Trusting Computations: a Mechanized Proof from Partial Differential Equations to Actual Program

Sylvie Boldo, François Clément, Jean-Christophe Filliâtre, Micaela Mayero, Guillaume Melquiond, Pierre Weis

Trusting Computations: a Mechanized Proof from Partial Differential Equations to Actual Program

Sylvie Boldo, François Clément, Jean-Christophe Filliâtre, Micaela Mayero, Guillaume Melquiond, Pierre Weis

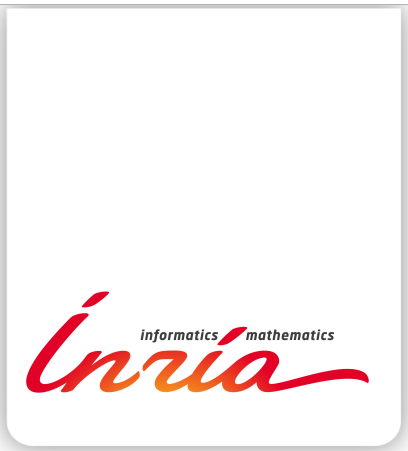
RESEARCH REPORT

N° 8197

December 2012

Project-Teams Pomdapi et Toccata

ISSN 0249-6399 ISRN INRIA/RR--8197--FR+ENG



Trusting Computations: a Mechanized Proof from Partial Differential Equations to Actual Program

Sylvie Boldo*†, François Clément‡, Jean-Christophe Filliâtre†*, Micaela Mayero§, Guillaume Melquiond*†, Pierre Weis‡

Project-Teams Pomdapi et Toccata

Research Report n° 8197 — December 2012 — 35 pages

Abstract: It is well-known that programs may fail 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. There exist methods and tools for proving the correct behavior of programs and we have extended them for the verification of 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.

Key-words: Acoustic wave equation, Formal proof of numerical program, Convergence of numerical scheme, Rounding error analysis

This research was supported by the ANR projects CerPAN (ANR-05-BLAN-0281-04) and Fφst (ANR-08-BLAN-0246-01).

* Projet Toccata. {Sylvie.Boldo,Jean-Christophe.Filliatre,Guillaume.Melquiond}@inria.fr.
† LRI, UMR 8623, Université Paris-Sud, CNRS, Orsay cedex, F-91405.
‡ Projet Pomdapi. {Francois.Clement,Pierre.Weis}@inria.fr.
§ LIPN, UMR 7030, Université Paris-Nord, CNRS, Villetaneuse, F-93430.
Micaela.Mayero@lipn.univ-paris13.fr.

RESEARCH CENTRE
PARIS – ROCQUENCOURT

Domaine de Voluceau, - Rocquencourt
B.P. 105 - 78153 Le Chesnay Cedex

Calculs en confiance : une preuve mécanisée de l'équation aux dérivées partielles au programme effectif

Résumé : Il est bien connu que les programmes peuvent échouer, que ce soit en raison de comportements exceptionnels, d'accès hors des bornes d'un tableau, ou simplement d'erreurs de codage. On ne peut donc pas leur faire confiance aveuglément. De ce point de vue, les programmes de calcul scientifique ne font pas exception, et ils présentent même des problèmes spécifiques de précision en raison de l'utilisation massive de calculs en virgule flottante. Cependant, leur correction est rarement garantie. Ils existe des méthodes et des outils pour prouver le comportement correct de programmes et nous les avons étendus à la vérification d'un programme d'analyse numérique existant. Ce programme C implémente le schéma explicite aux différences finies centrées du second ordre pour la résolution de l'équation des ondes mono-dimensionnelle. En fait, nous sommes allés beaucoup plus loin puisque nous avons vérifié mécaniquement la convergence du schéma numérique pour obtenir une preuve formelle complète couvrant tous les aspects de l'équation aux dérivées partielles aux résultats numériques effectifs. À notre connaissance, c'est la première fois qu'une telle preuve complète est obtenue.

Mots-clés : équation des ondes acoustiques, preuve formelle d'un programme numérique, convergence d'un schéma numérique, analyse d'erreur d'arrondi.

1 Introduction

Given an appropriate set of mathematical equations (such as ODEs or PDEs) modeling a physical object, the usual simulation process is as follows. First, these equations are approximated by a set of discrete equations, the numerical scheme; this scheme is then proved to be convergent. Second, the numerical scheme is implemented as a computer program. The program 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 phases.

Usually, the discretization phase is guaranteed by a pen-and-paper proof of the convergence of the selected scheme, while the implementation phase is simply validated by a set of tests. 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 test-based validation 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 is not a way to obtain the correctness of modeling programs.

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 has all its logical steps validated 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 with neither hole nor oversight left inside.

Fortunately, logical frameworks also support the definition of computer programs and the specification of their properties. Then, a correctness proof of a program is a formal proof that no execution of the program will go wrong and that the properties are always verified. A formal proof of a program can be considered as a comprehensive validation with an exhaustive set of tests.

Mechanical proof checkers are mainly used to formalize mathematics and are routinely used to prove programs in the field of integer arithmetics 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 phase 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 continuous solution. 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 real 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 phase uses real numbers and real analysis theory. Usual theorems and tools of real analysis are still in their infancy in mechanical proof checkers. In addition, numerical programs use floating-point arithmetic, and more specifically, the IEEE-754 standard. Properties of floating-point arithmetic are complex and highly nonintuitive, which is yet another show stopper for formal proofs of numerical programs.

To summarize, the field of scientific computing gathers regular formal proof difficulties 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 in mechanical proof checkers providing some real analysis and an IEEE-754 formalization now makes formal proofs of numerical programs

tractable.

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 [12]. 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 which cost they come at.

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, 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 we have seen that knowledge of arithmetic and analysis is required. In Coq, real numbers have been 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 will be too long to be given here *in extenso*. Only general ideas and difficulties will be explained. The annotated C program and the full Coq sources of the formal development are available from

`http://fost.saclay.inria.fr/wave_total_error.html`

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. Sections 4 is devoted to the formal proof of the convergence of the numerical scheme, and Section 5 to the formal proof of the C program implementing the numerical scheme. Finally, Section 6 paints a broader picture of the study.

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

2 Formal Proof

Modern mathematics can be seen as the science of abstract objects, *e.g.* real numbers, differential equations. In contrast, mathematical logic works on the languages used to define such abstract objects and reason about them. Once these languages of definitions and proofs are formalized, one can manipulate and reason about mathematical proofs: What makes a valid proof? How can we find 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 not be infallible: 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 matter, 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, this work can be delegated to a computer running a formal system. The computer will perform it much faster and much more systematically than a human ever could. 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 skip some proof steps, dismissing them by a wave of hand, 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 numeric analysis is not amenable to automatic proving yet, which means that related properties will require a lot of human interaction, as shown in Section 6.1.

It should have become apparent by now that formal systems are primarily aimed at proving and checking mathematical theorems. But programs are also among the abstract objects they can manipulate, thus allowing to prove theorems about them. These theorems might be about basic properties of a program, *e.g.* it will not crash. 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 continuous 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. 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 well 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 range of the trust 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, or that a computer processor might have design flaws. 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, people are also applying formal methods to improve the overall trust in a system: they are verifying hardware design on a daily basis, and are starting to formally verify compilers [32] and operating systems [30] too.

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 what kind of abstract objects we

[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-line long.

want to manipulate or prove properties about.

The first step lies in running the tool Frama-C (with the Jessie plug-in) over the program. 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 continuous 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 of 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 is suited for proving theorems stating that floating-point operations do not overflow or that their rounding error is bounded, while an *SMT solver** will tackle theorems stating that arrays are never accessed out of their bounds. Unfortunately, the more complicated theorems require some user interaction, so we have used the Coq proof assistant to help us in writing their formal proofs. This is especially true for theorems that deal with the more mathematically-oriented aspect of the verification, *e.g.* the 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 [7]. Coq's formal language combines both a *higher-order logic** and a richly-typed *functional programming language** [18]. In addition to functions and predicates, Coq allows to state 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. For instance, Coq has been used to verify an EAL7-grade[3] smart card [16].

