

2.2 Specification and verification of the order of accuracy

Recent years have seen impressive progress in the ability to specify and verify the accuracy of floating-point computations (e.g., [7, 17, 43, 54]). A promising approach relevant to this proposal is taken by the Caduceus framework [7]. The framework provides an annotation system (inspired by the Java Modeling Language [38]) for expressing assertions and assumptions about the floating-point computations performed by C functions. For example, one may specify that the value returned by a function differs by no more than a specified quantity from the value that would have been returned if all computations in the program had been performed with infinite precision. The tool attempts to prove these assertions using (partially) automated reasoning techniques that have been augmented with a theory of IEEE floating-point arithmetic.

The issue of floating-point accuracy arises in scientific software, but an independent and equally important notion is that of *order of accuracy*. The order of accuracy of a numerical method provides information on how the accuracy of the method varies with parameters such as grid resolution, and is generally considered one of the most fundamental qualities of the method.¹ Computing the order of accuracy of an abstract algorithm is difficult enough, but showing that a computer program implementing that algorithm actually preserves the order of accuracy is notoriously difficult.

A numerical method to approximate a real-valued function of one variable $f(x)$, for example, might compute a function $g(x, h)$ which depends on an additional parameter h that is the distance between two successive grid points. The algorithm is said to be n^{th} order accurate if there exist $C > 0$ and $\delta > 0$ such that

$$|g(x, h) - f(x)| \leq Ch^n$$

whenever $0 < h \leq \delta$. This is commonly written $f(x) - g(x, h) = O(h^n)$.

As an example, suppose $f: \mathbf{R} \rightarrow \mathbf{R}$ is at least 3-times differentiable and that there exists $M > 0$ such that $|f'''(x)| < M$ for all x . An approximation to the first derivative of f by *central differencing* is given by

$$g(x, h) \equiv \frac{f(x+h) - f(x-h)}{2h}$$

This can be shown to be second-order accurate using Taylor polynomials with Lagrangian remainder terms [88]: given $x \in \mathbf{R}$ and $h > 0$, there exist $\xi_1, \xi_2 \in [x-h, x+h]$ such that

$$f(x+h) = f(x) + f'(x)h + \frac{1}{2}f''(x)h^2 + \frac{1}{6}f'''(\xi_1)h^3 \quad (1)$$

$$f(x-h) = f(x) - f'(x)h + \frac{1}{2}f''(x)h^2 - \frac{1}{6}f'''(\xi_2)h^3 \quad (2)$$

from which one computes

$$\left| \frac{f(x+h) - f(x-h)}{2h} - f'(x) \right| = \frac{1}{12} |f'''(\xi_1) + f'''(\xi_2)| h^2 \leq \frac{1}{6} M h^2. \quad (3)$$

Working with a mathematician specializing in numerical solutions to differential equations (Louis Rossi), I plan to develop a method for specifying and automatically verifying the order of accuracy of numerical codes.² In this section I describe our preliminary thoughts on this problem and sketch our ideas for a solution.

¹For example, 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].

²We have obtained a small “seed” grant (\$35K) to begin exploring these ideas, but the full project will require at least four years to fully realize the plan described here.

```

/*@ C(3) real f(real x);
    @ real M;
    @ assume forall {int i|i>=0 && i<n} y[i] == f(i*h);
    @ assume forall {real x} fabs(\D[f,{x,3}](x))<=M;
    @ ensures forall {int i|i>0 && i<n-1}
    @   fabs(result[i]-\D[f,{x,1}](i*h)) <= (M/6)*(h^2); */
void differentiate(double h, int n, double[] y, double[] result) {
    int i;
    for (i = 1; i < n-1; i++) result[i] = (y[i+1]-y[i-1])/(2*h);
    result[0] = (y[1]-y[0])/h; result[n-1] = (y[n-1]-y[n-2])/h;
}

```

(a) Source code with annotation specifying second order accuracy in interior

$$\begin{aligned}
f((i+1)h) &= f(ih) + f'(ih)h + \frac{1}{2}f''(ih)h^2 + \frac{1}{6}f'''(\xi_1)h^3 \\
f((i-1)h) &= f(ih) - f'(ih)h + \frac{1}{2}f''(ih)h^2 - \frac{1}{6}f'''(\xi_2)h^3 \\
f((i+1)h) - f((i-1)h) &= 2f'(ih)h + \frac{1}{6}(f'''(\xi_1) + f'''(\xi_2))h^3 \\
|[f((i+1)h) - f((i-1)h)]/(2h) - f'(ih)| &= \frac{1}{12}|f'''(\xi_1) + f'''(\xi_2)|h^2 \leq \frac{1}{6}Mh^2
\end{aligned}$$

