Title: **Using Symbolic Execution to Verify the Order of Accuracy of Numerical Approximations**

Authors: Timothy K. Zirkel, Stephen F. Siegel, and Louis F. Rossi

Kind: Technical Report UD-CIS-2014/002

Status: *Submitted for publication*

Verified Software Laboratory
Department of Computer and Information Sciences
University of Delaware
Newark DE 19716
USA
`http://vsl.cis.udel.edu`

Using Symbolic Execution to Verify the Order of Accuracy of Numerical Approximations

Timothy K. Zirkel[1], Stephen F. Siegel[1], and Louis F. Rossi[2]

[1] Department of Computer and Information Sciences, University of Delaware, USA,
{zirkeltk|siegel}@udel.edu
[2] Department of Mathematical Sciences, University of Delaware, USA,
rossi@math.udel.edu

Abstract. The order of accuracy of a numerical method relates the scheme's error to the discretization parameters. Scientists must know the order of accuracy of any numerical approximation, and often prove that the method satisfies the claimed order of accuracy by hand. However, the actual code to implement a method might be more complex and veer from the abstract mathematics. We show that the claimed order of accuracy of a numerical method implemented in a C program can be (largely) automatically verified using formal methods. The automation cannot be complete, because the problem is undecidable in general and because the programmer must provide some minimal annotations relating the code to the underlying mathematics. We have implemented the technique in the Concurrency Intermediate Verification Language model checker, and demonstrated its effectiveness on several finite difference method examples.

1 Introduction

There are a variety of research avenues dealing with the specification and verification of numerical programs.

One avenue is equivalence checking. That technique takes two programs and attempts to establish their functional (input-output) equivalence. Typically, one program is "trusted" and serves as the specification of an algorithm, while the other is a complex, optimized, possibly parallel, implementation. This is the main technique used by TASS [21]. Equivalence checking can be effective for both floating-point and real number notions of equivalence.

Another approach uses rich specification languages to formulate assertions and code contracts concerning the numerical computations in a program. This is the approach taken by Frama-C [2]. The formulas can specify precise relationships between inputs and outputs to functions, and can refer to both the floating-point and real semantics of numeric operations; they can be verified using deductive techniques which rely on automated theorem provers and/or proof assistants. This approach has been particularly effective at verifying precise bounds on round-off errors.

One important aspect of numerical programs that has received relatively little attention in these research efforts is the notion of *order of accuracy*. This concept is essential to the analysis of a broad range of numerical programs, especially partial differential equation solvers. The order of accuracy of an algorithm is an integer which measures how quickly the solution computed using a discrete approximation converges to the continuous mathematical solution as grid resolution increases. In particular, order of accuracy deals with "discretization error" and depends solely on the real (not floating-point) semantics of the code. Since discretization error often dominates the error in numerical computations, it is seen as essential to get the order of accuracy "right" before focusing on floating-point error.

Many journals have strict requirements concerning the order of accuracy of methods presented in their submissions. The American Institute of Aeronautics and Astronautics requires that any article appearing in one of its journals that deals with the numerical solution to PDEs "should state the formal accuracy of the numerical method for interior points as well as the formal accuracy of the numerical boundary conditions," which should be "at least formally second-order accurate" and that "some level of verification testing" be performed on implementations [1]. The Journal of Fluids Engineering requires "[t]he numerical method used must be at least formally second-order accurate in space (based on a Taylor series expansion) for nodes in the interior of the computational grid" [14]. Authors are usually expected to carry out testing-based strategies which vary parameters in order to ascertain that the code meets the theoretical order of accuracy, but these are subject to well-known limitations of testing and cannot provide a proof.

We present a new technique, based on symbolic execution, for verifying the claimed order of accuracy of a numerical method. This technique takes as input an annotated C program implementing the method, but it treats all of the floating-point computations in the programs as full precision (mathematical) operations. The annotations specify the input-output signature of the program, as well as accuracy claims, such as "the output u is 3^{rd}-order accurate in input x and 2^{nd}-order accurate in input t."

Section 2 provides some background on numerical analysis. Section 3 describes the CIVL model checker used to demonstrate our technique. Section 4 analyzes several example problems. Section 5 discusses related work. Section 6 gives experimental results. Section 7 presents conclusions.

2 Formal Treatment of Concepts in Numerical Analysis

Numerical methods involve taking a problem from a continuous domain and accurately approximating it using a discrete set of parameters. This discretization introduces error. Analyzing this error is an essential component of any investigation using a numerical scheme. In this section, we provide precise definitions for the numerical concepts we wish to treat formally in programs.

2.1 Asymptotic behavior and order of accuracy

We begin by discussing the asymptotic behavior of functions. First we will look at functions of one variable. The following are standard; see for example [15].

Definition 1 (Big-O). *Let $a > 0$ and $I = (0, a)$. Suppose we have two functions $\phi : I \to \mathbb{R}$ and $\psi : I \to \mathbb{R}$. We write*

$$\phi(h) = O(\psi(h)) \text{ as } h \to 0$$

if there exist positive real numbers C and ϵ such that $|\phi(h)| \leq C|\psi(h)|$ whenever $0 < h < \epsilon$.

In the following definitions, assume $a > 0$, $I = (0, a)$, and $D \subseteq \mathbb{R}$.

Definition 2 (Order of Accuracy). *Let n be a positive integer. Given a function $f : D \to \mathbb{R}$, consider a function $g : D \times I \to \mathbb{R}$. Fix $x \in D$. We say g is an n^{th}* order accurate approximation to f at x *if*

$$f(x) - g(x, h) = O(h^n) \text{ as } h \to 0.$$

The idea is that the left hand side is approaching 0 at least as fast as h^n. The order of accuracy quantifies the rate at which the numerical method will converge to the exact solution as h decreases.

Notice that the constants in Def. 2 are dependent on the particular point x. A stronger notion is of having a single ϵ and C for the entire domain. This is the concept of *uniformly n^{th} order accurate*.

