Systèmes dynamiques non lisses. Feuille 2

V. Acary.
vincent.acary@inria.fr

2020–2021

Objectifs

L'objectif de ce TD est d'utiliser siconos `http://siconos.gforge.inria.fr` afin de simuler des systèmes dynamiques non réguliers issus de l'électronique et de la mécanique.

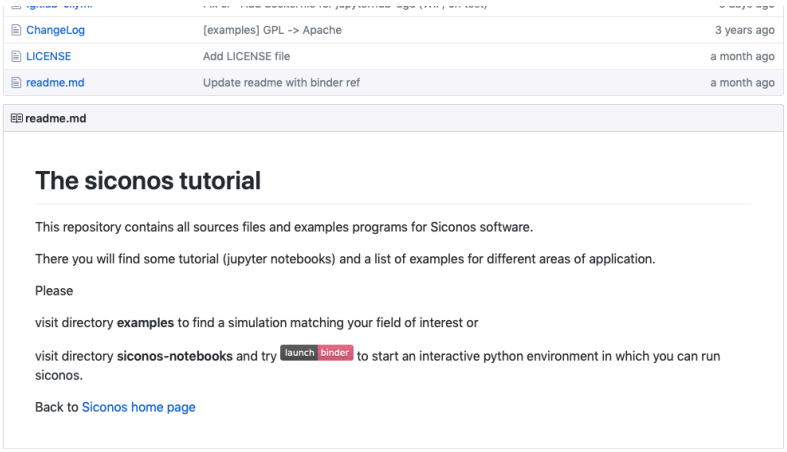
On s'intéressera en premier a la régularité des solutions obtenues et à leur stabilité. Dans un deuxième temps, on regardera un example où l'état du système n'est pas continue comme la boule rebondissante.

Procédure

Utilisation du binder.org

Si votre connexion a l'uga ne fonctionne pas, vous pouvez utiliser le site binder.org.

1. Aller à l'adresse suivante : `https://github.com/siconos/siconos-tutorials`
2. En bas de la page, clicker sur le logo "Launch binder »



3. Le chargement du serveur jupyter peut prendre quelques minutes. soyez patient
4. Selectionner le premier tutoriel `A 4 diodes bridge wave rectifier`

Pensez à sauvergarder régulièrement votre fichier en local (download/upload) car le serveur binder ne fait pas de sauvegarde!!!

Option2 : Utilisation du jupyterhub de l'UGA

1. Connecter vous au serveur `https://jupyterhub.u-ga.fr/` à travers un navigateur web en utilisant votre login Agalan.
2. Démarrer un processus « Terminal »
3. Dans la console du terminal :
   — `cd notebooks`

— `git clone https://github.com/siconos/siconos-tutorials.git`

4. Revenir au navigateur de fichiers jupyter pour ouvrir le premier tutoriel
   `st01_diode_bridge.ipynb`
   situé dans
   `~/notebooks/siconos-tutorials/siconos-notebooks`

Siconos tutorial : A 4 diodes bridge wave rectifier.

Prerequisites

Jupyter notebooks reminder

A notebook is a sequence of "cells" that can be executed.

Each cell can handle either python code or markdown for comments.

- Edit a cell : Enter
- Execute a cell: Shift + Enter
- Run all cells : kernel menu (top of the page) --> Run all
- Delete cell : DD
- Add cell : Ctrl-mb
- Shortcuts reminder : Ctrl-m h
- List all magic commands : %lsmagic

More : https://jupyter-notebook-beginner-guide.readthedocs.io/en/latest/what_is_jupyter.html#references (https://jupyter-notebook-beginner-guide.readthedocs.io/en/latest/what_is_jupyter.html#references)

Warning : cells can be executed in any order but results and variables are persistent (until a call to %reset or kernel->restart)

Import siconos components

You may need to set PYTHONPATH or sys.path if siconos python packages are not in standard places.

In [ ]:
```
import sys
print(sys.path)
# Update the following line depending on your siconos python instal
lation.
sys.path.append("/usr/local/lib/python3.5/site-packages/")
sys.path.append("~/Library/Python/3.5/lib/python3.5/site-packages/"
)
import siconos.numerics as sn
import siconos.kernel as sk
import numpy as np
```


I - Modeling : NonSmooth Dynamical Systems (NSDS) definition

--> Dynamical systems, constraints, nonsmooth laws ...

