Set-based Simulation and Control Programs

Alexandre Chapoutot

joint work with Julien Alexandre dit Sandretto and Olivier Mullier
U2IS, ENSTA ParisTech, Palaiseau, France

FEANICSES Meeting
May 25, 2018

Context

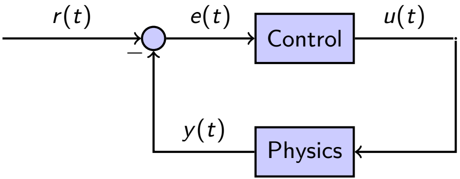
Context

Interval analysis

Validated numerical integration

Differential constraint satisfaction problems

A small cyber-physical system: closed-loop control



- **Physics** is usually defined by non-linear differential equations

$$\dot{\mathbf{x}} = f(\mathbf{x}(t), u(t)) \ , \qquad \mathbf{y}(t) = g(\mathbf{x}(t))$$

- **Control** may be a continuous-time PI algorithm

$$e(t) = r(t) - y(t) \ , \qquad u(t) = K_p e(t) + K_i \int_0^t e(\tau) d\tau$$

What is designing/synthesizing a controller?

Find values for K_p and K_i such that a **given specification** is satisfied.

Many classes of differential equations

1. Ordinary Differential Equations (ODE), *e.g.*,

$$\dot{\mathbf{y}}(t) = \mathbf{f}(t, \mathbf{y}(t))$$

2. Differential-Algrebraic equations (DAE), *e.g.*, semi-explicit DAE of index 1

$$\begin{aligned} \dot{\mathbf{y}}(t) &= \mathbf{f}(t, \mathbf{y}(t), \mathbf{x}(t)) \\ 0 &= \mathbf{g}(t, \mathbf{y}(t), \mathbf{x}(t)) \end{aligned}$$

3. Delay Differential Equations (DDE), *e.g.*,

$$\dot{\mathbf{y}}(t) = \mathbf{f}(t, \mathbf{y}(t), \mathbf{y}(t - \tau))$$

4. Sampled Switched Systems, *e.g.*,

$$\dot{\mathbf{y}}(t) = f_{\sigma(t)}(\mathbf{y}(t))$$

   with a piecewise constant switching rule $\sigma(t)$ updated every τ

5. and others: partial differential equations, hybrid systems, etc.

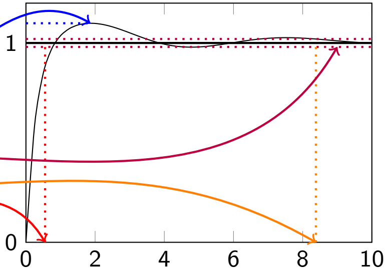
Note: DynIBEX can handle case 1, 2, 4

Specification of PID Controllers

PID controller: requirements based on closed-loop response

We observe the output of the plant

- Overshoot: Less than 10%
- Steady-state error: Less than 2%
- Settling time: Less than $10s$
- Rise time: Less than $2s$



Note: such properties come from the **asymptotic behavior** of the closed-loop system.

Classical method to study/verify closed-loop systems

Numerical simulations but

- do not take into account that models are only an approximation;
- produce approximate results.

and not adapted to deal with uncertainties

A global approach for verification or synthesis

Input

- a mathematical description of dynamical systems (ODE, DAE, etc.)
- specifications to fulfill or properties to verify

Output

- yes/no answer

Main algorithm

1. compute trajectories
2. check properties

But should take into account when computing trajectories

- uncertainties on mathematical models
- uncertainties on data
- approximation in numerical methods

which have an impact on how to express properties/specification $\Rightarrow$ **set-based approach**

Set-based simulation

Definition

numerical simulation methods implemented with interval analysis methods

Goals

takes into account various uncertainties (bounded) or approximations to produce rigorous results

Example

A simple nonlinear dynamics of a car

$$\dot{v} = \frac{-50.0v - 0.4v^2}{m} \qquad \text{with} \quad m \in [990, 1010] \quad \text{and} \quad v(0) \in [10, 11]$$

One Implementation **DynIBEX**: a combination of **CSP solver** (IBEX[1]) with **validated numerical integration methods** based on **Runge-Kutta**

`http://perso.ensta-paristech.fr/~chapoutot/dynibex/`

[1] Gilles Chabert (EMN) et al. `http://www.ibex-lib.org`

Interval analysis

Context

Interval analysis

Validated numerical integration

Differential constraint satisfaction problems

Basics of interval analysis

- **Interval arithmetic** (defined also for: sin, cos, etc.):

$$\begin{aligned}[\underline{x},\overline{x}] + [\underline{y},\overline{y}] =& [\underline{x}+\underline{y},\overline{x}+\overline{y}] \\ [\underline{x},\overline{x}] * [\underline{y},\overline{y}] =& [\min\{\underline{x}*\underline{y},\underline{x}*\overline{y},\overline{x}*\underline{y},\overline{x}*\overline{y}\}, \\ & \max\{\underline{x}*\underline{y},\underline{x}*\overline{y},\overline{x}*\underline{y},\overline{x}*\overline{y}\}]\end{aligned}$$

- Let an **inclusion function** $[f] : \mathbb{IR} \to \mathbb{IR}$ for $f : \mathbb{R} \to \mathbb{R}$ is defined as:

$$\{f(a) \mid \exists a \in [I]\} \subseteq [f]([I])$$

with $a \in \mathbb{R}$ and $I \in \mathbb{IR}$.

Example of inclusion function: Natural inclusion

$[x] = [1, 2]$, $[y] = [-1, 3]$, and $f(x, y) = xy + x$

$$[f]\big([x],[y]\big) := [x] * [y] + [x]$$

$$= [1, 2] * [-1, 3] + [1, 2] = [-2, 6] + [1, 2] = [-1, 8]$$

Numerical Constraint Satisfaction Problems

NCSP

A NCSP $(\mathcal{V}, \mathcal{D}, \mathcal{C})$ is defined as follows:

- $\mathcal{V} := \{v_1, \ldots, v_n\}$ is a finite set of variables which can also be represented by the vector $\mathbf{v}$;
- $\mathcal{D} := \{[v_1], \ldots, [v_n]\}$ is a set of intervals such that $[v_i]$ contains all possible values of v_i. It can be represented by a box $[\mathbf{v}]$ gathering all $[v_i]$;
- $\mathcal{C} := \{c_1, \ldots, c_m\}$ is a set of constraints of the form $c_i(\mathbf{v}) \equiv f_i(\mathbf{v}) = 0$ or $c_i(\mathbf{v}) \equiv g_i(\mathbf{v}) \leqslant 0$, with $f_i : \mathbb{R}^n \to \mathbb{R}$, $g_i : \mathbb{R}^n \to \mathbb{R}$ for $1 \leqslant i \leqslant m$.
  Note: Constraints $\mathcal{C}$ are interpreted as a conjunction of equalities and inequalities.

