Python for control purposes

Prof. Roberto Bucher
Scuola Universitaria Professionale della Svizzera Italiana
Dipartimento Tecnologie Innovative
6928 Manno
roberto.bucher@supsi.ch

March 23, 2015

Abstract

Computer Aided Control System Design (CACSD) environments play a key role in teaching Control Systems in our university. Since the late '90, Matlab and Simulink have been chosen as the reference CACSD for control system design. Lectures and laboratories take advantage of all the features in design and simulation offered by these products. On the other side, few companies are able to purchase the expensive licenses related to the different toolboxes required for a complete design environment. In addition, universities have to buy commercial licenses of Mathworks products for applications in applied research. There are free alternatives downloadable on the web, but nothing has the same quality as the cited products. One of the most promising tools is Python. This document will present a new approach to control systems based on Python and a set of additional functions developed at SUPSI for controlling real applications, in order to apply rapid prototyping methods in control design. The starting point is represented by the Python Control System toolbox, inititially developed at Caltech by Richard Murray. Most of the commands implemented in the Python Control toolbox are related to the book from Richard Murray and Carl Åström "Feedback Systems" [1].
In the first part of this "Learnmodule" we'll present some control activities that can be performed using the Python packages and the Python Control toolbox. The presented methods can be executed under Windows, Mac and Linux. In the second part we'll go deeper in hybrid simulation, code generation and real-time execution: at present this part runs only under the Linux OS and profits in articular of the RT improvements given by the PREEMPT RT patch.
A VirtualBox disk image [2] with a Debian ditribution and all the required packages is available.
Many thanks to Dr. Silvano Balemi for the help in reviewing this document.

1 Introduction

1.1 Install the packages

1.1.1 The simplest way

I prepared a VirtualBox disk image [2] with a Debian distribution and all the required packages. VirtualBox is available for Windows, Linux, OS X and Solaris. All the features described in this document are available.
Please contact me via email to receive the link to the zipped file.

1.1.2 Linux

The required modules can be simply installed using the usual package manager of the Linux distribution. It is also possible to install the Anaconda distribution [3] for Linux to get the basic Python

modules.
It is important to check the versions of the Python modules, in particular numpy, scipy and sympy. Old versions of these packages don't allows to perform all the tasks described in this document. In case of old versions, it is possible to download the last versions of these packages from the SciPy download page [4], and install them from a Linux shell.
Under Debian jessie we can use the apt manager to install the following packages:

- python-numpy (Vers. $\geq$ 1.8.2)
- python-scipy (Vers. $\geq$ 0.14)
- python-matplotlib
- python-sympy (Vers. $\geq$ 0.7.5)
- python-setuptools
- python-psutils
- ipython
- ipython-qtconsole

Under Debian and Ubuntu it is possible to check if all the required development packages are correctly installed using the shell command

```
sudo apt-get build-dep python-scipy
```

The following packages are not available as distribution packages and should be installed separately.

- The Python Control toolbox [5]
- The Slycot libraries [6]
- The SupsiCtrl package [7]

For the second part of the project (code generation etc.) the following packages are required

- python-pyqt4
- python-pyqt4-dev
- python-qwt5-qt4
- geda
- geda-gattrib
- geda-gnetlist
- geda-gschem
- geda-gsymcheck
- geda-symbols
- geda-utils

This features presented in the second part of this document are at present only available for the Linux OS.

1.1.3 Windows

Under Windows it is sufficient to install the "Anaconda" package [3], to have all the python and ipython modules installed. The Slycot libraries for Windows can be downloaded from here [8].
At present it is not possible to perform hybrid simulation and code generation under the Windows OS.

1.1.4 Mac OSX

The Anaconda package [3] is available for Mac OSX. The Slycot libraries can be downloaded from here [8].

2 Python - Some hints for Matlab users

2.1 Basics

There are important differences between Matlab and Python. In particular, the Python approach to matrices and to indexed objects is quite different compared to Matlab.
More information about a comparison between Python and Matlab is available online at [9].
The web contains a lot of documentation about Python and its packages. In particular, the book of David Pine [10] gives a good introduction about the features of Python for scientific applications.
Other links present tutorials for **numpy** [11], **scipy** [12], **matplotlib** [13] and **sympy** [14].

