GRKLib: a Guaranteed Runge Kutta Library

Olivier Bouissou
CEA LIST
olivier.bouissou@cea.fr

Matthieu Martel
CEA LIST
matthieu.martel@cea.fr

Abstract

In this article, we describe a new library for computing guaranteed bounds of the solutions of Initial Value Problems (IVP). Given an initial value problem and an end point, our library computes a sequence of approximation points together with a sequence of approximation errors such that the distance to the true solution of the IVP is below these error terms at each approximation point. These sequences are computed using a classical Runge-Kutta method for which truncation and roundoff errors may be over-approximated. We also compute the propagation of local errors to obtain an enclosure of the global error at each computation step. These techniques are implemented in a C++ library which provides an easy-to-use framework for the rigorous approximation of IVP. This library implements an error control technique based on step size reduction in order to reach a certain tolerance on local errors.

1. Introduction

Users of numerical solvers for Ordinary Differential Equations (ODEs) are generally interested in computing approximation points with an *estimate* of the error at each point. Many scientists and engineers are more interested in the efficiency of the method than on the quality of the error estimation. However, for many safety critical applications, this estimation is not enough, and safe bounds on the error are required. These applications that need *guaranteed* approximations include state estimation [8], hybrid systems analysis [6] or industrial systems where critical damage may occur. For such systems, the required feature of the solver is safety rather than efficiency, i.e. the numerical solver should not only give an approximation of the solution but it should also prove that the true solution lies between computed bounds. This problem has been studied over the past 40 years, i.e. almost since interval arithmetic was invented. However, using interval versions of classical algorithm gives validated solvers which usually suffer from a bad long term stability. The most successful methods are based on a Taylor Series expansion of the solution with respect to time and a fine algorithm for computing the remainder terms.

In the rest of this introduction, we briefly recall what an Initial Value Problem is and we give the foundations of Taylor Series based methods for computing rigorous bounds of IVPs. For a more complete description of these methods and tools implementing them, we invite the reader to refer to [13].

1.1. Initial Value Problems

An IVP consists of a system of ODEs together with an initial condition:

$$\dot{y} = f(y) \qquad y(x_0) = y_0. \tag{1.1}$$

Here, y is a function from $\mathbb{R}$ to $\mathbb{R}^n$ and f is a continuous function from $\mathbb{R}^n$ to $\mathbb{R}^n$. $\dot{y}$ denotes the derivative of y with respect to time x.

Solving the IVP means finding a (possibly unique) function $y(x; x_0, y_0)$ which satisfies Equation (1.1). Numerical solvers usually compute a sequence of approximation points $\{y_0, y_1, \ldots, y_M\}$ such that y_i is an approximation of the value $y(x_i; x_0, y_0)$, for some x_i. Let $y_i^v = y(x_i; x_0, y_0)$ be the real value of the solution of (1.1) at time x_i. The sequence $\big(x_n\big)$ is the sequence of *steps*, and we let $h_i = x_{i+1} - x_i$ denote the step sizes. In sections 2 and 3, we assume a fixed step size. The step size control mechanism and its influence on the algorithm are detailed in Section 4.

If we consider systems with uncertainties, the initial conditions are not always exactly known, or we may only have approximate values for the parameters of the equations. Therefore, we will focus on a more general IVP, where the initial conditions are given as follows:

$$y(x_0) \in [y_0], \;\; [y_0] \subseteq \mathbb{R}^n \tag{1.2}$$

Solving this interval IVP means finding the set of functions $y(x; x_0, [y_0]) = \{y(x; x_0, y_0) \mid y_0 \in [y_0]\}$. Again, this problem is difficult. All we can do is to compute a sequence of boxes $[y_n]$ such that $\forall n, \; y(x_n; x_0, [y_0]) \subseteq [y_n]$.

1.2. Taylor Series Methods in a Nutshell

The method studied and used most often to achieve guaranteed bounds on the solutions of IVPs is based on an interval version of classical Taylor Series algorithm [4, 10, 12, 17]. This method performs a Taylor decomposition of the solution of (1.1) with respect to time, in such a way that:

$$y_{j+1} = y_j + \sum_{k=1}^{N-1} f^{[k]}(y_j)h_j^k + h_j^N \cdot f^{[N]}\big(y(x_s)\big), \qquad (1.3)$$

where $x_s \in [x_j, x_{j+1}]$ and $f^{[k]} = \frac{1}{k}\left(\frac{\partial^{k-1} f}{\partial y^{k-1}}\right)(y)$. A direct translation of (1.3) into interval arithmetic gives Equation (1.4), where $[\tilde{y_j}]$ is an a priori enclosure of $y(x_s)$:

$$[y_{j+1}] = [y_j] + \sum_{k=1}^{N-1} f^{[k]}([y_j])h_j^k + f^{[N]}\big([\tilde{y_j}]\big). \qquad (1.4)$$

In a direct evaluation of (1.4), the width of $[y_j]$ grows, even if the system contracts. Thus, it is important to compute $[y_{j+1}]$ in a way which limits the overestimation inherent in interval arithmetic. This is achieved by expressing the interval valued evaluations by their mean value form:

$$f(\{a \in [\underline{a}, \overline{a}]\}) \subseteq \hat{a} + J(f, [\underline{a}, \overline{a}]) \cdot ([\underline{a}, \overline{a}] - \hat{a}) \subseteq f([\underline{a}, \overline{a}]),$$

where $J(f, [\underline{a}, \overline{a}])$ is the Jacobian of the function f evaluated on the whole interval $[\underline{a}, \overline{a}]$, and $\hat{a} \in [\underline{a}, \overline{a}]$. If we apply this formula to (1.4), we obtain:

$$\begin{aligned}[y_{j+1}] \quad = \quad & \hat{y_j} + \sum_{k=1}^{N-1} f^{[k]}(\hat{y_j})h_j^k + f^{[N]}\big([\tilde{y_j}]\big) + \\ & \big(I + \sum_{k=1}^{N-1} J(f^{[k]}, [y_j])h_j^k\big)([y_j] - \hat{y_j}).\end{aligned}$$

There are still two problems to solve to use this formula:

- finding the a priori enclosure $[\tilde{y_j}]$, i.e. a box such that $\forall x \in [x_j, x_{j+1}], y(x; x_j, [y_j]) \in [\tilde{y_j}]$.
- reducing the wrapping effect when computing $\big(I + \sum_{k=1}^{N-1} J(f, [y_j])h_j^k\big)([y_j] - \hat{y_j})$.

