Enclosing solutions of systems of equations involving ODE

Aurelien Lejeune

National Institute of Informatics
2-1-2 Hitotsubashi, Chyoda-ku
Tokyo 101-8430 – Japan
aurelien.lejeune@etu.univ-nantes.fr

Abstract. In engineering applications, proving the existence of a solution in a constraint problem involving a dynamic system is a central issue, and because many of these applications are critical, developing rigorous methods to simulate those systems is also a central issue. This report presents on the one hand a tool allowing the rigorous simulation of a dynamic system inside the Mathematica environment. On the other hand it provides a new method to enclose the discrete changes in a hybrid dynamic system based on the interval arithmetic.

Keywords: hybrid systems, ordinary differential equations, interval arithmetic.

Resumé : Prouver l'existence de solutions d'un problème sous contraintes impliquant un système dynamique est une problématique centrale en ingéniérie et en analyse numérique plus particulièrement, en effet de nombreuses applications critiques se basent sur des modèles dynamiques. Ce document presentera d'une part un outil permettant la simulation rigoureuse de systèmes dynamiques pour Mathematica, et une nouvelle méthode pour l'encadrement des changements discrets des systèmes hybrides d'autre part.

Mots-clefs : systèmes hybrides, équations differentielles ordinaires, analyse par intervalles.

1 Introduction

Over the last few years, interval arithmetic was developed to provide a rigorous enclosure of the solution of a numerical system. It is especially interesting for engineering in critical systems where the enclosure of a solution is more important than obtaining quickly an approximation. One of the limitations when using interval arithmetic is that an initial box, where the solution is supposed to be, is needed.

This document is divided into two main parts, in the first one we present a plug-in to manipulate and solve rigorously an ODE involving intervals. This plug-in is written to be used under the *Mathematica* environment. This plug-in is based on a famous ODE solver library, *VNODE*. The plug-in is also written to be user friendly for someone accustomed to using *Mathematica*. We describe briefly its implementation, then we present some examples of applications.

In the second part we present a new method for solving systems involving ordinary differential equations which does not need this initial box. The aim of the method is to determine if there exists a solution in the neighbourhood of an initial approximate solution and to rigorously compute its enclosure.

1.1 Preliminaries

Notations Vectors are denoted by boldface symbols, matrices by capital letters. Interval vectors and matrices by bracketed symbols.

Results notations The interval values in the result are denoted with two notation. A classical notation (e.g. $[1.845878548, 2.88745783]$) and a shorter notation for interval with a small range like $[1.444443, 1.444449]$ will be denoted $1.44444[3, 9]$; for intervals like $[1.99999, 2.00001]$ the convention is $1.99999[9, 11]$; for negative intervals such as $[-2.00001, -1.99999]$ it is $-1.99999[9, 11]$.

To improve the comparison between numerical approximate values and the interval enclosure, we have underlined the common part, v.gr. if we want to compare 1.8756847 and the enclosure $1.875679[9, 13]$, the approximation is denoted $\underline{1.8756}847$.

Interval arithmetics In the document some interval functions are used:

- $Mid([x])$ represents the midpoint of the interval $[x] = [\underline{x}, \overline{x}]$ defined by

$$Mid([x]) = \frac{1}{2}(\underline{x} + \overline{x})$$

- $int([x])$ is the interior of the interval $[x]$[1]:

$$int([x]) =]\underline{x}, \overline{x}[$$

- $Mid(\mathbf{x})$ and $int(\mathbf{x})$ are the vector extensions of those functions.

[1] The ISO 31-11 notation is used to avoid confusion with vector notation.

Ordinary differential equations Ordinary differential equations ODE are equations denoted by:

$$\mathbf{f}(t, \mathbf{x}(t), \mathbf{x}'(t), \mathbf{x}"(t), \ldots, \mathbf{x}^{(n-1)}(t)) = \mathbf{x}^n(t)$$

where n is the order of the ODE and $\mathbf{f}$ is a known function. In this document we only use first order ODE (every ODE can be expressed as a first order ODE by adding variables). $\Phi(\mathbf{x}_0, t)$, called the solution operator, is the solution of an initial value problem of an ODE at moment t and with the initial condition $\mathbf{x}_0$ (ie. $\Phi(\mathbf{x}_0, t) = \mathbf{x}(t)$).More information about ODE can be found in *An Introduction to Ordinary Differential Equations*[16].

2 Interval ODE Solver for Mathematica

Mathematica[12] is a useful computational software which allows the use of many different advanced mathematical features in the same environment (such as linear algebra, symbolic computation, numerical analysis, and in a more interesting way for us, interval arithmetic). Additionally it offers the possibility to display graphics with advanced parameters allowing an improved visualization of some results. Nevertheless it is still not possible to perform easily rigorous numerical simulation of dynamic systems described as *ordinary differential equations* (ODE hereafter).

In this part we will focus on the development of a Mathematica plug-in named *Mathematica IDSolve*[14] [2]. *Mathematica IDSolve* adds the possibility to enclose rigorously the solution of ODEs within the Mathematica environment. *Mathematica IDSolve* is based on the *VDNODE* library[13][15] written by Ned Nedialkov.

2.1 Existing work

A subroutine, directly implemented in Mathematica, called NDSolve can numerically solve an ODE. It takes as input an ODE such as:

$$\mathbf{x}'(t) = \mathbf{f}(\mathbf{x}(t)) \tag{1}$$

with the initial conditions:

$$\mathbf{x}(0) = \mathbf{x_0} \tag{2}$$

and a time interval $[\tau]$, and returns an interpolation of the solution:

$$\tilde{\mathbf{\Phi}}(\mathbf{x_0}, t) : [\tau] \longrightarrow \mathbb{R}^n \tag{3}$$

which approximates

$$\mathbf{\Phi}(\mathbf{x_0}, t) : [\tau] \longrightarrow \mathbb{R}^n \tag{4}$$

In the last section of this chapter we compare the results of our plug-in *Mathematica IDSolve* with the approximation of $\mathbf{x}(t)$ given by NDSolve.

