaume.Melquiond@inria.fr (G. Melquiond), Pierre.Weis@inria.fr (P. Weis).

http://dx.doi.org/10.1016/j.camwa.2014.06.004
0898-1221/

uses manufactured solutions; it is called *validation by tests* below. Solution verification (checking for convergence of the numerical scheme at runtime) usually uses *a posteriori* error estimates to control the numerical errors; it is out of scope for this paper, nevertheless we briefly address the issue in the final discussion. The drawback of pen-and-paper proofs is that human beings are fallible and errors may be left, for example in long and tedious proofs involving a large number of subcases. The drawback of validation by tests is that, except for exhaustive testing which is impossible here, it does not imply a proof of correctness in all possible cases. Therefore, one may overestimate the convergence rate, or miss coding errors, or underestimate round-off errors due to floating-point computations. In short, this methodology only hints at the correctness of modeling programs but does not guarantee it.

The fallibility of pen-and-paper proofs and the limitations of validation by tests is not a new problem, and has been a fundamental concern for a long time in the computer science community. The answer to this question came from mathematical logic with the notion of *logical framework* and *formal proof*. A logical framework provides tools to describe mathematical objects and results, and state theorems to be proved. Then, the proof of those theorems gets all its logical steps verified in the logical framework by a computer running a mechanical proof checker. This kind of proof forbids logical errors and prevents omissions: it is a formal proof. Therefore, a formal proof can be considered as a perfect pen-and-paper proof.

Fortunately, logical frameworks also support the definition of computer programs and the specification of their properties. The correctness of a program can then be expressed as a formal proof that no execution of the program will go wrong and that it has the expected mathematical properties. A formal proof of a program can be considered as a comprehensive validation by an exhaustive set of tests. Note, however, that we verify the program at the source code level and do not consider here compilation problems, nor external attacks.

Mechanical proof checkers are mainly used to formalize mathematics and are routinely used to prove programs in the field of integer arithmetic and symbolic computation. We apply the same methodology to numerical programs in order to obtain the same safety level in the scientific computing field. The simulation process is revisited as follows. The discretization stage requires some preliminary work in the logical framework; we must implement the necessary mathematical concepts and results to describe continuous and discrete equations (in particular, the notion of convergent numerical scheme). Given this mathematical setting, we can write a faithful formal proof of the convergence of the discrete solution towards the solution to the continuous problem. Then, we can specify the modeling program and the properties of the computed values, and obtain a formal proof of its correctness. If we specify that computed values are close enough to the actual solution of the numerical scheme, then the correctness proof of the program ensures the correctness of the whole simulation process.

This revised simulation process seems easy enough. However, the difficulty of the necessary formal proofs must not be underestimated, notably because scientific computing adds specific difficulties to specifications and proofs. The discretization stage uses real numbers and real analysis theory. The usual theorems and tools of real analysis are still in their infancy in mechanical proof checkers. In addition, numerical programs use floating-point arithmetic. Properties of floating-point arithmetic are complex and highly nonintuitive, which is yet another challenge for formal proofs of numerical programs.

To summarize, the field of scientific computing has the usual difficulties of formal proof for mathematics and programs, and the specific difficulties of real analysis and its relationships to floating-point arithmetic. This complexity explains why mechanical proof checkers are mostly unknown in scientific computing. Recent progress [2–5] in providing mechanical proof checkers with formalizations of real analysis and IEEE-754 floating-point arithmetic makes formal proofs of numerical programs tractable nowadays.

In this article, we conduct the formal proof of a very simple C program implementing the second-order centered finite difference explicit scheme for solving the one-dimensional acoustic wave equation. This is a first step towards the formal proof of more complex programs used in critical situations. This article complements a previous publication about the same experiment [6]. This time however, we do not focus on the advances of some formal proof techniques, but we rather present an overview of how formal methods can be useful for scientific computing and what it takes to apply them.

Formal proof systems are relatively recent compared with mathematics or computer science. The system considered as the first proof assistant is Automath. It has been designed by de Bruijn in 1967 and has been very influential for the evolution of proof systems. As a matter of comparison, the FORTRAN language was born in 1954. Almost all modern proof assistants then appeared in the 1980s. In particular, the first version of Coq was created in 1984 by Coquand and Huet. The ability to reason about numerical programs came much later, as it requires some formal knowledge of arithmetic and analysis. In Coq, real numbers were formalized in 1999 and floating-point numbers in 2001. One can note that some of these developments were born from interactions between several domains, and so is this work.

The formal proofs are too long to be given here *in extenso*, so the paper only explains general ideas and difficulties. The annotated C program and the full Coq sources of the formal development are available from [7].

The paper is organized as follows. The notion of formal proof and the main formal tools are presented in Section 2. Section 3 describes the PDE, the numerical scheme, and their mathematical properties. Section 4 is devoted to the formal proof of the convergence of the numerical scheme, Section 5 to the formal proof of the boundedness on the round-off error, and Section 6 to the formal proof of the C program implementing the numerical scheme. Finally, Section 7 paints a broader picture of the study.

A glossary of terms from the mathematical logic and computer science fields is given in Appendix A. The main occurrences of *such terms*⋆ are emphasized in the text by using italic font and superscript star.

## 2. Formal proof

Modern mathematics can be seen as the science of abstract objects, *e.g.* real numbers, differential equations. In contrast, mathematical logic researches the various languages used to define such abstract objects and reason about them. Once these languages are formalized, one can manipulate and reason about mathematical proofs: What makes a valid proof? How can we develop one? And so on. This paves the way to two topics we are interested in: mechanical *verification*⋆ of proofs, and automated deduction of theorems. In both cases, the use of computer-based tools will be paramount to the success of the approach.

### 2.1. What is a formal proof?

When it comes to abstract objects, believing that some properties are true requires some methods of judgment. Unfortunately, some of these methods might be fallible: they might be incorrect in general, or their execution might be lacking in a particular setting. Logical reasoning aims at eliminating any unjustified assumption and ensuring that only infallible inferences are used, thus leading to properties that are believed to be true with the greatest confidence.

The reasoning steps that are applied to deduce from a property believed to be true a new property believed to be true is called an *inference rule*⋆. They are usually handled at a syntactic level: only the form of the statements matters, their content does not. For instance, the *modus ponens* rule states that, if both properties "*A*" and "if *A* then *B*" hold, then property "*B*" holds too, whatever the meaning of *A* and *B*. Conversely, if one deduces "*B*" from "*A*" and "if *C* then *B*", then something is amiss: while the result might hold, its proof is definitely incorrect.

This is where *formal proofs*⋆ show up. Indeed, since inference rules are simple manipulations of symbols, applying them or checking that they have been properly applied do not require much intelligence. (The intelligence lies in choosing which one to apply.) Therefore, these tasks can be delegated to a computer running a formal system. The computer will perform them much more quickly and systematically than a human being could ever do it. Assuming that such formal systems have been designed with care,[1] the results they produce are true with the greatest confidence.

The downside of formal proofs is that they are really low-level; they are down to the most elementary steps of a reasoning. It is no longer possible to dismiss some steps of the proofs, trusting the reader to be intelligent enough to fill the blanks. Fortunately, since inference rules are mechanical by nature, a formal system can also try to apply them automatically without any user interaction. Thus it will produce new results, or at least proofs of known results. At worst, one could imagine that a formal system applies inference rules blindly in sequence until a complete proof of a given result is found. In practice, clever algorithms have been designed to find the proper inference steps for domain-specific properties. This considerably eases the process of writing formal proofs. Note that numerical analysis is not amenable to automatic proving yet, which means that related properties will require a lot of human interaction, as shown in Section 6.2.

It should have become apparent by now that formal systems are primarily aimed at proving and checking mathematical theorems. Fortunately, programs can be turned into *semantically*⋆ equivalent abstract objects that formal systems can manipulate, thus allowing to prove theorems about programs. These theorems might be about basic properties of a program, *e.g.* it will not evaluate arrays outside their bounds. They might also be about higher-level properties, *e.g.* the computed results have such and such properties. For instance, in this paper, we are interested in proving that the values computed by the program are actually close to the exact solution to the partial differential equation. Note that these verifications are said to be static: they are done once and for all, yet they cover all the future executions of a program.

Formal verification of a program comes with a disclaimer though, since a program is not just an abstract object, it also has a concrete behavior once executed. Even if one has formally proved that a program always returns the expected value, mishaps might still happen. Perfect certainty is unachievable. First and foremost, the *specification*⋆ of what the program is expected to compute might be wrong or just incomplete. For instance, a random generator could be defined as being a function that takes no input and returns a value between 0 and 1. One could then formally verify that a given function satisfies such a specification. Yet that does not tell anything about the actual randomness of the computed value: the function might always return the same number while still satisfying the specification. This means that formal proofs do not completely remove the need for testing, as one still needs to make sure specifications are meaningful; but they considerably reduce the need for exhaustive testing.

Another consideration regarding the extent of confidence in formally verified programs stems from the fact that programs do not run in isolation, so formal methods have to make some assumptions. Basically, they assume that the program executed in the end is the one that was actually verified and not some variation of it. This seems an obvious assumption, but practice has shown that a program might be miscompiled, that some malware might be poking memory at random, that a computer processor might have design flaws, or even that the electromagnetic environment might cause bit flips either when verifying the program, or when executing it. So the trust in what a program actually computes will still be conditioned to the trust in the environment it is executed in. Note that this issue is not specific to verified programs, so they still have the upper hand over unverified programs. Moreover, formal methods are also applied to improve the overall trust in a system: formal

[1] The core of a formal system is usually a very small program, much smaller than any proof it will later have to manipulate, and thus easy to check and trust. For instance, while expressive enough to tackle any proof of modern mathematics, the kernel of HOL Light is just 200 lines long.

verification of hardware design is now routine, and formal verification of compilers [8,9] and operating systems [10] are bleeding edge research topics.

### 2.2. Formal proof tools at work

There is not a single tool that would allow us to tackle the formal *verification*⋆ of the C program we are interested in. We will use different tools depending on the kind of abstract objects we want to manipulate or prove properties about.

The first step lies in running the tool Frama-C over the program (Section 2.2.2). We have slightly modified the C program by adding comments stating what the program is expected to compute. These *annotations*⋆ are just mathematical properties of the program variables, *e.g.* the result variables are close approximations to the values of the exact solution. Except for these comments, the code was not modified. Frama-C takes the program and the annotations and it generates a set of theorems. What the tool guarantees is that, if we are able to prove all these theorems, then the program is formally verified. Some of these theorems ensure that the execution will not cause exceptional behaviors: no accesses out of the bounds of the arrays, no overflow during computations, and so on. The other theorems ensure that the program satisfies all its annotations.

At this point, we can run tools over the generated theorems, in the hope that they will automatically find proofs of them. For instance, Gappa (Section 2.2.3) is suited for proving theorems stating that floating-point operations do not overflow or that their round-off error is bounded, while *SMT solvers*⋆ (Section 2.2.2) will tackle theorems stating that arrays are never accessed out of their bounds. Unfortunately, more complicated theorems require some user interaction, so we have used the Coq proof assistant (Section 2.2.1) to help us in writing their formal proofs. This is especially true for theorems that deal with the more mathematically-oriented aspect of verification, such as convergence of the numerical scheme.

#### 2.2.1. Coq

Coq[2] is a formal system that provides an expressive language to write mathematical definitions, executable algorithms, and theorems, together with an interactive environment for proving them [11]. Coq's formal language combines both a *higher-order logic*⋆ and a richly-typed *functional programming*⋆ language [12]. In addition to functions and predicates, Coq allows the specification of mathematical theorems and software *specifications*⋆, and to interactively develop formal proofs of those.

The architecture of Coq can be split into three parts. First, there is a relatively small *kernel* that is responsible for mechanically checking formal proofs. Given a theorem proved in Coq, one does not need to read and understand the proof to be sure that the theorem statement is correct, one just has to trust this kernel.

Second, Coq provides a proof development system so that the user does not have to write the low-level proofs that the kernel expects. There are some interactive proof methods (proof by induction, proof by contradiction, intermediate lemmas, and so on), some *decision*⋆ and semi-decision algorithms (*e.g.* proving the equality between polynomials), and a *tactic*⋆ language for letting the user define his or her own proof methods. Note that all these high-level proof tools do not have to be trusted, since the kernel will check the low-level proofs they produce to ensure that all the theorems are properly proved.

Third, Coq comes with a standard library. It contains a collection of basic and well-known theorems that have already been formally proved beforehand. It provides developments and axiomatizations about sets, lists, sorting, arithmetic, real numbers, and so on. In this work, we mainly use the Reals standard library [2], which is a classical axiomatization of an Archimedean ordered complete field. It comes from the Coq standard library and provides all the basic theorems about analysis, *e.g.* differentials, integrals. It does not contain more advanced topics such as the Fourier transform and its properties though.

Here is a short example taken from our `alpha.v` file [7]:

```
Lemma Rabs_le_trans: forall a b c d : R,
  Rabs (a − c) + Rabs (c − b) ≤ d → Rabs (a − b) ≤ d.
Proof.
intros a b c d H.
replace (a − b) with ((a − c) + (c − b)) by ring.
apply Rle_trans with (2 := H); apply Rabs_triang.
Qed.
```

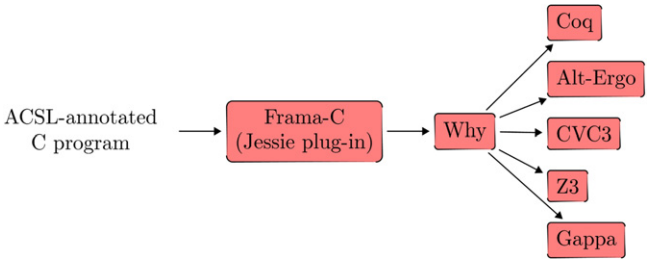
The function `Rabs` is the absolute value on real numbers. The lemma states that, for all real numbers $a$, $b$, $c$, and $d$, if $|a - c| + |c - b| \leq d$, then $|a - b| \leq d$. The proof is therefore quite simple. We first introduce variables and call `H` the hypothesis of the conditional stating that $|a - c| + |c - b| \leq d$. To prove that $|a - b| \leq d$, we first replace $a - b$ with $(a - c) + (c - b)$, the proof of that being automatic as it is only an algebraic ring equality. Then, we are left to prove that $|(a - c) + (c - b)| \leq d$. We use transitivity of $\leq$, called `Rle_trans`, and hypothesis `H`. Then, we are left to prove that $|(a - c) + (c - b)| \leq |a - c| + |c - b|$. This is exactly the triangle inequality, called `Rabs_triang`. The proof ends with the keyword **Qed**.

[2] http://coq.inria.fr/.

The standard library does not come with a formalization of floating-point numbers. For that purpose, we use a large Coq library called PFF[3] initially developed in [13] and extended with various results afterwards [14]. It is a high-level formalization of the IEEE-754 international standard for floating-point arithmetic [15,16]. This formalization is convenient for human interactive proofs as testified by the numerous proofs using it. The huge number of lemmas available in the library (about 1400) makes it suitable for a large range of applications. The library has been superseded since then by the Flocq library [3] and both libraries were used to prove the floating-point results of this work.

