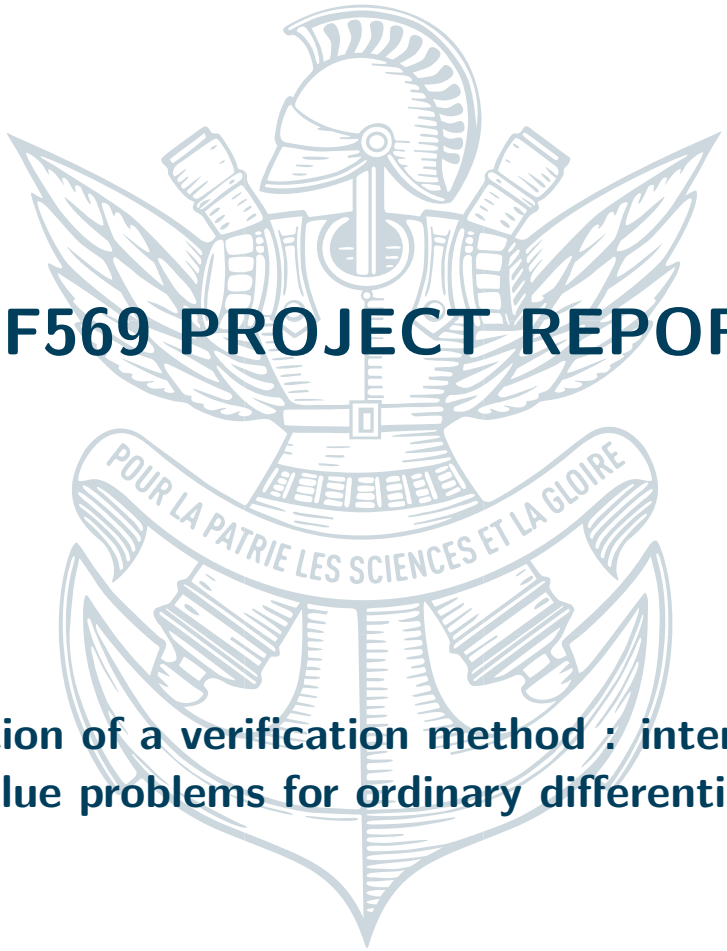


INF569 PROJECT REPORT

Implementation of a verification method : interval methods for initial-value problems for ordinary differential equations

March, 2018

Anouk Paradis
Maxime Voituriez

1 INTRODUCTION

Complex cyber-physical systems are often regulated by multiple differential equations where we are not only interested in how it will likely evolve, but also in proving that in cannot reach a given state, or that it will always satisfy a specific condition. Hence, merely calculating an approximate solution to the problem is not enough: a guaranteed error bound is often needed. Interval methods produce such bounds and verify that a unique solution to the given problem exists (and that it is in that guaranteed error bound). There are many different algorithms to calculate those bounds. The main difference between them is how they are trying to mitigate the "wrapping effect" of intervals we can see in many cases, and how they will perform on different differential equations with different initial values. By understanding which kind of problems each of them is better suited for we will be better users of verification systems, as it will teach us how to "tweak" equations to make verification more precise or more efficient and to critically examine any solution that we would want to use later. Our aim in this project was to implement a few different interval methods for initial-value problems in the case of ordinary differential equations. We then aimed at comparing them between each other, to find their pros and cons. We first tried to implement the methods described during the course. Then we implemented Eijgenraam's mean-value method to test how it improves the result. This allowed us to experiment the results obtained on different equations to see where were the best results.

2 PRELIMINARIES

The goal of these methods is to have a guaranteed set integration by Taylor methods. We consider an uncertain system defined by a differential equation where we want to get a set of trajectories from which the solution cannot escape. This is often seen as tubes of trajectories in which any solution to the equation with the given initial value is guaranteed to be. For each time step of the integration, 2 steps are performed:

- First we try to find a box in which the function exists and will be in for the next moments wherever was the initial value in the given interval. This step allows to verify the existence of the solution function and to enclose the evolutions of the function.
- Next we want to tighten the final interval. Since boxes are used in the first step, the precision on the interval at the end of the box can be very poor. That is why a tight enclosure is computed in this step, so we have to find what the final values can be when the initial values come from a restricted interval whatever were the evolutions between both intervals.