Besides *VNODE*, a C++ library developed since 2001 by Ned Nedialkov[7], allows the rigorous enclosure of $\mathbf{x}(t)$. *VNODE* is designed to be user friendly: the user just has to give the expression of the ODE, the initial conditions and the value of t, and *VDNODE* returns the enclosure of the solution (if such a solution exists) as an interval vector $[\mathbf{x}]$ containing $\{\mathbf{\Phi}(\mathbf{x_0}, t) : \mathbf{x_0} \in [\mathbf{x_0}], t \in [t]\}$

Our main approach will be to design a plug-in which interfaces Mathematica and *VDNODE* in order to use the rigorous enclosure of ODE directly under the Mathematica environment.

[2] The name IDSolve was chosen to match the NDSolve Mathematica function dedicated to approximate numerical simulation of ODE.

2.2 Design and features

The plug-in has been developed with the following specifications:

- The signature of the subroutine IDSolve should be as close as possible of the NDSolve signature,
- The plug-in has to be compatible with the standard version of *VDNODE*[3],
- The plug-in is designed with the following communication scheme:

$$\text{Mathematica} \overset{\text{Send ODE, } \mathbf{x_0},\, t}{} \text{VNODE} \overset{\mathbf{\Phi_{x_0}}(t)}{} \text{Mathematica}$$

MathLink

MathLink is a C library which allows the call of C/C++ program in Mathematica environment, and give it as input different classes of objects like integers, reals, list, character strings or directly Mathematica objects. Because of the possibility to use Mathematica objects in our library we can have a very close representation of our objects between Mathematica and our plug-in. We have also considered two other possibilities. Because *VNODE* needs the expression of the ODE to be hard coded into the sources, we need to recompile different versions for each problem. We considered using a temporary C++ file which describes the ODE as a C++ function, compile it on the fly as a dynamic library and use it in *VNODE*. The main issues with this solution is that the compiler of *VNODE* has to be the same as the compiler of the temporary file, and compilation is a very long process. The second possibility was to parse strings but in this case we need to write a specific parser for intervals to build the expression trees. MathLink directly builds those expression trees from their Mathematica representations so this is why we chose this option.

The output

In NDSolve the output of the subroutine is an interpolated function, but we cannot use such an interpolated function in Mathematica IDSolve without losing the rigor of the enclosure. Therefore, we return an enclosure of the solution at t_{end} instead.

The preprocessing

To avoid an improper usage of the C++ plug-in, we have implemented a preprocessing subroutine in the Mathematica language, which quickly checks the consistency of the problem (for example if the initial conditions are not well declared). The preprocessing adds the possibility to return explicit error messages to the user.

[3] Up to now the latest version of the *VDNODE* library is *VNODE-LP 0.3*.

Usage

The signature of the IDSolve subroutine is :

$$\mathsf{IDSolve}[[\mathbf{f}]([\mathbf{x}](t)) \cup [\mathbf{x}](t_0), [\mathbf{x}], \{t, t_0, t_{end}\}] \longrightarrow \mathbf{\Phi}(\mathbf{x}(t_0), t_{end})$$

For comparison the signature of NDSolve is :

$$\mathsf{IDSolve}[\mathbf{f}(\mathbf{x}(t)) \cup \mathbf{x}(t_0), \mathbf{x}, \{t, t_0, t_{end}\}] \longrightarrow \tilde{\mathbf{\Phi}}(\mathbf{x}(t_0), \cdot) : [t_0, t_{end}] \longrightarrow \mathbb{R}^n$$

For instance, if we consider this problem[4]:

$$\begin{cases} x'(t) = v(t) \\ v'(t) = -g \end{cases} \tag{5}$$

with the initial conditions $(x(0),\ v(0)) = (381,\ 0)$ and the parameter $g = 9.81$, and if we want the enclosure of the solution at $t = 5$, we have to write[5]:

```
IDSolve[{x'[t] == v[t], v'[t] == -9.81, x[0] == 381, v[0] == 0},
        {x, v}, {t, 0, 5}]
```

and Mathematica returns[6]:

```
{Interval[{258.375, 258.375}], Interval[{-49.05, -49.05}]}.
```

We can use intervals as initial conditions like $(x(0),\ v(0)) = ([379.986,\ 381.648],\ [0,\ 0.0001])$ and an interval as parameter $g = [9.81,\ 9.82]$, by writing:

```
IDSolve[{x'[t] == v[t], v'[t] == -Interval[{9.81, 9.82}],
         x[0] == Interval[{379.986, 381.648}],
         v[0] == Interval[{0, 0.0001}]},
         {x, v}, {t, 0, 5}]
```

and the result shown is:

```
{Interval[{257.234, 259.026}], Interval[{-49.1, -49.0499}]}.
```

If we want the solution within an interval of time like $t = [5,\ 6]$:

```
IDSolve[{x'[t] == v[t], v'[t] == -9.81, x[0] == 381, v[0] == 0},
        {x, v}, {t, 0, Interval[{5, 6}]}]
```

and the result shown is:

```
{Interval[{204.42, 258.375}], Interval[{-58.86, -49.05}]}.
```

[4] This simple problem represents the fall of an object from the top of the Empire State Building (the air drag is neglected).

[5] Be careful before processing to follow the installation procedure given in the appendix

[6] The number of significant figure is reduced for the visualization.

Limits

Up to now there is some limitations when using IDSolve instead of NDSolve. One of those limitations is the fact that IDSolve uses double precision floating point numbers to represent the interval whereas Mathematica can represent floats with a better precision. Not being able to get the derivatives with respect to the initial conditions is also a limitation. In addition, *VNODE* uses parallelotopes which is less efficient than more complex sets (like polytopes with multiple faces[10], or curving faces[11]). We decided to use *VNODE* it is the only stable open source software, furthermore it is widely used by academics. Finally we can obtain a precise enclosure of the final time and a crude enclosure for the time interval, while one may want to have an accurate enclosure of every time in the simulation. Avoiding this issue can be done by using VNODE on subintervals but the final enclosure could be worst.