Remark: The solution of a NCSP is a valuation of $\mathbf{v}$ ranging in $[\mathbf{v}]$ and satisfying the constraints $\mathcal{C}$.

Example

- $\mathcal{V} = \{x\}$
- $\mathcal{D}_x = \big\{[1, 10]\big\}$ $\Longrightarrow$ $x \in [1, 1.09861]$
- $\mathcal{C} = \big\{x \exp(x) \leqslant 3\big\}$

Remark: if $[\mathbf{v}] = \emptyset$ then the problem is not satistafiable

Interval constraints and contractor

Interval constraint

Given a inclusion function $[f]$, a box $[\mathbf{z}]$, we look for a box $[\mathbf{x}]$, s.t.

$$[f]([\mathbf{x}]) \subseteq [\mathbf{z}]$$

A simple resolution algorithm

```
put [x] in a list X
while X is not empty
   take [x] in X
   if [f]([x]) is included in [z] then keep [x] in S as a solution
   else if width([x]) < tol then split [x], put [x1] and [x2] in X
```

Contractor

A contractor $\mathcal{C}_{[f],[\mathbf{z}]}$ associated to constraint $[f]([\mathbf{x}]) \subseteq [\mathbf{z}]$ such that

- Reduction:

$$\mathcal{C}_{[f],[\mathbf{z}]}\left([\mathbf{x}]\right) \subseteq [\mathbf{x}]$$

- Soundness:

$$[f]\left([\mathbf{x}]\right) \cap [\mathbf{z}] = [f]\left(\mathcal{C}_{[f],[\mathbf{z}]}\left([\mathbf{x}]\right)\right) \cap [\mathbf{z}]$$

Note: several contractor algorithms exist, *e.g.*, FwdBwd, 3BCID, etc.

Contractor: example FwdBwd

Example

- $\mathcal{V} = \{x, y, z\}$
- $\mathcal{D} = \big\{[1,2], [-1,3], [0,1]\big\}$
- $\mathcal{C} = \big\{x + y = z\big\}$

Forward evaluation

- $[z] = [z] \cap ([x] + [y])$

  as $[x] + [y] = [1,2] + [-1,3] = [0,5]$ $\quad \Rightarrow [z] = [0,1] \cap [0,5]$ No improvement yet

Backward evaluation

- $[y] = [y] \cap ([z] - [x]) = [-1,3] \cap [-2,0] = [-1,0]$ Refinement of $[y]$
- $[x] = [x] \cap ([z] - [y]) = [1,2] \cap [0,2] = [1,2]$ No refinement of $[x]$

Remark: this process can be iterated until a fixpoint is reached

Remark: the order of constraints is important for a fast convergence

IBEX in one slide

- Easy definition of functions
- Numerical constraints
- Pruning methods
- Interval evaluation of functions

```
#include "ibex.h"

using namespace std;
using namespace ibex;

int main() {

  Variable x;
  Function f (x, x*exp(x));

  NumConstraint c1(x, f(x) <= 3.0);

  CtcFwdBwd contractor(c1);

  IntervalVector box(1);
  box[0]=Interval(1,10);

  cout << "f" << box << " = " << f.eval(box) << endl;
  contractor.contract(box);
  cout << "after contraction box = " << box << endl;
}
```

IBEX is also a parametric solver of constraints, an optimizer, etc.

Contractor: example Newton operator

Example

- $\mathcal{V} = \{x\}$
- $\mathcal{D} = \left\{[1, 2]\right\}$
- $\mathcal{C} = \left\{x^2 - 2 = 0\right\}$

Newton operator (uni dimensional case)

$$\mathcal{N}([x]) = [x] \cap (m - \frac{f(c)}{f'([x])})$$

m is the midpoint of $[x]$
Property
if $\mathcal{N}([x]) \subseteq \mathrm{int}([x])$ then there exists a unique fixed point (Brouwer fixed-point theorem)

Remark: this operator can be iterated until a fixpoint is reached

Example of Newton operator

```
#include "ibex.h"

using namespace ibex;
using namespace std;

int main(){

  Variable x;

  Function f(x,sqr(x)-2);

  Function df(f,Function::DIFF);

  IntervalVector box(1);
  box[0] = Interval(1,2);

  cout << box << endl;
  for (int i = 0; i < 3; i++) {
    box[0] &= box[0].mid() - f.eval(box.mid()) / df.eval(box);
    cout << box << endl;
  }

  return 0;
}
```

Output:

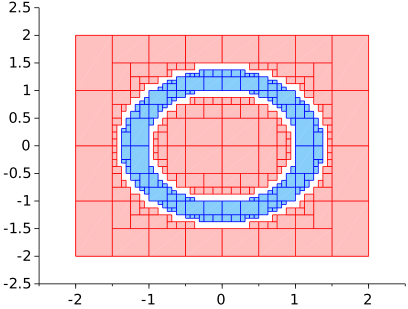
$$([1,2])$$
$$([1.375, 1.4375])$$
$$([1.41406, 1.41442])$$
$$([1.41421, 1.41421])$$

Paving

Methods used to represent complex sets $\mathcal{S}$ with

- inner boxes, *i.e.*, set of boxes included in $\mathcal{S}$
- outer boxes, *i.e.*, set of boxes that does not belong to $\mathcal{S}$
- the frontier, *i.e.*, set of boxes we do not know

Example, a ring $\mathcal{S} = \left\{(x, y) \mid x^2 + y^2 \in [1, 2]\right\}$ over $[-2, 2] \times [-2, 2]$



Remark: involving bisection algorithm and so complexity is exponential in the size of the state space (`contractor programming` to overcome this).

