The design and implementation of an object-oriented validated ODE solver *

Nedialko S. Nedialkov†
Department of Computing and Software, McMaster University, Hamilton, Ontario, L8S 4L7, Canada, (`nedialk@mcmaster.ca`).

Kenneth R. Jackson
Department of Computer Science, University of Toronto, Toronto, Ontario, M5S 3G4, Canada, (`krj@cs.toronto.edu`).

Abstract.
Validated methods for initial value problems (IVPs) for ordinary differential equations (ODEs) produce bounds that are guaranteed to contain the true solution of a problem. These methods use interval techniques to enclose rounding errors, truncation errors, and uncertainties in the model.

We describe the design and implementation of VNODE, Validated Numerical ODE, which is an object-oriented solver for computing rigorous bounds on the solution of an IVP for an ODE. We have designed VNODE as a set of C++ classes, which encapsulate data structures and algorithms employed in validated methods for IVPs for ODEs.

A new method can be added to VNODE by writing it as a subclass of the relevant class that is provided as part of VNODE; a new solver can be constructed by combining objects of appropriate classes. Furthermore, an object in a solver can be replaced by another one performing the same task. This enables the user to isolate and compare methods implementing the same part of a solver.

We illustrate how to integrate with VNODE, how to construct new solvers, and how to extend this package with new methods.

Keywords: ordinary differential equations, initial value problems, one-step methods, interval methods, Taylor series methods, Hermite-Obreschkoff methods, object-oriented design.

AMS subject classification: G.1.7, G.4, D.1.5, D.3.2.

1. Introduction

Standard numerical methods for initial value problems (IVPs) for ordinary differential equations (ODEs) attempt to compute an approximate solution that satisfies a user-specified tolerance. These methods are usually robust and reliable for most applications, but it is possible to

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

† This work started in 1997 when N. S. Nedialkov was a PhD student in the Department of Computer Science at the University of Toronto, Canada.

find examples for which they return very inaccurate results. Validated (also called interval) methods for IVPs for ODEs verify that a unique solution to a problem exists and produce bounds that are guaranteed to contain this solution.

There are situations when guaranteed bounds on the mathematically correct result are desired or needed. For example, such bounds can be used to prove a theorem [1, 29, 31]. Also, some calculations may be critical to the safety or reliability of a system [9]. Therefore, it may be necessary to ensure that the true result is within computed bounds. Furthermore, methods that produce guaranteed bounds may be used to check a sample calculation to be performed by a standard floating-point scheme, to indicate to the user the accuracy and reliability of the floating-point scheme.

This paper presents the design and implementation of the VNODE, Validated Numerical ODE, package, which is an object-oriented, C++ package for computing rigorous bounds on the solution of the IVP

$$y'(t) = f(y) \tag{1}$$
$$y(t_0) \in [y_0], \tag{2}$$

where $t \in [t_0, t_m]$ for some[1] $t_m > t_0$. Here t_0 and $t_m \in \mathbb{R}$, $f \in C^{k-1}(\mathcal{D})$, $k \geq 2$ (k is the order of the truncation error in the methods that we present later), $\mathcal{D} \subseteq \mathbb{R}^n$ is open, $f : \mathcal{D} \to \mathbb{R}^n$, y is an n-dimensional vector[2], and $[y_0] \subseteq \mathcal{D}$. The condition (2) permits the initial value $y(t_0)$ to be in an interval, rather than specifying a particular value. We assume that the representation of f contains a finite number of constants, variables, elementary operations, and standard functions. Since we assume $f \in C^{k-1}(\mathcal{D})$, we exclude functions that contain, for example, branches, abs, or min.

The methods in VNODE deal with autonomous systems only. This is not a restriction of consequence, since a nonautonomous system of ODEs can be converted into an autonomous one. Moreover, these methods can be extended easily to nonautonomous systems.

We consider a grid $t_0 < t_1 < \cdots < t_m$, which is not necessarily equally spaced, and denote the solution of (1) with an initial condition $y(t_{j-1}) = y_{j-1}$ at t_{j-1} by $y(t; t_{j-1}, y_{j-1})$. For an interval, or an interval vector $[y_{j-1}]$, we denote by $y(t; t_{j-1}, [y_{j-1}])$ the set of solutions

$$\{\, y(t; t_{j-1}, y_{j-1}) \mid y_{j-1} \in [y_{j-1}] \,\}.$$

[1] For expositional convenience, we consider only $t_m > t_0$. However, the methods discussed in this paper can be easily adapted to handle the case $t_m < t_0$. In fact, VNODE allows $t_m < t_0$.

[2] Throughout this paper we regard vectors as column vectors.

The main purpose of VNODE is to compute interval vectors $[y_j]$, $j = 1, 2, \ldots, m$, that are guaranteed to contain the solution of (1–2) at $t_1, t_2, \ldots, t_m$. That is,

$$y(t_j; t_0, [y_0]) \subseteq [y_j], \quad \text{for } j = 1, 2, \ldots, m. \tag{3}$$

In addition, this package is intended to facilitate the numerical study of methods used in validated ODE solving: the user can build solvers, replace part(s) of a solver, and add new methods to VNODE.

To compute the bounds in (3), we use interval arithmetic [14] and various interval techniques to enclose rounding and truncation errors and the excess in propagating the solution from step to step. Currently, the methods in VNODE are based on Taylor series [12, 16] and our recent extension of Hermite-Obreschkoff schemes to interval methods [16, 17]. These methods employ Taylor expansions with respect to time only; VNODE does not include Berz and Makino's scheme [5], which uses Taylor series expansions in both time and the initial conditions.

One reason for the popularity of the Taylor series approach [5, 7, 12, 14, 16, 27, 30] is the simple form of the error term. In addition, Taylor series coefficients can be readily generated by automatic differentiation, and both the order of the method and its stepsize can be easily changed from step to step. The Hermite-Obreschkoff scheme shares these advantages. Moreover, for the same stepsize and order of the truncation error, our interval Hermite-Obreschkoff (IHO) method is more efficient and produces tighter bounds than an interval Taylor series (ITS) method [16].

Since the current interval methods for IVPs for ODEs use high-order derivatives, interval arithmetic, and matrix operations, an interval method can be substantially more expensive than a traditional (point) method. Consequently, a validated method may not be appropriate for large, computationally intensive problems or ones that are time critical. However, if interval methods are criticized for being more expensive than traditional methods, we should emphasize that the former compute rigorous enclosures for the mathematically correct result, not just approximations, as the latter do. Moreover, with the speed of processors increasing and the cost of memory decreasing, many IVPs for ODEs are not expensive to solve by modern standards. For such problems, we believe the time is now ripe to shift the responsibility for determining the reliability of numerical results from the user to the computer.

If a problem is not computationally expensive to solve by standard methods, but the reliability of the results is important, it may be more cost-effective to let the computer spend more time computing a validated solution with a guaranteed error bound, rather than having the user attempt to verify the reliability of a numerical solution computed by approximate methods. The reliability issue does not arise for validated methods, in the sense that, if implemented correctly, software based on a validated method always produces reliable results.

This paper is organized as follows. Section 2 discusses interval arithmetic operations and computing ranges of functions. Section 3 gives an overview of the methods employed in VNODE. In section 4, we present various background information related to our package. Section 5 describes the class structure of VNODE and elaborates on some of the design decisions incorporated in VNODE. In section 6, we illustrate how VNODE can be used, how new solvers can be constructed, and how part of a solver can be replaced. Concluding remarks are given in section 7.

The source code of VNODE, its documentation, and instructions on how to install this package can be obtained from

`www.cas.mcmaster.ca/~nedialk/Software/VNODE/VNODE.shtml`

2. Interval arithmetic and ranges of functions

An interval $[a] = [\underline{a}, \bar{a}]$ is the set of real numbers

$$[a] = [\underline{a}, \bar{a}] = \{\, x \mid \underline{a} \leq x \leq \bar{a}, \quad \underline{a},\, \bar{a} \in \mathbb{R} \,\}.$$

If $[a]$ and $[b]$ are intervals and $\circ \in \{+, -, *, /\}$, then the interval-arithmetic operations are defined by

$$[a] \circ [b] = \{\, x \circ y \mid x \in [a],\ y \in [b] \,\}, \quad 0 \notin [b] \text{ when } \circ = / \tag{4}$$

[14]. This definition can be written in the following equivalent form (we omit $*$ in the notation):

$$\begin{aligned}
[a] + [b] &= [\underline{a} + \underline{b}, \bar{a} + \bar{b}], && (5)\\
[a] - [b] &= [\underline{a} - \bar{b}, \bar{a} - \underline{b}], && (6)\\
[a][b] &= [\min\{\underline{a}\underline{b}, \underline{a}\bar{b}, \bar{a}\underline{b}, \bar{a}\bar{b}\}, \max\{\underline{a}\underline{b}, \underline{a}\bar{b}, \bar{a}\underline{b}, \bar{a}\bar{b}\}], && (7)\\
[a]/[b] &= [\underline{a}, \bar{a}][1/\bar{b}, 1/\underline{b}], \quad 0 \notin [b]. && (8)
\end{aligned}$$

The formulas (5–8) allow us to express the interval arithmetic operations using the endpoints of the interval operands, thus obtaining efficient formulas for real interval arithmetic.

