Software verification and introduction to hybrid systems analysis

Master IP Paris – Cyber-Physical Systems

Taylor-based reachability analysis of continuous and hybrid systems

Émilie THOMÉ

Sunday, November 29th 2020

Introduction

The goal in this project was to implement a Taylor-based reachability analysis for non linear continuous systems and to extend it to hybrid systems. A Taylor-based reachability analysis consist in bounding the end result of a dynamic system. This end result computed using the Taylor-Lagrange expansion is not exact:

$$x(t+h) = x(t) + \sum_{i=1}^{n-1} \frac{h^i}{!i} \frac{d^i x}{dt^i}(t) + O(h^n)$$

By computing this reachability analysis we bound the real end result of the dynamic system and we can prove some properties specific to the system (do not reach zero for example).

Tools

In order to implement the analysis in C++ with interval as suggested, I used some existing tools:

FADBAD++: FADBAD++ implements the forward, backward and Taylor methods utilizing C++ templates and operator overloading. These AD-templates enable the user to differentiate functions that are implemented in arithmetic types, such as doubles and intervals. One of the major ideas in FADBAD++ is that the AD-template types also behave like arithmetic types. This property of the AD-templates enables the user to differentiate a C++ function by replacing all occurrences of the original arithmetic type with the AD-template version. This transparency of behavior also makes it possible to generate high order derivatives by applying the AD-templates on themselves, enabling the user to combine the AD methods very easily.

FILIB++: `filib++` is an extension of the interval library `filib` originally developed in Karlsruhe. The most important aim of the latter was the fast computation of guaranteed bounds for interval versions of a comprehensive set of elementary function. It adds a second mode, the "extended" mode, that extends the exception-free computation mode using special values to represent infinities and Not a Number known from the IEEE floating-point standard 754 to intervals. In this mode so-called containment sets are computed to enclose the topological closure of a range of a function defined over an interval.

Matplotlib for C++: Matplotlib for C++ is a C++ wrapper for Python's matplotlib (MPL) plotting library.

Program description

In this part, I will describe the overall project, the structure and the methods. I developed some examples to make it possible to see results we can have with this project. I focused also on the practicality of the project, but as I am not a professional it is far from being perfect. I also had difficulties to compile with "complex" C++ structures and to use the `Makefile` which made me lost a lot of time.

Configuration

First, we need to configure the project. So let me introduce my implementation structure. I tried to mimic the structure of the tools I used recently.

```
Mini_Projet/
├── examples/
│   ├── test_continuous.cpp
│   └── test_hybrid.cpp
├── include/
│   ├── box.h
│   ├── experiment.h
│   ├── fadbad_interval.h
│   ├── filib_interval.h
│   ├── hybrid.h
│   ├── ode.h
│   ├── plot_enclosure.h
│   └── polynome.h
├── lib
│   ├── FADBAD++/
│   ├── filibsrc/
│   └── matplotlib-cpp-master/
├── src
│   ├── box.cpp
│   ├── experiment.cpp
│   ├── hybrid.cpp
│   ├── ode.cpp
│   └── plot_enclosure.cpp
└── Makefile
```

In the `Makefile` are specific paths that needs to be adapted:

```
# Directories to adapt
FILIBHOME = lib/filibsrc
FADBADHOME = lib/FADBAD++
PYTHON3HOME = /usr/include/python3.8
NUMPYHOME = /home/emilie/.local/lib/python3.8/site-packages/numpy/core/include
MATPLOTHOME = lib/matplotlib-cpp-master
SOURCE_PATH = src
INCLUDE_PATH = include
EXAMPLE_PATH = examples
```

And now the project is ready.

Project method

The project was done in two times: first the reachability analysis for ODE and in a second time for hybrid systems.

ODE

In order to represent ODE, I created a simple `Class ODE` as presented in `FADBAD++` examples. The ODE represented are only the $\dot{x} = f(x)$ and not the $\dot{x}(t) = f(x(t), t)$.