Validated numerical integration

Context

Interval analysis

Validated numerical integration

Differential constraint satisfaction problems

Initial Value Problem of Ordinary Differential Equations

Consider an IVP for ODE, over the time interval $[0, T]$

$$\dot{\mathbf{y}} = f(\mathbf{y}) \quad \text{with} \quad \mathbf{y}(0) = \mathbf{y}_0$$

IVP has a unique solution $\mathbf{y}(t; \mathbf{y}_0)$ if $f : \mathbb{R}^n \to \mathbb{R}^n$ is Lipschitz in $\mathbf{y}$
but for our purpose we suppose f smooth enough, *i.e.*, of class C^k

Goal of numerical integration

- Compute a sequence of time instants: $t_0 = 0 < t_1 < \cdots < t_n = T$
- Compute a sequence of values: $\mathbf{y}_0, \mathbf{y}_1, \ldots, \mathbf{y}_n$ such that

$$\forall i \in [0, n], \quad \mathbf{y}_i \approx \mathbf{y}(t_i; \mathbf{y}_0) \ .$$

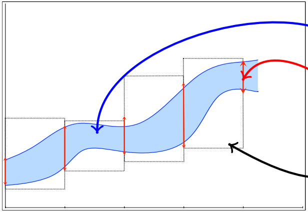
Validated solution of IVP for ODE

Goal of validated numerical integration

- Compute a sequence of time instants: $t_0 = 0 < t_1 < \cdots < t_n = T$
- Compute a sequence of values: $[\mathbf{y}_0], [\mathbf{y}_1], \ldots, [\mathbf{y}_n]$ such that

$$\forall i \in [0, n], \quad [\mathbf{y}_i] \ni \mathbf{y}(t_i; \mathbf{y}_0) \ .$$

A two-step approach



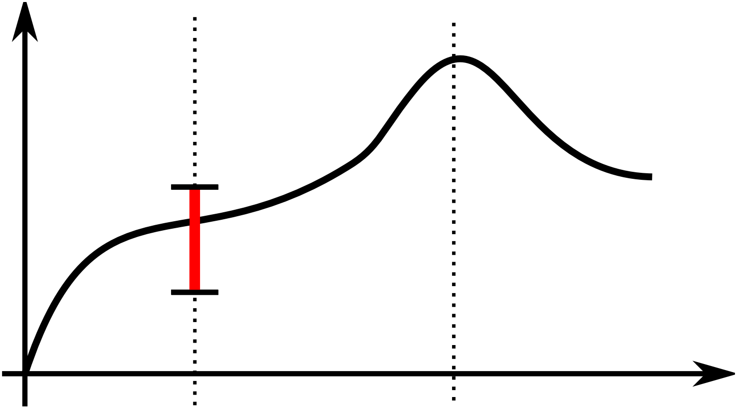
- Exact solution of $\dot{\mathbf{y}} = f(\mathbf{y}(t))$ with $\mathbf{y}(0) \in \mathcal{Y}_0$
- Safe approximation at discrete time instants
- Safe approximation between time instants

A priori enclosure: computation of $[\tilde{\mathbf{y}}]_\ell$

- Based on Picard-Lindelöf operator (naive approach)

$$\Psi([\mathbf{e}]) := [\mathbf{y}]_\ell + [0, h].\mathbf{f}([\mathbf{e}])$$

- If one has $[\mathbf{e}]_1$ such that $\Psi([\mathbf{e}]_1) \subseteq [\mathbf{e}]_1$, then one has a unique solution on $[t_\ell, t_\ell + h]$ **and** this solution is enclosed in $[\mathbf{e}]_1$.

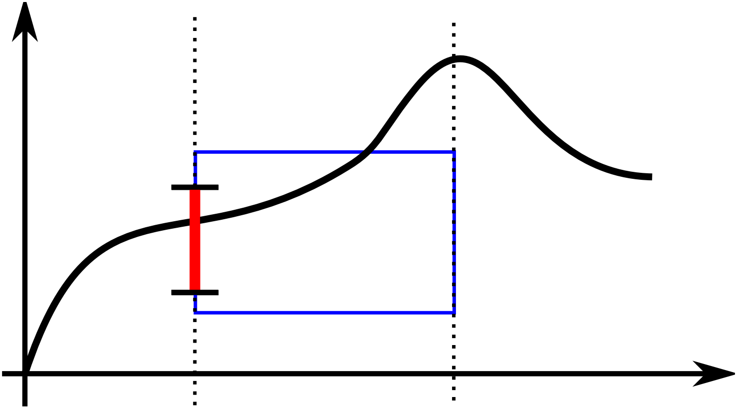


A priori enclosure: computation of $[\tilde{\mathbf{y}}]_\ell$

- Based on Picard-Lindelöf operator (naive approach)

$$\Psi([\mathbf{e}]) := [\mathbf{y}]_\ell + [0, h].\mathbf{f}([\mathbf{e}])$$

- If one has $[\mathbf{e}]_1$ such that $\Psi([\mathbf{e}]_1) \subseteq [\mathbf{e}]_1$, then one has a unique solution on $[t_\ell, t_\ell + h]$ **and** this solution is enclosed in $[\mathbf{e}]_1$.

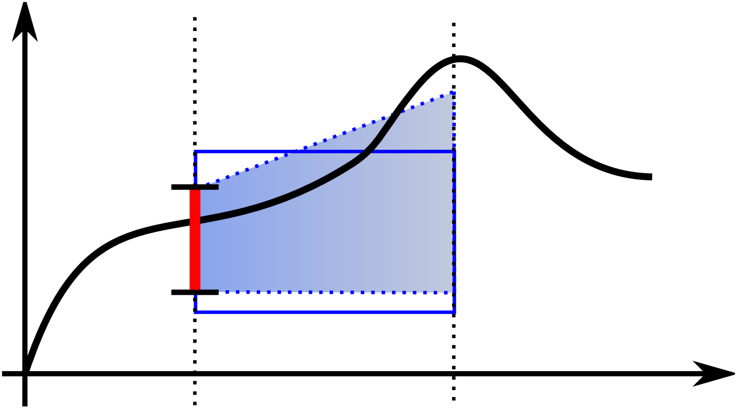


A priori enclosure: computation of $[\tilde{\mathbf{y}}]_\ell$

- Based on Picard-Lindelöf operator (naive approach)

$$\Psi([\mathbf{e}]) := [\mathbf{y}]_\ell + [0, h].\mathbf{f}([\mathbf{e}])$$

- If one has $[\mathbf{e}]_1$ such that $\Psi([\mathbf{e}]_1) \subseteq [\mathbf{e}]_1$, then one has a unique solution on $[t_\ell, t_\ell + h]$ **and** this solution is enclosed in $[\mathbf{e}]_1$.