(b) Symbolic arithmetic required to verify post-condition

```

f, f1, f2, f3 : REAL -> REAL; h, a, xi1, xi2, M, r : REAL; i : INT;
ASSERT a = i*h;
ASSERT FORALL (x : REAL): f3(x) <= M AND f3(x) >= -M;
ASSERT f((i+1)*h) = f(a) + f1(a)*h + f2(a)*h^2/2 + f3(xi1)*h^3/6;
ASSERT f((i-1)*h) = f(a) - f1(a)*h + f2(a)*h^2/2 - f3(xi2)*h^3/6;
ASSERT r = (f((i+1)*h)-f((i-1)*h))/(2*h) - f1(a);
TRANSFORM r/(h*h);
QUERY r <= M*h^2/6 AND r >= -M*h^2/6;

```

(c) Expression of query in input language of CVC3

```

(0 + (1/12 * f3(xi2)) + (1/12 * f3(xi1)))
Valid.

```

(d) Result of executing CVC3 on (c)

Figure 2: Specification, implementation, and accuracy verification for a numerical differentiation function. To verify second order accuracy, all expressions involving f are replaced by Taylor expansions evaluated at ih truncated at degree 3.

Specification is provided by the user in the form of annotations to functions in a C program. These annotations describe assertions on the accuracy of the values computed by those functions. The TASS verifier will parse these annotations with the source code, perform a special kind of symbolic execution with state exploration, and respond either that the assertions hold for all inputs to the function, or that they may not hold (and possibly include information on a counterexample). The functions may involve parallelism (such as MPI), but to simplify the presentation we will consider only sequential programs here.

An example encoding the central differencing scheme is shown in Fig. 2(a). (The annotations are a variation on those used by Caduceus and the derivative notation is borrowed from Mathematica.) The first line declares a real-valued function f of one variable which is 3-times differentiable; f does not occur in the code but will be used in the specification. The second line declares the constant M , which is also only used in the specification. The first assumption declares that the array y contains the values of f evaluated at the grid points ih ($0 \leq i < n$). The second assumption states that $|f'''|$ is bounded by M everywhere. An assertion declares that on interior points, the computed result will be a second-order approximation to the first derivative of f , specifically naming the constant $M/6$.

The assertion can be automatically proved from the assumption using a kind of symbolic execution in which the values may include symbolic functions evaluated at symbolic values. For example, the value of program variable h might be the symbolic constant h , the value of $y[2]$ would be the symbolic expression $f(2h)$, and the value of $result[2]$ when the function returns will be the symbolic expression $(f(3h) - f(h))/(2h)$.

The TASS verifier will execute the function symbolically and then call an automated theorem prover to check the assertion. The prover must be given the appropriate Taylor expansions, i.e., the first two lines of Fig. 2(b). Given these, the symbolic computations required to prove the assertion are straightforward, involving only standard arithmetic operations. In this computation, h , ξ_1 , ξ_2 , and M are all symbolic constants—they do not need to be assigned any concrete value. Likewise, f , f' , f'' , and f''' are all abstract symbolic functions from \mathbf{R} to \mathbf{R} .³ In particular, the theorem prover does not need any notion of derivatives or limits, and no interpretation is required for the functions f , f' , etc. The encoding of the query in the input language of the automatic theorem prover CVC3 is shown in Fig. 2(c). The **ASSERT** statements correspond to our **assume** and **QUERY** to our **ensures**. (The **TRANSFORM** instruction simplifies and prints an expression and is for illustrative purposes only.) CVC3 responds immediately that the query is valid, i.e., that it has proved the query from the given assumptions.

The question is then how the TASS verifier will know which Taylor expansions to provide to the prover. In this case, each term of the form $f(\dots)$ is expanded around the point ih and truncated at the third-order term. The choice of ih is suggested by the occurrence of the expression $f'(ih)$ in the query and the choice of third order is forced by the fact that f is declared $C(3)$. However, we can imagine situations in which these choices are not obvious, in which case the verifier might have to iterate over a parameter space, looking for a set of choices that leads to a valid conclusion. In any case, a “bad” choice can never lead the theorem prover to report that the assertion holds when it does not, since all of the expansions are valid, they just may not be useful.

Note that if, instead of central differencing, the *forward differencing* scheme $(y[i+1] - y[i])/h$ had been used, the result would be $O(h)$ instead of $O(h^2)$, and a violation would be reported. An error would also be reported if the assertion had included the boundary points, as these are

³To simplify matters, we can assume that a specific concrete value of n is specified. Then the symbolic execution of the loop terminates in a finite number of steps, and the assertion can be checked for one value of i at a time. However, with more work it is possible to extend this approach to deal with arbitrary n .

only computed to first-order accuracy. These are simple examples but illustrate the kinds of subtle errors that can corrupt the accuracy of numerical codes.

The same approach will work in more complex and realistic situations. For example, there might be another array \mathbf{z} for which $\mathbf{y}[i] - \mathbf{z}[i]$ is $O(h^4)$ and the differentiation formula might be $(y[i+1] - z[i-1])/(2h)$. The same technique can conclude that the result is still $O(h^2)$.

