INF575 (Lab 6 & 7)

Quentin LAO

1 Script

In Lab6&7, we computed a bounded horizon Taylor expansion based reachability analysis for non linear continuous systems. The whole program has been written in the Python file `taylor.py`.

1.1 Running commands

Before running the file, make sure to have the right Python libraries installed. To install them, type the following command in the project directory : `pip3 install -r requirements.txt`.

To modify the function $f(x,t) = (f_1(x,t), ..., f_n(x,t))$, open the Python file `taylor.py` and modify the list `f` defined at the beginning of the code. Each element of the list is the string expression of a dimension f_i. The only variables accepted are `t` and `x_0, x_1, ...` which respectively correspond to variables t and $x := (x_0, ..., x_n) \in \mathbb{R}^n$. In addition, a multivariate interval $X_0 = I_1 \times ... \times I_n$ has to set to specify the possible initial values of x.
For example, the following code definies the function $f(x := (x_0, x_1), t) = (-x_0, x_0 + x_1)$ and bounds initially $x_0 \in [1, 2]$ and $x_1 \in [-1, 3]$:

```
#### DEFINITION OF f(x,t) : (R^n,R) -> R^n #####
#List of strings // Coordonate : t, x_0 , x_1 , ...
f = ["-x_0", "x_0 + x_1"]
X_0 = [ Interval(1,2) , Interval(-1,3) ]
################################
```

Then, run `python3 taylor.py` and enjoy.

1.2 Taylor model process

Let's briefly go through how the program works step by step. The main library that has been used is `Sympy` as it deals with symbolic mathematics.

1. Parse the list `f` so that it transforms the string list into expression matrix easy to handle for `Sympy` :

   ```
   f = parse_function(f)
   ```

2. For each Taylor step, we compute the first step that is finding a priori bounding box and valid step size with the function `priori_enclosures`. Then the second step (tightening) with the function `tightening`. I grouped these two latter functions into one called `make_step` :

   ```
   def make_step(f : Matrix, X : Matrix, I : list, order : int) -> (list, float) :
       """
          Compute a hole step in Taylor Model

          Returns the new interval for the next step (list) and the step h made
       """
       h,B = priori_enclosures( I = I , h = 1 , a = 0.1, f = f)
       tight = tightening( I = I , order = order, B = B, f = f, X = X)
       return reachable_state(tight, h) , h
   ```

3. Now, starting with an initial set `I = X_0`, we can iterate `make_step` in the "main" part of the python script as long as we don't reach `T` :

```
T = 10
t = 0
while t < T :
    I,h = make_step(f, X, I, order = 4)
    t += h
```

For simplicity, I choose to follow the same process as described in the lecture.

1.3 Additional tool functions

To support the previous functions, I implemented as well some additional functions such as :

- `Lie_derivative_repeat(f : Matrix, g : Matrix, n : int) -> list` : Compute the n first Lie derivatives of g with respect to f. In other words, it computes $\mathcal{L}_f^1(g), ..., \mathcal{L}_f^n(g)$ where :

$$\mathcal{L}_f^n(g) = \begin{cases} \mathcal{L}_f(g) := \sum_{i=1}^{n} \frac{\partial g}{\partial x_i} f_i + \frac{\partial g}{\partial t} & \text{if } n = 1 \\ \mathcal{L}_f(\mathcal{L}_f^{n-1}(g)) & \text{otherwise} \end{cases}$$

- `sum_it(a, b) -> Interval` and `prod_it(a, b) -> Interval` compute operations on Intervals. The function `sum_it` returns the sum of two intervals defined as follow :

$$[a, a'] \oplus [b, b'] := [a + b, a' + b']$$

The function `prod_it` computes the follow product of two intervals :

$$[a, a'] \otimes [b, b'] := [\min_{(i,j) \in \{a,a'\} \times \{b,b'\}} ij, \max_{(i,j) \in \{a,a'\} \times \{b,b'\}} ij]$$

- `range_function_multivariant(f : Matrix , bounds_list : list) -> list` returns the range interval a multivariant function $f(x_0, ..., x_n)$ on account of a list of intervals $I_1 \times \cdots \times I_n$ where I_k represents the bounds of the variable x_k.