Thus, Taylor Series methods are generally two-step methods: they first compute an a priori enclosure of the solution on one integration step, then they reduce this enclosure to get $[y_{j+1}]$ as tight as possible. We will face these two problems in our method, as developed in sections 3.1 and 3.2.

1.3. Description of our Method

The main contribution of this article is to show the feasibility of another way for computing rigorous bounds on the solution of an IVP. Our approach is comparable to the one taken by ValEncIA-IVP [15]; we compute a sequence of non-validated approximation points together with a sequence of guaranteed bounds on the distance between these points and the exact solution. Therefore, this method may be seen as a predictor-corrector algorithm: we predict the value of the approximation points, and we correct them by computing an over-approximation of the global error. The interest of this approach is that the use of interval arithmetic is limited to verification tasks (computation of the errors), thus limiting the size of the intervals to grow too much.

We chose to base our method on a classical Runge-Kutta algorithm for the computation of the approximation points. The error is then estimated using the higher order derivatives of the function y we want to approximate. It is computed as the sum of three terms: the error due to the limited order of the Runge-Kutta method, the error propagated by the dynamical system, and finally the error due to the implementation of the algorithm on a finite precision machine. We start with a brief review of the RK4 algorithm we use (Section 2). Then we show how we compute the over-approximation of the global error (Section 3). Finally, we see how the step size may be controlled to achieve a required error bound in Section 4 and we give some numerical results and benchmarks in Section 5.

2. The RK4 Algorithm

The RK4 algorithm is a Runge-Kutta algorithm of order 4. Runge-Kutta algorithms are implicit schemes which compute the sequence of approximation points $\big(y_n\big)$ using only y_n and some intermediary points for the computation of y_{n+1}. Details on these methods may be found in many numerical analysis books, for example [3, 18]. The RK4 method can be seen as an extension of Euler's ($y_{n+1} = y_n + h \cdot f(y_n)$) and Midpoint's method ($y_{n+1} = y_n + h \cdot f(y_n + h/2 \cdot f(y_n))$). It uses four evaluations of f to compute y_{n+1}: one at the beginning of the interval, two at the middle and on at the end:

$$\begin{aligned} k_1 &= f(y_n) \\ k_2 &= f(y_n + h/2 \cdot k_1) \\ k_3 &= f(y_n + h/2 \cdot k_2) \\ k_4 &= f(y_n + h \cdot k_3) \\ y_{n+1} &= y_n + \tfrac{h}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right). \end{aligned} \qquad (2.1)$$

It is commonly known, as we will see it in the next section, that using these formulas gives an approximation of order 4, i.e. the difference between the exact value $y(x_n + h)$ and y_{n+1} is of the same magnitude as h^5. If we apply successively these formulas, we obtain a set of approximation points y_j for every $x_j = x_0 + j * h$. These points are often believed to be a good approximation of the real solution. We show in the next section how one can compute guaranteed bounds on the error $y(x_j) - y_j$ for every j. Let us define some functions which will help to express the error bounds. We make the k_i coefficients depend on y and h, and define the iteration function Φ:

$$\begin{array}{rcl} k_1(y,h) &=& f(y) \\ k_2(y,h) &=& f(y + h/2 \cdot k_1(y,h)) \\ k_3(y,h) &=& f(y + h/2 \cdot k_2(y,h)) \\ k_4(y,h) &=& f(y + h \cdot k_3(y,h)) \\ \Phi(y,h) &=& y + \frac{h}{6}\big(k_1 + 2k_2 + 2k_3 + k_4\big)(y,h) \end{array} \quad (2.2)$$

such that we have $y_j = \Phi^j\big(y_0, h\big)$, where Φ^j is the jth iterate of the function Φ. We also define two partial functions ψ_j and ϕ_j at every step:

$$\psi_j : y \mapsto \Phi(y, h_j) \qquad \phi_j : x \mapsto \Phi\big(y_j^v, x - x_j\big). \quad (2.3)$$

3. Computing the Error

Let us now focus on the central part of our algorithm, i.e. the computation of guaranteed bounds on the global error. Our goal is to compute upper bounds of $\big(y(x_j; x_0, y_0) - y_j\big)$ for all the approximation points. Hence, we need to address the following questions: what is the error introduced by the method and how is it propagated by the dynamical system from one step into another? Furthermore, we need to be very careful on the implementation of the method, as any roundoff error must be taken into consideration.

Let us consider Step $n+1$: we already computed y_n and $[\epsilon_n]$ at x_n such that $y_n^v \subseteq y_n + [\epsilon_n]$. We recall that $y_n^v = y(x_n; x_0, y_0)$ is the value of the solution at time x_n. Let y_{n+1}^r be the real valued point given by the Runge-Kutta fomulas, which approximates y_{n+1}^v, and let y_{n+1} be the corresponding floating point given by the implementation. This is the point we will actually use. A first source of error comes from the difference between y_{n+1} and y_{n+1}^r. In addition, there is the error propagated by the dynamical system itself. Let y_{n+1}^* be the real valued point that we would have computed if we had applied the RK4 formulas starting from y_n^v. The distance between y_{n+1}^r and y_{n+1}^* represents how $[\epsilon_n]$ has been propagated into $[\epsilon_{n+1}]$. Finally, there is the error introduced by the method itself, which is the distance between y_{n+1}^* and y_{n+1}^v. So, the global error at x_{n+1} may be decomposed into three kinds of errors (see Figure 1):

- truncation errors due to the method: $\eta_{n+1} = y_{n+1}^v - y_{n+1}^*$,
- propagation of the previous error due to the dynamical system: $\chi_{n+1} = y_{n+1}^* - y_{n+1}^r$,
- roundoff errors due to finite precision computations: $\epsilon_{n+1} = y_{n+1}^r - y_{n+1}$.

