

Implementing a Rigorous ODE Solver through Literate Programming

Nedialko S. Nedialkov

Abstract Interval numerical methods produce results that can have the power of a mathematical proof. Although there is a substantial amount of theoretical work on these methods, little has been done to ensure that an implementation of an interval method can be readily verified. However, when claiming rigorous numerical results, it is crucial to ensure that there are no errors in their computation. Furthermore, when such a method is used in a computer assisted proof, it would be desirable to have its implementation published in a form that is convenient for verification by human experts.

We have applied Literate Programming (LP) to produce `VNODE-LP`, a C++ solver for computing rigorous bounds on the solution of an initial-value problem (IVP) for an ordinary differential equation (ODE). We have found LP well suited for ensuring that an implementation of a numerical algorithm is a correct translation of its underlying theory into a programming language: we can split the theory into small pieces, translate each of them, and keep mathematical expressions and the corresponding code close together in a unified document. Then it can be reviewed and checked for correctness by human experts, similarly to how a scientific work is examined in a peer-review process.

1 Introduction

Interval numerical methods produce results that can have the power of a mathematical proof. Typically, such a method computes bounds that are guaranteed to contain the true solution of a problem, proves that a solution

N. S. Nedialkov

Department of Computing and Software, McMaster University, Hamilton, Ontario, Canada, L8S 4K1, e-mail: nedialk@mcmaster.ca

does not exist or it indicates that a solution cannot be found. For example, when computing an enclosure on the solution of an IVP in ODEs, an interval solver first proves that there exists a unique solution and then produces bounds that contain it [9]; when solving a nonlinear equation, an interval method can prove that a region does not contain a solution or computes bounds that contain a unique solution to the problem [29]. For an excellent, up-to-date survey of these methods, see [34].

To date, not much has been done to ensure that the implementation of such a method can be readily verified, and the bounds it computes are indeed rigorous. Showing that an implementation is correct is of paramount importance for these methods, as mathematical rigor cannot be claimed, if we miss to include even a single roundoff error in a computation. Furthermore, when interval software is used in a computer-assisted proof, it would be desirable to have the software published in a form that is convenient for inspection and verification by human experts.

The author released in 2001 `VNODE` [24, 27], Validated Numerical ODE, a C++ package for computing bounds on the solution of an IVP for an ODE. This package is carefully written and tested, and it had shown to be robust and reliable. While one can check the theory behind `VNODE` (e.g. in [24]), it would be difficult to show that its C++ translation does not contain errors. The same applies to the other packages for computing bounds in IVPs for ODEs: `ADIODES` [38], `COSY` [3], and `VSPODE` [19]. That is, it also would be difficult to establish the correspondence between underlying theory and source code in these packages. A notable exception is `AWA` [21], where there is a clear “match” between the theory and the program listing in [21].

The above solvers have been used to compute rigorous bounds on solutions in IVP ODEs. For example, `VNODE` had been employed in applications such as rigorous multibody simulations [2], reliable surface intersection [23, 31], robust evaluation of differential geometry properties of a Bezier surface patch [17], computing bounds on eigenvalues [4], parameter and state estimation [11, 33], rigorous shadowing [6, 7], and theoretical computer science [1].

The author had always been concerned about possible errors in the implementation of `VNODE`. Obviously, if an error is present, then the works that have employed `VNODE` may contain invalid results. Moreover, how can one establish that the computed bounds are rigorous, and further, how others can be convinced that the implementation and the results are correct? This came as a major concern of the author of [1]: how one can trust the numerical results of `VNODE`? He needed a rigorous proof that an algebraic expression involving the solution of a highly nonlinear scalar ODE is less than one; otherwise his theorem would not hold. The strongest assurance argument was of the sort “`VNODE` has been accurate and reliable”, but obviously this is not satisfactory. The value of this expression was approximately 0.999... in multiple precision in `MAPLE`, but it needed to be proved that it was always smaller than 1. With `VNODE` we showed that the exact value of this expres-

sion is always smaller than one, but still, we did not have an unquestionable proof.

This prompted the author to search for ways to show that not only the implementation is correct, but it can also be checked readily by others. Literate Programming (LP) [15] was found particularly suitable for this purpose. Using LP, we can produce a *verifiable* implementation in the sense that it can be reviewed and examined for correctness, similarly to how a scientific work is reviewed by human experts in a peer-review process. This is in contrast to mechanical software verification, when a proof tool is applied to verify code against given specifications.

We reimplemented `VNODE` entirely with LP (along with some algorithmic improvements), which resulted in the `VNODE-LP` solver [26]. This paper gives an overview of `VNODE-LP`, elaborates on LP, and illustrates the process of employing it for carrying out a verifiable implementation.

Section 2 discusses LP. Section 3 presents an overview of `VNODE-LP`. Examples from its implementation, illustrating our approach using LP, are given in Section 4. Section 5 elaborates on relevant work. Section 6 summarizes our experience.

2 Literate Programming and `VNODE-LP`

Literate programming was introduced as a programming methodology by D. Knuth [13, 14]. Its essence can be captured as in [15, pg. 99]: “...instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do”, and introducing concepts “...in an order that is best for human understanding, using a mixture of formal and informal methods that reinforce each other.”