### 2.2.2. Frama-C, Jessie, Why, and the SMT solvers

We use the Frama-C platform[4] to perform formal verification of C programs at the source-code level. Frama-C is an extensible framework that combines *static analyzers*⋆ for C programs, written as plug-ins, within a single tool. In this work, we use the Jessie plug-in [17] for *deductive verification*⋆. C programs are *annotated*⋆ with behavioral contracts written using the *ANSI C Specification Language* (ACSL for short) [18]. The Jessie plug-in translates them to the Jessie language [17], which is part of the Why verification platform [19]. This part of the process is responsible for translating the *semantics*⋆ of C into a set of Why logical definitions (to model C types, memory heap, and so on) and Why programs (to model C programs). Finally, the Why platform computes *verification conditions*⋆ from these programs, using traditional techniques of weakest preconditions [20], and emits them to a wide set of existing theorem provers, ranging from *interactive proof assistants*⋆ to *automated theorem provers*⋆. In this work, we use the Coq proof assistant (Section 2.2.1), *SMT solvers*⋆ Alt-Ergo [21], CVC3 [22], and Z3 [23], and the automated theorem prover Gappa (Section 2.2.3). Details about automated and interactive proofs can be found in Section 6.2. The dataflow from C source code to theorem provers can be depicted as follows:




More precisely, to run the tools on a C program, we use a graphical interface called gWhy. A screenshot is displayed in Fig. 1, in Section 6. In this interface, we may call one prover on several *goals*⋆. We then get a graphical view of how many goals are proved and by which prover.

In ACSL, annotations are written using *first-order logic*⋆. Following the *programming by contract* approach, the specifications involve preconditions, postconditions, and *loop invariants*⋆. The contract of the following function states that it computes the square of an integer `x`, or rather a lower bound on it:

```
//@ requires x ≥ 0;
//@ ensures \result * \result ≤ x;
int square_root(int x);
```

The precondition, introduced with `requires`, states that the argument `x` is nonnegative. Whenever this function is called, the toolchain will generate a theorem stating that the input is nonnegative. The user then has to prove it to ensure the program is correct. The postcondition, introduced with *ensures*, states the property satisfied by the return value `\result`. An important point is that, in the specification, arithmetic operations are mathematical, not machine operations. In particular, the product `\result * \result` cannot overflow. Simply speaking, we can say that C integers are reflected within specifications as mathematical integers, in an obvious way.

The translation of floating-point numbers is more subtle, since one needs to talk about both the value actually computed by an expression, and the ideal value that would have been computed if we had computers able to work on real numbers. For instance, the following excerpt from our C program specifies the relative error on the content of the `dx` variable, which represents the grid step $\Delta x$ (see Section 3.2):

```
dx = 1./ni;
/*@ assert
  @  dx > 0. && dx ≤ 0.5 &&
  @  \abs(\exact(dx) − dx) / dx ≤ 0x1.p−53;
  @ */
```

[3] http://lipforge.ens-lyon.fr/www/pff/.
[4] http://www.frama-c.cea.fr/.

The identifier `dx` represents the value actually computed (seen as a real number), while the expression `\exact(dx)` represents the value that would have been computed if mathematical operators had been used instead of floating-point operators. Note that `0x1.p-53` is a valid ACSL literal (and C too) meaning $1 \cdot 2^{-53}$ (which is also the machine epsilon on binary64 numbers).

*2.2.3. Gappa*

The Gappa tool[5] adapts the *interval-arithmetic*⋆ paradigm to the proof of properties that occur when verifying numerical applications [24]. The inputs are logical formulas quantified over real numbers whose *atoms*⋆ are usually enclosures of arithmetic expressions inside numeric intervals. Gappa answers whether it succeeded in verifying it. In order to support program *verification*⋆, one can use *rounding* functions inside expressions. These unary operators take a real number and return the closest real number in a given direction that is representable in a given binary floating-point format. For instance, assuming that operator $\circ$ rounds to the nearest binary64 floating-point number, the following formula states that the relative error of the floating-point addition is bounded [16]:

$$\forall x, y \in \mathbb{R},\ \exists \varepsilon \in \mathbb{R}, \quad |\varepsilon| \leq 2^{-53} \quad \text{and} \quad \circ\,(\circ(x) + \circ(y)) = (\circ(x) + \circ(y))(1 + \varepsilon).$$

Converting *straight-line*⋆ numerical programs to Gappa logical formulas is easy and the user can provide additional hints if the tool were to fail to verify a property. The tool is specially designed to handle codes that perform convoluted floating-point manipulations. For instance, it has been successfully used to verify a state-of-the-art library of correctly-rounded elementary functions [4]. In the current work, Gappa has been used to check much simpler properties. In particular, no user hint was needed to automatically prove them. Yet the length of their proofs would discourage even the most dedicated users if they were to be manually handled. One of the properties is the round-off error of a local evaluation of the numerical scheme (Section 5.1). Other properties mainly deal with proving that no exceptional behavior occurs while executing the program: due to the initial values, all the computed values are sufficiently small to never cause overflow.

Verification of some formulas requires reasonings that are so long and intricate [4], that it might cast some doubts on whether an automatic tool can actually succeed in proving them. This is especially true when the tool ends up proving a property stronger than what the user expected. That is why Gappa also generates a formal proof that can be mechanically checked by a proof assistant. This feature has served as the basis for a Coq *tactic*⋆ for automatically proving *goals*⋆ related to floating-point and real arithmetic [25]. Note that Gappa itself is not verified, but since Coq verifies the proofs that Gappa generates, the goals are formally proved.

This tactic has been used whenever a *verification condition*⋆ would have been directly proved by Gappa, if not for some confusing notations or encodings of matrix elements. We just had to apply a few basic Coq tactics to put the goal into the proper form and then call the Gappa tactic to prove it automatically.

## 3. Numerical scheme for the wave equation

We have chosen to study the numerical solution to the one-dimensional acoustic wave equation using the second-order centered explicit scheme as it is simple, yet representative of a wide class of scientific computing problems. First, following [26], we describe and state the different notions necessary for the implementation of the numerical scheme and its analysis. Then, we present the annotations added in the source code to specify the behavior of the program.

### 3.1. Continuous equation

We consider $\Omega = [x_{\min}, x_{\max}]$, a one-dimensional homogeneous acoustic medium characterized by the constant propagation velocity $c$. Let $p(x,t)$ be the acoustic quantity, *e.g.* the transverse displacement of a vibrating string, or the acoustic pressure. Let $p_0(x)$ and $p_1(x)$ be the initial conditions. Let us consider homogeneous Dirichlet boundary conditions.

The one-dimensional acoustic problem on $\Omega$ is set by

$$\forall t \geq 0,\ \forall x \in \Omega, \qquad (L(c)\,p)(x,t) \stackrel{\text{def}}{=} \frac{\partial^2 p}{\partial t^2}(x,t) + A(c)\,p(x,t) = 0, \tag{1}$$

$$\forall x \in \Omega, \qquad (L_1\,p)(x,0) \stackrel{\text{def}}{=} \frac{\partial p}{\partial t}(x,0) = p_1(x), \tag{2}$$

$$\forall x \in \Omega, \qquad (L_0\,p)(x,0) \stackrel{\text{def}}{=} p(x,0) = p_0(x), \tag{3}$$

$$\forall t \geq 0, \qquad p(x_{\min},t) = p(x_{\max},t) = 0 \tag{4}$$

where the differential operator $A(c)$ acting on function $q$ is defined as

$$A(c)q \stackrel{\text{def}}{=} -c^2 \frac{\partial^2 q}{\partial x^2}. \tag{5}$$

[5] http://gappa.gforge.inria.fr/.

We assume that under reasonable regularity conditions on the Cauchy data $p_0$ and $p_1$, for each $c > 0$, there exists a unique solution $p$ to the initial-boundary value problem defined by Eqs. (1)–(5). Of course, it is well-known that the solution to this partial differential equation is given by d'Alembert's formula [27]. But simply assuming existence of a solution instead of exhibiting it ensures that our approach scales to more general cases for which there is no known analytic expression of a solution, *e.g.* when propagation velocity $c$ depends on space variable $x$.

We introduce the positive definite quadratic quantity

$$E(c)(p)(t) \stackrel{\text{def}}{=} \frac{1}{2}\left\|\frac{\partial p}{\partial t}(\cdot,t)\right\|^2 + \frac{1}{2}\|p(\cdot,t)\|^2_{A(c)} \tag{6}$$

where $\langle q, r\rangle \stackrel{\text{def}}{=} \int_\Omega q(x)r(x)dx$, $\|q\|^2 \stackrel{\text{def}}{=} \langle q, q\rangle$ and $\|q\|^2_{A(c)} \stackrel{\text{def}}{=} \langle A(c)\,q, q\rangle$. The first term is interpreted as the kinetic energy, and the second term as the potential energy, making $E$ the mechanical energy of the acoustic system.

Let $\tilde{p}_0$ (resp. $\tilde{p}_1$) represent the function defined on the entire real axis $\mathbb{R}$ obtained by successive antisymmetric extensions in space of $p_0$ (resp. $p_1$). For example, we have, for all $x \in [2x_{\min} - x_{\max}, x_{\min}]$, $\tilde{p_0}(x) = -p_0(2x_{\min} - x)$. The image theory [28] stipulates that the solution of the wave equation defined by Eqs. (1)–(5) coincides on domain $\Omega$ with the solution of the same wave equation but set on the entire real axis $\mathbb{R}$, without the Dirichlet boundary condition (4), and with extended Cauchy data $\tilde{p}_0$ and $\tilde{p}_1$.

### 3.2. Discrete equations

Let us consider the time interval $[0, t_{\max}]$. Let $i_{\max}$ (resp. $k_{\max}$) be the number of intervals of the space (resp. time) discretization. We define[6]

$$\Delta x \stackrel{\text{def}}{=} \frac{x_{\max} - x_{\min}}{i_{\max}}, \qquad i_{\Delta x}(x) \stackrel{\text{def}}{=} \left\lfloor \frac{x - x_{\min}}{\Delta x} \right\rfloor, \qquad \Delta t \stackrel{\text{def}}{=} \frac{t_{\max}}{k_{\max}}, \qquad k_{\Delta t}(t) \stackrel{\text{def}}{=} \left\lfloor \frac{t}{\Delta t} \right\rfloor. \tag{7}$$

The regular discrete grid approximating $\Omega \times [0, t_{\max}]$ is defined by[7]

$$\forall k \in [0..k_{\max}],\ \forall i \in [0..i_{\max}], \quad \mathbf{x}_i^k \stackrel{\text{def}}{=} (x_i, t^k) \stackrel{\text{def}}{=} (x_{\min} + i\Delta x, k\Delta t). \tag{8}$$

For a function $q$ defined over $\Omega \times [0, t_{\max}]$ (resp. $\Omega$), the notation $q_{\text{h}}$ (with a roman index h) denotes any discrete approximation of $q$ at the points of the grid, *i.e.* a discrete function over $[0..i_{\max}] \times [0..k_{\max}]$ (resp. $[0..i_{\max}]$). By extension, the notation $q_{\text{h}}$ is also a shortcut to denote the matrix $(q_i^k)_{0\le i\le i_{\max}, 0\le k\le k_{\max}}$ (resp. the vector $(q_i)_{0\le i\le i_{\max}}$). The notation $\bar{q}_{\text{h}}$ (with a bar over it) is reserved to specify the values of $q_{\text{h}}$ at the grid points, $\bar{q}_i^k \stackrel{\text{def}}{=} q(\mathbf{x}_i^k)$ (resp. $\bar{q}_i \stackrel{\text{def}}{=} q(x_i)$).

Let $u_{0,\text{h}}$ and $u_{1,\text{h}}$ be two discrete functions over $[0..i_{\max}]$. Let $s_{\text{h}}$ be a discrete function over $[0..i_{\max}] \times [0..k_{\max}]$. Then, the discrete function $p_{\text{h}}$ over $[0..i_{\max}] \times [0..k_{\max}]$ is the *solution of the second-order centered finite difference explicit scheme*, when the following set of equations holds:

$$\forall k \in [2..k_{\max}],\ \forall i \in [1..i_{\max}-1], \qquad (L_{\text{h}}(c)\,p_{\text{h}})_i^k \stackrel{\text{def}}{=} \frac{p_i^k - 2p_i^{k-1} + p_i^{k-2}}{\Delta t^2} + (A_{\text{h}}(c)\,p_{\text{h}}^{k-1})_i = s_i^{k-1}, \tag{9}$$

$$\forall i \in [1..i_{\max}-1], \qquad (L_{1,\text{h}}(c)\,p_{\text{h}})_i \stackrel{\text{def}}{=} \frac{p_i^1 - p_i^0}{\Delta t} + \frac{\Delta t}{2}(A_{\text{h}}(c)\,p_{\text{h}}^0)_i = u_{1,i}, \tag{10}$$

$$\forall i \in [1..i_{\max}-1], \qquad (L_{0,\text{h}}\,p_{\text{h}})_i \stackrel{\text{def}}{=} p_i^0 = u_{0,i}, \tag{11}$$

$$\forall k \in [0..k_{\max}], \qquad p_0^k = p_{i_{\max}}^k = 0 \tag{12}$$

where the matrix $A_{\text{h}}(c)$ (discrete analog of $A(c)$) is defined on vectors $q_{\text{h}}$ by

$$\forall i \in [1..i_{\max}-1], \quad (A_{\text{h}}(c)\,q_{\text{h}})_i \stackrel{\text{def}}{=} -c^2\frac{q_{i+1} - 2q_i + q_{i-1}}{\Delta x^2}. \tag{13}$$

Note the use of a second-order approximation of the first derivative in time in Eq. (10).

A discrete analog of the energy is also defined by

$$E_{\text{h}}(c)(p_{\text{h}})^{k+\frac{1}{2}} \stackrel{\text{def}}{=} \frac{1}{2}\left\|\frac{p_{\text{h}}^{k+1} - p_{\text{h}}^k}{\Delta t}\right\|^2_{\Delta x} + \frac{1}{2}\left\langle p_{\text{h}}^k, p_{\text{h}}^{k+1}\right\rangle_{A_{\text{h}}(c)} \tag{14}$$

[6] *Floor* notation $\lfloor . \rfloor$ denotes rounding to an integer towards minus infinity.

[7] For integers $n$ and $p$, the notation $[n..p]$ denotes the integer range $[n, p] \cap \mathbb{N}$.

where, for any vectors $q_\mathrm{h}$ and $r_\mathrm{h}$,

$$\langle q_\mathrm{h}, r_\mathrm{h}\rangle_{\Delta x} \stackrel{\text{def}}{=} \sum_{i=1}^{i_{\max}-1} q_i r_i \Delta x, \quad \|q_\mathrm{h}\|^2_{\Delta x} \stackrel{\text{def}}{=} \langle q_\mathrm{h}, q_\mathrm{h}\rangle_{\Delta x}, \quad \langle q_\mathrm{h}, r_\mathrm{h}\rangle_{A_\mathrm{h}(c)} \stackrel{\text{def}}{=} \langle A_\mathrm{h}(c)\, q_\mathrm{h}, r_\mathrm{h}\rangle_{\Delta x}, \quad \|q_\mathrm{h}\|^2_{A_\mathrm{h}(c)} \stackrel{\text{def}}{=} \langle q_\mathrm{h}, q_\mathrm{h}\rangle_{A_\mathrm{h}(c)}.$$

Note that $\|.\|_{A_\mathrm{h}(c)}$ is a semi-norm. Thus, the discrete energy $E_\mathrm{h}(c)$ is targeted to be a positive semidefinite quadratic form.