In the following, we explain how these terms are computed.

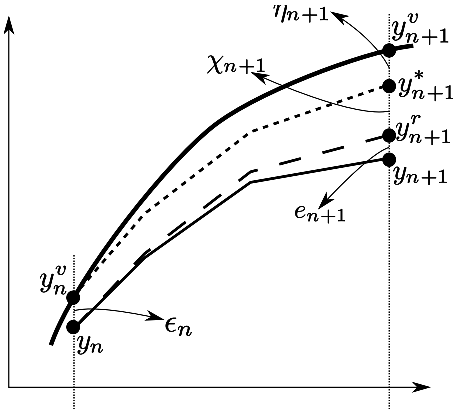

Figure 1. Three kinds of errors.

3.1. Truncation Errors

Truncation errors arise because the trajectory of the true solution to Equation (1.1) (curved line on Figure 1) and the trajectory given by the RK4 formulas (dotted line) differ. We suppose that they have a common starting point y_n^v at x_n, and we try to over-approximate their difference $y(x_{n+1}; x_n, y_n^v) - y_{n+1}^*$ at $x_{n+1} = x_n + h_n$. The following proposition then holds:

Proposition 3.1 $y_{n+1}^* = \phi_n\big(x_{n+1}\big)$, *and the four first derivatives of* y *and* ϕ_n *are equal at* $x = x_n$*:*

$$\forall i \in [0,4],\ \frac{d^i y}{dx^i}\big(x_n\big) = \frac{d^i \phi_n}{dx^i}\big(x_n\big).$$

The proof of this proposition is straightforward and may be read in [2]. Proposition 3.2 follows immediately:

Proposition 3.2 *There exists some* $\xi \in [x_n, x_{n+1}]$ *such that*

$$\eta_{n+1} = \frac{h^5}{120} \frac{d^5(y - \phi_n)}{dx^5}\big(\xi\big).$$

The proof is once again a straightforward application of the Taylor Series theorem with Lagrange remainder. Now, let us compute this fifth derivative. We have:

$$\begin{array}{rcl} \dfrac{d^5(y - \phi_n)}{dx^5} &=& \dfrac{d^5(y)}{dx^5} - \dfrac{d^5(\phi_n)}{dx^5} = \dfrac{d^4(f)}{dx^4} - \dfrac{d^5(\phi_n)}{dx^5} \\ \eta_{n+1} &=& \dfrac{d^4(f)}{dx^4}\big(y(\xi; x_n, y_n^v)\big) - \dfrac{d^5(\phi_n)}{dx^5}\big(\xi\big). \end{array}$$

The function ϕ_n only depends on x; so does $\frac{d^5(\phi_n)}{dx^5}$. Therefore, we have:

$$\frac{d^5(\phi_n)}{dx^5}\big(\xi\big) \in \frac{d^5(\phi_n)}{dx^5}\big([x_n, x_{n+1}]\big). \quad (3.1)$$

For the term $\frac{d^4(f)}{dx^4}\big(y(\xi; x_n, y_n^v)\big)$, the situation is a bit more complicated, as we cannot easily enclose the value

$y(\xi; x_n, y_n^v)$. We thus need an a priori enclosure of the function y on the interval $[x_n, x_{n+1}]$, i.e. a box $[\tilde{y_n}]$ such that $\forall x \in [x_n, x_{n+1}], y(x; x_n, y_n^v) \in [\tilde{y_n}]$. This will be done using Picard operator and Banach fixed point theorem, as in [10, 12]. Let us recall the definition of the operator and the main result that we will use (Proposition 3.3).

Defintion 3.1 *Given an ODE $\dot{y} = f(y)$, $y(x_n) = y_n^v$, the associated Picard-Lindelöf operator is defined by:*

$$T(y)(t) = y_n^v + \int_{x_n}^{t} f(y(s))\, ds$$

Furthermore, let S be a closed subset of $C^0\big([x_n, x_{n+1}], \mathbb{R}^d\big)$, the set of continuous functions from $[x_n, x_{n+1}]$ into $\mathbb{R}^d$.

Proposition 3.3 *If f satisfies a Lipschitz condition and if S is mapped into itself by T, then there exists a unique solution to the ODE on $[x_n, x_{n+1}]$.*

This proposition is used to find an a priori enclosure of the solution of the ODE over the step $[x_n, x_{n+1}]$ [17]. Let $S' = \{y | y \in C^0\big([x_n, x_{n+1}], B\big)\}$ be the set of continuous functions over $[x_n, x_{n+1}]$ with value in the box B. For any $y \in S'$, we have:

$$\begin{array}{rcl}(Ty)(t) & = & y_n^v + \int_{x_n}^{t} f(y(s))ds \\ & \subseteq & y_n^v + [0, h_n] \cdot f(B)\end{array}$$

Now, if we have $y_n^v + [0, h_n] \cdot f(B) \subseteq B$, then for every $u \in S'$, $Tu \in S'$, so that S' is mapped into itself by T. Thus there is a unique solution to the ODE on $[x_n, x_{n+1}]$ that has values in B. We then just need to get a box $[\tilde{y_n}]$ such that $y_n^v + [0, h_n] f([\tilde{y_n}]) \subseteq [\tilde{y_n}]$. Let P be the interval Picard-Lindelöf operator defined by $P(R) = [y_n] + [0, h] f(R)$. We find $[\tilde{y_n}]$ by iterating P: we start from R_0 which contains $[y_n]$ and y^*_{n+1}, and we compute $R_n = P(R_{n-1})$. We stop once we found R_m such that $R_{m+1} \subseteq R_m$. We use R_{m+1} as our a priori enclosure for the values of $y(x; x_n, y_n^v)$, and then compute the over-approximation:

$$\frac{d^4 f}{dx^4}\big(y(\xi; x_n, y_n^v)\big) \in \frac{d^4 f}{dx^4}\big(R_{m+1}\big). \tag{3.2}$$