When developing a literate program, we break down an algorithm into smaller, easy-to-understand parts, and explain, document, and implement each of them in an order that is more natural for human comprehension, versus order that is suitable for compilation. In a literate program, documentation and code are in one source. The program is an interconnected “web” of pieces of code, referred to as *sections* [13, 15] or *chunks* [10, 36], which can be presented in any sequence. They are assembled into a program ready for compilation in a *tangle* process, which extracts the source code from the LP source. The documentation is “weaved” in a *weave* process, which prepares it for typesetting [15, 16].

We developed `VNODE-LP` using the `CWEB` literate programming tool [16] and its `ctangle` and `cweave` utilities. `CWEB` enables the inclusion of documentation and C++ code in a `CWEB` source file, which is essentially a `LATEX` file with additional statements for dealing with source code.

From a CWEB file, `cweave` generates a \LaTeX file; `cweave` takes care of page layout, indentation, suitable fonts, pretty printing of C/C++ code, and generates extensive cross-index information. Originally, CWEB could deal with \TeX input only. The \LaTeX `cweb` [35] class allows using \LaTeX ; the `cweb-hy` class [36], an extension of `cweb`, allows structuring of a \LaTeX document in chapters, sections, subsections, etc., and also provides automatic generation of hyperlinks, which are convenient for navigation through the code in the resulting, e.g., PDF file.

The `ctangle` utility extracts the source code and writes C/C++ files. It also includes line information in the generated files so that handling errors when compiling and debugging can be done in terms of CWEB source files, and not the generated C/C++ files. That is, when syntax errors or warnings are encountered, a compiler gives line numbers in web files, and similarly, when runtime errors are detected, a debugger gives line numbers in web files.

Developing a literate program reduces to writing an article or a book: we present the program in an order that follows our thought process and strive to explain our ideas clearly in a document that should be of publishable quality. For each algorithm in [26], we present its theory first, and then translate parts of it, where the division is such that the code in each part is not difficult to inspect. During development, if errors in compilation or execution occur, we can review the manuscript and update accordingly the CWEB files, without looking into the generated program files (they are for compiler consumption). Similarly, when inspecting VNODE-LP, we can work only with the LP document [26].

This article and [26] are created with CWEB and the `cweb-hy` class. The latter is composed like a book: with a table of contents, list of figures, hierarchical structure of the presentation, index, and bibliography. This document contains everything related to VNODE-LP: user guide, theory, documentation, source code, example, test cases, makefiles, and gnuplot files used for generating the plots in [26].

All the theory of VNODE-LP is included in [26]. Our goal was to have a self-contained, detailed presentation, so a reviewer would need only [26] when evaluating VNODE-LP. Since all the pieces for verifying the theory and implementation are in [26], if their correctness is confirmed by human experts like in a peer-review process, we may trust, or at least have high-confidence, in the correctness of the implementation of VNODE-LP and accept the bounds it computes as rigorous. When claiming rigor, however, we presume that the operating system, compiler, and the packages VNODE-LP uses do not contain errors affecting its execution.

3 Overview of VNODE-LP

We state the IVP that is the subject of this work (§3.1) and discuss briefly the methods in VNODE-LP and the packages it uses (§3.2).

3.1 The IVP VNODE-LP Solves

We consider the IVP

$$y'(t) = f(t, y), \quad y(t_0) = y_0, \quad t \in \mathbb{R}, \quad y \in \mathbb{R}^n. \quad (1)$$

We denote the set of closed (finite, nonempty) intervals on \mathbb{R} by

$$\mathbb{IR} = \{ \mathbf{a} = [\underline{\mathbf{a}}, \overline{\mathbf{a}}] \mid \underline{\mathbf{a}} \leq x \leq \overline{\mathbf{a}}, \underline{\mathbf{a}}, \overline{\mathbf{a}} \in \mathbb{R} \}.$$

An interval vector is a vector with interval components. We denote the set of n -dimensional interval vectors by \mathbb{IR}^n .

Given a point $t_{\text{end}} \neq t_0$ ($t_{\text{end}} \in \mathbb{R}$) and $\mathbf{y}_0 \in \mathbb{IR}^n$, VNODE-LP tries to compute a $\mathbf{y}_{\text{end}} \in \mathbb{IR}^n$ at t_{end} that contains the solution to (1) at t_{end} for all $y_0 \in \mathbf{y}_0$. If VNODE-LP cannot reach t_{end} , for example the bounds on the solution become too wide, bounds at some t^* between t_0 and t_{end} are returned.