Note that the numerical scheme is parameterized by the discrete Cauchy data $u_{0,\mathrm{h}}$ and $u_{1,\mathrm{h}}$, and by the discrete source term $s_\mathrm{h}$. When these input data are respectively approximations of the exact functions $p_0$ and $p_1$ (*e.g.* when $u_{0,\mathrm{h}} = \bar{p}_{0\mathrm{h}}$, $u_{1,\mathrm{h}} = \bar{p}_{1\mathrm{h}}$, and $s_\mathrm{h} \equiv 0$), the discrete solution $p_\mathrm{h}$ is an approximation of the exact solution $p$.

The remark about image theory holds here too: we may replace the use of Dirichlet boundary conditions in Eq. (12) by considering extended discrete Cauchy data $\bar{\bar{p}}_{0\mathrm{h}}$ and $\bar{\bar{p}}_{1\mathrm{h}}$.

Note also that, as well as in the continuous case when a source term is considered on the right-hand side of Eq. (1), the discrete solution of the numerical scheme defined by Eqs. (9)–(13) can be obtained by the discrete space–time convolution of the discrete fundamental solution and the discrete source term. This will be useful in Sections 5.2 and 7.2.

### 3.3. Convergence

The main properties required for a numerical scheme are the convergence, the consistency, and the stability. A numerical scheme is *convergent* when the convergence error, *i.e.* the difference between exact and approximated solutions, tends to zero with respect to the discretization parameters. It is *consistent with the continuous equations* when the truncation error, *i.e.* the difference between exact and approximated equations, tends to zero with respect to the discretization parameters. It is *stable* if the approximated solution is bounded when discretization parameters tend to zero.

The Lax equivalence theorem stipulates that consistency implies equivalence between stability and convergence, *e.g.* see [29]. Consistency proof is usually straightforward, and stability proof is typically much easier than convergence proof. Therefore, in practice, the convergence of numerical schemes is obtained by using the Lax equivalence theorem once consistency and stability are established. Unfortunately, we cannot follow this popular path, since building a *formal proof*$^\star$ of the Lax equivalence theorem is quite challenging. Assuming such a powerful theorem is not an option either, since it would jeopardize the whole formal proof. Instead, we formally prove that consistency and stability imply convergence in the particular case of the second-order centered scheme for the wave equation.

The Fourier transform is a popular tool to study the convergence of numerical schemes. Unfortunately, the formalization of the Lebesgue integral theory and the Fourier transform theory does not yet exist in Coq (Section 2.2.1); such a development should not encounter any major difficulty, except for its human cost. As an alternative, we consider energy-based techniques. The stability is then expressed in terms of a discrete energy which only involves finite summations (to compute discrete dot products). Energy-based approaches are less precise because they follow *a priori* error estimates; but, unlike the Fourier analysis approach, they can be extended to the heterogeneous case, and to non uniform grids.

The CFL condition (for Courant–Friedrichs–Lewy, see [30]) states for the one-dimensional acoustic wave equation that the Courant number

$$\mathrm{CN} \stackrel{\text{def}}{=} \frac{c\Delta t}{\Delta x} \tag{15}$$

should not be greater than 1. To simplify the formal proofs, we use a more restrictive CFL condition. First, we rule out the particular case $\mathrm{CN} = 1$ since it is not so important in practice, and would imply a devoted structure of proofs. Second, the convergence proof based on energy techniques requires CN to stay away from 1 (for instance, constant $C_2$ in Eq. (27) may explode when $\xi$ tends to 0). Thus, we parameterize the CFL condition with a real number $\xi \in ]0, 1[$:

$$\mathrm{CFL}(\xi) \stackrel{\text{def}}{=} \mathrm{CN} \leq 1 - \xi. \tag{16}$$

Our formal proofs are valid for all $\xi$ in $]0, 1[$. However, to deal with floating-point arithmetic, the C code is annotated with values of $\xi$ down to $2^{-50}$.

For the numerical scheme defined by Eqs. (9)–(13), the convergence error $e_\mathrm{h}$ and the truncation error $\varepsilon_\mathrm{h}$ are defined by

$$\forall k \in [0..k_{\max}],\ \forall i \in [0..i_{\max}], \qquad e_i^k \stackrel{\text{def}}{=} \bar{p}_i^k - p_i^k, \tag{17}$$

$$\forall k \in [2..k_{\max}],\ \forall i \in [1..i_{\max}-1], \qquad \varepsilon_i^k \stackrel{\text{def}}{=} (L_\mathrm{h}(c)\,\bar{p}_\mathrm{h})_i^k, \tag{18}$$

$$\forall i \in [1..i_{\max}-1], \qquad \varepsilon_i^1 \stackrel{\text{def}}{=} (L_{1,\mathrm{h}}(c)\,\bar{p}_\mathrm{h})_i - \bar{p}_{1,i}, \tag{19}$$

$$\forall i \in [1..i_{\max}-1], \qquad \varepsilon_i^0 \stackrel{\text{def}}{=} (L_{0,\mathrm{h}}\bar{p}_\mathrm{h})_i - \bar{p}_{0,i}, \tag{20}$$

$$\forall k \in [0..k_{\max}], \qquad \varepsilon_0^k = \varepsilon_{i_{\max}}^k \stackrel{\text{def}}{=} 0. \tag{21}$$

Note that, when the input data of the numerical scheme for the approximation of the exact solution are given by $u_{0,\mathrm{h}} = \bar{p}_{0\mathrm{h}}$, $u_{1,\mathrm{h}} = \bar{p}_{1\mathrm{h}}$, and $s_\mathrm{h} \equiv 0$, then the convergence error $e_\mathrm{h}$ is itself solution of the same numerical scheme with discrete

inputs corresponding to the truncation error: $u_{0,\mathrm{h}} = \varepsilon_{\mathrm{h}}^0 = 0, u_{1,\mathrm{h}} = \varepsilon_{\mathrm{h}}^1$, and $s_{\mathrm{h}} = \left(k \mapsto \varepsilon_{\mathrm{h}}^{k+1}\right)$. This will be useful in Section 4.4.

In Section 4.4, we discuss the formal proof that the numerical scheme is *convergent of order* $(m, n)$ *uniformly on the interval* $[0, t_{\max}]$ if the convergence error satisfies[8]

$$\left\| e_{\mathrm{h}}^{k_{\Delta t}(t)} \right\|_{\Delta x} = O_{[0,t_{\max}]}(\Delta x^m + \Delta t^n). \tag{22}$$

In Section 4.2, we discuss the formal proof that the numerical scheme is *consistent with the continuous problem at order* $(m, n)$ *uniformly on interval* $[0, t_{\max}]$ if the truncation error satisfies

$$\left\| \varepsilon_{\mathrm{h}}^{k_{\Delta t}(t)} \right\|_{\Delta x} = O_{[0,t_{\max}]}(\Delta x^m + \Delta t^n). \tag{23}$$

In Section 4.3, we discuss the formal proof that the numerical scheme is *energetically stable uniformly on interval* $[0, t_{\max}]$ if the discrete energy defined by Eq. (14) satisfies

$$\exists \alpha, C_1, C_2 > 0,\ \forall t \in [0, t_{\max}],\ \forall \Delta x, \Delta t > 0,$$
$$\sqrt{\Delta x^2 + \Delta t^2} < \alpha \Rightarrow \sqrt{E_{\mathrm{h}}(c)(p_{\mathrm{h}})^{k_{\Delta t}(t)+\frac{1}{2}}} \leq C_1 + C_2 \Delta t \sum_{k'=1}^{k_{\Delta t}(t)} \left\| \left(i \mapsto s_i^{k'}\right) \right\|_{\Delta x}. \tag{24}$$

Note that constants $C_1$ and $C_2$ may depend on the discrete Cauchy data $u_{0,\mathrm{h}}$ and $u_{1,\mathrm{h}}$.

### 3.4. C program

We assume that $x_{\min} = 0, x_{\max} = 1$, and that the absolute value of the exact solution is bounded by 1. The main part of the C program is listed in Listing 1. Grid steps $\Delta x$ and $\Delta t$ are respectively represented by variables `dx` and `dt`, grid sizes $i_{\max}$ and $k_{\max}$ by variables `ni` and `nk`, and the propagation velocity $c$ by the variable `v`. The Courant number CN is represented by variable `a1`. The initial value $u_{0,\mathrm{h}}$ is represented by the function `p0`. Other input data $u_{1,\mathrm{h}}$ and $s_{\mathrm{h}}$ are supposed to be zero and are not represented. The discrete solution $p_{\mathrm{h}}$ is represented by the two-dimensional array `p` of size $(i_{\max}+1)(k_{\max}+1)$. Note that this is a simple implementation. A more efficient one would only store two time steps.

### 3.5. Program annotations

As explained in Section 2.2.2, the C code is enriched with *annotations*⋆ that specify the behavior of the program: what it requires on its inputs and what it ensures on its outputs. We describe here the chosen specification for this program. The full annotations are given in Appendix B.

The annotations can be separated into two sets. The first one corresponds to the mathematics: the exact solution of the wave equation and its properties. It defines the required values (the exact solution $p$, and its initialization $p_0$). It also defines the derivatives of $p$: `psol_1`, `psol_2`, `psol_11`, and `psol_22` respectively stand for $\frac{\partial p}{\partial x}$, $\frac{\partial p}{\partial t}$, $\frac{\partial^2 p}{\partial x^2}$, and $\frac{\partial^2 p}{\partial t^2}$. The value of the derivative of $f$ at point $x$ is defined as the limit of $\frac{f(x+h)-f(x)}{h}$ when $h \to 0$. As the ACSL annotations are only *first-order*⋆, these definitions are quite cumbersome: each derivative needs 5 lines to be defined. Here is the example of `psol_1`, *i.e.* $\frac{\partial p}{\partial x}$:

```
/*@ logic real psol_1(real x, real t);
  @ axiom psol_1_def:
  @   \forall real x; \forall real t;
  @   \forall real eps; \exists real C; 0 < C && \forall real dx;
  @   0 < eps ⇒ \abs(dx) < C ⇒
  @   \abs((psol(x + dx, t) − psol(x, t)) / dx − psol_1(x, t)) < eps;
  @ */
```

Note the different treatment of the positiveness for the existential variable `C`, and for the free variable `eps`.

[8] The big $O$ notation is defined in Section 4.1. The function $k_{\Delta t}$ is defined in Eq. (7).

Listing 1: The main part of the C code, without annotations.

```
/* Compute the constant coefficient of the stiffness matrix. */
a1 = dt/dx*v;
a  = a1*a1;

/* First initial condition and boundary conditions. */
/* Left boundary. */
p[0][0] = 0.;
/* Time iteration -1 = space loop. */
for (i=1; i<ni; i++) {
  p[i][0] = p0(i*dx);
}
/* Right boundary. */
p[ni][0] = 0.;

/* Second initial condition (with p1=0) and boundary conditions. */
/* Left boundary. */
p[0][1] = 0.;
/* Time iteration 0 = space loop. */
for (i=1; i<ni; i++) {
  dp = p[i+1][0] - 2.*p[i][0] + p[i-1][0];
  p[i][1] = p[i][0] + 0.5*a*dp;
}
/* Right boundary. */
p[ni][1] = 0.;

/* Evolution problem and boundary conditions. */
/* Propagation = time loop. */
for (k=1; k<nk; k++) {
  /* Left boundary. */
  p[0][k+1] = 0.;
  /* Time iteration k = space loop. */
  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;
  }
  /* Right boundary. */
  p[ni][k+1] = 0.;
}
```

We also assume that the solution actually solves Eqs. (1)–(5). The last property needed on the exact solution is its regularity. We require it to be close to its Taylor approximations of degrees 3 and 4 on the whole space interval (see Section 4). For instance, the following annotation states the property for degree 3:

```
/*@ axiom psol_suff_regular_3:
  @  0 < alpha_3 && 0 < C_3 &&
  @  \forall real x; \forall real t; \forall real dx; \forall real dt;
  @  0 ≤ x ≤ 1 ⇒ |sqrt(dx * dx + dt * dt) ≤ alpha_3 ⇒
  @  |abs(psol(x + dx, t + dt) - psol_Taylor_3(x, t, dx, dt)) ≤
  @    C_3 * |abs(|pow(|sqrt(dx * dx + dt * dt), 3));
  @ */
```

The second set of annotations corresponds to the properties and *loop invariant*⋆ needed by the program. For example, we require the matrix to be *separated*. This means that a line of the matrix should not overlap with another line. Otherwise a modification would alter another entry in the matrix. The predicate `analytic_error` that is used as a loop invariant is declared in the annotations and defined in the Coq files.

The initialization functions are only specified: there is no C code given and no proof. We assume a reasonable behavior for these functions. More precisely, this corresponds first to the function `array2d_alloc` that initializes the matrix and

*p_zero* that produces an approximation of the $p_0$ function. The use of *deductive verification*$^\star$ allows our proof to be generic with respect to $p_0$ and its implementation.

The preconditions of the main function are as follows:

- $i_{\max}$ and $k_{\max}$ must be greater than one, but small enough so that $i_{\max}+1$ and $k_{\max}+1$ do not overflow;
- the CFL($2^{-50}$) condition must be satisfied (see Eqs. (16) and (15));
- the grid sizes $\Delta x$ and $\Delta t$ must be small enough to ensure the convergence of the scheme (the precise value is given in Section 6.1);
- the floating-point values computed for the grid sizes must be close to their mathematical values;
- to prevent exceptional behavior in the computation of $a$, $\Delta t$ must be greater than $2^{-1000}$ and $\frac{c\Delta t}{\Delta x}$ must be greater than $2^{-500}$.

The last hypothesis gets very close to the underflow threshold: the smallest positive floating-point number is a subnormal of value $2^{-1074}$.

There are two postconditions, corresponding to a bound on the convergence error and a bound on the round-off error. See Sections 4 and 5 for more details.

## 4. Formal proof of convergence of the scheme

In [31], the method error is the distance between the discrete value computed with exact real arithmetic and the ideal mathematical value, *i.e.* the exact solution; here, it reduces to the convergence error. Round-off errors, due to the use of floating-point arithmetic, are handled in Section 5. First, we present the notions necessary to the *formal specification and proof*$^\star$ of boundedness of the method error. Then, taking inspiration from [26], we sketch the main steps of the proofs of consistency, stability, and convergence of the numerical scheme, and we point out some tricky aspects that may not be visible in pen-and-paper proofs. Full formal proofs are available from [7]. Feedback about issues showing up when formalizing a pen-and-paper proof is discussed in Section 7.1.

### 4.1. Big O, Taylor approximations, and regularity

Proving the consistency of the numerical scheme with the continuous equations requires some assumptions on the regularity of the exact solution. This regularity is expressed as the existence of Taylor approximations of the exact solution up to some appropriate order. For the sake of simplicity, we chose to prove uniform consistency of the scheme (see definitions at the end of Section 3.3); this implies the use of uniform Taylor approximations.