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 its 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, so that users do not have to start their proofs from scratch but can instead reuse well-known theorems that have already been formally proved beforehand. This general-purpose library contains various developments and axiomatizations about sets, lists, sorting, arithmetic, real numbers, etc. In this work, we mainly use the Reals standard library [34], which is a classical axiomatization of an Archimedean ordered complete field. It provides all the basic theorems about analysis, *e.g.* differentials, integrals. It does not contain more advanced topics such as Fourier transform and its properties though. Here is a short example for the irrationality of $\sqrt{2}$: statement of the theorem and start of the proof.

```
Definition irrational (x : R) : Prop :=
  forall (p : Z) (q : nat), q ≠ 0 → x ≠ p / q.
```

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

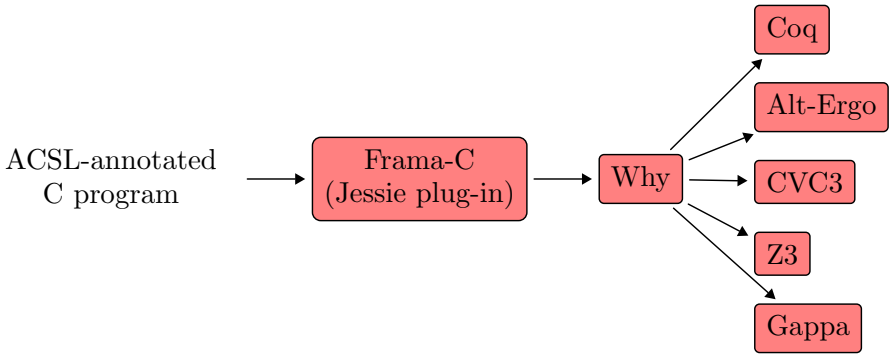
[3] EAL stands for Evaluation Assurance Level, and 7 is the highest level of security defined by the Common Criteria standard (ISO 15408). As a point of comparison, the security level commonly met by smart cards is only EAL4.

```
Theorem irrational_sqrt_2: irrational (sqrt 2).
Proof.
  intros p q H H0.
  ...
Qed.
```