The strength of interval arithmetic when implemented on a computer is in computing rigorous enclosures of real operations by including rounding errors in the computed bounds. To include such errors, we round the resulting (real) intervals in (5–8) outwards, thus obtaining machine interval arithmetic. For example, when adding intervals, we round $\underline{a}+\underline{b}$ down and round $\bar{a}+\bar{b}$ up. For simplicity of the discussion, we assume real interval arithmetic in this paper. Because of the outward roundings, intervals computed in machine interval arithmetic always contain the corresponding real intervals.

Definition (4) and formulas (5–8) can be extended to vectors and matrices having interval components. The arithmetic operations involving interval vectors and matrices are defined by the same formulas as in the scalar case, except that scalars are replaced by intervals and each standard arithmetic operation is replaced by the corresponding interval arithmetic operation defined above in (5–8).

An important property of interval-arithmetic operations is their monotonicity: if $[a]$, $[a_1]$, $[b]$, and $[b_1]$ are such that $[a] \subseteq [a_1]$ and $[b] \subseteq [b_1]$, then

$$[a] \circ [b] \subseteq [a_1] \circ [b_1], \qquad \circ \in \{+, -, *, /\}\,. \tag{9}$$

Using (9), we can *rigorously* enclose the range of a function.

Let $f : \mathcal{D} \to \mathbb{R}$ be a function, where $\mathcal{D} \subseteq \mathbb{R}^n$, and assume that we have an algorithm for evaluating $f(x)$, for $x \in \mathcal{D}$, that uses the arithmetic operations $\{+, -, *, /\}$ and standard built-in functions only. The interval-arithmetic evaluation of f on $[a] \subseteq \mathcal{D}$, which we denote by $f([a])$, is obtained by replacing each occurrence of x in the algorithm for $f(x)$ by $[a]$, by replacing the standard functions with enclosures of their ranges, and by performing interval-arithmetic operations instead of the real operations. From (9), the range of f, $\{\, f(x) \mid x \in [a]\,\}$, is always contained in $f([a])$.

A value for $f([a])$ is uniquely determined from the code list, or computational graph, of f. However, mathematically equivalent expressions for f can give different values for $f([a])$. For example, $x(y+z)$ and $xy + xz$ are mathematically equivalent for scalars, but may have different values if x, y, and z are intervals. Nevertheless, the value for $f([a])$ is guaranteed to enclose the true range of f, regardless of which expression for f is used.

As alluded to above, one reason for obtaining different values for $f([a])$ is that the distributive law, $x(y+z) = xy + xz$, does not hold in general in interval arithmetic [2]. However, for any three intervals $[a]$, $[b]$, and $[c]$, the subdistributive law

$$[a]([b] + [c]) \subseteq [a][b] + [a][c]$$

holds. This implies that if we rearrange an interval expression, we may obtain tighter bounds.

Addition of interval matrices is associative, but multiplication of interval matrices is not associative in general [22]. Also, the distributive law does not hold in general for interval matrices [22].

If $f : \mathcal{D} \to \mathbb{R}$ is continuously differentiable on $\mathcal{D} \subseteq \mathbb{R}^n$ and $[a] \subseteq \mathcal{D}$, then, for any x and $b \in [a]$, $f(x) = f(b) + f'(\eta)(x - b)$ for some $\eta \in [a]$ by the mean-value theorem. Thus, for any x and $b \in [a]$,

$$f(x) \in f_{\mathrm{M}}([a], b) \equiv f(b) + f'([a])([a] - b) \tag{10}$$

[14]. Consequently, $\{ f(x) \mid x \in [a] \} \subseteq f_{\mathrm{M}}([a], b)$ for any $b \in [a]$. The mean-value form, $f_{\mathrm{M}}([a], b)$, is popular in interval methods since it often gives better enclosures for the range of f than the straightforward interval-arithmetic evaluation of f itself.

3. An overview of the methods employed in VNODE

Currently, the jth step $(j \geq 1)$ of a validated method in VNODE consists of two phases:

ALGORITHM I: Compute a stepsize $h_{j-1} = t_j - t_{j-1}$ and an a priori enclosure $[\tilde{y}_{j-1}]$ of the solution such that $y(t; t_{j-1}, y_{j-1})$ is guaranteed to exist for all $t \in [t_{j-1}, t_j]$ and all $y_{j-1} \in [y_{j-1}]$, and $y(t; t_{j-1}, [y_{j-1}]) \subseteq [\tilde{y}_{j-1}]$ for all $t \in [t_{j-1}, t_j]$.

ALGORITHM II: Compute a tight enclosure $[y_j] \subseteq [\tilde{y}_{j-1}]$ such that

$$y(t_j; t_0, [y_0]) \subseteq [y_j].$$

In VNODE, a stepsize is predicted by a stepsize control strategy, and this stepsize is supplied as an input to Algorithm I. The method implementing it attempts to validate existence and uniqueness of the solution with the input stepsize or a stepsize that is smaller than the input one.

Then, Algorithm II computes a tight enclosure of the solution with h_{j-1}. With a variable stepsize control, if a tolerance requirement is satisfied, h_{j-1} is accepted; otherwise, h_{j-1} is reduced and tight bounds are computed with the reduced stepsize.

In this section, we briefly discuss automatic generation of Taylor coefficients, methods for implementing these two algorithms, and a variable stepsize strategy. More details can be found in [16, 20]. The methods in VNODE are of constant order, and developing a good order control strategy remains a problem for further research.

3.1. Automatic generation of Taylor coefficients

Since the interval methods considered here use Taylor series coefficients, we outline the scheme for their generation. More details can be found in [4] or [14], for example.

If we know the Taylor coefficients $(u_j)_i \equiv u^{(i)}(t_j)/i!$ and $(v_j)_i \equiv v^{(i)}(t_j)/i!$ for $i = 0, 1, \dots, p$ for two functions u and v, then we can compute the pth Taylor coefficient of $u \pm v$, uv, and u/v by standard calculus formulas [14].

We introduce the sequence of functions

$$\begin{aligned} f^{[0]}(y) &= y, \\ f^{[i]}(y) &= \frac{1}{i}\left(\frac{\partial f^{[i-1]}}{\partial y} f\right)(y), \qquad \text{for } i \geq 1. \end{aligned}$$

For the IVP $y' = f(y)$, $y(t_j) = y_j$, the ith Taylor coefficient of $y(t; t_j, y_j)$ at t_j,

$$(y_j)_i = \frac{y^{(i)}(t_j)}{i!}, \tag{11}$$

satisfies

$$(y_j)_i = f^{[i]}(y_j) = \frac{1}{i}(f(y_j))_{i-1}, \quad i \geq 1, \tag{12}$$

where $(f(y_j))_{i-1}$ is the $(i-1)$st Taylor coefficient of f evaluated at y_j. That is,

$$(f(y_j))_{i-1} = \frac{1}{(i-1)!} \left. \frac{d^{i-1} f(y(t; t_j, y_j))}{dt^{i-1}} \right|_{y(t)=y_j}.$$

Using (12) and formulas for the Taylor coefficients of sums, products, quotients, and the standard functions, we can recursively evaluate $(y_j)_i$, for $i \geq 1$. It can be shown that, to generate k Taylor coefficients, we need at most $O(k^2)$ times as much computational work as we require to evaluate $f(y)$ [14]. This is far more efficient than the standard symbolic generation of Taylor coefficients.

If we have a procedure to compute the ith point Taylor coefficients $f^{[i]}(y_j)$ of $y(t; t_j, y_j)$ and perform the computations in interval arithmetic with $[y_j]$ instead of y_j, to compute $f^{[i]}([y_j])$, we obtain a procedure to compute the interval Taylor coefficients of $y(t; t_j, y_j)$ at t_j, since

$$\left\{ \left. (y(t; t_j, y_j))_i \right|_{t=t_j} \mid y_j \in [y_j] \right\} \subseteq f^{[i]}([y_j]).$$

3.2. Validating existence and uniqueness of the solution

Using the Picard-Lindelöf operator and the Banach fixed-point theorem, one can show that if h_{j-1} and $[\tilde{y}^0_{j-1}] \supseteq [y_{j-1}]$ satisfy

$$[\tilde{y}_{j-1}] = [y_{j-1}] + [0, h_{j-1}]\, f([\tilde{y}^0_{j-1}]) \subseteq [\tilde{y}^0_{j-1}], \tag{13}$$

then the ODE (1) with the initial condition $y(t_{j-1}) = y_{j-1}$ has a unique solution $y(t; t_{j-1}, y_{j-1})$ that satisfies $y(t; t_{j-1}, y_{j-1}) \in [\tilde{y}_{j-1}]$ for all $t \in [t_{j-1}, t_j]$ and all $y_{j-1} \in [y_{j-1}]$, [7].

We refer to a method based on (13) as a first-order enclosure (FOE) method. Such a method can be easily implemented, but a serious disadvantage of this approach is that it often restricts the stepsize Algorithm II can take.