These uniform approximations involve *uniform big O* equalities of the form

$$F(\mathbf{x},\,\mathbf{\Delta x}) = O_{\Omega_{\mathbf{x}},\Omega_{\mathbf{\Delta x}}}(g(\mathbf{\Delta x})). \tag{25}$$

Variable $\mathbf{x}$ stands for points around which Taylor approximations are considered, and variable $\mathbf{\Delta x}$ stands for a change in $\mathbf{x}$ in the neighborhood of zero. Note that we explicitly state the set $\Omega_{\mathbf{x}}$ on which the approximation is uniform. As for the set $\Omega_{\mathbf{\Delta x}}$, we will later omit it when it spans the full space, here $\mathbb{R}^2$. Function $F$ goes from $\Omega_{\mathbf{x}}\times\Omega_{\mathbf{\Delta x}}$ to $\mathbb{R}$, and $g$ goes from $\Omega_{\mathbf{\Delta x}}$ to $\mathbb{R}$. As all equalities embedding big O notations, Eq. (25) actually means that function $\mathbf{\Delta x}\mapsto F(\mathbf{x},\,\mathbf{\Delta x})$ belongs to the set $O_{\Omega_{\mathbf{x}}}(g(\mathbf{\Delta x}))$ of functions of $\mathbf{\Delta x}$ with the same asymptotic growth rate as $g$, uniformly for all $\mathbf{x}$ in $\Omega_{\mathbf{x}}$.

The formal definition behind Eq. (25) is

$$\exists\alpha,\,C>0,\ \forall\mathbf{x}\in\Omega_{\mathbf{x}},\ \forall\mathbf{\Delta x}\in\Omega_{\mathbf{\Delta x}},\quad \|\mathbf{\Delta x}\|<\alpha\Rightarrow|F(\mathbf{x},\,\mathbf{\Delta x})|\le C|g(\mathbf{\Delta x})|. \tag{26}$$

Its translation in Coq is straightforward.[9]

```
Definition
  OuP (A : Type) (P : R * R → Prop) (f : A → R * R → R) (g : R * R → R) :=
exists alpha : R, exists C : R, 0 < alpha ∧ 0 < C ∧
forall X : A, forall dX : R * R, P dX → norm_l2 dX < alpha →
Rabs (f X dX) ≤ C * Rabs (g dX).
```

Argument type `A` stands for domain $\Omega_{\mathbf{x}}$, allowing to deal with either a subset of $\mathbb{R}^2$ or of $\mathbb{R}$. Domain $\Omega_{\mathbf{\Delta x}}$ is formalized by its indicator function on $\mathbb{R}^2$, the argument predicate `P` (`Prop` is the type of logical propositions).

Let $\mathbf{x}=(x,\,t)$ be a point in $\mathbb{R}^2$, let $f$ be a function from $\mathbb{R}^2$ to $\mathbb{R}$, and let $n$ be a natural number. The Taylor polynomial of order $n$ of $f$ at point $\mathbf{x}$ is the function $\mathrm{TP}_n(f,\mathbf{x})$ defined over $\mathbb{R}^2$ by

$$\mathrm{TP}_n(f,\mathbf{x})(\Delta x,\,\Delta t)\ \stackrel{\text{def}}{=}\ \sum_{p=0}^{n}\frac{1}{p!}\left(\sum_{m=0}^{p}\binom{p}{m}\frac{\partial^p f}{\partial x^m\partial t^{p-m}}(\mathbf{x})\cdot\Delta x^m\cdot\Delta t^{p-m}\right).$$

[9] `Rabs` is the absolute value of real numbers, and `norm_l2` is the $L^2$ norm on $\mathbb{R}^2$.

Let $\Omega_{\mathbf{x}}$ be a subset of $\mathbb{R}^2$. The Taylor polynomial is a *uniform approximation of order n of f on* $\Omega_{\mathbf{x}}$ when the following uniform big O equality holds[10]:

$$f(\mathbf{x}+\boldsymbol{\Delta}\mathbf{x}) - \mathrm{TP}_n(f,\mathbf{x})(\boldsymbol{\Delta}\mathbf{x}) = O_{\Omega_{\mathbf{x}}}\left(\|\boldsymbol{\Delta}\mathbf{x}\|^{n+1}\right).$$

Function $f$ is then *sufficiently regular of order n uniformly on* $\Omega_{\mathbf{x}}$ when all its Taylor polynomials of order smaller than $n$ are uniform approximations on $\Omega_{\mathbf{x}}$.

Of course, we could have expressed regularity requirements by using the usual notion of differentiability class. Indeed, it is well-known that functions of class $C^{n+1}$ over some compact domain are actually sufficiently regular of order $n$ uniformly on that domain. But, besides the suboptimality of the statement, we did not want to formalize all the technical details of its proof.

### 4.2. Consistency

The numerical scheme defined by Eqs. (9)–(13) is well-known to be of order 2, both in space and time. Furthermore, this result holds as soon as the exact solution admits Taylor approximations up to order 4. Therefore, we formally prove Lemma 4.1, where sufficient regularity is defined in Section 4.1, and uniform consistency is defined by Eq. (23).

**Lemma 4.1.** *If the exact solution of the wave equation defined by Eqs. (1)–(5) is sufficiently regular of order* 4 *uniformly on* $\Omega \times [0, t_{\max}]$*, then the numerical scheme defined by Eqs. (9)–(13) is consistent with the continuous problem at order* $(2, 2)$ *uniformly on interval* $[0, t_{\max}]$.

The proof uses properties of uniform Taylor approximations. It deals with long and complex expressions, but it is still straightforward as soon as discrete quantities are banned in favor of uniform continuous quantities that encompass them. Indeed, the key point is to use Taylor approximations that are uniform on a compact set including all points of all grids showing up when the discretization parameters tend to zero.

For instance, to handle the initialization phase of Eq. (10), we consider a uniform Taylor approximation of order 1 of the following function (for any $v$ sufficiently regular of order 3):

$$((x,t),(\Delta x,\Delta t)) \mapsto \frac{v(x,t+\Delta t)-v(x,t)}{\Delta t} - \frac{\Delta t}{2}c^2\frac{v(x+\Delta x,t)-2v(x,t)+v(x-\Delta x,t)}{\Delta x^2}.$$

Thanks to the second-order approximation used in Eq. (10), the initialization phase does not deteriorate the consistency order.

Note that, to provide Taylor approximations for both expressions $v(x+\Delta x,t)$ and $v(x-\Delta x,t)$, the formal definition of the uniform Taylor approximation via Eq. (25) must accept negative changes in $\mathbf{x}$, and not only the positive space grid steps, as a naive specification would state.

### 4.3. Stability

When showing convergence of a numerical scheme, energy-based techniques use a specific stability statement involving an estimation of the discrete energy. Therefore, we formally prove Lemma 4.2 where the CFL($\xi$) condition is defined by Eqs. (15) and (16).

**Lemma 4.2.** *For all* $\xi \in ]0, 1[$*, if discretization parameters satisfy the* CFL($\xi$) *condition, then the numerical scheme defined by Eqs. (9)–(13) is energetically stable uniformly on interval* $[0, t_{\max}]$*. Moreover, constants in Eq. (24) defining uniform energy stability are*

$$C_1 = \sqrt{E_{\mathrm{h}}(c)(p_{\mathrm{h}})^{\frac{1}{2}}} \quad \textit{and} \quad C_2 = \frac{1}{\sqrt{2\xi(2-\xi)}}. \tag{27}$$

First, we show the variation of the discrete energy between two consecutive time steps to be proportional to the source term. In particular, the discrete energy is constant when the source is inactive. Then, we establish the following lower bound for the discrete energy:

$$\frac{1}{2}(1-\mathrm{CN}^2)\left\|\left(i \mapsto \frac{p_i^{k+1}-p_i^k}{\Delta t}\right)\right\|_{\Delta x}^2 \leq E_{\mathrm{h}}(c)(p_{\mathrm{h}})^{k+\frac{1}{2}}$$

where CN is the Courant number defined by Eq. (15), and $k = k_{\Delta t}(t)$ for some $t$ in $[0, t_{\max}]$. Therefore, the discrete energy is a positive semidefinite quadratic form, since its nonnegativity directly follows the CFL($\xi$) condition. Finally, Lemma 4.2

[10] Domain $\Omega_{\boldsymbol{\Delta}\mathbf{x}}$ is implicitly considered to be $\mathbb{R}^2$.

follows in a straightforward manner from the estimate:

$$\left\| (i \mapsto p_i^{k+1} - p_i^k) \right\|_{\Delta x} \leq 2C_2 \Delta t E_{\mathrm{h}}(c)(p_{\mathrm{h}})^{k+\frac{1}{2}}$$

where $C_2$ is given in Eq. (27).

Note that this stability result is valid for any discrete Cauchy data $u_{0,\mathrm{h}}$ and $u_{1,\mathrm{h}}$, so it may be suboptimal for specific choices.

### 4.4. Convergence

We formally prove Theorem 4.1, where sufficient regularity is defined in Section 4.1, the CFL($\xi$) condition is defined by Eqs. (15) and (16), and uniform convergence is defined by Eq. (22).

**Theorem 4.1.** *For all $\xi \in ]0, 1[$, if the exact solution of the wave equation defined by Eqs. (1)–(5) is sufficiently regular of order 4 uniformly on $\Omega \times [0, t_{\max}]$, and if the discretization parameters satisfy the* CFL($\xi$) *condition, then the numerical scheme defined by Eqs. (9)–(13) is convergent of order $(2, 2)$ uniformly on interval $[0, t_{\max}]$.*

First, we prove that the convergence error $e_{\mathrm{h}}$ is itself solution of the same numerical scheme but with different input data. (Of course, there is no associated continuous problem here.) In particular, the source term on the right-hand side of Eq. (9) is here the truncation error $\varepsilon_{\mathrm{h}}$ associated with the numerical scheme for $p_{\mathrm{h}}$. Then, from Lemma 4.2 (stating uniform energy stability), we have an estimation of the discrete energy associated with the convergence error $E_{\mathrm{h}}(c)(e_{\mathrm{h}})$ that involves the sum of the corresponding source terms, *i.e.* the truncation error. Finally, from Lemma 4.1 (stating uniform consistency), this sum is shown to have the same growth rate as $\Delta x^2 + \Delta t^2$.

## 5. Formal proof of boundedness of the round-off error

Beyond the proof of convergence of the numerical scheme, which is here mechanically checked, we can now go into the sketch of the main steps of the proof of the boundedness of the round-off error due to the limited precision of computations in the program. Full formal proofs are available from [7]. Feedback about the novelty of this part is discussed in Section 7.1.

For the C program presented in Section 3, naive *forward error analysis*⋆ gives an error bound that is proportional to $2^{k-53}$ for the computation of any $p_i^k$. If this bound was tight, it would cause the numerical scheme to compute only noise after a few steps. Fortunately, round-off errors actually cancel themselves in the program. To take into account cancellations and hence prove a usable error bound, we need a precise statement of the round-off error [32] to exhibit the cancellations made by the numerical scheme.

The formal proof uses a comprehensive formalization of the IEEE-754 standard, and hence both normal and subnormal (*i.e.* very small) numbers are handled. Other floating-point special values (infinities, NaNs) are proved in Section 6.2 not to appear in the program.

### 5.1. Local round-off errors

Let $\delta_i^k$ be the local floating-point error that occurs during the computation of $p_i^k$. For $k = 0$ (resp. $k = 1$, and $k \geq 2$), the local error corresponds to the floating-point error at Line 9 of Listing 1 (resp. at Lines 19–20, and at Lines 32–33). To distinguish them from the discrete values of previous sections, the floating-point values as computed by the program are underlined below. Quantities $\underline{a}$ and $\underline{p_i^k}$ match the expressions `a` and `p[i][k]` in the *annotations*⋆, while $a$ and $p_i^k$ are represented by `\exact(a)` and `\exact(p[i][k])`, as described in Section 2.2.2.

The local floating-point error is defined as follows, for all $k$ in $[1..k_{\max} - 1]$, for all $i$ in $[1..i_{\max} - 1]$:

$$\begin{aligned}
\delta_i^{k+1} &= \left(2\underline{p_i^k} - \underline{p_i^{k-1}} + a(\underline{p_{i+1}^k} - 2\underline{p_i^k} + \underline{p_{i-1}^k})\right) - \underline{p_i^{k+1}},\\
\delta_i^1 &= \left(\underline{p_i^0} + \frac{a}{2}(\underline{p_{i+1}^0} - 2\underline{p_i^0} + \underline{p_{i-1}^0})\right) - \underline{p_i^1} - \left(\delta_i^0 + \frac{a}{2}\left(\delta_{i+1}^0 - 2\delta_i^0 + \delta_{i-1}^0\right)\right),\\
\delta_i^0 &= p_i^0 - \underline{p_i^0}.
\end{aligned}$$

Note that the program presented in Section 3.4 gives us that, for all $k$ in $[1..k_{\max} - 1]$, for all $i$ in $[1..i_{\max} - 1]$:

$$\begin{aligned}
\underline{p_i^{k+1}} &= \mathrm{fl}\left(2\underline{p_i^k} - \underline{p_i^{k-1}} + a(\underline{p_{i+1}^k} - 2\underline{p_i^k} + \underline{p_{i-1}^k})\right),\\
\underline{p_i^1} &= \mathrm{fl}\left(\underline{p_i^0} + \frac{a}{2}(\underline{p_{i+1}^0} - 2\underline{p_i^0} + \underline{p_{i-1}^0})\right),\\
\underline{p_i^0} &= \mathrm{fl}\left(p_0(i\underline{\Delta x})\right)
\end{aligned}$$

where $\mathrm{fl}(\cdot)$ means that all the arithmetic operations that appear between the parentheses are actually performed by floating-point arithmetic, hence a bit off.

In order to get a bound on $\delta_i^k$, we need to know the range of $\underline{p}_i^k$. For the bound to be usable in our correctness proof, the range has to be $[-2, 2]$. We assume here that the exact solution is bounded by 1; if not, we normalize by the maximum value and use the linearity of the problem. We have proved this range by induction on a simple triangle inequality taking advantage of the fact that at each point of the grid, the floating-point value computed by the program is the sum of the exact solution, the method error, and the round-off error.

To prove the bound on $\delta_i^k$, we perform forward error analysis and then use *interval arithmetic*⋆ to bound each intermediate error [24]. We prove that, for all $i$ and $k$, we have $|\delta_i^k| \leq 78 \cdot 2^{-52}$ for a reasonable error bound for $a$, that is to say $|\underline{a} - a| \leq 2^{-49}$.

### 5.2. Global round-off error

Let $\Delta_i^k = \underline{p}_i^k - p_i^k$ be the global floating-point error on $p_i^k$. The global floating-point error depends not only on the local floating-point error at position $(i, k)$, but also on all the local floating-point errors inside the space–time dependency cone of apex $(i, k)$.

From the linearity of the numerical scheme, the global floating-point error is itself solution of the same numerical scheme with discrete inputs corresponding to the local floating-point error:

$$u_{0,\mathrm{h}} = \delta_{\mathrm{h}}^0 = 0, \qquad u_{1,\mathrm{h}} = \frac{\delta_{\mathrm{h}}^1}{\Delta t}, \qquad s_{\mathrm{h}} = \left(k \mapsto \frac{\delta_{\mathrm{h}}^{k+1}}{\Delta t^2}\right).$$

Taking advantage of the remark about image theory made in Section 3.2, we see the global floating-point error as the restriction to the domain $\Omega$ of the solution of the same numerical scheme set on the entire real axis without the Dirichlet boundary condition of Eq. (12), and with extended discrete inputs. Therefore, the expression of the global floating-point error is given by the convolution of the discrete fundamental solution on the entire real axis and the extended local floating-point error.

Let us denote by $\lambda_{\mathrm{h}}$ the discrete fundamental solution. It is the solution of the same numerical scheme set on the entire real axis with null discrete inputs except $u_{1,0} = \frac{1}{\Delta t}$. Then, we can state the following result. (See [32] for a direct proof that does not follow above remarks.)