The standard library does not come with a formalization of floating-point numbers. For that purpose, we use a large Coq library[4] initially developed in [21] and extended with various results afterwards [9]. It is a high-level formalization of the IEEE-754 international standard for floating-point arithmetic. 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. This library has since then been superseded by the Flocq library [15] 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[5] 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 for *deductive verification*⋆. C programs are *annotated*⋆ with behavioral contracts written using the *ANSI C Specification Language* (ACSL for short) [5]. The Jessie plug-in translates them to the Jessie language [33], which is part of the Why verification platform [26]. 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, etc.) and Why programs (to model C programs). Finally, the Why platform computes *verification conditions*⋆ from these programs, using traditional techniques of weakest preconditions [24], 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 [17], CVC3 [4] and Z3 [23], and the automated theorem prover Gappa (Section 2.2.3). Details about automated and interactive proofs can be found in Section 6.1. 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 Appendix C. 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);
```

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

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 Δx (see Section 3.2). 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).

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

2.2.3 Gappa

The Gappa tool[6] adapts the *interval-arithmetic** paradigm to the proof of properties that occur when verifying numerical applications [20]. The inputs are logical formulas quantified over real numbers whose atoms are usually bounds on 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 [28]:

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

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 are performing convoluted manipulations. For instance, it has been successfully used to verify a state-of-the-art library of correctly-rounded elementary functions [22]. 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 5.1.1). 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.

The verification of some formulas requires reasonings that are so long and intricate [22], that it might cast some doubts on whether an automatic tool actually succeeded 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 solving goals related to floating-point and real arithmetic [13]. Note that Gappa itself is not verified, but since Coq verifies the proofs that Gappa generates, the goals are formally proved.

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

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

The case of the numerical resolution of the one-dimensional acoustic wave equation using the second-order centered explicit scheme is chosen for its simplicity, and for its representativity of a wide class of scientific computing problems. We describe and state the different notions necessary for the implementation of the numerical resolution of the partial differential equation and its analysis.

This section, as well as the steps taken at Section 4 to conduct the convergence proof of the numerical scheme, is inspired by [6].

3.1 Continuous Equation

We consider a one-dimensional homogeneous acoustic medium $\Omega = [x_{\min}, x_{\max}]$ 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 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 equation on Ω writes

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

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

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

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

where the differential operator $A(c)$ is defined by

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

We admit that under reasonable conditions on the Cauchy data p_0 and p_1, there exists a unique solution p to the initial-boundary value problem (1)–(4) for each $c > 0$. Of course, it is well-known that the solution to this partial differential equation is given by d'Alembert's formula [31]. But simply assuming the existence of a solution instead of exhibiting it opens the way to scale our approach to more general cases for which there is no known analytic expression of a solution, *e.g.* in the case of a nonuniform propagation velocity c.

We associate the positive definite quadratic quantity

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

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. $\tilde{p}_1$) represents the functions defined on the entire real axis $\mathbb{R}$ obtained by successive antisymmetric extensions in space respectively of p_0 (resp. p_1). For example, we have, for all $x \in [2x_{\min} - x_{\max}, x_{\min}]$, for all t, $\tilde{p_0}(x,t) = -p_0(2x_{\min} - x, t)$. The image theory [29] stipulates that the solution of the wave equation (1)–(4) coincides on the domain Ω with the solution of the same wave equation but set on the entire real axis $\mathbb{R}$, without the Dirichlet boundary condition (4), with extended Cauchy data $\tilde{p}_0$, $\tilde{p}_1$.

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[7]

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

$$\Delta t \stackrel{\text{def}}{=} \frac{t_{\max}}{k_{\max}}, \qquad k_{\Delta t}(t) \stackrel{\text{def}}{=} \left\lfloor \frac{t}{\Delta t} \right\rfloor. \tag{8}$$

The regular discrete grid approximating $\Omega \times [0, t_{\max}]$ is defined by[8]

$$\forall k \in [0..k_{\max}],\ \forall i \in [0..i_{\max}], \quad \mathbf{x}_i^k \stackrel{\text{def}}{=} (x_i, t^k) \stackrel{\text{def}}{=} (x_{\min} + i\Delta x, k\Delta t). \tag{9}$$

For a function q defined over $\Omega \times [0, t_{\max}]$ (resp. Ω), the notation q_{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_{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 $\bar{q}_{\mathrm{h}}$ is reserved to the evaluation at the points of the grid, $\bar{q}_i^k \stackrel{\text{def}}{=} q(\mathbf{x}_i^k)$ (resp. $\bar{q}_i \stackrel{\text{def}}{=} q(x_i)$).

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

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

$$\forall i \in [1..i_{\max} - 1], \quad (L_{1,\mathrm{h}}(c)\, p_{\mathrm{h}})_i \stackrel{\text{def}}{=} \frac{p_i^1 - p_i^0}{\Delta t} + \frac{\Delta t}{2}(A_{\mathrm{h}}(c)\, p_{\mathrm{h}}^0)_i = u_{1,i}, \tag{11}$$

$$\forall i \in [1..i_{\max} - 1], \quad (L_{0,\mathrm{h}}\, p_{\mathrm{h}})_i \stackrel{\text{def}}{=} p_i^0 = u_{0,i}, \tag{12}$$

$$\forall k \in [0..k_{\max}], \quad p_0^k = p_{i_{\max}}^k = 0 \tag{13}$$

where the matrix $A_{\mathrm{h}}(c)$ (discrete analog of $A(c)$) is defined by

$$\forall q_{\mathrm{h}}, \forall i \in [1..i_{\max} - 1], \quad (A_{\mathrm{h}}(c)\, q_{\mathrm{h}})_i \stackrel{\text{def}}{=} -c^2 \frac{q_{i+1} - 2q_i + q_{i-1}}{\Delta x^2}. \tag{14}$$

Note the use of a second order approximation of the first derivative in time in (11).

A discrete analog of the energy is also defined by

$$E_{\mathrm{h}}(c)(p_{\mathrm{h}})^{k+\frac{1}{2}} \stackrel{\text{def}}{=} \frac{1}{2} \left\| \frac{p_{\mathrm{h}}^{k+1} - p_{\mathrm{h}}^k}{\Delta t} \right\|_{\Delta x}^2 + \frac{1}{2} \left\langle p_{\mathrm{h}}^k, p_{\mathrm{h}}^{k+1} \right\rangle_{A_h(c)} \tag{15}$$

where, for any vectors q_{h} and r_{h},

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

Note that $\|.\|_{A_h(c)}$ is a semi-norm.

[7] The *floor* notation $\lfloor . \rfloor$ denotes the rounding to an integer towards minus infinity.
[8] For integers n and p, the notation $[n..p]$ denotes the integer range $[n, p] \cap \mathbb{N}$.

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_h. When these input data are respectively approximations of the continuous 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_h is an approximation of the continuous solution p.

The remark about image theory holds here too: we may replace the use of Dirichlet boundary conditions (13) by considering extended discrete Cauchy data $\tilde{p}_{0,\mathrm{h}}$, and $\tilde{p}_{1,\mathrm{h}}$. Note also that, as in the continuous case when a source term is considered, the discrete solution of the numerical scheme (10)–(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 5.1.2 and 6.2.

3.3 Convergence

Let ξ be in $]0,1[$. The CFL(ξ) condition (for Courant-Friedrichs-Lewy, see [19]) states that the discretization steps satisfy the relation

$$\frac{c\Delta t}{\Delta x} \leq 1 - \xi. \tag{16}$$

The convergence error e_h and the truncation error ε_h are defined by

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

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

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

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

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

When the input data of the numerical scheme for the approximation of the continuous solution are given by $u_{0,\mathrm{h}} = \bar{p_0}_\mathrm{h}$, $u_{1,\mathrm{h}} = \bar{p_1}_\mathrm{h}$, and $s_\mathrm{h} \equiv 0$, the convergence error e_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 $s_\mathrm{h} = \left(k \mapsto \varepsilon_\mathrm{h}^{k+1}\right)$.

The numerical scheme is said to be convergent of order (m,n) uniformly on the interval $[0, t_{\max}]$ if the convergence error satisfies[9]

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

The numerical scheme is said to be consistent with the continuous problem at order $(m,\ n)$ uniformly on interval $[0, t_{\max}]$ if the truncation error satisfies

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

The numerical scheme is said to be stable uniformly on interval $[0, t_{\max}]$ if the discrete solution of the problem without any source term satisfies

$$\exists \alpha, C_1, C_2 > 0,\ \forall t \in [0, t_{\max}],\ \forall \Delta x, \Delta t > 0, \quad \sqrt{\Delta x^2 + \Delta t^2} < \alpha \Rightarrow \left\| p_\mathrm{h}^{k_{\Delta t}(t)} \right\|_{\Delta x} \leq (C_1 + C_2 t)(\|u_{0,\mathrm{h}}\|_{\Delta x} + \|p_{0,\mathrm{h}}\|_{A_h(c)} + \|u_{1,\mathrm{h}}\|_{\Delta x}). \tag{24}$$

The result *formally proved** at Section 4 states that if the continuous solution p is regular enough on $\Omega \times [0, t_{\max}]$ and if the discretization steps satisfy the CFL(ξ) condition, then the second-order centered scheme is convergent of order (2, 2) uniformly on interval $[0, t_{\max}]$.

We do not admit (nor prove) the Lax equivalence theorem which stipulates that for a wide variety of problems and numerical schemes, consistency implies the equivalence between stability

[9] See Section 4.1 for the precise definition of the big O notation. The function $k_{\Delta t}$ is defined in (8).

and convergence. Instead, we establish in the particular case of the one-dimensional acoustic wave equation that consistency and stability imply convergence.

The Fourier transform is a very popular tool to study the convergence of numerical schemes. The formalization in Coq of Lebesgue integral theory and Fourier transform theory should not encounter any major difficulty, except for the human cost of such a development. Since there was no Coq support for these theories when we started this work (and there is still not), we decided to consider energy-based techniques, as they only involve finite summations (to compute discrete dot products). The energy-based approaches rely on a priori error estimations, so they are less precise; they can be applied to the heterogeneous case though, and to irregular approximation grids.

3.4 C Program

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 assume that $x_{\min} = 0$, $x_{\max} = 1$, and that the continuous solution is bounded by 1. The main part of the C program is listed in Listing 1. The grid steps Δx and Δt are respectively represented by the variables dx and dt, the grid sizes $i_{\max}$ and $k_{\max}$ by the variables ni and nk, and the propagation velocity c by the variable v. The initial value $u_{0,\mathrm{h}}$ is represented by the function p0. The other input data $u_{1,\mathrm{h}}$ and s_{h} are supposed to be zero and are not represented. The discrete solution p_{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

The full *annotations** are given in Appendix B. We give here some hints about how to specify this program.

There are two axiomatics that define the logical formulas needed for the program annotations. The first one corresponds to the mathematics: the exact solution of the wave equation and its properties. It defines the needed values (the exact solution p, and its initialization p_0). It also defines the derivatives of p ($psol_1$, first derivative for the first variable of p, and $psol_{11}$, second derivative for the first variable, and $psol_2$ and $psol_{22}$ for the second variable). The value of the derivative of f 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 $psol_1$, that is to say $\frac{\partial p}{\partial x}$:

```
/*@ 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`.

We also put as axioms the fact that the solution has the expected properties (1–4). The last property needed on the exact solution is its regularity. We require it to be near its Taylor approximations of degrees 3 and 4 on the whole interval $[x_{\min}, x_{\max}]$. 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 axiomatic corresponds to the properties and *loop invariant** needed by the program. For example, we require the matrix to be *separated*. It means that a line of the matrix should not overlap with another line. Otherwise a modification would alter another point of 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 specified, but not stated. This corresponds firstly to the function `array2d_alloc` that initializes the matrix and `p_zero` that produces an approximation of the p_0 function. Our program verification is modular: our proofs are 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 grid sizes $\mathbf{\Delta x}$ must be small enough to ensure the convergence of the scheme;
- the floating-point values computed for the grid sizes must be near their mathematical values;
- to prevent exceptional behavior in the computation of a, Δt 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 round-off errors. See Sections 4 and 5.1 for more details.

4 Formal Proof of the Convergence of the Scheme

We first present the notions necessary to formalize and prove the method error. The method error is the distance between the discrete value (with exact real arithmetic) and the continuous mathematical value; it reduces to the convergence error here. *Formal specifications and proofs** often highlight mathematical details that are not visible in a pen-and-paper proof. So we detail how the proof is conducted: we establish the consistency and the stability, and prove that these two properties imply convergence in the case of the one-dimensional acoustic wave equation.

