Enclosing Temporal Evolution of Dynamical Systems Using Numerical Methods*

Olivier Bouissou[1], Alexandre Chapoutot[2], and Adel Djoudi[2]

[1] CEA Saclay Nano-INNOV Institut CARNOT, Gif-sur-Yvette, France
[2] ENSTA ParisTech, Palaiseau, France

Abstract. Numerical methods are necessary to understand the behaviors of complex hybrid systems used to design control-command systems. Especially, numerical integration methods are heavily used in simulation to compute approximations of the solution of differential equations, including non-linear and stiff solutions. Nevertheless, these methods only produce approximate results and they should not be used in formal verification methods as is. We propose a systematic way to make explicit Runge-Kutta integration method safe with respect to the mathematical solution. As side effect, we can hence compare different integration schemes in order to pick the right one in different situations.

1 Introduction

Verification techniques for embedded, control-command systems usually involve the modeling of the system using a hybrid automata-like formalism and then the computation of the reachable states of the system [1,2]. To compute these reachable states, one of the crucial points is the post operator for the continuous trajectories which requires to compute over-approximations of trajectories defined by ordinary differential equations (ODE in short). In the linear case (i.e. when the differential equations are linear), this can be exactly and efficiently solved using an efficient representation of convex sets as in [3]. For the non-linear case, one cannot in general compute exactly the continuous trajectories and approximation techniques such as hybridization [4,5] have been proposed. This however may result in an explosion of the number of discrete jumps and thus does not scale well to large, industrial systems.

In an industrial context, the validation of these systems (which differs to the formal verification) usually involves the modeling of the system in a Simulink-like formalism and then performing numerical simulations of the system to *test* its behavior under some input scenarios [6,7]. Numerical simulation techniques are very efficient and scale very well to large systems with many state variables. Moreover, system designers are used to tune and use these simulations to have good approximations of the system trajectories. Such simulations are however of little help for the formal verification of hybrid systems.

* This work was partially supported by the ANR project CAFEIN.

G. Brat, N. Rungta, and A. Venet (Eds.): NFM 2013, LNCS 7871, pp. 108–123, 2013.

In this article, we propose to use and adapt the numerical methods used in tools as Matlab/Simulink to define a new post operator for continuous trajectories. More formally, we define a way to transform any explicit Runge-Kutta numerical method to solve ODE into a guaranteed manner that computes over-approximations of the exact solution. We focus on explicit Runge-Kutta-like methods as they are the most widely used methods to solve differential equations. For example, in the Matlab/Simulink tool, there are 13 integration methods, 8 of which are explicit Runge-Kutta methods. It is well known that each method has its particularities and is suited for a particular kind of ODEs. So, having a collection of numerical methods allows one to choose the best one for solving its particular ODE. Our framework proposes different methods and thus allows to efficiently and precisely solve different kinds of equations.

In the rest of this article, we give in Section 2 an overview of numerical methods to solve ODEs, then in Section 3 we explain how we modified them to enclose the solution of ODEs. Then, in Section 4, we present experimentation that show the benefits of our approach compared to related work.

2 Numerical Integration

We now recall the principles of numerical integration of ordinary differential equations. An ordinary differential equation (ODE) is a relation between a function $y : \mathbb{R} \to \mathbb{R}^n$ and its derivative $\dot{y} = \frac{dy}{dt}$, written as $\dot{y} = f(t, y)$. An initial value problem (IVP) is an ODE together with an initial condition and a final time:

$$\dot{y} = f(t, y) \quad \text{with} \quad y(0) = y_0,\ y_0 \in \mathbb{R}^n \quad \text{and} \quad t \in [0, t_{\text{end}}]\ . \tag{1}$$

Example 1 (Running example). We use the following IVP as our running example:

$$\begin{cases} \dot{y} = z \\ \dot{z} = z^2 - \dfrac{3}{0.001 + y^2} \end{cases} \quad \text{with} \quad \begin{cases} y(0) = 10 \\ z(0) = 0 \end{cases}$$

and we set the final time at $t_{\text{end}} = 50$. We call this IVP the "oil-reservoir" problem, this example comes from [8]. This IVP is particularly stiff around $t = 35$, while its evolution elsewhere is slow, which makes it difficult to solve.

Solving the IVP means finding a continuous and differentiable function y_∞ such that $y_\infty(0) = y_0$ and $\forall t \in [0, t_{\text{end}}],\ \dot{y}_\infty(t) = f(t, y_\infty(t))$. We do not address here the problem of existence of the solution and we shall always assume that $f : \mathbb{R} \times \mathbb{R}^n \to \mathbb{R}^n$ is continuous in t and globally Lipschitz in y, so Equation 1 admits a unique solution on $\mathbb{R}$ [9]. We denote the solution of (1) with initial condition y_0 at $t = 0$ as $y(t; y_0)$. Higher order differential equations can be translated into first-order ODEs by introducing additional variables for the derivatives of y.

2.1 Approximate Solution

An exact solution of Equation 1 is rarely computable so that in practice, approximation algorithms are used. The goal of an approximation algorithm is to

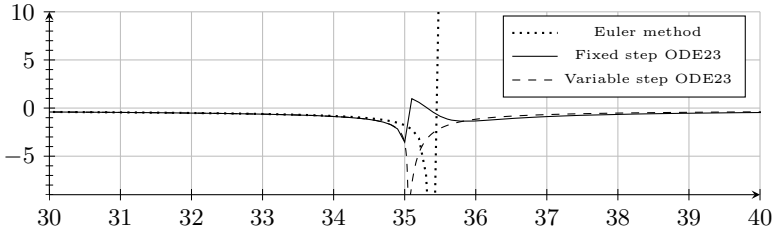


Fig. 1. Some numerical solutions of the oil-reservoir problem with different numerical methods, zooming on $t \in [30, 40]$

compute a sequence of time instants $0 = t_0 < t_1 < \cdots < t_n = t_{end}$ and a sequence of values $y_0, \ldots, y_n$ such that $\forall i \in [0, n]$, $y_i \approx y(t_i; y_0)$. In this article, we focus on single-step methods that only use y_i and approximations of $\dot{y}(t)$ to compute y_{i+1}.