Denote by $y(t_j; t_0, y_0)$ the solution to (1) with an initial condition y_0 at t_0 , and denote by \mathbf{y}_j an enclosure of the solution at t_j . That is,

$$y(t_j; t_0, y_0) \in \mathbf{y}_j \quad \text{for all } y_0 \in \mathbf{y}_0.$$

This solver proceeds in a one-step manner from t_0 to t_{end} , where it computes bounds at (adaptively) selected points $t_j \in (t_0, t_{\text{end}}]$. On a step from t_j to t_{j+1} , VNODE-LP computes first a priori bounds $\tilde{\mathbf{y}}_j$ such that

$$y(t; t_j, y_j) \in \tilde{\mathbf{y}}_j \quad \text{for all } t \in [t_j, t_{j+1}] \quad \text{and all } y_j \in \mathbf{y}_j.$$

Then it finds tight bounds \mathbf{y}_{j+1} at t_{j+1} such that

$$y(t_{j+1}; t_0, y_0) \in \mathbf{y}_{j+1} \quad \text{for all } y_0 \in \mathbf{y}_0;$$

see Figure 1.

To compute these bounds, we use interval arithmetic (IA), Taylor series expansion of the solution to (1) at each integration point, and various interval techniques. Since we compute high-order derivatives of the solution at each integration point, this package is applicable to ODE problems for which derivatives of the solution $y(t)$ exist to some order. As a consequence, the code list of f should not contain for example branches, abs, or min. For more details see [9, 24, 25, 26, 27, 28].

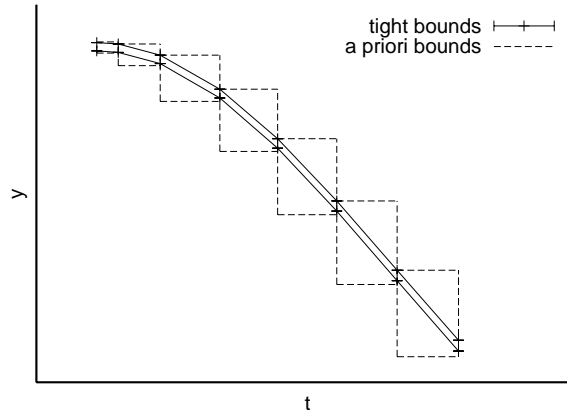


Fig. 1 A priori and tight bounds. For this visualization, the tight bounds are connected with lines, which do not necessarily enclose the true solution.

3.2 Methods and Packages

A user of VNODE-LP does not need to know how the underlying methods work. Minimal background in interval computations is sufficient, and the least knowledge requires that, if \mathbf{a} and $\mathbf{b} \in \mathbb{IR}$ and $\bullet \in \{+, -, \times, /\}$, then VNODE-LP builds on the IA operations defined as

$$\mathbf{a} \bullet \mathbf{b} = \{x \bullet y \mid x \in \mathbf{a}, y \in \mathbf{b}\},$$

where division is undefined if $0 \in \mathbf{b}$.

VNODE-LP is based on Taylor series and the Hermite-Obreschkoff [24] methods. It is a fixed-order, variable-stepsize solver. The stepsize is varied such that an estimate of the *local excess* per unit step is below a user-specified tolerance. Typical values for the order (for efficient integration) can be between 20 and 30 [25]; the default order is set to 20.

Generally, VNODE-LP is suitable for computing bounds on the solution of an IVP ODE with point initial conditions or interval initial conditions with a sufficiently small width. If the initial condition set is not small enough and/or long time integration is desired, the COSY package [3] of Berz and Makino can produce tighter bounds than VNODE-LP. Alternatively, one can subdivide the initial interval vector (box) \mathbf{y}_0 into smaller boxes, perform integrations with them as initial conditions, and build an enclosure of the solution at the desired t_{end} .

We tried to avoid advanced C++ constructs and tried to minimize the dependence of VNODE-LP on the IA package. VNODE-LP compiles with either of the IA packages PROFIL/BIAS [12] or FILIB++ [18]. Recently, the IA package GAOL [5] was used as the IA package in VNODE-LP [8].

The interface to an IA package is encapsulated in 26 small (most of them single line), inline wrapper functions that call functions from it. We aimed at keeping this interface as small as possible, such that another IA package can be incorporated easily. For this reason, we do not use, for example, the matrix and vector classes of PROFIL/BIAS, but implement our own matrix and vector operations through the C++ standard template library.

A major component of our solver is the tool for generating Taylor coefficients and Jacobians of Taylor coefficients through automatic differentiation (AD). This is done using the FADBAD++ [39] AD package. The necessary (non-rigorous) linear algebra is done through LAPACK and BLAS.

Finally, all the computations in VNODE-LP are in IEEE double precision arithmetic.

4 Examples from VNODE-LP

We illustrate typical steps when developing VNODE-LP: we give examples of two simple functions (§4.1) and an example of translating an expression that is part of a function (§4.2). We also present a simple program for integrating the Lorenz system (§4.3).

4.1 Computing h such that $[0, h] \mathbf{a} \subseteq \mathbf{b}$

The following problem is from the VNODE-LP implementation: given finite machine intervals \mathbf{a} and \mathbf{b} , where $0 \in \mathbf{b}$, find the largest $h \geq 0$ such that $[0, h] \mathbf{a} \subseteq \mathbf{b}$. Here, for $\mathbf{x}, \mathbf{y} \in \mathbb{IR}$, $\mathbf{x} \subseteq \mathbf{y}$ iff $\underline{\mathbf{x}} \geq \underline{\mathbf{y}}$ and $\overline{\mathbf{x}} \leq \overline{\mathbf{y}}$.