One can obtain methods that enable larger stepsizes by using polynomial enclosures [13] or more Taylor series terms in the sum in (13), thus obtaining a high-order enclosure (HOE) method [6, 21]. In the latter, we determine h_{j-1} and $[\tilde{y}_{j-1}]$ such that

$$\sum_{i=0}^{k-1} (t - t_{j-1})^i f^{[i]}(y_{j-1}) + (t - t_{j-1})^k f^{[k]}([\tilde{y}_{j-1}]) \subseteq [\tilde{y}_{j-1}] \tag{14}$$

holds for all $t \in [t_{j-1}, t_j]$ and all $y_{j-1} \in [y_{j-1}]$. Then the ODE (1) with the initial condition $y(t_{j-1}) = y_{j-1}$ has a unique solution $y(t; t_{j-1}, y_{j-1})$ that satisfies $y(t; t_{j-1}, y_{j-1}) \in [\tilde{y}_{j-1}]$ for all $t \in [t_{j-1}, t_j]$ and all $y_{j-1} \in [y_{j-1}]$.

In [21], we show that, for many problems, an interval method based on (14) is more efficient than one based on (13). In VNODE, we implement the HOE method described in [21].

3.3. Computing a tight enclosure of the solution

3.3.1. *Interval Taylor series methods*

Consider the Taylor series expansion

$$y_j = y_{j-1} + \sum_{i=1}^{k-1} h^i_{j-1} f^{[i]}(y_{j-1}) + h^k_{j-1} f^{[k]}(y; t_{j-1}, t_j), \tag{15}$$

where $y_{j-1} \in [y_{j-1}]$, and $f^{[k]}(y; t_{j-1}, t_j)$ denotes $f^{[k]}$ with its lth component ($l = 1, 2, \ldots, n$) evaluated at $y(\xi_{j-1,l})$ for some $\xi_{j-1,l} \in [t_{j-1}, t_j]$.

The remainder term $h^k_{j-1} f^{[k]}(y; t_{j-1}, t_j)$ can be enclosed by evaluating $f^{[k]}$ at $[\tilde{y}_{j-1}]$ and multiplying it by h^k_{j-1}, since $y(t; t_{j-1}, y_{j-1}) \in [\tilde{y}_{j-1}]$ for all $t \in [t_{j-1}, t_j]$ and all $y_{j-1} \in [y_{j-1}]$. We denote the enclosure of this term by

$$[z_j] = h^k_{j-1} f^{[k]}([\tilde{y}_{j-1}]). \tag{16}$$

Let $J\left(f^{[i]};[y_{j-1}]\right)$ be the Jacobian of $f^{[i]}$ evaluated at $[y_{j-1}]$ and denote

$$[S_{j-1}] = I + \sum_{i=1}^{k-1} h_{j-1}^{i}\, J\left(f^{[i]};[y_{j-1}]\right). \tag{17}$$

These Jacobians can be computed by generating the Taylor coefficients for the solution of the associated variational equation

$$Y' = \frac{\partial f}{\partial y} Y, \quad Y(t_{j-1}) = I, \tag{18}$$

where I is the identity matrix [12]; see also [30, 20].

If we apply the mean-value theorem to $f^{[i]}$ in (15) and use (16–17), we obtain that, for any $\hat{y}_{j-1} \in [y_{j-1}]$,

$$\begin{aligned} y(t_j;t_0,[y_0]) \in [y_j] = \hat{y}_{j-1} \;&+\; \sum_{i=1}^{k-1} h_{j-1}^{i} f^{[i]}(\hat{y}_{j-1}) \\ &+\; [z_j] + [S_{j-1}]([y_{j-1}] - \hat{y}_{j-1}). \end{aligned} \tag{19}$$

We can interpret the above formula for the enclosure $[y_j]$ as an approximate point solution, $\hat{y}_{j-1} + \sum_{i=1}^{k-1} h_{j-1}^{i} f^{[i]}(\hat{y}_{j-1})$, plus an enclosure of the local error, $[z_j]$, plus an enclosure of the global error, $[S_{j-1}]([y_{j-1}] - \hat{y}_{j-1})$, that we propagate to t_j.

We refer to a method based on (19) as the *direct method* [20]. It is less expensive than the methods discussed later in this paper, but the direct method frequently performs poorly, because the interval vector $[S_{j-1}]([y_{j-1}] - \hat{y}_{j-1})$ may significantly overestimate the set

$$\{\, S_{j-1}(y_{j-1} - \hat{y}_{j-1}) \mid S_{j-1} \in [S_{j-1}],\; y_{j-1} \in [y_{j-1}] \,\}.$$

Such overestimations normally accumulate as the integration proceeds, and the computed bounds become unreasonably wide. This is an instance of the *wrapping effect* [14]. An extensive study of this phenomenon, and references to works related to it, can be found in [19].

3.3.2. *Lohner's QR-factorization method*

We present a brief description of Lohner's QR-factorization method [12], which is one of the most successful general-purpose methods for reducing the wrapping effect in interval methods for IVPs for ODEs.

Denote the midpoint of an interval $[a]$ by $\mathrm{m}([a]) = (\underline{a} + \bar{a})/2$; the midpoint of an interval vector or an interval matrix is defined componentwise. In this method, we compute

$$[y_j] = \hat{y}_{j-1} + \sum_{i=1}^{k-1} h_{j-1}^{i} f^{[i]}(\hat{y}_{j-1}) + [z_j] + ([S_{j-1}]A_{j-1})\,[r_{j-1}], \tag{20}$$

instead of (19), and propagate

$$[r_j] = \left(A_j^{-1}([S_{j-1}]A_{j-1})\right)[r_{j-1}] + A_j^{-1}([z_j] - \mathrm{m}([z_j])), \tag{21}$$

where

$$\begin{aligned}
\hat{y}_0 &= \mathrm{m}([y_0]), \quad [r_0] = [y_0] - \hat{y}_0, \quad \text{and} \\
\hat{y}_j &= \hat{y}_{j-1} + \sum_{i=1}^{k-1} h_{j-1}^i f^{[i]}(\hat{y}_{j-1}) + \mathrm{m}([z_j]) \quad (j \geq 1).
\end{aligned}$$

Here, $A_0 = I$, and for $j \geq 1$, $A_j \in \mathbb{R}^{n\times n}$ is nonsingular.

To reduce the wrapping effect in propagating the solution, on each step of this method, the enclosure of the solution is also represented in the form of a parallelepiped,

$$y(t_j; t_0, [y_0]) \subseteq \{\, \hat{y}_j + A_j r_j \mid r_j \in [r_j] \,\}.$$

If A_j is the orthogonal matrix from the QR factorization of the matrix $\mathrm{m}([S_{j-1}]A_{j-1})$, we have the *QR-factorization method*. One explanation why this method is successful at reducing the wrapping effect is that it encloses the solution on each step in a moving orthogonal coordinate system that "matches" the solution set [12]. Another explanation is that the QR-factorization technique improves the stability of the interval Taylor series method [19].

3.3.3. *The interval Hermite-Obreschkoff method*

We have developed an interval Hermite-Obreschkoff (IHO) method [16, 17], which is based on the formula

$$\begin{aligned}
\sum_{i=0}^{q} (-1)^i c_i^{q,p} h_{j-1}^i f^{[i]}(y_j) &= \sum_{i=0}^{p} c_i^{p,q} h_{j-1}^i f^{[i]}(y_{j-1}) \\
&+ (-1)^q \frac{q!p!}{(p+q)!} h_{j-1}^k f^{[k]}(y; t_{j-1}, t_j),
\end{aligned} \tag{22}$$

where $k = p + q + 1$,

$$c_i^{q,p} = \frac{q!\,(q+p-i)!}{(p+q)!\,(q-i)!} \qquad (q, p, \text{ and } i \geq 0),$$

$y_{j-1} = y(t_{j-1}; t_0, y_0)$, and $y_j = y(t_j; t_0, y_0)$, [23, 24].

This method consists of a predictor phase and a corrector phase. The predictor computes an enclosure $[y_j^0]$ of the solution at t_j, and using this enclosure, the corrector computes a tighter enclosure $[y_j] \subseteq [y_j^0]$ at t_j. If $q > 0$, (22) is an implicit scheme and the corrector applies a Newton-like step to tighten $[y_j^0]$. The QR-factorization technique is used to reduce the wrapping effect in this method.

It is shown in [16] that for the same order and stepsize, the IHO method has smaller local error, has better stability, and requires fewer Jacobian evaluations than an ITS method. The extra cost of the Newton step is one matrix inversion and a few matrix multiplications.

3.4. Variable stepsize control

We outline the variable stepsize strategy employed in VNODE. We consider first the case when Algorithm II is implemented by an ITS method and then the case when this part of VNODE is implemented by an IHO method.

3.4.1. *ITS method*

Denote the width of an interval $[a]$ by $\mathrm{w}([a]) = \bar{a} - \underline{a}$. The width of an interval vector is defined componentwise. We measure the local excess [16] on each step by

$$\mathrm{err}_j = h_{j-1}^k \|\mathrm{w}(f^{[k]}([\tilde{y}_{j-1}]))\| \quad (j \geq 1),$$

where $\|\cdot\|$ denotes the maximum norm.