- `reachable_state(coef_tightening_list : list, t : float) -> list` evaluates the final polynomial expression t after tightening. That is to say it calculates $\sum_{k=1}^{n-1} \frac{t^k}{k!} I_k + \frac{t^n}{n!} B$ where $I_1, ..., I_{n-1}, B$ are intervals stored in `coef_tightening_list`.
This function uses the Horner's method (very efficient evaluation of polynomial) to evaluate this polynomial expression.

1.4 Tests

Let's put into test our program.

1.4.1 Example from the lecture

Value of $x(1)$ In the lecture, we manually calculated a priori enclosure B for $t \in [0, 1]$. Then, we got a interval for $x(1)$ with a Taylor Model of order $k = 4$. We found $B = [0, 1]$, $h = 1$, $x(t) = 1 - t + \frac{t^2}{2} - \frac{t^3}{3!} + \frac{t^4}{4!}[0, 1]$ and $x(1) \in [\frac{1}{3}, \frac{1}{3} + \frac{1}{24}]$. Here is the data :

- $\dot{x} = -x$
- $x(0) = 1$

Now, let's modify the program parameters to fit with the data above :

```
#### DEFINITION OF f(x,t) : (R^n,R) -> R^n #####
#List of strings // Coordonate : t, x_0 , x_1 , ...
f = ["-x_0"]
X_0 = [ Interval(1,1)]
################################
```

Here are the outputs I got when I print the variables in `make_step` :

```
>>> print(h, B)
1 [Interval(0, 1)]
>>> print(tight)
[[{1}], [{-1}], [{1}], [{-1}], [Interval(0, 1)]]
>>> print( reachable_state(tight, h) )
[Interval(0.333333333333333, 0.375000000000000)]
```

Let's dig a big deeper of what all of this means :

- `print(h, B)` are the outputs of the first step `priori_enclosures`. The program successfully calculated B and the step size h.
- `print(tight)` prints the Interval coefficients of the Taylor expression in increasing order of polynomial orders. Thus, at the very left hand, `[{1}]` should be the interval coefficient in front of the constant coefficient which is the case. Remember, in the lecture, we got manually $x(t) = 1 - t + \frac{t^2}{2} - \frac{t^3}{3!} + \frac{t^4}{4!}[0, 1]$. The next `[{-1}]` corresponds to the interval coefficient in front of t, so one...
- `print( reachable_state(tight, h) )` evaluates the polynome at h (here $h = 1$) so it displays the possible value for $x(1)$. As we can see, the program found the same interval as the lecture did.

The whole Taylor process until an arbitrary T Let's see now how our program performs making steps until a time T is reached. The purpose of the example of the lecture is x's expression is mathematically calculable. In fact, $x(t) = e^{-t}$. So for each step, we can observe the error the program makes by over-approximating the possible values of x.

Let's fix $T = 4$ and plot for each step the interval values proposed by the program at different Taylor orders and the effective value of $x(t) = e^{-t}$ (cf figure 1).

As the time flows, the red curve and the blue curve are going further apart. This is because of wrapping effect : the approximation is becoming grosser and grosser step by step. Step i takes only input information from step $i - 1$ which could probably contains error. Therefore, at first sight, step i transfers the previous error to the next step, even worse it could increase it. Some algorithms attempt to reduce this wrapping effect such as the one proposed by Makino and Berz[1] in 2003. Most recently, Florian Bunger presents a new wrapping shrinker, with some simplifications[2] (cf figure 2).

The wrapping effect decreases as the Taylor order increases. It is actually expected because the program is more accurate at each step.