2.2 The python shell

A Python script can run within a Python shell, but can also be launched as executable.
The basic python shell is similar to the Matlab shell without the java improvements (**matlab -nojvm**).
A better shell is for example **ipython**. In this interactive form, when started with the extension –**pylab**, ipython already loads at startup a set of functions and modules.
Another interesting environment, more similar to the Matlab shell, is represented by the **Spyder** application. In this application it is possible to debug scripts and functions like in the Matlab environment.
In this document we are mostly working with **ipython** launched with the shell commands

```
ipython qtconsole --pylab inline
```

or

```
ipython --pylab
```

Sometimes not all the functions and modules are explicitly loaded at the beginning of the examples. In addition, **ipython** implements some useful commands like for example **whos** and **run** (for launching scripts).
In the ipython shell it is possible to start single commands, paste a set of commands or launch a ".py" program using **run**.

```
In [1]: # single command

In [2]: a = 5

In [3]: # paste a set of commands

In [4]: a=5
   ...: b=7
   ...: c=a*b
   ...: print c
   ...:
35

In [5]: # run a .py file

In [6]: run DCmotorKane.py
Matrix([[-Dm*w(t) + kt*I(t)]])
Matrix([[-J*Derivative(w(t), t)]])
[[0  1]
 [0 -Dm/J]]
[[0]
 [kt/J]]
```

2.3 Python vs. Matlab

Differently from Matlab, Python implements more types of variables

```
In [1]: a=5

In [2]: b=2.7

In [3]: c=[[1,2,3],[4,5,6]]

In [4]: d='Ciao'

In [5]: whos
Variable   Type      Data/Info
-----------------------------
a          int       5
b          float     2.7
c          list      n=2
d          str       Ciao
```

2.4 List, array and matrix

Python implements three kind of multidimensional objects: **list**, **array** and **matrix**. These objects are handled differently than in Matlab.

2.4.1 List

A Python **list** implements the Matlab **cell**. It represents the simplest and default indexed object.

```
In [1]: a=[[[1,2],[3,4]],'abcd',2]

In [2]: b=[[1,2,3],[4,5,6],[7,8,9]]

In [3]: whos
Variable   Type      Data/Info
------------------------------
a          list      n=3
b          list      n=3
```

2.4.2 Arrays

In Python the **array** is a multidimensional variable that implements sets of values of the same type. Usually the elements of an array are numbers, but can also be booleans, strings, or other objects. An array is the basic instance for most scientific applications.
Operations like *****, **/**, ****** etc. implement the **dot** operations of the Matlab environment (**.***, **./** and **.^**). For example, the multiplication of two arrays $a * a$ represents the value-by-value multiplication implemented in Matlab with the operation $a. * a$.

```
In [1]: from numpy import mat, matrix, array

In [2]: a=array([[1,2,3],[4,5,6]])

In [3]: b=array([[1],[2]])

In [4]: print a*a
[[ 1  4  9]
 [16 25 36]]

In [5]: print a*b
[[ 1  2  3]
 [ 8 10 12]]
```

2.4.3 Matrices

The **matrix** object is useful in case of linear algebra operations. In this case the variables are instanced using the **mat** or the **matrix** function.

```
In [1]: from numpy import mat, matrix, array

In [2]: a=mat(a)

In [3]: b=array([[1],[2],[3]])

In [4]: a*b
Out[5]:
matrix([[14],
        [32]])

In [6]: a=array(a)
```

```
In [7]: a*b
---------------------------------------------------------------
ValueError                                Traceback (most recent
    call last)
<ipython-input-9-8201c27d19b7> in <module>()
----> 1 a*b

ValueError: operands could not be broadcast together with
    shapes (2,3) (3,1)

In [8]: b=mat(b)

In [9]: a*b
Out[10]:
matrix([[14],
        [32]])
```

2.5 Indexing

Indexing in Python is quite different compared with the syntax used in Matlab. Indices start from **0** (and not **1** as in Matlab). In addition, the syntax is different for lists, arrays and matrices.

