1

ODE Software That Computes Guaranteed Bounds on the Solution

N. S. Nedialkov[1]
K. R. Jackson[1]

ABSTRACT
Validated methods for initial value problems (IVPs) for ordinary differential equations (ODEs) produce bounds that are guaranteed to enclose the true solution of a problem. In this chapter, we briefly survey validated methods for IVPs for ODEs, discuss software issues related to the implementation of a validated ODE solver, and describe the structure of a package for computing rigorous bounds on the solution of an IVP for an ODE.

1.1 Introduction

Standard numerical methods aim at computing approximations that satisfy a user-specified tolerance. These methods are robust and reliable for most applications, but it is possible to find examples for which they return inaccurate results. However, 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 [SA93]. Also, some calculations may be critical to the safety or reliability of a system. 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 give the user an indication of the accuracy and reliability of the floating-point scheme.

Validated (also called interval) methods produce bounds that are guaranteed to contain a mathematically correct result. These methods use interval arithmetic [Moo66] to enclose rounding errors, truncation errors, and possibly even uncertainties in the model, assuming these can be quantified.

A key application of interval methods is to compute rigorous enclosures

[1]Department of Computer Science, University of Toronto, Toronto, Ontario, CANADA, M5S 3G4. Email: {ned,krj}@cs.toronto.edu.

for the range of a function. In addition to bounding errors in the computation, such enclosures can be used, for example, to prove the existence of a solution to a nonlinear system of equations and to compute interval vectors that are guaranteed to contain the true solution of the system [Neu90]. One of the most successful applications of interval computations is in global optimization where bounds on the range of a function are used to reject regions that cannot contain global optima [Kea95]. We can also obtain tight bounds on the true solution of a linear system of equations [KU93].

Interval techniques can also be used to bound the error of a numerical approximation. For example, as explained below, we can enclose the truncation error of a Taylor series expansion by generating Taylor coefficients in interval arithmetic. Such enclosures can be used to compute guaranteed bounds either for an integral [CR87] or for an initial value problem (IVP) for an ordinary differential equation (ODE) (see [NJC99] for a survey). An extensive summary of applications of and resources for interval computations can be found in [Kea96a].

Interval methods have not been popular in the past for several reasons. Chief amongst these are:

- a validated method frequently requires much more computational work and storage than an approximate method for the same problem;
- the bounds produced by many early validated methods were not tight;
- it is difficult to develop good, robust, portable software for validated methods in older languages such as Fortran or C.

However, many of these difficulties are not as serious now as they were several decades ago when interval methods were first proposed.

It remains true that a validated method may run several orders of magnitude slower than an approximate method for the same problem. (We hope to quantify the extent to which this is true in another paper.) Consequently, a validated method may not be appropriate for large computationally intensive problems or ones that are time critical, such as real-time control problems. However, when interval methods are criticized for being more expensive than traditional methods, we should remember 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 accuracy is important, it may be much more cost-effective to let the computer spend a few more minutes 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. Note that the reliability issue does not arise for validated methods, since, if implemented correctly, software based on a validated method always produces reliable results.

Also note that in some cases an interval computation is not more work than a standard floating-point calculation. For example, interval methods can be faster than standard floating-point schemes for finding all solutions or global optima of a nonlinear equation [Han92] (see [Kea96a] for references).

The bounds produced by many early interval methods were not tight. One reason is that a straightforward replacement of real operations by interval arithmetic operations in a point algorithm often leads to unacceptably wide bounds. For example, to bound the solution of a linear system, we can apply Gaussian elimination with floating-point numbers replaced by intervals. This, though, often produces large overestimations of the error. A good interval linear equation solver is significantly different from the corresponding floating-point algorithm [Neu90]. Similarly, early interval methods for ODEs suffered from the *wrapping effect*. As a result, they often produced very poor bounds. However, as described below, more sophisticated modern methods cope quite effectively with this difficulty and produce tight bounds for a wide range of problems.

Until recently, there has been a scarcity of interval-arithmetic software. In part, this is because it is difficult to write good, robust, portable interval-arithmetic software in older languages such as Fortran or C. This task is greatly facilitated by modern programming languages such as C++ and Fortran 90, which allow user-defined types and operator overloading. As a result, we can implement an interval data type and write interval arithmetic operations without explicit function calls. An important step towards wider use of interval computations is the development of a Fortran compiler that supports intervals as an intrinsic data type [SZAB98].

This chapter focuses on interval methods for IVPs for ODEs: we give a brief overview of these methods, discuss software issues related to the implementation of a validated ODE solver, and describe the structure of the VNODE (Validated Numerical ODE) package that we are developing.

Sections 1.2 to 1.4 present the mathematical background necessary for understanding the rest of this chapter. Section 1.2 discusses interval arithmetic operations and computing ranges of functions. Section 1.3 states in more detail the IVP that is a subject of this work. Section 1.4 reviews validated methods for computing guaranteed bounds on the solution of an IVP for an ODE. We consider one step of these methods, automatic generation of Taylor coefficients, validating existence and uniqueness of the solution, explain the wrapping effect, and describe Lohner's method for reducing it. We also discuss an Hermite-Obreschkoff scheme for computing bounds on the solution of an IVP for an ODE.

Sections 1.5 to 1.10 are devoted to software issues in validated ODE solving. First, we discuss the goals that we have set to achieve with the design of VNODE (§1.5), available software for computing validated solutions of IVPs for ODEs (§1.6), object-oriented concepts (§1.7), and the choice of C++ to implement VNODE (§1.8). Then, in §1.9, we describe the structure of VNODE and illustrate how it can be used. Section 1.10 discusses problems that we have encountered during the development of VNODE.

1.2 Interval Arithmetic and Ranges of Functions

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

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

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

$$[a] \circ [b] = \left\{ x \circ y \mid x \in [a], \ y \in [b] \right\}, \quad 0 \notin [b] \ \text{ when } \circ = / \tag{1.1}$$

[Moo66]. Definition (1.1) can be written in the following equivalent form (we omit $*$ in the notation):

$$[a] + [b] = \left[\underline{a} + \underline{b}, \bar{a} + \bar{b} \right], \tag{1.2}$$

$$[a] - [b] = \left[\underline{a} - \bar{b}, \bar{a} - \underline{b} \right], \tag{1.3}$$

$$[a]\,[b] = \left[\min\left\{ \underline{a}\underline{b}, \underline{a}\bar{b}, \bar{a}\underline{b}, \bar{a}\bar{b} \right\}, \ \max\left\{ \underline{a}\underline{b}, \underline{a}\bar{b}, \bar{a}\underline{b}, \bar{a}\bar{b} \right\} \right], \tag{1.4}$$

$$[a] / [b] = [\underline{a}, \bar{a}] \left[1/\bar{b}, 1/\underline{b} \right], \quad 0 \notin [b]. \tag{1.5}$$

The formulas[1] (1.2–1.5) allow us to express the interval arithmetic operations using the endpoints of the interval operands, thus obtaining 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 real intervals in (1.2–1.5) 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 chapter. Because of the outward roundings, intervals computed in machine interval arithmetic always contain the corresponding real intervals.

Definition (1.1) and formulas (1.2–1.5) can be extended to vectors and matrices. If their components are intervals, we obtain interval vectors and matrices. The arithmetic operations involving interval vectors and matrices

[1] Addition and subtraction are nearly always expressed as in (1.2) and (1.3), but different sources may give different formulas for multiplication and division.

are defined by the same formulas as in the scalar case, except that scalars are replaced by intervals.

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

$$[a] \circ [b] \subseteq [a_1] \circ [b_1] , \quad \circ \in \{+, -, *, /\} . \tag{1.6}$$

Using (1.6), we can *rigorously* enclose the range of a function.

Let $f : \mathbb{R}^n \to \mathbb{R}$ be a function on $\mathcal{D} \subseteq \mathbb{R}^n$. The interval-arithmetic evaluation of f on $[a] \subseteq \mathcal{D}$, which we denote by $f([a])$, is obtained by replacing each occurrence of a real variable with a corresponding interval, by replacing the standard functions with enclosures of their ranges, and by performing interval-arithmetic operations instead of the real operations. From (1.6), the range of f, $\{f(x) \mid x \in [a]\}$, is always contained in $f([a])$.