To sum up, the goal of the first step is to compute those approximative bounding boxes on big time-intervals in order not to compute for too many of them while the goal of the second is to be as precise as possible on the interval of values the solutions can take.

Function type	$f : \mathbb{R} \to \mathbb{R}$	$f : \mathbb{R}^2 \to \mathbb{R}^2$
Step 1	Simple method and Taylor series enclosure	Simple method
Step 2	Simple method	Simple method and Eijgenraam's method

Figure 1: Algorithms implemented

3 OUR WORK : ALGORITHMS AND IMPLEMENTATIONS

Before we could start implementing anything, we had to understand the alogrithms presented in [4]. This proved to be quite time-consumming. Indeed, notations are introduced all along the article, and not all of them are clearly defined. We hence had to thoroughly read and understand a great part of the paper before we trully understood what did each variable refer to in all the algorithms. Besides, even though all algorithms seem nicely presented in figures at first sight, we discovered that some of them cannot be implemented, based solely on this paper. For instance, the Lohner polynomial enclosure for step 1 of the algorithm requires some calculations described in another paper, itself based on a particular kind of arithmetic that can be found in yet another paper. We had the same kind of problem's with Moore's and Krückeberg's algorithms. We hence decided to limit our implementation to the algorithms fully described in the paper, as there already were quite a few of them, and we wanted to have enough time to conduct interesting experimentations.

We finally chose to implement four different algorithms. On the one hand, for the first step of the algorithms, the simple method described in class and the Taylor series enclosure described in section 5.3 of [4]. On the other hand, for the second step of the algorithms, again the simple method and the Eijgenraam's method described in section 7.4 of [4].

The Eijgenraam's method is usefull only for functions whose codomain is of dimension at least two. However, using the library FADBAD++ [1]even in one dimension proved to be extremely challenging. We hence decided to implement some code for real functions, using FADBAD++, and some other code where we hardcoded derivations for functions with codomain $\mathbb{R}^2$. This is summarized in the table 3. The code for real function can be used for any functions, thanks to FADBAD++ automated differentiation, whereas the other codes can only be used on the example we implemented it for.

3.1 f WITH CODOMAIN IN 1D

Using the PROFIL/BIAS [3] and FADBAD++ libraries, we implemented the calculations of flowpipes for any time independant ODE, that is any equation of the form $x' = f(x)$. This implementation is in the files `simpleMethod.cpp` and `taylorStep1.cpp`. The function f and other constants can be changed from within the files. To build the files one can use the Makefile, after changing the PROFIL/BIAS and FADBAD++ paths, using `make simpleMethod1D` or `make taylorStep1`. When run, the executable `simpleMethod1D` will write its results in the file `values.txt`, and `taylorStep1`

will write in `valuesT.txt`. Those files can then be used to plot the result using the python file described in section 3.3.

3.2 f WITH CODOMAIN IN 2D

After needing almost an hour to merely calculate the Taylor coefficients of a $f : \mathbb{R}^2 \to \mathbb{R}^2$ function using FADBAD++, we decided that trying to implement the calculation of the Jacobian matrix of those coefficients using the same library was probably out of our reach. Hence we could only implement the calculations of flowpipes in $\mathbb{R}^2$ on a specific differential equation. We chose the following differential equation :

$$\begin{cases} y_1' = y_2 \\ y_2' = -y_1 \end{cases} \tag{1}$$

We chose it because its solution "rotates". Hence the wrapping effect is really strong on it, and the upgraded precision the Eijgenraam's method brings is sensible.