2.3 Experimentations

In this section, we present some results obtained using IDSolve and NDSolve to compare the results of different systems of increasing complexity.

A simple ODE

In this example we solve a very simple problem :

$$\begin{cases} y'(t) = \lambda y(t) \\ y(0) = 1 \end{cases} \tag{6}$$

with : $\lambda = -1$ and $t = 1$. The exact result of this system is e^{-1}.

NDSolve	0.36787945780186754
IDSolve	$0.367879441171442[10, 56]$
e^{-1}	0.36787944117144233

As we can see the IDSolve solution encloses the approximation of e^{-1}, whereas the approximation of the ODE solution by NDSolve is out of the range of the rigorous enclosure.

Falling ball system in a Euclidean plane

In this example IDSolve is tested with an easy mechanics problem, we simulate the free fall of a ball (the ball is considered as a single point). The ODE which describes this phenomena is :

$$\begin{cases} x_1'(t) = v_1(t) \\ x_2'(t) = v_2(t) \\ v_1'(t) = 0 \\ v_2'(t) = -g \end{cases} \tag{7}$$

With gravity $g = 9.81$ and the following initial conditions :

$$\begin{cases} x_1(0) = 0 \\ x_2(0) = 5 \\ v_1(0) = 2 \\ v_2(0) = 0 \end{cases} \tag{8}$$

The problem is solved at $t = 1$

$$\text{NDSolve} \quad \begin{pmatrix} \underline{1.9999999999999998} \\ \underline{0.0949999893696859} \\ 2.0 \\ \underline{-9.809999999999999} \end{pmatrix} \tag{9}$$

$$\text{IDSolve} \quad \begin{pmatrix} 1.999999999999999[6,14] \\ 0.0949999999999997[4,7] \\ 1.999999999999999[6,14] \\ -9.80999999999999[9,11] \end{pmatrix} \tag{10}$$

$$\text{Formal} \quad \begin{pmatrix} 2 \\ \underline{0.0949999999999975} \\ 2 \\ \underline{-9.81} \end{pmatrix} \tag{11}$$

In this example the approximation computed by NDSolve is out of the range of the rigorous enclosure too.

Lorenz system

As a last example we solve a chaotic system, the Lorenz system. For this ODE there is no explicit expression of $\mathbf{x}(t)$. A simulation of the following system is shown on Figure 1. The ODE is defined by :

$$\begin{cases} x'(t) = & \sigma \times (y(t) - x(t)) \\ y'(t) = & x(t) \times (\rho - z(t)) - y(t) \\ z'(t) = & x(t) \times y(t) - \beta\ z(t) \end{cases} \tag{12}$$

With the parameters $(\sigma,\ \beta,\ \rho) = (10,\ \frac{8}{3},\ 28)$ and the initial conditions $(x(0),\ y(0),\ z(0)) = (1,\ 1,\ 1)$.

Solving the system in the interval $t \in [0, 50]$ by NDSolve gives the results shown on Figure 1. The resolution with IDSolve has failed because the enclosure range was too big for VNODE. An upper bound $t = 32.9$ has been computed to be an approximation of the limit. Therefore we compare the results at $t = 32.85$

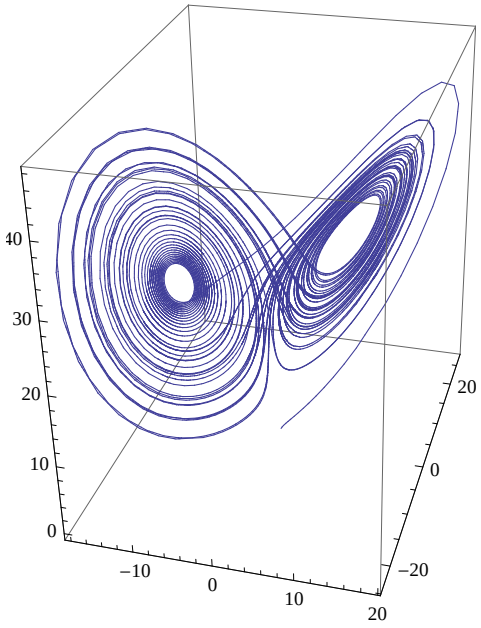

Fig. 1. Projection of the Lorenz attractor

$$\text{NDSolve} \quad \begin{pmatrix} 4.1540680321294445 \\ 1.648428551076282 \\ 25.972107496866006 \end{pmatrix} \tag{13}$$

$$\text{IDSolve} \quad \begin{pmatrix} [\ -57.07243398595995,\ 79.82067089488758\ ] \\ [-253.41013266123872,\ 242.97953722353984] \\ [\ -210.3477603673299,\ 293.065017639242\ ] \end{pmatrix} \tag{14}$$

On this example the width of the enclosure quickly gets so big that it becomes useless. The figures 2 and 3 shows multiple iterations of IDSolve with the same problem to have a better visualization of this effect.