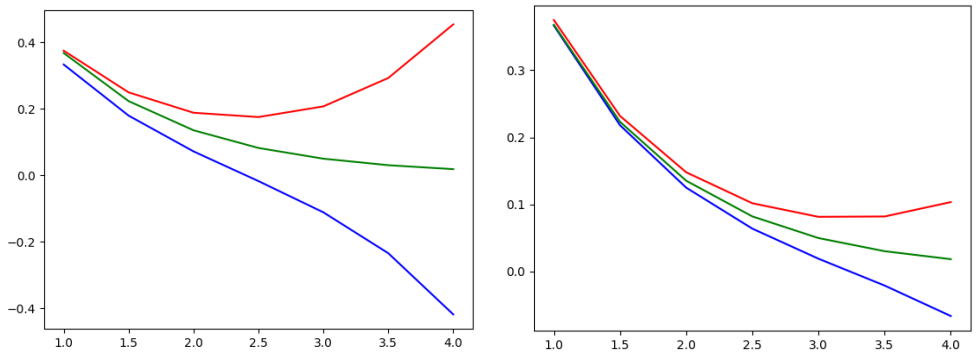
1.4.2 Variation of the lecture example

Instead of having a sure initial state, let's say $x(0)$ is in a certain interval $I = [1, 1 + \varepsilon]$ (where $\varepsilon > 0$. Let's fix the Taylor order $k = 4$ and $\varepsilon \in \{0.05, 0.1, 0.15, 0.2\}$. Here is what I get : (see figure 3)

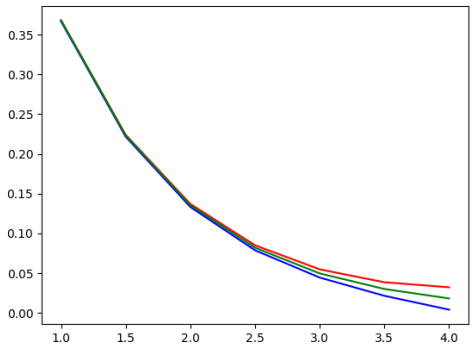
As we can see, the error is becoming greater as the size of the interval increases which is an issue. The error can be defined for instance as the area between the red and the blue curve that is not is the

[1] Hoefkens, J., Berz, M., Makino, K.: Controlling the Wrapping Effect in the Solution of ODEs for Asteroids. Reliab. Comput. 8, 21–41 (2003)

[2] Bünger, Florian (2017). Shrink wrapping for Taylor models revisited. Numerical Algorithms



(a) $k = 4$ (b) $k = 5$



(c) $k = 6$

Figure 1: Whole Taylor process of the lecture example (blue : lower bound, red : upper bound, green : $x(t) = e^{-t}$)

band defined by the 2 green curves i.e for the discret step of the Taylor process $0 = t_0 < ... < t_N$ (with $t_{N-1} < T < t_N$).:

$$\text{Error} = \sum_{i=0}^{N} \text{upper\_bound}(t_i) - \text{lower\_bound}(t_i) - \varepsilon e^{-t_i}$$

Here :

- $\varepsilon = 0$: Error = 2.14064720476888
- $\varepsilon = 0.05$: Error = 7.43774091774349
- $\varepsilon = 0.1$: Error = 14.4705765239056
- $\varepsilon = 0.15$: Error = 21.5082492679577
- $\varepsilon = 0.2$: Error = 28.5459546058894

The error looks to be linear with the size of the interval ε : Error $\sim 140 \times \varepsilon$.

1.5 Complexity

As the function `make_step` shows, each step can be divided in 3 parts :

- `priori_enclosures` has n-long loops ($O(n)$) and calls function `range_function_multivariant` several times ($O(n^2 M)$ where M is the complexity for Sympy to compute the minimum/maximum of a one-variable function).
- `tightening` calls function `range_function_multivariant` at each iteration of a k-loop (k is the Taylor order) : $O(kn^2 M)$

Input: A Taylor model n-vector $p + I$ which shall be shrink wrapped.