Definition 3 (Uniform Order of Accuracy). *Let n be a positive integer, $f : D \to \mathbb{R}$, and $g : D \times I \to \mathbb{R}$. Define $\phi : I \to \mathbb{R}$ by*

$$\phi(h) = \sup_{x \in D} |f(x) - g(x, h)|.$$

We say that g is a uniformly n^{th} order accurate approximation of f on D *if*

$$\phi(h) = O(h^n) \text{ as } h \to 0.$$

Clearly if g is uniformly n^{th} order accurate on f, then it is n^{th} order accurate at each point. However, the converse is not necessarily true. The stronger condition is usually the desired one.

2.2 Grid approximations

Definition 3 is often used in the analysis of finite difference methods where the solution is approximated on a grid. A grid is a discrete subset of the domain where an approximate solution is computed. For example, a domain of interest might be an interval $D = [b_0, b_1]$. One possible discretization of this domain is choosing m uniformly spaced points to form $\Delta(h) = \{b_0 + hk | 0 \leq k \leq m\}$ where $h = b_1/m$. Note that the grid resulting from a smaller h is not necessarily a refinement of that resulting from a larger h (e.g., the grid from $h = 1/4$ does not refine the grid from $h = 1/3$). For a convergent scheme, the approximation converges to the exact solution on this discrete set, as we refine the grid.

Definition 4 (n^{th} **order** Δ **convergence).** *Let n be a positive integer, $D \subseteq \mathbb{R}$, $f\colon D \to \mathbb{R}$, $a > 0$, and $I = (0, a)$. Suppose $\Delta\colon I \to \wp(D)$, where $\wp(D)$ is the set of all subsets of D. Let $S = \bigcup_{h \in I} (\Delta(h) \times \{h\}) \subseteq D \times I$. Suppose $g\colon S \to R$. Define $\phi\colon I \to \mathbb{R}$ by*

$$\phi(h) = \sup_{x \in \Delta(h)} |f(x) - g(x, h)|.$$

We say g is a Δ-uniformly n^{th} order accurate approximation of f *if*

$$\phi(h) = O(h^n) \text{ as } h \to 0.$$

Given the parameter h, Δ returns a subset of D that is the grid. For problems of interest in our analysis, the grid will typically consist of evenly spaced points. We are interested in a numerical method's error as h goes to 0.

2.3 Functions of several variables

Functions of multiple variables may have different orders of accuracy in each variable. This may be represented by a separate big-O term for each variable.

Definition 5. *Let $I_0 = (0, a), I_1 = (0, b)$, with $a, b > 0$. Suppose we have functions $\phi : I_0 \times I_1 \to \mathbb{R}$, $\psi_0 : I_0 \to \mathbb{R}$, and $\psi_1 : I_1 \to \mathbb{R}$. We write*

$$\phi(h_0, h_1) = O(\psi_0(h_0)) + O(\psi_1(h_1)) \text{ as } h_0 \to 0 \text{ and } h_1 \to 0$$

if there exist positive real numbers $C_0, C_1, \epsilon_0, \epsilon_1$ such that

$$|\phi(h_0, h_1)| \leq C_0|\psi_0(h_0)| + C_1|\psi_1(h_1)|$$

whenever $0 < h_0 < \epsilon_0$ and $0 < h_1 < \epsilon_1$.

Definitions 2, 3, and 4 generalize to several variables in the obvious way. For these definitions, we either give the accuracy for each variable separately or say the approximation is accurate of order $(n_0, n_1, \ldots, n_m)$.

3 CIVL Model Checker

A proof of concept for the technique presented in this paper is implemented in the Concurrency Intermediate Verification Language (CIVL) model checker [6]. The CIVL tool uses model checking and symbolic execution to verify assertions and safety properties about programs written in CIVL-C. CIVL-C is an extension to C that provides a variety of useful features for implementing and verifying models of various languages and parallel frameworks. CIVL is a good fit for our approach to accuracy verification because of its ability to simplify, manipulate, and reason about real-valued mathematical expressions, including (multivariate) polynomials and quotients of polynomials. It also treats all floating-point values as full precision rational numbers, so error in the numerical method can be verified independently of floating-point error.

The technique in this paper takes advantage of several of the statements and expressions in CIVL-C. Note that CIVL-C keywords begin with `$`.

- *Quantified expressions:* CIVL-C supports universally and existentially quantified expressions.
- *Input variables:* Variables declared with the modifier `$input` are treated as symbolic inputs to the CIVL-C program. Unless an explicit value is given at the command line, an input variable will be initialized to a new symbolic constant during the symbolic execution.
- *Assumptions:* CIVL-C provides an `$assume` statement which allows the programmer to specify information that should be added to the path condition. For example, one could use assumptions to specify bounds on input variables.
- *Assertions:* The CIVL-C `$assert` is much like assertions in other languages, but can contain quantifiers and other CIVL-C-specific expressions.

In addition, we modified CIVL to include some additional features that are helpful for accuracy verification.

- *Big-O terms:* A new expression `$O(e)`, along with modifications to the underlying symbolic algebra library to support arithmetic on big-O expressions.
- *Abstract functions:* Abstract functions are uninterpreted and represent mathematical functions. Like C functions, they have a type associated with each argument and a return type, but they also have information about continuity. Abstract functions are declared with the keyword `$abstract`. The declaration includes a parameter `$contin(n)` which indicates that n derivatives exist, are continuous, and are bounded on the domain of consideration.
- *Real type:* As mentioned above, all floating point types are mapped to full precision rational numbers. However, the built-in types in CIVL-C are the same as C types. For specifying abstract functions it is more accurate and intuitive to have an explicit `$real` type.
- *Partial derivatives:* Since abstract functions represent mathematical functions, it is possible to take their derivative. The notation for a derivative in CIVL-C is based on the notation used in Mathematica [25] and allows for the specification of the number of partial derivatives taken for each parameter. e.g. `$D[`u`,{`x`,2}](`y`)` is the second partial derivative with respect to x of the function u, evaluated at the point y.
- *Uniform:* A new quantifier `$uniform` designed to capture the uniformity criteria in Definition 4.