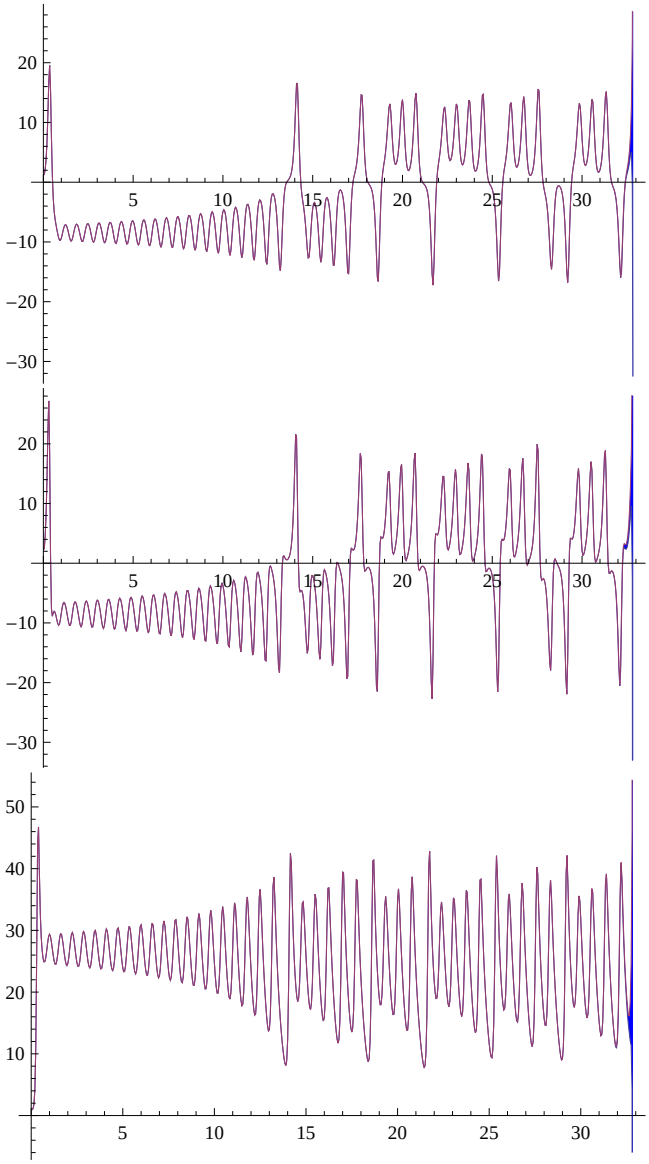

Fig. 2. The three graphics represent respectively the values of $x(t)$, $y(t)$, $z(t)$. We can see the explosion of the enclosure width (the dark area representing the width)

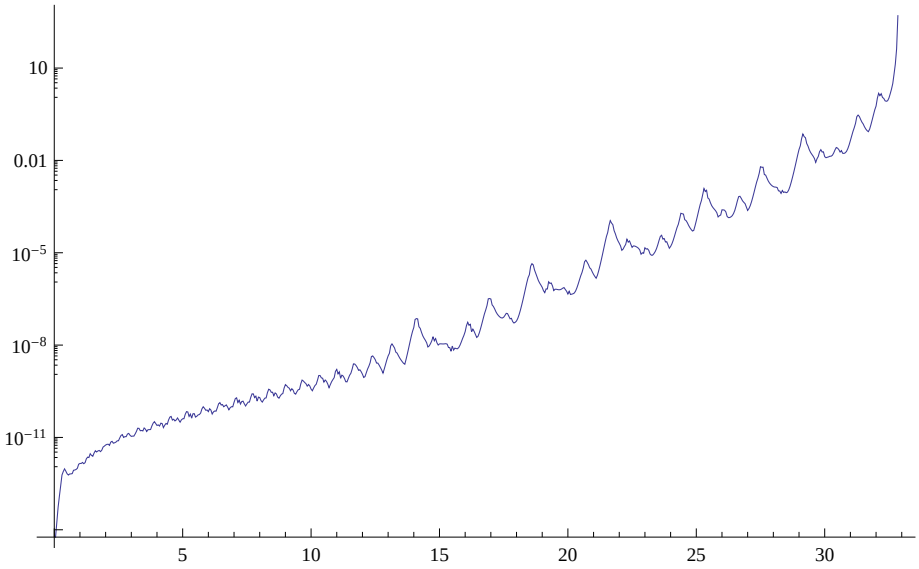

Fig. 3. Explosion of the enclosure. The x coordinate representing the time and the y coordinate the maximum width of the enclosure with regard to the $x(t),\ y(t),\ z(t)$

Performance

In this part we compare the performance with regard to the computation time when using IDSolve and NDSolve. These times are expressed in seconds.

	NDSolve	IDSolve
e^{-1}	0.003125	0.1062500
Falling Ball	0.003125	0.0812500
Lorenz	0.0375000	2.9968750

Using IDSolve is slower than NDSolve for several reasons:

- interval analysis is intrinsically slower than classical numerical analysis.
- IDSolve is not a Mathematica built-in subroutine as opposed to NDSolve.
- to have less differences as possible with the NDSolve signature, we have made some compromises which can decrease the performances when integrating repeatedly the same ODE.

2.4 Limitations and perspectives

In conclusion Mathematica IDSolve is generally suitable to compute the enclosure of an interval valued ODE. If a integration over a long time is needed or if the interval initial conditions are too wide, the plug-in cannot give useful results. A possible solution is to split the initial condition vector into smaller boxes and then perform integrations on each of them as initial conditions.

3 Rigorous enclosure of discrete change during a hybrid system simulation

Hybrid dynamic systems (HDSs) are systems described by a mix of discrete and continuous components. The continuous components are generally expressed by initial valued problems of an ordinary differential equation (IVPs-ODE), whereas when a discrete event occurs the continuous components of the systems change. For more informations about HDSs confer [5]. Up to now many methods used to detect those discrete changes are performed using numerical computation and return incorrect results due to the rounding operations performed.

An HDS can be described by two constraints, one time-dependent constraint:

$$\mathbf{f} : \mathbb{R}^n \longrightarrow \mathbb{R}^{n-1}$$

$$\mathbf{f}(\mathbf{y}(t)) \;=\; \mathbf{y}'(t)$$

and a time-independent guard constraint:

$$\mathbf{g} : \mathbb{R}^n \longrightarrow \mathbb{R}$$

$$\mathbf{g}(\mathbf{x}) \;=\; 0$$

with a bijection between $\mathbf{y}(t)$ and $\mathbf{x}$. Solving this HDS is done by integrating those two constraints together. The method we propose uses a non-linear ODE solver (in our example we use our plug-in *Mathematica IDSolve*) to integrate $\mathbf{f}(\mathbf{y}(t))$. This method employs a modified version of the interval Newton algorithm to guarantee the existence of the solutions.