Although $f([a])$ is not unique, since expressions that are mathematically equivalent for scalars, such as $x(y + z)$ and $xy + xz$, may have different values if x, y, and z are intervals, a value for $f([a])$ can be determined from the code list, or computational graph, for f.

One reason for obtaining different values for $f([a])$ is that the distributive law does not hold in general in interval arithmetic [AH83, pp. 3–5]. 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. An important case when the distributive law does hold is when $[a]$ is a degenerate interval ($\underline{a} = \bar{a}$).

Addition of interval matrices is associative, but multiplication of interval matrices is not associative in general [Neu90, pp. 80–81]. Also, the distributive law does not hold in general for interval matrices [Neu90, p. 79].

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

$$f(y) \in f_M([a], b) = f(b) + f'([a])([a] - b) \tag{1.7}$$

[Moo66, p. 47]. The mean-value form, $f_M([a], b)$, 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. Like f, f_M is not uniquely defined, but a value for it can be determined from the code lists for f and f'.

1.3 The Initial Value Problem

We consider the set of autonomous IVPs

$$y'(t) = f(y) \tag{1.8}$$

$$y(t_0) \in [y_0], \tag{1.9}$$

where $t \in [t_0, T]$ for some $T > t_0$. Here $t_0, T \in \mathbb{R}$, $f \in C^{k-1}(\mathcal{D})$, $\mathcal{D} \subseteq \mathbb{R}^n$ is open, $f : \mathcal{D} \to \mathbb{R}^n$, and $[y_0] \subseteq \mathcal{D}$. The condition (1.9) 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 only 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. For expositional convenience, we consider only autonomous systems, but this is not a restriction of consequence since a nonautonomous system of ODEs can be converted into an autonomous system, and our methods can be extended easily to nonautonomous systems.

We consider a grid $t_0 < t_1 < \cdots < t_m = T$, which is not necessarily equally spaced, and denote the stepsize from t_j to t_{j+1} by h_j ($h_j = t_{j+1} - t_j$). The step from t_j to t_{j+1} is referred to as the $(j+1)$st step. We denote the solution of (1.8) with an initial condition y_j at t_j by $y(t; t_j, y_j)$. For an interval, or an interval vector in general, $[y_j]$, we denote by $y(t; t_j, [y_j])$ the set of solutions $\{y(t; t_j, y_j) \mid y_j \in [y_j]\}$.

Our goal is to compute intervals (or interval vectors) $[y_j]$, $j = 1, 2, \ldots, m$, that are guaranteed to contain the solution of (1.8–1.9) at $t_1, t_2, \ldots, t_m$. That is,

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

1.4 An Overview of Validated Methods for IVPs for ODEs

Most validated methods for IVPs for ODEs are based on Taylor series [Moo66], [Krü69], [Eij81], [Loh88], [Rih94], [BM98]. The Taylor series approach has been popular because of the simple form of the error term, which can be readily bounded using interval arithmetic. In addition, the Taylor series coefficients can be efficiently generated by automatic differentiation, the order of the method can be changed easily by adding or deleting Taylor series terms, and the stepsize can be changed without doing extra work to recompute Taylor series coefficients.

1.4.1 One Step of a Validated Method

The $(j+1)$st step ($j \geq 0$) in most Taylor series methods consists of two phases [NJC99]:

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

ALGORITHM II: Compute a tighter enclosure $[y_{j+1}]$ for $y(t_{j+1}; t_0, [y_0])$.

In this section, we briefly review methods for implementing these two algorithms. A detailed survey of Taylor series methods can be found in [NJC99].

1.4.2 Automatic Generation of Taylor Coefficients

Since the interval methods for IVPs for ODEs considered here use Taylor series, we outline the recursive generation of Taylor coefficients.

If we know the Taylor coefficients $u^{(i)}/i!$ and $v^{(i)}/i!$ for $i = 0, \ldots, p$ 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 [Moo66, pp. 107–130].

We introduce the sequence of functions

$$f^{[0]}(y) = y, \tag{1.10}$$

$$f^{[i]}(y) = \frac{1}{i}\left(\frac{\partial f^{[i-1]}}{\partial y} f\right)(y), \quad \text{for } i \geq 1. \tag{1.11}$$

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

$$(y_j)_i = f^{[i]}(y_j) = \frac{1}{i}\big(f(y_j)\big)_{i-1}, \quad i \geq 1, \tag{1.12}$$

where $\big(f(y_j)\big)_{i-1}$ is the $(i-1)$st coefficient of f evaluated at y_j. Using (1.12) 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 coefficients, we need at most $O(k^2)$ operations [Moo66, pp. 111–112]. Note that this is far more efficient than the standard symbolic generation of Taylor coefficients.

If we have a procedure to compute the point Taylor coefficients of $y(t)$ and perform the computations in interval arithmetic with $[y_j]$ instead of y_j, we obtain a procedure to compute the interval Taylor coefficients of $y(t)$.

1.4.3 Algorithm I: 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 and $[\tilde{y}_j^0] \supseteq [y_j]$ satisfy

$$[\tilde{y}_j] = [y_j] + [0, h_j]\, f\left([\tilde{y}_j^0]\right) \subseteq [\tilde{y}_j^0], \tag{1.13}$$

then (1.8) with $y(t_j) = y_j$ has a unique solution $y(t; t_j, y_j)$ that satisfies $y(t; t_j, y_j) \in [\tilde{y}_j]$ for all $t \in [t_j, t_{j+1}]$ and all $y_j \in [y_j]$, [Eij81, pp. 59–67].

We refer to a method based on (1.13) as a constant enclosure method. Such a method can be easily implemented, but a serious disadvantage of this approach is that the stepsize is restricted to Euler steps. Usually, Algorithm II allows larger stepsizes, but they are often limited by Algorithm I.

One can obtain methods that enable larger stepsizes by using polynomial enclosures [Loh95] or more Taylor series terms in the sum in (1.13), thus obtaining a Taylor series enclosure method [Moo66, pp. 100–103], [CR96], [Ned99].

1.4.4 Algorithm II: Computing a Tight Enclosure

Consider the Taylor series expansion

$$y_{j+1} = y_j + \sum_{i=1}^{k-1} h_j^i f^{[i]}(y_j) + h_j^k f^{[k]}(y; t_j, t_{j+1}), \tag{1.14}$$

where $y_j \in [y_j]$ and $f^{[k]}(y; t_j, t_{j+1})$ denotes $f^{[k]}$ with its lth component ($l = 1, \ldots, n$) evaluated at $y(\xi_{jl})$, for some $\xi_{jl} \in [t_j, t_{j+1}]$. Using the a priori bounds $[\tilde{y}_j]$ on $[t_j, t_{j+1}]$, we can enclose the local truncation error of the Taylor series (1.14) on $[t_j, t_{j+1}]$ for any $y_j \in [y_j]$ by $h_j^k f^{[k]}([\tilde{y}_j])$. We can also bound with $f^{[i]}([y_j])$ the ith Taylor coefficient at t_j. Therefore,

$$[y_{j+1}] = [y_j] + \sum_{i=1}^{k-1} h_j^i f^{[i]}([y_j]) + h_j^k f^{[k]}([\tilde{y}_j]) \tag{1.15}$$

contains $y(t_{j+1}; t_0, [y_0])$. However, as explained below, (1.15) illustrates how replacing real numbers by intervals often leads to large overestimations.

Denote by $w([a]) = \bar{a} - \underline{a}$ the width of an interval $[a]$; the width of an interval vector or interval matrix is defined componentwise. Since $w([a] + [b]) = w([a]) + w([b])$,

$$w([y_{j+1}]) = w([y_j]) + \sum_{i=1}^{k-1} h_j^i w(f^{[i]}([y_j])) + h_j^k w(f^{[k]}([\tilde{y}_j])) \geq w([y_j]),$$

which implies that the width of $[y_j]$ almost always increases[2] with j, even if the true solution contracts.

A better approach is to apply the mean-value theorem to $f^{[i]}$, $i =$

[2] An equality is possible only in the trivial cases $h_j = 0$ or $w(f^{[i]}([y_j])) = 0$, $i = 1, \ldots k-1$, and $w(f^{[k]}([\tilde{y}_j])) = 0$.

