DynIbex-2.0: The User Guide [1]

Julien Alexandre dit Sandretto, Alexandre Chapoutot, Olivier Mullier
ENSTA ParisTech, Palaiseau, France
{alexandre, chapoutot, mullier}@ensta.fr

November 8, 2018

[1]This research benefited from the support of the "Chair Complex Systems Engineering – Ecole Polytechnique, THALES, DGA, FX, DASSAULT AVIATION, DCNS Research, ENSTA ParisTech, Télécom ParisTech, Fondation ParisTech, FDO ENSTA"

Abstract

This document aims to present the release for DynIbex done in October 2018. It presents the main methods, the new features and the results of the experimentations on the Vericomp database.

Contents

Chapter 1

Main methods

This chapter gathers the main methods proposed in DynIbex.

1.1 Objects initialisation

The main objects are as follow.

1.1.1 Equation

The Ibex constructors `Variable` and `Function` are used to define the differential equation. For example, the Ordinary Differential Equation (ODE) $\dot{y} = f(y)$ defined as follows:

$$\begin{cases} \dot{y_1} = 2.0 * y_1 * (1.0 - y_2) \\ \dot{y_2} = -y_2 * (1.0 - y_1) \end{cases} \tag{1.1}$$

can be declared in DynIbex such that:

```
Variable y(2);
Function ydot(y, Return(2.0 * y[0]*(1.0 - y[1]),
                        -y[1] * (1.0 - y[0])));
```

A Differential Algebraic Equation (DAE) $\dot{y} = f(y,x); g(y,x) = 0$ defined as follows:

$$\begin{cases} \dot{y_1} = y_2 + x_1 \\ \dot{y_2} = y_1 - y_2 * x_1 \\ \dot{y_3} = y_1 * y_3 - x_2 \\ \quad y_1 - x_2 = 0 \\ \quad y_2 - 2 * x_1 = 0 \end{cases} \tag{1.2}$$

can be declared in DynIbex such that:

```
Variable x(2);
Variable y(3);
Function ydot = Function(y,x,Return(y[1]+x[0],
```

```
                                        y[0]-y[1]*x[0],
                                        y[0]*y[2]-x[1]));

Function g = Function(y,x,Return(y[0]-x[1],
                                 y[1]-2*x[0]));
```

1.1.2 Initial value problems

Two classes are available to declare an initial value problem (IVP): one for the ODEs and one for the DAEs.

For Ordinary Differential Equations

The class to be used is `ivp_ode`. The constructor comes with 5 parameters:

- the ODE, a `Function`
- the initial time, a `double`, generally 0.0
- the initial value, an `IntervalVector` or an `Affine2Vector`
- a set of constraints for a constrained ODE, an `Array<NumConstraint>` (Optional)
- the method used to compute the local truncation error, options are `AUTODIF`, `SYMBOLIC` or `AUTO`. See section "new feature" for more details (optional)

A simple example is, with the ODE declared previsouly:

```
IntervalVector yinit(2);
yinit[0] = Interval(1.0);
yinit[1] = Interval(0);
ivp_ode problem = ivp_ode(ydot,0.0,yinit);
```

For a constrained ODE $\dot{y} = f(y); h(y) = 0$, defined by:

$$\begin{cases} \dot{y_1} = -y_2 \\ \dot{y_2} = y_1 \\ \quad y_1^2 + y_2^2 = 1.0 \end{cases} \tag{1.3}$$

the following code can be written:

```
Variable y(2);
Function ydot(y, Return(-y[1],
                        y[0]));
NumConstraint csp1(y,sqr(y[0])+sqr(y[1]) -1.0  = 0);
Array<NumConstraint> csp(csp1);
ivp_ode problem = ivp_ode(ydot,0.0,yinit,csp);
```

For Differential Algebraic Equations

The class to be used is `ivp_dae_h1`. Only the DAEs in Hessenberg form of index 1 are available. The constructor comes with 6 parameters:

- the differential part of the DAE, a `Function`
- the constraint part of the DAE, a `Function`
- the initial time, a `double`, generally 0.0
- the initial value for the variable y, an `IntervalVector` or an `Affine2Vector`
- the initial value for the variable x, an `IntervalVector` or an `Affine2Vector`
- a set of constraints for a constrained DAE, an `Array<NumConstraint>` (Optional)

A simple example is, with the DAE declared previsouly:

```
ivp_dae_h1 problem = ivp_dae_h1(ydot,g,0.0,yinit,xinit);
```

1.1.3 Simulation

The most important class in DynIbex is `simulation`. It constructs and contains the result of a validated simulation. The constructor comes with 5 parameters:

- the initial value problem, `ivp_ode` or `ivp_dae_h1`
- the final time to reach by the simulation, a `double`
- a method for the integration scheme, the available ones are: `IMIDPOINT`, `RADAU3`, `HEUN`, `TAYLOR4`, `LA3`, `LC3`, `RK4`, `RADAU3_DAE` (this method is only for DAEs), `RADAU5`, `GL4`, `GL6`, `KUTTA3` (optional, `RK4` by default)
- a threshold on the LTE, the absolute tolerance, a `double` (optional, by default 10^{-6})
- an initial step-size, a `double` (optional, by default 0.001)