The simplest method is Euler's method in which $t_{i+1} = t_i + h$ for some step-size h and $y_{i+1} = y_i + h \times f(t_i, y_i)$; so the derivative of y at time t_i, $f(t_i, y_i)$, is used as an approximation of the derivative on the whole time interval to perform a linear interpolation. This method is very simple and fast, but requires small step-sizes. More advanced methods use a few intermediate computations to improve the approximation of the derivative. For example, Bogacki-Shampine method (also named ODE23) performs three evaluations of f and then a linear interpolation from y_n using a weighted sum of the three derivative approximations (h is the chosen step-size):

$$k_1 = f(t_n, y_n) \tag{2a}$$

$$k_2 = f(t_n + (1/2)h, y_n + (1/2)hk_1) \tag{2b}$$

$$k_3 = f(t_n + (3/4)h, y_n + (3/4)hk_2) \tag{2c}$$

$$y_{n+1} = y_n + h\left((2/9)k_1 + (1/3)k_2 + (4/9)k_3\right) \tag{2d}$$

Example 2. If we consider the "oil-reservoir" problem of Example 1, Euler and Bogacki-Shampine with a fixed step-size of 0.1 produce two very different solutions, as plotted in Figure 1 in dotted and dashed lines. Note that Euler method diverges around $t = 35$, where the dynamics of the solution is very stiff.

When the derivatives of the solution exhibit high variations, for example around $t = 35$ for the "oil-reservoir" problem, it is important to adapt the step-size h to ensure that the approximate solution does not deviate too far from the exact solution. So called variable step-size methods use a second, more precise interpolation that is used as a reference of the solution. The distance between both interpolations is considered as the error made by the first interpolation. For the Bogacki-Shampine method, the second interpolation is given by:

$$k_4 = f(t_n + h, x_{n+1}) \tag{3a}$$

$$z_{n+1} = x_n + h\left((7/24)k_1 + (1/4)k_2 + (1/3)k_3 + (1/8)k_4\right) \tag{3b}$$

Then, $\| y_{n+1} - z_{n+1} \|$ is the estimated error attached to the approximation point y_{n+1} and is used to both validate the step from t_n to t_{n+1} and adapt the step-size.

$$\begin{array}{c|cccc} d_2 & a_{21} & & & \\ d_3 & a_{31} & a_{32} & & \\ \vdots & \vdots & & \ddots & \\ d_s & a_{s1} & a_{s2} & \cdots & a_{s,s-1} \\ \hline & w_1 & w_2 & \cdots & w_s \\ & w'_1 & w'_2 & \cdots & w'_s \end{array}$$

(a) General form.

$$\begin{array}{c|cccc} 0 & & & & \\ \frac{1}{2} & \frac{1}{2} & & & \\ \frac{3}{4} & 0 & \frac{3}{4} & & \\ 1 & \frac{2}{9} & \frac{1}{3} & \frac{4}{9} & \\ \hline & \frac{2}{9} & \frac{1}{3} & \frac{4}{9} & 0 \\ & \frac{7}{24} & \frac{1}{4} & \frac{1}{3} & \frac{1}{8} \end{array}$$

(b) Bogacki-Shampine table.

Fig. 2. Butcher table

Step-size control strategy is at the core of the performance of a numerical integration algorithms, both in terms of precision (the step-size is reduced when needed) and computation time (the step-size is increased when the solution is flat). We will present in Section 3 our method for controlling the step-size when performing guaranteed numerical integration. We refer to [9,10] for details about the step-size strategy for numerical algorithms. On Figure 1, we show the values of the approximated solution of the "oil-reservoir" problem for the Bogacki-Shampine method with a variable step-size: it greatly differs from the fixed step-size method, and it is actually very close to the actual solution (see Section 4).

In this article, we only consider methods based on Runge-Kutta methods, either fixed or variable step-size. All these methods can be described by a Butcher table (see Figure 2(a)). The d_i represent the time instants of the intermediate steps needed to compute the solution of $\dot{y} = f(t, y)$ over the interval $[t_n, t_n + h]$. The matrix made of the elements a_{ij} represents the weights used to approximate the interval solution from the previous intermediate steps. The elements w_i, w'_i (only for variable step-size methods) represent the latest weights to approximate the solution at time $t_n + h_n$ with two methods of different orders. For example, the Butcher table associated to Bogacki-Shampine is given at Figure 2(b). In consequence the elements of the Butcher table give a unified description for all the numerical integration methods members of the Rung-Kutta family. In the rest of this article, we denote by Φ a numerical method described by such a Butcher table: it relates two successive approximation points: $\forall n \in \mathbb{N},\ (t_{n+1}, y_{n+1}) = \Phi(t_n, y_n)$.

2.2 Problems with Numerical Integration

Numerical integration only provides *approximations* of the solution of the IVP. Even when using variable step-size methods, there is no guarantee that the chosen method is close to the solution, we merely know that the smaller the step-size is, the closer the approximations are to the solution. So, if we want to use numerical methods in cases when an over-approximation of the trajectories is needed, we need to compute error bounds $y(t_n; y_0) - y_n$ for all $n \in \mathbb{N}$.

Moreover, numerical integration only concerns IVP with a single initial value, i.e. $y_0 \in \mathbb{R}^n$. In hybrid systems model verification, it is necessary to enclose all the solutions of a differential equation starting from any point in a given set. More formally, given an initial set $S_0 \subseteq \mathbb{R}^n$, we want to compute bounds on the set of trajectories $\{y(t; y_0) \mid y' = f(t, y),\ y_0 \in S_0\}$. The method we develop in the rest of the article encodes sets of values using *affine arithmetic* and compute bounds on the solutions of the differential equations by computing bounds on $y(t_n; y_0) - y_n$ whatever the initial value is within some set S_0.

Finally, implementations of numerical methods very often suffer from the use of floating-point numbers which explain why, even if theoretically a Runge-Kutta method converges towards the solution when the step-size converges towards 0, it is in practice not the case. Our method handles these errors in a safe way.