4.1 Big O, Differentiability, and Regularity

When considering a big O equality $a = O(b)$ (which must be understood as "a belongs to $O(b)$"), one usually assumes that a and b are two expressions defined over the same domain and its interpretation as a quantified formula comes naturally. For Taylor approximations, the situation is a bit more different. Consider

$$f(\mathbf{x}, \mathbf{\Delta x}) = O(g(\mathbf{\Delta x}))$$

when $\|\mathbf{\Delta x}\|$ goes to 0. If one were to assume that the equality holds for any $\mathbf{x} \in \mathbb{R}^2$, one would interpret it as

$$\forall \mathbf{x}, \exists \alpha > 0, \exists C > 0, \forall \mathbf{\Delta x}, \quad \|\mathbf{\Delta x}\| \leq \alpha \Rightarrow |f(\mathbf{x}, \mathbf{\Delta x})| \leq C \cdot |g(\mathbf{\Delta x})|,$$

which means that constants α and C are in fact functions of $\mathbf{x}$. Such an interpretation happens to be useless, since the infimum of α may well be zero while the supremum of C may be $+\infty$.

A proper interpretation requires the introduction of a uniform big O relation with respect to the additional variable $\mathbf{x}$:

$$\exists \alpha > 0,\ \exists C > 0,\ \forall \mathbf{x} \in \Omega_{\mathbf{x}},\ \forall \mathbf{\Delta x} \in \Omega_{\mathbf{\Delta x}},\ \|\mathbf{\Delta x}\| < \alpha \Rightarrow |f(\mathbf{x}, \mathbf{\Delta x})| \leq C \cdot |g(\mathbf{\Delta x})|. \tag{25}$$

In Coq, this is written as follows, where Prop is the type of the logical propositions, norm_l2 is the usual L^2 norm and Rabs the absolute value:

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

Here, type A stands for domain $\Omega_{\mathbf{x}}$, and predicate P for domain $\Omega_{\mathbf{\Delta x}}$.

To emphasize the dependency on the two subsets $\Omega_{\mathbf{x}}$ and $\Omega_{\mathbf{\Delta x}}$, uniform big O equalities are now written

$$f(\mathbf{x}, \mathbf{\Delta x}) = O_{\Omega_{\mathbf{x}}, \Omega_{\mathbf{\Delta x}}}(g(\mathbf{\Delta x})).$$

Now, we precisely define the notion of "sufficiently regular" functions in terms of the full-fledged notation for the big O. The further result on the convergence of the numerical scheme requires that the solution of the continuous equation is actually sufficiently regular. We can define the usual Taylor polynomial of order n of a function f:

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

Let $\Omega_{\mathbf{x}} \subset \mathbb{R}^2$. We say that the previous Taylor polynomial is a uniform approximation of order n of f on $\Omega_{\mathbf{x}}$ when the following uniform big O equality holds:

$$f(\mathbf{x}+\mathbf{\Delta x}) - \mathrm{TP}_n(f,\mathbf{x})(\mathbf{\Delta x}) = O_{\Omega_{\mathbf{x}},\mathbb{R}^2}\left(\|\mathbf{\Delta x}\|^{n+1}\right).$$

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

4.2 Consistency

We formally prove that, when the continuous solution of the wave equation (1)–(4) is sufficiently regular of order 4 uniformly on $[x_{\min}, x_{\max}] \times [0, t_{\max}]$, the numerical scheme (10)–(13) is consistent with the continuous problem at order (2, 2) uniformly on interval $[0, t_{\max}]$ (see definition (23) in Section 3.3). This is obtained using the properties of Taylor approximations; the proof is straightforward while involving long and complex expressions.

The key idea is to always manipulate uniform Taylor approximations that are valid in a compact subset that includes all points of all grids when the discretization steps goes down to zero.

For instance, to take into account the initialization phase corresponding to Equation (11), we have to derive a uniform Taylor approximation of order 1 for the following continuous function (for any v sufficiently regular of order 3):

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

Note that the expression of this function involves both $x+\Delta x$ and $x-\Delta x$, meaning that we need a Taylor approximation that is valid for both positive and negative variations. The formal proof would have been impossible if we had required $0 < \Delta x$ (as a space grid step must be) in the definition of the Taylor approximation. In contrast with the case of an infinite string [11], we do not need here a lower bound for $c\frac{\Delta t}{\Delta x}$.

4.3 Stability

For the proof of the round-off error (see Section 5.1), we need a statement of the same form as definition (24) of Section 3.3. Therefore, we formally prove that, under the CFL(ξ) condition (16), the numerical scheme (10)–(13) is stable uniformly on interval $[0, t_{\max}]$.

As stated in Section 3.3, we use an energy-based technique to prove the convergence of the numerical scheme, hence we need to express the stability in terms of the discrete energy. More precisely, we formally prove that under the CFL(ξ) condition (16), the discrete energy (15) satisfies the following overestimation:

$$\sqrt{E_{\mathrm{h}}(c)(p_{\mathrm{h}})^{k+\frac{1}{2}}} \leq \sqrt{E_{\mathrm{h}}(c)(p_{\mathrm{h}})^{\frac{1}{2}}} + \frac{\sqrt{2}}{2\sqrt{2\xi-\xi^2}} \cdot \Delta t \cdot \sum_{k'=1}^{k} \left\| \left(i \mapsto s_i^{k'}\right) \right\|_{\Delta x} \tag{26}$$

for all $t \in [0, t_{\max}]$ and with $k = \left\lfloor \frac{t}{\Delta t} \right\rfloor - 1$.

The evolution of the discrete energy between two consecutive time steps is shown to be proportional to the source term. In particular, the energy is constant when the source is inactive. Then, we obtain the following underestimation of the discrete energy:

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

Therefore, the nonnegativity of the discrete energy is directly related to the CFL(ξ) condition.

Note that this stability result is valid for any 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 that, when the continuous solution of the wave equation (1)–(4) is sufficiently regular of order 4 uniformly on $[x_{\min}, x_{\max}] \times [0, t_{\max}]$, and under the CFL($\xi$) condition (16), the numerical scheme (10)–(13) is convergent of order (2, 2) uniformly on interval $[0, t_{\max}]$ (see definition (22) in Section 3.3).

Firstly, we prove that the convergence error e_{h} is itself the discrete solution of a numerical scheme of the same form but with different input data. (Of course, there is no associated continuous problem.) In particular, the source term (on the right-hand side) is here the truncation error ε_{h} associated with the initial numerical scheme for p_{h}. Then, the previous stability result holds, and we have an overestimation of the square root of the discrete energy associated with the convergence error $E_{\mathrm{h}}(c)(e_{\mathrm{h}})$ that involves a sum of the corresponding source terms, *i.e.* the truncation error. Finally, the consistency result also makes this sum a big O of $\Delta x^2 + \Delta t^2$, and a few more technical steps and a study of initializations conclude the proof.

4.5 Difficulties

Big O. In Section 4.1, we present two interpretations of the big O notation. Usual mathematical pen-and-paper proofs switch from one interpretation to the other depending on which one is the most adapted, without noticing that they may not be equivalent. The formal development was helpful in bringing into light the erroneous reasoning hidden by the usage of big O notations. We introduced the notion of uniform big O in [11] in the context of an infinite string. In the present paper, we consider the case of the finite string, hence for compactness reasons, both notions are in fact equivalent. However, we still use the more general uniform big O notion to share most of the proof developments between the finite and infinite cases. Regarding automation, a *decision procedure** has been developed in [3]; unfortunately, those results were not applicable since we needed a uniform big O.

Annotations describing differential equations. As long as we were studying only the method error, we did not have to define the differential operators nor assume anything about them [11]. Their only properties appeared through their usage: function p is a solution of the partial differential equation and is sufficiently regular. This is no longer possible for the *annotated** C program. We therefore need some total operator equal to the derivative when the function is differentiable, which was not available in the standard Coq library. We previously added an axiom stating its existence; we recently removed it thanks to [14].

Differential equations introduce another issue: due to the underlying logic, the annotations have to define p as a 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.