2.5.1 Lists

1-dimension lists can be accessed using one index (ex. $a[2]$). Multidimensional lists require multiple indices in the form $[i][j]\ldots$

```
In [1]: a=[1,2,3,4,5]

In [2]: %whos
Variable   Type      Data/Info
-------------------------------
a          list      n=5

In [3]: a[3]
Out[3]: 4

In [4]: b=[[1,2,3],[4,5,6]]

In [5]: %whos
Variable   Type      Data/Info
-------------------------------
a          list      n=5
b          list      n=2

In [6]: b[1][2]
Out[6]: 6

In [7]: b[0]
Out[7]: [1, 2, 3]
```

2.5.2 Arrays

Multidimensional arrays allow the use of indices in the forms $[i, j]$ and $[i][j]$.

```
In [1]: from numpy import array

In [2]: a=array([1,2,3,4,5])

In [3]: b=array([[1,2,3],[4,5,6]])

In [4]: %whos
Variable   Type        Data/Info
-------------------------------
a          ndarray     5: 5 elems, type 'int64', 40 bytes
b          ndarray     2x3: 6 elems, type 'int64', 48 bytes

In [5]: a.shape
Out[5]: (5,)

In [6]: b.shape
Out[6]: (2, 3)

In [7]: a[3]
Out[7]: 4

In [8]: b[0,2]
Out[8]: 3

In [9]: b[0][2]
Out[9]: 3

In [10]: b[:,0]
Out[10]: array([1, 4])

In [11]: b[0,:]
Out[11]: array([1, 2, 3])

In [12]: b[0]
Out[12]: array([1, 2, 3])
```

2.5.3 Matrices

Matrices can be only indexed using the $[i, j]$ syntax. A matrix has always a minimum of 2 dimensions.

```
In [1]: from numpy import mat

In [2]: a=array([1,2,3,4,5])

In [3]: b=array([[1,2,3],[4,5,6]])

In [4]: %whos
Variable   Type        Data/Info
-------------------------------
a          matrix      [[1 2 3 4 5]]
b          matrix      [[1 2 3]\n [4 5 6]]

In [5]: a.shape
Out[5]: (1, 5)

In [6]: b.shape
Out[6]: (2, 3)
```

```
In [7]: a[0,2]
Out[7]: 3

In [8]: b[1,1]
Out[8]: 5

In [9]: b[:,0]
Out[9]:
matrix([[1],
        [4]])

In [10]: b[0,:]
Out[10]: matrix([[1, 2, 3]])
```

2.5.4 Multidimensional arrays and matrices

Matrices and arrays can be defined with more than 2 dimensions.

```
In [1]: from numpy import array, mat

In [2]: a=zeros((3,3,3),int8)

In [3]: a.shape
Out[3]: (3, 3, 3)

In [4]: %whos
Variable   Type        Data/Info
-----------------------------------
a          ndarray     3x3x3: 27 elems, type 'int8', 27
    bytes

In [5]: a[1,1,1]
Out[5]: 0
In [6]: a[1,1,1]=5

In [7]: a
Out[7]:
array([[[0, 0, 0],
        [0, 0, 0],
        [0, 0, 0]],

       [[0, 0, 0],
        [0, 5, 0],
        [0, 0, 0]],

       [[0, 0, 0],
        [0, 0, 0],
        [0, 0, 0]]], dtype=int8)
```

3 The Python Control System toolbox

3.1 Basics

The Python Control Systems Library, is a package initially developed by Richard Murray at Caltech. This toolbox contains a set of python classes and functions that implement common operations for the analysis and design of feedback control systems. In addition, a MATLAB compatibility package (control.matlab) has been integrated in order to provide functions equivalent to the commands available in the MATLAB Control Systems Toolbox.

3.2 Models

LTI systems can be described in state-space form or as transfer functions.

3.3 Continuous systems

3.3.1 State-space representation

```
In [1]: from control import *

In [2]: a=[[0,1],[-1,-1]]

In [3]: b=[[0],[1]]

In [4]: c=[1,0]

In [5]: d=0

In [6]: sys = ss(a,b,c,d)

In [7]: print sys
A = [[ 0  1]
 [-1 -1]]

B = [[0]
 [1]]

C = [[1  0]]

D = [[0]]
```

3.3.2 Transfer function

```
 In [1]: from control import *

In [2]: g=tf(1,[1,1,1])

In [3]: print g

     1
-------------
s^2 + s + 1
```

3.3.3 Zeros-Poles-Gain

This method is not implemented in control toolbox yet. It is available in the package **scipy.signal** but it is not completely compatible with the class of LTI objects defined in the Python control toolbox.

3.4 Discrete time systems

An additional fields (**dt**) in the **StateSpace** and **TransferFunction** classes is used to differentiate continuous-time and discrete-time systems.