We derive a formula for h and then produce the C++ code. By $\nabla(x/y)$, we denote the rounded towards $-\infty$ result of x/y .

1. If $\underline{\mathbf{a}} = \overline{\mathbf{a}} = 0$, then $[0, h] \mathbf{a} = [0, 0] \subseteq \mathbf{b}$ for any h , and we set $h = \text{numeric_limits}\langle\text{double}\rangle::\text{max}()$, the largest double precision number.
Below we assume $\mathbf{a} \neq [0, 0]$.
2. If $\underline{\mathbf{a}} \geq 0$, then $\overline{\mathbf{a}} > 0$ and $[0, h] \mathbf{a} = [0, h\overline{\mathbf{a}}] \subseteq [\underline{\mathbf{b}}, \overline{\mathbf{b}}]$ when $h \leq \overline{\mathbf{b}}/\overline{\mathbf{a}}$.
We set $h = \nabla(\overline{\mathbf{b}}/\overline{\mathbf{a}})$.
3. If $\overline{\mathbf{a}} \leq 0$, then $\underline{\mathbf{a}} < 0$ and $[0, h] \mathbf{a} = [h\underline{\mathbf{a}}, 0] \subseteq [\underline{\mathbf{b}}, \overline{\mathbf{b}}]$ when $h \leq \underline{\mathbf{b}}/\underline{\mathbf{a}}$.
We set $h = \nabla(\underline{\mathbf{b}}/\underline{\mathbf{a}})$.
4. If $0 \in \mathbf{a}$, then $[0, h] \mathbf{a} = [h\underline{\mathbf{a}}, h\overline{\mathbf{a}}] \subseteq [\underline{\mathbf{b}}, \overline{\mathbf{b}}]$ when $h = \min\{\underline{\mathbf{b}}/\underline{\mathbf{a}}, \overline{\mathbf{b}}/\overline{\mathbf{a}}\}$.
We set $h = \min\{\nabla(\underline{\mathbf{b}}/\underline{\mathbf{a}}), \nabla(\overline{\mathbf{b}}/\overline{\mathbf{a}})\}$.

We translate the above cases into:

```

1 < h such that  $[0, h]\mathbf{a} \subseteq \mathbf{b}$  (intervals) 1 > ≡
#include <climits>
using namespace std;
using namespace v_bias;

inline double compH(const interval &a, const interval &b)
{
    /* inf(a) returns  $\underline{a}$ ; sup(a) returns  $\bar{a}$  */
    if (inf(a) ≡ 0 ∧ sup(a) ≡ 0)
        return numeric_limits<double>::max();
    round_down(); /* set rounding mode to  $-\infty$  */
    if (inf(a) ≥ 0) return sup(b)/sup(a);
    if (sup(a) ≤ 0) return inf(b)/inf(a);
    return std::min(inf(b)/inf(a), sup(b)/sup(a));
}

```

This code is used in chunk 2

This is a *chunk* of code. It is identified by its name, here “ h such that $[0, h]\mathbf{a} \subseteq \mathbf{b}$ (intervals)”. The `ctangle` program, when extracting the code, orders the chunks based on their names. Each chunk is numbered by `cweave`, and these numbers are convenient for referencing them in the LP document.

A nice feature of `cweave` is that it typesets the code in a very readable form, while the code that is typed in a web file does not even need to be indented. Mathematics can be included in a L^AT_EX form as a comment, and `if` conditions are typeset more like math, rather than C++.

Now, given interval vectors \mathbf{a} and \mathbf{b} , with each component of \mathbf{b} containing 0, we wish to find the largest representable $h \geq 0$ such that $[0, h]\mathbf{a} \subseteq \mathbf{b}$. We write

```

2 < h such that  $[0, h]\mathbf{a} \subseteq \mathbf{b}$  (interval vectors) 2 > ≡
  < h such that  $[0, h]\mathbf{a} \subseteq \mathbf{b}$  (intervals) 1 >
double compH(const iVector&a, const iVector&b)
{
    double hmin = compH(a[0], b[0]);
    for (unsigned int i = 1; i < sizeV(a); i++) {
        double h = compH(a[i], b[i]);
        if (h < hmin) hmin = h;
    }
    return hmin;
}

```

This chunk includes the previous one and calls `compH` on each two components to find h .

4.2 Translating expressions

A method in VNODE-LP can be broken down into a sequence of formulas, and each formula must be implemented carefully, to ensure that all truncation and roundoff errors in a computation are included in the resulting bounds. To achieve this, each formula (or a few formulas) is translated into a chunk. The resulting chunks are put together by `ctangle`, thus obtaining an implementation of the complete method.