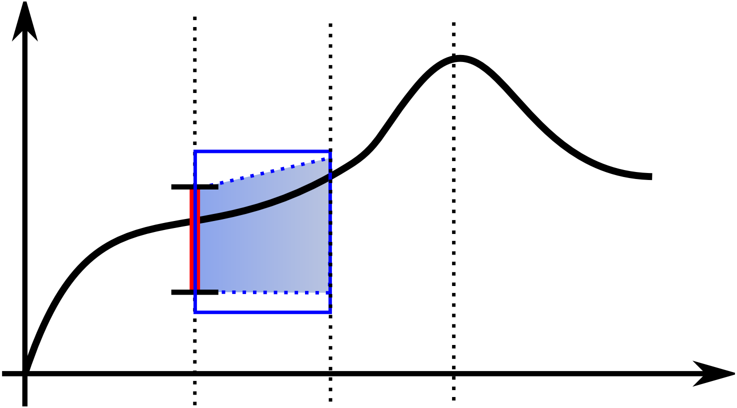


A priori enclosure: computation of $[\tilde{\mathbf{y}}]_\ell$

- Based on Picard-Lindelöf operator (naive approach)

$$\Psi([\mathbf{e}]) := [\mathbf{y}]_\ell + [0, h].\mathbf{f}([\mathbf{e}])$$

- If one has $[\mathbf{e}]_1$ such that $\Psi([\mathbf{e}]_1) \subseteq [\mathbf{e}]_1$, then one has a unique solution on $[t_\ell, t_\ell + h]$ **and** this solution is enclosed in $[\mathbf{e}]_1$.



A priori enclosure: computation of $[\tilde{\mathbf{y}}]_\ell$

- Based on Picard-Lindelöf operator (naive approach)

$$\Psi([\mathbf{e}]) := [\mathbf{y}]_\ell + [0, h].\mathbf{f}([\mathbf{e}])$$

- If one has $[\mathbf{e}]_1$ such that $\Psi([\mathbf{e}]_1) \subseteq [\mathbf{e}]_1$, then one has a unique solution on $[t_\ell, t_\ell + h]$ **and** this solution is enclosed in $[\mathbf{e}]_1$.

Note on the Variation of the step-size

In function of

- the Picard-Lindelöf
- the size of the Local Truncation Error

Runge-Kutta validated methods

Numerical solutions of IVP for ODEs are produced by

- Adams-Bashworth/Moulton methods
- BDF methods
- Runge-Kutta methods
- etc.

each of these methods is adapted to a particular class of ODEs/DAEs

Runge-Kutta methods

- have **strong stability** properties for various kinds of problems (A-stable, L-stable, algebraic stability, etc.)
- may **preserve quadratic algebraic invariant** (symplectic methods)
- can produce **continuous output** (polynomial approximation of $\mathbf{y}(t; \mathbf{y}_0)$)

We try to benefit these properties in validated computations

Examples of Runge-Kutta methods

Single-step fixed step-size explicit Runge-Kutta method

e.g. explicit Trapzoidal method (or Heun's method)[2] is defined by:

$$\mathbf{k}_1 = f(t_\ell, \mathbf{y}_\ell) \ , \qquad \mathbf{k}_2 = f(t_\ell + 1h, \mathbf{y}_\ell + h1\mathbf{k}_1)$$

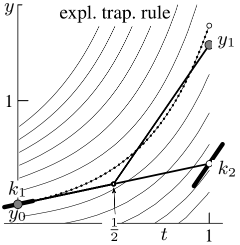
$$\mathbf{y}_{i+1} = \mathbf{y}_\ell + h\left(\frac{1}{2}\mathbf{k}_1 + \frac{1}{2}\mathbf{k}_2\right)$$

$$\begin{array}{c|cc} 0 & & \\ 1 & 1 & \\ \hline & \frac{1}{2} & \frac{1}{2} \end{array}$$

Intuition

- $\dot{y} = t^2 + y^2$
- $y_0 = 0.46$
- $h = 1.0$

dotted line is the exact solution.



[2] example coming from "Geometric Numerical Integration", Hairer, Lubich and Wanner.

Validated Runge-Kutta methods

A validated algorithm

$$[\mathbf{y}_{\ell+1}] = [\mathsf{RK}]\,(h, [\mathbf{y}_\ell]) + \mathsf{LTE}\ .$$

Challenges

1. Computing with sets of values (intervals) taking into account dependency problem and wrapping effect;
2. Bounding the approximation error of Runge-Kutta formula.

Our approach

- **Problem 1** is solved using **affine arithmetic** replacing centered form and QR decomposition
- **Problem 2** is solved by bounding the **Local Truncation Error** (LTE) of Runge-Kutta methods based on **B-series**

Order condition for Runge-Kutta methods

Method order of Runge-Kutta methods and Local Truncation Error (LTE)

$$\mathbf{y}(t_\ell; \mathbf{y}_{\ell-1}) - \mathbf{y}_\ell = C \cdot h^{p+1} \quad \text{with} \quad C \in \mathbb{R}.$$

we want to bound this!

Order condition

This condition states that a method of Runge-Kutta family is of order p **iff**

- the Taylor expansion of the exact solution
- and the Taylor expansion of the numerical methods

have the same $p+1$ first coefficients.

Consequence

The LTE is the **difference of Lagrange remainders of two Taylor expansions**

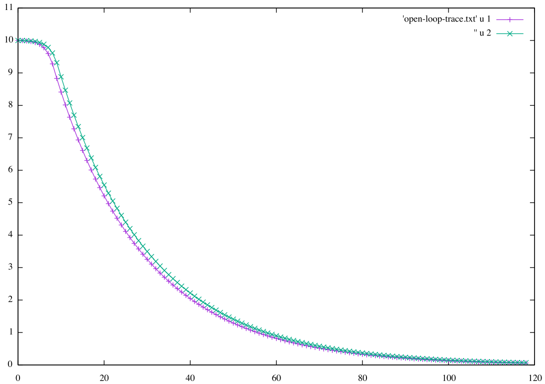
. . . but how to compute it?
using tools coming from Butcher's theory

Simulation of an open loop system

A simple dynamics of a car

$$\dot{y} = \frac{-50.0y - 0.4y^2}{m} \qquad \text{with} \quad m \in [990, 1010]$$