The `Experiment` with ODE is a successive computation of the polynome enclosing $x(t_i + t)$:

$$P_i(t) = X_i + \sum_{k=1}^{n-1} \frac{t^k}{!k} L_f^k(x)(X_i) + O(t^n)$$

The $L_f^k(x)(X_i)$ were computed by `FADBAD++` adapted to intervals. The only thing I had to compute was the $O(t^n)$. And in order to do so I used the Lohner's Theorem:

Given $\dot{x} = f(x)$, $x(t_i) \in X_i$, and an initial guess B_0 for the enclosure of x on $[t_i, t_i + h]$. If $B_1 := X_i + [0, h].f(B_0) \subseteq B_0$ then the initial value problem above has exactly one solution over $[t_i, t_i + h]$, which lies entirely within the B_1 bounding box.

It permits to compute a large bounding box including every $x(t_i + t)$ for $t \in [0, h]$. Then the computed polynome is:

$$P_i(t) = X_i + \sum_{k=1}^{n-1} \frac{t^k}{!k} L_f^k(x)(X_i) + \frac{t^n}{!n} L_f^n(x)(B)$$

We know that for every $t \in [0, h]$, $x(t_i + t) \in P(t)$, so we can define $X_{i+1} := P(h)$ and continue to compute the following X_k by recurrence.

I tried to be careful with infinite enclosure and loops so that the program stops when their is nothing that can be done further with the current parameters. In case an infinite enclosure is reached, a shorter step of computation can be helpful.

Complementary

In complementary of the current bounding box algorithm:

```
void Box::bounding_box   (ODE ode,
                          v_interval xi,
                          double step,
                          double epsilon,
                          double alpha)
{
          // Set h as step in the bounding box
          h = step;
          // And the current state Xi
          B = xi;
          B = phi(ode, xi);

          /* Finding a bouncing box*/
          // Check if xi + [0, h]*f(B) in B
          while (not inclusion_reached(ode, xi)){
                          widen(alpha);
          }
}
```

I found an additional step that permits to reduce the bounding box in order to be more precise.

```
void Box::bounding_box   (ODE ode,
                          v_interval xi,
                          double step,
                          double epsilon,
                          double alpha)
{
        // Set h as step in the bounding box
        h = step;
        // And the current state Xi
        B = xi;
        B = phi(ode, xi);

        /* Finding a bouncing box*/
        // Check if xi + [0, h]*f(B) in B
        while (not inclusion_reached(ode, xi)){
                        widen(alpha);
        }

        /* Reducing the bouncing box*/
        // Check the tolerance is reached
        int iter = 0;
        while (not (tolerance_reached(ode, xi, epsilon))
                   and (iter < MAX_ITER)){
                iter++;
                B = phi(ode, xi);
        }
        // This permit to avoid infinite loops
        if (iter >= MAX_ITER){
                printf("Tolerance not reached.\n");
        }
}
```

It is not necessary but I have seen differences when using it for the first time.

Hybrid system

Hybrid systems were more complex. I defined two additional classes: `class State` and `class Hybrid`. A `State` is a reached state of the automate with its number and the enclosure of the Hybrid System in this state (different States can represent the same state of the automate but with different enclosure because they came from a different path). An `Hybrid` has multiple `State`(s) that represent its enclosure. An `Hybrid` is composed of ODEs that represent the general system and the states of the automate ; inGards that are the guards keeping the current system in its state ; outGards that are the guards allowing the current system to change of state ; outMod that are the modification in the current system when changing of state. Guards are represented as set of intervals representing the allowed enclosure in the state or to take the arrow.

To compute the enclosure of the hybrid system, we have to repeat the computation of the next `Hybrid` from the current one.