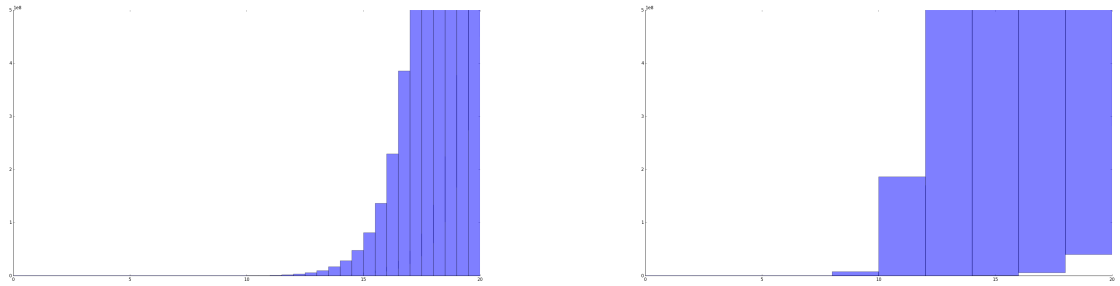
The implementation is in the file `eijgenraamMethod.cpp`. At the beginning of the program, many constants can be changed. It can be compiled into the executable `eij` using the command `make eij`. When run, this executable first asks for which method to use (simple or Eijgenraam's), then writes its results in `valuesE.txt`. Those can then be printed using the python file. To use this implementation on any other differential equation, one would have to change the calculations of the Taylor coefficients of the equation (the `fi` function) and the calculation of their Jacobian matrix (the `jacobFi` function). Besides, in the code, we exploit the fact that these Jacobian are constant (ie do not depend on the value of (y_1, y_2)). If that is not verified by the new function f, the results would not be correct.

3.3 PRETTY-PRINTING

In order to visualise the results of our methods, we wanted to have a little program that would display the boxes and the interval at the same time as the exact solution of the equation. We decided to implement this in Python to take advantage of its well-known libraries designed for graph plotting and drawing. Rather than use complex drawing libraries, we focused on using matplotlib since we were already familiar with it. However, this made the conception of the boxes difficult because of its basic drawing-tools. It was even worse in 3D where no specific tools could be used, so we had to draw the cubes line after line including the diagonal lines to ensure we can clearly see them. The visual results we obtained helped us to debug our code with an improved efficiency and clearly see how each of our methods was different from the others.

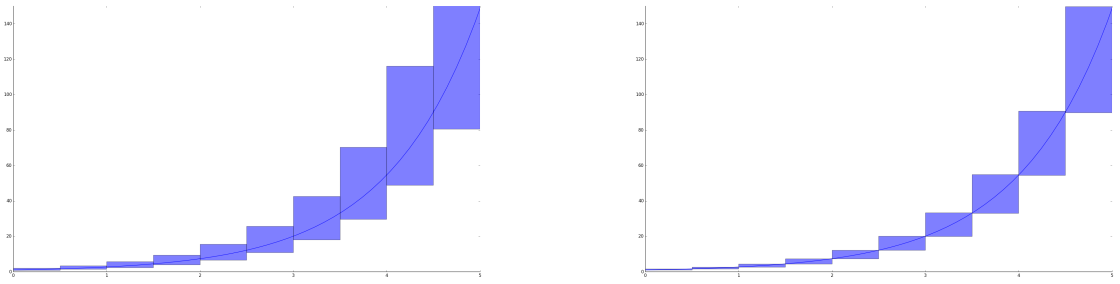
To visualize the results of any of the algorithms, simply run the program `graphplot.py`.

4
EXPERIMENTATIONS



Simple method | Taylor series enclosure

Figure 2: Efficiency of Taylor series enclosure



Simple method | Taylor series enclosure

Figure 3: Precision of Taylor series enclosure

4.1 Efficiency/Precision of the Taylor series enclosure

According to [4], the main advantage of the Taylor series enclosure is that the stepsize h can be bigger, as the enclosure is more precise and hence its image is included in itself for h smaller. We could indeed see this in our experimentations, for instance in figure 4.1. Here the equation is $x' = x$, and the initial condition is $x(0) = 1$. In this experiment, we set the maximum time step (`maxH`) to 4 for both algorithms. The simple method disminishes `h` to be able to create bounding boxes, whereas the Taylor series enclosure is able to use this maximum step. However, using this maximum step, while making calculation a lot faster, makes the bounding boxes bigger. To check that this is indeed only due to the step size, we ran a second experiment, shown in figure 4.1, with `maxH`. We can see that when both methods use the same step, Taylor series enclosure produces tighter boxes than the simple method.