Simulation for 100 seconds with $y(0) = 10$



The last step is $y(100) = [0.0591842, 0.0656237]$

Simulation of an open loop system

- ODE definition
- IVP definition
- Parametric simulation engine

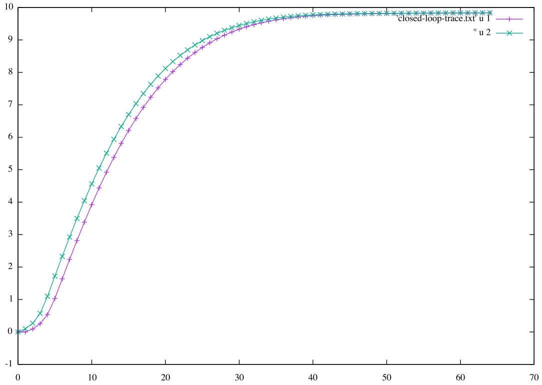
```
int main(){

  const int n = 1;
  Variable y(n);

  IntervalVector state(n);
  state[0] = 10.0;

  // Dynamique d'une voiture avec incertitude sur sa masse
  Function ydot(y, ( -50.0 * y[0] - 0.4 * y[0] * y[0])
                    / Interval (990, 1010));
  ivp_ode vdp = ivp_ode(ydot, 0.0, state);

  // Integration numerique ensembliste
  simulation simu = simulation(&vdp, 100, RK4, 1e-5);
  simu.run_simulation();

  //For an export in order to plot
  simu.export1d_yn("export-open-loop.txt", 0);

  return 0;
}
```

Simulation of a closed-loop system

A simple dynamics of a car with a PI controller

$$\begin{pmatrix} \dot{y} \\ \dot{w} \end{pmatrix} = \begin{pmatrix} \frac{k_p(10.0-y)+k_i w-50.0y-0.4y^2}{m} \\ 10.0-y \end{pmatrix} \quad \text{with} \quad m \in [990, 1010], k_p = 1440, k_i = 35$$

Simulation for 10 seconds with $y(0) = w(0) = 0$



The last step is $y(10) = [9.83413, 9.83715]$

Simulation of a closed-loop system

```
#include "ibex.h"

using namespace ibex;

int main(){

  const int n = 2;
  Variable y(n);

  IntervalVector state(n);
  state[0] = 0.0;
  state[1] = 0.0;

  // Dynamique d'une voiture avec incertitude sur sa masse + PI
  Function ydot(y, Return ((1440.0 * (10.0 - y[0]) + 35.0 * y[1]  - y[0] * (50.0  + 0.4 * y[0]))
                          / Interval (990, 1010),
                          10.0 - y[0]));
  ivp_ode vdp = ivp_ode(ydot, 0.0, state);

  // Integration numerique ensembliste
  simulation simu = simulation(&vdp, 10.0, RK4, 1e-7);
  simu.run_simulation();

  simu.export1d_yn("export-closed-loop.txt", 0);

  return 0;
}
```

Simulation of an hybrid closed-loop system

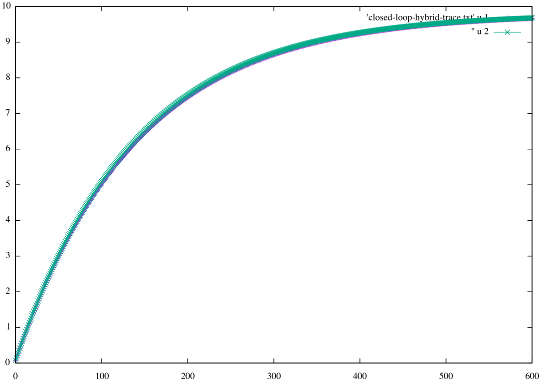
A simple dynamics of a car with a discrete PI controller

$$\dot{y} = \frac{u(k) - 50.0y - 0.4y^2}{m} \qquad \text{with} \quad m \in [990, 1010]$$

$$i(t_k) = i(t_{k-1}) + h(c - y(t_k)) \qquad \text{with} \quad h = 0.005$$

$$u(t_k) = k_p(c - y(t_k)) + k_i i(t_k) \qquad \text{with} \quad k_p = 1400, k_i = 35$$

Simulation for 3 seconds with $y(0) = 0$ and $c = 10$



Simulation of an hybrid closed-loop system

```
#include "ibex.h"

using namespace ibex;
using namespace std;

int main(){
  const int n = 2; Variable y(n);
  Affine2Vector state(n);
  state[0] = 0.0; state[1] = 0.0;

  double t = 0; const double sampling = 0.005;
  Affine2 integral(0.0);

  while (t < 3.0) {
    Affine2 goal(10.0);
    Affine2 error = goal - state[0];

    // Controleur PI discret
    integral = integral + sampling * error;
    Affine2 u = 1400.0 * error + 35.0 * integral;
    state[1] = u;

    // Dynamique d'une voiture avec incertitude sur sa masse
    Function ydot(y, Return((y[1] - 50.0 * y[0] - 0.4 * y[0] * y[0])
                       / Interval (990, 1010), Interval(0.0)));
    ivp_ode vdp = ivp_ode(ydot, 0.0, state);

    // Integration numerique ensembliste
    simulation simu = simulation(&vdp, sampling, RK4, 1e-6);
    simu.run_simulation();

    // Mise a jour du temps et des etats
    state = simu.get_last(); t += sampling;
  }
```

- Manual handling of discrete-time evolution

Differential Algebraic Equations

Index-1 DAE are considered or semi-explicit DAE of the form

$$\dot{\mathbf{y}} = f(t, \mathbf{x}, \mathbf{y}), \tag{1}$$

$$0 = g(t, \mathbf{x}, \mathbf{y}) \ . \tag{2}$$

Adaptation of validated Runge-Kutta methods for DAE.
Main ideas

- A priori enclosure of $y(t)$ and $x(t)$
  - Interval Picard operator for y
  - Interval contractor *à la* Newton for x
- Computation of the solution at t_n.

Differential constraint satisfaction problems

Context

Interval analysis

Validated numerical integration

Differential constraint satisfaction problems

Dynamical systems

A general settings of dynamical systems

$$S \equiv \begin{cases} \dot{\mathbf{y}}(t) = \mathbf{f}(t, \mathbf{y}(t), \mathbf{x}(t), \mathbf{p}), \\ \quad 0 = \mathbf{g}(t, \mathbf{y}(t), \mathbf{x}(t)) \\ \quad 0 = \mathbf{h}(\mathbf{y}(t), \mathbf{x}(t)) \end{cases} .$$

we denote by

$$\mathcal{Y}(\mathcal{T}, \mathcal{Y}_0, \mathcal{P}) = \{\mathbf{y}(t; \mathbf{y}_0, \mathbf{p}) : t \in \mathcal{T}, \mathbf{y}_0 \in \mathcal{Y}_0, \mathbf{p} \in \mathcal{P}\} \; .$$

the set of solutions

Example of ODEs with constraints

Production-Destruction systems based on an ODE with parameter $a = 0.3$

$$\begin{pmatrix} \dot{y}_0 \\ \dot{y}_1 \\ \dot{y}_2 \end{pmatrix} = \begin{pmatrix} \dfrac{-y_0 y_1}{1 + y_0} \\ \dfrac{y_0 y_1}{1 + y_0} - a y_1 \\ a y_1 \end{pmatrix}$$

and associated to constraints:

$$\begin{aligned} y_0 + y_1 + y_2 &= 10.0 \\ y_0 &\geqslant 0 \\ y_1 &\geqslant 0 \\ y_2 &\geqslant 0 \end{aligned}$$