Method error. The Coq proof of the method error is about 5000-line long. About half of it is dedicated to the wave equation, while the other half is re-usable: Hilbert space analysis, asymptotic analysis, Taylor approximations, and so on. We formally proved without any axiom that the numerical scheme is convergent of order 2 both in space and time, which is the known mathematical result. An interesting aspect of the formal proof in Coq is that we were able to extract the constants α and C appearing in the big O for the convergence result in order to obtain their precise values. The recursive extraction was fully automatic after making explicit some inlining. The mathematical expressions are given in Section 5.2.

5 Formal Proof of the C Program

5.1 Round-off Error

As is well-known, each operation on real numbers is implemented with IEEE-754 floating-point numbers [35, 28], so round-off errors will occur and may endanger the accuracy of the final results. On this program, naive *forward error analysis** gives an error bound that is proportional to $2^k 2^{-53}$ for the computation of a p_i^k. If this bound was sensible, it would cause the numerical scheme to compute only noise after a few steps. Fortunately, round-off errors actually compensate themselves in this program. To take into account compensations and hence prove a usable error bound, we need a precise statement of the round-off error [10] to exhibit the cancellations made by the numerical scheme.

Note that bounding the round-off errors and exhibiting the compensations is a novel idea. This is also very complex to prove, first on paper (several pages of error-prone computations with double summations) and then with Coq (with few automations and cumbersome notations). The formalizations of floating-point arithmetic we used are previous works [21, 15]: they reflect the IEEE-754 standard and therefore handle both normal and very small numbers (called subnormal). Other floating-point special values (infinities, NaNs) are proved not to appear in the program.

5.1.1 Local Round-off Errors

Let δ_i^k be the local floating-point error made for 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 made in line 9 of Listing 1 (resp. in lines 19–20, and in lines 32–33). To distinguish them from the discrete values of previous sections, we decorate the floating-point values as computed by the program with an underline. 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:

$$\begin{aligned}
\forall k \in [1..k_{\max}-1],\, \forall i \in [1..i_{\max}-1], \qquad \delta_i^{k+1} &= \left(2\underline{p}_i^k - \underline{p}_i^{k-1} + a(\underline{p}_{i+1}^k - 2\underline{p}_i^k + \underline{p}_{i-1}^k)\right) - \underline{p}_i^{k+1},\\
\forall i \in [1..i_{\max}-1], \qquad \delta_i^1 &= \left(\underline{p}_i^0 + \frac{a}{2}(\underline{p}_{i+1}^0 - 2\underline{p}_i^0 + \underline{p}_{i-1}^0)\right) - \underline{p}_i^1\\
&\qquad - \left(\delta_i^0 + \frac{a}{2}\left(\delta_{i+1}^0 - 2\delta_i^0 + \delta_{i-1}^0\right)\right),\\
\forall i \in [1..i_{\max}-1], \qquad \delta_i^0 &= p_i^0 - \underline{p}_i^0.
\end{aligned}$$

Note that the program presented in Section 3.4 gives us that

$$\begin{aligned}
\forall k \in [1..k_{\max}-1],\, \forall i \in [1..i_{\max}-1], \qquad \underline{p}_i^{k+1} &= \mathrm{fl}\left(2\underline{p}_i^k - \underline{p}_i^{k-1} + a(\underline{p}_{i+1}^k - 2\underline{p}_i^k + \underline{p}_{i-1}^k)\right),\\
\forall i \in [1..i_{\max}-1], \qquad \underline{p}_i^1 &= \mathrm{fl}\left(\underline{p}_i^0 + \frac{a}{2}(\underline{p}_{i+1}^0 - 2\underline{p}_i^0 + \underline{p}_{i-1}^0)\right),\\
\forall i \in [1..i_{\max}-1], \qquad \underline{p}_i^0 &= \mathrm{fl}\left(p_0(i * \underline{\Delta x})\right).
\end{aligned}$$

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 δ_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 continuous 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 continuous solution, the method error, and the round-off error.

To prove the bound on δ_i^k, we perform forward error analysis and then use *interval arithmetic** to bound each intermediate error [20]. 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 $|\underline{a} - a| \leq 2^{-49}$.

5.1.2 Global Round-off Error

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

Indeed, 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:

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

Taking advantage on the remark about image theory made in Section 3.2, we see the global floating-point error as the restriction to the domain Ω of the solution of the same numerical scheme set on the entire real axis without the Dirichlet boundary condition (13), 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 local floating-point error.

Let us denote by λ_{h} the discrete fundamental solution. It is 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 [10] for a direct proof that does not follow the above remarks.)

Theorem 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 λ_i^k vanishes as soon as $|i| \geq k$, hence the sum over j could be replaced by an infinite sum over all integers.

5.1.3 Bound on the Global Round-off Error

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

Lemma 1.

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

Proof. The sum σ_{h} 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 2.

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

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

Theorem 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 1, Δ_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 5.1.1 that for all j and l, $|\tilde{\delta}_j^l| \leq 78 \cdot 2^{-52}$, and from Lemma 1 that $\sum \lambda_i^{l+1} = l + 1$. Since the λ's are nonnegative (Lemma 2), the error is easily bounded by $78 \cdot 2^{-52} \sum_{l=0}^{k} (l+1)$. □

Except for Lemma 2, all the proofs described in this section have been machine-checked using Coq. In particular, the proof of the bound on δ_i^k was done automatically by calling Gappa from Coq. Lemma 2 is a technical detail compared to the rest of our work, that is not worth the immense Coq development it would require: keen results on integrals but also definitions and results about the Legendre, Laguerre, Chebychev, and Jacobi polynomials (see Section 6.2).

5.2 Total Error

Let $\mathcal{E}_\mathrm{h}$ be the total error. It is the sum of the method error (or convergence error) e_h of Sections 3.3 and 4.4, and of the round-off error Δ_h of Section 5.1.

From Theorem 2, we can estimate[10] 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 triangular inequality for the spatial norm, we obtain the following estimation of the total error:

$$\forall t \in [0, t_{\max}],\ \forall \mathbf{\Delta x},\quad \|\mathbf{\Delta x}\| \leq \min(\alpha_e, \alpha_\Delta) \Rightarrow \left\| \left(i \mapsto \mathcal{E}_i^{k_{\Delta t}(t)} \right) \right\|_{\Delta x} \leq C_e(\Delta x^2 + \Delta t^2) + \frac{C_\Delta}{\Delta t^2} \tag{27}$$

where the method error constants α_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 (α_3 and C_3), and at degree 4 (α_4 and C_4) by

$$\alpha_e = \min(1, t_{\max}, \alpha_3, \alpha_4), \tag{28}$$

$$C_e = 2\mu t_{\max} \sqrt{x_{\max} - x_{\min}} \left(\frac{C'}{\sqrt{2}} + \mu(t_{\max}+1)C'' \right) \tag{29}$$

with $\mu = \frac{\sqrt{2}}{\sqrt{2\xi - \xi^2}}$, $C' = \max(1, C_3 + c^2 C_4 + 1)$, and $C'' = \max(C', 2(1+c^2)C_4)$, and where the round-off constants α_Δ and C_Δ, as explained above, are given by

$$\alpha_\Delta = \min(1, t_{\max}/2), \tag{30}$$

$$C_\Delta = 234 \cdot 2^{-53} t_{\max}^2 \sqrt{x_{\max} - x_{\min} + 1}. \tag{31}$$

Of course, decreasing the size of the grid step decreases the method error, but in 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 when ξ is small. The main reason is the μ^2 factor in the expression of C_e, which is asymptotically equivalent to $1/\xi$ for small values of ξ (see Section 6.2). 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 [12].