$1, \ldots, (k-1)$, in (1.14), obtaining that, for any $\hat{y}_j \in [y_j]$,

$$\begin{aligned} y_{j+1} = \hat{y}_j + \sum_{i=1}^{k-1} h_j^i f^{[i]}(\hat{y}_j) + h_j^k f^{[k]}(y; t_j, t_{j+1}) \\ + \left\{ I + \sum_{i=1}^{k-1} h_j^i J\left(f^{[i]}; y_j, \hat{y}_j\right) \right\} (y_j - \hat{y}_j), \end{aligned} \tag{1.16}$$

where $J\left(f^{[i]}; y_j, \hat{y}_j\right)$ is the Jacobian of $f^{[i]}$ with its lth row evaluated at $y_j + \theta_{il}(\hat{y}_j - y_j)$ for some $\theta_{il} \in [0,1]$ $(l = 1, \ldots, n)$. This formula is the basis of the interval Taylor series (ITS) methods of Moore [Moo66], Eijgenraam [Eij81], Lohner [Loh88], and Rihm [Rih94] (see also [NJC99]).

Let

$$[S_j] = I + \sum_{i=1}^{k-1} h_j^i J\left(f^{[i]}; [y_j]\right) \quad \text{and} \quad [z_{j+1}] = h_j^k f^{[k]}([\tilde{y}_j]). \tag{1.17}$$

Using the notation (1.17), we can bound (1.16) in interval arithmetic by

$$[y_{j+1}] = \hat{y}_j + \sum_{i=0}^{k-1} h_j^i f^{[i]}(\hat{y}_j) + [z_{j+1}] + [S_j]([y_j] - \hat{y}_j). \tag{1.18}$$

The ith Jacobian $(i = 1, \ldots, k-1)$ in (1.17) can be computed by generating the Taylor coefficient $f^{[i]}([y_j])$ and then differentiating it [BS96], [BS97]. Alternatively, these Jacobians can be computed by generating Taylor coefficients for the associated variational equation [Loh88].

If we compute enclosures with (1.18), the widths of the computed intervals may decrease. However, this approach may work poorly in some cases, because the interval vector $[S_j]([y_j] - \hat{y}_j)$ may significantly overestimate the set $\left\{S_j(y_j - \hat{y}_j) \mid S_j \in [S_j],\ y_j \in [y_j]\right\}$. This is called the wrapping effect.

The Wrapping Effect

The wrapping effect is clearly illustrated by Moore's example [Moo66],

$$\begin{aligned} y_1' &= y_2 \\ y_2' &= -y_1, \\ y(0) &\in [y_0]. \end{aligned} \tag{1.19}$$

The interval vector $[y_0]$ can be viewed as a rectangle in the (y_1, y_2) plane. At $t_1 > 0$, the true solution of (1.19) is the rotated rectangle shown in Figure 1.1. If we want to enclose this rectangle in an interval vector, we have to wrap it by another rectangle with sides parallel to the y_1 and y_2 axes. On the next step, this larger rectangle is rotated and so must be enclosed in a still larger one. Thus, at each step, the enclosing rectangles become larger and larger, but the solution set remains a rectangle of the same size. Moore showed that at $t = 2\pi$, the interval inclusion is inflated by a factor of $e^{2\pi} \approx 535$ as the stepsize approaches zero [Moo66, p. 134].

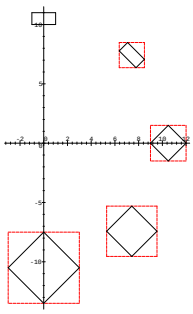

FIGURE 1.1. The rotated rectangle is wrapped at $t = \frac{\pi}{4}n$, $n = 1, 2, 3, 4$.

Lohner's Method

Here, we describe Lohner's QR-factorization method [Loh87], which is one of the most successful, general-purpose methods for reducing the wrapping effect.

Denote by $m([a]) = (\bar{a}+\underline{a})/2$ the midpoint of an interval $[a]$; the midpoint of an interval vector or interval matrix is defined componentwise. Let

$$A_0 = I, \;\; \hat{y}_0 = m([y_0]), \;\; [r_0] = [y_0] - \hat{y}_0, \quad \text{and} \tag{1.20}$$

$$\hat{y}_j = \hat{y}_{j-1} + \sum_{i=1}^{k-1} h_{j-1}^i f^{[i]}(\hat{y}_{j-1}) + m([z_j]) \quad (j \geq 1) \tag{1.21}$$

(I is the identity matrix). From (1.16–1.17) and (1.20–1.21), we compute

$$[y_{j+1}] = \hat{y}_j + \sum_{i=1}^{k-1} h_j^i f^{[i]}(\hat{y}_j) + [z_{j+1}] + \big([S_j]\, A_j\big)[r_j] \tag{1.22}$$

and propagate for the next step the vector

$$[r_{j+1}] = \big(A_{j+1}^{-1}([S_j]\, A_j)\big)[r_j] + A_{j+1}^{-1}\big([z_{j+1}] - m([z_{j+1}])\big). \tag{1.23}$$

Here $A_j \in \mathbb{R}^{n\times n}$ is a nonsingular matrix. If $A_{j+1} \in [S_j]\, A_j$, $w([S_j])$ is small, and A_j is well conditioned, then $A_{j+1}^{-1}([S_j]\, A_j)$ is close to the identity matrix. Since we can also keep $w([z_{j+1}])$ small, there is not a large overestimation in forming $[r_{j+1}]$. However, this choice of A_{j+1} does not guarantee that it is well conditioned or even nonsingular. In fact, A_{j+1} may be ill conditioned, and a large overestimation may arise in this evaluation. Usually, a good choice for A_{j+1} is the Q-factor from the QR-factorization of $\widehat{A}_{j+1} = m([S_j]A_j)$. As a result, the condition number of A_{j+1} is small and

the wrapping effect is reduced (on each step) by computing an enclosure in an orthogonal coordinate induced by this Q-factor. A detailed explanation of the reasons for this and other choices for A_{j+1} can be found in [Loh87] and [NJC99].

We should note that the wrapping effect does not occur in some problems, for example, in quasi-monotone systems (see [NJC99]) or one-dimensional problems. Also, for a particular problem, one anti-wrapping strategy may be more appropriate than another. Evaluating such strategies is not trivial and will not be discussed in this chapter.

An Interval-Hermite Obreschkoff Method

For more than 30 years, Taylor series has been the only effective approach for computing rigorous bounds on the solution of an IVP for an ODE. Recently, we developed a new scheme, an interval Hermite-Obreschkoff (IHO) method [NJ98], [Ned99]. Here, we outline the method and its potential.

Let

$$c_i^{q,p} = \frac{q!}{(p+q)!}\frac{(q+p-i)!}{(q-i)!}$$

(q, p, and $i \geq 0$), $y_j = y(t_j; t_0, y_0)$, and $y_{j+1} = y(t_{j+1}; t_0, y_0)$. It can be shown that

$$\sum_{i=0}^{q}(-1)^i c_i^{q,p} h_j^i f^{[i]}(y_{j+1}) = \sum_{i=0}^{p} c_i^{p,q} h_j^i f^{[i]}(y_j) + (-1)^q \frac{q!p!}{(p+q)!} h_j^{p+q+1} f^{[k]}(y; t_j, t_{j+1}) \tag{1.24}$$

[Wan77], [Ned99]. This formula is the basis of our IHO method. The method we propose in [Ned99] consists of two phases, which can be considered as a predictor and a corrector. The predictor computes an enclosure $[y_{j+1}^{(0)}]$ of the solution at t_{j+1}, and using this enclosure, the corrector computes a tighter enclosure $[y_{j+1}] \subseteq [y_{j+1}^{(0)}]$ at t_{j+1}. Since $q > 0$, (1.24) is an implicit scheme. The corrector applies a Newton-like step to tighten $[y_{j+1}^{(0)}]$.

We have shown in [NJ98] and [Ned99] that for the same order and stepsize, our IHO method has smaller local error, 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.