Our approach is aimed at proving the existence of a solution close to an approximation (computed or intuitive). So we provide no guaranty about the non-existence of a previous solution.

3.1 Related work

A method based on interval arithmetic for simulating HDSs was proposed by Nedialkov and von Mohrenschildt [6]. In 2004 Hickey and Wittenberg proposed a method based on constraint logic programming [8]. Recently another method was published[9]. Like our method, this one is based on the Newton operator but it aims at finding the earliest solution of an HDS. This method also uses a branch and prune algorithm[17] and a wide initial box.

3.2 Our method

Our method uses an approximation of the expected solution to compute an enclosure of the expected solution in the neighborhood of the approximation. Let us review what information is available:

– The expression of the time dependent constraint as an ODE:

$$\mathbf{f}(\mathbf{y}(t)) \;=\; \mathbf{y}'(t)$$

– The guard constraint:

$$\mathbf{g}(\mathbf{x}) \;=\; 0$$

– An approximation of the solution expected: $\tilde{\mathbf{z}}$.

Then we define the following function:

$$\mathbf{h} : \mathbb{R}^n \longrightarrow \mathbb{R}^n$$

$$\mathbf{h}(\mathbf{z}) \;=\; 0$$

where:

$$\mathbf{z} \;=\; (x_1, x_2, \ldots, x_{n-1}, t)$$

and:

$$\begin{cases} h_1(\mathbf{z}) = & \Phi_1(\mathbf{x_0}, z_n) - z_1 \\ h_2(\mathbf{z}) = & \Phi_2(\mathbf{x_0}, z_n) - z_2 \\ \vdots & \\ h_{n-1}(\mathbf{z}) = & \Phi_{n-k}(\mathbf{x_0}, z_n) - z_{n-1} \\ h_n(\mathbf{z}) = & g_k(z_1, \ldots, z_n) \end{cases} \tag{15}$$

The system we have to solve is :

$$\mathbf{h}(\mathbf{z}) - \hat{\mathbf{z}} \;=\; 0 \tag{16}$$

where:

$$\hat{\mathbf{z}} \;\approx\; \tilde{\mathbf{z}}$$

For now we assume that we have of a box $[\mathbf{z^0}]$ such that:

$$\tilde{\mathbf{z}} \in [\mathbf{z^0}]$$

With those hypotheses we can use the classical interval Newton method [2] (algorithm 1).

Algorithm 1: Classical interval Newton method

Input: $[\mathbf{h}] : \mathbb{IR}^n \to \mathbb{IR}^n$, $[\mathbf{z^0}] \subseteq \mathbb{IR}^n$
Output: $[\mathbf{z}] \in \mathbb{IR}^n$

```
success ← false;
[z] ← [z^0];
repeat
    [z*] ← [z];
    [z] ← [InnerStep]([h], [z*]) ∩ [z*];
    if ( [z] ⊆ int[z*] ) then success ← true;
until (stopping criterion not reach) ;
return ([z], success);
```

One of the differencess with our method is the necessity to have the initial box $[\mathbf{z^0}]$. Instead, we have the approximate solution $\tilde{\mathbf{z}}$ which we can use to create the degenerate box:

$$[\tilde{\mathbf{z}}] \quad | \quad sup[\tilde{\mathbf{z}}] = inf[\tilde{\mathbf{z}}] = \tilde{\mathbf{z}}.$$

However we cannot use this box with the classical Newton algorithm. At line 5 of Algorithm 1 the Newton method computes the intersection between the new box computed by the inner step and the old box, therefore an intersection between a box and a vector does not make sense. We will use instead the modified version propose by A. Goldsztejn[4] rewritten in Algorithm 2. Removing the intersection step keeps the rigor of the method because:

$$[\mathcal{I}nner\mathcal{S}tep]([\mathbf{h}],[\mathbf{z}]) \subseteq int([\mathbf{z}]) \Rightarrow ([\mathcal{I}nner\mathcal{S}tep] \cap [\mathbf{z}]) \subseteq int([\mathbf{z}]) \tag{17}$$

The Inner Step of the interval Newton algorithm used in our method is the Krawczik operator (or ad lib. another operator like several versions of the Hansen-Sengupta operator).

$$\begin{cases} [K]_{[\mathbf{h}]}([\mathbf{z}]) = \mathbf{z_m} + [K](C \cdot [Z], -[\mathbf{z_C}], [\mathbf{z}] - \mathbf{z_m}) \\ [K]([A],[\mathbf{b}],[\mathbf{z}]) = [\mathbf{b}] + (I - [A]) \cdot [\mathbf{z}] \\ \mathbf{z_m} = Mid([\mathbf{z}]) \\ [Z] = \left[\frac{d\mathbf{h}}{d\mathbf{z}}\right]([\mathbf{z}]) \\ C = Mid([Z])^{-1} \\ [y] = C \cdot [\mathbf{h}](\mathbf{z_m}) \\ I = \textit{the identity} \end{cases} \tag{18}$$

The Krawczik operator needs $[\mathbf{h}](\mathbf{z_m})$ and $\left[\frac{d\mathbf{h}}{d\mathbf{z}}\right]([\mathbf{z}])$. $[\mathbf{h}](\mathbf{z_m})$ can be computed using an rigorous ODE solver for the time-dependent constraint (like *VNODE*) and easily for the guard constraint. Beside computing $\left[\frac{d\mathbf{h}}{d\mathbf{z}}\right]([\mathbf{z}])$ could be easy with some systems but it is not a trivial problem.