For an interval $[a]$, we define its magnitude by $|[a]| = \max\{\,|\underline{a}|\,, |\bar{a}|\,\}$. For an interval vector, we define its magnitude componentwise. Then the maximum norm of an interval vector $[a]$ is $\|[a]\| = \||[a]|\|$.

Let E_a and E_r be absolute and relative tolerances, respectively, and set

$$\mathrm{Tol}_j = E_a + E_r\,\|[y_j]\|.$$

Here, the tolerance, Tol_j, is a bound on the local excess per unit step. That is, we satisfy on each successful step

$$\mathrm{err}_j \leq h_{j-1}\mathrm{Tol}_j.$$

On the first attempt on the first step, we compute

$$h_{0,0} = 0.5\left(\frac{\mathrm{Tol}_0}{\|(k+1)f^{[k+1]}([y_0])\|}\right)^{1/k}. \tag{23}$$

(In [20, 21] we considered the above formula for the initial stepsize, but without the factor 0.5.) Subsequently, we set

$$h_{j,0} = h_{j-1}\left(\frac{h_{j-1}\mathrm{Tol}_j}{\mathrm{err}_j}\right)^{1/(k-1)} \tag{24}$$

after a rejected step, and

$$h_{j,0} = 0.9\, h_{j-1} \left(\frac{0.5\, h_{j-1} \mathrm{Tol}_j}{\mathrm{err}_j} \right)^{1/(k-1)} \tag{25}$$

after a successful step. For more details, see [16, 20].

3.4.2. *IHO method*
Assuming that $p + q + 1 = k$, the local excess in the IHO method is given by

$$\mathrm{err}'_j = \gamma_{p,q}\, h^k_{j-1} \|\mathrm{w}(f^{[k]}([\tilde{y}_{j-1}]))\| \quad (j \geq 1),$$

where $k = p + q + 1$ and $\gamma_{p,q} = q!p!/(p+q)!$; cf. (22). With the IHO method in Algorithm II, we predict an initial stepsize with (23), and after the first step, we use (24–25), but with err_j replaced by err'_j.

4. Background to the software design

Computing validated solutions for IVPs for ODEs is not as developed an area as that of computing approximate solutions. For example, we still do not have well-studied order control strategies for validated methods or schemes suitable for stiff problems. Moreover, it seems difficult to obtain "derivative-free methods", such as an interval version of a Runge-Kutta or a multistep method.

With respect to the software tools involved, building a validated solver appears inherently more complex than building a standard ODE solver. One reason is that, in a validated solver, we need to incorporate an interval arithmetic package and a package for computing interval Taylor coefficients for the solution to an ODE and for the solution of the associated variational equation.

Currently, there are three published packages for computing guaranteed bounds on the solution of an IVP for an ODE: AWA [12], ADIODES [30] and COSY INFINITY [5].

AWA is an implementation of Lohner's method incorporating the FOE method for validating existence and uniqueness of the solution in Algorithm I. This package is written in Pascal-XSC [10], an extension of Pascal for scientific computing.

ADIODES is a C++ implementation of a solver using the FOE method in Algorithm I and Lohner's method in Algorithm II. The stepsize in both ADIODES and AWA is often severely restricted by the FOE method.

COSY INFINITY is a Fortran-based code to study and design beam physics systems. The method for verified integration of ODEs uses high-order Taylor polynomials with respect to time and the initial conditions. The wrapping effect is reduced by establishing functional dependency between initial and final conditions. For that purpose, the computations are carried out with Taylor polynomials with real floating-point coefficients and a guaranteed error bound for the remainder term.

VNODE is a set of C++ classes that encapsulate the methods discussed in this paper and the related data structures. The main purpose of our package is to compute validated solutions of IVPs for ODEs. However, VNODE is also intended to facilitate the numerical study and comparison of schemes and heuristics used in validated methods for IVPs for ODEs. Currently, there are two built-in solvers in this package, but we can also assemble other solvers by choosing methods from sets of methods. This enables us to isolate and compare methods implementing the same part of a solver. For example, we can construct two solvers that differ only in the method implementing Algorithm II. Then, the difference in the numerical results, obtained by executing the two solvers, will indicate the difference in the performance of Algorithm II (at least in the context of the other methods and heuristics used in VNODE).

We have chosen an object-oriented approach in designing VNODE and C++ to implement it. Our choice of C++ was determined mainly by the availability of software for automatic generation of interval Taylor coefficients. VNODE is built on top of the automatic differentiation (AD) packages TADIFF [4] and FADBAD [3], which are both implemented in C++. TADIFF is used to generate Taylor coefficients with respect to time for the solution of an ODE. FADBAD is employed to generate the associated variational equation by applying the forward mode of AD [26], and then TADIFF is used to generate the Taylor coefficients for the solution to this equation. Both VNODE and FADBAD/TADIFF are built on top of the PROFIL/BIAS [11] interval-arithmetic package, which is also written in C++.

For more details on the choice of C++, the automatic differentiation package, and the interval-arithmetic package, see [18].

5. Class structure of VNODE and design decisions

From an object-oriented perspective, it is useful to think of a numerical problem as an object containing all the information necessary to compute its solution. Also, we can think of a particular solver as an object containing the necessary data and functions to perform the integration of a given problem. From this perspective, we have split VNODE into classes related to the description of a numerical problem and classes that encapsulate methods and solvers. Furthermore, we categorize the classes in VNODE as

1. problem classes,
2. method classes, and
3. classes for generating Taylor coefficients,

where we consider the classes for generating Taylor coefficients as a separate set of method classes. In the next three subsections, we present a high-level description of the classes in (1)–(3) and some of the design decisions we have made. We do not list or describe the methods of these classes in this paper. Such descriptions are provided in the documentation of VNODE, which can be obtained from the URL address[3] given at the end of section 1.

These classes are depicted in Figures 1, 2, and 3, respectively. Each box in these figures denotes a class; a shaded box denotes an abstract class[4], the main purpose of which is to define a common interface for its subclasses. A line with $\triangle$ indicates an "is-a", or inheritance, relationship, while a line with $\Diamond$ indicates a "part-of", or aggregation, relationship. The latter is realized either by a containment of an object of a class `Y` within an object of a class `X` or by a pointer from `X` to `Y`. The line with $\circ$ in Figure 3 denotes aggregation of more than one object of the same class. The notation in these figures is similar to that suggested in [28].

5.1. Problem classes

The problem classes are shown in Figure 1. Class `ODE_PROBLEM` specifies the initial condition and the function to compute the right side. The initial condition is stored in an object of class `SOLUTION`, and a pointer to such an object is maintained in `ODE_PROBLEM`. This class also contains an object of the class `PROBLEM_INFO`. It indicates, for example, if the problem is constant coefficient, is scalar, has a closed form solution, or has a point initial condition.

[3] Throughout this paper, by the "distribution of VNODE", we refer to this address.

[4] We use shaded boxes to make the distinction between abstract and concrete classes visually clearer.

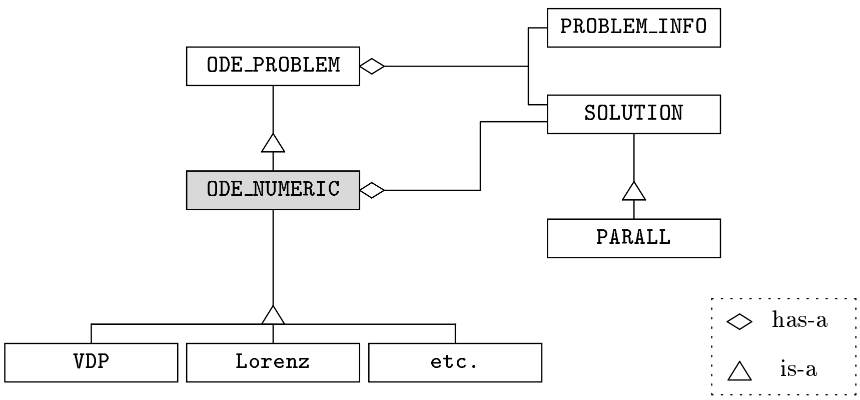

Figure 1. Problem classes.

Class ODE_NUMERIC specifies the numerical problem. In addition to inheriting the data and methods from ODE_PROBLEM, ODE_NUMERIC contains data such as absolute and relative error tolerances, the end point of the integration, and a pointer to a solution object. (ODE_NUMERIC aggregates two solution objects: one for representing the initial condition, inherited from ODE_PROBLEM, and one for representing the solution.)

Class SOLUTION stores the most recently obtained a priori and tight enclosures of the solution, $[\tilde{y}_{j-1}]$ and $[y_j]$ respectively, the value t_j where the tight enclosure is computed, and the value of the previous integration point, t_{j-1}. The PARALL class inherits the data from SOLUTION and also specifies an enclosure of the solution in the form of the parallelepiped $\{\, y_j + A_j r_j \mid r_j \in [r_j] \,\}$. This parallelepiped representation of the solution is used in reducing the wrapping effect in the ITS and IHO methods; see subsection 3.3.

A concrete "numerical" object is created by instantiating the class ODE_NUMERIC or a class derived from it. In Figure 1, the classes VDP and Lorenz denote classes that represent Van der Pol's equation and Lorenz's system, respectively. In section 6, we explain how such classes can be created "almost" automatically.