**Theorem 5.1.**

$$\forall k \geq 0,\ \forall i \in [0..i_{\max}], \quad \Delta_i^k = \sum_{l=0}^{k} \sum_{j=-l}^{l} \tilde{\delta}_{i-j}^{k-l}\, \lambda_j^{l+1}.$$

Note that, for all $k$, we have $\Delta_0^k = \Delta_{i_{\max}}^k = 0$. Note also that $\lambda_i^k$ vanishes as soon as $|i| \geq k$, hence the sum over $j$ could be replaced by an infinite sum over all integers.

### 5.3. Bound on the global round-off error

The analytic expression of $\Delta_i^k$ can be used to obtain a bound on the round-off error. We will need two lemmas for this purpose.

**Lemma 5.1.**

$$\forall k \geq 0, \quad \sigma^k \stackrel{\text{def}}{=} \sum_{i=-\infty}^{+\infty} \lambda_i^k = k.$$

**Proof.** The sum satisfies the following linear recurrence: for all $k \geq 1$, $\sigma^{k+1} - 2\sigma^k + \sigma^{k-1} = 0$. Since $\sigma^0 = 0$, and $\sigma^1 = 1$, we have, for all $k \geq 0$, $\sigma^k = k$. □

**Lemma 5.2.**

$$\forall k \geq 0,\ \forall i, \quad \lambda_i^k \geq 0.$$

The sketch of the proof is given in Appendix C. It uses complex algebraic results (the positivity of sums of Jacobi polynomials), and was not formalized in Coq.

**Theorem 5.2.**

$$\forall k \geq 0,\ \forall i \in [0..i_{\max}], \quad \left|\Delta_i^k\right| \leq 78 \cdot 2^{-53}(k+1)(k+2).$$

**Proof.** According to Theorem 5.1, $\Delta_i^k$ is equal to $\sum_{l=0}^{k} \sum_{j=-l}^{l} \lambda_j^{l+1}\ \tilde{\delta}_{i-j}^{k-l}$. We know from Section 5.1 that for all $j$ and $l$, $|\tilde{\delta}_j^l| \leq 78 \cdot 2^{-52}$, and from Lemma 5.1 that $\sum \lambda_i^{l+1} = l + 1$. Since the $\lambda$'s are nonnegative (Lemma 5.2), the error is easily bounded by $78 \cdot 2^{-52} \sum_{l=0}^{k} (l+1)$. □

Except for Lemma 5.2, all proofs described in this section have been machine-checked using Coq. In particular, the proof of the bound on $\delta_i^k$ was done automatically by calling Gappa from Coq. Lemma 5.2 is a technical detail compared to the rest of our work, and is not worth the tremendous work it would require to be implemented in Coq: keen results on integrals but also definitions and results about the Legendre, Laguerre, Chebyshev, and Jacobi polynomials (see Appendix C).

## 6. Formal proof of the C program

To completely prove the actual C program, we bound the total error incorporating both the method error and the round-off error. Then, we show how we prove the absence of coding errors (such as overflow and out-of-bound access) and how we relate the behavior of the program to the preceding Coq proofs.

### 6.1. Total error

Let $\mathcal{E}_\mathrm{h}$ be the total error. It is the sum of the method error (or convergence error) $e_\mathrm{h}$ of Sections 3.3 and 4.4, and of the round-off error $\Delta_\mathrm{h}$ of Section 5.

From Theorem 5.2, we can estimate[11] the following upper bound for the spatial norm of the round-off error when $\Delta x \leq 1$ and $\Delta t \leq t_{\max}/2$: for all $t \in [0, t_{\max}]$,

$$\begin{aligned}\left\|\left(i \mapsto \Delta_i^{k_{\Delta t}(t)}\right)\right\|_{\Delta x} &= \sqrt{\sum_{i=0}^{i_{\max}} \left(\Delta_i^{k_{\Delta t}(t)}\right)^2 \Delta x} \\ &\leq \sqrt{(i_{\max}+1)\Delta x} \cdot 78 \cdot 2^{-53} \left(\frac{t_{\max}}{\Delta t}+1\right)\left(\frac{t_{\max}}{\Delta t}+2\right) \\ &\leq \sqrt{x_{\max}-x_{\min}+1} \cdot 78 \cdot 2^{-53} \cdot 3\frac{t_{\max}^2}{\Delta t^2}.\end{aligned}$$

Thus, from the triangle inequality for the spatial norm, we obtain the following estimation of the total error:

$$\forall t \in [0, t_{\max}],\ \forall \mathbf{\Delta x},\quad \|\mathbf{\Delta x}\| \leq \min(\alpha_e, \alpha_\Delta) \Rightarrow \left\|\left(i \mapsto \mathcal{E}_i^{k_{\Delta t}(t)}\right)\right\|_{\Delta x} \leq C_e(\Delta x^2 + \Delta t^2) + \frac{C_\Delta}{\Delta t^2} \tag{28}$$

where the method error constants $\alpha_e$ and $C_e$ were extracted from the Coq proof (see Section 4.4) and are given in terms of the constants for the Taylor approximation of the exact solution at degree 3 ($\alpha_3$ and $C_3$), and at degree 4 ($\alpha_4$ and $C_4$) by

$$\alpha_e = \min(1, t_{\max}, \alpha_3, \alpha_4), \qquad C_e = 4C_2 t_{\max}\sqrt{x_{\max}-x_{\min}}\left(\frac{C'}{\sqrt{2}} + 2C_2(t_{\max}+1)C''\right) \tag{29}$$

with $C_2$ coming from Eq. (27), $C' = \max(1, C_3 + c^2C_4 + 1)$, $C'' = \max(C', 2(1+c^2)C_4)$, and where the round-off constants $\alpha_\Delta$ and $C_\Delta$, as explained above, are given by

$$\alpha_\Delta = \min(1, t_{\max}/2), \qquad C_\Delta = 234 \cdot 2^{-53} t_{\max}^2 \sqrt{x_{\max}-x_{\min}+1}. \tag{30}$$

Of course, decreasing the size of the grid step decreases the method error, but at the same time, it increases the round-off error. Therefore, there exists a minimum for the upper bound on the total error, corresponding to optimal grid step sizes that may be determined using above formulas.

On specific examples, one observes that this upper bound on the total error can be highly overestimated (see Section 7.1). Nevertheless, one also observes that the asymptotic behavior of the upper bound of the total error for high values of the grid steps is close to the asymptotic behavior of the effective total error [6].

### 6.2. Automation and manual proofs

Given the program code, the Frama-C/Jessie/Why tools generate 150 *verification conditions*⋆ that have to be proved (see Section 2.2.2). While possible, proving all of them in Coq would be rather tedious. Moreover, systematically using Coq would lead to a rather fragile construct: any later modification to the program, however small it is, would cause different proof obligations to be generated, which would then require additional human interaction to adapt the Coq proofs. We prefer to have *automated theorem provers*⋆ (*SMT solvers*⋆ and Gappa, see Sections 2.2.2 and 2.2.3) prove as many of them as possible, so that only the most intricate ones are left to be proved in Coq.

[11] When $\tau \geq 2$, we have $(\tau+1)(\tau+2) \leq 3\tau^2$.

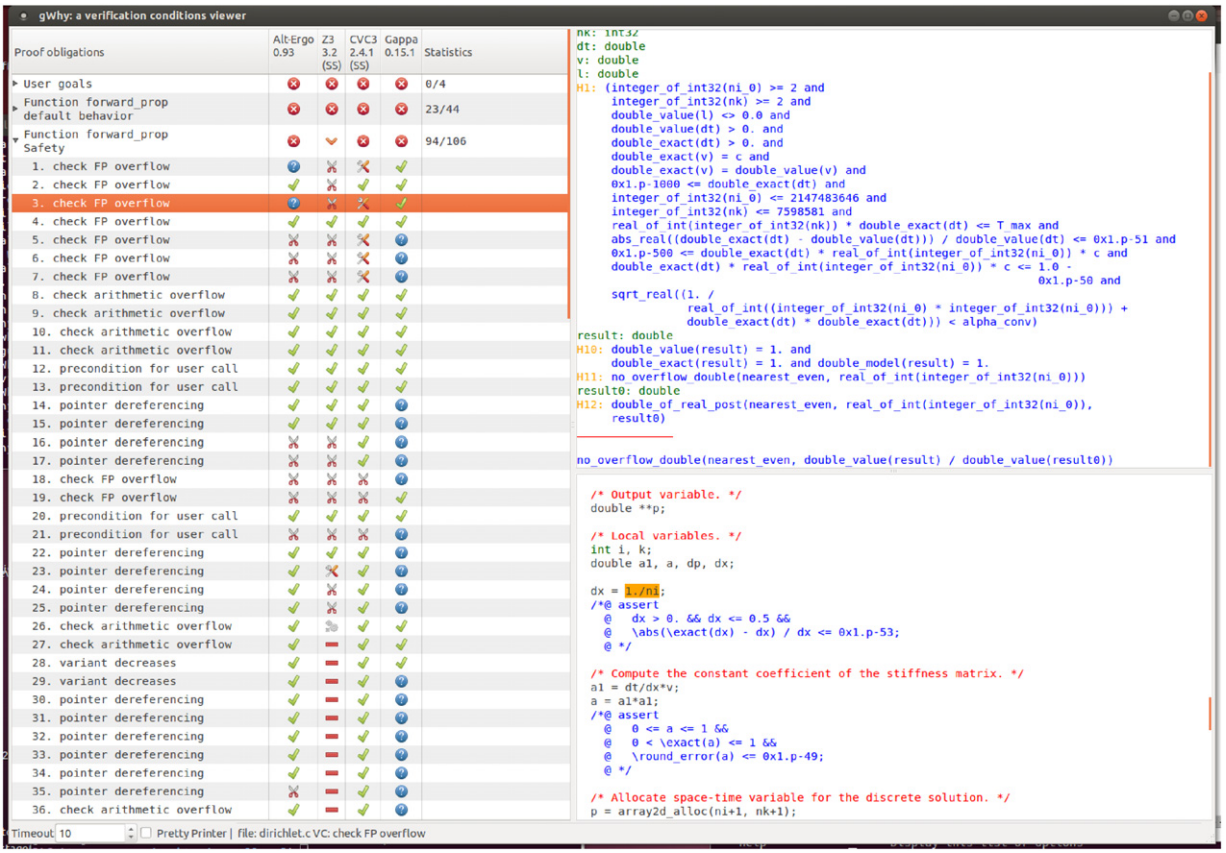



**Fig. 1.** Screenshot of gWhy. Left: list of all verification conditions (VCs) and their proof status with respect to the various automatic tools. Upper right: statement of the currently selected VC. Lower right: location in the source code where this VC originates from.

Fig. 1 displays a screenshot of the gWhy graphical interface. For instance, any discrepancy between the code and its *specification*⋆ would be highlighted on the lower right part when the corresponding verification condition is selected. A comprehensive view on all the verification conditions and how they are proved can be found in [7].

Verification conditions split into safety *goals*⋆ and behavior goals. Safety goals are always generated, even in the absence of any specification. Proving them ensures that the program always successfully terminates. They check that matrix accesses are in range, that the loop variants decrease and are nonnegative (thus loops terminate), that integer and floating-point arithmetic operations do not overflow, and so on. On such goals, automatic provers are helpful, as they prove about 90% of them.

Behavior goals relate the program to its specification. Proving them ensures that, if the program terminates without error, then it returns the specified result. They check that *loop invariants*⋆ are preserved, that assertions hold, that preconditions hold before function calls, that postconditions are implied by preconditions, and so on. Automatic provers are a bit less helpful, as they fail for half of the behavior goals. That is why we resort to an *interactive higher-order theorem prover*⋆, namely Coq.

Coq developments split into two sets of files. The first set, for a total of more than 4000 lines of script, is dedicated to the proof of the boundedness of the method error, *i.e.* the convergence of the numerical scheme. About half of it is generic, meaning it can be re-used for other PDEs or ODEs: definitions and lemmas about big O, scalar product, $\mathbb{R}^n$ and so on. The other half is dedicated to the 1D wave equation and the chosen numerical scheme. The second set, for a total of more than 12,000 lines, deals with both the proof of the boundedness of the round-off error, and the proof of the absence of runtime errors. Note that slightly less than half of those Coq developments are statements of the second set that are automatically generated by the Frama-C/Jessie/Why framework, whereas the other half – statements for the convergence and all proofs – is manually edited.

## 7. Discussion

We emphasize now the difficulties and the originality of this work that may strike the computational scientist's mind when confronted to the formal proof of a numerical program. Finally, we discuss some future work.

### 7.1. Applied mathematicians and formal proof of programs

From the point of view of computational scientists, the big surprise of a *formal proof*⋆ development may come from the requirement of writing an extremely detailed conventional pen-and-paper proof. Indeed, an *interactive proof assistant*⋆ such as Coq is not able to elaborate a sketch of proof, nor to automatically find a proof of a lemma—but for the most trivial facts. Therefore, a formal proof is completely human driven, up to the utmost detail. Fortunately, we do not need to specify all the mathematical facts from scratch, since Coq provides a collection of known facts and theories that can be reused to establish new results. However, classical results from Hilbert space analysis, uniform asymptotic comparison of functions, and Taylor approximations have not yet been proved in Coq. Thus, we had to develop them, as well as in the detailed pen-and-paper proof. In the end, the detailed pen-and-paper proof is about 50 pages long. Surprisingly, it is roughly of the same size as the complete Coq formal proof: both collections of source files are approximately 5 kLOC long, and weigh about 160 kB.

The case of the big O notation deserves some comments. Typically, this is a notion about which pen-and-paper proofs do not elaborate on. Of course, misinterpreting the involved existential constants leads to erroneous reasoning. This issue is seldom dealt with in the literature; however, we point out an informal mention in [33] with a relevant remark about pointwise consistency versus norm consistency. We had to focus on the uniform notion of big O equalities in [34] in the context of the infinite string, for which the space domain $\Omega$ is the whole real axis. In the present paper, we deal with finite strings. Thus, for compactness reasons, both uniform and nonuniform notions of big O notations are obviously equivalent. Nevertheless, we still use the more general uniform big O notion to share most of the proof developments between finite and infinite cases.

With pen-and-paper proofs, it is uncommon to expand constants involved in the estimation of the method error (namely $\alpha$, and $C$ in the big O equality for the norm of the convergence error). In contrast, Coq can automatically extract from the formal proof the constants appearing in the upper bound for the total error, which also takes the round-off error into account. The mathematical expressions given in Section 6.1 are as sharp as the estimations used in the proof can provide. The same expressions could be obtained by hand with extreme care from the pen-and-paper proof, at least for the method error contribution. However, in specific examples, the actual total error may be much smaller than the established estimation [6]. This is essentially due to the use of *a priori* error estimations, worsened by the energy-based techniques which are known to become less precise as the CFL condition becomes optimal (*i.e.* as the Courant number goes up to 1). Indeed, constant $C_e$ in Eq. (29) grows as $1/\xi$ for small values of $\xi$. An alternative to obtain more accurate *a priori* bounds is to use a leapfrog scheme for the equivalent first-order system [26].