So, combining 3.1 and 3.2 gives an enclosure of η_{n+1}:

$$\eta_{n+1} \in \frac{d^4 f}{dx^4}\big(R_{m+1}\big) - \frac{d^5(\phi_n)}{dx^5}\big([x_n, x_{n+1}]\big) \tag{3.3}$$

3.2. Propagation of the Error

The propagation of the error at the nth step $[\epsilon_n]$ into the error at the $(n+1)$st step must account for the separation between the solutions of Equation (1.1) with an initial value y_n^v at x_n and with an initial value y_n: the point y^*_{n+1} is the application ψ_n at y_n^v, and y^r_{n+1} is the application of ψ_n at y_n. These two flows are functions defined over $D \subseteq \mathbb{R}^d$ with values in $D' \subseteq \mathbb{R}^d$. The separation of such functions is computed using the Jacobian matrix. For every differentiable function $f : D \rightarrow \mathbb{R}^d$, let

$$J(f) = \left(\begin{array}{cccc} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_d} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_d} \\ & & \cdots & \\ \frac{\partial f_d}{\partial x_1} & \frac{\partial f_d}{\partial x_2} & \cdots & \frac{\partial f_d}{\partial x_d} \end{array}\right).$$

be its Jacobian matrix, and let

$$J(f, Y) = \left(\begin{array}{cccc} \frac{\partial f_1}{\partial x_1}(Y) & \frac{\partial f_1}{\partial x_2}(Y) & \cdots & \frac{\partial f_1}{\partial x_d}(Y) \\ \frac{\partial f_2}{\partial x_1}(Y) & \frac{\partial f_2}{\partial x_2}(Y) & \cdots & \frac{\partial f_2}{\partial x_d}(Y) \\ & & \cdots & \\ \frac{\partial f_d}{\partial x_1}(Y) & \frac{\partial f_d}{\partial x_2}(Y) & \cdots & \frac{\partial f_d}{\partial x_d}(Y) \end{array}\right).$$

be its Jacobian matrix with all derivatives evaluated at some $Y \in D$. We may now express the extension of the mean value theorem to multivariate functions in a compact formulation:

Theorem 3.1 *Let $D \subseteq \mathbb{R}^d$ be an open subset of $\mathbb{R}^d$ and let $f : D \rightarrow \mathbb{R}^d$ be a differentiable function on D. Then, for all a and u such that $[a, a+u] \subset D$, there exists $\theta \in]0, 1[$ such that*

$$f(a+u) - f(a) = J(f, a + \theta \cdot u) \cdot u.$$

Now, as $y^*_{n+1} = \psi_n(y_n^v)$ and $y^r_{n+1} = \psi_n(y_n)$, there must exist $\theta \in]0, 1[$ and $c = y_n^v + \theta(y_n - y_n^v)$ such that:

$$y^*_{n+1} - y^r_{n+1} = J\big(\psi_n, y_n^v + c\big) \cdot \big(y_n - y_n^v\big). \tag{3.4}$$

This formula gives the exact value of χ_{n+1}, assuming we know the exact value of c and y_n^v, which we do not. However, the following holds:

$$\forall \theta \in]0, 1[, y_n^v + \theta\big(y_n + y(x_n)\big) \in [y_n, y(x_n)] \subseteq y_n + [\epsilon_n].$$

Furthermore, we know that $y_n^v \in y_n + [\epsilon_n]$, so we may enclose the propagation of $[\epsilon_n]$ as follows:

$$\begin{array}{rcl}\chi_{n+1} & = & J\big(\psi_n, y_n^v + c\big) \cdot \big(y_n - y_n^v\big) \\ & \in & J\big(\psi_n, y_n + [\epsilon_n]\big) \cdot [\epsilon_n].\end{array} \tag{3.5}$$

Equation (3.5) gives a method for computing χ_{n+1}. However, if we apply this formula naively, the wrapping effect mentioned in Section 1.2 makes the error grow. Actually, if the Jacobian matrix is a rotation matrix, then huge over-estimations are performed at each step, as shown by Figure 2 (where $J = J\big(\psi_n, y_n + [\epsilon_n]\big)$). To limit this problem, we can use the same techniques as Taylor Series methods, for example Löhner's QR factorization method [10]. Its idea is the following: instead of representing errors as boxes in the standard orthogonal coordinate system, we will express them in a different, better basis before performing the multiplication with the Jacobian matrix. The new coordinate system is constructed as follows: the first axis is chosen parallel to the longest edge of J, and we construct the other axis so that they are as parallel as possible to other edges. This is achieved by permuting the columns of J and then computing its QR-factorization. For more details, see [12].

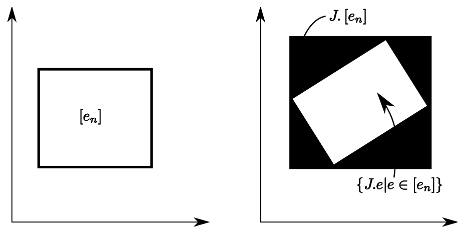

Figure 2. Wrapping effect.

$$a = f_a + e_a\overrightarrow{\varepsilon_e} \text{ and } b = f_b + e_b\overrightarrow{\varepsilon_e}$$

$$\begin{aligned}
a+b &= \uparrow_\sim (f_a+f_b) + (e_a+e_b+ \downarrow_\sim (f_a+f_b))\overrightarrow{\varepsilon_e} \\
a-b &= \uparrow_\sim (f_a-f_b) + (e_a-e_b+ \downarrow_\sim (f_a-f_b))\overrightarrow{\varepsilon_e} \\
a\times b &= \uparrow_\sim (f_a\times f_b) \\
&\quad +\big(e_af_b+e_bf_a+e_ae_b+ \downarrow_\sim (f_a\times f_b)\big)\overrightarrow{\varepsilon_e}
\end{aligned}$$

Table 1. Global error arithmetic.