3 Guaranteed Integration

We present our solutions of the drawbacks associated to the numerical solutions of IVP. Firstly, we present our approach to manipulate sets of values for handling uncertainties. Secondly, we describe the method to bound the truncation error introduced in numerical methods to provide guaranteed numerical integration.

3.1 Computing with Sets

The simplest and most common way to represent and manipulate sets of values is interval arithmetic [11]. Nevertheless, this representation usually produces too much over-approximated results in particular because of the *dependency problem*.

Example 3. Consider the ordinary differential equation $\dot{x}(t) = -x$ solved with the Euler's method with an initial value ranging in the interval $[0, 1]$ and with a step-size of $h = 0.5$. For one step of integration, we have to compute with interval arithmetic the expression $e = x + h \times (-x)$ which produces as a result the interval $[-0.5, 1]$. Rewriting the expression e such that $e' = x(1 - h)$, we obtain the interval $[0, 0.5]$ which is the exact result. Unfortunately, we cannot in general rewrite expressions with only one occurrence of each variable.

More generally, it can be shown that for most integration schemes the width of the result can only grow if we interpret sets of values as intervals.

To avoid this problem we use an improvement over interval arithmetic named *affine arithmetic* [12] which can track linear correlation between program variables. A set of values in this domain is represented by an *affine form* $\hat{x}$ (also called a *zonotope*), i.e. a formal expression of the form $\hat{x} = \alpha_0 + \sum_{i=1}^{n} \alpha_i \varepsilon_i$ where the coefficients α_i are real numbers, α_0 being called the *center* of the affine form, and the ε_i are formal variables ranging over the interval $[-1, 1]$. Obviously, an interval $a = [a_1, a_2]$ can be seen as the affine form $\hat{x} = \alpha_0 + \alpha_1 \varepsilon$ with $\alpha_0 = (a_1 + a_2)/2$ and $\alpha_1 = (a_2 - a_1)/2$. Moreover, affine forms encode linear dependencies between variables: if $x \in [a_1, a_2]$ and y is such that $y = 2x$, then x will be represented by the affine form $\hat{x}$ above and y will be represented as $\hat{y} = 2\alpha_0 + 2\alpha_1 \varepsilon$.

Affine arithmetic extends usual operations on real numbers in the expected way. For instance, the affine combination of two affine forms $\hat{x} = \alpha_0 + \sum_{i=1}^{n} \alpha_i \varepsilon_i$ and $\hat{y} = \beta_0 + \sum_{i=1}^{n} \beta_i \varepsilon_i$ with $a, b, c \in \mathbb{R}$, is given by:

$$a\hat{x} + b\hat{y} + c \quad = \quad (a\alpha_0 + b\beta_0 + c) + \sum_{i=1}^{n} (a\alpha_i + b\beta_i)\varepsilon_i \quad . \tag{4}$$

However, unlike the addition, most operations create new noise symbols. Multiplication for example is defined by:

$$\hat{x} \times \hat{y} \quad = \quad \alpha_0\alpha_1 + \sum_{i=1}^{n} (\alpha_i\beta_0 + \alpha_0\beta_i)\varepsilon_i + \nu\varepsilon_{n+1} \tag{5}$$

where $\nu = (\sum_{i=1}^{n} |\alpha_i|) \times (\sum_{i=1}^{n} |\beta_i|)$ over-approximates the error between the linear approximation of multiplication and multiplication itself. Other operations, like sin, exp, are evaluated using their Taylor expansions. Note that the set-based evaluation of an expression only consists in substituting all the mathematical operators, like + or sin, by their counterpart in affine arithmetic. We will denote by $\text{Aff}(e)$ the evaluation of the expression e using affine arithmetic.

Example 4. Consider again $e = x + h \times (-x)$ with $h = 0.5$ and $x = [0, 1]$ which is associated to the affine form $\hat{x} = 0.5 + 0.5\varepsilon_1$. Evaluating e with affine arithmetic without rewriting the expression, we obtain $[0, 0.5]$ as a result.

One of the main difficulties when implementing affine arithmetic using floating-point numbers is to take into account the unavoidable numerical errors due to the use of finite-precision representations for values (and thus rounding on operations). We use an approach based on computations of floating-point arithmetic named *error free transformations*: the round-off error can be represented by a floating-point number and hence it is possible to exactly compute it (we refer to [13] for more details on such methods). For instance, in the case of addition, the round-off error e generated by the sum $s = a + b$ is given by ($\odot$ stands for floating-point operations): $e = (a \ominus (s \ominus (s \ominus a))) \oplus (b \ominus (s \ominus a))$.

A second comment on the implementation is that an affine form $\hat{x}$ could be represented as an array of floats encoding the coefficients α_i. However, since in practice most of those coefficients are null, it is much more efficient to adopt a sparse representation and encode it as a list of pairs (i, α_i), sorted w.r.t. the first component, containing only coefficients $\alpha_i \neq 0$. Moreover, in order to limit the growth of the number of noise symbols in affine forms, we gather during simulation all the coefficients below a given threshold into a new noise symbol.

3.2 Enclosing the Truncation Error

We recall from Section 2 that a numerical integration method computes a sequence of approximations (t_n, y_n) of the solution $y(t; y_0)$ of the IVP defined in Equation (1) such that $y_n \approx y(t_n; y_0)$. Every numerical method member of the Runge-Kutta family follows the *condition order* [9]. This condition states that

a method of this family is of order p iff the $p+1$ first coefficients of the Taylor expansion of the solution and the Taylor expansion of the numerical methods are equal. Hence, at a time instant t_n the Taylor expansion of the solution with the Lagrange remainder states that $\exists \xi \in]t_n, t_{n+1}[$ such that:

$$\begin{aligned} y(t_{n+1}; y_0) &= y(t_n; y_0) + \sum_{i=1}^{p} \frac{h_n^i}{i!} y^{(i)}(t_n; y_0) + \frac{h_n^{p+1}}{(p+1)!} y^{(p+1)}(\xi; y_0) \\ &= y(t_n; y_0) + \sum_{i=1}^{p} \frac{h_n^i}{i!} f^{(i-1)}\left(t_n, y(t_n; y_0)\right) + \frac{h_n^{p+1}}{(p+1)!} f^{(p)}\left(\xi, y(\xi; y_0)\right) \quad . \end{aligned} \tag{6}$$