Initial values, for $t \in [0, 100]$, are

$$\begin{pmatrix} y_0(0) \\ y_1(0) \\ y_2(0) \end{pmatrix} = \begin{pmatrix} 9.98 \\ 0.01 \\ 0.01 \end{pmatrix}$$

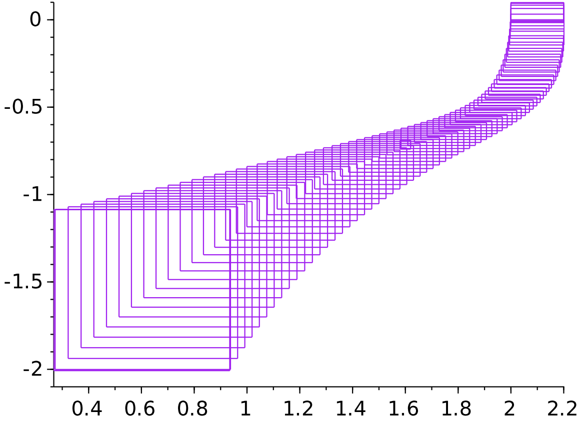
ODEs with constraints in DynIBEX

```
const int n= 3;
Variable y(n);

IntervalVector yinit(n);
yinit[0] = Interval(9.98);
yinit[1] = Interval(0.01);
yinit[2] = Interval(0.01);
Interval a(0.3);

Function ydot = Function(y,Return(-y[0]*y[1]/(1+y[0]),
                                   y[0]*y[1]/(1+y[0]) - a*y[1],
                                   a*y[1]));

NumConstraint csp1(y,y[0]+y[1]+y[2] -10.0 = 0);
NumConstraint csp2(y,y[0]>=0);
NumConstraint csp3(y,y[1]>=0);
NumConstraint csp4(y,y[2]>=0);

Array<NumConstraint> csp(csp1,csp2,csp3,csp4);

ivp_ode problem = ivp_ode(ydot,0.0,yinit,csp);
```

ODEs with constraints in DynIBEX – results

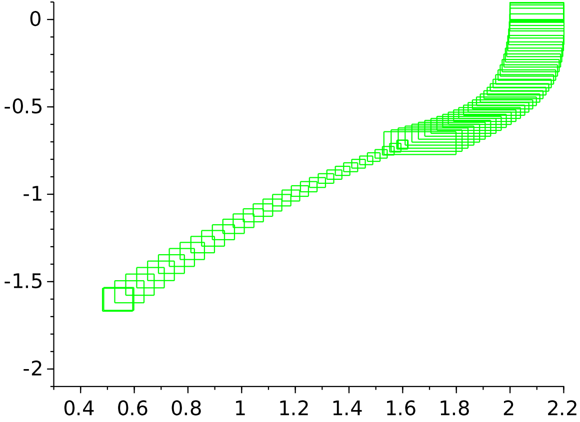
Contractors on trajectories

Add a measure and contract localy



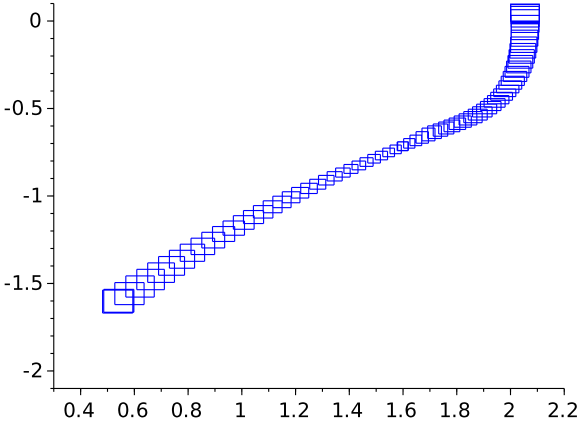
Contractors on trajectories

Forward



Contractors on trajectories

Backward



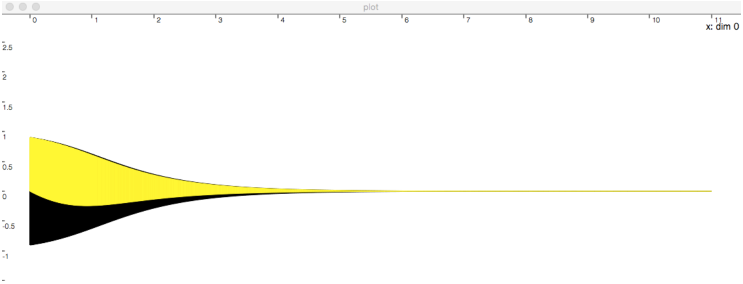
A simple example DynIBEX

ODEs considered

$$\dot{x} = x^3 - 1.0$$

with $x(0) = [-0.9, 0.9]$ and $x(0) = [0, 0.9]$

Simulation result



- Black area is with $x(0) = [-0.9, 0.9]$ (full integration: Picard+Integration)
- Yellow area is with $x(0) = [0, 0.9]$ (contraction+propagation)

Example in DynIBEX

```
const int n = 2;
Variable y(n);

IntervalVector state(n);
state[0] = Interval(0.0);
state[1] = Interval(-0.9, 0.9);

Function ydot(y, Return (Interval(1.0), y[1]*(y[1]*y[1]-1.0)));

ivp_ode vdp = ivp_ode(ydot, 0.0, state);


simulation* simu = new simulation(&vdp, 11.0, LC3, 1e-12, 0.001);

simu -> run_simulation();

simulation* simu1 = new simulation(*simu);

IntervalVector state1(n);
state1[0] = Interval(0.0);
state1[1] = Interval(0.0, 0.9);

simu1 -> propag (state1);
simu1 -> fixed_point (1e-5);
```

Constraint Satisfaction Differential Problems

CSDP

Let S be a differential system and $t_{\text{end}} \in \mathbb{R}_+$ the time limit. A CSDP is a NCSP defined by