3.4.1 State-space representation

```
In [4]: a=[[0,1],[-1,1]]

In [5]: b=[[0],[1]]

In [6]: c=[1,-1]

In [7]: d=0

In [8]: sysd = ss(a,b,c,d,0.01)

In [9]: print sysd
A = [[ 0  1]
 [-1  1]]

B = [[0]
 [1]]

C = [[ 1 -1]]

D = [[0]]

dt = 0.01
```

3.4.2 Transfer function

```
In [1]: from control import *

In [2]: g=tf([1,-1],[1,-1,1],0.01)

In [3]: print g

   z - 1
-----------
z^2 - z + 1

dt = 0.01
```

3.4.3 Conversions

The Python control system toolbox only implements conversion from continuous time systems to discrete-time systems (**c2d**) with the methods “zoh”, “tustin” and “matched”. No conversion from discrete to continuous has been implemented yet.

The supsictrl.yottalab package implements both functions **c2d** and **d2c** with the methods “zoh’, ”foh“, ”tustin“ and ”matched“ (”matched“ is only implemented in c2d).

```
In [1]: from control import *

In [2]: from control.Matlab import *

In [3]: g=tf(1,[1,1,1])

In [4]  # Matlab compatibility

In [5]: gd = c2d(g,0.01)

In [6]  # control toolbox

In [7]: gd2 = sample_system(g,0.01)

In [8]: print g

     1
-------------
s^2 + s + 1


In [9]: print gd

4.983e-05 z + 4.967e-05
-----------------------
  z^2 - 1.99 z + 0.99

dt = 0.01
```

```
In [1]: from control import *

In [2]: from control.Matlab import c2d

In [3]: from supsictrl.yottalab import d2c

In [4]: g=tf(1,[1,1,1])

In [5]: gd =c2d(g,0.01)

In [6]: g2=d2c(gd)

In [7]: print g

     1
-------------
s^2 + s + 1


In [8]: print g2

1.729e-14 s + 1
---------------
  s^2 + s + 1
```

3.5 Casting

The control.matlab module implements the casting functions to transform LTI systems to a transfer function (**tf**) or to a state-space form (**ss**).

```
In [8]: g = tf(sys)

In [9]: print g

      1
-------------
s^2 + s + 1
```

and transfer functions into one of the state-space representation

```
In [10]: sys = ss(g)

In [11]: print sys
A = [[ 0. -1.]
 [ 1. -1.]]

B = [[-1.]
 [ 0.]]

C = [[ 0. -1.]]

D = [[ 0.]]
```

3.6 Models interconnection

Commands like **parallel** and **series** are available in order to interconnect systems. The operators + and * have been overloaded for the LTI class to perform the same operations. In addition the command **feedback** is implemented exactly as in Matlab.

```
In [1]: from control import *

In [2]: g1=tf(1,[1,1])

In [3]: g2=tf(1,[1,2])

In [4]: print parallel(g1,g2)

   2 s + 3
-------------
s^2 + 3 s + 2


In [5]: print g1+g2

   2 s + 3
-------------
s^2 + 3 s + 2
```

```
In [6]: print series(g1,g2)

       1
-----------------
s^2 + 3 s + 2


In [7]: print g1*g2

       1
-----------------
s^2 + 3 s + 2
```

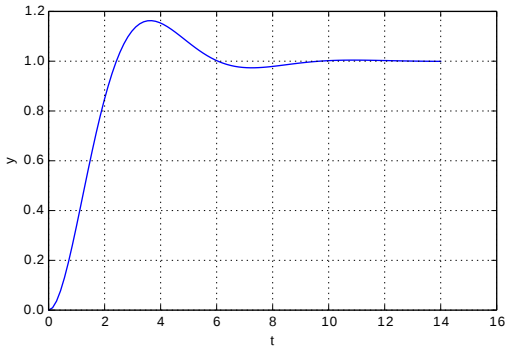
```
In [8]: print feedback(g1,g2)

     s + 2
-----------------
s^2 + 3 s + 3
```

4 System analysis

4.1 Time response

The Python Control toolbox offers own functions to simulate the time response of systems. For Matlab users, the control.matlab module gives the possibility to work with the same syntax as in Matlab. Please take care about the order of the return values!
Examples of time responses are shown in the figures 1, 2, 3, 4 and 5.

```
In [1]: from control import *

In [2]: import matplotlib.pyplot as plt

In [3]: g = tf(1,[1,1,1])

In [4]: t,y = step_response(g)

In [5]: plt.plot(t,y)
   ...: plt.grid()
   ...: plt.xlabel('t')
   ...: plt.ylabel('y')
```