3.3. Roundoff Errors

Roundoff errors occur during the computation of the next approximation point y_{n+1}. They are due to the difference between the computation made on a finite precision machine, which leads to the floating point number y_{n+1}, and the computation that would be performed on real numbers, which leads to the real value y^r_{n+1}. This separation comes from the implementation of floating point numbers as defined by the IEEE754 Standard. They may be reduced using multi-precision arithmetic, such as the MPFR library [5], but not eliminated and we thus need to consider very carefully these error terms. We compute together with the floating point number y_{n+1} an interval $[e_{n+1}]$ such that $y^r_{n+1} \in y_{n+1} + [e_{n+1}]$. This interval which over-approximates all computation errors is computed using the *global error* arithmetic, first defined and used in the field of numerical validation of C programs [14]. The idea is to attach to each floating point number a formal term which represents the distance between the floating point and the real number it is supposed to represent. Thus, a global error number a may be written as:

$$a = f_a + [e_a]\overrightarrow{\varepsilon_e},$$

where f_a is a floating point number and $\overrightarrow{\varepsilon_e}$ is a formal variable. $[e_a]$ is supposed to be the difference between the real value x_a and its floating point representation f_a. It should thus be a real number, and we naturally implement it as an interval. This representation means that a is a floating point number with value f_a which represents a real number x_a such that $x_a \in f_a + [e_a]$. We have two kinds of information: the result of a floating point computation and its distance to the real value. The main advantage of this representation is that the final enclosure on x_a, namely $f_a + [e_a]$, does not need to contain f_a, whereas all interval based representations (either infimum/supremum or midpoint/radius) produce enclosures that contain both the real value and the floating point number. Thus, the width of our error term $[e_a]$ is usually smaller than the width of the otherwise computed interval. Moreover, it is possible to use more precision for the computation of the error term than for the floating point term, in order to decrease the width of $[e_a]$. In our implementation, we use double the precision for the error terms.

Let us give an example to explain the gain of this arithmetic compared to interval arithmetic. Suppose that we have floating point numbers with a mantissa of 5 digits only. We start from $y = 1$ and subtract 10^6 times 10^{-6} to y. In real number arithmetic, the result is obviously 0. In floating point arithmetic, the result will be $y = 1$ as a cancellation occurs at each subtraction. If we do the same computation with the infimum/supremum interval arithmetic, we obtain a final interval of width 10^{66}. With the midpoint/radius arithmetic, using doubled precision for the radius terms, we obtain $(1, 1.0001)$, i.e. an enclosure of width 2 for the real result. With the global error arithmetic, we obtain $1 + [-1.0001, -9.9977]\overrightarrow{\varepsilon_e}$, i.e. an enclosure of width $4 \cdot 10^{-4}$. This difference[1] is due to the fact that the error term in the global error representation is *directed*: it is not a radius indicating in which circle the real value lies but rather an arrow aiming at it. For a more complete comparison between various arithmetic used for the validation of numerical programs, see [11].

The casting of a real number x into a global error number a proceeds as follows: the floating point part of a is the closest floating point number to x, denoted $\uparrow_\sim (x)$, and its error part is the distance between x and $\uparrow_\sim (x)$, denoted $\downarrow_\sim (x)$. Thus we have:

$$a = \uparrow_\sim (x) + \downarrow_\sim (x)\overrightarrow{\varepsilon_e}.$$

In a computer, the value $\downarrow_\sim (x)$ is enclosed by the interval $[-u, u]$, where u is the value of the last bit of the representation of $\uparrow_\sim (x)$, and doubled precision is used. Basic operations and elementary functions are defined over such numbers. The rules for addition, subtraction and multiplication are given in Table 1. So, if we use this arithmetic to compute y_{n+1}, we not only get the floating point value of the next step but also its distance to the real value y^r_{n+1}:

$$\begin{aligned}
y^r_{n+1} &= y_{n+1} + [e_{n+1}]\overrightarrow{\varepsilon_e} \\
y^r_{n+1} - y_{n+1} &\in [e_{n+1}].
\end{aligned} \quad (3.6)$$

[1]A c++ program showing these results can be downloaded at `http://www.lix.polytechnique.fr/Labo/Olivier.Bouissou/progs/patriot.cc`

3.4. Putting Things Together

Using formulas (3.3), (3.4) and (3.6), we have:

$$\begin{aligned}\epsilon_{n+1} &= \eta_{n+1} + \chi_{n+1} + e_{n+1} \\ &\in \frac{d^4 f}{dx^4}(R) - \frac{d^5(\phi_n)}{dx^5}([x_n, x_{n+1}]) + \\ &\qquad J\big(\psi_n, y_n + [\epsilon_n]\big).[\epsilon_n] + [e_{n+1}]. \qquad (3.7)\end{aligned}$$

4. Controlling the Step Size

The global error can be estimated with our method and interval arithmetic. Using this error estimator, one may want to control the error by modifying the step-size in order to achieve a prescribed accuracy. The main difficulty is that it is in general very expensive to control the global error. Actually, as it has been shown, the global error after $n+1$ steps is computed as:

$$y_{n+1} - y(x_{n+1}) = \big(y_{n+1} - y^*_{n+1}\big) + \big(y^*_{n+1} - y(x_{n+1})\big).$$

The second term represents the local error due to the method and its implementation, while the first one represents the stability of the dynamical system (in the sense of Lyapunov's stability [16]): given two close initial points, a stable system will join them whereas an unstable system will separate them. The second term depends on the chosen method, and we can control it, whereas the first term depends on the problem itself, and it is therefore not directly observable. If the problem is unstable between x_n and x_{n+1}, this term grows, and a reduction in the step-size does not affect this. Moreover, if we want to achieve a certain tolerance at time T, it may not be efficient to control the global error before this point, as the integral curves may converge only near T. So, controlling the global error generally requires at least two integrations of the problem, as the step-size selection at one point strongly depends on what happens next. Here we concentrate on the local error control for performance issues. Another reason why we should do that comes from the previous formula. Clearly, keeping the local error (the second term) small will affect the global error and help to keep it small. Moreover, control techniques often try to prevent from instability phenomenas and thus keep the values of y_n inside the stability region of the problem: the control techniques we present will reduce the step-size as soon as the derivative of the solution grows. Thus, the accuracy of the computation will be increased in the regions where the derivative is high, so that the first term of the formula will be kept low.