All of our modifications have been committed to the CIVL open source project in version 0.7.

4 Examples

We present some simple examples here. For each example, we begin by discussing the mathematical analysis of the algorithm and then provide CIVL-C code with annotations. One of our objectives was to develop annotations that are both useful to the verifier and easy for the programmer to add.

The mathematical analysis described in each example is presented to give an idea of the kind of reasoning the tool needs to perform. The programmer does

not need to work out these equations in order to use the tool. Given the minimal annotations described below, the tool will perform the reasoning automatically.

4.1 Example: Differentiation

Mathematical Analysis. Our first example is a centered difference approximation to the first derivative.

Let $\rho : \mathbb{R} \to \mathbb{R}$ be three times differentiable and assume there exists $M > 0$ such that $|\rho'''(x)| < M$ for all x. A discrete approximation of the derivative is given by

$$g(x,h) \equiv \frac{\rho(x+h) - \rho(x-h)}{2h}. \quad (1)$$

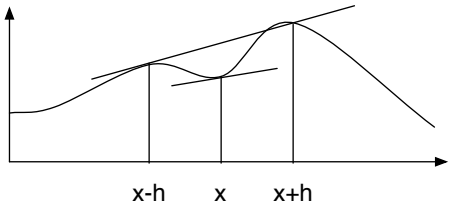

Fig. 1. Central differencing approximates the derivative of $f(x)$ as the slope through the points $f(x-h)$ and $f(x+h)$.

In Definition 3, $f(x) = \rho'(x)$. We claim that g is a uniformly 2^{nd} order approximation for f on $\mathbb{R}$. To verify, we use Taylor polynomials with Lagrangian remainders. Given $x \in \mathbb{R}$ and $h > 0$, there exist $\xi_1, \xi_2 \in [x-h, x+h]$ such that

$$\rho(x+h) = \rho(x) + \rho'(x)h + \frac{1}{2}\rho''(x)h^2 + \frac{1}{6}\rho'''(\xi_1)h^3$$
$$\rho(x-h) = \rho(x) - \rho'(x)h + \frac{1}{2}\rho''(x)h^2 - \frac{1}{6}\rho'''(\xi_2)h^3.$$

From this we conclude

$$\left|\frac{\rho(x+h) - \rho(x-h)}{2h} - \rho'(x)\right| = \frac{|\rho'''(\xi_1) + \rho'''(\xi_2)|}{12}h^2 \leq \frac{1}{6}Mh^2.$$

We see that the difference between the discrete approximation and the exact derivative is bounded by a constant times h^2, and thus g is a uniformly 2^{nd} order accurate approximation of f on $\mathbb{R}$.

Code Annotations. In order to connect the code to the math, programmers may annotate programs with additional specification information.

Fig. 2 is an excerpt of CIVL-C code. The code takes as input h and an array which holds values of the function at the points ih, where $i \in \mathbb{Z}$ and $0 \leq i \leq n$. These points form the grid $\Delta(h)$ from Definition 4. The values stored in `result` at the end are the output of g_h. Since central differencing cannot be performed on the endpoints, we use forward and backward differencing at the first and last positions, respectively. These are first order accurate methods. Hence the result is not Δ-uniformly second order accurate. Instead we will only consider points in the interior of the domain, which we will call Δ'. Define $\Delta'(h) = \{ih | 1 \leq i < n-1\}$. All points in Δ' are then computed using central

```
1  $input double h;
2  $input int num_elements;
3  $input double initial[num_elements];
4  $abstract $contin(3) $real rho($real x);
5  $assume h > 0;
6
7  void differentiate(int n, double y[], double result[]){
8    $assume $forall {m=0 .. n-1} y[m] == rho(m*h);
9    for(int i = 1; i < n-1; i++)
10   result[i] = (y[i+1]-y[i-1])/(2*h);
11   result[0] = (y[1]-y[0])/h;
12   result[n-1] = (y[n-1] - y[n-2])/h;
13   $assert($uniform{k=1 .. n-2} result[k]-$D[rho,{x,1}](k*h)==$O(h*h));
14 }
```

Fig. 2. Annotated CIVL-C code for differentiation. The code does central differencing on the interior of the array and forward/backward differencing for the endpoints.

differencing. Note that the bounds on i are the same as the bounds on the quantified variable `k` in the assertion at line 13.

Three annotations are needed to provide the accuracy specification. Line 4 is a declaration of the abstract function ρ, which is declared to have 3 continuous, bounded derivatives. The assumption at line 8 relates the values in the input array `y` to the function ρ. Line 13 is the assertion about the relationship between `result` and the actual derivative. The `$uniform` quantifier has the same syntax as the existential or universal quantifiers. Its general form is

`$uniform {`*decl*`=`*lower* `..` *upper*`}` *expression* `=` *O-expression*,

where *decl* declares the name of a variable v, *lower* and *upper* are expressions bounding the values of v. `$uniform` indicates that the constant C (from Definition 1) in the big-O term is independent of any local variables or any values inside of the big-O. At line 13, this indicates that the code is Δ'-uniformly 2^{nd} order accurate.

By applying heuristics to the given annotations, the verifier can automatically add information about Taylor expansions of ρ to the model, and then prove that the results computed by the code are Δ'-uniformly 2^{nd} order accurate.