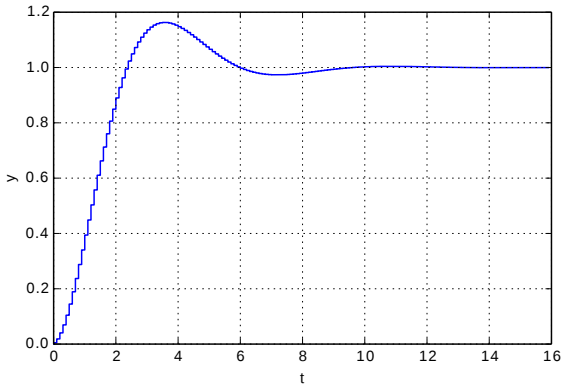


or alternatively

```
In [1]: from control import *

In [2]: from control.matlab import *

In [3]: import matplotlib.pyplot as plt

In [4]: g = tf(1,[1,1,1])

In [5]: y,t = step(g)

In [6]: plt.plot(t,y)
   ...: plt.xlabel('t')
   ...: plt.ylabel('y')
   ...: plt.grid()
```

Figure 1: Step response for continuous-time systems

```
In [1]: from control import *

In [2]: from control.matlab import c2d

In [3]: import matplotlib.pyplot as plt

In [4]: g = tf(1,[1,1,1])

In [5]: gz=c2d(g,0.1)

In [6]: t=arange(0,16,0.1)

In [7]: t1,y = step_response(gz,t)

In [8]: plt.step(t,y.T[0]) # transpose
    col matrix y[0]
   ...: plt.grid()
   ...: plt.xlabel('t')
   ...: plt.ylabel('y')
```

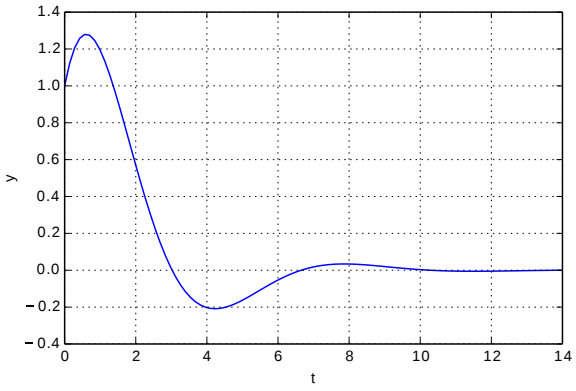


or alternatively

```
In [1]: from control import *

In [2]: from control.matlab import *

In [3]: import matplotlib.pyplot as plt

In [4]: g = tf(1,[1,1,1])

In [5]: gz=c2d(g,0.1)

In [6]: t=arange(0,16,0.1)

In [7]: y,t1 = step(gz,t)

In [8]: plt.step(t,y[0])  # get first
    row from y matrix
   ...: plt.grid()
   ...: plt.xlabel('t')
   ...: plt.ylabel('y')
```

Figure 2: Step response for discrete-time systems

```
In [1]: from control import *

In [2]: import matplotlib.pyplot as plt

In [3]: a=[[0,1],[-1,-1]]

In [4]: b=[[0],[1]]

In [5]: c=[1,0]

In [6]: d=[0]

In [7]: sys=ss(a,b,c,d)

In [8]: t,y=initial_response(sys,
                             X0=[1,1])

In [9]: plt.plot(t,y)
   ...: plt.grid()
   ...: plt.xlabel('t')
   ...: plt.ylabel('y')
```

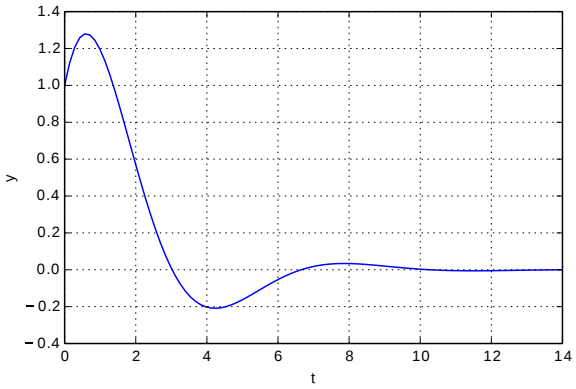


or alternatively

```
In [1]: from control import *

In [2]: from control.matlab import *

In [3]: import matplotlib.pyplot as plt

In [4]: a=[[0,1],[-1,-1]]

In [5]: b=[[0],[1]]

In [6]: c=[1,0]

In [7]: d=[0]

In [8]: sys=ss(a,b,c,d)

In [9]: y,t=initial(sys,X0=[1,1])

In [10]: plt.plot(t,y)
    ...: plt.xlabel('t')
    ...: plt.ylabel('y')
    ...: plt.grid()
```