More generally still, the approach is applicable to functions of several variables, and can therefore be applied to programs used to solve systems of partial differential equations. Suppose u is a function of several variables x_1, x_2, \dots , L is a linear differential operator, and the PDE has the form $L[u] = 0$. A program approximating a solution might store the values of u on the points of a discrete grid $\Gamma = \Gamma(\Delta x_1, \Delta x_2, \dots)$, where Δx_i is the distance between two grid points in the x_i -direction. The program computes some discrete approximation \hat{u} to a solution on Γ . The approximation satisfies an operator \hat{L} on discrete arrays that is determined by the program code, i.e., $\hat{L}[\hat{u}] = 0$. The standard notion of order of accuracy in this case involves applying the discrete operator to the exact solution and specifying that the result should be sufficiently small. For example, to say the algorithm is first-order accurate in x_1 and second-order accurate in x_2 would mean $\|\hat{L}[u|_\Gamma]\| = O(\Delta x_1) + O(\Delta x_2^2)$, where $\|\mathbf{v}\|$ returns the maximum absolute value of the elements in the discrete array \mathbf{v} .

Consider, for example, the 1-dimensional diffusion equation. In this case, u is a function of two variables, x (space) and t (time), and a solution must satisfy the operator

$$L[u] = \frac{\partial u}{\partial t} - \kappa \frac{\partial^2 u}{\partial x^2}. \quad (4)$$

Fig. 3 shows an excerpt of a program to solve this equation by stepping through time. The function `update` takes the solution at time n in array `v1` and computes the solution at time $n+1$, storing it in `v2`. The assumptions state that $L[u] = 0$ and give bounds on certain derivatives of u . The verifier is asked to show that the computed solution satisfies \hat{L} , where

$$(\hat{L}[v])[n, i] = \frac{v[n+1, i] - v[n, i]}{\Delta t} - \kappa \frac{v[n, i+1] - 2v[n, i] + v[n, i-1]}{\Delta x^2} \quad (5)$$

and to show that $\hat{L}[u] = O(\Delta t) + O(\Delta x^2)$. Without going into detail, this can be proved using partial Taylor expansions in both the x and t directions, all around the point $(\mathbf{n} * \mathbf{dt}, \mathbf{i} * \mathbf{dx})$, much like the single-variable example of Fig. 2(b).

Our plan is to first refine the specification language sketched here by working out specifications for progressively more complex examples, such as multi-dimensional diffusion equation solvers with various initial value and boundary conditions, solutions to Laplace's equation, and so on. Ideally, we would like to find ways for the tool to do more of the specification work: for example, by automatically discovering constants such as $M/6$ in Fig. 2(a), and the discrete linear operator \hat{L} of (5).

At the same time, we will work out by hand the queries that will need to be verified automatically, such as the one of Fig. 2(c), and apply various automated theorem provers to these. The results will guide our decisions on the best way to express the queries.

We will then extend the front-end of the TASS toolkit in order to parse the query specification language, add the functionality to generate Taylor expansions, and so on, in order to automatically verify the examples starting from C source code. After this, we will apply the tool to progressively larger "real-world" codes in order to improve and finally evaluate the method. If time permits, we will explore other extensions of these ideas, such as methods to deal with floating-point arithmetic and numerical stability.

```

int n; /* current time step */
/*@ C(4) real u(real t, real x);
    @ real M1, M2;
    @ assume forall {real t | t >= 0} forall {real x}
    @   \D[u,{t,1}](t,x) - kappa*\D[u,{x,2}](t,x) == 0;
    @ assume forall {real t} forall {real x}
    @   fabs(\D[u,{t,2}](t,x)) <= M1 && fabs(\D[u,{x,4}](t,x)) <= M2;
    @ ensures forall {int i | 1 <= i && i < nx-1}
    @   (v2[i]-v1[i])/dt - kappa*(v1[i+1]-2*v1[i]+v1[i-1])/dx^2 == 0;
    @ ensures forall {int i | 1 <= i && i < nx-1}
    @   fabs( (u((n+1)*dt,i*dx)-u(n*dt,i*dx))/dt
    @         - kappa*(u(n*dt,(i+1)*dx) - 2*u(n*dt,i*dx) + u(n*dt,(i-1)*dx))/dx^2
    @         ) <= (M1/2)*dt + (kappa*M2/12)*dx^2;
    */
void update(int nx, double *v1, double *v2, double dt, double dx, double kappa) {
    int i;
    for (i = 1; i < nx-1; i++)
        v2[i] = v1[i] + (kappa*dt)/(dx*dx)*(v1[i+1] - 2*v1[i] + v1[i-1]);
}

```

Figure 3: Excerpt of one-dimension diffusion equation solver, with accuracy specification: second order in space, first order in time.