3.3 Numerical Experiments

We now present some examples. For each one we used Mathematica's built-in subroutines to obtain an approximation and then Algorithm 2 where $[\mathbf{h}](\mathbf{z_m})$ and $\left[\frac{d\mathbf{h}}{d\mathbf{z}}\right]([\mathbf{z}])$ are evaluated using Mathematica IDSolve to obtain an enclosure. The stopping criterion of our method is when the algorithm performs 10 iterations. The last example is not an HDS but presents another application of Algorithm 2.

Algorithm 2:

Input: $\mathbf{h} : \mathbb{IR}^n \to \mathbb{IR}^n$, $\tilde{\mathbf{z}} \in \mathbb{R}^n$
Output: $[\mathbf{x}] \in \mathbb{IR}^n$
$\delta \leftarrow 1.01$; `/* Inter-step inflation ratio */`
$\Delta \leftarrow [-10^{-10}, 10^{-10}]$; `/* Inter-step inflation ratio */`
success $\leftarrow$ **false**;
$[\mathbf{z}] \leftarrow \tilde{\mathbf{z}}$;
repeat
  $[\mathbf{z}^*] \leftarrow \mathbf{Mid}[\mathbf{z}^*] + \delta\ ([\mathbf{z}] - \mathbf{Mid}[\mathbf{z}]) + \Delta$;
  $[\mathbf{z}] \leftarrow [\mathit{InnerStep}]([\mathbf{h}], [\mathbf{z}^*])$;
  if ($[\mathbf{z}] \subseteq \text{int}[\mathbf{z}^*]$) **then** success $\leftarrow$ **true**;
until *(stopping criterion not reach)* ;
return ($[\mathbf{z}]$, success);

The bouncing ball With this example we show the possibility to simulate the behavior of a ball bouncing on a sinusoidal floor. The figure 4 represents the trajectory of the ball. The ODE of the system is:

$$\begin{cases} x_1'(t) = v_1(t) \\ x_2'(t) = v_2(t) \\ v_1'(t) = 0 \\ v_2'(t) = -a \end{cases} \tag{19}$$

with the following initial conditions:

$$\begin{cases} x_1(0) = 0 \\ x_2(0) = 5 \\ v_1(0) = 2 \\ v_2(0) = 0 \end{cases} \tag{20}$$

And the constraint guard is :

$$g(x_1, x_2) = sin(x_1) - x_2$$

Last the floor has a hardness factor of $\sigma = 0.8$ and gravity $a = 9.81$ We define h as:

$$\begin{cases} h_1(z) = x_1(z_5) - z_1 \\ h_2(z) = x_2(z_5) - z_2 \\ h_3(z) = v_1(z_5) - z_3 \\ h_4(z) = v_2(z_5) - z_4 \\ h_5(z) = Sin(z_1) - z_2 \end{cases} \tag{21}$$

The solutions computed for the first impact are (the time is the last number):

$$\text{Mathematica} \quad \begin{pmatrix} \underline{1.8126534861}972665 \\ \underline{0.970894848}346366 \\ 2. \\ -\underline{8.891065}49797593 \\ \underline{0.9063267430}986333 \end{pmatrix} \tag{22}$$

$$\text{Algorithm 2} \quad \begin{pmatrix} 1.81265348645262[63,76] \\ 0.9708948482852050[0,6] \\ 1.999999999999999[3,11] \\ -8.8910653510501[30,41] \\ 0.906326743226313[2,7] \end{pmatrix} \tag{23}$$

The solutions computed for the second impact are:

$$\text{Mathematica} \quad \begin{pmatrix} \underline{7.25351756}183085 \\ \underline{0.8250734912}027126 \\ \underline{4.648698364}7258056 \\ -\underline{5.86543142}8810479 \\ \underline{2.0767326}60789209 \end{pmatrix} \tag{24}$$

$$\text{Algorithm 2} \quad \begin{pmatrix} 7.253517570611[542,623] \\ 0.8250734961640[269,723] \\ 4.6486983678135[045,440] \\ -5.865431440461[656,856] \\ 2.0767326619734[114,257] \end{pmatrix} \tag{25}$$

The resolution of this system takes 14.578125 seconds. For the first bounce the existence is proven at the sixth iteration and at the fifth for the second. The numerical results computed by Mathematica are not included in the computed boxes.

The falling ball This time we simulate a falling ball crossing three times a sinusoidal floor, the only difference with the previous system being the initials= conditions which are:

$$\begin{cases} x_1(0) = 0 \\ x_2(0) = 1 \\ v_1(0) = 8 \\ v_2(0) = 0 \end{cases} \tag{26}$$

With Algorithm 2 we can compute the third crossing point of the system (the trajectory is represented in the figure 5). First we look for an approximative solution $\tilde{\mathbf{x}}$ of the third crossing point:

$$\tilde{\mathbf{x}} = \begin{pmatrix} 5.0 \\ -1.0 \\ 8.0 \\ -6.0 \\ 0.6 \end{pmatrix} \tag{27}$$

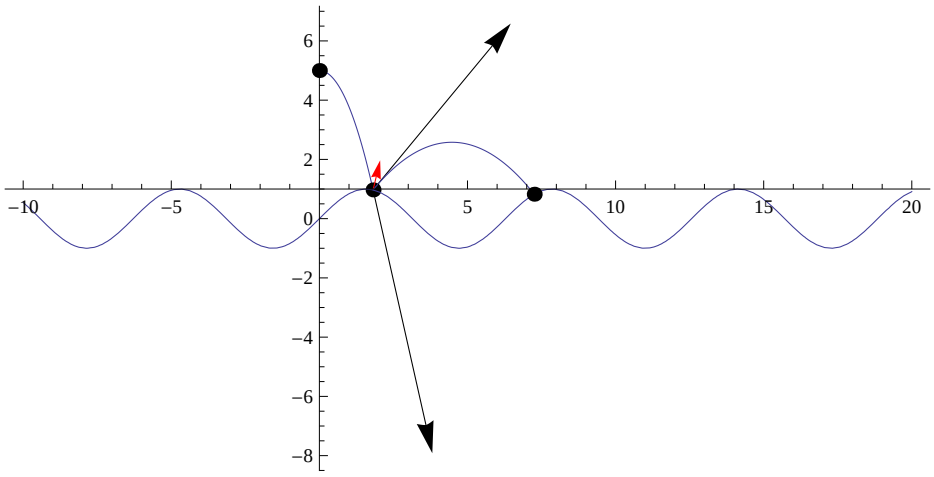

Fig. 4. A ball bouncing twice on a sinusoidal floor. Arrows representing the velocity vector just before the first impact and just after, the small arrow representing the normal of the floor at the position on the impact