A simple example is, with the IVP with ODE declared previsouly:

```
simulation simu(&problem,8.0,LC3,1e-10);
```

and for DAE:

```
simulation simu(&problem,0.5,RADAU3_DAE, 1e-14);
```

1.2 Simulation and options

The main method of DynIbex from the class `simulation` is `run_simulation()`. It computes the solution of the initial value problem and stores the result (a tube). Following the previous exemple, the computation of a validated simulation is launched by:

```
simu.run_simulation();
```

Few methods on a `simulation` object allow to tune the simulation:

- `active_monotony()` and `inactive_monotony()` to choose to exploit or not the local monotony of the system to reduce the diameter of the boxes in the tube (it could increase the computation time).
- `getHmin()` and `setHmin()` to set and get the minimal stepsize authorized by the integration scheme
- `getHmax()` and `setHmax()` to set and get the maximal stepsize authorized by the integration scheme

Some methods allow to access to a specific element of the computed tube:

- `IntervalVector get_last()` and `Affine2Vector get_last_aff()` to acces to the last element in a box or in a zonotope
- `IntervalVector get(double t)` returns a box containing $y(t)$
- `IntervalVector get(Interval t)` returns a box containing $y(\tau), \tau \in t$
- `IntervalVector get_tight(double t)` returns a box containing $y(t)$ after an additional integration scheme to reduce as much as possible the diameter of the solution
- `IntervalVector get_attractor()` returns a box contained in the next box (possibly an attractor)
- `IntervalVector get_domain()` returns the domain covered by the simulation (the hull of $\{y(\tau), \tau \in [t_0, T]\}$)

It is also easy to export the result of a simulation in a file with the following methods:

- `void export2d_yn(const char* filename, int a, int b)` exports the dimension a and b of each discretized instant
- `void export3d_yn(const char* filename, int a, int b, int c)` exports the dimension a, b and c of each discretized instant
- `void export1d_yn(const char* filename, int a)` exports the dimension a of each discretized instant
- `void export_y0(const char* filename)` exports the Picard boxes
- `void export_yn(const char* filename)` export all dimensions off discretized boxes wrt time

1.3 Constraints on tube

This section gathers the methods used in the constraint satisfaction differential problem as proposed in [2]. The available methods are:

- `bool finished_in(IntervalVector y_final)` checks if the final solution is included in a box

- `bool finished_in(std::list<IntervalVector> *stack)` checks if final solution is included in at least one box of a list
- `bool has_crossed(IntervalVector y)` checks if the tube crosses a box
- `Interval has_crossed_when(IntervalVector& y)` returns the first instant (in an interval) when the tube crosses a box
- `bool has_crossed_before(IntervalVector& y, double time)` tests if the tube crosses a box before a given time
- `bool stayed_in(IntervalVector y_hull)` checks if the tube stays in a box
- `bool go_out(IntervalVector y_hull)` checks if at least one element of the tube is outside a box
- `bool stayed_in_till(IntervalVector y_hull, double t)` checks if the tube stays in a box till a given time
- `bool has_reached(IntervalVector y_final)` checks if the final solution crosses a box
- `bool has_reached(std::list<IntervalVector> *stack)` checks if the final solution crosses at least one box of a list
- `double one_in(std::list<IntervalVector> *stack)` checks if one solution is inside at least one box of a list (return the instant or -1)

Chapter 2

New feature

2.1 Local Truncation Error Computation

A guaranteed set-membership simulation of an ordinary differential equation requires to bound the *local truncation error* (LTE) of the method. The previous version provided such bound using symbolic derivation. This new version features the computation of the LTE using algorithmic differentiation (A.K.A. automatic differentiation). Using the result of the work in [1], computation of the LTE is chosen between symbolic and algorithmic differentiation according to the dimension of the ordinary differential equation state space.

2.2 Zonotopic simulation

From the first version, DynIbex uses zonotopes to compute the solution of differential equations. Nevertheless, it was not easy to construct a zonotope and provide it as initial state. We add the following method to initialize a zonotope with a specific center, a list of noise symbols and a garbage:
`initialize(double x0, std::list<std::pair<int,double> > xn, Interval xg)`

Combined with the getter `Affine2Vector get_last_aff()`, it is now easy to give a zonotope as initial state and collect a zonotope as final state.

Chapter 3

Experimentations

Bibliography

[1] Olivier Mullier, Alexandre Chapoutot, and Julien Alexandre dit Sandretto. Validated computation of the local truncation error of runge–kutta methods with automatic differentiation. *Optimization Methods and Software*, pages 1–11, 2018.

[2] Julien Alexandre Dit Sandretto, Alexandre Chapoutot, and Olivier Mullier. Formal verification of robotic behaviors in presence of bounded uncertainties. In *First IEEE International Conference on Robotic Computing, IRC 2017, Taichung, Taiwan, April 10-12, 2017*, pages 81–88, 2017.