As a conclusion of those experiments, the Taylor series enclosure permits either faster calculations (by using a bigger stepsize than is possible with the simple method) or more precise ones (when using the same timestep).

4.2 Precision of Eijgenraam's method

In the Eijgenraam's method, instead of storing the results of the second step calculations as an interval vector $([y_1], [y_2])$, it is stored as the product of a matrix and an interval vector $A * ([x_1], [x_2])$. At each time step, both A and $([x_1], [x_2])$ are updated, and their product $([y_1], [y_2]) = A * ([x_1], [x_2])$ is used to calculate the next bounding box. However when calculating the next tight interval, we will use A and $([x_1], [x_2])$, and not $([y_1], [y_2])$. This helps avoid the wrapping effect, as A "stores the rotation" to be applied to the "straight" interval vector $([x_1], [x_2])$. We tested this method on the initial value problem:

$$\begin{cases} y_1' = y_2 \\ y_2' = -y_1 \end{cases} \quad \begin{cases} y_1(0) = 0 \\ y_2(0) = 1 \end{cases} \tag{2}$$

whose solutions are $y_1(t) = sin(t), y_2(t) = cos(t)$. On this problem, the difference of precision between the simple method and Eijgenraam's method were rather impressive, as can be seen in figure 4.2.

Indeed, the matrix A almost perfectly captures the rotation, as can be seen in 4.2, where we showed the matrix A calculated by the algorithm at different times t alongside the rotation matrices for the angle t.

Since those results seemed really precise, we decided to try our program for really long time intervals. The results were still quite satisfying. For instance, for `tSup = 1000`, the algorithm calculated that the solution was enclosed in :

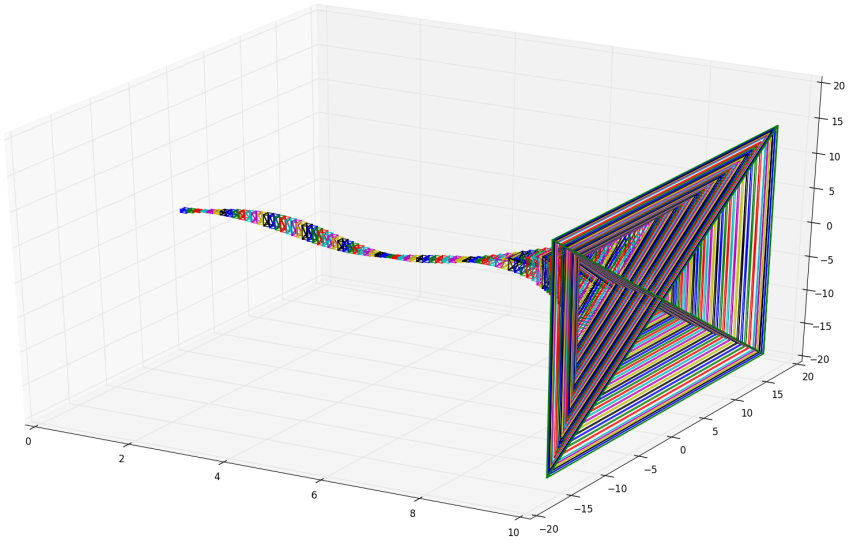
$$\begin{pmatrix} [-2.52935, 4.12479] \\ [-2.78863, 3.86549] \end{pmatrix} \tag{3}$$