The methods in VNODE interact with the representation of a problem through a pointer to the ODE_NUMERIC class. Thus, we have separated the description of the problem being solved from the methods employed to compute its solution. Moreover, the concrete representation of a solution is disengaged from the description of a numerical problem. That is, in an ODE_NUMERIC object, we can keep a pointer to an object of class SOLUTION, but we can also initialize this pointer with an instance of a class derived from SOLUTION. Currently, VNODE uses mainly the parallelepiped representation. A new solution representation can be added by encapsulating it in a class and deriving this class from SOLUTION.

5.2. Method classes

The abstract class ODE_SOLVER, Figure 2, is a general description of a solver that solves an ODE_NUMERIC problem. We have tried to split ODE_SOLVER into classes that are likely to be needed by a concrete implementation of a solver. Brief descriptions of these classes follow.

SOLVER_FLAGS encapsulates several flags (e.g., indicating if a step is accepted, or if a step is the first one) needed in an integration, and SOLVER_STATS encapsulates different statistics (e.g. CPU time and number of accepted steps) collected during an integration.

The abstract class DATA_REPR is introduced to connect a solver with additional data and methods that may be needed. For example, the classes for generating Taylor coefficients (described in the next subsection) are derived from DATA_REPR. When creating a solver (see section 6), we specify a concrete class that is derived from DATA_REPR.

The STEP_CTRL class provides the interface to the methods for selecting an initial stepsize and a stepsize after the first stepsize has been selected. CONST_STEP returns a constant stepsize on each step, and VAR_STEP implements the stepsize selection scheme based on controlling the width of the remainder term; see subsection 3.4.

The abstract ODE_SOLVER class declares the function Integrate, whose purpose is to integrate an ODE problem. Integrate is abstract[5]; that is, it is declared, but no body is provided for it. As a result, instances of ODE_SOLVER cannot be created. However, the body for Integrate is given in the derived class VODE_SOLVER, which implements the scheme outlined in section 3.

We have divided this solver into four abstract methods: for selecting an order, selecting a stepsize, and computing a priori and tight enclosures of the solution. These methods are realized by the abstract classes ORDER_CTRL, STEP_CTRL, INIT_ENCL, and TIGHT_ENCL, respectively. Their purpose is to provide general interfaces to particular methods. In VODE_SOLVER, Integrate performs an integration by calling methods belonging to classes derived from ORDER_CTRL, STEP_CTRL, INIT_ENCL, and TIGHT_ENCL. We have used polymorphism in implementing this function: it calls abstract functions that are declared, but not defined, in these abstract classes. During execution, depending on the type (class) of the object set, a particular function will be called.

[5] In C++, abstract functions are normally called *pure virtual* functions.

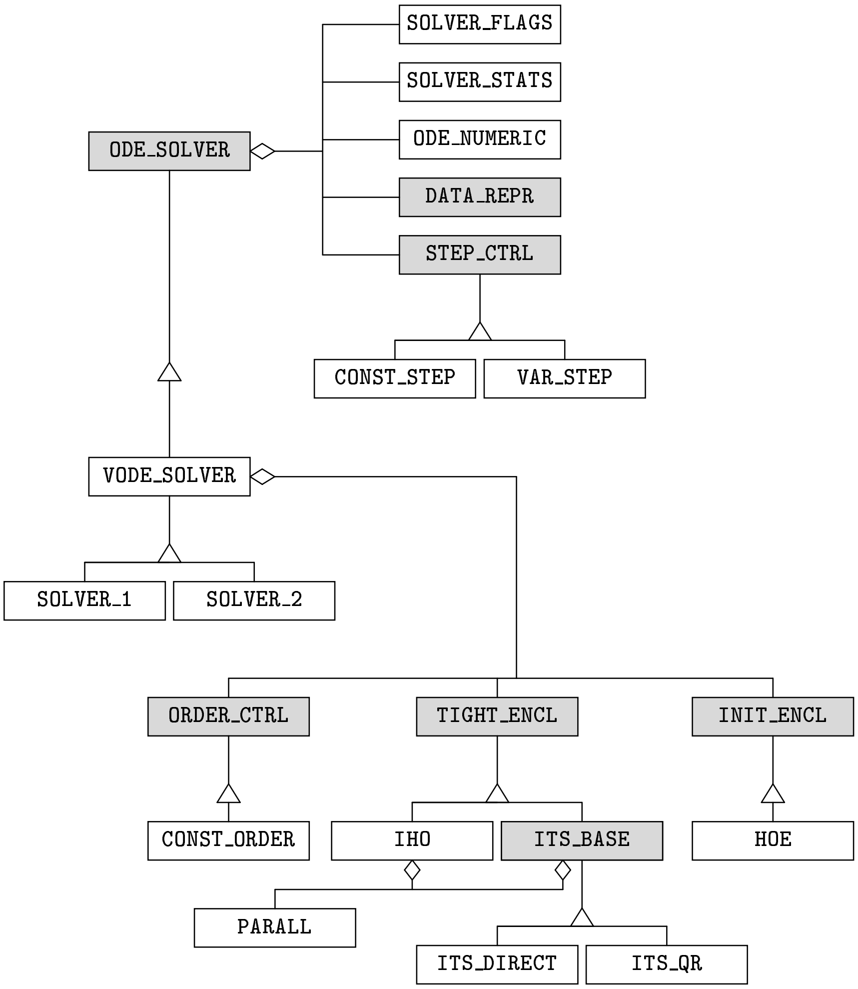

Figure 2. Method classes. The classes associated with `ODE_NUMERIC` are shown in Figure 1, and the classes associated with `DATA_REPR` are shown in Figure 3.

Currently, VNODE does not implement variable-order methods, and the class `ORDER_CTRL` has only one derived class, `CONST_ORDER`, whose role is to return a constant order.

There are three methods for implementing Algorithm II: an IHO method with the QR-factorization anti-wrapping strategy (`IHO`), the direct method (`ITS_DIRECT`), and the QR-factorization method (`ITS_QR`). We have factored out the invariant part of the ITS methods in the class `ITS_BASE`.

Algorithm I is implemented by the HOE method, which is encapsulated in the class `HOE` (derived from `INIT_ENCL`).

`SOLVER_1` and `SOLVER_2` encapsulate two "pre-configured" solvers. The former implements a solver with

- constant order (`CONST_ORDER`),
- variable stepsize control (`VAR_STEP`),
- the HOE method for validating existence and uniqueness of the solution (`HOE`), and
- an ITS method for computing tight bounds on the solution with the QR-factorization technique for reducing the wrapping effect (`ITS_QR`).

The latter solver is configured with objects of `CONST_ORDER`, `VAR_STEP`, `HOE`, and `IHO`. Hence, these two solvers differ only in the method for implementing Algorithm II.

We should note here that, in creating the classes `SOLVER_1` and `SOLVER_2`, we did not have to re-implement the `Integrate` function from `VODE_SOLVER` in these classes. `SOLVER_1` and `SOLVER_2` inherit the implementation of `Integrate`, which refers to abstract classes in its definition in `VODE_SOLVER` and does the work in `SOLVER_1` and `SOLVER_2` by referring to the concrete classes mentioned above.

The methods for order and stepsize control, computing a priori and tight bounds on the solution, and the method for integration are implemented through the *strategy* design pattern [8]. Using this pattern, we define classes that encapsulate algorithms performing the same task. An abstract class, strategy, declares the interface, but not the implementation, that is common to all supported algorithms. A derived class, *concrete strategy*, implements a concrete algorithm.

Here, we illustrate this pattern with the stepsize selection classes. The abstract class `STEP_CTRL` (see Figure 2) provides the interface to the method for selecting an initial stepsize and the method for predicting a stepsize, after the first stepsize has been selected. The derived classes CONST_STEP and VAR_STEP supply concrete implementations of these two methods.

Consider the function for predicting a stepsize, after the initial stepsize has been selected. This function is declared in STEP_CTRL as

```
virtual bool PredictStep( double & hPred, double h,
                          const PtrODENumeric ODE,
                          const INTERVAL_VECTOR  & LocErr,
                          const INTERVAL_VECTOR  & Ytight,
                          int order ) = 0;
```

PtrODENumeric is a pointer to ODE_NUMERIC, and INTERVAL_VECTOR is declared in PROFIL/BIAS. The input to PredictStep is the current stepsize, h, a pointer to an ODE problem, ODE, a reference to an interval vector enclosing the local error, LocErr, a reference to an interval vector enclosing the tight enclosure of the solution, Ytight, and the order of the method, order. PredictStep returns the new stepsize in hPred, and it returns true, if the just attempted step is accepted, and false otherwise.

Since this function is abstract, it is not defined in STEP_CTRL. The classes derived from STEP_CTRL must provide the body for this function. PredictStep in VAR_STEP implements the stepsize selection scheme described in subsection 3.4, while PredictStep in CONST_CTRL returns the value of the stepsize that is set in creating an object of CONST_CTRL (see Figure 9 in subsection 6.2). If we want to add a new stepsize strategy to VNODE, we have to define PredictStep in a class and derive it from STEP_CTRL.