4.2 Example: Second Derivative

Mathematical Analysis. Approximating a second derivative using central differencing has a similar analysis to Section 4.1 above, but requires an additional term in the Taylor expansions of ρ. Let $\rho : \mathbb{R} \to \mathbb{R}$ be four times differentiable. Suppose there exists $M > 0$ such that $|\rho''''(x)| < M$ for all x. The approximation is given by

$$g(x,h) \equiv \frac{\rho(x+h) - 2\rho(x) + \rho(x-h)}{h^2}. \tag{2}$$

In Definition 3, $f(x) = \rho''(x)$. We claim that g is a uniformly 2^{nd} order approximation for f on $\mathbb{R}$. To verify, we use Taylor polynomials with Lagrangian remainders. Given $x \in \mathbb{R}$ and $h > 0$, there exist $\xi_1, \xi_2 \in [x-h, x+h]$ such that

$$\rho(x+h) = \rho(x) + \rho'(x)h + \frac{1}{2}\rho''(x)h^2 + \frac{1}{6}\rho'''(x)h^3 + \frac{1}{24}\rho''''(\xi_1)h^4$$
$$\rho(x-h) = \rho(x) - \rho'(x)h + \frac{1}{2}\rho''(x)h^2 - \frac{1}{6}\rho'''(x)h^3 + \frac{1}{24}\rho''''(\xi_2)h^4.$$

From this we compute the accuracy:

$$\left|\frac{\rho(x+h) - 2\rho(x) + \rho(x-h)}{h^2} - \rho''(x)\right| = \frac{|\rho'''(\xi_1) + \rho'''(\xi_2)|}{24}h^2 \leq \frac{1}{12}Mh^2.$$

We see that the difference between the discrete approximation and second derivative is bounded by a constant times h^2, and thus g is a uniformly 2^{nd} order accurate approximation of f on $\mathbb{R}$.

```
1 $input double h;
2 $input int num_elements;
3 $input double initial[num_elements];
4 $abstract $contin(4) $real rho($real x);
5 $assume h > 0;
6
7 void secondDerivative(double h, int n, double y[], double result[]){
8   $assume $forall {m=0 .. n-1} y[m] == rho(m*h);
9   for(int i = 1; i < n-1; i++)
10    result[i] = (y[i+1]-2*y[i]+y[i-1])/(h*h);
11  result[0] = (y[2]-2*y[1]+y[0])/h;
12  result[n-1] = (y[n-3] - 2*y[n-2]-y[n-1])/h;
13  $assert($uniform{k=1 .. n-2} result[k]-$D[rho,{x,2}](k*h)==$O(h*h));
14 }
```

Fig. 3. Annotated CIVL-C code for second derivative. The code does central differencing on the interior of the array and forward/backward differencing for the endpoints.

Code Annotations. Fig. 3 is an excerpt of the CIVL-C code for computing the second derivative. As in Section 4.1, the code takes the grid separation h and an array of inputs holding the values of $\rho(ih)$.

Line 4 is the declaration of the abstract function ρ. Unlike the previous example, ρ is specified here to have four continuous derivatives. In practice this ρ might be the same ρ as the differentiation examples, but specifying four continuous derivatives gives the verifier a useful hint on how many terms to create in the Taylor expansions. The assumption at line 8 tells the verifier the relationship between the array values and the function ρ. Line 13 asserts that the approximation is uniformly 2^{nd} order accurate in h.

4.3 Example: Laplace Operator

Mathematical Analysis. The Laplace operator ∇^2 is a differential operator that is useful for describing a wide range of problems in mathematics, science and engineering. The two-dimensional cartesian form is

$$\nabla^2 u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}.$$

Using a grid size of h in both dimensions, the Laplace operator can be approximated using a finite difference scheme on the five point stencil shown in Fig. 4. The approximation is given by

$$g(x, y, h) \equiv \frac{u(x-h, y) + u(x, y-h) - 4u(x, y) + u(x+h, y) + u(x, y+h)}{h^2}.$$

The truncated Taylor polynomials used here are analogous to those in Section 4.2, but expansions must happen about $u(x, y)$ in both the x and y directions. Substituting the appropriate expansions into the finite difference scheme and simplifying shows that the approximation is $O(h^2)$ accurate.

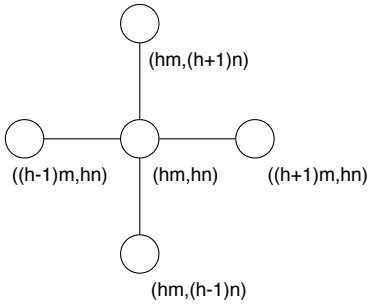

Fig. 4. Five point stencil for the 2D Laplace operator.

Code Annotations. Code for computing the Laplace operator is given in Fig. 5. The abstract function $\phi(x, y)$ is declared at line 6. It indicates that four bounded, continuous derivatives may be taken. The assumption at line 7 equates elements in the two-dimensional array `u` to values of the abstract function. Note that variables from the quantifiers are involved in both arguments to ϕ. That information assists the verifier in making choices about which Taylor polynomials to create. Lines 14–15 assert that the result is uniformly 2^{nd} order accurate in h.

4.4 Example: Diffusion

Mathematical analysis. Next we consider solving the 1-dimensional diffusion equation. We again begin by describing the math to give insight into what the tool must do automatically. This example is a differential equation presented in terms of a differential operator. Solutions to differential equations may be numerically computed using approximation schemes for the differential operator. We present the notion of order of accuracy for a numerical approximation scheme.

In the discussion below, $Func(X, Y)$ denotes the set of functions from X to Y. If X and Y are continuous domains, we assume the functions are sufficiently smooth to take the necessary number of derivatives.

```
$input double h;
$input int rows, cols;
$input double u[rows][cols];
double result[rows][cols];
$assume h > 0;
$abstract $contin(4) $real phi($real x, $real y);
$assume $forall{m=0..rows-1} $forall{n=0..cols-1} u[m][n]==phi(m*h,n*h);

void laplace() {
  for (int i=1; i < rows-1; i++)
    for (int j=1; j < cols-1; j++)
      result[i][j]=(u[i-1][j]+u[i][j-1]-4*u[i][j]+u[i+1][j]+u[i][j+1]) \
                   /(h*h);
  $assert($uniform{i=1 .. rows-2} $uniform{j=1 .. cols-2} result[i][j]-\
          ($D[phi,{x,2}](i*h,j*h)+$D[phi,{y,2}](i*h,j*h))==$O(h*h));
}
```

Fig. 5. Annotated CIVL-C code for the Laplace operator in two dimensions. The code does central differencing on the interior of the array. The boundary is held constant.