6 Discussion

6.1 Automation and Manual Proofs

Given the program code, the Frama-C/Jessie/Why tools generate 150 *verification conditions** that have to be proved. While possible, proving all of them in Coq would be rather tedious. Indeed,

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

while Coq provides a few automations, it is more of a proof checker, so the proofs end up being mostly written by the user. Moreover, a systematic usage of 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) prove as many of them as possible, so that only the most intricate ones are left to be proven in Coq. The following table in Figure 1 shows how many *goals*⋆ are proved automatically and how many are left to the user.[11]

Prover	Proved Behavior VC	Proved Safety VC	Total
Alt-Ergo	18	80	98
CVC3	18	89	107
Gappa	2	20	22
Z3	21	63	84
Automatically proved	**23**	**94**	**117**
Coq	**21**	**12**	**32**
Total	44	106	150

Figure 1: Number and type of proved verification conditions (VC) depending on the prover.

Safety goals are the ones that are always generated, even in the absence of any *specification*⋆. Proving them ensures that the program terminates without any error when executed. 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: they prove about 90 % of the goals.

Behavior goals are the ones that relate a 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 goals. Some of these failed goals depend on an uninterpreted predicate, so automatic provers are missing crucial information. But even if that information was provided, the goals would still be too complicated for them, since they use mathematical theories out of their scope. That is why we resort to an *interactive higher-order theorem prover*⋆, namely Coq.

Coq proofs are split into two sets: first, the mathematical proof of convergence and second, the proofs of bounded round-off errors and absence of runtime errors.

Type of proofs	Nb statement lines	Nb proof lines
Convergence	999	4 108
Round-off + runtime errors	7 776	12 336

Note that most proof statements regarding round-off and runtime errors are automatically generated (7 289 lines out of 7 776) by the Frama-C/Jessie/Why framework.

6.2 Applied Mathematicians and Formal Proof of Programs

6.2.1 Method Error

From the computational scientist point of view, the big surprise of a *formal proof*⋆ development is the requirement to start by writing an extremely detailed conventional pen-and-paper proof. Indeed, a 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

[11] Note that verification conditions might be proved by one or several automated provers, and that no single prover is able to prove all the automatically proved conditions.

human driven, up to the utmost detail. Fortunately, we have not to specify all the mathematical facts since Coq provides a collection of known facts and theories that can be reused to establish new results. However, classical results from Hilbertian analysis, Taylor approximations, and asymptotic comparison of functions, have not yet been proved in Coq; thus, we had to develop them, hence also in the detailed pen-and-paper proof. In the end, the detailed pen-and-paper proof is about 50-page long. Surprisingly, it is roughly the same size as the complete Coq formal proof (both source files have approximately 5,000 lines and 200,000 characters).

The other unexpected fact is the necessity to precisely state the definition of the big O notion in the context of Taylor expansions and discrete grids. This leads to focus on the question of uniformity of the existential constants, and use a uniform version of the big O notion in the proofs. This point does not seem to be dealt with in the literature; there is an informal mention in [37] with a relevant remark about pointwise consistency versus norm consistency.

With pen-and-paper proofs, it is uncommon to expand the constants on the upper bound of the method error. In contrast, Coq can automatically extract the constants from the formal proof. On specific examples, we can see that they are far from being optimal. A priori error estimations provided by energy-based techniques are known to be less precise as the CFL condition becomes optimal. Indeed, the factor $\mu = \frac{\sqrt{2}}{\sqrt{2\xi - \xi^2}}$ in the expression of the overestimation of the discrete energy (26) grows to $+\infty$ when ξ goes down to 0^+. This makes the constant C_e in (29) grow as $1/\xi$ for small values of ξ. The alternative to obtain more accurate bounds is known: use a leapfrog scheme for the equivalent first-order system.

6.2.2 Round-off Error

In numerical analysis, it is common to evaluate bounds for the norm of quantities. 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 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 D).

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 a slightly worse bound on the global round-off error $|\Delta_i^k| \leq 156 \cdot 2^{-52}(k+1)^2$.

More generally, a numeric scheme is said stable if the values do not diverge. This typically means that, given initial values (such as the values and derivative(s) at time 0), the values of the numerical scheme do not diverge. In floating-point arithmetic, an algorithm is said numerically stable if it has a small forward error: a small error on the input gives a small error on the output. This is of course the kind of algorithm we are looking for. Computational scientists believe (as a rule of thumb) that stable schemes are numerically stable. This is a belief funded by experiments and examples. We think this fact to be true. To prove it, we would need to formalize the notion of numerical scheme, the property of numerical scheme stability, and then deduce the numerical stability of any stable numerical scheme. This result should be generic enough to be applied to a large class of numerical schemes. It should also state a good enough bound to guarantee a reasonable accuracy.

6.3 Future Work

Another way to explore is the mechanism of program extraction. Instead of proving an existing program, one formalizes a problem, proves it has a solution, and obtains for free an OCaml program from the Coq proof terms. This program is usually not efficient, but it is a zero-defect program, provided that it is not modified by hand. In our case, efficiency is not the only issue; the extracted program would also be hindered by the omnipresence of classical real numbers in our formalization. So being able to extract efficient and usable zero-defect programs would be an

interesting long term goal for this kind of critical numerical problems.

For this exploratory work, we considered the simple three-point scheme for the one-dimensional wave equation. Further works involve scaling to higher-dimension. The one-dimensional case showed us that summations and finite support functions play a much more important role in the development than we first expected. We are therefore moving to the SSReflect interface and libraries for Coq [8], so as to simplify the manipulations of these objects in the higher-dimensional case.

Another perspective is to generalize our approach to other PDEs and ODEs. However, the proofs of Section 4 are entangled with particulars of the presented problem, and would therefore have to be redone for other problems. Our basic definitions have been tested and we think we have encountered most of the hurdles met in the formalization and proof of numerical analysis programs. In particular, we intend to generalize our method for bounding the round-off error using convolution with the discrete fundamental solution.

A more ambitious perspective is to formally deal with the finite element method. This first requires a Coq formalization of mathematical tools from Hilbertian analysis on Sobolev spaces such as the Lax-Milgram theorem (for existence and uniqueness of continuous 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 goal is to go beyond Laplace's equation set on the unit square: *e.g.* handle mixed boundary conditions, extend to mixed and mixed hybrid finite element methods.

7 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

- *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, under the assumptions that the continuous solution is sufficiently regular of order 4 uniformly 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 compensate, to get a meaningful bound at the end.
- *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. The total size of that formal proof is huge, if not frightening: several man-months of work, more annotations to be inserted in the C program than lines of code (see appendix B), over 16,000 lines of Coq proof scripts, 30 minutes of CPU time to check them.

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 a hot topic that logicians now consider important. Formal tools for computational science are actively developed and progress is done to get them mature and usable for non specialists. It seems to be the 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 2). We are also thankful to Vincent Martin for his constructive remarks on this article.

References

[1] George E. Andrews, Richard Askey, and Ranjan Roy. *Special functions*. Cambridge University Press, Cambridge, 1999.

[2] Richard Askey and George Gasper. Certain rational functions whose power series have positive coefficients. *The American Mathematical Monthly*, 79:327–341, 1972.

[3] Jeremy Avigad and Kevin Donnelly. A Decision Procedure for Linear "Big O" Equations. *Journal of Automated Reasoning*, 38(4):353–373, 2007.

[4] Clark Barrett and Cesare Tinelli. CVC3. In *19th International Conference on Computer Aided Verification (CAV '07)*, volume 4590 of *LNCS*, pages 298–302. Springer-Verlag, July 2007. Berlin, Germany.

[5] Patrick Baudin, Pascal Cuoq, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL: ANSI/ISO C Specification Language, version 1.5*, 2009.

[6] Éliane Bécache. Étude de schémas numériques pour la résolution de l'équation des ondes. Master Modélisation et simulation, Cours ENSTA, 2009.

[7] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer, 2004.

[8] Yves Bertot, Georges Gonthier, Sidi Ould Biha, and Ioana Pasca. Canonical Big Operators. In *21st International Conference on Theorem Proving in Higher Order Logics (TPHOLs'08)*, volume 5170 of *LNCS*, pages 86–101, Montreal, Canada, 2008. Springer.