In Equation (6), $g^{(n)}$ stands for the n-th derivative of function g w.r.t. time t that is $\frac{d^n g}{dt^n}$ and $h_n = t_{n+1} - t_n$ is the step-size. Moreover, the general form of an explicit s-stage Runge-Kutta formula, that is using s evaluations of f, is:

$$y_{n+1} = y_n + h \sum_{i=1}^{s} b_i k_i \quad , \tag{7a}$$

$$k_1 = f\big(t_n,\, y_n\big) \quad , \qquad k_i = f\Big(t_n + c_i h,\, y_n + h \sum_{j=1}^{i-1} a_{ij} k_j\Big) \quad , \quad i = 2, 3, \ldots, s \quad . \tag{7b}$$

The coefficients c_i, a_{ij} and b_i are those given in a Butcher table (see Section 2). We define the function $\phi : \mathbb{R} \to \mathbb{R}^n$ by $\phi(t) = y_n + h_t \sum_{i=1}^{s} b_i k_i(t)$, $k_i(t)$ is defined as Equation (7b) where h is $h_t = t - t_n$. The Taylor expansion around t_n of the numerical solution with a Lagrange remainder states that there exists $\eta \in]t_n, t_{n+1}[$ such that:

$$y_{n+1} = \sum_{i=0}^{p} \frac{h_n^i}{i!} \frac{d^i \phi}{dt^i}(t_n) + \frac{h_n^{p+1}}{(p+1)!} \frac{d^{p+1}\phi}{dt^{p+1}}(\eta) \quad .$$

The truncation error measures the distance between the true solution and the numerical solution and it is defined by $y(t_n; y_0) - y_n$. If we express the truncation error with the Taylor expansions, the consequence of the condition order is that the numerical integration makes an error proportional to the Lagrange remainders. More precisely, the truncation error is defined by:

$$y(t_n; y_0) - y_n = \frac{h_n^{p+1}}{(p+1)!} \left(f^{(p)}\left(\xi, y(\xi)\right) - \frac{d^{p+1}\phi}{dt^{p+1}}(\eta) \right) \quad \xi \in]t_k, t_{k+1}[\text{ and } \eta \in]t_n, t_{n+1}[\quad . \tag{8}$$

The challenge to make Runge-Kutta integration schemes safe w.r.t. the true solution of IVP is then to compute a bound of the result of Equation (8). In other words we have to bound the value of $f^{(p)}\left(\xi, y(\xi; y_0)\right)$ and the value of $\frac{d^{p+1}\phi}{dt^{p+1}}(\eta)$. The latter expression is straightforward to bound because the function ϕ only depends on the value of the step-size h, and so does its $(p+1)$-th derivative. The bound is then obtain using the affine arithmetic by:

$$\frac{d^{p+1}\phi}{dt^{p+1}}(\eta) \in \mathrm{Aff}\left(\frac{d^{p+1}\phi}{dt^{p+1}}([t_n, t_{n+1}]) \right) \quad . \tag{9}$$

However, the expression $f^{(p)}\left(\xi, y(\xi; y_0)\right)$ is not so easy to bound as it requires to evaluate f for a particular value of the IVP solution $y(\xi; y_0)$ at a unknown

time $\xi \in]t_n, t_{n+1}[$. The solution used is the same as the one found in [14,15] and it requires to bound the solution of IVP on the interval $[t_n, t_{n+1}]$. We briefly recall the main mathematical tool used to bound the solution of IVP and we refer to [14] for a complete presentation. We consider the space of continuously differentiable functions $C^0([t_n, t_{n+1}], \mathbb{R}^n)$ and the Picard-Lindelöf operator:

$$P(f; t_n; y_n)(t) = y_n + \int_{t_n}^{t} f(s, y(s))ds \quad . \tag{10}$$

Note that this operator is associated to the integral form of Equation (1). So the solution of this operator is also the solution of Equation (1).

The Picard-Lindelöf operator is used to check the contraction of the solution on a integration step in order to prove the existence and the uniqueness of the solution of Equation (1) as stated by the Banach's fixed-point theorem. Furthermore, this operator is used to compute an enclosure of the solution of IVP over a time interval $[t_n, t_{n+1}]$ using affine arithmetic. Affine arithmetic can be used to compute a bound of integral expression such that: $\int_a^b f(x)dx \in (b-a)\text{Aff}(f([a,b])$. Using an affine version of the Picard-Lindelöf operator, we can try to prove the contraction of this operator by computing a post fixed-point over the interval $[t_n, t_{n+1}]$ that is we want to find a value z such that:

$$z \supseteq y_n + [0, h]\text{Aff}\left(f([t_n, t_{n+1}], z)\right) \quad . \tag{11}$$

Note that Equation (11) is associated to an iterative process to compute z. Starting from z_0 being the interval hull of y_n and y_{n+1}, we define the sequence of affine forms z_k as $z_{k+1} = y_n + [0, h]\text{Aff}(f([t_n, t_{n+1}], z_k))$ and stop when we find k such that $z_{k+1} \subseteq z_k$. If we cannot find a post fixed-point in a given fixed number of iterations, this may be the case that the step-size is too large. Then we reject the integration step and keep going the simulation with a reduced step-size (usually $\frac{h_n}{2}$). That is Equation (11) is also used to control the integration step-size.

Furthermore, the value z is also used as an enclosure of the solution of IVP over the time interval $[t_n, t_{n+1}]$. We can hence bound the Lagrange remainder of the true solution with z such that:

$$f^{(p)}\left(\xi, y(\xi; y_0)\right) \in \text{Aff}\left(f^{(p)}\left([t_n, t_{n+1}], z\right)\right) \quad . \tag{12}$$

Finally, using Equation (9) and Equation (12) we can prove Theorem 1 and thus bound the distance between the approximation points of any explicit Runge-Kutta method and any solution of the IVP.