Definition 6 (Accuracy of a Scheme). *Let n be a positive integer, $D \subseteq \mathbb{R}$, and $L : Func(D, \mathbb{R}) \to Func(D, \mathbb{R})$. Let $I = (0, a)$, where a is a positive real number and suppose $\Delta : I \to \wp(D)$. Let $r_{\Delta_h} : Func(D, \mathbb{R}) \to Func(\Delta(h), \mathbb{R})$ be the operator which restricts a function to $\Delta(h)$. Suppose for each h there is an operator $\hat{L}_h : Func(\Delta(h), \mathbb{R}) \to Func(\Delta(h), \mathbb{R})$. For any smooth function $u : D \to \mathbb{R}$, define $\psi_u : I \to \mathbb{R}$ by*

$$\psi_u(h) = \sup_{x \in \Delta(h)} \left| r_{\Delta_h}[L[u]](x) - \hat{L}_h[r_{\Delta_h}[u]](x) \right|.$$

*We say $\hat{L}$ is a Δ-*uniformly n^{th} order accurate scheme for L *if for all u*

$$\psi_u(h) = O(h^n) \text{ as } h \to 0.$$

See [23] Def. 3.1.1. This is the special case when the forcing function $f = 0$.

This definition can be generalized to multiple variables by defining an appropriate Δ and modifying ψ accordingly.

We now apply the notion of accuracy of a scheme to the diffusion example. Let $D \subseteq \mathbb{R} \times \mathbb{R}$. Define an operator L which takes a function $v : D \to \mathbb{R}$ that is twice differentiable in x and once in t and returns a function $L[v] : D \to \mathbb{R}$ as follows:

$$L[v] = \frac{\partial v}{\partial t} - \kappa \frac{\partial^2 v}{\partial x^2} \tag{3}$$

where κ is a positive constant. The 1-dimensional diffusion equation is the equation $L[u] = 0$. A typical problem specifies boundary conditions in addition to

the basic equation $L[u] = 0$. The goal is to find a function u which satisfies all of these constraints. We will also suppose that the solution u must be four times differentiable in x and twice differentiable in t.

Given positive real numbers h_0, h_1, we define a uniform mesh

$$\Delta = \Delta(h_0, h_1) = \{(ih_0, nh_1) \in D | i, n \in \mathbb{Z}\}. \tag{4}$$

Given a function f from Δ to $\mathbb{R}$, we will write f_i^n for the value of f at the point (ih_0, nh_1). We define a restriction operator r_Δ which takes a function $v : D \to \mathbb{R}$ and returns the restriction $r_\Delta[v]$ of v to Δ. To summarize:

$$r_\Delta[v]_i^n = v(ih_0, nh_1) \tag{5}$$

where i and n are integers.

Using finite differences, we obtain the discretized operator $\hat{L} = \hat{L}_{h_0,h_1}$ (we will typically omit the subscript for brevity) as follows. $\hat{L}$ takes a function $\hat{v} : \Delta \to \mathbb{R}$ and returns a function $\hat{L}[\hat{v}] : \Delta \to \mathbb{R}$ and is defined by

$$\hat{L}[\hat{v}]_i^n = \frac{\hat{v}_i^{n+1} - \hat{v}_i^n}{h_1} - \kappa \frac{\hat{v}_{i+1}^n - 2\hat{v}_i^n + \hat{v}_{i-1}^n}{h_0^2}. \tag{6}$$

We claim that $\hat{L}$ is a scheme for L that is Δ-uniformly accurate of order (2,1). That is, it is second order accurate in h_0 and first order accurate in h_1. In order to show this, we must do several Taylor expansions. Given $i, n \in \mathbb{Z}$ and $h_0, h_1 > 0$, there exist $\xi_0, \xi_1 \in [(i-1)h_0, (i+1)h_0]$ and $\xi_2 \in [(n-1)h_1, (n+1)h_1]$ such that the following hold:

$$\begin{aligned} r_\Delta[u]_{i+1}^n &= u(ih_0, nh_1) + h_0 \frac{\partial u}{\partial x}(ih_0, nh_1) + \frac{1}{2}h_0^2 \frac{\partial^2 u}{\partial x^2}(ih_0, nh_1) \\ &\quad + \frac{1}{6}h_0^3 \frac{\partial^3 u}{\partial x^3}(ih_0, nh_1) + \frac{1}{24}h_0^4 \frac{\partial^4 u}{\partial x^4}(\xi_0, nh_1) \end{aligned} \tag{7}$$

$$\begin{aligned} r_\Delta[u]_{i-1}^n &= u(ih_0, nh_1) - h_0 \frac{\partial u}{\partial x}(ih_0, nh_1) + \frac{1}{2}h_0^2 \frac{\partial^2 u}{\partial x^2}(ih_0, nh_1) \\ &\quad - \frac{1}{6}h_0^3 \frac{\partial^3 u}{\partial x^3}(ih_0, nh_1) + \frac{1}{24}h_0^4 \frac{\partial^4 u}{\partial x^4}(\xi_1, nh_1) \end{aligned} \tag{8}$$

$$r_\Delta[u]_i^{n+1} = u(ih_0, nh_1) + h_1 \frac{\partial u}{\partial t}(ih_0, nh_1) + \frac{1}{2}h_1^2 \frac{\partial^2 u}{\partial t^2}(ih_0, \xi_2). \tag{9}$$

Assume that on the domain of interest, the absolute value of the fourth derivative of u with respect to x is bounded by $M_0 > 0$ and the absolute value of the second derivative of u with respect to t is bounded by $M_1 > 0$. Substituting the expansions into (6),

$$\begin{aligned} \hat{L}[r_\Delta[u]]_i^n &= \frac{\partial u}{\partial t}(ih_0, nh_1) - \kappa \frac{\partial^2 u}{\partial x^2}(ih_0, nh_1) + \frac{1}{2}h_1 \frac{\partial^2 u}{\partial t^2}(ih_0, \xi_2) \\ &\quad + \frac{1}{24}h_0^2 \left(\frac{\partial^4 u}{\partial x^4}(\xi_0, nh_1) + \frac{\partial^4 u}{\partial x^4}(\xi_1, nh_1)\right). \end{aligned} \tag{10}$$