The main advantage of the strategy pattern is that we can have different implementations for the same task, which allows us to vary algorithms during runtime. Since we cannot predict what parameters a new algorithm may require, the main difficulty in applying this pattern is in designing an interface, such that it contains a minimal set of parameters that will be sufficient for new algorithms.

One way to avoid this "interface" problem is to pass a pointer (or a reference) to a class that encapsulates the necessary parameters. In some sense, this approach would ensure a simple and uniform parameter passing. For example, we can pass a pointer to ODE_SOLVER to PredictStep, and then PredictStep determines the parameters it needs by referring back to ODE_SOLVER. This approach may require downcasting the pointer to ODE_SOLVER to a pointer to VODE_SOLVER, before PredictStep extracts the necessary parameters. Similarly, we can pass only a pointer to ODE_SOLVER to the methods for validating existence and uniqueness, computing tight bounds, and selecting an order. However, this approach has the following disadvantages, which are discussed in the next two paragraphs.

Passing an object that contains all parameters normally leads to invoking many "set" and "get" methods inside the called function, whereas we want to keep our programs concise. Of course, such methods can be avoided by making the associated data public, but in general, we do not want to expose the internal representation of a class. Furthermore, setting a value may involve a test to ensure that this value satisfies certain conditions. Thus, usually, we prefer to have "set" and "get" methods.

We would like to store data only when they are accepted. For example, suppose that `Ytight` is computed and is stored in the solution object of `ODE_NUMERIC`, and only a pointer to an `ODE_SOLVER` object is passed to `PredictStep`, which extracts `Ytight` through the pointer from `ODE_SOLVER` to `ODE_NUMERIC`. With a variable stepsize control in `PredictStep`, we do not accept a solution, if a tolerance requirement is not satisfied. Thus, if `Ytight` is not accepted, we have to ensure that the previous enclosure is available to repeat the integration. If we pass `Ytight` as a parameter, then we can store `Ytight` only after the solution is accepted.

In VNODE, we have tried to factor out the parameters that are likely to be used in most of the algorithms implementing the same task (for example, `hPred`, `h`, `LocErr`, `Ytight`, and `order` in `PredictStep`), and we encapsulate additional data and functions by passing a reference to `ODE_NUMERIC` or `ODE_SOLVER`. If a new method requires an interface that cannot be accommodated with the current one, then the new interface can be adapted to the current interface through the *adaptor* pattern [8].

Another difficulty associated with the strategy pattern is that a client of a strategy must be aware of different existing implementations. In VNODE, we have set default algorithms for the general user, for example, the methods in `SOLVER_1` and `SOLVER_2`, and give more advanced users the possibility to vary algorithms and add new ones.

A minor drawback of the strategy pattern is the communication overhead, if not all parameters to a method are used. For example, in `CONST_STEP`, each call to `PredictStep` returns a constant stepsize without any computations being done.

5.3. Classes for generating Taylor coefficients

The abstract classes `TAYLOR_ODE` and `TAYLOR_VAR` (see Figure 3) provide the interfaces to the methods for generating Taylor coefficients for the solution to an ODE and for the solution of its variational equation, respectively. Currently, VNODE uses the TADIFF package [4] to

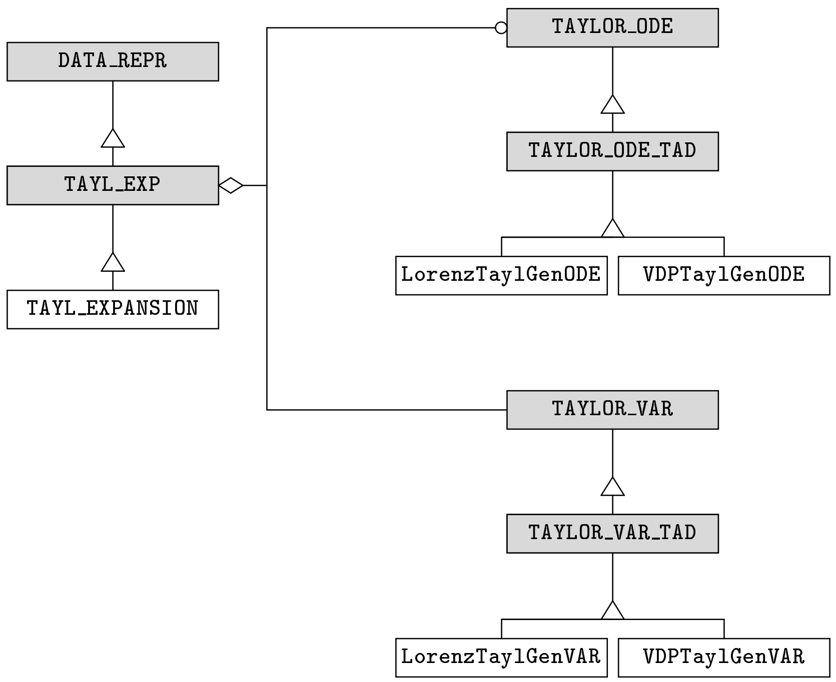

Figure 3. Classes for generating Taylor coefficients.

generate Taylor coefficients for the solution to an ODE, through the class `TAYLOR_ODE_TAD`; and TADIFF in combination with the FADBAD package [3] to compute the Taylor coefficients for the solution to the associated variational equation, through the class `TAYLOR_VAR_TAD`.

Concrete classes for generating Taylor coefficients can be derived from the classes `TAYLOR_ODE_TAD` and `TAYLOR_VAR_TAD`. In Figure 3, `LorenzTaylGenODE` and `VDPTaylGenODE` denote classes for computing the Taylor coefficients for the solution to the Lorenz and Van der Pol's systems, respectively, and `LorenzTaylGenVAR` and `VDPTaylGenVAR` denote the classes for computing the Taylor coefficients for the solution to the variational equations related to the Lorenz and Van der Pol's systems, respectively. Section 6 shows how such classes can be created.

The `TAYL_EXP` class contains pointers to `TAYLOR_ODE` objects for computing enclosures of point and interval Taylor coefficients, and a pointer to a `TAYLOR_VAR` object for generating Taylor coefficients for the solution to the associated variational equation.

`TAYL_EXPANSION` is a template class that inherits the data from `TAYL_EXP`. The purpose of `TAYL_EXPANSION` is to allow configuration with concrete Taylor coefficient generators. For example,

```
PtrDataRepr Data = new TAYL_EXPANSION<VDPTaylGenODE,VDPTaylGenVAR>
```

creates an object with instances of the classes `VDPTaylGenODE` and `VDPTaylGenVAR`. The `VODE_SOLVER` class accesses the Taylor coefficient generators through a pointer to `DATA_REPR`, which is initialized with a pointer to a `TAYL_EXPANSION` object.

If we want to incorporate a different AD package in VNODE, we have to encapsulate the interface to this package in classes, for example, `TAYLOR_ODE_NEW` and `TAYLOR_VAR_NEW`, and derive them from `TAYLOR_ODE` and `TAYLOR_VAR`, respectively. Then, concrete Taylor generators can be derived from `TAYLOR_ODE_NEW` and `TAYLOR_VAR_NEW`.

6. Integrating with VNODE

In this section, we illustrate how VNODE can be used to integrate Van Der Pol's equation, written as a first-order system:

$$\begin{aligned} y_1' &= y_2 \\ y_2' &= \mu(1-y_1^2)y_2 - y_1, \end{aligned} \tag{26}$$

where $\mu \in \mathbb{R}$. First, we discuss a basic usage, where a predefined solver is employed, and then we consider a more advanced usage, where we show how a solver can be constructed and how part(s) of it can be replaced.

More examples of how VNODE can be used are given in the distribution of VNODE.

6.1. Basic usage

To use VNODE to solve the ODE (26) numerically, we must

- write a function to compute the right side of (26),
- set the initial condition and the end point of the integration interval,

  and if necessary, the absolute and relative error tolerances, and
- create a main program.

Each of these steps is discussed in more detail in the following subsections.

6.1.1. *Specifying the function for computing the right side*

A template function for computing the right side of (26) should be specified in a `.h` file as shown in Figure 4. In this file, we also include

```
// FILE VDP.h

#ifndef INCLUDED_VDP_H
#define INCLUDED_VDP_H

#include "odenum.h"

template <typename Y_TYPE>
void VDPtemplate( Y_TYPE *yp, const Y_TYPE *y )
{
  double MU = 2.0;

  yp[0] = y[1];
  yp[1] = MU*(1 - sqr(y[0]))*y[1] - y[0];
}

#include "declode.h"

DECLARE_ODE_PROBLEM(VDP,2,VDPtemplate,"Van Der Pol's Equation");

#endif
```

Figure 4. Specifying the function for computing the right side of (26). `odenum.h` contains the declaration of `ODE_NUMERIC`.

the file `declode.h` and then the macro `DECLARE_ODE_PROBLEM`. This macro expands to the declaration of three classes: `VDP`, `VDPTaylODE`, and `VDPTaylVAR`; cf. Figures 1 and 3. The class `VDP` is a description of the problem to be integrated, `VDPTaylODE` is the Taylor coefficient generator for the solution to (26), and `VDPTaylVAR` is the Taylor coefficient generator for the solution to the associated variational equation. The parameters for this macro are

- a name for the ODE class, `VDP`,
- the size of the problem, `2`,
- the name of the template function, `VDPtemplate`, and
- the name of the problem, `"Van Der Pol's Equation"`.