The method I used was to compute over every `State` of the `Hybrid` enclosure:

- The intersection of the `State` enclosure and the inGard related to the automate state of `State`:

  ```
  intersection(current_state.get_enclosure(),
          hybrid.get_inGards(current_state.get_num()))
  ```

- The intersection of the exclusion of the inGard over the `State` enclosure and the outGard of every state of the automate:

```
v_interval xOutState = exclude(current_state.get_enclosure(),
            hybrid.get_inGards(current_state.get_num()));
if (not isEmpty(xOutState)){
    for (int s = 0; s < STATES; s++){
        // Modifications application
        v_interval xInS = hybrid.get_outMod(current_state.get_num(),s)
        (intersection(xOutState,
                    hybrid.get_outGards(current_state.get_num(),s)));
        if (not isEmpty(xInS)){
            // Set the ODE experiment:
            Experiment expODE = Experiment(hybrid.get_system(s),
                                xInS, n, h, alpha, t, t_end, epsilon);
           // Compute the polinomal enclosure of
           //x'i+1 in the current state
            P.push_back(expODE.nextPolynome());
            states.push_back(s);
        }
    }
}
```

Then I find the minimum step for which every polynomial enclosure is true, and for every polynome of `P` (associated by a state in `states` by the same index it has in `P`) I compute the application over the minimum step found. I order to plot the enclosure I re-use the `Experiment.xi` (used to defined the enclosure of the ODE, but not useful with hybrid systems) by defining him has the union of the `State`(s) enclosure of the `Hybrid`.

The `xi` parameter of `Experiment` is not useful in hybrid systems because it removes knowledge about the enclosure: it removes the association between states and the current enclosure. In other words and using the bouncing ball example, the falling ball is considered exactly as a ball that just bump the ground and that is going up. We could add inGuards in order to exclude impossible enclosure in a state but it is quite laborious.

Examples

I tested the project over different examples for ODE and only other the Bouncing Ball test with for hybrid system.

ODE

For ODE, I tested the project with several ODE definitions:

```
/* ODE definitions */
vector<T<interval>> Brusselator(vector<T<interval>> x){
        vector<T<interval>> f = vector<T<interval>>(DIM);
        f[0] = interval(1.) + x[0]*x[0]*x[1] - 2.5*x[0];
        f[1] = 1.5*x[0] - x[0]*x[0]*x[1];
        return f;
}
vector<T<interval>> Linear(vector<T<interval>> x){
        vector<T<interval>> f = vector<T<interval>>(DIM);
        f[0] = 1;
        return f;
}
vector<T<interval>> Exponential(vector<T<interval>> x){
        vector<T<interval>> f = vector<T<interval>>(DIM);
        f[0] =-x[0];
        return f;
}
vector<T<interval>> Cos(vector<T<interval>> x){
        vector<T<interval>> f = vector<T<interval>>(DIM);
        f[0] = x[1];
        f[1] =-interval(39.478417, 39.478418)*x[0];
        return f;
}
```

Those functions represent the ODE dynamic.

Be careful to accord the returned dimension with the DIM external variable,which represents the dimension of `x` in the ODE, initialized in the `main()` function. It also has to be defined as `vector<T<interval>> f(vector<T<interval>> x)` unless the project does not compile.

The `main()` function can looks like:

```
/* Main function */
int main()
{
        /* Initialise external variables: */
        DIM = 2;

        /* Construct ODE: */
        ODE ode((functionODE)Brusselator);

        /* Set initial interval: */
        v_interval x0(DIM);
        x0[0] = interval(0.9999, 1.0001);
        x0[1] = interval(-0.0001, 0.0001);

        /* Set experience parameters: */
        Experiment exp = Experiment(ode, x0, 10, 0.001, 0.5, 0., 1.02, 0.00001);

        /* Run solving program: */
        exp.solveODE();

        return 0;
}
```

Here I was experimenting with the Brusselator ODE, which is quite difficult to have a good enclosure with such an ODE.