Figure 3: Continuous time systems - Initial condition response

```
In [1]: from control import *

In [2]: import matplotlib.pyplot as plt

In [3]: g = tf(1,[1,1,1])

In [4]: t,y = impulse_response(g)

In [5]: plt.plot(t,y)
   ...: plt.grid()
   ...: plt.xlabel('t')
   ...: plt.ylabel('y')
```

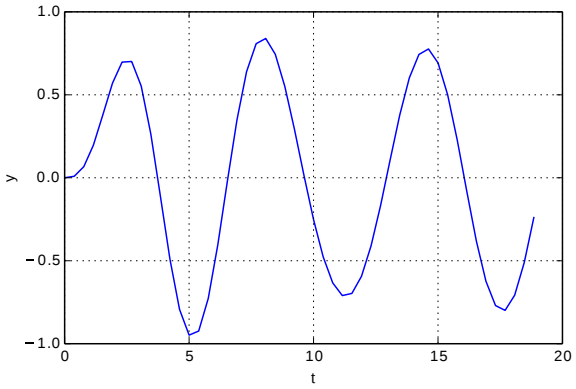


or alternatively

```
In [1]: from control import *

In [2]: from control.matlab import *

In [3]: import matplotlib.pyplot as plt

In [4]: g = tf(1,[1,1,1])

In [5]: y,t = impulse(g)

In [6]: plt.plot(t,y)
   ...: plt.grid()
   ...: plt.xlabel('t')
   ...: plt.ylabel('y')
```

Figure 4: Continuous time systems - Impulse response

```
In [1]: from control import *

In [2]: import matplotlib.pyplot as plt

In [3]: g=tf([1,2],[1,2,3,4])

In [4]: t=linspace(0,6*pi)

In [5]: u=sin(t)

In [6]: t,y,x = forced_response(g,t,u)

In [7]: plt.plot(t,y)
   ...: plt.xlabel('t')
   ...: plt.ylabel('y')
   ...: plt.grid()
```



or alternatively

```
In [1]: from control import *

In [2]: from control.matlab import *

In [3]: import matplotlib.pyplot as plt

In [4]: g=tf([1,2],[1,2,3,4])

In [5]: t=linspace(0,6*pi)

In [6]: u=sin(t)

In [7]: y,t,x = lsim(g,u,t)

In [8]: plt.plot(t,y)
   ...: plt.xlabel('t')
   ...: plt.ylabel('y')
   ...: plt.grid()
```

Figure 5: Continuous time systems - Generic input

4.2 Frequency analysis

The frequency analysis includes some commands like **bode_response**, **nyquist_response**, **nichols_response** and the corresponding Matlab versions **bode**, **nyquist** and **nichols**. (See figures 6, 7 and 8)

```
In [1]: from control import *

In [2]: g=tf([1],[1,0.5,1])

In [3]: bode_plot(g, dB=True);
```

or alternatively

```
In [1]: from control import *

In [2]: from control.matlab import *

In [3]: g=tf([1],[1,0.5,1])

In [4]: bode(g, dB=True);
```

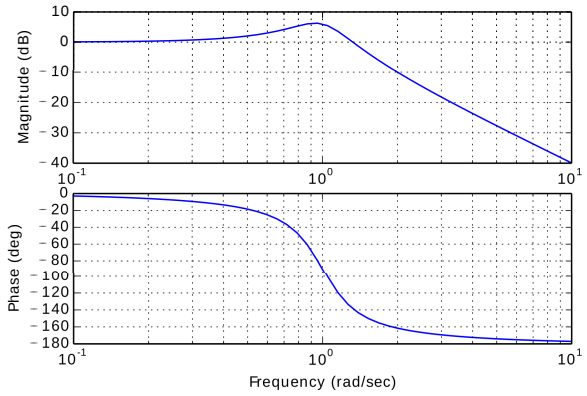


Figure 6: Bode plot

The command **margins** returns the gain margin, the phase margin and the corresponding crossover frequencies.

```
In [1]: from control import *

In [2]: g=tf(2,[1,2,3,1])

In [3]: gm, pm, wg, wp = margin(g)

In [4]: gm                          # Gain, not dB!
Out[4]: 2.5000000000000013

In [5]: pm
Out[5]: 76.274075256921392          # deg

In [6]: wg
Out[6]: 0.85864877610167201         # rad/s

In [7]: wp
Out[7]: 1.7320508075688776          # rad/s
```