[9] Sylvie Boldo. *Preuves formelles en arithmétiques à virgule flottante*. PhD thesis, École Normale Supérieure de Lyon, November 2004.

[10] Sylvie Boldo. Floats & Ropes: a case study for formal numerical program verification. In *36th International Colloquium on Automata, Languages and Programming*, volume 5556 of *LNCS - ARCoSS*, pages 91–102, Rhodos, Greece, July 2009. Springer.

[11] Sylvie Boldo, François Clément, Jean-Christophe Filliâtre, Micaela Mayero, Guillaume Melquiond, and Pierre Weis. Formal proof of a wave equation resolution scheme: the method error. In Matt Kaufmann and Lawrence C. Paulson, editors, *1st Interactive Theorem Proving Conference (ITP)*, volume 6172 of *LNCS*, pages 147–162, Edinburgh, Scotland, 2010. Springer.

[12] Sylvie Boldo, François Clément, Jean-Christophe Filliâtre, Micaela Mayero, Guillaume Melquiond, and Pierre Weis. Wave equation numerical resolution: a comprehensive mechanized proof of a c program. *Journal of Automated Reasoning*, 2012.

[13] Sylvie Boldo, Jean-Christophe Filliâtre, and Guillaume Melquiond. Combining Coq and Gappa for certifying floating-point programs. In Jacques Carette, Lucas Dixon, Claudio Sarcedoti Coen, and Stephen M. Watt, editors, *16th Calculemus Symposium*, volume 5625 of *Lecture Notes in Artificial Intelligence*, pages 59–74, Grand Bend, ON, Canada, 2009.

[14] Sylvie Boldo, Catherine Lelay, and Guillaume Melquiond. Improving Real Analysis in Coq: A User-Friendly Approach to Integrals and Derivatives. In C. Hawblitzel and D. Miller, editors, *Proceedings of the Second International Conference on Certified Programs and Proofs*, number 7679 in LNCS, pages 289–304, Kyoto, Japan, December 2012. Springer.

[15] Sylvie Boldo and Guillaume Melquiond. Flocq: A unified library for proving floating-point algorithms in Coq. In Elisardo Antelo, David Hough, and Paolo Ienne, editors, *20th IEEE Symposium on Computer Arithmetic*, pages 243–252, Tübingen, Germany, 2011.

[16] Boutheina Chetali and Quang-Huy Nguyen. Industrial use of formal methods for a high-level security evaluation. In *Proceedings of the 15th International Symposium on Formal Methods (FM'08)*, volume 5014 of *LNCS*, pages 198–213, Turku, Finland, 2008.

[17] Sylvain Conchon, Évelyne Contejean, Johannes Kanig, and Stéphane 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)*, volume 198-2 of *Electronic Notes in Computer Science*, pages 51–69. Elsevier Science Publishers, 2008.

[18] Thierry Coquand and Christine Paulin-Mohring. Inductively defined types. In P. Martin-Löf and G.Mints, editors, *Colog'88*, volume 417 of *LNCS*. Springer-Verlag, 1990.

[19] Richard Courant, Kurt Friedrichs, and Hans Lewy. On the partial difference equations of mathematical physics. *IBM Journal of Research and Development*, 11(2):215–234, 1967.

[20] Marc Daumas and Guillaume Melquiond. Certification of bounds on expressions involving rounded operators. *Transactions on Mathematical Software*, 37(1):1–20, 2010.

[21] Marc Daumas, Laurence Rideau, and Laurent Théry. A generic library for floating-point numbers and its application to exact computing. In *TPHOLs*, pages 169–184, 2001.

[22] Florent de Dinechin, Christoph Lauter, and Guillaume Melquiond. Certifying the floating-point implementation of an elementary function using Gappa. *IEEE Transactions on Computers*, 60(2):242–253, 2011.

[23] Leonardo de Moura and Nikolaj Bjørner. Z3, an efficient SMT solver. In *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[24] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

[25] Léopold Fejér. Sur le développement d'une fonction arbitraire suivant les fonctions de Laplace. *Comptes-Rendus de l'Académie des Sciences*, 146:224–227, 1908.

[26] Jean-Christophe Filliâtre and Claude Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In *19th International Conference on Computer Aided Verification*, volume 4590 of *LNCS*, pages 173–177, Berlin, Germany, July 2007. Springer.

[27] George Gasper. Positive sums of the classical orthogonal polynomials. *SIAM Journal on Mathematical Analysis*, 8(3):423–447, 1977.

[28] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23(1):5–48, March 1991.

[29] Fritz John. *Partial Differential Equations*. Springer, 1986.

[30] Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an operating system kernel. *Communications of the ACM*, 53(6):107–115, 2010.

[31] Jean 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 (Année 1747)*, volume 3, pages 214–249. Haude et Spener, Berlin, 1749.

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

[33] Claude Marché. Jessie: an intermediate language for Java and C verification. In *Programming Languages meets Program Verification (PLPV)*, pages 1–2, Freiburg, Germany, 2007. ACM.

[34] Micaela Mayero. *Formalisation et automatisation de preuves en analyses réelle et numérique*. PhD thesis, Université Paris VI, 2001.

[35] Microprocessor Standards Committee. IEEE Standard for Floating-Point Arithmetic. *IEEE Std. 754-2008*, pages 1–58, August 2008.

[36] Marko Petkovšek, Herbert S. Wilf, and Doron Zeilberger. *A=B*. A K Peters Ltd, Wellesley, MA, 1996.

[37] James William Thomas. *Numerical Partial Differential Equations: Finite Difference Methods*. Number 22 in Texts in Applied Mathematics. Springer, 1995.

[38] Doron Zeilberger. A fast algorithm for proving terminating hypergeometric identities. *Discrete Math.*, 80:207–211, 1990.

[39] Doron Zeilberger. The method of creative telescoping. *J. Symbolic Computation*, 11:195–204, 1991.

A Glossary

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

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

automated theorem prover a software tool that automatically proves logical properties. It may fail to find a proof, even for valid properties.

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

goal logical property to *formally prove*⋆. The conclusion of a 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 a 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 the meaning associated to each syntactic construct of a language.

SMT solver a 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.

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

validation process of guaranteeing the correctness of a program by experiments such as tests.

verification process of guaranteeing 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*⋆.

B Fully Annotated Source Code

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

    @ axiom c_pos: 0 < c;