Theorem 1. *Let $S_0 \subseteq \mathbb{R}^n$ be a set of initial states and let y_0 be an affine form such that $S_0 \subseteq y_0$. Let Φ be a numerical integration scheme and Φ_{Aff} be the evaluation of Φ using affine arithmetic. Let (t_n, y_n) be a sequence of time instants and affine forms defined by $y_{n+1} = y'_{n+1} + e_{n+1}$ where $(t_{n+1}, y'_{n+1}) = \Phi_{Aff}(t_n, y_n)$ and e_{n+1} is the truncation error as defined by Equation (8) and is evaluated using Equations (9) and (12). Then, we have that $\forall y'_0 \in S_0 \colon \forall n \in \mathbb{N},\ y(t_n; y'_0) \in y_n$.*

Example 5. We present the main steps of our method on the system $\dot{x} = x^2$ solved with Heun's method: $x_{n+1} = x_n + h/2(x_n^2 + (x_n + hx_n^2)^2)$. First, if $x(t_n) \in \hat{x}_n$, we let

$\hat{x}_{n+1} = \hat{x}_n + h/2(\hat{x}_n^2 + (\hat{x}_n + h\hat{x}_n^2)^2)$, evaluated using affine arithmetic. Next we bound the truncation error $\hat{x}_{n+1} - x(t_{n+1})$. Heun's method being of order 2 we need to bound over $[t_n, t_{n+1}]$ the third derivative of the problem and the third derivative of the method w.r.t. time, i.e. we want to bound the expressions $\dddot{x} = 6x^4$ and $\phi^{(3)}(t) = 3x_n^4$ with $\phi(t) = x_n + (t - t_n)/2(x_n^2 + (x_n + (t - t_n)x_n^2))$. For the latter, we bound it with $3\hat{x}_n^4$ using affine arithmetic. For the former, we must bound x on the whole interval $[t_n, t_{n+1}]$ into an affine form $\hat{z}$ and then we use $6\hat{z}^4$ as a bound of $\dddot{x}$. To compute $\hat{z}$, we use Equation (11) and iteratively compute a post-fixpoint of $\hat{z} = \hat{x}_n + [0, h]\hat{z}^2$. We start from the hull of $\hat{x}_n$ and $\hat{x}_{n+1}$ and evaluate the expression $\hat{x}_n + [0, h]\hat{z}^2$ using affine arithmetic until we reach a post-fixpoint.

3.3 Step-Size Strategy

Our method automatically adapts the step-size in order to validate the existence of the solution and improve the stability of the computed enclosure.

The iteration defined by Equation (11) successively computes sets z_k until $z_{k+1} = y_n + [0, h]\mathrm{Aff}(f([t_n, t_{n+1}], z_k)) \subseteq z_k$. At this point, we know that IVP (1) has a solution on $[t_n, t_{n+1}]$ and that this solution remains in z_k. As we assumed that the IVP has a solution, there exists some $h > 0$ such that Picard iteration converges. However, given some h, the iteration may diverge or take too long to converge. So we fix a maximal number of iterations K, and if we do not converge after K steps, the step is rejected and we set the step-size to $h/2$. As we start from a good approximation of the fixpoint (the hull of the enclosure at t_n and the numerical approximation at t_{n+1}), the iteration generally converges quickly.

Then, we let the user define two values, the absolute tolerance *atol* and the relative tolerance *rtol* that defines the acceptable error at each integration step. More formally, our method computes for each instant t_n an error e_n which is the distance between the true solution and the numerical approximation. We say that the step from t_n to t_{n+1} is accepted if e_n is such that: $\sup(e_n) \leq \texttt{err}$ where $\texttt{err} = \max(\mathrm{atol}, \mathrm{rtol} \times \max(\sup(y_{n+1}), \sup(y_n)))$. If the step is not accepted, we set the step-size to $h/2$ and restart from t_n. If the step is accepted, the next step-size is $h' = h(\mathrm{rtol}/\mathrm{err})^{1/(q+1)}$, q being the order of the numerical method. So, the step-size is automatically adapted so that the error introduced at each step converges towards the user-defined tolerances. This is similar to what is done for variable step-size numerical methods, except that we now use the guaranteed error to accept and control the step-sizes. Note that we also transform fixed step-size methods such as Euler or RK4 methods into variable step-size algorithms.

Finally, our implementation also offers another algorithm to adapt the step-size: we use the PI algorithm from [15]. The main idea is the use a *proportional-integral* controller scheme to adapt h to achieve the desired error `err`. This algorithm makes the step-size more stable and thus reduces the number of rejected steps due to the Picard-Lindelöf iteration and in our experiments it showed to be the best choice for controlling the step-size.

4 Experiments

In this section we present the effectiveness of our approach to make every explicit Runge-Kutta method guaranteed through different examples mainly coming from the DETEST problem set [16]. This set has been specifically defined to test numerical integration methods on various kinds of problems classified according linear/non-linear and non-stiff/stiff categories. We compare our approach against the VNODE (VNODE-LP version 0.3) software which implements the state of the art of guaranteed numerical integration methods based on interval Taylor series [14]. Despite VNODE can handle high order Taylor series we restrict our comparison to order 4 which is the highest order of the Runge-Kutta methods we consider in this article. We use the following integration methods: Euler, Heun, Runge-Kutta 4 (RK4), Bogacki-Shampine (ode23), Dormand-Prince (ode45) [10].

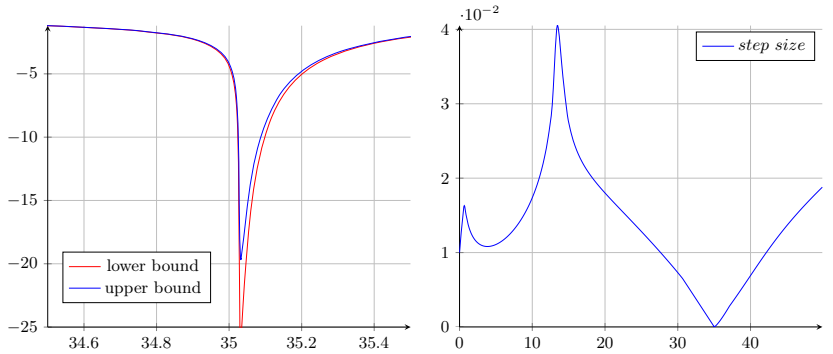
All the simulations were executed on a desktop (two 2.33GHz processors with 2Go of RAM) running Fedora Linux. The implementation was done in OCaml using the GiNaC library [17] to symbolically compute derivatives. In the following tables, we present: T the time (in seconds) required to simulate the problem excluding time spent to compute derivatives and TT the total time (in seconds) including time used to compute and compile derivatives[1]; $Tol = rtol = atol$ the chosen tolerance; Rej and Acc are respectively the number of rejected and accepted steps; $Evals$ is the number of function evaluations and $Prec$ is the precision, taken as the greatest width of the guaranteed enclosures calculated.