We subtract $L[u]$ from (10) and apply Def. 5 to get the desired result:

$$\hat{L}[r_\Delta[u]]_i^n - r_\Delta[L[u]]_i^n = O(h_1) + O(h_0^2). \tag{11}$$

Code annotations. Fig. 6 gives an example snippet of a diffusion CIVL-C code annotated for accuracy. The view of the computation in the program is slightly different than the presentation of the math above. In the code, time steps are computed iteratively. This involves rearranging (6). Since $\hat{L}[\hat{u}] = 0$,

$$\hat{u}_i^{n+1} = \hat{u}_i^n + h_1\kappa\frac{\hat{u}_{i+1}^n - 2\hat{u}_i^n + \hat{u}_{i-1}^n}{h_0^2}. \tag{12}$$

```
1 $input int n; /* Number of points */
2 $input double h; /* Distance between points */
3 $input double dt; /* Size of a time step */
4 $input double k; /* Constant for rate of diffusion */
5 $abstract $contin(4) double u(double x, double t);
6 $assume h > 0 && dt > 0 && k > 0;
7 double v[n], v_new[n];
8 int iter;
9
10 void update() {
11   $assume $forall {j=0 .. n-1} v[j] == u(j*h, iter*dt);
12   for (int i = 1; i < n-1; i++)
13     v_new[i] = v[i]+dt*k*(v[i+1]-2*v[i]+v[i-1])/(h*h);
14   for (int i = 1; i < n-1; i++)
15     v[i] = v_new[i];
16   $assert($uniform{m=1 .. n-2} (u(m*h, (iter+1)*dt)-v[m])/dt \
17    -$D[u,{t,1}](m*h,iter*dt)+k*$D[u,{x,2}](m*h,iter*dt)==$O(dt)+$O(h*h));
18 }
```

Fig. 6. Annotated CIVL-C code for iterative diffusion in one dimension.

Define $\Phi : Func(\Delta, \mathbb{R}) \to Func(\Delta, \mathbb{R})$ by

$$\Phi[\hat{v}] = \hat{v}_i^n + h_1\kappa\frac{\hat{v}_{i+1}^n - 2\hat{v}_i^n + \hat{v}_{i-1}^n}{h_0^2}$$

This is what is actually computed in Fig. 6 at lines 16–17. CIVL can extract Φ by symbolically executing the update function.

We define a shift operator $S : Func(\Delta, \mathbb{R}) \to Func(\Delta, \mathbb{R})$ by $S[\hat{v}]_i^n = \hat{v}_i^{n+1}$. Eq. (12) then becomes $S[\hat{u}] = \Phi[\hat{u}]$. We rewrite $\hat{L}$ in terms of S and Φ to get

$$\hat{L}[\hat{v}] = \frac{S[\hat{v}] - \Phi[\hat{v}]}{h_1}. \tag{13}$$

Note that $\hat{L}[\hat{u}] = 0$ (since $\hat{u}$ satisfies $S[\hat{u}] = \Phi[\hat{u}]$), but $\hat{L}[\hat{v}]$ is not necessarily 0 for arbitrary $\hat{v}$.

Since Φ is what the code actually computes, we want to rewrite (11) in terms of Φ. We use (13) to get

$$\hat{L}[r_\Delta[u]]_i^n - r_\Delta[L[u]]_i^n = \frac{S[r_\Delta[u]]_i^n - \Phi[r_\Delta[u]]_i^n}{h_1} - r_\Delta[L[u]]_i^n. \tag{14}$$

The assertion in the code is

$$\frac{S[r_\Delta[u]]_i^n - \Phi[r_\Delta[u]]_i^n}{h_1} - r_\Delta[L[u]]_i^n = O(h_1) + O(h_0^2). \tag{15}$$

By phrasing the assertion in this way, the information about $\hat{L}$ is determined implicitly.

5 Related Work

Numerical approximation schemes are a crucial tool for providing solutions to hard mathematical problems, and require careful analysis of their accuracy. Several methods exist for analyzing accuracy and stability properties for various classes of problems. These include the modified equation approach [24], backward error analysis [12, 17], and grid convergence error analysis [19]. Often, an approximation scheme must handle the boundary of a domain differently from the interior points, requiring additional analysis of accuracy and stability [22]. For a parallel algorithm implemented using domain decomposition, extra care must be taken to account for all boundaries of the subdomains [18]. Our approach can be applied to analyzing the accuracy of numerical solutions to problems with or without boundary conditions. We do not address the issue of stability, which is much trickier to prove in general.

In addition to error from approximation schemes, scientific program results contain error introduced by floating point operations. Even simple operations such as summation can be approached in different ways, with different resulting accuracy [13]. As noted above, our technique treats all operations as full precision. However, several of the tools described below can determine bounds on this type of error.

ASTRÉE is an abstract interpretation-based static analyzer for a subset of C [8]. It can reason precisely about floating-point and limited-precision integer arithmetic and verify absence of many runtime errors, but does not deal with dynamic memory allocation or recursion; its main applications have been to real-time embedded software. FLUCTUAT [10] is another AI-based static analyzer providing information about rounding errors in C programs.

KLEE [9] is a symbolic execution tool for generating tests. It can check functional equivalence in some cases, but does not deal with parallel programs, and uses "bit-precise" reasoning instead of mathematical real arithmetic. KLEE-FP [7] treats floating-point values as uninterpreted symbolic expressions, and so cannot perform algebraic operations.

The Why/Krakatoa/Caduceus [11] and Frama-C [2] frameworks provide a set of tools for checking Java and C programs. These use special comments or annotations to specify numerical accuracy requirements.