1.5 Objectives

Our primary goal is to provide a software package that will assist researchers in the numerical study and comparison of schemes and heuristics used in computing validated solutions of IVPs for ODEs. The VNODE

(Validated Numerical ODE) package that we are developing is intended to be *a uniform implementation of a generic validated solver for IVPs for ODEs.* Uniform means that the design and implementation of VNODE follow well-defined patterns. As a result, implementing, modifying, and using methods can be done systematically. Generic means that the user can construct solvers by choosing appropriate methods from sets of methods. This property enables us to isolate and compare methods implementing the same part of a solver. For example, we can assemble two solvers that differ only in the module 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). Since we would like to investigate algorithms, being able to isolate them is an important feature of such an environment.

We list and briefly explain some of the goals we have tried to achieve with the design of VNODE. Provided that a validated method for IVPs for ODEs is implemented correctly, the reliability issue does not arise: if a validated solver returns an enclosure of the solution, then the solution is guaranteed to exist within the computed bounds.

Modularity The solver should be organized as a set of modules with well-defined interfaces. The implementation of each module should be hidden, but if necessary, the user should be able to modify the implementation.

Flexibility Since we require well-defined interfaces, we should be able to replace a method, inside a solver, without affecting the rest of the solver. Furthermore, we should be able to add methods following the established structure, without modifying the existing code.

Efficiency The methods incorporated in VNODE do not have theoretical limits on the size of the ODE system or order of a method. However, these methods require the computation of high-order Taylor coefficients and Jacobians of Taylor coefficients. As a result, the efficiency of a validated solver is determined mainly by the efficiency of the underlying automatic differentiation package. Other factors that contribute to the performance are: the efficiency of the interval-arithmetic package, the programming language, and the actual implementation of the methods. One difficulty is that to achieve flexibility, we may need to repeat the same calculations in two parts of a solver. For example, to separate Algorithm I and Algorithm II, we may need to generate the same Taylor coefficients in both algorithms. However, the repetition of such computations should be avoided, where possible.

Since VNODE is to be used for comparing and assessing methods, it should

contain the most promising ones. Moreover, VNODE should support rapid prototyping.

1.6 Background

The area of computing validated solutions of IVPs for ODEs is not as developed as the area of computing approximate solutions [NJC99]. For example, we still do not have well-studied stepsize and order control strategies nor methods suitable for stiff problems. Also, reducing the wrapping effect in Algorithm II and taking larger stepsizes in Algorithm I are still open problems in validated ODE solving. With respect to the tools involved, a validated solver is inherently more complex than a classical ODE solver. In addition to an interval-arithmetic package, a major component of a validated solver is the module for automatic generation of interval Taylor coefficients (see §1.8.1).

Currently, there are three available packages for computing guaranteed bounds on the solution of an IVP for an ODE: AWA [Loh88], ADIODES [Sta97] and COSY INFINITY [Ber97]. We briefly summarize each in turn.

AWA is an implementation of Lohner's method (§1.4.4) and the constant enclosure method (§1.4.3) for validating existence and uniqueness of the solution. This package is written in Pascal-XSC [KKN+92], an extension of Pascal for scientific computing.

ADIODES is a C++ implementation of a solver using the constant enclosure method in Algorithm I and Lohner's method in Algorithm II. The stepsize in both ADIODES and AWA is often restricted to Euler steps by Algorithm I.

COSY INFINITY is a Fortran-based code to study and to 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 (see [BM98]). 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. Thus, the arithmetic operations and standard functions are executed with such Taylor polynomials as operands. Although the approach described in [BM98] reduces the wrapping effect substantially, working with polynomials is significantly more expensive than working with intervals.

1.7 Object-Oriented Concepts

We have chosen an object-oriented approach in designing VNODE and C++ [ES90] to implement it. This is not the first object-oriented design

of an ODE solver. The Godess [Ols97] and TIDE [Hoh93b] (a successor of CODE++ [Hoh93a]) packages offer generic ODE solvers that implements traditional methods for IVPs for ODEs. Another successful package is Diffpack [Lan99], which is devised for solving partial differential equations. In [Lan99], there is also an example of how to construct an object-oriented ODE solver.

In this section, we briefly review some object-oriented concepts supported in C++. We use them in the discussion of the choice of programming language (§1.8) and in the description of VNODE (§1.9). A reader familiar with object-oriented terms can skip this section (§1.7).

A good discussion of object-oriented concepts, analysis, and design can be found in [Boo94]. An excellent book on advanced C++ styles and idioms is [Cop92]. A study of nonprocedural paradigms for numerical analysis, including object-oriented ones, is presented in [SZ95].

Data Abstraction

In the object model, a software system can be viewed as a collection of objects that interact with each other to achieve a desired functionality. An object is an instance of a class, which defines the structure and behavior of its objects. By grouping data and methods inside a class and specifying its interface, we achieve encapsulation, separating the interface from the implementation. Hence, the user can change the data representation and the implementation of a method[3] (or methods) of a class without modifying the software that uses it. By encapsulating data, we can avoid function calls with long parameter lists, which are intrinsic to procedural languages like Fortran 77. A class can encapsulate data or algorithms, or both.

Inheritance and Polymorphism

Inheritance and polymorphism are powerful features of object-oriented languages. Inheritance allows code reuse: a derived class can reuse data and functions of its base class(es). The role of polymorphism is to apply a given function to different types of objects. Often polymorphism and inheritance are used with abstract classes. An abstract class defines abstract functions, which are implemented in its subclasses; it has no instances and an object of such a class cannot be created.

Operator Overloading

Operator overloading allows the operators of the language to be overloaded for user defined types. To program interval operations without explicit function calls, we have to use a language that supports operator overloading.

[3] We use *method* in two different contexts: to denote a member function of a class or a method in VNODE.

Without it, programming interval-arithmetic expressions is cumbersome, because we have to code interval-arithmetic operations by using function calls. Both C++ and Fortran 90 provide operator overloading. This feature is used to build interval-arithmetic libraries like PROFIL/BIAS [Knü94] (C++) and INTLIB (Fortran 90) [Kea96b], as discussed in §1.8.2.

1.8 Choice of Language: C++ versus Fortran 90

We considered C++ and Fortran 90 [MR90] as candidates to implement VNODE, but we preferred C++. Procedural languages like C or Fortran 77 can be used to implement an object-oriented design [BGMS97]. However, using a language that supports object-oriented programming usually reduces the effort for implementing object-oriented software.

Our choice was determined by the following considerations, listed in order of importance:

1. availability of software for automatic generation of interval Taylor coefficients;
2. performance and built-in functions of the available interval-arithmetic packages;
3. support of object-oriented concepts; and
4. efficiency.

In this section, we address each of these considerations in turn.

1.8.1 Software for Automatic Generation of Interval Taylor Coefficients

Although packages for automatic differentiation (AD) are available (see for example [Jue91] and [YC96]), to date, only two free packages for automatic generation of interval Taylor coefficients for the solution of an ODE and the Jacobians of these coefficients are known to the authors. These are the FADBAD/TADIFF [BS96], [BS97] and IADOL-C [Iwa97] packages. They are written in C++ and implement AD through operator overloading.

TADIFF and FADBAD are two different packages. TADIFF can generate Taylor coefficients with respect to time. Then, FADBAD can be used to compute Jacobians of Taylor coefficients by applying the forward mode of AD [Ral81] to these coefficients. FADBAD and TADIFF are not optimized to handle large and sparse systems. Also, they perform all the work in main memory.

The IADOL-C package is an extension of ADOL-C [GJU96] that allows generic data types. ADOL-C can compute Taylor coefficients by using the

forward mode and their Jacobians by applying the reverse mode [Spe80] to these coefficients. The basic data type of ADOL-C is double. To use a new data type in IADOL-C, the user has to overload the arithmetic and comparison operations and the standard functions for that data type. Then, using IADOL-C is essentially the same as using ADOL-C. Since IADOL-C replaces only the double data type of ADOL-C, IADOL-C inherits all the functionality of ADOL-C. However, this overloading in IADOL-C causes it to run about three times slower than ADOL-C. This appears to be due largely to the C++ compilers rather than the AD package [Iwa97].