4.1 Oil-Reservoir Problem

We consider again the "oil-reservoir" problem introduced in Section 2 on which we applied different guaranteed Runge-Kutta methods. In order to give a hint on the kind of stiffness we deal with this example, in Figure 3(a) we give the temporal evolution of the variable z around $t = 35$, where the derivative varies a lot. We see that, even if the precision of the bounds decreases when the stiffness is important, our method is precise enough to make the bounds contract when the dynamics is simpler. In Figure 3(b) we give the step-size evolution of the Heun's method to emphasize the importance of the step-size control mechanism presented in Section 3.3, even for initially fixed step-size integration scheme.

Next, we present in Table 1 the results of the application of different Runge-Kutta methods on the "oil-reservoir" example. Note that VNODE is not able to solve this problem, even with an order of 50: it is not able to go beyond 2 seconds of simulation. We remark in Table 1 that the execution time increases with the chosen tolerance and the complexity of the Runge-Kutta method (number of rows in a Butcher table). Moreover, the precision varies with the chosen tolerances and methods. We recall that the precision taken in this article is the

[1] We distinguish both as in our implementation, we only need to compute the derivatives once, if we want to re-integrate the same problem with other parameters or from another starting point, we do not need to compute the derivatives again.



(a) Bounds for z, for $34.5 \leq t \leq 35.5$ (b) Step size evolution

Fig. 3. Oil-reservoir with guaranteed Heun's method

Table 1. Simulation results on "oil-reservoir" problem

Meth	Tol	Acc	Rej	Evals	T	TT	Prec	Prec(t_{end})
Heun	10^{-6}	2566	2561	10254	1.099	4.322	2.791	$4.541 \cdot 10^{-2}$
	10^{-9}	17626	36373	107998	8.878	12.101	$1.438 \cdot 10^{-2}$	$3.971 \cdot 10^{-3}$
	10^{-12}	220092	665081	1770346	141.848	145.071	$7.579 \cdot 10^{-5}$	$7.579 \cdot 10^{-5}$
ode23	10^{-6}	2453	2449	14706	4.833	10.713	5.412	$7.063 \cdot 10^{-2}$
	10^{-9}	8320	16633	74859	23.015	26.538	0.107	$1.891 \cdot 10^{-2}$
	10^{-12}	45495	113940	478305	132.578	136.101	$6.996 \cdot 10^{-4}$	$4.000 \cdot 10^{-4}$
RK4	10^{-6}	604	481	4340	0.909	38.646	1.413	$4.824 \cdot 10^{-2}$
	10^{-9}	1553	2031	14336	2.778	40.514	$1.368 \cdot 10^{-2}$	$3.061 \cdot 10^{-3}$
	10^{-12}	7224	14441	86660	15.409	53.145	$3.683 \cdot 10^{-5}$	$3.683 \cdot 10^{-5}$
ode45	10^{-6}	1163	1177	16380	15.791	5653.939	7.772	$1.729 \cdot 10^{-1}$
	10^{-9}	1772	2316	28616	26.046	5642.939	1.002	$6.619 \cdot 10^{-2}$
	10^{-12}	7669	15330	160993	114.609	5731.502	$2.787 \cdot 10^{-5}$	$2.787 \cdot 10^{-5}$

greatest width of the guaranteed enclosures computed during the simulation. In this example, the greatest width is computed around $t = 35$. In Table 1 the last column gives the width of the solution enclosure at the end of the simulation, which shows that we obtain precise results at $t = 50$ even if locally the error increases.

4.2 Non-stiff Problems

DETEST Problem A3. We study the behaviors of the Heun's method and the RK4 method on the following problem, for a simulation time $t \in [0, 20]$:

$$\dot{y} = y \cos(t) \quad \text{with} \quad y(0) = 1 \ . \tag{13}$$

More precisely, we emphasize the importance of the choice of the numerical methods in the trade-off of efficiency and precision even for this simple example.

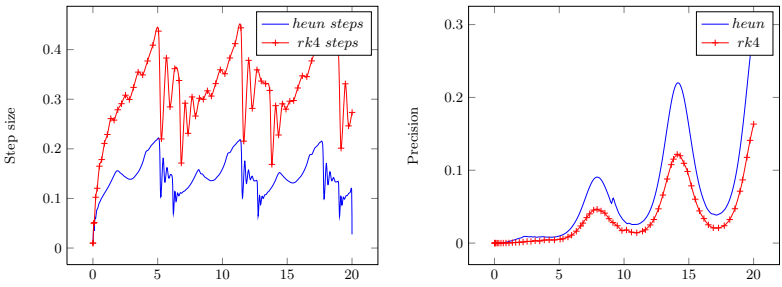


Fig. 4. Step-size (left) and precision (right) evolution for Problem A3 (Tol=10^{-3})

For this example, Heun's method and RK4 method can solve Equation (13) more efficiently than variable step-size methods as Bogacki-Shampine without loosing too much precision. Figure 4, left, shows the step-size evolution of these two methods. Note that the length of the step-size adapts to the dynamics of the problem. Furthermore, the steps chosen by RK4 method are about four times wider than those taken by the Heun's method. This is explained by the order of the method used (see Section 2). Furthermore, the precision evolution of the two methods depicted in Figure 4, right, shows that RK4 method offers more precise enclosures than Heun's method.