    @ logic real psol_1(real x, real t);
10  @ 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;
15
    @ 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;
20  @   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 ==>
```

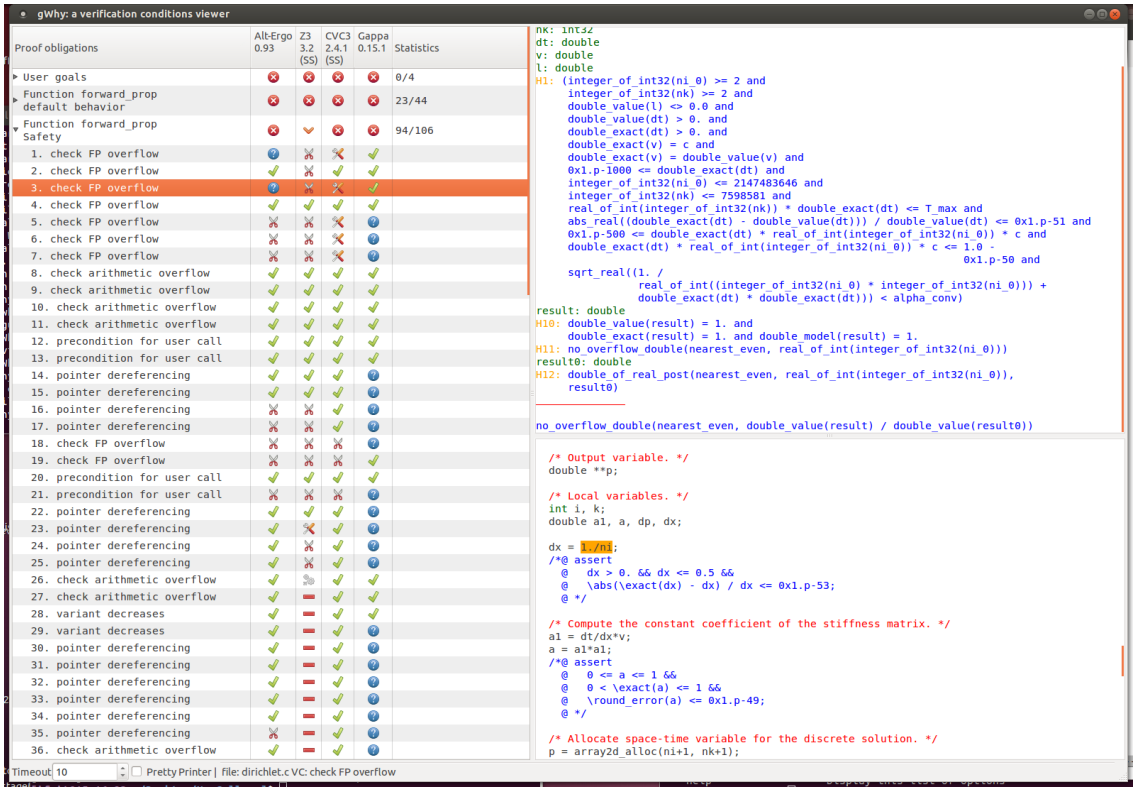
```
  @    0 < \exact(dt) ==>
  @    analytic_error(p, ni, i, k, a, dt) ==>
  @    \sqrt(1. / (ni * ni) + \exact(dt) * \exact(dt)) < alpha_conv ==>
  @    k <= nk ==> nk <= 7598581 ==> nk * \exact(dt) <= T_max ==>
  @    \exact(dt) * ni * c <= 1 - 0x1.p-50 ==>
  @    \forall integer i1; \forall integer 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;

}
```

C Screenshot

This is a screenshot of gWhy: we have the list of all the verification conditions and their proof status with respect to the various automatic tools.



D Fundamental Solution and Jacobi Polynomials

Let λ be the sequence defined in Section 5.1.2. 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:

$$\forall i, \quad \lambda_i^{-1} = 0, \tag{32}$$

$$\forall i \neq 0, \quad \lambda_i^0 = 0, \quad \lambda_0^0 = 1, \tag{33}$$

$$\forall i, \forall k \geq 0, \quad \lambda_i^{k+1} = a(\lambda_{i-1}^k + \lambda_{i+1}^k) + 2(1-a)\lambda_i^k - \lambda_i^{k-1}. \tag{34}$$

We want to prove Lemma 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 λ's, the creative telescoping method of Zeilberger [36] to express those λ's in terms of Jacobi polynomials, and finally a result by Askey and Gasper [2] 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 allows an easy validation 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 (34) rewrites

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

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

$$\begin{aligned} T\sum_i \sum_{k\geq 0} \lambda_i^{k+1} X^i T^k &= \Lambda(X,T) - 1, & \sum_i \sum_{k\geq 0} \lambda_{i-1}^k X^i T^k &= X\Lambda(X,T), \\ X\sum_i \sum_{k\geq 0} \lambda_{i+1}^k X^i T^k &= \Lambda(X,T), & \sum_i \sum_{k\geq 0} \lambda_i^k X^i T^k &= \Lambda(X,T), \\ \multicolumn{4}{c}{\sum_i \sum_{k\geq 0} \lambda_i^{k-1} X^i T^k = T\Lambda(X,T).} \end{aligned}$$

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

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

Now, we can evaluate the power series expansion of the generating function Λ. 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 Λ, we have for all i, k, $0 \leq |i| \leq k$,

$$\lambda_i^k = \sum_{n=|i|}^{n=k} \binom{2n}{n+i}\binom{n+k+1}{2n+1}(-1)^{n+i} a^n. \tag{35}$$

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 [25] 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 [2, 27] Fejér's result for $\beta \geq 0$ and $\alpha + \beta \geq -2$. See also [1], 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 (35), we are led to prove, for all i, n, k, $0 \leq i \leq n \leq k$,

$$\sum_{p=i}^{n} \sum_{q=n}^{k} \binom{q+i}{p+i} \binom{q-i}{p-i} \binom{q-p}{n-p} = \binom{2n}{n+i} \binom{n+k+1}{2n+1}. \tag{36}$$

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

$$\sum_{p=i}^{n} \binom{k+i}{p+i} \binom{k-i}{p-i} \binom{k-p}{n-p} = \binom{2n}{n+i} \binom{k+n}{2n}. \tag{37}$$

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 (37) 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 [38, 39]. 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 (37) 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, $0 \le i \le n \le k$,

$$(n+1+i)If = (n-i)f, \tag{38}$$

$$(n+1+i)(n+1-i)Nf = (k+1+n)(k-n)f, \tag{39}$$

$$(k+1-n)Kf = (k+1+n)f. \tag{40}$$

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. equations (38), (39), and (40), 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). On has just to check the simpler equations with rational expressions

$$b_{l,0} + \sum_{m=1}^{m'} b_{l,m} \frac{L^m F}{F} = P R_l \frac{PF}{F} - R_l. \tag{41}$$

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

$$\begin{array}{lll} b_{i,0} = n-i, & b_{i,1} = -(n+1+i), & R_i = \dfrac{(1+2i)(p-i)}{k-i}, \\ b_{n,0} = (k+1+n)(k-n), & b_{n,1} = -(n+1+i)(n+1-i), & R_n = \dfrac{(k-n)(p+i)(p-i)}{n+1-p}, \\ b_{k,0} = k+1+n, & b_{k,1} = -(k+1-n), & R_k = \dfrac{(p+i)(p-i)}{k+1-p}. \end{array}$$

And, simply using the symmetry and absorption properties of binomial coefficients, we successively have, for $l = i$,

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

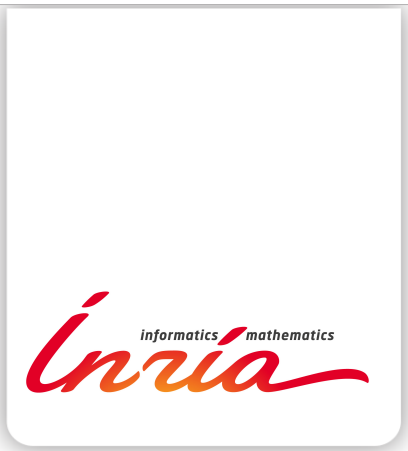
for $l = n$,

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

and for $l = k$,

$$\begin{aligned}
(41) \quad &\Leftrightarrow \quad (k+1+n) - (k+1-n)\frac{(k+1+i)}{(k+1-p)}\frac{(k+1-i)}{(k+1-p)}\frac{(k+1-p)}{(k+1-n)} \\
&\qquad = \frac{(p+1+i)(p+1-i)}{(k-p)}\frac{(k-p)}{(p+1+i)}\frac{(k-p)}{(p+1-i)}\frac{(n-p)}{(k-p)} - \frac{(p+i)(p-i)}{(k+1-p)} \\
&\Leftrightarrow \quad (k+1+n)(k+1-p) - (k+1+i)(k+1-i) = (n-p)(k+1-p) - (p+i)(p-i) \\
&\Leftrightarrow \quad (k+1)(n-p) - np + i^2 = (n-p)(k+1) - np + i^2
\end{aligned}$$

which are all true.



RESEARCH CENTRE
PARIS – ROCQUENCOURT

Domaine de Voluceau, - Rocquencourt
B.P. 105 - 78153 Le Chesnay Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399