The ADOL-C package records the computation graph on a so-called tape. This tape is stored in the main memory, but, when necessary, is paged to disk. When generating Jacobians of Taylor coefficients, ADOL-C can exploit the sparsity structure of the Jacobian of the function for computing the right side. Since optimization techniques are used in ADOL-C, we expect the interval version, IADOL-C, to perform better than FADBAD/TADIFF on large and complex problems. On the other hand, FADBAD/TADIFF should perform well on small to medium-sized problems.

Currently, VNODE is configured with FADBAD/TADIFF, but we have also used IADOL-C. VNODE with these AD packages is based on the INTERVAL data type from the PROFIL/BIAS package, which we discuss in §1.8.2 and §1.8.4.

1.8.2 Interval Arithmetic Packages

The most popular and free interval-arithmetic packages are PROFIL/BIAS [Knü94], written in C++, and INTLIB [KDDH95], written in Fortran 77 and available with a Fortran 90 interface [Kea96b]. The Fortran 90 version of INTLIB uses operator overloading. For references and comments on other available packages, see for example [Kea96b] or [Knü94]. Recently, an extension of the Gnu Fortran compiler which supports intervals as an intrinsic data type was reported [SZAB98].

PROFIL/BIAS seems to be the fastest interval package. In comparison with other such packages, including INTLIB, PROFIL/BIAS is about one order of magnitude faster [Knü94]. For efficiency, it uses directional rounding facilities of the processor on the machines on which it is installed. Portability is provided by isolating the machine dependent code in small assembler files, which are distributed with the package. Another reason for the efficiency of PROFIL/BIAS is that the number of rounding mode switches and sign tests are minimized in vector and matrix operations. In modern RISC architectures, sign tests and rounding mode switches cost nearly as much or even more than floating-point operations [Knü94], [SZAB98]. In addition, PROFIL/BIAS is easy to use, and provides matrix and vector operations and essential routines, such as guaranteed linear equation solvers and optimization methods.

INTLIB is intended to be a totally portable Fortran 77 package. It does

not assume IEEE arithmetic and does not assume anything about the accuracy of the standard functions of the underlying Fortran compiler. For portability, directed roundings are simulated by multiplying the computed results by $(1+\epsilon)$, for an appropriate epsilon. Significant additional execution time is required for these multiplications.

1.8.3 Support of Object-Oriented Concepts

C++ is a fully object-oriented language, while Fortran 90 is not, because it does not support inheritance and polymorphism. The features of C++ (e.g., data abstraction, operator overloading, inheritance, and polymorphism) allow the goals in §1.5 to be achieved in a relatively simple way. Inheritance and polymorphism can be simulated in Fortran 90 [DNS97], but this is cumbersome.

1.8.4 Efficiency

Compared to Fortran, C++ has been criticized for its poor performance for scientific computing. Here, we discuss an important performance problem: the pairwise evaluation of arithmetic expression with arguments of array types (e.g., matrices and vectors). More detailed treatment of this and other problems can be found in [Rob96], [Vel97], and [VJ97].

In C++, executing overloaded arithmetic operations with array data types creates temporaries, which can introduce a significant overhead, particularly for small objects. For example, if `A, B, C`, and `D` are vectors, the evaluation of the expression

```
D = A + B + C
```

creates two temporaries: one to hold the result of `A + B`, and another to hold the result of `(A + B) + C`. Furthermore, this execution introduces three loops. Clearly, it would be better to compute this sum in one loop without temporaries. In Fortran 90, mathematical arrays are represented as elementary types and optimization is possible at the compiler level.

Because of better optimizing compilers and template techniques [Vel95], [VJ97], C++ is becoming more competitive for scientific computing. A good technique for reducing the overhead in the pairwise evaluation of expressions involving arrays is to use expression templates [Vel95]. The expression template technique is based on performing compile-time transformation of the code using templates. With this technique, expressions containing vectors and matrices can be computed in a single pass without allocating temporaries. For example, with expression templates, it is possible to achieve a loop fusion [Vel95], allowing the above sum to be evaluated in a single loop:

```
for ( int i = 1; i <= N; i++ )
      D(i) = A(i) + B(i) + C(i);
```

However, executing this loop in interval arithmetic may not be the best solution for the following reason. Each interval addition in this loop involves two changes of the rounding mode. As we pointed out, the approach of PROFIL/BIAS is to minimize the number of rounding mode switches. Suppose that we want to compute in PROFIL/BIAS

```
C = A + B,
```

where `A`, `B`, and `C` are vectors of the same dimensions. If we denote the components of `A, B`, and `C` by a_i, b_i, and c_i, respectively, PROFIL/BIAS changes the rounding mode downwards and computes $\underline{c}_i = \underline{a}_i + \underline{b}_i$, for $i = 1, 2, \ldots, n$. Then, this package changes the rounding mode upwards and computes $\bar{c}_i = \bar{a}_i + \bar{b}_i$, for $i = 1, 2, \ldots n$. Thus, the result of `A + B` is computed with two rounding mode switches. However, PROFIL/BIAS still creates temporaries.

1.9 The VNODE package

1.9.1 Structure

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 method, or a solver, as an object containing the necessary data and functions to perform the integration of a given problem. Then, we can compute a solution by "applying" a method object to a problem object. Most functions in VNODE have such objects as parameters. The description of the numerical problem and the methods in VNODE are implemented as classes in C++.

The problem classes are shown in Figure 1.2, and the method classes are shown in Figure 1.3. A box in Figures 1.2 and 1.3 denotes a class; the rounded, filled boxes denote abstract classes. Each of them declares one or more virtual functions, which are not defined in the corresponding abstract class, but must be defined in the derived classes.

The lines with $\triangle$ indicate an *is-a* relationship, which here denotes inheritance or a specialization of a base class; the lines with $\Diamond$ indicate a *has-a* relationship. It is realized either by a complete containment of an object `Y` within another object `X` or by a pointer from `X` to `Y`. The notation in these figures is similar to that suggested in [RBP+91].

In the next two subsections, we list the problem and method classes and provide brief explanations. Here, we do not discuss the classes for generating Taylor coefficients in VNODE. A description of VNODE will be given in the documentation of the code at `http://www.cs.toronto.edu/NA`.

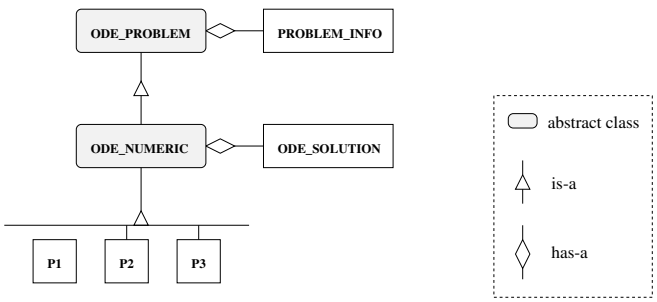

FIGURE 1.2. Problem classes.

Problem Classes

Class `ODE_PROBLEM` specifies the mathematical problem, that is, t_0, $[y_0]$, T, and a pointer to a function to compute the right side of the ODE. It also contains a pointer to a class `PROBLEM_INFO`. It indicates, for example, if the problem is constant coefficient, scalar, has a closed form solution, or has a point initial condition. Such information is useful since the solver can determine from it which part of the code to execute.

`ODE_NUMERIC` specifies the numerical problem. This class contains data such as absolute and relative error tolerances,[4] and a pointer to a class `ODE_NUMERIC` representing a solution. The user-defined problems, `P1`, `P2`, and `P3` in Figure 1.2 are derived from this class. New problems can be added by deriving them from `ODE_NUMERIC`.

`ODE_SOLUTION` contains the last obtained a priori and tight enclosures of the solution and the value of t where the tight enclosure is computed. `ODE_SOLUTION` contains also a pointer to a file that stores information from the preceding steps (e.g., enclosures of the solution and stepsizes).

Method Classes

Class `ODE_SOLVER` is a general description of a method that "solves" an `ODE_NUMERIC` problem. The `ODE_SOLVER` class declares the pure virtual function `Integrate`, whose definition is not provided in it. As a result, instances of `ODE_SOLVER` cannot be created. This class also contains class `METHOD_CONTROL`, which includes different flags (encapsulated in `FLAGS`) and statistics collected during the integration (encapsulated in `STATISTICS`).