For all these reasons, we focus on the following problem: given a user-defined, absolute tolerance tol, adapt the step-size so that the error introduced at each step (both method and roundoff error) is smaller than tol. Obviously, as the step-size decreases, the method error decreases. However, the dependence of the roundoff error into the step-size is not so obvious, and this error clearly limits the accuracy one can obtain. Therefore, we only control the method error by adapting the step-size. If one wants to control the roundoff error as well, then increasing the precision of the computation by using multi-precision arithmetic such as the MPFR library would be the easiest way.

The method for controlling the step-size is derived from general methods of control theory for the automatic control of physical systems. The main idea is that the step-size is the adjustment variable to control the truncation error, which can be seen as a physical variable with an optimal value, tol, and that we must bring as close as possible to it. The dependence between the error and the step-size is given by the formula

$$\epsilon_{n+1} = |\varphi_n| \cdot h_n^5,$$

where $|\varphi_n|$ is the enclosure of the value of the fifth derivative of f on the a priori enclosure for the nth step (between x_n and x_{n+1}). This gives us a first control method, based on the assumption that $|\varphi_n| \approx |\varphi_{n+1}|$:

$$h_{n+1} = \left(\frac{tol}{|\varphi_{n+1}|h_n^5}\right)^{\frac{1}{5}} h_n$$

This controller is called the *integral* controller. Actually, if you use this formula, then the error after the $n+1$st step is $\epsilon_{n+2} = |\varphi_{n+2}|.h_{n+1}^5 = |\varphi_{n+1}|.\frac{tol}{|\varphi_{n+1}|.h_n^5}.h_n^5 = tol$. However, the assumption that $|\varphi_n| \approx |\varphi_{n+1}|$ is in general false for non trivial problems. Hence, this control mechanism tends to overcompensate, leading to many variations in the step-size and to rejected steps. To overcome this problem, a first solution would be to use $\theta.tol$ instead of tol in the formula, with $\theta \in [0,1]$. This smooths slightly the variations of h_n, but this strategy is still not satisfactory for complex problems. Hence, we build a more complex controller which considers not only the previous step error to compute the next step-size, but also takes into account the variation of the error over the last two steps. In this way, if the error is growing, then the previous controller will be adjusted so that the step-size does not increase too much. On the contrary, if the error is decreasing, the controller will amplify the variations of the step-size. The idea of this second controller is to add, as in control theory, a term proportional to the control error to the previous integral term. This leads to the formula:

$$h_{n+1} = \left(\frac{\theta.tol}{r_{n+1}}\right)^{k_1} \left(\frac{r_n}{r_{n+1}}\right)^{k_p} h_n,$$

with $r_n = |\varphi_{n+1}|.h_n^5$. k_1 and k_p are the controller parameters. The difficulty is to chose them to achieve a good performance. Ideally, they should be computed independently for every problem, as they strongly depend on its condition. However, choosing $k_1 = \frac{0.3}{k}$ and $k_p = \frac{0.2}{k}$ is a good choice for many problems.

5. Numerical Results and Benchmarks

We implemented this method in a C++ library that offers an easy-to-use framework for solving differential equations. Our implementation depends on two libraries:

- GiNaC [1], a formal derivation C++ library. With this tool, we generate at compile time C++ code for the higher order derivatives. We thus provide a source code transformation system which computes all the function needed for the computation of the error term.
- Profil/BIAS [9], a C++ interval library.

In this section, we study the performances of our library. First, we compare the speed and accuracy of our library with Taylor series methods; we chose to compare with AWA [10] and VNODELP [13]. AWA was the first software to use Taylor series to achieve guaranteed integration, and VNODELP is the new version of VNODE, which is probably the validated solver that is used the most. We run the tests with GRKLib, VNODELP and AWA but also with VNODELP and AWA limited to order 5. Actually, the main limitation of our method is its relatively low order, but our RK4 method is much more efficient than Taylor series method of the same order, while as precise as them. We then show how the method scales with problem size, tolerance and initial error. The tests were run on a laptop (two 1.7Ghz processors, 1Gb of RAM) running Ubuntu Linux. We used g++ (version 4.1) with optimization options -O2.

5.1. Linear problem: pure contraction

We integrate the IVP (5.1) on the interval $[0, 2000]$, and with an absolute tolerance of 10^{-12} at each step. This problem is a pure contraction, meaning that the Jacobian matrix does not rotate the error terms. Thus, the wrapping effect described in 3.2 is very limited.

$$\dot{Y} = \begin{pmatrix} -0.4375 & 0.0625 & 0.2652 \\ 0.0625 & 0.4375 & 0.2652 \\ -0.2652 & 0.2652 & 0.375 \end{pmatrix} Y \quad Y_0 = \begin{pmatrix} 1.0 \\ 1.0 \\ 1.0 \end{pmatrix} \tag{5.1}$$

Figure 3 shows how the CPU time depends on the integration time. VNODELP is 1.5 times faster than our library, but we are 3 times faster than VNODELP with order 5. AWA is more than 10 times slower than VNODELP of order 5, whatever the order is. The time for one step is much smaller for GRKLib ($4.6 \cdot 10^-5$ seconds against $3.8 \cdot 10^-4$ for VNODELP). The width of the final enclosure is $2 \cdot 10^{-9}$ for our method, 3.10^{-10} for VNODELP with order 5 and 10^{-12} for VNODELP and AWA. The error is bigger for GRKLib because the error control method controls the step size so that the local error introduced is of 10^{-12}, and we do not control the global error. If we integrate the same problem with fixed, small step size, we can achieve a final global error of 10^{-14} while being only 5 times slower. Thus, we would greatly benefit from a global error control which would predict the evolution of the step size in a better way.