Then we use this solution to compute the enclosure:

$$\begin{pmatrix} 5.04001761407106[1,5] \\ -0.94680810755420[16,25] \\ 7.99999999999999[7,14] \\ -6.1803215992546[38,44] \\ 0.63000220175888[27,31] \end{pmatrix} \tag{28}$$

The enclosure is computed in 7.15625 seconds, and the existence is proven at the fifth iteration. We can try to find the second one with:

$$\tilde{\mathbf{x}} = \begin{pmatrix} 2.0 \\ 0.4 \\ 8.0 \\ -3.0 \\ 0.3 \end{pmatrix} \tag{29}$$

And the enclosure computed, in 7.171875 seconds, is:

$$\begin{pmatrix} 2.67138860568146[26,50] \\ 0.4530681985940[5995,6133] \\ 7.99999999999999[7,14] \\ -3.27579027771689[24,77] \\ 0.33392357571018[280,316] \end{pmatrix} \tag{30}$$

and the existence is proved at the fifth iteration too.

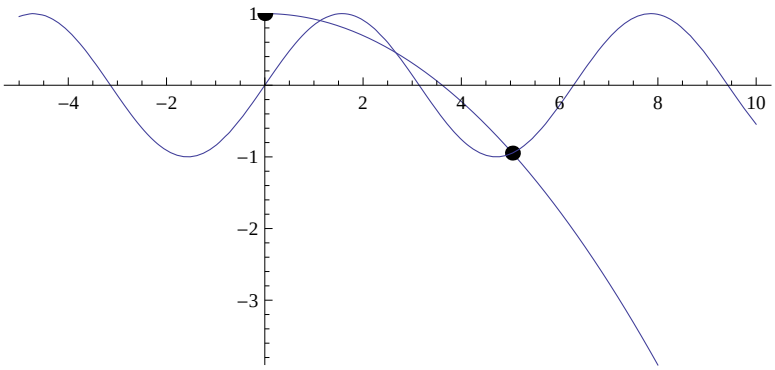

Fig. 5. A ball falling and crossing three times a sinusoidal floor

Enclosure of a periodic orbit In this last example we have used Algorithm 2 to find the solution of a periodic Lorenz system. The ODE is:

$$\begin{cases} x'(t) = & \sigma \times (y(t) - x(t)) \\ y'(t) = & x(t) \times (\rho - z(t)) - y(t) \\ z'(t) = & x(t) \times y(t) - \beta\ z(t) \end{cases} \tag{31}$$

with the parameters $(\sigma,\ \beta,\ \rho) = (10,\ \frac{8}{3},\ 28)$. We have to solve:

$$\begin{cases} \Phi x(t), t = x(t) \\ \Phi y(t), t = y(t) \\ \Phi z(t), t = z(t) \end{cases} \tag{32}$$

We have four variables and only three functions, so we add a constraint:

$$z(t) - z_0 = 0. \tag{33}$$

Finally we have the following system:

$$\begin{cases} \Phi x(t), t - x(t) \\ \Phi y(t), t - y(t) \\ \Phi z(t), t - z(t) \\ z(t) - z_0 \end{cases} \tag{34}$$

and we fix arbitrarily $z_0 = 5$. Figure 6 is a simulation of this system computed with an approximated solution.

In this case computing the Jacobian of the system is not as trivial as the bouncing ball system, so we do not explain how to do this in this document.

The approximate solution computed by Mathematica is:

$$\begin{pmatrix} \underline{2.0239}543360613745 \\ \underline{4.093}776265928416 \\ 5. \\ \underline{6.46025}907180557 \end{pmatrix} \tag{35}$$

With Algorithm 2, we proved the existence of a solution in 3 iterations in the box:

$$\begin{pmatrix} 2.02394337[0783742, 17309705] \\ 4.0937539[86488837, 99157982] \\ 4.99999999999999[9, 11] \\ 6.46025921[1807459, 2013839] \end{pmatrix} \tag{36}$$

The result given by Mathematica is exact up to $10^{-}4$.