- a finite set of variables $\mathcal{V}$ including the parameters of the differential systems S_i, *i.e.*, $(\mathbf{y}_0, \mathbf{p})$, a time variable t and some other algebraic variables $\mathbf{q}$;
- a domain $\mathcal{D}$ made of the domain of parameters $\mathbf{p} : \mathcal{D}_p$, of initial values $\mathbf{y}_0 : \mathcal{D}_{y_0}$, of the time horizon $t : \mathcal{D}_t$, and the domains of algebraic variables $\mathcal{D}_q$;
- a set of constraints $\mathcal{C}$ which may be defined by set-based constraints over variables of $\mathcal{V}$ and special variables $\mathcal{Y}_i(\mathcal{D}_t, \mathcal{D}_{y_0}, \mathcal{D}_p)$ representing the set of the solution of S_i in $\mathcal{S}$.

with set-based constraints considered:

$$\mathbf{g}(\mathcal{A}) \subseteq \mathcal{B} \qquad \mathbf{g}(\mathcal{A}) \supseteq \mathcal{B}$$
$$\mathbf{g}(\mathcal{A}) \cap \mathcal{B} = \emptyset \qquad \mathbf{g}(\mathcal{A}) \cap \mathcal{B} \neq \emptyset$$

Remark translation to intervals should be done with precautions
Note: we follow the same approach that Goldsztejn et al.[3]

[3] Including ODE Based Constraints in the Standard CP Framework, CP10

Particular problems considered and temporal properties

We focus on particular problems of robotics involving quantifiers

- Robust controller synthesis: $\exists \mathbf{u}$, $\forall \mathbf{p}$, $\forall \mathbf{y}_0$ + temporal constraints
- Parameter synthesis: $\exists \mathbf{p}$, $\forall \mathbf{u}$, $\forall \mathbf{y}_0$ + temporal constraints
- etc.

We also defined a set of temporal constraints useful to analyze/design robotic application.

Verbal property	QCSDP translation
Stay in $\mathcal{A}$	$\forall t \in [0, t_{end}],\ [\mathbf{y}](t, \mathbf{v}') \subseteq \mathrm{Int}(\mathcal{A})$
In $\mathcal{A}$ at τ	$\exists t \in [0, t_{end}],\ [\mathbf{y}](t, \mathbf{v}') \subseteq \mathrm{Int}(\mathcal{A})$
Has crossed $\mathcal{A}$*	$\exists t \in [0, t_{end}],\ [\mathbf{y}](t, \mathbf{v}') \cap \mathrm{Hull}(\mathcal{A}) \neq \emptyset$
Go out $\mathcal{A}$	$\exists t \in [0, t_{end}],\ [\mathbf{y}](t, \mathbf{v}') \cap \mathrm{Hull}(\mathcal{A}) = \emptyset$
Has reached $\mathcal{A}$*	$[\mathbf{y}](t_{end}, \mathbf{v}') \cap \mathrm{Hull}(\mathcal{A}) \neq \emptyset$
Finished in $\mathcal{A}$	$[\mathbf{y}](t_{end}, \mathbf{v}') \subseteq \mathrm{Int}(\mathcal{A})$

*: shall be used in negative form

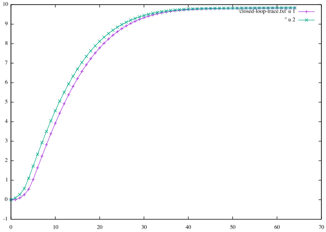
Simulation of a closed-loop system with safety

A simple dynamics of a car with a PI controller

$$\begin{pmatrix} \dot{y} \\ \dot{w} \end{pmatrix} = \begin{pmatrix} \frac{k_p(10.0-y)+k_i w-50.0y-0.4y^2}{m} \\ 10.0 - y \end{pmatrix} \quad \text{with} \quad m \in [990, 1010], k_p = 1440, k_i = 35$$

and **a safety propriety**

$$\forall t, y(t) \in [0, 11]$$



Failure

$$y([0, 0.0066443]) \in [-0.00143723, 0.0966555]$$

Simulation of a closed-loop system with safety property

```
#include "ibex.h"

using namespace ibex;

int main(){
  const int n = 2;
  Variable y(n);

  IntervalVector state(n);
  state[0] = 0.0; state[1] = 0.0;

  // Dynamique d'une voiture avec incertitude sur sa masse + PI
  Function ydot(y, Return ((1440.0 * (10.0 - y[0]) + 35.0 * y[1]  - y[0] * (50.0  + 0.4 * y[0]))
                         / Interval (990, 1010),
                         10.0 - y[0]));
  ivp_ode vdp = ivp_ode(ydot, 0.0, state);

  simulation simu = simulation(&vdp, 10.0, RK4, 1e-6);
  simu.run_simulation();

  // verification de surete
  IntervalVector safe(n);
  safe[0] = Interval(0.0, 11.0);
  bool flag = simu.stayed_in (safe);
  if (!flag) {
    std::cerr << "error safety violation" << std::endl;
  }

  return 0;
}
```

Case study – tuning PI controller [SYNCOP'15]

A cruise control system two formulations:

- uncertain linear dynamics;

$$\dot{v} = \frac{u - bv}{m}$$

- uncertain non-linear dynamics

$$\dot{v} = \frac{u - bv - 0.5\rho CdAv^2}{m}$$

with

- m the mass of the vehicle
- u the control force defined by a PI controller
- bv is the rolling resistance
- $F_{\text{drag}} = 0.5\rho CdAv^2$ is the aerodynamic drag (ρ the air density, CdA the drag coefficient depending of the vehicle area)

Case study – settings and algorithm

Embedding the PI Controller into the differential equations:

- $u = K_p(v_{set} - v) + K_i \int (v_{set} - v)ds$ with v_{set} the desired speed
- Transforming $\text{int}_{\text{err}} = \int (v_{set} - v)ds$ into differential form

$$\frac{\text{int}_{\text{err}}}{dt} = v_{set} - v$$

$$\dot{v} = \frac{K_p(v_{set} - v) + K_i \text{int}_{\text{err}} - bv}{m}$$

Main steps of the algorithm

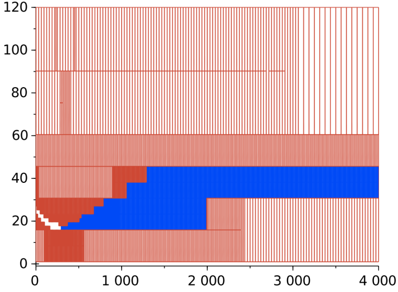
- Pick an interval values for K_p and K_i
- **Simulate** the closed-loop systems with K_p and K_i
  - if `specification is not satisfied`: **bisect** (up to minimal size) intervals and run simulation with smaller intervals
  - if `specification is satisfied` try other values of K_p and K_i