Class `VODE_SOLVER` implements a validated solver by defining the function `Integrate`. We have divided this solver into four abstract methods: for selecting an order, selecting a stepsize, and computing initial and tight enclosures of the solution. These methods are realized by the abstract classes `ORDER_CONTROL`, `STEP_CONTROL`, `INIT_ENCL`, and `TIGHT_ENCL`, respectively.

[4]How to specify and interpret error tolerances will be discussed in the documentation of VNODE.

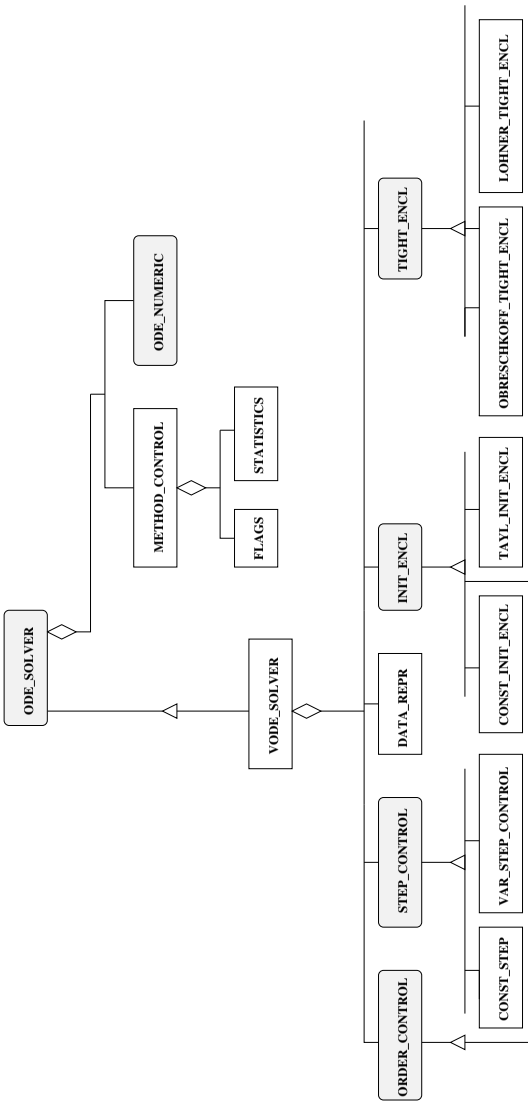

FIGURE 1.3. Method classes.

Their purpose is to provide general interfaces to particular methods. A new method can be added by deriving it from these abstract classes. `Integrate` performs an integration by calling objects that are instances of classes derived from `ORDER_CONTROL`, `STEP_CONTROL`, `INIT_ENCL`, and `TIGHT_ENCL`. We have used polymorphism in implementing this function: it calls virtual functions that are declared, but not defined, in these abstract classes. During execution, depending on the type (class) of the object, a particular function will be called (see §1.9.2 for an example).

Currently, VNODE does not implement variable-order methods, and the class `ORDER_CONTROL` has only one derived class, `CONST_ORDER`, whose role is to return a constant order. For selecting a stepsize, `CONST_STEP` returns a constant stepsize on each step, and `VAR_STEP_CONTROL` implements a stepsize selection scheme based on controlling the width of the remainder term [Ned99].

There are two methods for validating existence and uniqueness of the solution in VNODE: a constant enclosure method (`CONST_INIT_ENCL`) and a Taylor series method (`TAYL_INIT_ENCL`). The purpose of `FIXED_INIT_ENCL` is to compute a priori enclosures of the solution from the formula for the true solution, if the problem has a closed form solution. This class turns out to be helpful when we want to isolate the influence of Algorithm I, because it often reduces the input stepsize.

There are two methods for computing a tight enclosure of the solution: an interval Hermite-Obreschkoff method (`OBRESCHKOFF_TIGHT_ENCL`) and Lohner's method (`LOHNER_TIGHT_ENCL`). They both use Lohner's QR-factorization technique for reducing the wrapping effect. The `VODE_SOLVER` class also has a pointer to `DATA_REPR`, which is responsible for generating and storing Taylor coefficients and their Jacobians.

As we briefly discussed in §1.6, interval methods for IVPs for ODEs are not as developed as point methods for IVPs for ODEs. As a result, the number of validated ODE methods is significantly smaller than the number of standard ones. Thus, we have fewer classes in VNODE than in Godess [Ols97] or TIDE [Hoh93b], for example.

Finally, we should emphasize again that the methods implemented in VNODE do not have restrictions on the number of equations or the number of Taylor coefficients that can be generated.

1.9.2 An Example Illustrating the Use of VNODE

Suppose that we want to compare two solvers that differ only in the method implementing Algorithm II. In addition, we want to compare them with a constant enclosure method and then with a Taylor series enclosure method in Algorithm I. Here, we show and discuss part of the VNODE code that can be employed for this study. For simplicity of presentation, we use Van der Pol's equation, written as a first-order system,

$$\begin{aligned} y_1' &= y_2 \\ y_2' &= \mu(1-y_1^2)y_2 - y_1. \end{aligned} \tag{1.25}$$

where $\mu \in \mathbb{R}$. In a traditional ODE solver, we provide a function for computing the right side. In a validated solver, we also have to provide functions for generating Taylor coefficients and their Jacobians. Since we use an AD package for generating such coefficients, we have to specify a function for computing the right side of (1.25) for this package. We write the template function

```
template <class YTYPE> void VDPtemplate(YTYPE *yp, const YTYPE *y)
{
   yp[0] =  y[1];
   yp[1] =  MU*(1-sqr(y[0]))*y[1] - y[0];
}
```

which is used by FADBAD/TADIFF and IADOL-C to store the computation graph, and by VNODE to create a function for computing the right side. Then we derive a class `VDP` from `ODE_NUMERIC`. Since the details about the declaration of `VDP` are not essential to understand our example, we omit this declaration.

Figure 1.4 shows a typical use of VNODE classes. First, we create an `ODE_NUMERIC` object,[5] `VDP`, and load the initial condition, the interval of integration, and tolerance by calling the function `LoadProblemParam` (Part A). For testing, it is convenient to have a function that supplies different sets of data depending on the parameter to this function.

Then, we create methods and return pointers to them (Part B), as described below. `ITS` and `IHO` are pointers to objects for computing enclosures using Lohner's and the IHO methods, respectively. `InitEncl` is a pointer to an object for validating existence and uniqueness of the solution with the constant enclosure method; `StepControl` refers to an object that implements a variable stepsize control; and `OrderControl` points to an object that provides a constant value for the order.

The purpose of class `TAYLOR_EXPANSION` is to generate and store Taylor coefficients and their Jacobians. It is a template class, for which instances are created by specifying a class for generating Taylor coefficients and a class for generating Jacobians of Taylor coefficients. Here, we create such an instance with parameters `VDPTaylGenODE` and `VDPTaylGenVar`, which are classes[6] for generating Taylor coefficients and their Jacobians for (1.25).

In part C, we create two solvers, `SolverITS` and `SolverIHO` and integrate the problem by calling the `Integrate` function on these solvers.[7] Note that they differ only in the method for computing a tight enclosure of the

[5] `Ptr` stands for pointer in Figure 1.4.

[6] We do not describe these classes here.

[7] We omit the details about extracting data after an integration.

```
// ...
// A. Create the ODE problem.
PtrODENumeric ODE = new VDP;
ODE->LoadProblemParam(1);

// B. Create the methods.
int K, P, Q;
K = 11;            // order
P = Q = (K-1)/2;

PtrTightEncl ITS = new LOHNER_TIGHT_ENCL(K);
PtrTightEncl IHO = new OBRESCHKOFF_TIGHT_ENCL(P,Q);

PtrInitEncl  InitEncl  = new CONST_INIT_ENCL(ODE->Size, new VDPTaylGenVAR);
PtrStepCtrl  StepCtrl  = new VAR_STEP_CONTROL(ODE->Size);
PtrOrderCtrl OrderCtrl = new CONST_ORDER(K);
PtrDataRepr  DataRepr  = new TAYLOR_EXPANSION<VDPTaylGenODE,VDPTaylGenVAR>;

// Part C. Create the solvers and integrate.
PtrVODESolver SolverITS = new
   VODE_SOLVER(ODE, DataRepr, OrderCtrl, StepCtrl, InitEncl, ITS);

PtrVODESolver SolverIHO = new
   VODE_SOLVER(ODE, DataRepr, OrderCtrl, StepCtrl, InitEncl, IHO);

SolverITS->Integrate();
SolverIHO->Integrate();

// Part D. Replace the method implementing Algorithm I and integrate.
InitEncl =
   new TAYL_INIT_ENCL(ODE->Size, new VDPTaylGenODE, new VDPTaylGenVAR);

SolverITS->SetInitEncl(InitEncl);
SolverIHO->SetInitEncl(InitEncl);

SolverITS->Integrate();
SolverIHO->Integrate();
// ...
```

FIGURE 1.4. The test code.

solution. Thus, we can isolate and compare the two methods implementing Algorithm II.