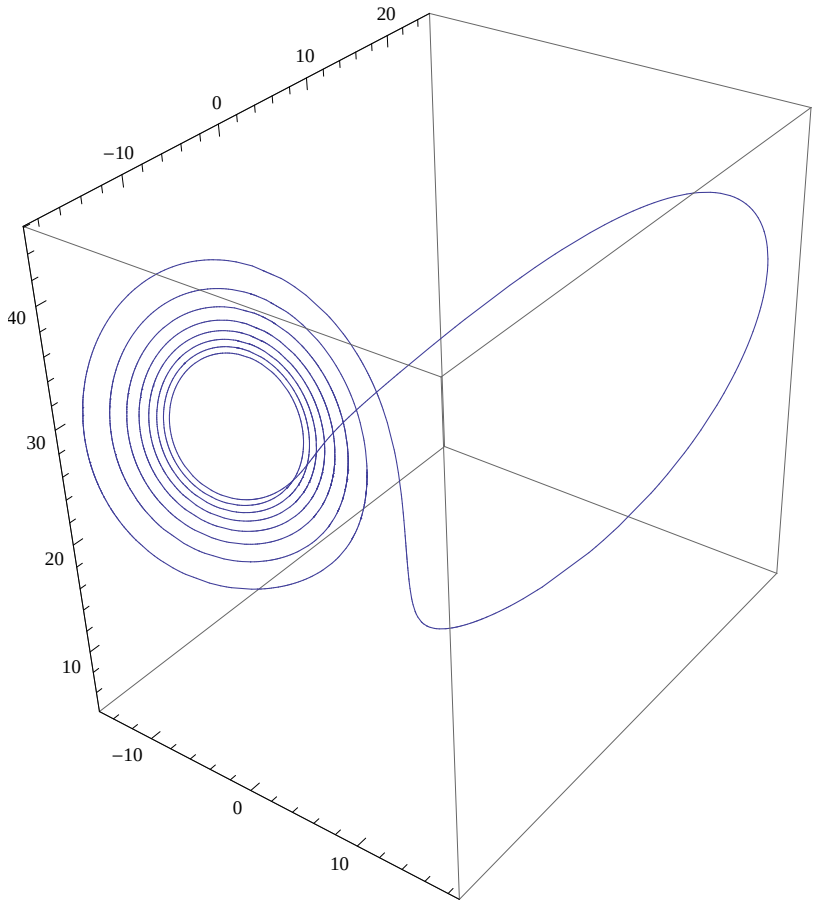

Fig. 6. Projection of a periodic Lorenz system

4 Conclusion and perspectives

We have presented a new method for enclosing the solutions of HDSs. This method can prove the existence of such solutions (vide supra Section 3.2). This method could be used to enclose the solutions computed by classical numerical methods. An other advantage is that an HDS with uncertain initial values can be solved. Currently this method is limited by the ODE solver used. Another limitation is if the solution is near t_0 the inner step operator tries and fails in computing a enclosure with a lower bound earlier than t_0. Dr Goldstzejn and Pr. Hosobe approved starting to write a paper that presents Algorithm 2 used with Mathematica IDSolve. It will be submitted to *Reliable Computing*, the main journal dedicated to interval analysis.

References

1. Hansen, E. and Sengupta, S.: Bounding solutions of systems of equations using interval analysis. BIT 21, 203–211 (1981)
2. Neumaier, A.: Interval Methods for Systems of Equations. Cambridge Univ. Press, (1990)
3. Goldsztejn, A.: A Comparison of the Hansen-Sengupta and Frommer-Lang-Schnurr Existence Theorems. Computing 79(1), 53–60 (2007)
4. Goldsztejn, A.: Sensitivity Analysis Using a Fixed Point Interval Iteration. CNRS, hal-00339377 (2008)
5. R. Alur, C. Courcoubetis, et al. Discrete abstractions of hybrid systems. Proceeding of IEEE 88, 970–983 (2000)
6. Nedialkov, N.S., Mohrenschildt, M.v.: Rigorous Simulation of Hybrid Dynamic Systems with Symbolic and Interval Methods. Reliable Implementation of Real Number Algorithms: Theory and Practice, 140-147 (2002)
7. Nedialkov N., N.S. VNODE-LP: a validated solver for initial value problems in ordinary differential equations. McMaster University, TR CAS-06-06-NN (2006)
8. Hickey, Wittenberg: Rigorous Modeling of Hybrid Systems Using Interval Arithmetic Constraints. Hybrid Systems: Computation and Control, 139–142 (2004)
9. Ishii D., Ueda K., Hosobe H., and Goldsztejn A.: Interval-based Solving of Hybrid Constraint Systems. 3rd IFAC Conference on Analysis and Design of Hybrid Systems (ADHS'09) (to appear)
10. Kühn, W.: Rigorously Computed Orbits of Dynamical Systems Without the Wrapping Effect. Computing 61, 47–67 (1998)
11. Berz M. and Makino K.: Verified Integration of ODEs and Flows Using Differential Algebraic Methods on High-Order Taylor Models. rcomp 4(4) 361–369 (1998)
12. Wolfram Research Mathematica 7.0. (2008): www.wolfram.com/products/mathematica
13. VNODE-LP: http://www.cas.mcmaster.ca/ nedialk/vnodelp/
14. Mathematica IDolve: http://code.google.com/p/mathematica-idsolve/
15. Windows modification for VNODE-LP: http://sites.google.com/site/lejeuneaurelien/
16. Agarwal R.P. , O'Regan D.: An introduction to ordinary differential equations. Springer, Universitext (2008)
17. L. Granvilliers: A Symbolic-Numerical Branch and Prune Algorithm for Solving Non-linear Polynomial Systems. j-jucs 4(2) 125–146 (1998)

5 Appendix

5.1 Installation of Mathematica IDSolve

The Mathematica IDSolve plug-in can be downloaded at:

`http://code.google.com/p/mathematica-idsolve/`

Up to now the source code and a compiled version for Microsoft Windows are available. You have two options: if you are running under a Microsoft OS you should probably download the win32 compiled version; if you are running under another OS you have to download the source code and recompile the plug-in yourself, or wait until a compiled version will be available for your OS, you need also a compatible version of VNODE[13][7].

If you want to compile the sources yourself, you need to read the Readme file where this procedure is explained, otherwise you can directly follow these instruction to use it in Mathematica.

For instance the file IDSolve.m and the executable file DSolve[8] are located in the following directory:

`/mmaPlugIn/IDSolve/`

Into the Mathematica environment type the following line before processing a computation:

```
Get["/mmaPlugIn/IDSolve/IDSolve.m"];
IDSetLinkPath["/mmaPlugIn/IDSolve/IDSolve.exe"];
```

Now you can use the example described in Section 2.2

[7] a VNODE-LP version for Windows is available at:
`http://sites.google.com/site/lejeuneaurelien/`

[8] `IDSolve.exe` for the win32 version