DETEST Problem C3. In this case study we only consider RK4 method to show the scalability of our approach. We solve for $t \in [0, 2]$ and various values of n the problem defined by:

$$\begin{pmatrix} \dot{y}_1 \\ \dot{y}_2 \\ \dot{y}_3 \\ \vdots \\ \dot{y}_n \end{pmatrix} = \begin{pmatrix} -2 & 1 & 0 & 0 & \cdots & 0 \\ 1 & -2 & 1 & 0 & \cdots & 0 \\ 0 & 1 & -2 & 1 & \cdots & 0 \\ & & & \ddots & & \\ 0 & 0 & \cdots & 0 & 1 & -2 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_n \end{pmatrix} \quad \text{with} \quad y(0) = \begin{pmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix} . \tag{14}$$

In particular, we solved this problem with $n = \{40, 80, 120, 140\}$. Table 2 shows the time spent in simulation using RK4 method and the precision generated for each dimension and each tolerance considered. Compared to VNODE with order 4 results of the RK4 method exhibits a linear time complexity, while VNODE spends much more time to solve it. Indeed, VNODE uses standard interval arithmetic so to limit the wrapping effect during the simulation it uses a technique based on QR matrix decomposition (see [14]) which has a $O(n^3)$ complexity. Our solution to fight the wrapping effect is the use affine arithmetic which prevents the use of QR matrix decomposition then we have a better scalability without losing to much precision. Note also that with high order, e.g. 40, VNODE is able to solve this problem with only one integration step.

Table 2. Results for Problem C3

Dim	Tol	T	Prec	T VNODE	Prec VNODE
40	10^{-3}	0.378	$7.381 \cdot 10^{-4}$	0.9	$4.404 \cdot 10^{-4}$
	10^{-6}	1.064	$1.284 \cdot 10^{-5}$	7.46	$2.175 \cdot 10^{-07}$
	10^{-9}	4.628	$2.530 \cdot 10^{-8}$	72.28	$3.517 \cdot 10^{-11}$
80	10^{-3}	0.959	$1.886 \cdot 10^{-3}$	6.92	$4.404 \cdot 10^{-4}$
	10^{-6}	2.551	$1.432 \cdot 10^{-5}$	58.89	$2.175 \cdot 10^{-07}$
	10^{-9}	10.641	$2.295 \cdot 10^{-8}$	565.57	$3.517 \cdot 10^{-11}$
120	10^{-3}	1.49	$1.753 \cdot 10^{-3}$	23.21	$4.404 \cdot 10^{-4}$
	10^{-6}	4.297	$1.386 \cdot 10^{-5}$	196.56	$2.175 \cdot 10^{-07}$
	10^{-9}	17.782	$2.446 \cdot 10^{-8}$	2314.46	$3.517 \cdot 10^{-11}$
140	10^{-3}	1.846	$1.137 \cdot 10^{-3}$	37.43	$4.404 \cdot 10^{-4}$
	10^{-6}	5.285	$1.440 \cdot 10^{-5}$	334.08	$2.175 \cdot 10^{-07}$
	10^{-9}	22.286	$2.710 \cdot 10^{-8}$	3904.96	$3.517 \cdot 10^{-11}$

Table 3. Results on Problem E2

Meth	Tol	Rej	Acc	Evals	T	Prec
Heun	10^{-3}	38	11	98	0.013	$6.451 \cdot 10^{-4}$
	10^{-6}	130	129	518	0.036	$2.073 \cdot 10^{-5}$
	10^{-9}	1047	2092	6278	0.354	$8.060 \cdot 10^{-8}$
ode23	10^{-3}	36	9	135	0.046	$1.369 \cdot 10^{-3}$
	10^{-6}	99	113	636	0.156	$3.630 \cdot 10^{-5}$
	10^{-9}	653	1580	6699	1.329	$1.513 \cdot 10^{-7}$
RK4	10^{-3}	36	18	216	0.071	$5.693 \cdot 10^{-5}$
	10^{-6}	48	34	328	0.106	$7.538 \cdot 10^{-6}$
	10^{-9}	134	171	1220	0.371	$1.592 \cdot 10^{-8}$
VNODE	10^{-3}	–	–	–	0	$1.278 \cdot 10^{-4}$
	10^{-6}	–	–	–	0.02	$2.554 \cdot 10^{-07}$
	10^{-9}	–	–	–	0.18	$2.623 \cdot 10^{-10}$

4.3 Stiff Problem

We consider for a simulation time $t \in [0, 1]$ the DETEST Problem E2 defined by:

$$\begin{cases} \dot{y}_1 = y_1 \\ \dot{y}_2 = 5(1 - y_1^2)y_2 - y_1 \end{cases} \quad \text{with} \quad \begin{cases} y_1(0) = 2 \\ y_2(0) = 0 \end{cases} .$$

We look at the behaviors of different explicit Runge-Kutta methods which are known to behave not very well on such kind of problems. Table 3 gives the result on this example. For the result of VNODE, we did not succeed to access the information associated to columns Rej, Acc and Evals. Nevertheless, we note that VNODE at order 4 is more efficient and precise in this example than Runge-Kutta methods which already behave well.

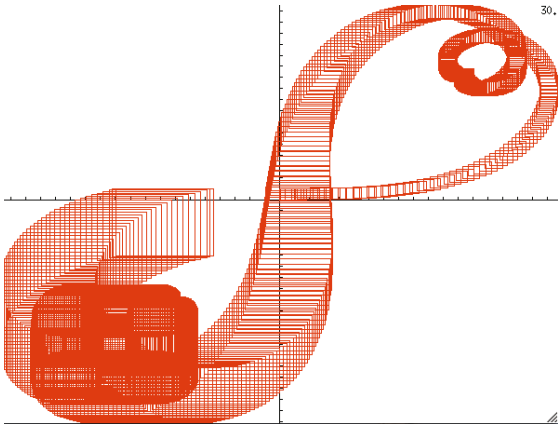


Fig. 5. Temporal evolution in (x, y)-space of the car ODE

4.4 Problem with Uncertainties

Finally, we show an example of a highly non-linear IVP, representing the movement of a car in 2D space) with some initial uncertainty:

$$\dot{x} = v\cos(\delta)\cos(\theta) \quad \dot{y} = v\cos(\delta)\sin(\theta) \quad \dot{\theta} = 0.2v\sin(0.2t)\ .$$

We integrate it up to $t = 30$ with the initial values $x(0) \in [0, 1],\ y(0) \in [0, 1], \theta(0) = 0$ and $v \in [7, 7.1]$. Figure 5 shows the evolution of the bounds on x and y with time. This was computed using the RK4 method, with a tolerance of 10^{-8}, in $15.6s$, with $v \in [7, 7.1]$. We hence remark that our approach is efficient and robust enough to handle uncertainties.