Here is another simple example from VNODE-LP's implementation. When propagating bounds on the global excess [24, 26], we need to evaluate

$$\mathbf{r}_{j+1} = (A_{j+1}^{-1} \mathbf{A}_{j+1}) \mathbf{r}_j + A_{j+1}^{-1} \mathbf{v}_{j+1},$$

where \mathbf{r}_j and \mathbf{v}_{j+1} are interval vectors, \mathbf{A}_{j+1} is an interval matrix, and A_{j+1} is a nonsingular point matrix. The chunk implementing this formula (we omit the declarations of objects and variables) is:

```

3  ⟨  $\mathbf{r}_{j+1} = (A_{j+1}^{-1} \mathbf{A}_{j+1}) \mathbf{r}_j + A_{j+1}^{-1} \mathbf{v}_{j+1}$  3 ⟩ ≡    /*
    trial_solution→A = Aj+1
      A ⊇ Aj+1
      z ⊇ vj+1
    solution→r ⊇ rj
    -----
      Ainv ⊃ Aj+1-1 if ok
      temp ⊇ Aj+1-1 vj+1
      M ⊇ Aj+1-1 Aj+1
      trial_solution→r ⊇ (Aj+1-1 Aj+1) rj
      trial_solution→r ⊇ rj+1 = (Aj+1-1 Aj+1) rj + Aj+1-1 vj+1
    */
    bool ok = matrix_inverse_encloseMatrixInverse(Ainv, trial_solution→A);
    if (ok) {
      multMiVi(temp, Ainv, z);
      multMiMi(M, Ainv, A);
      multMiVi(trial_solution→r, M, solution→r);
      addViVi(trial_solution→r, temp);
    }

```

In the comment above the horizontal line, we state informally where the vectors and matrices are stored before executing the code: `trial_solution→A` stores¹ A_{j+1} , `z` contains \mathbf{v}_{j+1} , `A` contains \mathbf{A}_{j+1} , and `solution→r` contains \mathbf{r}_j .

¹ For readers not familiar with C++, the operator `→` selects a field in a structure when a pointer is being used.

After the horizontal line, we state each step of the computation, so we can easily check the code that follows against it.

The `encloseMatrixInverse` function computes an interval matrix, output argument `Ainv`, which encloses A_{j+1}^{-1} . If this function computes an enclosure (A_{j+1} is nonsingular and not badly conditioned), then we evaluate the expression. Here M_i and V_i stand for interval matrix and interval vector, respectively. Obviously, it is not difficult to establish the validity of this code.

Remark 1. One may find the explanations here and in [26] containing too much detail. However, our goal is to provide as much detail as possible such that one can readily verify all the steps when going from theory to code.

Remark 2. For better understanding, the author has found it helpful to write in comments what is computed, in addition to the exposition before a chunk. We could comment separate lines of code, but it becomes less readable.

4.3 Integrating the Lorenz System

We give an example illustrating basic integration with VNODE-LP and showing in more detail how LP works. More examples are given in [26].

With VNODE-LP, the user has to specify first the right side of an ODE problem and then provide a main program. An ODE must be given by a template function for evaluating $y' = f(t, y)$ of the form

```
4 <template ODE function 4> ≡
   template<typename var_type>
   void ODENAME(int n, var_type *yp, const var_type *y, var_type t,
                 void *param)
   {
       /* body */
   }
```

Here n is the size of the problem, t is the time variable, y is a pointer to input variables, yp is a pointer to output variables, and $param$ is a pointer to additional parameters that can be passed to this function.

Consider the Lorenz system

$$\begin{aligned} y_1' &= \sigma(y_2 - y_1) \\ y_2' &= y_1(\rho - y_3) - y_2 \\ y_3' &= y_1y_2 - \beta y_3, \end{aligned}$$

where σ , ρ , and β are constants. This system is encoded in the *Lorenz* function below. The constants have values $\sigma = 10$, $\beta = 8/3$, and $\rho = 28$. We initialize *beta* with the interval containing $8/3$: `interval(8.0)` creates an interval with endpoints 8.0, and `interval(8.0)/3.0` is the interval containing $8/3$.

```

5 <Lorenz 5> ≡
  template(typename var_type)
  void Lorenz(int n, var_type *yp, const var_type *y, var_type t,
             void *param)
  {
    interval sigma(10.0), rho(28.0);
    interval beta = interval(8.0)/3.0;

    yp[0] = sigma * (y[1] - y[0]);
    yp[1] = y[0] * (rho - y[2]) - y[1];
    yp[2] = y[0] * y[1] - beta * y[2];
  }

```

This code is used in chunk 6

We give a simple main program and develop its parts.

```

6 <simple main program 6> ≡
  <Lorenz 5>
  int main()
  {
    <set initial condition and endpoint 7>
    <create AD object 8>
    <create a solver 9>
    <integrate (basic) 10>
    <check if success 11>
    <output results 12>
    return 0;
  }

```

This code is used in chunk 13

The initial condition and endpoint are represented as intervals in VNODE-LP. In this example, they are all point values stored as intervals. The components of *iVector* (interval vector) are accessed like a C/C++ array is accessed.

```

7 <set initial condition and endpoint 7> ≡
  const int n = 3;
  interval t = 0.0, tend = 20.0;

  iVector y(n);
  y[0] = 15.0;
  y[1] = 15.0;
  y[2] = 36.0;