5 CONCLUSION

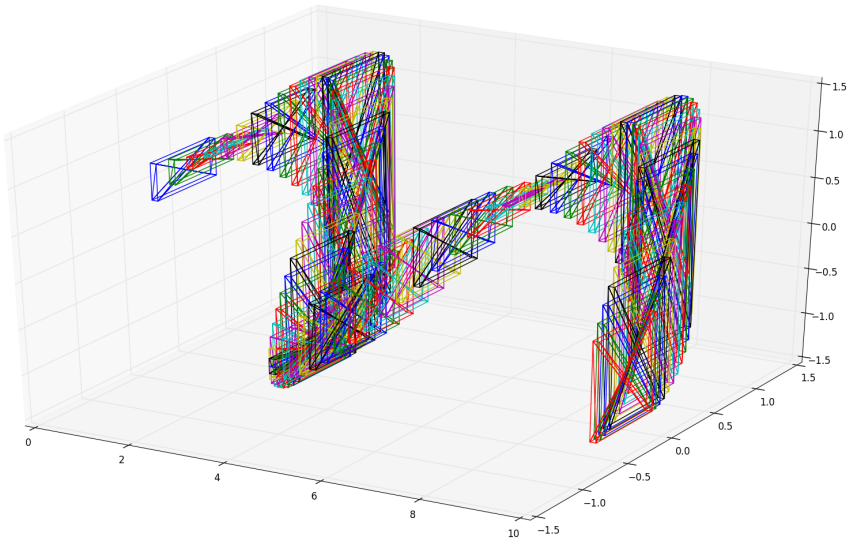
This project made us realise the complexity of verification methods, especially when in more than one dimension. In comparison with what we initially wanted to do, we couldn't implement all the methods we wanted. This took us some time as we spent quite some time understanding Moore's local coordinate transformation and Krückeberg's three process method, before realizing that the explanations given in [4] weren't complete enough for us to implement them.

An interesting part of this project was learning about the FADBAD++ automatic differentiation library. However its minimalistic documentation did not allow us to use it for 2D-examples, so we had to calculate Jacobian coefficients by ourselves before hard-coding them into the program.

If we had to do this project again, we would implement the visualisation of the results of the methods sooner. Once we could see how it looked like compared to the exact solution, it was much easier to debug and to understand what was happening. For instance, it took us only a few minutes to understand that a factor was missing once we saw the result, whereas we hadn't seen it in the previous debuggings of our code. This project also made us aware of the problem of using too big libraries which are difficult to install, understand and use in an appropriate manner. We found out too late that we should have used the PROFIL library instead of BIAS since it was more suitable for our needs.



Simple method



Eijgenraam's method

Figure 4: Precision of Eijgenraam's method

Time t	Matrix A at time t	Rotation matrix for the angle t
1	$\begin{pmatrix} 0.538 & 0.843 \\ -0.843 & 0.538 \end{pmatrix}$	$\begin{pmatrix} 0.540 & 0.841 \\ -0.841 & 0.540 \end{pmatrix}$
10	$\begin{pmatrix} 0.839 & -0.544 \\ 0.544 & 0.839 \end{pmatrix}$	$\begin{pmatrix} -0.827 & 0.841 \\ -0.841 & -0.8267 \end{pmatrix}$

Figure 5: Rotation matrices

Taking time to fully explore the documentation of the library should have been a bigger priority on our list to avoid some time consuming mistakes, such as fully implementing the simple method using the wrong library... Finally, if less time had been lost on those issues, we would have liked to compare the result of what we found to classical Taylor-based reachability tools such as Flow*[2] to see where and to what extent our results are different.

REFERENCES

[1] Claus Bendtsen and Ole Stauning. Fadbad, a flexible c++ package for automatic differentiation. Technical report, 1996.

[2] Xin Chen, Sriram Sankaranarayanan, and Erika Abraham. Flow* 1.2: More effective to play with hybrid systems. In Goran Frehse and Matthias Althoff, editors, *ARCH14-15. 1st and 2nd International Workshop on Applied veRification for Continuous and Hybrid Systems*, volume 34 of *EPiC Series in Computing*, pages 152–159. EasyChair, 2015.

[3] O Knüppel. Profil/bias—a fast interval library. computing, 53, 277-287. 53:277–287, 09 1994.

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