Hybrid system

For hybrid system, I tested the Bouncing Ball:

```
/* ODE definitions */
vector<T<interval>> BouncingBall(vector<T<interval>> x){
        vector<T<interval>> f = vector<T<interval>>(DIM);
        f[0] = x[1];
        f[1] = interval(-g);
        return f;
}

/* Modifications definitions */
v_interval Bump(v_interval x){
        v_interval new_x = v_interval(DIM);
        new_x[0] = x[0];
        new_x[1] = -interval(amortissement)*x[1];
        return new_x;
}
```

Here the bouncing ball system only has one state so it is not too laborious to represent the system in C++. But if there were n states, there must be also n ODE definitions and a maximum of n*n modification functions definitions (an identity function has to be defined if no modification is needed because one of those functions has to be applied for each automate state movement).

But even with a single state automate, it is still laborious to represent. Here is an example of a `main()` function:

```
/* Main function */
int main()
{
        /* Initialise external variables: */
        DIM = 2;
        STATES = 1;

        /* Construct System of ODE: */
        vector<ODE> system;
        ODE ode((functionODE)BouncingBall);
        system.push_back(ode);

        /* Construct inGards: */
        vector<v_interval> inGards;
        v_interval x_pos(DIM);
        x_pos[0] = interval(0., entire().sup());
        x_pos[1] = entire();
        inGards.push_back(x_pos);

        /* Construct outGards: */
        vector<vector<v_interval>> outGards;
        vector<v_interval> one_gards;
        v_interval x_zero(DIM);
        x_zero[0] = interval(0.);
        x_zero[1] = entire();
        one_gards.push_back(x_zero);
        outGards.push_back(one_gards);

        /* Construct outMod: */
        vector<vector<functionMod>> outMod;
        vector<functionMod> one_mod(1, (functionMod)Bump);
        outMod.push_back(one_mod);

        /* Set initial interval: */
        v_interval x0(DIM);
        x0[0] = interval(1., 1.1);
        x0[1] = interval(0., 0.);

        /* Set initial hybrid state: */
        vector<State> enclosure;
        State init_state = State(0, x0);
        enclosure.push_back(init_state);

        /* Construct Hybrid: */
        Hybrid hybrid = Hybrid(system, inGards, outGards, outMod, enclosure);

        /* Set experience parameters: */
        Experiment exp = Experiment(hybrid, x0, 10, 0.001, 0.5, 0., 1.0, 0.00001);

        /* Run solving program: */
        exp.solveHybrid();

        return 0;
}
```

End results

In this section I will present the results that we can reach using this project.

ODE

To begin with, we will look at the ODE reachability analysis of the presented examples. In all the examples, I first try to reach the end-time limit of the analysis for given parameters. This permit to have a quick look at where it begins to exponentially increase.

Brusselator

In order to reach the limit of the Brusselator ODE, I used the following parameters:

```
/* Set initial interval: */
v_interval x0(DIM);
x0[0] = interval(0.9999, 1.0001);
x0[1] = interval(-0.0001, 0.0001);

/* Set experience parameters: */
Experiment exp = Experiment(ode, x0, 10, 0.001, 0.5, 0., 10, 0.00001);
```

And it prints me the following ”enclosure”:

```
$ Tolerance not reached for the bouncing box computation
$ Infinite reached in the ODE solving at t=2.587000.
$ x0(10) = [ ENTIRE ]
$ x1(10) = [ ENTIRE ]
```

Followed by this graph:

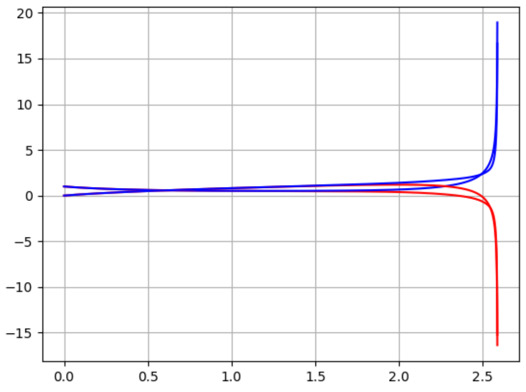

Figure 1: Brusselator Limit

Then I look where it could be good to have a reachability analysis, here I choose `t_end = 1.` and replaced it in the parameters. So I have a real enclosure of the system at `t = 1.`:

```
$ x0(1.0005) = [0.519, 0.524]
$ x1(1.0005) = [0.816, 0.82]
```

And the graph is more readable:

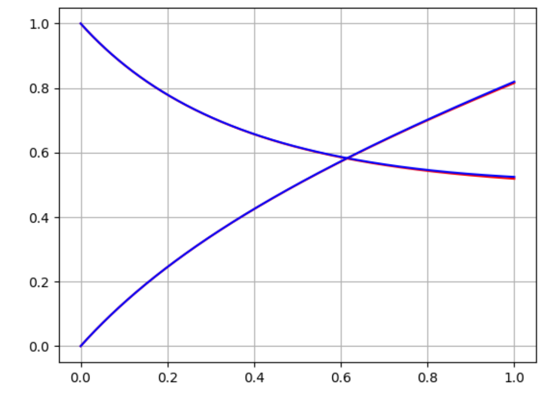

Figure 2: Brusselator

Cosinus

In order to reach the limit of the Cosinus ODE, I used the following parameters:

```
/* Set initial interval: */
v_interval x0(DIM);
x0[1] = interval(-0.0001, 0.0001);
x0[0] = interval(0.9999, 1.0001);

/* Set experience parameters: */
Experiment exp = Experiment(ode, x0, 10, 0.001, 0.5, 0., 2.5, 0.00001);
```

And it prints me the following enclosure:

```
  $ x0(2.5) = [-386, 384]
  $ x1(2.5) = [-2.42e+03, 2.42e+03]
```

Followed by this graph:

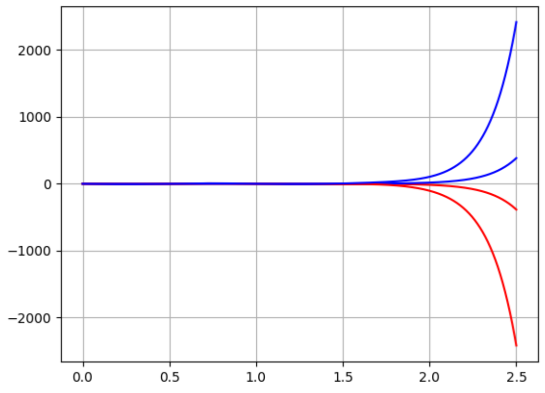

Figure 3: Cosinus Limit

The graph might be readable at `t = 1.` so I set `t_end = 1.` in the previous parameters:

```
$ x0(1.0005) = [0.969, 1.03]
$ x1(1.0005) = [-0.215, 0.176]
```

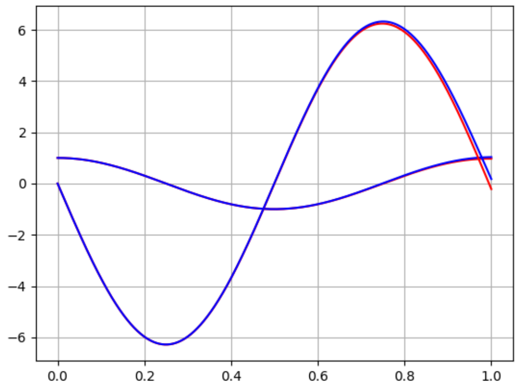

Figure 4: Cosinus

Exponential

In order to reach the limit of the Exponential ODE, I used the following parameters:

```
/* Set initial interval: */
v_interval x0(DIM);
x0[0] = interval(0.9999, 1.0001);

/* Set experience parameters: */
Experiment exp = Experiment(ode, x0, 10, 0.001, 0.5, 0., 10., 0.00001);
```