Nowadays, *a posteriori* error estimates have become popular tools to control the method error, *e.g.* see [35,1]. Indeed, the measure at runtime of the error due to the different stages of approximation provides much tighter bounds than any *a priori* technique. For instance, the approach allows to adapt stopping criteria of internal iterative solvers to the overall numerical error, and to save a significant amount of computations [36]. From the certification point of view, such developments present themselves as part of the numerical method, as a complement to the numerical scheme: *a posteriori* error estimates are first designed and proved on paper, then implemented into the program, and finally make use of floating-point operators in their computations. They deserve exactly the same care in terms of code *verification*⋆: first, their definition and behavior must be specified in the *annotations*⋆, and then the generated *verification conditions*⋆ must be formally proved.

The other big surprise for computational scientists is the possibility to take into account the round-off error cancellations occurring during the computations, and to evaluate a sensible bound that encompasses all aspects of the IEEE-754 floating-point arithmetic. Of course, this is typically the kind of results that are not accessible outside the logical framework of a mechanical interactive proof assistant as Coq.

Furthermore, in numerical analysis, one usually evaluates bounds on the absolute value of quantities of interest. To obtain an upper bound on the round-off error, we needed a result about the sign of the fundamental solution of the discrete scheme, not about a bound for its absolute value. To our best knowledge, the only way to capture such a result is purely algebraic: the closed-form expression of the solution is first obtained using a generating function, then it may be recognized as a combination of Jacobi polynomials that happens to be nonnegative (see Appendix C).

Such a result about the sign of the discrete fundamental solution may not hold for other numerical schemes. In our case, just assuming a bound on the absolute value of the discrete fundamental solution would lead to an estimation of the global round-off error about 4 times higher than that of Theorem 5.2.

Finally, differential equations introduce an issue in the annotations. Due to the underlying logic, the annotations have to define the solution of the PDE by using *first-order formulas*⋆ stating differentiability, instead of second-order formulas involving differential operators. This makes the annotations especially tedious and verbose. In the end, the C code grows by a factor of about 4, as can be seen when comparing the original source in Listing 1 with the fully annotated source in Appendix B.

### 7.2. Future work

Given its cost, we do not plan to apply the same methodology to any other scheme or any other PDE at random. We nevertheless want to take advantage of our acquired expertise on several related topics.

In floating-point arithmetic, an algorithm is said to be *numerically stable* if it has a small *forward error*⋆. This property is certainly related to the notion of stability used in scientific computing for numerical schemes, which states that computed values do not diverge. As a rule of thumb, stable schemes are considered numerically stable. We plan to formally prove this statement for a large class of numerical schemes. This will require the formalization of the notion of numerical scheme, and of the property of stability. Indeed, the definition given in Section 3.3 is dedicated to numerical schemes for second-order in time evolution problems.

The mechanism used to extract the constants involved in the estimation of the total error can be deployed on a wider scale to generate the whole program. Instead of formally proving properties specified in an existing program, one formalizes a problem, proves it has a solution, and obtains for free an OCaml program by automatic extraction from the Coq proof. Such a program is usually inefficient, but it is zero-defect, provided it is not modified afterwards. For computational science problems involving real numbers, efficiency is not the only issue: the extracted program would also be hindered by the omnipresence of classical real numbers in the formalization. Therefore, being able to extract efficient and usable zero-defect programs would be an interesting long-term goal for critical numerical problems.

In the present exploratory work, we consider the simple second-order centered finite difference scheme for the one-dimensional wave equation. Further works involve scaling to higher-order numerical schemes, and/or to higher-dimensional problems. Finite support functions, and summations on such support played a much more important role in the Coq developments than we initially expected. Therefore, we consider using the SSReflect interface and libraries for Coq [37], so as to simplify manipulations of these objects in higher-dimensional cases.

Another long-term perspective is to generalize the mathematical notions formalized in Coq to be able to apply the approach to finite difference schemes for other PDEs, and ODEs. Steps in this direction concern the generalization of the use of convolution with the discrete fundamental solution to bound the round-off error, and the formal proof of the Lax equivalence theorem.

Finally, a more ambitious perspective is to formally deal with the finite element method. This first requires a Coq formalization of mathematical tools from Hilbert space analysis on Sobolev spaces such as the Lax–Milgram theorem (for existence and uniqueness of exact and approximated solutions), and the handling of meshes (that may be regular or not, and structured or not). Then, the finite element method may be proved convergent in order to formally guarantee a large class of numerical analysis programs that solve linear PDEs on complex geometries. The purpose is to go beyond Laplace's equation set on the unit square, *e.g.* handle mixed boundary conditions, and extend to mixed and mixed hybrid finite element methods.

## 8. Conclusion

We have presented a comprehensive formal proof of a C program solving the 1D acoustic wave equation using the second-order centered finite difference explicit scheme. Our proof includes the following aspects.

- *Safety*: we prove that the C program terminates and is free of runtime failure such as division by zero, array access out of bounds, null pointer dereference, or arithmetic overflow. The latter includes both integer and floating-point overflows.
- *Method error*: we show that the numerical scheme is convergent of order (2, 2) uniformly on the time interval, under the assumptions that the exact solution is sufficiently regular (*i.e.* its Taylor polynomials of order up to 4 are uniform approximations on the space–time domain) and that the Courant–Friedrichs–Lewy condition holds.
- *Round-off error*: we bound all round-off errors resulting from floating-point computations used in the program. In particular, we show how some round-off errors cancel, to eventually get a meaningful bound.
- *Total error*: altogether, we are able to provide explicit (and formally proved) bounds for the sum of method and round-off errors.

To our knowledge, this is the first time such a comprehensive proof is achieved. Related to the small length of the program (a few tens of lines), the total cost of the formal proof is huge, if not frightening: several man-months of work, three times more annotations to be inserted in the program than lines of code, over 16,000 lines of Coq proof scripts, 30 min of CPU time to check them on a 3-GHz processor.

The recent introduction of formal methods in DO-178C[12] shows the need for the verification of numerical programs in the context of embedded critical software. Considering the work we have presented, one can hardly think of verifying numerical codes on the scale of a large airborne system. Yet we think our techniques and a large subset of our proof can be reused and would significantly decrease the workload of such a proof. This is to be combined with increased proof automation, so that user interaction is minimized. Finally, the challenge is to provide tools that are usable by computational scientists that are not specialists of formal methods.

[12] DO-178C is the newest version of Software Considerations in Airborne Systems and Equipment Certification, which is used by national certification authorities.

In conclusion, combining scientific computing and formal proofs is now considered an important matter by logicians. Formal tools for scientific computing are being actively developed and progress is done to get them mature and usable for non specialists. It seems that it is now time for computational scientists to take a keen interest in this area.

**Acknowledgments**

We are grateful to Manuel Kauers, Veronika Pillwein, and Bruno Salvy, who provided us help with the nonnegativity of the fundamental solution of the discrete wave equation (Lemma 5.2). We are also thankful to Vincent Martin for his constructive remarks on this article. Last, we feel indebted to the reviewers for their invaluable suggestions.

## Appendix A. Glossary

This section gives the definition of concepts used in mathematical logic and computer science that appear in this paper.

**annotation** comment added to the C code to specify the logical properties of the program. Tools turn them into *verification conditions*⋆.

**atom** atomic component of a logical formula. In the setting of classical logic, any propositional formula can be rewritten using literals (atoms or negated atoms), conjunctions, and disjunctions only.

**automated theorem prover** software tool that automatically proves *goals*⋆. It may fail to find a proof, even for valid formulas.

**decision procedure** an algorithm dedicated to proving specific properties. This is one of the basic blocks of theorem provers.

**deductive verification** process of verifying, with the help of theorem provers, that a program satisfies its *specification*⋆.

**first-order logic** formal language of logical formulas that use quantifications over values only, and not over predicates and functions.

**formal proof** a finite sequence of deduction steps which are checked by a computer.

**forward error analysis** process of propagating error bounds from inputs to outputs of functions.

**functional programming** programming paradigm that treats computation as mathematical evaluation of functions and considers functions as ordinary values.

**goal** a set of hypotheses and a logical formula. The proof of a goal is a way to deduce the logical formula from the hypotheses. When proving a theorem, the statement of the theorem is the initial goal.

**higher-order logic** formal language of logical formulas that use quantifications over values, predicates and functions.

**inference rule** generic way of drawing a valid conclusion based on the form of hypotheses. Each deduction step of a *formal proof*⋆ must satisfy one of the inference rules of the logical system.

**interactive proof assistant** software tool to assist with the development of *formal proofs*⋆ by human–machine collaboration.

**interval arithmetic** arithmetic that operates on sets of values (typically intervals) instead of values.

**loop invariant** logical formula about the state of a program, that is valid before entering a loop and remains valid at the end of each iteration of the loop.

**semantics** meaning associated to each syntactic construct of a language.

**SMT solver** variety of *automated theorem prover*⋆ combining a SAT solver (propositional logic), equality reasoning, *decision procedures*⋆ (*e.g.*, for linear arithmetic), and quantifier instantiation.

**specification** description of the expected behavior of a program.

**static analysis** analysis of a program without executing it.

**straight-line program** program that does not contain any loop and thus does not need any *loop invariant*⋆ to be verified.

**tactic** command for an *interactive proof assistant*⋆ to transform the current *goal*⋆ into one or more goals that imply it.

**validation** process of making sure of the correctness of a program by experiments such as tests.

**verification** process of making sure of the correctness of a program by mathematical means such as *formal proofs*⋆.

**verification conditions** *goals*⋆ that need to be proved to guarantee the adequacy between the program and its *specification*⋆.

## Appendix B. Fully annotated source code

```
0  /*@ axiomatic dirichlet_maths {
     @
     @ logic real c;
     @ logic real p0(real x);
     @ logic real psol(real x, real t);
5
     @ axiom c_pos: 0 < c;
```

```
@ logic real psol_1(real x, real t);
@ axiom psol_1_def:
@   \forall real x; \forall real t;
@   \forall real eps; \exists real C; 0 < C && \forall real dx;
@   0 < eps ⇒ \abs(dx) < C ⇒
@   \abs((psol(x + dx, t) − psol(x, t)) / dx − psol_1(x, t)) < eps;

@ logic real psol_11(real x, real t);
@ axiom psol_11_def:
@   \forall real x; \forall real t;
@   \forall real eps; \exists real C; 0 < C && \forall real dx;
@   0 < eps ⇒ \abs(dx) < C ⇒
@   \abs((psol_1(x + dx, t) − psol_1(x, t)) / dx − psol_11(x, t)) < eps;

@ logic real psol_2(real x, real t);
@ axiom psol_2_def:
@   \forall real x; \forall real t;
@   \forall real eps; \exists real C; 0 < C && \forall real dt;
@   0 < eps ⇒ \abs(dt) < C ⇒
@   \abs((psol(x, t + dt) − psol(x, t)) / dt − psol_2(x, t)) < eps;

@ logic real psol_22(real x, real t);
@ axiom psol_22_def:
@   \forall real x; \forall real t;
@   \forall real eps; \exists real C; 0 < C && \forall real dt;
@   0 < eps ⇒ \abs(dt) < C ⇒
@   \abs((psol_2(x, t + dt) − psol_2(x, t)) / dt − psol_22(x, t)) < eps;

@ axiom wave_eq_0: \forall real x; 0 ≤ x ≤ 1 ⇒ psol(x, 0) = p0(x);
@ axiom wave_eq_1: \forall real x; 0 ≤ x ≤ 1 ⇒ psol_2(x, 0) = 0;
@ axiom wave_eq_2:
@   \forall real x; \forall real t;
@   0 ≤ x ≤ 1 ⇒ psol_22(x, t) − c ∗ c ∗ psol_11(x, t) = 0;
@ axiom wave_eq_dirichlet_1: \forall real t; psol(0, t) = 0;
@ axiom wave_eq_dirichlet_2: \forall real t; psol(1, t) = 0;

@ logic real psol_Taylor_3(real x, real t, real dx, real dt);
@ logic real psol_Taylor_4(real x, real t, real dx, real dt);

@ logic real alpha_3; logic real C_3;
@ logic real alpha_4; logic real C_4;

@ axiom psol_suff_regular_3:
@   0 < alpha_3 && 0 < C_3 &&
@   \forall real x; \forall real t; \forall real dx; \forall real dt;
@   0 ≤ x ≤ 1 ⇒ \sqrt(dx ∗ dx + dt ∗ dt) ≤ alpha_3 ⇒
@   \abs(psol(x + dx, t + dt) − psol_Taylor_3(x, t, dx, dt)) ≤
@     C_3 ∗ \abs(\pow(\sqrt(dx ∗ dx + dt ∗ dt), 3));

@ axiom psol_suff_regular_4:
@   0 < alpha_4 && 0 < C_4 &&
@   \forall real x; \forall real t; \forall real dx; \forall real dt;
@   0 ≤ x ≤ 1 ⇒ \sqrt(dx ∗ dx + dt ∗ dt) ≤ alpha_4 ⇒
@   \abs(psol(x + dx, t + dt) − psol_Taylor_4(x, t, dx, dt)) ≤
@     C_4 ∗ \abs(\pow(\sqrt(dx ∗ dx + dt ∗ dt), 4));

@ axiom psol_le:
@   \forall real x; \forall real t;
@   0 ≤ x ≤ 1 ⇒ 0 ≤ t ⇒ \abs(psol(x, t)) ≤ 1;

@ logic real T_max;
```

```
  @ axiom T_max_pos: 0 < T_max;
  @
  @ logic real C_conv; logic real alpha_conv;
  @ lemma alpha_conv_pos: 0 < alpha_conv;
  @
  @ } */

/*@ axiomatic dirichlet_prog {
  @
  @ predicate analytic_error{L}
  @   (double **p, integer ni, integer i, integer k, double a, double dt)
  @   reads p [..][..];
  @
  @ lemma analytic_error_le{L}:
  @   \forall double **p; \forall integer ni; \forall integer i;
  @   \forall integer nk; \forall integer k;
  @   \forall double a; \forall double dt;
  @   0 < ni ⇒ 0 ≤ i ≤ ni ⇒ 0 ≤ k ⇒
  @   0 < \exact(dt) ⇒
  @   analytic_error(p, ni, i, k, a, dt) ⇒
  @   \sqrt(1. / (ni * ni) + \exact(dt) * \exact(dt)) < alpha_conv ⇒
  @   k ≤ nk ⇒ nk ≤ 7598581 ⇒ nk * \exact(dt) ≤ T_max ⇒
  @   \exact(dt) * ni * c ≤ 1 − 0x1.p−50 ⇒
  @   \forall integer i1; \forall integer k1;
  @   0 ≤ i1 ≤ ni ⇒ 0 ≤ k1 < k ⇒
  @   \abs(p[i1][k1]) ≤ 2;
  @
  @ predicate separated_matrix{L}(double **p, integer leni) =
  @   \forall integer i; \forall integer j;
  @   0 ≤ i < leni ⇒ 0 ≤ j < leni ⇒ i != j ⇒
  @   \base_addr(p[i]) != \base_addr(p[j]);
  @
  @ logic real sqr_norm_dx_conv_err{L}
  @  (double **p, real dx, real dt, integer ni, integer i, integer k)
  @  reads p [..][..];
  @ logic real sqr(real x) = x * x;
  @ lemma sqr_norm_dx_conv_err_0{L}:
  @  \forall double **p; \forall real dx; \forall real dt;
  @  \forall integer ni; \forall integer k;
  @  sqr_norm_dx_conv_err(p, dx, dt, ni, 0, k) = 0;
  @ lemma sqr_norm_dx_conv_err_succ{L}:
  @  \forall double **p; \forall real dx; \forall real dt;
  @  \forall integer ni; \forall integer i; \forall integer k;
  @  0 ≤ i ⇒
  @  sqr_norm_dx_conv_err(p, dx, dt, ni, i + 1, k) =
  @    sqr_norm_dx_conv_err(p, dx, dt, ni, i, k) +
  @    dx * sqr(psol(1. * i / ni, k * dt) − \exact(p[i][k]));
  @ logic real norm_dx_conv_err{L}
  @  (double **p, real dt, integer ni, integer k) =
  @  \sqrt(sqr_norm_dx_conv_err(p, 1. / ni, dt, ni, ni, k));
  @
  @ } */

/*@ requires leni ≥ 1 && lenj ≥ 1;
  @ ensures
  @  \valid_range(\result, 0, leni − 1) &&
  @  (\forall integer i; 0 ≤ i < leni ⇒
  @   \valid_range(\result[i], 0, lenj − 1)) &&
  @  separated_matrix(\result, leni);
  @ */
double **array2d_alloc(int leni, int lenj);
```

```
/*@ requires (l != 0) && \round_error(x) ≤ 5./2*0x1.p−53;
  @ ensures
  @   \round_error(\result) ≤ 14 * 0x1.p−52 &&
  @   \exact(\result) = p0(\exact(x));
  @ */
double p_zero(double xs, double l, double x);

/*@ requires
  @   ni ≥ 2 && nk ≥ 2 && l != 0 &&
  @   dt > 0. && \exact(dt) > 0. &&
  @   \exact(v) = c && \exact(v) = v &&
  @   0x1.p−1000 ≤ \exact(dt) &&
  @   ni ≤ 2147483646 && nk ≤ 7598581 &&
  @   nk * \exact(dt) ≤ T_max &&
  @   \abs(\exact(dt) − dt) / dt ≤ 0x1.p−51 &&
  @   0x1.p−500 ≤ \exact(dt) * ni * c ≤ 1 − 0x1.p−50 &&
  @   \sqrt(1. / (ni * ni) + \exact(dt) * \exact(dt)) < alpha_conv;
  @
  @ ensures
  @   \forall integer i; \forall integer k;
  @   0 ≤ i ≤ ni ⇒ 0 ≤ k ≤ nk ⇒
  @   \round_error(\result[i][k]) ≤ 78. / 2 * 0x1.p−52 * (k + 1) * (k + 2);
  @
  @ ensures
  @   \forall integer k; 0 ≤ k ≤ nk ⇒
  @   norm_dx_conv_err(\result, \exact(dt), ni, k) ≤
  @     C_conv * (1. / (ni * ni) + \exact(dt) * \exact(dt));
  @ */
double **forward_prop(int ni, int nk, double dt, double v,
                      double xs, double l) {

  /* Output variable. */
  double **p;

  /* Local variables. */
  int i, k;
  double a1, a, dp, dx;

  dx = 1./ni;
  /*@ assert
    @   dx > 0. && dx ≤ 0.5 &&
    @   \abs(\exact(dx) − dx) / dx ≤ 0x1.p−53;
    @ */

  /* Compute the constant coefficient of the stiffness matrix. */
  a1 = dt/dx*v;
  a = a1*a1;
  /*@ assert
    @   0 ≤ a ≤ 1 &&
    @   0 < \exact(a) ≤ 1 &&
    @   \round_error(a) ≤ 0x1.p−49;
    @ */

  /* Allocate space−time variable for the discrete solution. */
  p = array2d_alloc(ni+1, nk+1);

  /* First initial condition and boundary conditions. */
  /* Left boundary. */
  p[0][0] = 0.;
  /* Time iteration −1 = space loop. */
```

```
  /*@ loop invariant
    @  1 ≤ i ≤ ni &&
    @  analytic_error(p, ni, i − 1, 0, a, dt);
    @ loop variant ni − i; */
  for (i=1; i<ni; i++) {
    p[i][0] = p_zero(xs, l, i*dx);
  }
  /* Right boundary. */
  p[ni][0] = 0.;
  /*@ assert analytic_error(p, ni, ni, 0, a, dt); */

  /* Second initial condition (with p_one=0) and boundary conditions. */
  /* Left boundary. */
  p[0][1] = 0.;
  /* Time iteration 0 = space loop. */
  /*@ loop invariant
    @  1 ≤ i ≤ ni &&
    @  analytic_error(p, ni, i − 1, 1, a, dt);
    @ loop variant ni − i; */
  for (i=1; i<ni; i++) {
    /*@ assert \abs(p[i−1][0]) ≤ 2; */
    /*@ assert \abs(p[i][0]) ≤ 2; */
    /*@ assert \abs(p[i+1][0]) ≤ 2; */
    dp = p[i+1][0] − 2.*p[i][0] + p[i−1][0];
    p[i][1] = p[i][0] + 0.5*a*dp;
  }
  /* Right boundary. */
  p[ni][1] = 0.;
  /*@ assert analytic_error(p, ni, ni, 1, a, dt); */

  /* Evolution problem and boundary conditions. */
  /* Propagation = time loop. */
  /*@ loop invariant
    @  1 ≤ k ≤ nk &&
    @  analytic_error(p, ni, ni, k, a, dt);
    @ loop variant nk − k; */
  for (k=1; k<nk; k++) {
    /* Left boundary. */
    p[0][k+1] = 0.;
    /* Time iteration k = space loop. */
    /*@ loop invariant
      @  1 ≤ i ≤ ni &&
      @  analytic_error(p, ni, i − 1, k + 1, a, dt);
      @ loop variant ni − i; */
    for (i=1; i<ni; i++) {
      /*@ assert \abs(p[i−1][k]) ≤ 2; */
      /*@ assert \abs(p[i][k]) ≤ 2; */
      /*@ assert \abs(p[i+1][k]) ≤ 2; */
      /*@ assert \abs(p[i][k−1]) ≤ 2; */
      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;
    }
    /* Right boundary. */
    p[ni][k+1] = 0.;
    /*@ assert analytic_error(p, ni, ni, k + 1, a, dt); */
  }

  return p;

}
```