Model checking techniques have successfully verified safety properties and functional equivalence for mature scientific codes [20]. This project connects the code to the numerical scheme.

The results of a comprehensive formal verification of a 1d wave equation code were reported in [5]. Using the Frama-C platform and various automated theorem provers and proof assistants, the authors of that study produced a mechanized proof of all correctness properties of the program, including convereegence, order of accuracy, and bounds on floating-point error. This work differs from ours in two significant ways. First, many of the proofs required extensive interaction with the proof assistant. For example, a 5000-line long Coq proof of method error was required (though around half of this may be re-usable for similar problems), and 32 verification conditions required Coq interaction to discharge. Second, the annotational burden is much larger than in our approach: for example, there are 174 lines of annotations (including axioms, lemmas, and definitions) added to a program which is 32 lines of uncommented C code (see `http://fost.saclay.inria.fr/coq_total/dirichlet.c.html`). In contrast, our method is fully automatic after adding the relatively small number of annotations we have shown. On the other hand, the method of [5] provides an extremely high level of assurance (a proof based on first principles, with no bounds on input parameters), and deals with floating-point issues in addition to order of accuracy.

6 Experimental Results

We have modified the CIVL verifier to monitor the model building process and automatically insert truncated Taylor polynomials where appropriate. The expansions are inserted as assumptions relating the evaluation of an abstract function at a particular point to its truncated Taylor polynomial. Heuristics are used to determine when to insert expansions and around which points.

Consider lines 4 and 8 from Fig. 2. From the declaration of the abstract function the verifier knows that three derivatives of ρ exist and are continuous. The assumption provides a clue about which points to expand around. In this case, the verifier recognizes the argument to the abstract function as having the form of a bound variable from a quantifier expression (m) times an expression (h). The new assumptions will quantify over the same range, and expand around points $(m+1)h$ and $(m-1)h$. The automatically generated assumptions are:

```
$assume $forall {m=0 .. n-1} rho((m+1)*h)==rho(m*h)+$D[rho,{x,1}](m*h)*h
         +$D[rho,{x,2}](m*h)*h*h/2+$O(h*h*h);
$assume $forall {m=0 .. n-1} rho((m-1)*h)==rho(m*h)-$D[rho,{x,1}](m*h)*h
         +$D[rho,{x,2}](m*h)*h*h/2+$O(h*h*h);
```

The code at lines 6 and 7 in Fig. 5 leads the verifier to create more and higher order Taylor polynomials. The specification of four continuous derivatives in the definition of ϕ tells the verifier to expand the Taylor polynomial until the $O(h^4)$

term. The forms of the arguments in the assumption indicate to the verifier that expansions are necessary for each parameter, so four assumptions are added.

For the diffusion example, the verifier adds Taylor expansions similarly to the previous examples for the spacial parameter. The temporal parameter requires additional heuristics. In iterative numerical methods, the temporal parameter often has the form of an integer iterator counter ($iter$) times the input variable time step size (dt). When the verifier encounters such an argument to an abstract function f, it makes the guess that there is forward differencing in the scheme and adds the Taylor polynomial of degree two for $f(..., (iter + 1)dt, ...)$.

The proofs all involve creating truncated Taylor expansions around certain parameters. By treating big-O expressions as terms in the polynomials, CIVL is able to automatically verify the accuracy assertions using algebraic manipulations similar to those described in Section 4. Fig. 7 gives the results of some scaling experiments for verifying correct programs. For the Laplace example, the number of rows is held constant and the column dimension is scaled. In addition, CIVL was correctly unable to verify modified assertions that claimed too high of an order of accuracy.

7 Conclusions and Future Work

We have presented a method for verifying that a numerical C program meets its specified order of accuracy. The method is based on symbolic execution, and uses the novel approach of taking automated Taylor expansions of symbolic representations of mathematical functions. We have given a rigorous definition of order of accuracy as it is applied to a program, and shown that the analysis resulting from the symbolic execution is valid. We have implemented this technique in a model checking tool and applied it to several numerical approximation codes.

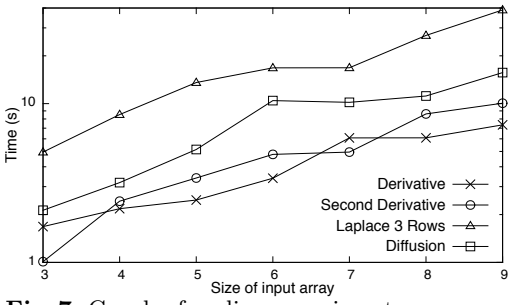

Fig. 7. Graph of scaling experiments run on a 2.6 GHz Intel Core i7 Mac Mini; log. time axis

An important aspect to this work is keeping the annotation effort reasonable. By focusing on automating the bulk of the theorem proving effort, we are able to only require a few extra lines of code from the programmer. These lines reflect information about the underlying mathematics that should be readily available to anybody implementing a numerical method.

Future work will focus on improving heuristics in order to verify more complex codes. Additionally, we may experiment with modifying CIVL to use different theorem provers. Currently CIVL uses CVC3 [4]. Preliminary experimentation with hand coded theorem prover queries showed that CVC3 performed comparably to the newer CVC4 [3] or other provers such as Z3 [16] for these types of problems, but further exploration with more complex examples is warranted.