And it prints me the following enclosure which is enormous considering what it is really:

```
$ x0(10) = [-2.2, 2.2]
```

Followed by this graph:

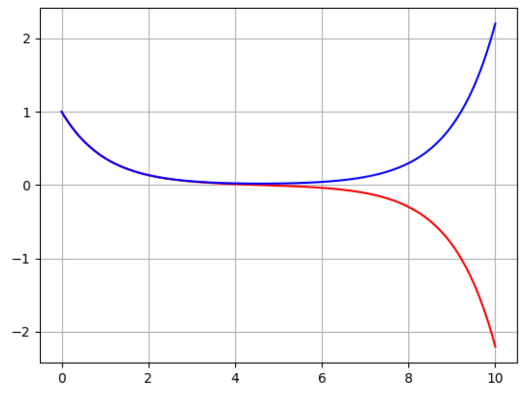

Figure 5: Exponential Limit

The graph might be readable at `t = 5.` so I set `t_end = 5.` in the previous parameters:

```
$ x0(5.0005) = [−0.00811, 0.0216]
```

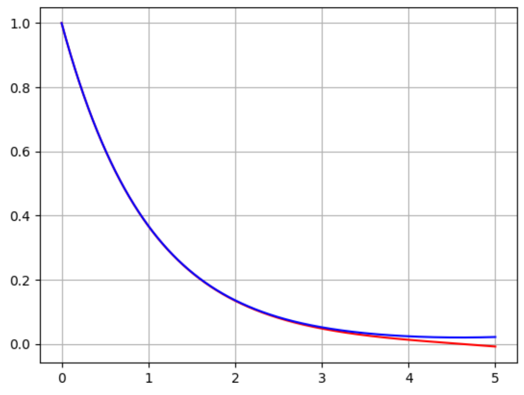

Figure 6: Exponential

Linear

For the linear example, I knew that it would be an unlimited reachability analysis:

```
/* Set initial interval: */
v_interval x0(DIM);
x0[0] = interval(0.9999, 1.0001);

/* Set experience parameters: */
Experiment exp = Experiment(ode, x0, 10, 0.001, 0.5, 0., 1000., 0.00001);
```

And it prints me the following enclosure which is the result:

```
$ x0(1000) = [1e+03, 1e+03]
```

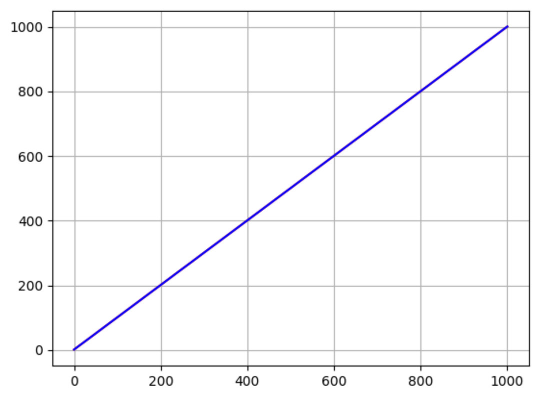

Figure 7: No Limit for Linear

Hybrid system

Bouncing Ball

I did the same process for the Bouncing Ball:

```
/* Set experience parameters: */
Experiment exp = Experiment(hybrid, x0, 10, 0.001, 0.5, 0., 5., 0.00001);
```

And the bouncing ball reached a surprising state:

```
$ x0(1.79769e+308) = [ EMPTY ]
$ x1(1.79769e+308) = [ EMPTY ]
```

I did not had the time to elude this problem for now.