## Appendix C. Fundamental solution and Jacobi polynomials

Let $\lambda$ be the sequence defined in Section 5.2. Note that adding zero initial values for the fictitious time step $k = -1$ makes this sequence be a time shift by one of the fundamental solution of the discrete acoustic wave equation, associated with the input data $s_{\mathrm{h}} \equiv 0$, $u_{0,\mathrm{h}} \equiv 0$, for all $i$, $i \neq 0$, $u_{1,i} = 0$, and $u_{1,0} = 1$. The items of the sequence satisfy the following equations:

$$\forall i, \quad \lambda_i^{-1} = 0, \tag{C.1}$$

$$\forall i \neq 0, \quad \lambda_i^0 = 0, \qquad \lambda_0^0 = 1, \tag{C.2}$$

$$\forall i, \forall k \geq 0, \quad \lambda_i^{k+1} = a(\lambda_{i-1}^k + \lambda_{i+1}^k) + 2(1-a)\lambda_i^k - \lambda_i^{k-1}. \tag{C.3}$$

We want to prove Lemma 5.2, *i.e.* that for all $i$, $k$, $k \geq 0$, we have $\lambda_i^k \geq 0$.

The proof is highly indebted to computer algebra: we use a generating function to obtain a closed-form expression for the $\lambda$'s, the creative telescoping method of Zeilberger [38] to express those $\lambda$'s in terms of Jacobi polynomials, and finally a result by Askey and Gasper [39] to ensure the nonnegativity. We have not mechanically checked this proof. For example, the Askey and Gasper result would have required enormous Coq developments, but parts of it could have been formalized, in particular Zeilberger's algorithm provides a certificate that eases the *verification*⋆ of its result.

Consider the associated bivariate generating function formally defined by

$$\Lambda(X,T) = \sum_i \sum_{k\geq -1} \lambda_i^k X^i T^k.$$

The above recurrence relation in Eq. (C.3) rewrites

$$\sum_i \sum_{k\geq 0} \lambda_i^{k+1} X^i T^k = a\left(\sum_i \sum_{k\geq 0} \lambda_{i-1}^k X^i T^k + \sum_i \sum_{k\geq 0} \lambda_{i+1}^k X^i T^k\right) + 2(1-a)\sum_i \sum_{k\geq 0} \lambda_i^k X^i T^k - \sum_i \sum_{k\geq 0} \lambda_i^{k-1} X^i T^k.$$

Since coefficients for $k$'s smaller than 1 are almost all equal to zero, we formally have

$$T\sum_i \sum_{k\geq 0} \lambda_i^{k+1} X^i T^k = \Lambda(X,T) - 1, \qquad \sum_i \sum_{k\geq 0} \lambda_{i-1}^k X^i T^k = X\Lambda(X,T),$$

$$X\sum_i \sum_{k\geq 0} \lambda_{i+1}^k X^i T^k = \Lambda(X,T), \qquad \sum_i \sum_{k\geq 0} \lambda_i^k X^i T^k = \Lambda(X,T), \qquad \sum_i \sum_{k\geq 0} \lambda_i^{k-1} X^i T^k = T\Lambda(X,T).$$

Therefore, the generating function satisfies

$$\frac{\Lambda(X,T)-1}{T} = a\left(X + \frac{1}{X}\right)\Lambda(X,T) + 2(1-a)\Lambda(X,T) - T\Lambda(X,T),$$

which solution is given by

$$\Lambda(X,T) = \frac{1}{(1-T)^2\left[1 - a\frac{T}{X}\left(\frac{1-X}{1-T}\right)^2\right]}.$$

Now, we can evaluate the power series expansion of the generating function $\Lambda$. Using the following power series expansion, valid for $|u| < 1$,

$$\frac{1}{(1-u)^{p+1}} = \sum_{n\geq 0} \binom{p+n}{p} u^n,$$

and the properties of binomial coefficients, we successively have

$$\begin{aligned}
\Lambda(X,T) &= \frac{1}{(1-T)^2} \sum_{n\geq 0} a^n \frac{T^n}{X^n} \left(\frac{1-X}{1-T}\right)^{2n} \\
&= \sum_{n\geq 0} a^n \sum_{i=0}^{i=2n} \binom{2n}{i} (-1)^i X^{i-n} \sum_{k\geq 0} \binom{2n+1+k}{2n+1} T^{n+k} \\
&= \sum_i X^i \sum_{k\geq 0} T^k \sum_{n=|i|}^{n=k} \binom{2n}{n+i} \binom{n+k+1}{2n+1} (-1)^{n+i} a^n.
\end{aligned}$$

Finally, by identification of the two power series expansions of the generating function $\Lambda$, we have for all $i$, $k$, $0 \leq |i| \leq k$,

$$\lambda_i^k = \sum_{n=|i|}^{n=k} \binom{2n}{n+i} \binom{n+k+1}{2n+1} (-1)^{n+i} a^n. \tag{C.4}$$

For $i = 0$, sharp eyes may recognize that $\lambda_0^k = \sum_{n=0}^{k} P_n(1-2a)$ where the $P_n$'s are Legendre polynomials. Fejér showed in [40] the nonnegativity of the sum of Legendre polynomials when the argument is in $[-1, 1]$, which is satisfied here since we consider $0 < a < 1$. More generally, we have $\lambda_i^k = a^{|i|} \sum_{n=0}^{k-|i|} P_n^{(2|i|,0)}(1-2a)$ where the $P_n^{(\alpha,\beta)}$'s are Jacobi polynomials. Askey and Gasper generalized in [39,41] Fejér's result for $\beta \geq 0$ and $\alpha + \beta \geq -2$. See also [42], pages 314 and 384.

Indeed, from the definition of Jacobi polynomials, for all $\alpha, \beta > -1$, for all $n \in \mathbb{N}$, for all $x \in [-1, 1]$,

$$P_n^{(\alpha,\beta)}(x) = \sum_{p=0}^{n} \binom{n+\alpha}{p} \binom{n+\beta}{n-p} \left(\frac{x+1}{2}\right)^p \left(\frac{x-1}{2}\right)^{n-p},$$

we have, for all $i, k,\ 0 \leq i \leq k$,

$$\begin{aligned}
a^i \sum_{n=0}^{k-i} P_n^{(2i,0)}(1-2a) &= a^i \sum_{n=0}^{k-i} \sum_{p=0}^{n} \binom{n+2i}{p} \binom{n}{n-p} (1-a)^p (-a)^{n-p} \\
&= \sum_{n=i}^{k} \sum_{p=0}^{n-i} \sum_{q=0}^{p} \binom{n+i}{p} \binom{n-i}{p} \binom{p}{q} (-1)^{n-p+q+i} a^{n-p+q} \\
&= \sum_{n=i}^{k} \sum_{p=i}^{n} \sum_{q=p}^{n} \binom{n+i}{n-p} \binom{n-i}{n-p} \binom{n-p}{n-q} (-1)^{q+i} a^q \\
&= \sum_{n=i}^{k} \sum_{p=i}^{n} \sum_{q=n}^{k} \binom{q+i}{p+i} \binom{q-i}{p-i} \binom{q-p}{n-p} (-1)^{n+i} a^n.
\end{aligned}$$

We have successively shifted $n$ by $i$, replaced $n - p$ by $p$, then shifted $q$ by $p$. To obtain the last equality, notice that the previous triple sum is actually taken over $\{(n, p, q) \in \mathbb{N}^3 / i \leq p \leq q \leq n \leq k\}$, hence we can take $q$ in $[i..k]$, $p$ in $[i..q]$, $n$ in $[q..k]$, and then switch notations $n$ and $q$, and use the symmetry of binomial coefficients. Identifying with the expression of Eq. (C.4), we are led to prove, for all $i, n, k,\ 0 \leq i \leq n \leq k$,

$$\sum_{p=i}^{n} \sum_{q=n}^{k} \binom{q+i}{p+i} \binom{q-i}{p-i} \binom{q-p}{n-p} = \binom{2n}{n+i} \binom{n+k+1}{2n+1}. \tag{C.5}$$

Suppose we have the following identity, for all $i, n, k, 0 \leq i \leq n \leq k$,

$$\sum_{p=i}^{n} \binom{k+i}{p+i} \binom{k-i}{p-i} \binom{k-p}{n-p} = \binom{2n}{n+i} \binom{k+n}{2n}. \tag{C.6}$$

Then, we would have

$$\sum_{p=i}^{n} \sum_{q=n}^{k} \binom{q+i}{p+i} \binom{q-i}{p-i} \binom{q-p}{n-p} = \sum_{q=n}^{k} \binom{2n}{n+i} \binom{q+n}{2n} = \binom{2n}{n+i} \sum_{q=2n}^{k+n} \binom{q}{2n} = \binom{2n}{n+i} \binom{k+n+1}{2n+1}.$$

The last equality comes directly from the recurrence relation for the binomial coefficients (column-sum property of Pascal's triangle).

Proving identity of Eq. (C.6) is a bit more technical. The hypergeometric nature of its terms makes it a good candidate for Zeilberger's algorithm, a.k.a. the method of creative telescoping, see [43,44]. Let us introduce some new notations, for all $i, n, k,\ 0 \leq i \leq n \leq k$,

$$\begin{aligned}
F(i, n, k; p) &= \binom{k+i}{p+i} \binom{k-i}{p-i} \binom{k-p}{n-p}, \\
f(i, n, k) &= \sum_{p} F(i, n, k; p), \\
g(i, n, k) &= \binom{2n}{n+i} \binom{k+n}{2n}.
\end{aligned}$$

Note that $F$ vanishes when $p$ is outside the interval $[i..n]$. Thus, identity of Eq. (C.6) now writes $f = g$.