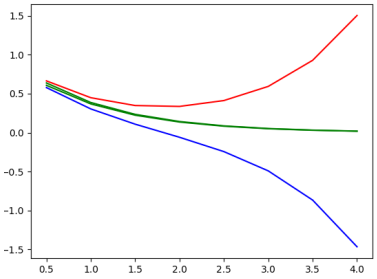
1. Set $c := p(0)$ and $p := p - c$ so that $p(0) = 0$ is valid in the sequel.
2. Decompose $p(x) = Ax + h(x)$, $x := (x_1, \ldots, x_n)^T$, with matrix $A \in \mathbb{R}^{n,n}$ and h having zero constant and linear terms, i.e., $h(0) = 0 = h'(0)$.
3. If A is not invertible or if its condition number is too large, then shrink wrapping is not applicable. Then, try other fallback strategies or return $p + I$ unchanged.
4. Compute an approximate inverse R of A.
5. Determine $r \in (\mathbb{R}_{\geq 0})^n$ such that $I \subseteq rB = [-r, r] \in \mathbb{IR}^n$ and set $\tilde{r} := |R|r$.
6. Compute $g(x) := Rp(x) - x$ and its Jacobian matrix $g'(x)$.
7. Estimate the shrink wrap vector q as follows with `rs`$:= \tilde{r}$, `dgi_dxj`$:= \frac{\partial g_i}{\partial x_j}$:

```
q_max = 1.01;  % heuristic bound for shrink wrap factors
q_tol = 1E-12; % heuristic minimum rate of change
iter_max = 3;  % heuristic upper bound for iterations
q = 1+rs;
improve = true;
iter = 0;
while improve && iter < iter_max
  s = zeros(size(rs));
  q_old = q;
  for i = 1:n
    for j = 1:n
      compute upper bound t of |dgi_dxj([-q,q])|
      s(i) = s(i) + t * (q(j)-1);
    end
    q(i) = 1 + rs(i) + s(i);
    if q(i) > q_max shrink wrapping seems not promising
                   -> try other fallback strategies
  end
  improve = any((q-q_old)./q > q_tol);
  iter = iter + 1;
end
```

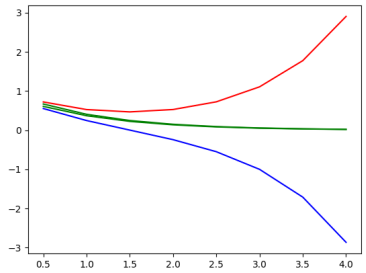
8. Relax s by multiplication with a factor slightly greater 1 and set $q := \mathbf{1} + \tilde{r} + s$.
9. Verify the new shrink wrap vector q a posteriori. If this is not successful, other fallback strategies might be used again.
10. Compute $\tilde{p}(x) := p(q_1 x_1, \ldots, q_n x_n) + c$.

Output: The shrink-wrapped Taylor model n-vector $\tilde{p} + J$ with an almost zero remainder $J \in \mathbb{IR}^n$ that originates from rounding errors only.

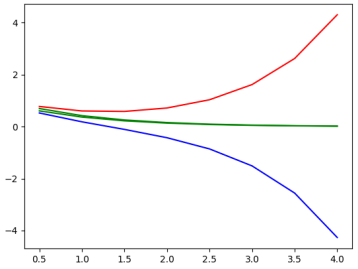
Figure 2: Pseudo-code of a recent wrapping shrinker in Florian Bunger's paper



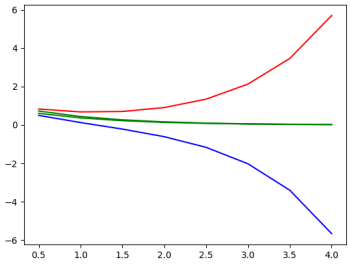
(a) $\varepsilon = 0.05$



(b) $\varepsilon = 0.1$



(c) $\varepsilon = 0.15$



(d) $\varepsilon = 0.2$

Figure 3: Variante of the lecture example with a initial interval $[1, 1 + \varepsilon]$ (blue : lower bound, red : upper bound, green curves : $x(t) = e^{-t}$ and $x(t) = (1 + \varepsilon)e^{-t}$)

- `reachable_state` computes Horner's method over each dimension : $O(kn)$

The overall complexity of the program is $O(kn^2M)$ which is quite adequate. The program could

probably be quicker by using other programming language such as C++. I choose Python for several reasons: first, it is the language I master the most. Second, Python has a lot of libraries like Sympy or numpy to plot curves that make our job easier but often at the expense of program effectiveness.

References

[1] Xin Chen, Erika Abraham, Sriram Sankaranarayanan (2012), Taylor Model Flowpipe Construction for Non-linear Hybrid Systems.

[2] Xin Chen (2015), Reachability Analysis of Non-Linear Hybrid Systems Using Taylor Models