```

This code is used in chunk 6

Then we create an AD object of class `FADBAD_AD`. It is instantiated with data types for computing Taylor coefficients (TCs) of the ODE solution and TCs of the solution to its variational equation, respectively [24]. To compute these coefficients, we employ the `FADBAD++` package [39]. The first parameter in the constructor of `FADBAD_AD` is the size of the problem. The second and third parameters are the name of the template function.

```
8 <create AD object 8> ≡
   AD * ad = new FADBAD_AD(n, Lorenz, Lorenz);
```

This code is used in chunk 6

Now, we create a solver:

```
9 <create a solver 9> ≡
   VNODE * Solver = new VNODE(ad);
```

This code is used in chunk 6

The integration is carried out by the `integrate` function. It attempts to compute bounds on the solution at `tend`. When `integrate` returns, either $t = tend$ or $t \neq tend$. In both cases, `y` contains the ODE solution at `t`.

```
10 <integrate (basic) 10> ≡
   Solver->integrate(t, y, tend);
```

This code is used in chunk 6

We check if an integration is successful by calling `Solver->successful()`:

```
11 <check if success 11> ≡
   if (!Solver->successful())
       cout << "VNODE-LP could not reach t=" << tend << endl;
```

This code is used in chunk 6

Finally, we report the computed enclosure of the solution at `t` by

```
12 <output results 12> ≡
   cout << "Solution enclosure at t=" << t << endl;
   printVector(y);
```

This code is used in chunk 6

The `VNODE-LP` package is in the namespace `vnode lp`. The interface to `VNODE-LP` is stored in the file `vnode.h`, which must be included in any file using `VNODE-LP`. We store our program in the file `basic.cc`.

```
13 <basic.cc 13> ≡
#include <ostream>
#include "vnode.h"
using namespace std;
using namespace vnode lp;
<simple main program 6>
```

When compiled and executed, the output of this program is

```
Solution enclosure at t = [20,20]
14.30[38383897955050,44533114627495]
9.5[786468306960781,800894448257097]
39.038[2433999742133,4067233742798]
```

It is interpreted as

$$y(20) \in \begin{pmatrix} [14.3038159449608937, 14.3044694855332662] \\ [9.5785941360078012, 9.5801274302834650] \\ [39.0382373597549516, 39.0384111043348412] \end{pmatrix}. \quad (2)$$

These results are produced using PROFIL/BIAS, and the output format is due to the output format of this package. For comparison, if we integrate the Lorenz system with MAPLE using `dsolve` with options `method=taylorseries` and `abserr=Float(1,-18)`, and with `Digits := 20`, we obtain

$$y(20) \approx \begin{pmatrix} 14.304146251277895001 \\ 9.5793690774871976695 \\ 39.038325167739731729 \end{pmatrix},$$

which is contained in the bounds (2).

Needless to say, one can write application programs without LP. In Figure 2, we show the code of the above example written in “plain” C++.

Remark 3. Here, the chunks are presented in a consecutive order, but as mentioned earlier, they can be in any order.

5 Relevant Work

A comprehensive collection of resources on LP, including extensive bibliography is [20]; annotated bibliography of LP until 1991 is [37]. To the best of the author’s knowledge, VNODE-LP is the first LP implementation of an interval package, and the only other implementation of non-trivial numerical software appears to be [32].

In [22], LP is used to facilitate the verification of a network security device. The authors propose in [22] that LP techniques are used to “document the entire assurance argument.” According to their experience, rigorous arguments, including machine-generated proofs of theory and implementation, “did not significantly improve the certifier’s confidence” in their validity. One of the main reasons is that specifications and proofs were documented in a

```

#include <ostream>
#include "vnode.h"
using namespace std;
using namespace vnodelp;

template<typename var_type>
void Lorenz(int n, var_type*yp, const var_type*y,
            var_type t, void*param) {
    interval sigma(10.0), rho(28.0);
    interval beta = interval(8.0)/3.0;

    yp[0] = sigma*(y[1]-y[0]);
    yp[1] = y[0]*(rho-y[2])-y[1];
    yp[2] = y[0]*y[1]-beta*y[2];
}

int main() {
    const int n = 3;
    interval t = 0.0, tend = 20.0;
    iVector y(n);
    y[0] = 15.0;
    y[1] = 15.0;
    y[2] = 36.0;
    AD *ad= new FADBAD_AD(n,Lorenz,Lorenz);
    VNODE *Solver= new VNODE(ad);
    Solver->integrate(t,y,tend);
    if(!Solver->successful())
        cout<<"VNODE-LP could not reach t = "<<tend<<endl;
    cout<<"Solution enclosure at t = "<<t<<endl;
    printVector(y);
    return 0;
}

```

Fig. 2 “Plain” C++ code for the Lorenz example

manner to facilitate acceptance by mechanical tools rather than humans. Essentially, the authors conclude that LP greatly facilitates the development of assurance arguments that would be more naturally understood by (human) certifiers than descriptions of machine-generated proofs.

A notable methodology for inspecting an implementation is the program function tables approach of D. Parnas [30]. Before considering LP, the author assessed this approach for inspecting VNODE. However, program function tables are suitable when the relation between input and output arguments is represented by a relatively simple function, which is hardly the case with VNODE.