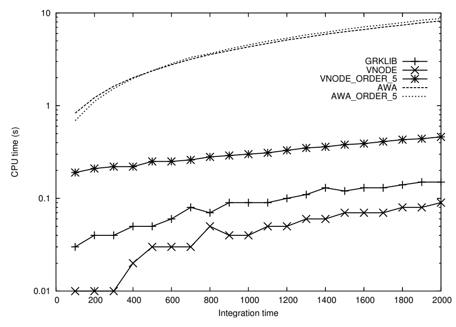

Figure 3. CPU time versus t for Problem (5.1)

5.2. Linear problem: pure rotation

We solve the IVP (5.2), on the interval $[0, 2000]$, with an absolute tolerance of 10^{-12} at each step. This problem is a pure rotation, meaning that the Jacobian matrix is almost a rotation matrix. Thus, it widely suffers from wrapping effect, and this problem shows the efficiency of Löhner's QR factorization method.

$$\dot{Y} = \begin{pmatrix} 0 & -0.707107 & 0.5 \\ 0.707107 & 0 & 0.5 \\ 0.5 & 0.5 & 0 \end{pmatrix} Y \quad Y_0 = \begin{pmatrix} 1.0 \\ 1.0 \\ 1.0 \end{pmatrix} \tag{5.2}$$

Figure 4 shows how the CPU time depends on the integration time. The time per step remains 10 times smaller for GRKLib, but we need to make more steps because of our limited order. The width of the error at $t = 2000$ is 2.10^{-9} for GRKLib, 10^{-11} for VNODELP and AWA and 6.10^{-10} for VNODELP with order 5. Once again, a global error control method would help to reduce the error width.

5.3. Non linear problem: Lorenz equations

Finally, we applied our method on Lorenz equations (5.3). We set the absolute tolerance to 10^{-12} and performed the integration on the interval $[0, 15]$.

$$\begin{cases} \dot{y_1} = \sigma.(y_2 - y_1) \\ \dot{y_2} = y_1.(\rho - y_3) - y_2 \\ \dot{y_3} = y_1.y_2 - \beta.y_3 \end{cases} \quad \begin{cases} y_1(x_0) = 15.0 \\ y_2(x_0) = 15.0 \\ y_3(x_0) = 36.0 \end{cases} \tag{5.3}$$

Figure 5 shows how the computation time depends on the time. Once again, the time per step is much smaller for GRKLib, but as the derivatives of the function take very high values, we need to make very small steps to achieve

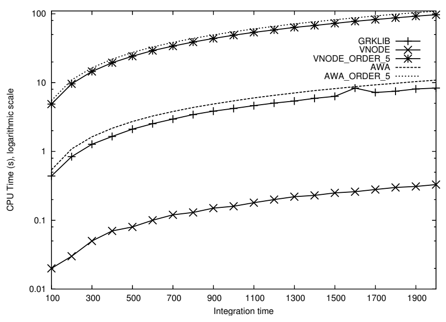

Figure 4. CPU time versus t for Problem (5.2)

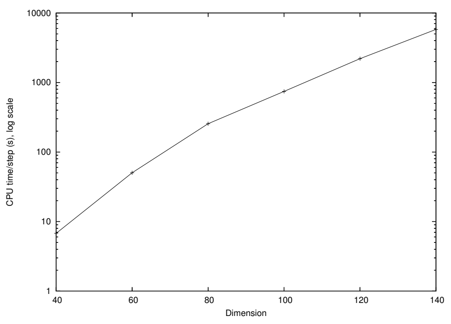

Figure 6. CPU time per step versus n

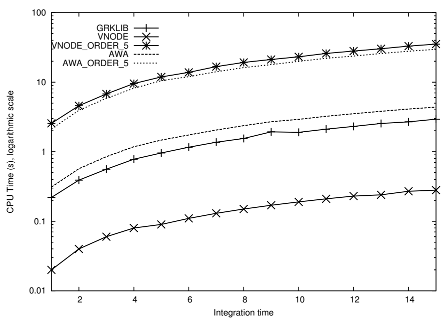

Figure 5. CPU time versus t for Problem (5.3)

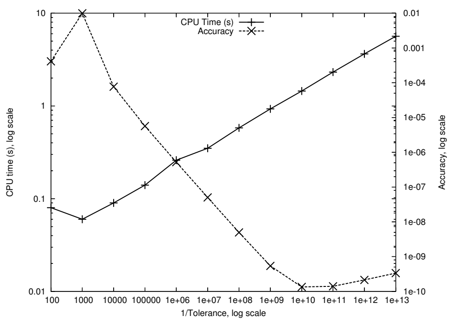

Figure 7. CPU time and accuracy versus tol

the given tolerance. However, we remain 10 times faster than VNODELP with order 5 and 2 times faster than AWA. The final error at $t = 15$ is $4 \cdot 10^{-3}$ for GRKLib, $5 \cdot 10^{-6}$ for VNODELP, $2 \cdot 10^{-3}$ for VNODELP with order 5 and 10^{-4} for AWA. AWA with order 5 did not make it to $t = 15$. The bigger number of steps we have to make explains this difference.

5.4. Performance

Work versus problem size

To study how computation time depends on the problem dimension, we used the DETEST problem C3 [7], as suggested by Nedialkov [13]. The problem is given by:

$$\dot{Y} = \begin{pmatrix} -2 & 1 & 0 & 0 & \dots & 0 \\ 1 & -2 & 1 & 0 & \dots & 0 \\ 0 & 1 & -2 & 1 & \dots & 0 \\ & & & \vdots & & \\ 0 & 0 & \cdots & 0 & 1 & -2 \end{pmatrix} Y$$

with $y_0 = (1, 0, \dots, 0)^T$. We solved the equation from 0 to 2 for dimensions $n = 40, 60, 80, \dots, 140$. Figure 7 shows the computation time per step (we need 42 steps to reach 2). The complexity is $O(n^3)$ because the QR method to reduce the wrapping effect requires a matrix decomposition which is the most time expensive part of the algorithm.