The dynamical system

Consider the following example, a 4-diodes bridge wave rectifier

Using the Kirchhoff current and voltage laws and branch constitutive equations, the dynamics of the system writes

$$\begin{bmatrix} \dot{v_C} \\ \dot{i}_L \end{bmatrix} = \begin{bmatrix} 0 & \frac{-1}{C} \\ \frac{1}{L} & 0 \end{bmatrix} . \begin{bmatrix} v_C \\ i_L \end{bmatrix} + \begin{bmatrix} 0 & 0 & \frac{-1}{C} & \frac{1}{C} \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} -v_{DR1} \\ -v_{DF2} \\ i_{DF1} \\ i_{DR2} \end{bmatrix}$$

and if we denote

$$x = \begin{bmatrix} \dot{v_C} \\ \dot{i}_L \end{bmatrix}, \lambda = \begin{bmatrix} -v_{DR1} \\ -v_{DF2} \\ i_{DF1} \\ i_{DR2} \end{bmatrix}, \quad A = \begin{bmatrix} 0 & \frac{-1}{C} \\ \frac{1}{L} & 0 \end{bmatrix}, \quad r = \begin{bmatrix} 0 & 0 & \frac{-1}{C} & \frac{1}{C} \\ 0 & 0 & 0 & 0 \end{bmatrix} \lambda$$

we get a first order linear system

$$\dot{x} = A.x + r$$

Such systems are defined in Siconos with FirstOrderLinearDS, in a very simple way:

In [ ]:
```
# dynamical system parameters
Lvalue = 1e-2    # inductance
Cvalue = 1e-6    # capacitance
Rvalue = 1e3     # resistance
Vinit = 10.0     # initial voltage
x0 = [Vinit, 0.] # initial state
# A matrix of the linear oscillator
A = np.zeros((2, 2), dtype=np.float64)
A.flat[...] = [0., -1.0/Cvalue, 1.0/Lvalue, 0.]

# build the dynamical system
ds = sk.FirstOrderLinearTIDS(x0, A)
```

To get more details on this (or any other) class of DS, try:

In [ ]:
```
help(sk.FirstOrderLinearDS)
```


Nonsmooth laws and constraints

Now, the nonsmooth part of the system must be defined, namely what are the nonsmooth laws and constraints between the variables. In Siconos, the definition of a nonsmooth law and a relation between one or two dynamical systems is called an Interaction (see Interactions between dynamical systems). Thus, the definition of a set of dynamical systems and of interactions between them will lead to the complete nonsmooth dynamical system.

For the oscillator of fig 1: Diode bridge, there exist some linear relations (constraints) between voltage and current inside the diode, given by

$$\begin{bmatrix} i_{DR1} \\ i_{DF2} \\ -v_{DF1} \\ -v_{DR2} \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ -1 & 0 \\ 1 & 0 \end{bmatrix} . \begin{bmatrix} v_C \\ i_L \end{bmatrix} + \begin{bmatrix} \frac{1}{R} & \frac{1}{R} & -1 & 0 \\ \frac{1}{R} & \frac{1}{R} & 0 & -1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} . \begin{bmatrix} -v_{DR1} \\ -v_{DF2} \\ i_{DF1} \\ i_{DR2} \end{bmatrix}$$

with

$$y = \begin{bmatrix} i_{DR1} \\ i_{DF2} \\ -v_{DF1} \\ -v_{DR2} \end{bmatrix}, D = \begin{bmatrix} \frac{1}{R} & \frac{1}{R} & -1 & 0 \\ \frac{1}{R} & \frac{1}{R} & 0 & -1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix},$$

and recalling that

$$\lambda = \begin{bmatrix} -v_{DR1} \\ -v_{DF2} \\ i_{DF1} \\ i_{DR2} \end{bmatrix}, \quad B = \begin{bmatrix} 0 & 0 & \frac{-1}{C} & \frac{1}{C} \\ 0 & 0 & 0 & 0 \end{bmatrix}, \quad r = B\lambda$$

which is indeed a linear relation between (x, r) and (y, λ):

$$\begin{cases} y = Cx + D\lambda, \\ r = B\lambda \end{cases}$$

implemented in siconos as:

In [ ]:
```
# B, C, D matrices of the relation
C = [[0.,   0.],
    [0,    0.],
    [-1.,  0.],
    [1.,   0.]]

D = [[1./Rvalue, 1./Rvalue, -1.,  0.],
    [1./Rvalue, 1./Rvalue,  0., -1.],
    [1.,        0.,         0.,  0.],
    [0.,        1.,         0.,  0.]]

B = [[0.,        0., -1./Cvalue, 1./Cvalue],
    [0.,        0.,  0.,        0.       ]]

# set relation type
relation= sk.FirstOrderLinearTIR(C, B)
relation.setDPtr(D)
```

Each diode of the bridge is supposed to be ideal, with the behavior shown on left-hand sketch of the figure below

Such a behavior can be described with a **complementarity condition** between current and reverse voltage.