Case study – paving results

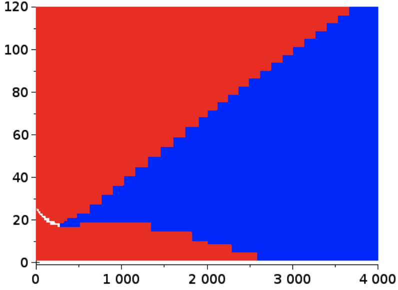
Result of paving for both cases with

- $K_p \in [1, 4000]$ and $K_i \in [1, 120]$
- $v_{\text{set}} = 10$, $t_{\text{end}} = 15$, $\alpha = 2\%$ and $\epsilon = 0.2$ and minimal size=1
- constraints: $y(t_{end}) \in [r - \alpha\%, r + \alpha\%]$ and $\dot{y}(t_{end}) \in [-\epsilon, \epsilon]$

Linear case (CPU $\approx$ 10 minutes)



Non-linear case (CPU $\approx$ 80 minutes)



Towards solving optimal controls

Optimal control of the form

$$\dot{\mathbf{y}}(t) = \mathbf{f}_{\mathbf{u}_2}\left(t, \mathbf{y}(t), \mathbf{u}_1(t)\right) \quad \text{avec} \quad \mathbf{y}(0) = \mathbf{y}_0 \quad \text{and} \quad t \in [0, t_{\text{end}}]$$

$$\mathbf{J}(\mathbf{u}_1(t)) = \psi\left(\mathbf{y}(t_{\text{end}})\right) + \int_0^{t_{\text{end}}} \mathbf{L}(t, \mathbf{y}(t), \mathbf{u}_1(t))$$

can be solved with many different approaches

- **direct method**: full discretization and cast into an optimization problems
- **indirect method**: apply PMP and solve a BVP with shooting methods
- **HJB approaches**: solve a PDE

Remark: we are interested in the indirect approach

Solving BVP ODE

A simple example[4]

$$\ddot{w} = 1.5w^2 \quad \text{with} \quad w(0) = 4 \quad \text{and} \quad w(1) = 1$$

so we have to found the initial condition $\dot{w}(0) = s$ such as the boundary conditions are fulfilled.

Note: There are 2 solutions $s = -8$ and $s \approx -35.9$

A combination of validated numerical integration, contractors and bissection algorithms can do the job.

[4] coming from Stoer, J. and Burlisch, R. Introduction to Numerical Analysis. New York: Springer-Verlag, 1980.

BVP in DynIBEX – 1

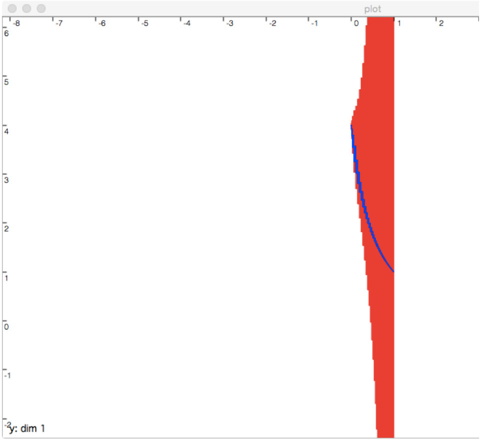
```
const int n = 2;

const double horizon = 1.0;
const double tol = 1e-3;
std::stack<simulation*> stack_sim;

Variable y(n);

IntervalVector initialState(n);
initialState[0] = Interval(-10.0,0.0);
initialState[1] = 4.0;

IntervalVector finalState(n);
finalState[0] = Interval::ALL_REALS;
finalState[1] = 1.0;

Function ydot(y, Return( 1.5 * y[1] * y[1], y[0]));
ivp_ode vdp = ivp_ode(ydot, 0.0, initialState);

simulation simu = simulation(&vdp, horizon, RK4, 1e-6, 0.01);
simu.run_simulation();
plot_simu (&simu, "red[red]");
```

BVP in DynIBEX – 2

```
stack_sim.push (&simu);
while (stack_sim.size() != 0) {
  simulation* s = stack_sim.top();  stack_sim.pop();

  IntervalVector temp = s->get_last();
  if (temp.is_subset(finalState) && temp.max_diam() <= tol) {
    plot_simu (s, "blue[blue]");
  }
  else if ((temp & finalState).is_empty()) {
    std::cerr << "do nothing : FORGET s with init = " << s->get(0) << std::endl;
    free(s);
  }
  else {
    IntervalVector init = s->get(0);
    LargestFirst bbb(tol, 0.5);
    if (init.max_diam() >= tol) {
      std::pair<IntervalVector,IntervalVector> p = bbb.bisect(init);
      simulation* s1 = new simulation(*s);  s1 -> propag (p.first);

      simulation* s2 = new simulation(*s);  s2 -> propag (p.second);

      stack_sim.push(s1);  stack_sim.push(s2);
    }
    else {
      std::cerr << "UNKNOWN case : with initial condition " << init << std::endl;
    }
  }
```

BVP results

A huge over-approximation of the trajectory is computed (red) and then bissection and contractors are used to enclose the solution



One over-approximated solution is $s = [-8.00049, -7.99988]$

What is missing to solve optimal control problems in DynIBEX ?

Example: minimal time problem[5]

$$\dot{\mathbf{x}} = A\mathbf{x} + Bu(t) \quad \text{with} \quad A = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} \quad \text{and} \quad B = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

$u(t)$ is scalar, $|u| \leqslant 1$, and we try to reach $\mathbf{0}$ from $\mathbf{x}_0$ as fast as possible.
In this case, $u(t) = \text{sign}(p(t)B(t))$

- where $p(t)B(t)$ is the commutation function
- $p(t)$ is solution of $\dot{p}(t) = -p(t)A(t)$

Problem: control function are not continuous and it is an issue for validated numerical integration methods

Consequence we need to deal with hybrid systems (here 2 modes: $u = 1$ and $u = -1$)

[5] coming from E. Trela lecture notes on Optimal Control

Conclusion

DynIBEX is one **ingredient** of verification tools for cyber-physical systems.
It can **handle uncertainties**, can **reason on sets of trajectories**.

Also applied on

- Controller synthesis of sampled switched systems [SNR'16]
- Parameter tuning in the design of mobile robots [MORSE'16]
- RRT-based trajectory generation [CDC17]

Future work (a piece of)

- Pursue and improve cooperation with IBEX language
- Improve algorithm of validated numerical integration (*e.g.*, sensitivity)
- Simulation of hybrid systems
- SMT modulo ODE