Now, in part D, we want to replace the constant enclosure method for validating existence and uniqueness of the solution with a Taylor series method and repeat the same integrations. We create an instance of `TAYL_INIT_ENCL` by

```
InitEncl =
     new TAYL_INIT_ENCL(ODE->Size,new VDPTaylGenODE,new VDPTaylGenVAR);
```

set it by calling the `SetInitEncl` function, and integrate.

We explain how class `INIT_ENCL` works; the same idea is used in the other abstract classes. `INIT_ENCL` is an abstract class since it contains the pure virtual function

```
virtual void Validate( ... ) = 0;
```

(for simplicity, we leave out the parameters). Each of the derived classes of `INIT_ENCL` must declare a function with the same name and parameters and specify the body of the function. In `Integrate`, there is a call to `Validate`. During execution, depending on the object set, the appropriate `Validate` function will be called. We use polymorphism here: the function that is called is determined by the type of object during the execution of the program. In our example, the method for validating existence and uniqueness is replaced, but the integrator function is not changed. A user who wants to implement a new Algorithm I has to define a derived class of `INIT_ENCL` and an associated `Validate` function.

1.10 Notes from Our Experience

Here, we present some general notes and then discuss exception handling issues and wrapping effect in generating interval Taylor coefficients.

The most difficult part in designing VNODE was to choose a structure that would allow us to implement existing schemes efficiently while providing the flexibility to allow us to add other methods later without requiring an extensive modification of existing classes. This was complicated by the lack of a broad range of validated methods for ODEs to test our proposed structure. Although our design is appropriate for the few promising existing validated methods and for the extensions of standard methods to validated methods that we currently envision, we cannot, of course, guarantee that the VNODE structure will accommodate all novel schemes that may be developed in the future. However, we have made several extensions to VNODE already and plan several more soon. These are all easy to fit within the existing structure of VNODE.

As noted earlier, VNODE does not support variable-order methods yet. There are now different classes for order and stepsize control, but it may

be more convenient to merge them into one class. Future experience will show if we should keep them separate.

In earlier versions of VNODE, we created new problems by instantiating the `ODE_NUMERIC` class. Now, for a particular problem, we derive a class from it. Although the current version is "more object-oriented", creating problems by instantiating the `ODE_NUMERIC` class seems simpler than deriving problem classes.

We decided to implement classes for generating Taylor coefficients, so that when a new package is implemented, the user can easily substitute the new one for an old one. Given a function for computing the right side of an ODE, a package for generating Taylor coefficients should provide (at least) the following operations: *generate* coefficients (and store them), *read* coefficients, *sum* coefficients, and *scale* coefficients. These four "basic" operations are sufficient for building a validated ODE solver based on the theory discussed in this chapter.

1.10.1 Exception Handling

We discuss two situations, in the context of PROFIL/BIAS, in which a flexible exception handling mechanism of the interval-arithmetic package would facilitate the development of validated ODE software and, we believe, interval software in general.

PROFIL/BIAS clears the exception flags of the processor when setting the rounding mode. With respect to division by an interval containing zero, this package can be compiled such that it interrupts (default mode) or continues the execution, if such a division occurs. (The user cannot switch between these two modes after PROFIL/BIAS has been compiled.) Although terminating program execution when division by zero occurs is a reasonable action in many applications, in the following situation, it is desirable to raise a flag and continue the execution.

When validating existence and uniqueness of the solution, we normally guess a priori bounds for the solution. Then, we verify whether these bounds satisfy some condition (as in (1.4.3); see also [Ned99]). It may happen that we have guessed bounds that are too wide and contain zero. If there is a division by such a bound, we want to recognize division by zero without interrupting the execution, so we can try with a priori bounds that do not contain zero.

PROFIL/BIAS continues the execution if there is a NaN or an infinity. In some circumstances, as in the example described below, we prefer to interrupt the computations immediately if a NaN occurs.

After computing tight bounds for the solution, we intersect the tight and a priori enclosures: the true solution must be contained in the intersection. The `Intersect` function from PROFIL/BIAS returns true or false, depending on whether two intervals intersect. If the implementation is correct, this function must return true when computing an intersection of tight

and a priori bounds. Even when the algorithms are implemented correctly, it is often difficult to predict if a floating-point exception will occur, and it is possible that the tight or a priori bounds contain a NaN. Then this function returns false. Although determining what has happened may not be very difficult, this may be time consuming. If the computations stop immediately when a NaN occurs, it would be easier to locate where a NaN has occurred.

1.10.2 Wrapping Effect in Generating Taylor Coefficients

Generating point Taylor coefficients for the solution of an IVP for an ODE is not a difficult task, though not trivial to implement. In the interval case, however, there may be a wrapping effect that would lead to large overestimation in these coefficients.

Consider the constant coefficient problem

$$y' = By, \quad y(0) \in [y_0]. \tag{1.26}$$

In practice, the relation (1.12) is used for generating interval Taylor coefficients. As a result, the ith coefficient is computed by

$$[y]_i = \frac{1}{i}B[y]_{i-1} = \frac{1}{i!}B\big(B\cdots\big(B\,(B[y_0])\big)\cdots\big). \tag{1.27}$$

Therefore, the computation of the ith Taylor coefficient may involve i wrappings. In general, this implies that the computed enclosure of the kth Taylor coefficient, $f^{[k]}([\tilde{y}_j])$, on $[t_j, t_{j+1}]$ may be a significant overestimation of the range of $y^{(k)}(t)/k!$ on $[t_j, t_{j+1}]$. As a result, a variable stepsize method that controls the width of $h_j^k f^{[k]}([\tilde{y}_j])$ may impose a stepsize limitation much smaller than one would expect. In this example, it would be preferable to compute the coefficients directly by

$$[y]_i = \frac{1}{i!}B^i[y_0],$$

which involves at most one wrapping.

We do not know how to reduce the wrapping effect in generating interval Taylor coefficients. The TADIFF package implements the relation (1.12), thus generating coefficients for (1.26) as in (1.27). We have not verified whether IADOL-C produces interval Taylor coefficients for (1.26) as in (1.27), but we expect that this is the case.

Acknowledgments: This work was supported in part by the Natural Sciences and Engineering Research Council of Canada, the Information Technology Research Centre of Ontario, and Communications and Information Technology Ontario.

1.11 References

[AH83] G. Alefeld and J. Herzberger. *Introduction to Interval Computations*. Academic Press, New York, 1983.

[Ber97] M. Berz. COSY INFINITY version 8 reference manual. Technical Report MSUCL–1088, National Superconducting Cyclotron Lab., Michigan State University, East Lansing, Mich., 1997. COSY INFINITY is available at http://www.beamtheory.nscl.msu.edu/cosy/.

[BGMS97] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. Parallelism in object-oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhäuser, Boston, 1997. See also http://www.mcs.anl.gov/petsc/.

[BM98] M. Berz and K. Makino. Verified integration of ODEs and flows using differential algebraic methods on high-order Taylor models. *Reliable Computing*, 4:361–369, 1998.

[Boo94] G. Booch. *Object-Oriented Analysis and Design*. The Benjamin/Cummings Publishing Company Inc., Rational, Santa Clara, California, 2nd edition, 1994.

[BS96] 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. FADBAD is available at http://www.imm.dtu.dk/fadbad.html.

[BS97] 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. TADIFF is available at http://www.imm.dtu.dk/fadbad.html.