In addition, the command **stability_margins** returns the stability margin and the corresponding frequency. The stability margin values w_s and s_m, which correspond to the shortest distance from the Nyquist curve to the critical point -1, are useful for the sensitivity analysis.

```
In [1]: from control import *
In [2]: import matplotlib.pyplot as plt
In [3]: g=tf([1],[1,2,1])
In [3]: nyquist_plot(g), plt.grid()
```

or alternatively

```
In [1]: from control import *
In [2]: import matplotlib.pyplot as plt
In [3]: from control.matlab import *
In [4]: g=tf(1,[1,2,3,4,0])
In [5]: nyquist(g), plt.grid()
```

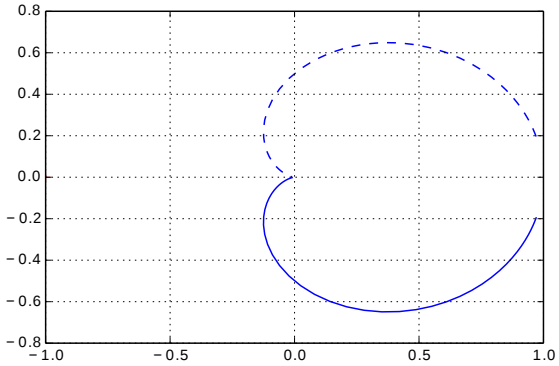


Figure 7: Nyquist plot

```
In [1]: from control import *
In [2]: g=tf(1,[1,2,3,4,0])
In [3]: nichols_plot(g)
```

or alternatively

```
In [1]: from control import *
In [2]: g=tf(1,[1,2,3,4,0])
In [3]: nichols(g)
```

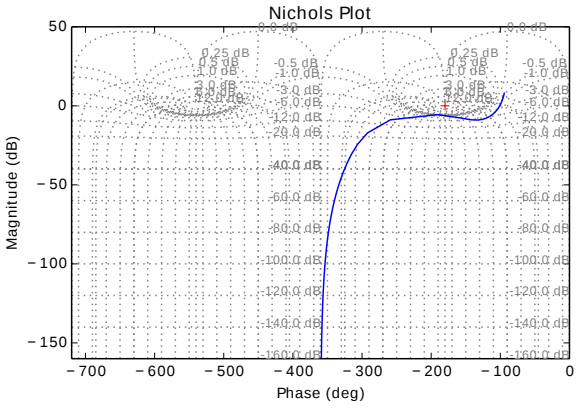


Figure 8: Nichols plot

```
In [1]: from control import *

In [2]: g=tf(2,[1,2,3,1])

In [3]: gm, pm, sm, wg, wp, ws = stability_margins(g)

In [4]: gm
Out[4]: 2.5000000000000013       # Gain not dB'

In [5]: pm
Out[5]: 76.274075256921392       # deg

In [6]: wg
Out[6]: 1.7320508075688776       # rad/s

In [7]: wp
Out[7]: 0.85864877610167201      # rad/s

In [8]: sm
Out[8]: 0.54497577553096421      #

In [9]: ws
Out[9]: 1.3669371206538097       # rad/s
```

4.3 Poles, zeros and root locus analysis

Poles and zeros of an open loop system can be calculated with the commands **pole**, **zero** or plotted and calculated with **pzmap**.

```
In [1]: from control import *

In [2]: from control.pzmap import pzmap

In [3]: g=tf([1,1],[1,2,3,4,0])

In [4]: g.pole()
Out[4]:
array([-1.65062919+0.j          ,
       -0.17468540+1.54686889j,
           -0.17468540-1.54686889j,
               0.00000000+0.j          ])

In [5]: g.zero()
Out[5]: array([-1.])

In [6]: poles, zeros = pzmap(g), grid()

In [7]: poles

Out[7]:
array([-1.65062919+0.j          ,
       -0.17468540+1.54686889j,
           -0.17468540-1.54686889j,
               0.00000000+0.j          ])

In [8]: zeros
Out[8]: array([-1.])
```