Work versus tolerance

To study the dependence of our method on the tolerance, we integrate the linear problem (5.1) on the interval $[0, 1000]$ with a tolerance $tol = 5.10^{-2}, 10^{-2}, 5.10^{-3}, \dots, 5.10^{-14}$. Figure 7 shows how CPU time and accuracy depends on the tolerance. As excepted, the accuracy increases as the tolerance decrease, whereas the CPU time increases.

Accuracy versus initial conditions

To study how the method depends on initial conditions, we integrate the Lorenz equations (5.3) with initial errors $tol = 10^{-4}, 10^{-5}, \dots, 10^{-9}$. We present on Figure 5.4 the maximum time at which we were able to integrate (5.3) for each value of tol. As excepted, the bounds diverge much quicker than for point initial conditions; on this problem, the bounds tend to explode as soon as they start diverging.

6. Conclusion

In this article, we have shown the feasibility of a guaranteed version of the classical numerical algorithm RK4. The guaranteed nature of our method differs from previous

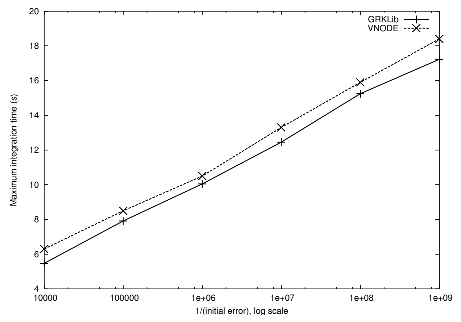

Figure 8. Maximum T_{end} versus initial error

works in the sense that we compute non validated points in a first step and then guaranteed error bounds. To compute these bounds, we localize every type of errors that may occur during the computation of the approximating points: truncation errors due to the method, propagation errors due to the dynamical system, and roundoff errors due to the precision of machines on which the method is implemented. We use a constant a priori enclosure to compute truncation errors, Löhner's method to reduce the wrapping effect in propagation errors, and the global error domain for roundoff errors. We also use a step-size control mechanism that gives good performance results.

We believe that this method is promising because it does not mix interval and floating point computations. Actually, intervals are used only for verification tasks and not for the computation of the next approximation points. This clear separation is analogous to the separation between floating point numbers and intervals in global error arithmetic. This has given very good results in the field of numerical validation, so we are confident that it should perform well for guaranteed integration.

The idea of separation between interval and floating point computations is orthogonal to the choice of the numerical method. Actually, all we need is an iteration function for which the derivatives are computable. We chose to base our library first on Runge-Kutta methods because they are widely used for numerical (non guaranteed) integration and that users generally know how to tune the step size control mechanism to make the method as precise as possible. However, we do not want to limit ourselves to this particular integration scheme, and we plan to add more sophisticated ones to our library to fit better to more types of problems. Especially, we are planning to use higher order numerical integration schemes; experimentations showed that the mean step size is the main limitation to performance and accuracy, the use of higher order methods will allow bigger step sizes and consequently much better performances.

References

[1] C. Bauer, A. Frink, and R. Kreckel. Introduction to the GiNaC framework for symbolic computation within the C++ programming language. *Journal of Symbolic Computation*, 33(1):1–12, 2002.

[2] L. Bieberbach. On the remainder of the runge-kutta formula in the theory of ordinary differential equation. *ZAMP*, 2:233–248, 1951.

[3] J. C. Butcher. *The numerical analysis of ordinary differential equations: Runge-Kutta and general linear methods*. Wiley-Interscience, New York, 1987.

[4] G. Corliss and Y. F. Chang. Solving ordinary differential equations using Taylor series. *ACM Transaction on Mathematical Software*, 8(2):114–144, 1982.

[5] G. Hanrot, V. Lefevre, R. F., and P. Zimmermann. The MPFR library. Available at `www.mpfr.org`.

[6] T. Henzinger and P. Ho. Algorithmic analysis of nonlinear hybrid systems. In *CAV*, volume 939 of *LNCS*, pages 225–238. Springer Verlag, 1995.

[7] T. Hull, W. Enright, B. Fellen, and A. Sedgwick. Comparing numerical methods for ordinary differential equations. *Journal on Numerical Analysis*, 9(4):603–637, 1972.

[8] M. Kieffer and E. Walter. Guaranteed nonlinear state estimator for cooperative systems. *Journal of Numerical Algorithms*, 37(1–4):187–198, Dec. 2004.

[9] O. Knüppel. PROFIL/BIAS – A fast interval library. *Computing*, 53:277–287, 1994.

[10] R. Löhner. *Einschliessung der Lösung gewöhnlicher Anfangs- und Randwertaufgaben und Anwendungen*. PhD thesis, Universität Karlsruhe, 1988.

[11] M. Martel. An overview of semantics for the validation of numerical programs. In *VMCAI*, volume 3385 of *LNCS*, pages 59–77. Springer, 2005.

[12] N. Nedialkov, K. Jackson, and G. Corliss. Validated solutions of initial value problems for ordinary differential equations. *Applied Mathematics and Computation*, 105(1):21–68, 1999.

[13] N. S. Nedialkov. Interval tools for ODEs and DAEs. Technical Report CAS 06-09-NN, Dept. of Computing and Software, McMaster University, 2006.

[14] S. Putot, E. Goubault, and M. Martel. Static analysis-based validation of floating-point computations. In *Numerical Software with Result Verification*, volume 2991 of *LNCS*, pages 306–313. Springer, 2003.

[15] A. Rauh, E. Auer, E. Hofer, and W. Luther. Validated modeling of mechanical systems with SmarMOBILE: Improvement of performance by ValEncIA-IVP. In *Reliable Implementation of Real Number Algorithms: Theory and Practice*. Springer Verlag, to appear.

[16] S. Sastry. *Nonlinear Systems Analysis, Stability and Control*. Spinger, Berlin, 1999.

[17] O. Stauning. *Automatic Validation of Numerical Solutions*. PhD thesis, Technical University of Denmark, Lyngby, Denmark, 1997.

[18] J. Stoer and R. Bulirsch. *Introduction to Numerical Analysis*. Springer Verlag, New York and Berlin, 1993.