[Cop92] J. O. Coplien. *Advanced C++ Programming Styles and Idioms*. Addison-Wesley, AT & T Bell Laboratories, 1992.

[CR87] G. F. Corliss and L. B. Rall. Adaptive, self-validating quadrature. *SIAM J. Sci. Stat. Comput.*, 8(5):831–847, 1987.

[CR96] G. F. Corliss and R. Rihm. Validating an a priori enclosure using high-order Taylor series. In G. Alefeld and A. Frommer,

editors, *Scientific Computing, Computer Arithmetic, and Validated Numerics*, pages 228–238. Akademie Verlag, Berlin, 1996.

[DNS97] V. K. Decyk, C. D. Norton, and B. K. Szymanski. Expressing object-oriented concepts in Fortran 90. *ACM Fortran Forum*, 16(1):13–18, April 1997.

[Eij81] P. Eijgenraam. *The Solution of Initial Value Problems Using Interval Arithmetic*. Mathematical Centre Tracts No. 144. Stichting Mathematisch Centrum, Amsterdam, 1981.

[ES90] M. A. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual*. Addison–Wesley, 1990.

[GJU96] A. Griewank, D. Juedes, and J. Utke. ADOL-C, a package for the automatic differentiation of algorithms written in C/C++. *ACM Trans. Math. Softw.*, 22(2):131–167, June 1996.

[Han92] E. R. Hansen. *Global Optimization Using Interval Analysis*. Marcel Dekker, New York, 1992.

[Hoh93a] A. Hohmann. An implementation of extrapolation codes in C++. Technical Report TR 93-08, Konrad-Zuse-Zentrum für Informationstechnik, Berlin, Germany, 1993. Available at http://www.zib.de/bib/pub/pw/index.en.html.

[Hoh93b] A. Hohmann. TIDE, 1993. Available at http://www.rt.e-technik.tu-darmstadt.de/~mali/DOC/tide/welcome.html.

[Iwa97] R. V. Iwaarden. IADOL-C, personal communications, 1997. IADOL-C is available through the author. E-mail: vaniwaar@metsci.com.

[Jue91] D. Juedes. A taxonomy of automatic differentiation tools. In A. Griewank and G. F. Corliss, editors, *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, pages 315–329. SIAM, Philadelphia, Penn., 1991.

[KDDH95] R. B. Kearfott, M. Dawande, K. Du, and C. Hu. Algorithm 737: INTLIB: A portable Fortran 77 interval standard function library. *ACM Trans. Math. Softw.*, 20(4):447–459, December 1995.

[Kea95] R. B. Kearfott. Fortran 90 environment for research and prototyping of enclosure algorithms for constrained and unconstrained nonlinear equations. *ACM Trans. Math. Softw.*, 21(1):63–78, March 1995.

[Kea96a] R. B. Kearfott. Interval computations: Introduction, uses, and resources. *Euromath Bulletin*, 2(1):95–112, 1996.

[Kea96b] R. B. Kearfott. INTERVAL_ARITHMETIC: A Fortran 90 module for an interval data type. *ACM Trans. Math. Softw.*, 22(4):385–392, 1996.

[KKN+92] R. Klatte, U. Kulisch, M. Neaga, D. Ratz, and C. Ullrich. *Pascal–XSC: Language Reference with Examples.* Springer-Verlag, Berlin, 1992.

[Knü94] O. Knüppel. PROFIL/BIAS – a fast interval library. *Computing*, 53(3–4):277–287, 1994. PROFIL/BIAS is available at http://www.ti3.tu-harburg.de/Software/PROFILEnglisch.-html.

[Krü69] F. Krückeberg. Ordinary differential equations. In E. Hansen, editor, *Topics in Interval Analysis*, pages 91–97. Clarendon Press, Oxford, 1969.

[KU93] F. Korn and C. Ullrich. Verified solution of linear systems based on common software libraries. *Interval Computations*, 3:116–132, 1993.

[Lan99] H. P. Langtangen. *Computational Partial Differential Equations – Numerical Methods and Diffpack Programming.* Springer-Verlag, 1999.

[Loh87] R. J. Lohner. Enclosing the solutions of ordinary initial and boundary value problems. In E. W. Kaucher, U. W. Kulisch, and C. Ullrich, editors, *Computer Arithmetic: Scientific Computation and Programming Languages*, pages 255–286. Wiley-Teubner Series in Computer Science, Stuttgart, 1987.

[Loh88] R. J. Lohner. *Einschließung der Lösung gewöhnlicher Anfangs– und Randwertaufgaben und Anwendungen.* PhD thesis, Universität Karlsruhe, 1988. AWA is available at ftp://iamk4515.mathematik.uni-karlsruhe.de/pub/awa/.

[Loh95] R. J. Lohner. Step size and order control in the verified solution of IVP with ODE's, 1995. SciCADE'95 International Conference on Scientific Computation and Differential Equations, Stanford, California, March 28 – April 1, 1995.

[Moo66] R. E. Moore. *Interval Analysis.* Prentice-Hall, Englewood Cliffs, N.J., 1966.

[MR90] M. Metcalf and J. Reid. *Fortran 90 Explained.* Oxford University Press, Oxford, England, 1990.

[Ned99] 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. Available at http://www.cs.toronto.edu/NA/reports.html.

[Neu90] A. Neumaier. *Interval Methods for Systems of Equations*. Cambridge University Press, Cambridge, 1990.

[NJ98] 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*, Submitted, October 1998. Available at http://www.cs.toronto.edu/NA/reports.html.

[NJC99] N. S. Nedialkov, K. R. Jackson, and G. F. Corliss. Validated solutions of initial value problems for ordinary differential equations. *Appl. Math. Comp.*, To appear, 1999. Available at http://www.cs.toronto.edu/NA/reports.html.

[Ols97] H. Olsson. Object-oriented solvers for initial value problems. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools for Scientific Computing*. Birkhäuser, 1997. Godess is available at http://www.-cs.lth.se/home/Hans_Olsson/Godess/.

[Ral81] L. B. Rall. *Automatic Differentiation: Techniques and Applications*, volume 120 of *Lecture Notes in Computer Science*. Springer Verlag, Berlin, 1981.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, New York, 1991.

[Rih94] R. Rihm. On a class of enclosure methods for initial value problems. *Computing*, 53:369–377, 1994.

[Rob96] A. D. Robinson. C++ gets faster for scientific computing. *Computers in Physics*, 10(5):458–462, Sep/Oct 1996.

[SA93] 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.*, 9:233–246, 1993.

[Spe80] B. Speelpenning. *Compiling Fast Partial Derivatives of Functions Given by Algorithms*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana-Champaign, Ill., January 1980.

[Sta97] O. Stauning. *Automatic Validation of Numerical Solutions*. PhD thesis, Technical University of Denmark, Lyngby, Denmark, October 1997. This thesis and ADIODES are available at http://www.imm.dtu.dk/fadbad.html#download.

[SZ95] S. J. Sullivan and B. G. Zorn. Numerical analysis using nonprocedural paradigms. *ACM Trans. Math. Softw.*, 21(3):267–298, Sept 1995.

[SZAB98] M. J. Schulte, V. Zelov, A. Akkas, and J. C. Burley. The interval-enhanced GNU Fortran compiler. *Reliable Computing*, Submitted, October 1998.

[Vel95] T. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, June 1995.

[Vel97] T. Veldhuizen. Scientific computing: C++ versus Fortran. *Dr. Dobb's Journal*, 34, November 1997.

[VJ97] T. Veldhuizen and M. E. Jernigan. Will C++ be faster than Fortran? In *Proceedings of the 1st International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'97)*, Lecture Notes in Computer Science. Springer-Verlag, 1997.

[Wan77] G. Wanner. On the integration of stiff differential equations. In *Proceedings of the Colloquium on Numerical Analysis*, volume 37 of *Internat. Ser. Numer. Math.*, pages 209–226, Basel, 1977. Birkhäuser.

[YC96] W. Yang and G. F. Corliss. Bibliography of computational differentiation. In M. Berz, C. H. Bischof, G. F. Corliss, and A. Griewank, editors, *Computational Differentiation: Techniques, Applications, and Tools*, pages 393–418. SIAM, Philadelphia, Penn., 1996.