Complementarity between two variables $y \in R^m, \lambda \in R^m$ reads as

$$\text{if } \lambda = 0 \text{ then } y \geq 0 \text{ and if } \lambda > 0 \text{ then } y = 0$$

or, using $\perp$ symbol,

$$0 \leq y \perp \lambda \geq 0$$

which means that $y^T \lambda = 0$. The inequalities must be considered component-wise.

Then, back to our circuit, the complementarity conditions, coming from the ideal diodes characteristics, are given by:

$$\begin{array}{l} 0 \leq -v_{DR1} \perp i_{DR1} \geq 0 \\ 0 \leq -v_{DF2} \perp i_{DF2} \geq 0 \\ 0 \leq i_{DF1} \perp -v_{DF1} \geq 0 \\ 0 \leq i_{DR2} \perp -v_{DR2} \geq 0 \end{array} \quad or \quad 0 \leq y \perp \lambda \geq 0$$

Note that depending on the diode position in the bridge, y_i stands for the reverse voltage across the diode or for the diode current.

To represent such a nonsmooth law Siconos has a class ComplementarityConditionNSL (you will find NSL in each class-name defining a nonsmooth law):

In [ ]:
```
interaction_size = 4  # number of constraints
nonsmooth_law = sk.ComplementarityConditionNSL(interaction_size)
```

A nonsmooth law and a relation define something called **Interaction** in Siconos

In [ ]:
```
interaction = sk.Interaction(nonsmooth_law, relation)
```

Notice that this interaction just describes some relations and laws but is not connected to any real dynamical system, for the moment.

The modeling part is almost complete, since only one dynamical system and one interaction are needed to describe the problem. They must be gathered into a specific object, the **Model**. A model contains a nonsmooth dynamical system and the description of its simulation. The building of this object is quite simple: just set the time window for the simulation, include dynamical systems and link them to the correct interactions.

In [ ]:
```
# dynamical systems and interactions must be gathered into a model
t0 = 0. # initial time
T = 5.0e-3 # duration of the simulation
DiodeBridge = sk.Model(t0, T)
# add the dynamical system in the nonsmooth dynamical system of the
model
DiodeBridge.nonSmoothDynamicalSystem().insertDynamicalSystem(ds)
# link the interaction and the dynamical system
DiodeBridge.nonSmoothDynamicalSystem().link(interaction, ds)
```


II - Simulation definition

It's time to describe how our nonsmooth dynamical system will be discretized, formulated and solved. We need first to define how the nonsmooth dynamical system will be integrated over time. This is the role of the **Simulation**, which must define:

- how dynamical systems are discretized and integrated over a time step
- how the resulting One-Step NonSmooth Problem (OSNSP) will be formalized and solved

Two different strategies are available : event-capturing time-stepping schemes (a.k.a time stepping schemes) and event-detecting time--stepping schemes (a.k.a event-driven schemes).

For the Diode Bridge example, an event-capturing strategy will be used, with an Euler-Moreau integrator and a LCP (Linear Complementarity Problem) formulation for the OSNSP.

Let us start with the 'one-step integrator', i.e. the description of the discretisation and integration of the dynamics over a time step, between time t_k and t_{k+1}. The integration of the equation over the time step is based on a θ-method, leads to:

$$\begin{array}{rcl} x_{k+1} & = & x_k^{free} + hW^{-1} r_{k+1} \\ W & = & (I - h\theta A) \\ x_k^{free} & = & x_k + hW^{-1}(Ax_k + b) \end{array}$$

implemented as:

In [ ]:
```
theta = 0.5
osi = sk.EulerMoreauOSI(theta)
```


Based on the simulation strategy and the time-integration, a one-step nonsmooth problem will be formalized in Siconos.

Considering the following discretization of the previously defined relations and nonsmooth law

$$\begin{array}{rcl} y_{k+1} & = & Cx_{k+1} + D(t_{i+1})\lambda_{k+1} \\ R_{k+1} & = & B\lambda_{k+1} \\ 0 \leq y_{k+1} & \perp & \lambda_{k+1} \geq 0 \end{array}$$

we get

$$\begin{array}{rcl} y_{k+1} & = & q + M\lambda_{k+1} \\ 0 \leq y_{k+1} & \perp & \lambda_{k+1} \geq 0 \end{array}$$

with $q = Cx_{k+1}^{free}, \quad M = hCW^{-1}B + D$

This is known as a Linear Complementarity Problem, written in siconos thanks **LCP** class.

As usual, check user documentation for a complete review of the nonsmooth problems formulations available in Siconos.