6 Summary of Experience

Developing a non-trivial literate program can be time consuming, which manifests itself into a substantial “up-front” investment of time: we focus on writing a high-quality, well-structured, and easy-to-understand document. This requires paying attention to detail and ensuring that no errors are present. Since this process is inherently slow, one is “forced” to write code carefully, reducing the likelihood of errors.

Once the effort is put into writing a good LP document, then little time goes into debugging and testing—instead of trying to discover errors through them, we simply proofread the LP document. Moreover, theory and code can be cross checked against each other, and error in one may be revealed in the other. In addition, since documentation and code are in one source, they can be naturally kept in sync.

In the author’s opinion, if one shows that (a) the theory of a method is correct and (b) its implementation is a provably correct translation of the theory, then minimal testing is required. From the author’s experience, if he had implemented the original `VNODE` solver through LP, then less time would have been spent on checking the implementation, debugging, and testing. More importantly, the confidence in the implementation would have been much higher.

There are 14 tests in the distribution of `VNODE-LP`. Their main purpose is to ensure that the IA package and `VNODE-LP` are installed properly. Indeed, the few problems reported to the author about `VNODE-LP` not being able to execute a test successfully were all related to problems in the installation of the underlying IA package.

It does not appear appropriate to use LP at early stages of program development, when prototyping and experimenting with algorithms, design, and interfaces. When a design is settled, and no major changes are anticipated, then one can “cement” the implementation with LP. In our case, `VNODE` was in a stable state, and no experimenting was needed before investing into `VNODE-LP`.

The number of C/C++ lines (without comments) in `VNODE-LP` is 2,030. This is not a large package, but complex “per line of code.” The LP document [26] is 218 pages. For much larger programs, LP may not be an attractive option, especially when a software product must be delivered on time. In academia, researchers rarely go beyond prototype, research codes and releasing software packages, let alone devoting a substantial amount of time into producing a book-like manuscript (which may not count as a publication). At least for the above two reasons, LP is not ubiquitous, even though it has existed for more than 25 years.

Although LP may appear prohibitively time consuming, the author believes that the cumulative effort for producing and maintaining a complex program is smaller using LP compared to “traditional” program development. The author also believes that establishing program correctness by reviewing

a literate program may be more effective than employing a software verification tool. It requires not only that a proof mechanism is constructed, but also that the corresponding theory, documentation, and software are checked—we may as well inspect the original program.

Finally, an interval method, which is theoretically guaranteed to produce rigorous results, should be implemented and documented with the same rigor as its theory is derived. Guaranteeing rigor due to theory and not of its implementation diminishes the purpose of such a method.

Acknowledgments. The author thanks George Corliss, Math Pharr, John Pryce, Spencer Smith, and Bingzhou Zheng for careful reading of this article and providing valuable comments.

This work was supported in part by the Natural Sciences and Engineering Research Council of Canada.

References

1. Achlioptas, D.: Setting 2 variables at a time yields a new lower bound for random 3-SAT. Tech. Rep. MSR-TR-99-96, Microsoft Research, Microsoft Corp., One Microsoft Way, Redmond, WA 98052 (1999)
2. Auer, E., Kecske m thy, A., T ndl, M., Traczinski, H.: Interval algorithms in modelling of multibody systems. In: Numerical Software with Result Verification, *LNCS*, vol. 2991, pp. 132–159. Springer-Verlag (2004)
3. Berz, M.: COSY INFINITY version 8 reference manual. Technical Report MSUCL–1088, National Superconducting Cyclotron Lab., Michigan State University, East Lansing, Mich. (1997)
4. Brown, B.M., Langer, M., Marletta, M., Tretter, C., Wagenhofer, M.: Eigenvalue bounds for the singular Sturm-Liouville problem with a complex potential. *J. Phys. A: Math. Gen.* **36**(13), 3773–3787 (2003)
5. Goualard, F.: Gaol: Not just another interval library (version 2.0.2). <http://sourceforge.net/projects/gaol/> (2006)
6. Hayes, W.: Rigorous shadowing of numerical solutions of ordinary differential equations by containment. Ph.D. thesis, Department of Computer Science, University of Toronto, Toronto, Canada (2001)
7. Hayes, W., Jackson, K.R.: Rigorous shadowing of numerical solutions of ordinary differential equations by containment. *SIAM J. Numer. Anal.* **42**(5), 1948–1973 (2003)
8. Ishii, D.: Simulation and verification of hybrid systems based on interval analysis and constraint programming. Ph.D. thesis, Graduate School of Science and Engineering, Waseda University, Japan (2010)
9. Jackson, K.R., Nedialkov, N.S.: Some recent advances in validated methods for IVPs for ODEs. *Appl. Numer. Math.* **42**, 269–284 (2002)
10. Johnson, A., Johnson, B.: Literate programming using `noweb`. *Linux J.*, Article No 1 (1997)
11. Kieffer, M., Walter, E.: Nonlinear parameter and state estimation for cooperative systems in a bounded-error context. In: Numerical Software with Result Verification, *LNCS*, vol. 2991, pp. 107–123. Springer-Verlag (2004)
12. Kn ppel, O.: PROFIL/BIAS – a fast interval library. *Computing* **53**(3–4), 277–287 (1994)
13. Knuth, D.E.: The `WEB` system of structured documentation. Stanford Computer Science Report CS980, Stanford University, Stanford, CA (1983)