References

1. American Institute of Aeronautics and Astronautics: Editorial policy statement on numerical and experimental accuracy. Journal of Guidance, Control, and Dynamics 31(1), 9–9 (2014), http://arc.aiaa.org/doi/abs/10.2514/1.G000410
2. Ayad, A., Marché, C.: Multi-prover verification of floating-point programs. In: Giesl, J., Hähnle, R. (eds.) Automated Reasoning, Lecture Notes in Computer Science, vol. 6173, pp. 127–141. Springer Berlin / Heidelberg (2010), http://dx.doi.org/10.1007/978-3-642-14203-1_11
3. Barrett, C., Conway, C., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) Computer Aided Verification, Lecture Notes in Computer Science, vol. 6806, pp. 171–177. Springer Berlin Heidelberg (2011), http://dx.doi.org/10.1007/978-3-642-22110-1_14
4. Barrett, C., Tinelli, C.: CVC3. In: Damm, W., Hermanns, H. (eds.) Proceedings of the 19^{th} International Conference on Computer Aided Verification (CAV '07). Lecture Notes in Computer Science, vol. 4590, pp. 298–302. Springer Berlin Heidelberg (Jul 2007), http://dx.doi.org/10.1007/978-3-540-73368-3_34
5. Boldo, S., Clément, F., Filliâtre, J.C., Mayero, M., Melquiond, G., Weis, P.: Wave equation numerical resolution: A comprehensive mechanized proof of a C program. Journal of Automated Reasoning 50(4), 423–456 (2013), http://dx.doi.org/10.1007/s10817-012-9255-4
6. CIVL: The Concurrency Intermediate Verification Language. http://vsl.cis.udel.edu/civl (accessed Feb 7, 2014)
7. Collingbourne, P., Cadar, C., Kelly, P.H.: Symbolic crosschecking of floating-point and SIMD code. In: Proceedings of the Sixth Conference on Computer Systems. pp. 315–328. EuroSys '11, ACM, New York, NY, USA (2011), http://doi.acm.org/10.1145/1966445.1966475
8. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: The ASTRÉE Analyser. In: Sagiv, M. (ed.) Proc. European Symposium on Programming (ESOP'05). Lecture Notes in Computer Science, vol. 3444, pp. 21–30. Springer, Edinburgh (April 2–10 2005), http://dx.doi.org/10.1007/978-3-540-31987-0_3
9. Cristian Cadar, Daniel Dunbar, D.E.: Klee: Unassisted and automatic generation of high-coverage tests for complex system programs. In: Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (2008), http://dl.acm.org/citation.cfm?id=1855741.1855756
10. Delmas, D., Goubault, E., Putot, S., Souyris, J., Tekkal, K., Védrine, F.: Towards an industrial use of FLUCTUAT on safety-critical avionics software. In: Alpuente, M., Cook, B., Joubert, C. (eds.) Formal Methods for Industrial Critical Systems, Lecture Notes in Computer Science, vol. 5825, pp. 53–69. Springer Berlin / Heidelberg (2009), http://dx.doi.org/10.1007/978-3-642-04570-7_6
11. Filliâtre, J.C., Marché, C.: The Why/Krakatoa/Caduceus platform for deductive program verification. In: Damm, W., Hermanns, H. (eds.) Proceedings of the 19^{th} International Conference on Computer Aided Verification (CAV '07). Lecture Notes in Computer Science, vol. 4590, pp. 173–177. Springer-Verlag, Berlin, Heidelberg (2007), ttp://dx.doi.org/10.1007/978-3-540-73368-3_21
12. Hairer, E., Lubich, C.: The life-span of backward error analysis for numerical integrators. Numerische Mathematik 76(4), 441–462 (June 1997), http://dx.doi.org/10.1007/s002110050271

13. Higham, N.J.: The accuracy of floating point summation. SIAM J. Sci. Comp. 14(4), 783 (1993), http://dx.doi.org/10.1137/0914050
14. Journal of Fluid Engineering: Editorial policy statement on the control of numerical accuracy. http://journaltool.asme.org/templates/JFENumAccuracy.pdf, accessed Feb. 9, 2014
15. Kincaid, D., Cheney, W.: Numerical Analysis. Brooks/Cole (1996)
16. Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C., Rehof, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science, vol. 4963, pp. 337–340. Springer Berlin Heidelberg (2008), http://dl.acm.org/citation.cfm?id=1792734.1792766
17. Reich, S.: Backward error analysis for numerical integrators. SIAM Journal on Numerical Analysis 36, 1549–1570 (July 1996), http://dx.doi.org/10.1137/S0036142997329797
18. Rivera-Gallego, W.: Stability analysis of numerical boundary conditions in domain decomposition algorithms. Applied Mathematics and Computation 137(2-3), 375 – 385 (2003), http://www.sciencedirect.com/science/article/B6TY8-45MDW9T-4/2/b7e93b6e8fad999a2cbd1747a2585f86
19. Roy, C.J.: Grid convergence error analysis for mixed-order numerical schemes. AIAA Journal 41(4), 595–604 (Apr 2003), http://dx.doi.org/10.2514/2.2013
20. Siegel, S.F., Rossi, L.F.: Analyzing BlobFlow: A case study using model checking to verify parallel scientific software. In: Lastovetsky, A., Kechadi, T., Dongarra, J. (eds.) Recent Advances in Parallel Virtual Machine and Message Passing Interface, 15th European PVM/MPI User's Group Meeting, Proceedings. LNCS, vol. 5205. Springer (2008), http://dx.doi.org/10.1007/978-3-540-87475-1_37
21. Siegel, S.F., Zirkel, T.K.: TASS: The Toolkit for Accurate Scientific Software. Mathematics in Computer Science 5(4), 395–426 (2011), http://dx.doi.org/10.1007/s11786-011-0100-7
22. Sousa, E., Sobey, I.: On the influence of numerical boundary conditions. Applied Numerical Mathematics 41(2), 325–344 (May 2002), http://dx.doi.org/10.1016/S0168-9274(01)00122-2
23. Strikwerda, J.C.: Finite Difference Schemes and Partial Differential Equations. Society for Industrial and Applied Mathematics, Philadelphia, PA, 2nd edn. (2004), www.cambridge.org/0898715679
24. Warming, R.F., Hyett, B.J.: The modified equation approach to the stability and accuracy analysis of finite-difference methods. Journal of Computational Physics 14(2), 159 – 179 (1974), http://www.sciencedirect.com/science/article/pii/0021999174900114
25. Wolfram Research, Inc.: Wolfram Mathematica: Technical Computing Software. http://www.wolfram.com/products/mathematica/index.html, accessed Feb. 9, 2014