In [ ]:
```
osnspb = sk.LCP()
```

Depending on the chosen formulation, different solvers are available. You can for example change the default (Lemke) for a non-symmetric QP, as below. A complete list of available solvers can be found in documentation (LCP solvers: http://siconos.gforge.inria.fr/users_guide/lcp_solvers.html#lcp-solvers (http://siconos.gforge.inria.fr/users_guide/lcp_solvers.html#lcp-solvers)).

In [ ]:
```
osnspb = sk.LCP(sn.SICONOS_LCP_NSQP)
```

Then the last step consists in the simulation creation, with its time discretisation

In [ ]:
```
# simulation and time discretisation
time_step =  1.0e-6
td = sk.TimeDiscretisation(t0, time_step)
simu = sk.TimeStepping(td, osi, osnspb)
```

The connection with the nonsmooth dynamical system is done through the **Model**

In [ ]:
```
DiodeBridge.setSimulation(simu)
DiodeBridge.initialize()
```

The model is now complete and ready to run

III - Running the simulation

There are several options to run the simulation. The most simple is as follows. In this version, the events are the instants of the time discretization but other events of differents may be scheduled.

In [ ]:
```
#while simu.hasNextEvent():
#    simu.computeOneStep() # Solve the LCP
#    simu.nextStep() # Save current vars and prepare next step
```

For the present case, x,y and λ at each time step are needed for postprocessing. Here is an example on how to get and save them in a numpy array

In [ ]:
```
N = int((T - t0) / simu.timeStep()) + 1
data_plot = np.zeros((N, 8))
y = interaction.y(0)
lamb = interaction.lambda_(0)
x = ds.x()
k = 0
data_plot[k, 1] = x[0] #  inductor voltage
data_plot[k, 2] = x[1] # inductor current
data_plot[k, 3] = y[0] # diode R1 current
data_plot[k, 4] = -lamb[0] # diode R1 voltage
data_plot[k, 5] = -lamb[1] # diode F2 voltage
data_plot[k, 6] = lamb[2] # diode F1 current
data_plot[k, 7] = y[0] + lamb[2] # resistor current
k += 1

while simu.hasNextEvent():
    simu.computeOneStep() # Solve the LCP
    data_plot[k, 0] = simu.nextTime()
    data_plot[k, 1] = x[0]
    data_plot[k, 2] = x[1]
    data_plot[k, 3] = y[0]
    data_plot[k, 4] = - lamb[0]
    data_plot[k, 5] = - lamb[1]
    data_plot[k, 6] = lamb[2]
    data_plot[k, 7] = y[0] + lamb[2]
    k += 1
    simu.nextStep() # Save current vars and prepare next step
```


- *hasNextEvent()* is true as long as there are events to be considered, i.e. until T is reached
- *nextStep()* is mainly used to increment the time step, save current state and prepare initial values for next step.
- *computeOneStep()* performs computation over the current time step. In the Moreau's time stepping case, it will first integrate the dynamics to obtain the so-called free-state, that is without non-smooth effects, then it will formalize and solve a LCP before re-integrate the dynamics using the LCP results.

The results can now be postprocessed, using matplotlib pyplot for example

IV - Post-processing

In [ ]:
```
import matplotlib.pyplot as plt
%matplotlib inline
plt.figure(figsize=(15,10))
plt.subplot(411)
plt.title('inductor voltage')
plt.plot(data_plot[1:k - 1, 0], data_plot[1:k - 1, 1])
plt.grid()
plt.subplot(412)
plt.title('inductor current')
plt.plot(data_plot[1:k - 1, 0], data_plot[1:k - 1, 2])
plt.grid()
plt.subplot(413)
plt.title('diode R1 (blue) and F2 (green) voltage')
plt.plot(data_plot[1:k - 1, 0], -data_plot[1:k - 1, 4])
plt.plot(data_plot[1:k - 1, 0], data_plot[1:k - 1, 5])
plt.grid()
plt.subplot(414)
plt.title('resistor current')
plt.plot(data_plot[1:k - 1, 0], data_plot[1:k - 1, 7])
plt.grid()
```


V - Questions

Let us consider the following function $V : \mathbb{R}^2 \to \mathbb{R}$:

$$V(x) = \frac{1}{2} x^T P x$$

with $P = \begin{bmatrix} C & 0 \\ 0 & L \end{bmatrix}$

1. Plot the phase portrait of the system
2. Compute the equilibria of the system
3. Plot the contour (level set) of the function V
4. Show that the system is passive.
5. Is the equilibrium is stable in the sense of Lyapunov ?

In [ ]:
```
```