5 Conclusion

In this article, we presented a novel method to compute guaranteed bounds on the solution of differential equations. This method is an extension of the previous work of one of the authors [15]. The main advantages of this work is that it may use various numerical methods to obtain the guaranteed bounds, so that we can treat different kinds of equations (stiff or not, linear or not). Moreover, as we use affine forms in order to enclose sets of values, we avoid the well known wrapping effect which is present in [14]. This results in a more precise and more effective method as we can make larger step-sizes. Remark that our tool computes both over-approximations at discrete time stamps t_n but also, using the Picard-Lindelöf operator, over-approximations over each intervals $[t_n, t_{n+1}]$.

To compute such over-approximations, various other tools exist. Developed for the verification of hybrid systems, SpaceEx [3] handles linear differential equations exactly using support functions and matrix exponentiation. However, when facing non-linear equations, a hybridization [5] must be performed, which

can end in an explosion of the number of discrete states if the equation is stiff. Compared to tools such as VNODE [14], ValenciaIVP [18] or [19], our method relies on well-known numerical methods and can thus treat more differential equations. For example, VNODE could not integrate the "oil-reservoir" problem.

As should be clear from our experimentation, the fact that we can use various numerical methods is very interesting as each method is well adapted to a specific kind of problems. So we are confident that by adding more and more methods to our framework we will have a large enough collection to handle most kinds of problems. Three challenges arise towards this goal. First, we want to handle implicit methods in which y_{n+1} is defined via a fixpoint equation. These methods are more stable than explicit ones and thus handle better stiff systems and allow for larger step-sizes. Second, we will investigate multi-step methods that use $y_n,\ y_{n-1}, \ldots,\ y_{n-k}$ to compute y_{n+1} for some $k > 0$. Such methods are more efficient than single-step methods as they require less evaluation of f, however bounding the error is much more complicated. Finally, we will study variable order methods as in [8]. Such methods embed in one Butcher table various methods with different orders. Then, at each step, the best method is automatically chosen. This method would allow us to efficiently change the order during the integration process.

References

1. Guéguen, H., Lefebvre, M.A., Zaytoon, J., Nasri, O.: Safety verification and reachability analysis for hybrid systems. Annual Reviews in Control 33(1), 25–36 (2009)
2. Alur, R.: Formal verification of hybrid systems. In: Conference on Embedded Software, pp. 273–278. ACM (2011)
3. Frehse, G., Le Guernic, C., Donzé, A., Cotton, S., Ray, R., Lebeltel, O., Ripado, R., Girard, A., Dang, T., Maler, O.: SpaceEx: Scalable verification of hybrid systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 379–395. Springer, Heidelberg (2011)
4. Asarin, E., Dang, T., Girard, A.: Hybridization methods for the analysis of nonlinear systems. Acta Inf. 43(7), 451–476 (2007)
5. Dang, T., Maler, O., Testylier, R.: Accurate hybridization of nonlinear systems. In: Hybrid Systems: Computation and Control, pp. 11–20. ACM (2010)
6. Shenoy, R., McKay, B., Mosterman, P.J.: On simulation of simulink models for model-based design. Handbook of Dynamic System Modeling (2007)
7. Conrad, M., Mosterman, P.J.: Model-based design using Simulink modeling, code generation, verification, and validation. Formal Methods: Industrial Use from Model to the Code, 159–178 (2012)
8. Cash, J.R., Karp, A.H.: A variable order Runge-Kutta method for ivp with rapidly varying right-hand sides. ACM Trans. Math. Softw. 16(3), 201–222 (1990)
9. Hairer, E., Norsett, S.P., Wanner, G.: Solving Ordinary Differential Equations I: Nonstiff Problems, 2nd edn. Springer (2009)
10. Shampine, L.F., Gladwell, I., Thompson, S.: Solving ODEs with MATLAB. Cambridge Univ. Press (2003)
11. Moore, R.: Interval Analysis. Prentice Hall (1966)
12. de Figueiredo, L.H., Stolfi, J.: Self-Validated Numerical Methods and Applications. Brazilian Mathematics Colloquium monographs. IMPA/CNPq (1997)

13. Muller, J.M., Brisebarre, N., De Dinechin, F., Jeannerod, C.P., Lefèvre, V., Melquiond, G., Revol, N., Stehlé, D., Torres, S.: Handbook of Floating-Point Arithmetic. Birkhauser (2009)
14. Nedialkov, N., Jackson, K., Corliss, G.: Validated solutions of initial value problems for ordinary differential equations. Appl. Math. and Comp. 105(1), 21–68 (1999)
15. Bouissou, O., Martel, M.: GRKLib: a Guaranteed Runge Kutta Library. In: Scientific Computing, Computer Arithmetic and Validated Numerics (2006)
16. Enright, W.H., Pryce, J.D.: Two FORTRAN packages for assessing initial value methods. ACM Transations on Mathematical Software 13(1), 1–27 (1987)
17. Bauer, C., Frink, A., Kreckel, R.: Introduction to the GiNaC framework for symbolic computation within the C++ programming language. J. Symb. Comput. 33(1), 1–12 (2002)
18. Rauh, A., Brill, M., Günther, C.: A novel interval arithmetic approach for solving differential-algebraic equations with ValEncIA-IVP. Int. J. Appl. Math. Comput. Sci. 19(3), 381–397 (2009)
19. Combastel, C.: A state bounding observer for uncertain non-linear continuous-time systems based on zonotopes. In: Conference on Decision and Control. IEEE (2005)