6.1.2. *Specifying the problem parameters*

The user must specify t_0, $[y_0]$, t_m, and if necessary, absolute and relative error tolerances. These parameters can be set by writing the body of the function `LoadProblemParam`, which is a member of `ODE_NUMERIC`, as shown in Figure 5. Depending on the value of the parameter to

```
// FILE VDP.cc

#include "VDP.h"

void VDP :: LoadProblemParam( int ParamSet )
{
  INTERVAL_VECTOR Y(2);

  if ( ParamSet == 1 )
    {
      // Set initial and final points.
      SetT0(0);
      SetTend(10);

      Y(1) = INTERVAL(2.0);
      Y(2) = INTERVAL(0.0);

      // Set absolute and relative tolerances.
      SetAtol(1e-10);
      SetRtol(0);
    }

  if ( ParamSet == 2 )
    {
      SetT0(0);
      SetTend(20);

      Y(1) = INTERVAL(2.0);
      Y(2) = INTERVAL(0.0);
    }

  // Set an initial condition.
  SetInitCond(Y);
}
```

Figure 5. Specifying the problem parameters.

this function, different sets of parameters for the ODE problem can be specified. For example, when `ParamSet` is `1`, we have $y(0) = (2,0)^T$, $t_0 = 0$, $t_m = 10$, $E_a = 10^{-10}$, and $E_r = 0$; while if `ParamSet` is `2`, we have $y(0) = (2,0)^T$, $t_0 = 0$, $t_m = 20$, and default values of 10^{-12} for the relative and absolute tolerances.

6.1.3. *The main function*

In the file `DemoVDP.cc` in Figure 6, we include the files `VNODE.h` and `VDP.h`. The former includes several files containing declarations of functions and classes. The inclusion of the latter results in creating the class

```
// FILE DemoVDP.cc

#include "VNODE.h"
#include "VDP.h"

int main()
{
  // Initialize VNODE.
  VnodeInit();

  // Create the ODE problem and the data representation object.
  PtrODENumeric ODE = new VDP;
  PtrDataRepr DataRepr =
    new TAYLOR_EXPANSION<VDPTaylGenODE,VDPTaylGenVAR>;

  // Set the problem parameters.
  ODE->LoadProblemParam(1);

  // Create a solver.
  PtrODESolver Solver = new SOLVER_2( ODE, DataRepr );

  // The solution will be plotted.
  Solver->GraphOutput(true);

  cout << endl << "*** Integrating " << ODE->GetName() << endl;

  Solver -> Integrate();

  cout << endl
       << "   Enclosure at T = "
       << INTERVAL( ODE->GetTcur() )
       << endl;

  PrintVec( ODE->GetTightEncl() );

  // Plot Y(1) vs. t.
  ODE->DisplaySolution(1);

  // Save this plot in VDP.eps.
  ODE->DisplaySolution( 1, "VDP.eps" );

  return 0;
}
```

Figure 6. The main program for integrating Van Der Pol's equation. "Ptr" stands for pointer in VNODE.

VDP, for describing the ODE problem, and the classes VDPTaylGenODE and VDPTaylGenVAR, for generating the Taylor coefficients.

In the main function in this figure, we create a VDP object, and an instance of TAYLOR_EXPANSION, which is configured here with the classes VDPTaylGenODE and VDPTaylGenVAR. Then, we load the problem parameters, create an instance of the solver SOLVER_2, integrate the problem, and output the results.

In the next section, we illustrate how a solver can be constructed and how one or more parts of a solver can be replaced.

6.1.4. *Creating the executable*

To create an executable file, the files in Figures 4–6 must be compiled and linked with the library `libvnode.a`. For more details, see the makefile `Makefile.am` in the `examples` directory of the distribution of VNODE.

The output of the main program with `ParamSet` equal to 1 is shown in Figure 7. The timing results[6] in this paper are produced on a Sun UltraSPARC-IIi 440MHz. The global error[7] is measured by $\|\mathrm{w}([y_m])\|$ at t_m.

The plot of the first component of the solution of (26) versus t is shown in Figure 8. The dashed boxes in this figure denote the a priori bounds computed in Algorithm I. The upper and (respectively) lower bounds of the tight enclosures of the solution are connected with straight lines.

This plot is generated by the `DisplaySolution` function. The first call to it produces the plot on the screen; the second call writes this plot into an encapsulated postscript file `VDP.eps`, which is included in this paper in Figure 8.

6.2. ADVANCED USAGE

6.2.1. *Creating a solver*

When creating an instance of the VODE_SOLVER class, we can specify order and stepsize control objects, an object for validating existence and uniqueness of the solution, and an object for computing tight bounds on the solution. For example, suppose that we want to create a solver that uses

[6] Since the values for the user time are small, they may not be very accurate.

[7] It is reasonable to call $\mathrm{w}([y_m])$ the global error in this case because the integration is started with a point initial condition. The interpretation of $\mathrm{w}([y_m])$ is a little more complicated when the integration is started with an interval initial condition, $[y_0]$, since then $[y_m]$ includes both $y(t_m; t_0, [y_0])$ and the *excess* due to the numerical integration.

```
*** Integrating Van Der Pol's Equation

 *** Integrated from [0,0] to [10,10]
 *** Global Error    1.21e-07
 *** User Time       2.80e-01(sec)
 *** Accepted Steps  54
 *** Rejected Steps  0

   Enclosure at T = [10,10]
        Y(1) =  0.8415536[072893656,972837094]
        Y(2) = -1.089047[7959574447,9173730153]
```

Figure 7. Output of the program from Figure 6.
The notation `0.8415536[072893656,972837094]` is a compact representation of the interval $[0.8415536\underline{072893656}, 0.8415536\underline{972837094}]$, and similarly, the notation `-1.089047[7959574447,9173730153]` is a compact representation of the interval $-[1.089047\underline{9173730153}, 1.089047\underline{7959574447}]$.

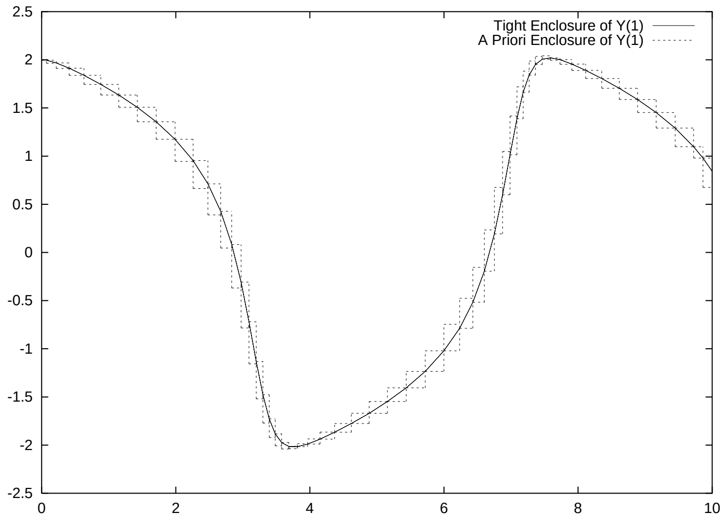

Figure 8. Plot of the first component of the solution of (26) versus t.

1. constant order,

2. a stepsize controller that returns a constant stepsize,

3. a high-order Taylor series method for validating existence and uniqueness of the solution, and

4. a Taylor series method for computing tight bounds on the solution.

In Figure 9, we first create the objects implementing (1)–(4) and then create an instance of `VODE_SOLVER`. In this solver, validating existence and uniqueness and computing tight bounds is done with the order that is set in the creation of the `CONST_ORDER` object. The stepsize controller always returns the constant stepsize that is set in creating the `CONST_STEP` object. This stepsize is an input to Algorithm I, which may reduce it.

6.2.2. *Replacing part of a solver*
Suppose that we want to compare the ITS method with the IHO method for the ODE (26), and we want to have the same methods implementing (1)–(3) in this comparison. After we integrate with the ITS method (see Figure 9), we can replace it by the IHO method in the line

```
Solver->SetTightEncl( new IHO ( (order-1)/2, (order-1)/2 ) );
```

and integrate again. In creating the `IHO` object, we specify the two parameters p and q (cf. section 3.3.3), which are set to `(order-1)/2`. Note that we do not change the rest of the methods in this solver.

The output of the program in Figure 9 is shown in Figure 10.

7. Concluding Remarks

We have presented a high-level description of VNODE and have illustrated how it can be used. For "simple" integrations with this package, we recommend that one follows the steps outlined in subsection 6.1. For "more complicated" integrations, one should also consider the examples in the distribution of VNODE (cf. section 1), in addition to the example in subsection 6.2.

Generally, `SOLVER_2` should be preferred over `SOLVER_1`, since the IHO method in `SOLVER_2` performs better than the ITS method in `SOLVER_1`. The direct method can be used for problems where it is known that the wrapping effect does not occur. Such problems include, for example, quasi-monotone problems [20].