14. Knuth, D.E.: Literate programming. Technical report STAN-CS-83-981, Stanford University, Department of Computer Science (1983)
15. Knuth, D.E.: Literate Programming. Center for the Study of Language and Information, Stanford, CA, USA (1992)
16. Knuth, D.E., Levy, S.: The CWEB System of Structured Documentation. Addison-Wesley, Reading, Massachusetts (1993)
17. Lee, C.K.: Robust evaluation of differential geometry properties using interval arithmetic techniques. Master's thesis, Massachusetts Institute of Technology, Department of Ocean Engineering (2005)
18. Lerch, M., Tischler, G., Gudenberg, J.W.V., Hofschuster, W., Krämer, W.: FILIB++, a fast interval library supporting containment computations. *ACM Trans. Math. Softw.* **32**(2), 299–324 (2006)
19. Lin, Y., Stadtherr, M.A.: Validated solution of initial value problems for ODEs with interval parameters. In: R.L. Muhanna, R.L. Mullen (eds.) *Proceedings of 2nd NSF Workshop on Reliable Engineering Computing*. Savannah, GA (2006)
20. Literate programming web site. <http://www.literateprogramming.com>
21. Lohner, R.J.: Einschließung der Lösung gewöhnlicher Anfangs- und Randwertaufgaben und Anwendungen. Ph.D. thesis, Universität Karlsruhe (1988)
22. Moore, A.P., Payne, C.N., Jr.: Increasing assurance with literate programming techniques. In: *Proceedings of 11th Annual Conference on Computer Assurance*. COMPASS '96, pp. 187–198 (1996)
23. Mukundan, H., Ko, K.H., Maekawa, T., Sakkalis, T., Patrikalakis, N.M.: Tracing surface intersections with a validated ODE system solver. In: G. Elber, G. Taubin (eds.) *Proceedings of the Ninth EG/ACM Symposium on Solid Modeling and Applications*. Eurographics Press, June 2004 (2004)
24. Nedialkov, N.S.: Computing rigorous bounds on the solution of an initial value problem for an ordinary differential equation. Ph.D. thesis, Department of Computer Science, University of Toronto, Toronto, Canada, M5S 3G4 (1999)
25. Nedialkov, N.S.: Interval tools for ODEs and DAEs. In: *SCAN Conference Proceedings*, <http://www.computer.org/portal/web/csdl/doi/10.1109/SCAN.2006.28> (2006)
26. Nedialkov, N.S.: VNODE-LP — a validated solver for initial value problems in ordinary differential equations. Tech. Rep. CAS-06-06-NN, Department of Computing and Software, McMaster University, Hamilton, Canada, L8S 4K1 (2006). VNODE-LP is available at <http://www.cas.mcmaster.ca/~nedialk/vnodelp>
27. Nedialkov, N.S., Jackson, K.R.: The design and implementation of a validated object-oriented solver for IVPs for ODEs. Tech. Rep. 6, Software Quality Research Laboratory, Department of Computing and Software, McMaster University, Hamilton, Canada, L8S 4K1 (2002)
28. Nedialkov, N.S., Jackson, K.R., Corliss, G.F.: Validated solutions of initial value problems for ordinary differential equations. *Appl. Math. Comp.* **105**(1), 21–68 (1999)
29. Neumaier, A.: *Interval Methods for Systems of Equations*. Cambridge University Press, Cambridge (1990)
30. Parnas, D.L.: Inspection of safety-critical software using program-function tables. In: *Software fundamentals: collected papers by David L. Parnas*, pp. 371–382. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2001)
31. Patrikalakis, N.M., Maekawa, T., Ko, K.H., Mukundan, H.: Surface to surface intersection. In: L. Piegl (ed.) *International CAD Conference and Exhibition, CAD'04*. Thailand (2004)
32. Pharr, M., Humphreys, G.: *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2004)
33. Ramdani, N., Meslem, N., Raïssi, T., Candau, Y.: Set-membership identification of continuous-time systems. In: B. Ninness, H. Hjalmarsson (eds.) *14th IFAC Symposium on System Identification, 2006*, vol. 14. IFAC (2007)
34. Rump, S.: Verification methods: Rigorous results using floating-point arithmetic. In: *Acta Numerica*, pp. 287–449. Cambridge University Press (2010)

35. Schrod, J.: The `cweb` class. CTAN, the Comprehensive T_EX Archive Network (1995)
36. Schrod, J.: Typesetting CWEAVE output. CTAN, the Comprehensive T_EX Archive Network (1995)
37. Smith, L.M.C., Samadzadeh, M.H.: An annotated bibliography of literate programming. ACM SIGPLAN Notices **26**(1), 14–20 (1991)
38. Stauning, O.: Automatic validation of numerical solutions. Ph.D. thesis, Technical University of Denmark, DK-2800, Lyngby, Denmark (1997)
39. Stauning, O., Bendtsen, C.: FADBAD++ web page (2003).
<http://www.imm.dtu.dk/fadb主ad.html>