Let $I$ (resp. $N$, $K$, and $P$) be the forward shift operator in $i$ (resp. in $n$, $k$, and $p$). E.g., $If(i, n, k) = f(i+1, n, k)$. We first assume that the function $f$ satisfies the following first-order recurrence relations, for all $i, n, k,\ 0 \leq i \leq n \leq k$,

$$(n+1+i)If = (n-i)f, \tag{C.7}$$

$$(n+1+i)(n+1-i)Nf = (k+1+n)(k-n)f, \tag{C.8}$$

$$(k+1-n)Kf = (k+1+n)f. \tag{C.9}$$

Then, it is easy to show that the function $g$ satisfies exactly the same first-order recurrence relations, and since $f(0,0,0) = g(0,0,0) = 1$, we have $f = g$. Indeed, simply using the symmetry and absorption properties of binomial coefficients, we have, for all $i, n, k,\ 0 \le i \le n \le k$,

$$\frac{g(i+1,n,k)}{g(i,n,k)} = \frac{\binom{2n}{n+i+1}}{\binom{2n}{n+i}} = \frac{n-i}{n+1+i},$$

$$\frac{g(i,n+1,k)}{g(i,n,k)} = \frac{\binom{2n+2}{n+1+i}\binom{k+n+1}{2n+2}}{\binom{2n}{n+i}\binom{k+n}{2n}} = \frac{k+1+n}{n+1+i}\frac{\binom{2n+1}{n+i}}{\binom{2n}{n+i}}\frac{\binom{k+n}{2n+1}}{\binom{k+n}{2n}} = \frac{(k+1+n)(k-n)}{(n+1+i)(n+1-i)},$$

$$\frac{g(i,n,k+1)}{g(i,n,k)} = \frac{\binom{k+1+n}{2n}}{\binom{k+n}{2n}} = \frac{k+1+n}{k+1-n}.$$

Finding the first-order recurrence relations for $f$, *i.e.* Eqs. (C.7), (C.8), and (C.9), is the job of the method of creative telescoping. Actually, Zeilberger's algorithm provides recurrence relations for the hypergeometric summand $F$ of the form

$$\sum_{m=0}^{m'} b_{l,m} L^m F = (P-1)(R_l F)$$

where coefficients $b_{l,m}$ are polynomials independent of $p$, and $R_l$ is a rational function. There are actually one such recurrence relation per free variable $l$ (here $i$, $n$, and $k$), and $L$ is the generic forward shift operator in the generic free variable $l$. Thus, since coefficients $b_{l,m}$ do not depend on $p$, when summing over $p$, the right-hand terms telescope, and we can deduce similar recurrence relations for the sum $f$

$$\sum_{m=0}^{m'} b_{l,m} L^m f = 0.$$

Note that although those recurrence relations are difficult to obtain, and even to check on the sum $f$, they are easy to check on the summand $F$ (since there is no more sum over $p$). One has just to check the simpler equations with rational expressions

$$b_{l,0} + \sum_{m=1}^{m'} b_{l,m} \frac{L^m F}{F} = P R_l \frac{PF}{F} - R_l. \tag{C.10}$$

In the present case, Zeilberger's algorithm provides first-order recurrence relations with

$$b_{i,0} = n-i, \qquad b_{n,0} = (k+1+n)(k-n), \qquad b_{k,0} = k+1+n$$
$$b_{i,1} = -(n+1+i), \qquad b_{n,1} = -(n+1+i)(n+1-i), \qquad b_{k,1} = -(k+1-n),$$
$$R_i = \frac{(1+2i)(p-i)}{k-i}, \qquad R_n = \frac{(k-n)(p+i)(p-i)}{n+1-p}, \qquad R_k = \frac{(p+i)(p-i)}{k+1-p}.$$

And, simply using the symmetry and absorption properties of binomial coefficients, Eq. (C.10) is successively equivalent to, for $l = i$,

$$\begin{aligned}
(\text{C.10}) \Leftrightarrow\ & (n-i) - (n+1+i)\frac{(k+1+i)}{(p+1+i)}\frac{(p-i)}{(k-i)} \\
&= \frac{(1+2i)(p+1-i)}{(k-i)}\frac{(k-p)}{(p+1+i)}\frac{(k-p)}{(p+1-i)}\frac{(n-p)}{(k-p)} - \frac{(1+2i)(p-i)}{(k-i)} \\
\Leftrightarrow\ & (n-i)(k-i)(p+1+i) - (n+1+i)(k+1+i)(p-i) \\
&= (1+2i)(k-p)(n-p) - (1+2i)(p-i)(p+1+i) \\
\Leftrightarrow\ & \frac{(1+2i)}{4}\left[(2n+1)(2k+1) - (2n+1)(2p+1) - (2k+1)(2p+1) + (1+2i)^2\right] \\
&= (1+2i)\left[(k-p)(n-p) - (p-i)(p+1+i)\right] \\
\Leftrightarrow\ & \frac{(1+2i)^2}{4} - \frac{(2p+1)^2}{4} = -(p-i)(p+1+i),
\end{aligned}$$

for $l = n$,

$$\begin{aligned}
(\text{C.10}) &\Leftrightarrow (k+1+n)(k-n) - (n+1+i)(n+1-i)\frac{(k-n)}{(n+1-p)} \\
&= \frac{(k-n)(p+1+i)(p+1-i)}{(n-p)}\frac{(k-p)}{(p+1+i)}\frac{(k-p)}{(p+1-i)}\frac{(n-p)}{(k-p)} - \frac{(k-n)(p+i)(p-i)}{(n+1-p)} \\
&\Leftrightarrow (n+1+k)(n+1-p) - (n+1+i)(n+1-i) \\
&\quad = (k-p)(n+1-p) - (p+i)(p-i) \\
&\Leftrightarrow (n+1)(k-p) - kp + i^2 = (k-p)(n+1) - kp + i^2,
\end{aligned}$$

and for $l = k$,

$$\begin{aligned}
(\text{C.10}) &\Leftrightarrow (k+1+n) - (k+1-n)\frac{(k+1+i)}{(k+1-p)}\frac{(k+1-i)}{(k+1-p)}\frac{(k+1-p)}{(k+1-n)} \\
&= \frac{(p+1+i)(p+1-i)}{(k-p)}\frac{(k-p)}{(p+1+i)}\frac{(k-p)}{(p+1-i)}\frac{(n-p)}{(k-p)} - \frac{(p+i)(p-i)}{(k+1-p)} \\
&\Leftrightarrow (k+1+n)(k+1-p) - (k+1+i)(k+1-i) \\
&\quad = (n-p)(k+1-p) - (p+i)(p-i) \\
&\Leftrightarrow (k+1)(n-p) - np + i^2 = (n-p)(k+1) - np + i^2,
\end{aligned}$$

which are all true.

## References


[1] W.L. Oberkampf, C.J. Roy, Verification and Validation in Scientific Computing, Cambridge University Press, 2010.
[2] M. Mayero, Formalisation et Automatisation de Preuves en analyses réelle et Numérique (Ph.D. Thesis), Université Paris VI, 2001.
[3] S. Boldo, G. Melquiond, Flocq: a unified library for proving floating-point algorithms in Coq, in: Antelo, E., Hough, D., Ienne, P. (Eds.), 20th IEEE Symposium on Computer Arithmetic, 2011, pp. 243–252.
[4] F. de Dinechin, C. Lauter, G. Melquiond, Certifying the floating-point implementation of an elementary function using Gappa, IEEE Trans. Comput. 60 (2) (2011) 242–253.
[5] S. Boldo, C. Lelay, G. Melquiond, Improving real analysis in Coq: a user-friendly approach to integrals and derivatives, in: C. Hawblitzel, D. Miller (Eds.), Proceedings of the Second International Conference on Certified Programs and Proofs, in: LNCS, vol. 7679, Springer, 2012, pp. 289–304.
[6] S. Boldo, F. Clément, J.-C. Filliâtre, M. Mayero, G. Melquiond, P. Weis, Wave equation numerical resolution: a comprehensive mechanized proof of a C program, J. Automat. Reason. 50 (4) (2013) 423–456.
[7] S. Boldo, F. Clément, J.-C. Filliâtre, M. Mayero, G. Melquiond, P. Weis, Annotated source code and Coq proofs, 2012. http://fost.saclay.inria.fr/wave_total_error.html.
[8] X. Leroy, Formal verification of a realistic compiler, Commun. ACM 52 (7) (2009) 107–115.
[9] S. Boldo, J.-H. Jourdan, X. Leroy, G. Melquiond, A formally-verified C compiler supporting floating-point arithmetic, in: Proceedings of the 21th IEEE Symposium on Computer Arithmetic, 2013, pp. 107–115.
[10] G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, S. Winwood, seL4: formal verification of an operating system kernel, Commun. ACM 53 (6) (2010) 107–115.
[11] Y. Bertot, P. Castéran, Interactive theorem proving and program development, in: Coq'Art: The Calculus of Inductive Constructions, in: Texts in Theoretical Computer Science, Springer, 2004.
[12] T. Coquand, C. Paulin-Mohring, Inductively defined types, in: P. Martin-Löf, G. Mints (Eds.), Colog'88, in: LNCS, vol. 417, Springer-Verlag, 1990.
[13] M. Daumas, L. Rideau, L. Théry, A generic library for floating-point numbers and its application to exact computing, in: Proceedings of the 14th International Conference on Theorem Proving in Higher-Order Logics, (TPHOL'01), Springer-Verlag, 2001, pp. 169–184.
[14] S. Boldo, Preuves Formelles en Arithmétiques à virgule Flottante, (Ph.D. thesis) École Normale Supérieure de Lyon, 2004.
[15] Microprocessor Standards Committee, 2008. IEEE Standard for Floating-Point Arithmetic. IEEE Std. 754-2008, pp. 1–58.
[16] D. Goldberg, What every computer scientist should know about floating-point arithmetic, ACM Comput. Surv. 23 (1) (1991) 5–48.
[17] C. Marché, Jessie: an intermediate language for Java and C verification, in: Programming Languages meets Program Verification (PLPV), ACM, 2007, pp. 1–2.
[18] P. Baudin, P. Cuoq, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy, V. Prevosto, ACSL: ANSI/ISO C specification language, version 1.5., 2009. http://frama-c.com/acsl.html.
[19] J.-C. Filliâtre, C. Marché, The Why/Krakatoa/Caduceus platform for deductive program verification, in: 19th International Conference on Computer Aided Verification, in: LNCS, vol. 4590, Springer, 2007, pp. 173–177.
[20] E.W. Dijkstra, Guarded commands, nondeterminacy and formal derivation of programs, Commun. ACM 18 (8) (1975) 453–457.
[21] S. Conchon, E. Contejean, J. Kanig, S. Lescuyer, CC(X): semantical combination of congruence closure with solvable theories, in: Post-Proceedings of the 5th International Workshop on Satisfiability Modulo Theories, (SMT 2007), in: Electronic Notes in Computer Science, vol. 198 (2), Elsevier Science Publishers, 2008, pp. 51–69.
[22] C. Barrett, C. Tinelli, CVC3, in: 19th International Conference on Computer Aided Verification, (CAV'07), in: LNCS, vol. 4590, Springer-Verlag, Berlin, Germany, 2007, pp. 298–302.
[23] L. de Moura, N. Bjørner, Z3, an efficient SMT solver, in: Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, (TACAS'08), in: Lecture Notes in Computer Science, vol. 4963, Springer, 2008, pp. 337–340.
[24] M. Daumas, G. Melquiond, Certification of bounds on expressions involving rounded operators, Trans. Math. Software 37 (1) (2010) 1–20.
[25] S. Boldo, J.-C. Filliâtre, G. Melquiond, Combining Coq and Gappa for certifying floating-point programs, in: J. Carette, L. Dixon, C.S. Coen, S.M. Watt (Eds.), 16th Calculemus Symposium, in: LNAI, vol. 5625, 2009, pp. 59–74.
[26] E. Bécache, Étude de schémas numériques pour la résolution de l'équation des ondes. Master Modélisation et simulation, Cours ENSTA, 2009. http://www.ensta-paristech.fr/~becache/COURS-ONDES/Poly-num-0209.pdf.
[27] J. le Rond D'Alembert, Recherches sur la courbe que forme une corde tendue mise en vibrations, in: Histoire de l'Académie Royale des Sciences et Belles Lettres, in: (Année 1747), vol. 3, Haude et Spener, Berlin, 1749, pp. 214–249.
[28] F. John, Partial Differential Equations, Springer, 1986.
[29] J.C. Strikwerda, Finite Difference Schemes and Partial Differential Equations, Chapman & Hall, 1989.

[30] R. Courant, K. Friedrichs, H. Lewy, On the partial difference equations of mathematical physics, IBM J. Res. Dev. 11 (2) (1967) 215–234.
[31] S. Boldo, J.-C. Filliâtre, Formal verification of floating-point programs, in: 18th IEEE International Symposium on Computer Arithmetic, 2007, pp. 187–194.
[32] S. Boldo, Floats & ropes: a case study for formal numerical program verification, in: 36th International Colloquium on Automata, Languages and Programming, in: LNCS—ARCoSS, vol. 5556, Springer, 2009, pp. 91–102.
[33] J.W. Thomas, Numerical Partial Differential Equations: Finite Difference Methods, in: Texts in Applied Mathematics, vol. 22, Springer, 1995.
[34] S. Boldo, F. Clément, J.-C. Filliâtre, M. Mayero, G. Melquiond, P. Weis, Formal proof of a wave equation resolution scheme: the method error, in: M. Kaufmann, L.C. Paulson (Eds.), 1st Conference on Interactive Theorem Proving Conference, (ITP 2010), in: LNCS, vol. 6172, Springer, 2010, pp. 147–162.
[35] R. Verfürth, A Review of a Posteriori Error Estimation and Adaptive Mesh-Refinement Techniques, Teubner-Wiley, Stuttgart, Germany, 1996.
[36] A. Ern, M. Vohralík, Adaptive inexact Newton methods with a posteriori stopping criteria for nonlinear diffusion PDEs, SIAM J. Sci. Comput. 35 (4) (2014) A1761–A1791.
[37] Y. Bertot, G. Gonthier, S. Ould Biha, I. Pasca, Canonical big operators, in: 21st International Conference on Theorem Proving in Higher Order Logics, (TPHOLs'08), in: LNCS, vol. 5170, Springer, 2008, pp. 86–101.
[38] M. Petkovšek, H.S. Wilf, D. Zeilberger, A = B, A K Peters Ltd., Wellesley, MA, 1996, http://www.cis.upenn.edu/~wilf/AeqB.html.
[39] R. Askey, G. Gasper, Certain rational functions whose power series have positive coefficients, Amer. Math. Monthly 79 (1972) 327–341.
[40] L. Fejér, Sur le développement d'une fonction arbitraire suivant les fonctions de Laplace, C. R. Acad. Sci. 146 (1908) 224–227.
[41] G. Gasper, Positive sums of the classical orthogonal polynomials, SIAM J. Math. Anal. 8 (3) (1977) 423–447.
[42] G.E. Andrews, R. Askey, R. Roy, Special Functions, Cambridge University Press, Cambridge, 1999.
[43] D. Zeilberger, A fast algorithm for proving terminating hypergeometric identities, Discrete Math. 80 (1990) 207–211.
[44] D. Zeilberger, The method of creative telescoping, J. Symbolic Comput. 11 (1991) 195–204.