We hope that VNODE will be of use to researchers interested in computing validated solutions of IVPs for ODEs. We also hope that VNODE will assist numerical analysts in developing, testing and evaluating schemes and heuristics for validated ODE solvers. As mentioned earlier, we still need good heuristics for order control and methods suitable for stiff problems. Although the former may not be difficult to

```
// FILE DemoVDP2.cc

#include "VNODE.h"
#include "VDP.h"

int main()
{
  // Initialize VNODE.
  VnodeInit();

  // Create the ODE problem and the data representation object and
  // load the problem parameters.
  PtrODENumeric ODE = new VDP;
  PtrDataRepr DataRepr =
    new TAYLOR_EXPANSION<VDPTaylGenODE,VDPTaylGenVAR>;

  ODE->LoadProblemParam(2);

  int order = 15;
  double h = 0.1;

  // Create methods for order control, stepsize control, validating
  // existence and uniqueness and computing tight bounds on the
  // solution.
  PtrOrderCtrl OrderCtrl = new CONST_ORDER(order);
  PtrStepCtrl  StepCtrl  = new CONST_STEP(h);
  PtrInitEncl  InitEncl  = new HOE;
  PtrTightEncl TightEncl = new ITS_QR;

  // Create a solver.
  PtrVODESolver Solver = new VODE_SOLVER( ODE, DataRepr,
                                          OrderCtrl, StepCtrl,
                                          InitEncl, TightEncl );

  cout << endl << "*** Integrating " << ODE->GetName() << endl;
  cout << endl << "*** With the "
       << Solver->GetTightEncl()->GetName() << endl;

  Solver -> Integrate();

  // Replace the ITS_QR method with the IHO method.
  Solver->SetTightEncl( new IHO( (order-1)/2, (order-1)/2 ) );
  cout << endl << "*** With the "
       << Solver->GetTightEncl()->GetName() << endl;

  Solver -> Integrate();

  return 0;
}
```

Figure 9. Creating a solver and replacing a method in it.

```
*** Integrating Van Der Pol's Equation

*** With the ITS: QR Method

 *** Integrated from [0,0] to [20,20]
 *** Global Error    1.42e-06
 *** User Time       7.40e-01(sec)
 *** Accepted Steps  204
 *** Rejected Steps  0

*** With the IHO Method

 *** Integrated from [0,0] to [20,20]
 *** Global Error    9.50e-08
 *** User Time       7.10e-01(sec)
 *** Accepted Steps  204
 *** Rejected Steps  0
```

Figure 10. The output of the program from Figure 9.

achieve, the latter is a challenging task, due to the explicit nature of the formula for the associated truncation error (see [16] for more details).

As reported in [15], most of the computing time, 50% to 80%, in VNODE is spent on generating Taylor coefficients for the solutions to the variational equation. Thus, an interval method for IVPs for ODEs may be made more competitive if a "derivative-free" method is developed. Although there is some preliminary work on interval Runge-Kutta methods [25], constructing such methods and comparing them with the existing ones would involve substantial work.

Acknowledgements

Dr. Emil Sekerinski made several helpful comments on subtle issues related to the object-oriented design and use of notation.

References

[1] D. Achlioptas, Setting 2 variables at a time yields a new lower bound for random 3-SAT, Technical report MSR-TR-99-96, Microsoft Research, Microsoft Corp., One Microsoft Way, Redmond, WA 98052, December 1999.

[2] G. Alefeld and J. Herzberger, *Introduction to interval computations*, Academic Press, New York, 1983.

[3] C. Bendsten and O. Stauning, FADBAD, a flexible C++ package for automatic differentiation using the forward and backward methods, Technical report 1996-x5-94, Department of Mathematical Modelling, Technical University of Denmark, DK-2800, Lyngby, Denmark, August 1996.

[4] C. Bendsten and O. Stauning, TADIFF, a flexible C++ package for automatic differentiation using Taylor series, Technical report 1997-x5-94, Department of Mathematical Modelling, Technical University of Denmark, DK-2800, Lyngby, Denmark, April 1997.

[5] M. Berz and K. Makino, Verified integration of ODEs and flows using differential algebraic methods on high-order Taylor models, Reliable Computing, 4 (1998), 361–369.

[6] G.F. Corliss and R. Rihm, Validating an a priori enclosure using high-order Taylor series, in: *Scientific Computing, Computer Arithmetic, and Validated Numerics*, eds. G. Alefeld and A. Frommer, Akademie Verlag, Berlin, 1996, pp. 228–238.

[7] P. Eijgenraam, *The Solution of initial value problems using interval arithmetic*, Mathematical Centre Tracts No. 144, Stichting Mathematisch Centrum, Amsterdam, 1981.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*, Addison Wesley Professional Computing Series, Addison-Wesley, 1994.

[9] T. Henzinger, B. Horowitz, R. Majumdar, and H. Wong-Toi, Beyond HyTech: hybrid systems analysis using interval numerical methods, in: *HSCC 00: Hybrid Systems—Computation and Control*, Lecture Notes in Computer Science 1790, eds. N. Lynch and B. Krogh, Springer-Verlag, 2000, pp. 130–144.

[10] R. Klatte, U. Kulisch, M. Neaga, D. Ratz, and C. Ullrich, *Pascal–XSC: language reference with examples*, Springer-Verlag, Berlin, 1992.

[11] O. Knüppel, PROFIL/BIAS – a fast interval library, Computing, 53 (1994), 277–287.

[12] R.J. Lohner, Einschließung der Lösung gewöhnlicher Anfangs– und Randwertaufgaben und Anwendungen, PhD thesis, Universität Karlsruhe, 1988.

[13] R.J. Lohner, Step size and order control in the verified solution of IVP with ODE's, Talk at SciCADE'95, International Conference on Scientific Computation and Differential Equations, Stanford University, Calif., March 28 – April 1, 1995.

[14] R.E. Moore, *Interval analysis*, Prentice-Hall, Englewood Cliffs, N.J., 1966.

[15] N.S. Nedialkov and K.R. Jackson, Towards assessing validated methods for IVPs for ODEs, May 1999, Talk at the Minisymposium on Advances in Validated Solution of ODEs: Theory, Software, and Applications, SIAM Annual Meeting, Atlanta, GA, USA, May 1999.

[16] N.S. Nedialkov, Computing rigorous bounds on the solution of an initial value problem for an ordinary differential equation, PhD thesis, Department of Computer Science, University of Toronto, Toronto, Canada, M5S 3G4, February 1999.

[17] N.S. Nedialkov and K.R. Jackson, An interval Hermite-Obreschkoff method for computing rigorous bounds on the solution of an initial value problem for an ordinary differential equation, Reliable Computing, 5 (1999), 289–310, 1999. Also in: *Developments in Reliable Computing*, ed. T. Csendes, Kluwer, Dordrecht, Netherlands, 1999, pp. 289–310.

[18] N.S. Nedialkov and K.R. Jackson, ODE software that computes guaranteed bounds on the solution, in: *Advances in Software Tools for Scientific Computing*, eds. H.P. Langtangen, A.M. Bruaset, and E. Quak, Springer-Verlag, 1999, pp. 197–224.

[19] N.S. Nedialkov and K.R. Jackson, A new perspective on the wrapping effect in interval methods for initial value problems for ordinary differential equations, in: *Perspectives on Enclosure Methods*, eds. A. Facius, U. Kulisch, and R. Lohner, Springer-Verlag, Vienna, 2001, pp. 219–264.

[20] N.S. Nedialkov, K.R. Jackson, and G.F. Corliss, Validated solutions of initial value problems for ordinary differential equations, Applied Mathematics and Computation, 105 (1999), 21–68.

[21] N.S. Nedialkov, K.R. Jackson, and J.D. Pryce, An effective high-order interval method for validating existence and uniqueness of the solution of an IVP for an ODE, Reliable Computing, 7 (2001), pp. 449–465.

[22] A. Neumaier, *Interval methods for systems of equations*, Cambridge University Press, Cambridge, 1990.

[23] N. Obreschkoff, Neue Quadraturformeln, Abh. Preuss. Akad. Wiss. Math. Nat. Kl., 4, 1940.

[24] N. Obreschkoff, Sur le quadrature mecaniques, Spisanie Bulgar. Akad. Nauk. (Journal of the Bulgarian Academy of Sciences), 65 (1942), 191–289.

[25] K. Petras, Validating Runge-Kutta methods for ODEs with analytic right-hand side, Talk at SCAN 2000, Karlsruhe, Germany, September 2000.

[26] L.B. Rall, *Automatic differentiation: techniques and applications*, Lecture Notes in Computer Science 120, Springer Verlag, Berlin, 1981.

[27] R. Rihm, On a class of enclosure methods for initial value problems, Computing, 53 (1994), 369–377.

[28] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-oriented modeling and design*, Prentice Hall, New York, 1991.

[29] H. Spreuer and E. Adams, On the existence and the verified determination of homoclinic and heteroclinic orbits of the origin for the Lorenz system, Computing Suppl., 9 (1993), 233–246.

[30] O. Stauning, Automatic validation of numerical solutions, PhD thesis, Technical University of Denmark, DK-2800, Lyngby, Denmark, October 1997.

[31] W. Tucker, A rigorous ODE solver and Smale's 14th problem, Found. Comput. Math. 2 (2002), pp. 53–117.