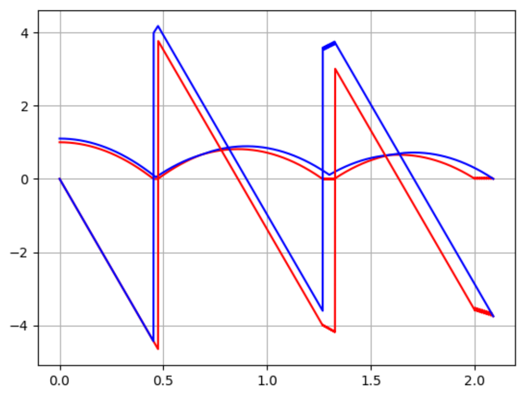

Figure 8: Bouncing Ball Limit

But we can see that at `t = 1.` we can have a printed enclosure of the bouncing ball system in the terminal, so I changed `t_end = 5.` to `t_end = 1.`:

```
$ x0(1.0005) = [0.713, 0.841]
$ x1(1.0005) = [-1.39, -0.989]
```

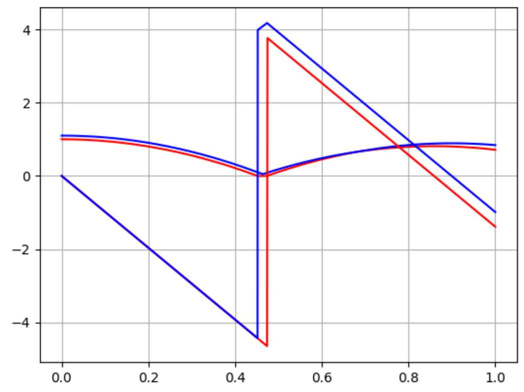

Figure 9: Bouncing Ball

Reliability

I first verified the enclosure for the simple ODEs (Cosinus and Exponential) and the results printed are enclosure of the "real" result (calculator result).

For the Linear, it is obviously a good "enclosure".

For the Brusselator, I can not compute the result without approximation methods.

For the Bouncing Ball, I can solve the system by hand:

1. until $x > 0$: $\begin{cases} \dot{x} = v \\ \dot{v} = -g \end{cases}$

2. ie until $x > 0$: $\begin{cases} x = 1 - \frac{g}{2}t^2 \\ v = -gt \end{cases}$

3. for $x = 0$: $\begin{cases} t = \sqrt{\frac{2}{g}} \\ v = -\sqrt{2g} \end{cases}$

4. now, for $x > 0$ and $t > \sqrt{\frac{2}{g}}$: $\begin{cases} \dot{x} = v \\ \dot{v} = -g \end{cases}$ but we have $\begin{cases} x = -\sqrt{2g}(t - \sqrt{\frac{2}{g}}) - \frac{g}{2}(t - \sqrt{\frac{2}{g}})^2 \\ v = 0.9 * \sqrt{2g} - g(t - \sqrt{\frac{2}{g}}) \end{cases}$

5. so for $t = 1.0005$ and $g = 9.80665$ I computed $x = 0.71048975284 \notin [0.713, 0.841]$ and $v = -1.39704127735 \notin [-1.39, -0.989]$.

To conclude, the hybrid system reachability analysis should be fixed because it does not compute an enclosure of the solution.

Observations

During the project, I observed that some times the intervals computed by `filib++` were bad approximations of the result. For example, when doing `cout << interval(1., 1.00001) << endl;` it would print $[1, 1]$. I hope it is just the printing that makes approximation and that the project is still fixable for hybrid systems.

Compare with VNODE

A comparison between reachability analysis of `VNODE` and this project will be done during the presentation.

Conclusion

This project was a good opportunity to compute a reachability analysis based on Taylor-Lagrange expansion. I used recently the reachability analysis package `Dynibex` which is based on Runge-Kutta schemes so it is a different approach (and it uses `ibex` for interval computation). Making my ”own” reachability analysis tool was a challenge and it did not succeed completely, but it permits me to make a whole project structure and organization.

Thank you for reading.