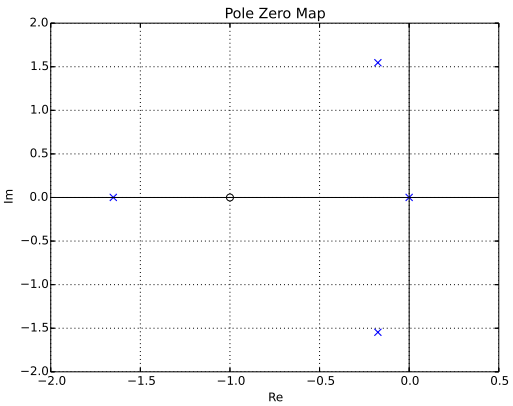


Figure 9: Poles and zeros

In addition there are two functions that implement the root locus command: **rlocus** and **root_locus**. At present no algorithm to automatically choose the values of K has been implemented: if not provided, the K vector is calculated in rlocus with log values between 10^{-3} and 10^{3}. For the **root_locus** function the K values should be provided.

```
In [1]: from control import *

In [2]: g=tf(1,[1,2,3,0])

In [3]: rlocus(g); grid()
```

or alternatively

```
In [1]: from control import *

In [2]: g=tf(1,[1,2,3,0])

In [3]: k=logspace(-3,3,100)

In [4]: root_locus(g,k); grid()
```

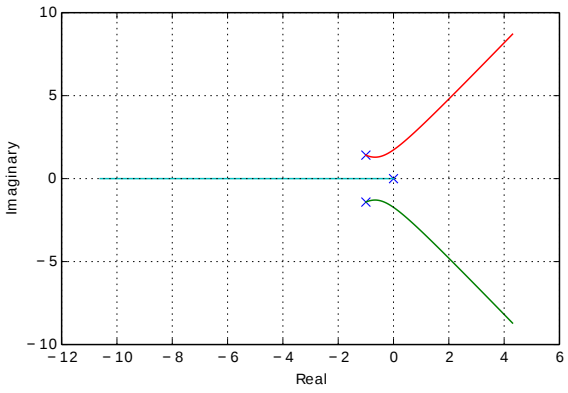


Figure 10: Root locus plot

If the ipython shell is not launched with the **−inline** flag, the root locus is plotted on an external window and it is possible to get the values of gain and damp by clicking with the mouse on the curves.

```
Clicked at     −0.5724     +1.293j gain       1.722 damp
    0.4048
Clicked at      −1.119   +0.01874j gain       2.252 damp
    0.9999
Clicked at     −0.7545     +1.293j gain       1.114 damp
         0.504
```

5 Modeling

The **sympy** module (symbolic python) contains a full set of operations to manage physical systems. In particular, it is possible to find the linearized model of a mechanical system using the Lagrange's method or the Kane's method. More details about the Kane's method are available at [15], [16], [17], [18], [19] and [20].
In the next sections we present the modelling of 3 plants that we can find in our laboratories and that are quite familiar to us.

5.1 Model of a DC motor

5.1.1 Plant

In this first example we model a DC servo motor with a current input in order to find its state-space representation. The motor is characterized by a torque constant k_t, an inertia (motor+load) J and a friction constant D_m.

The input of the plant is the current I and the output is the position φ. The rotation center is the point **O**, the main coordinates system is **N** and we add a local reference frame **Nr** which rotates with the load (angle φ and speed ω).

5.1.2 Modules and constants

```
n [1]: from sympy import symbols, Matrix, pi
   ...: from sympy.physics.mechanics import *
   ...: import numpy as np
   ...:
   ...: # Modeling the system with Kane method
   ...:
   ...: # Signals
   ...: ph = dynamicsymbols('ph')      # motor angle
   ...: w  = dynamicsymbols('w')       # motor rot. speed
   ...: I  = dynamicsymbols('I')       # input current
   ...:
   ...: # Constants
   ...: Dm = symbols('Dm')             # friction
   ...: M, J = symbols('M_J')          # Mass and inertia
   ...: t = symbols('t')               # time
   ...: kt  = symbols('kt')            # torque constant
   ...:
```

5.1.3 Reference frames

```
In [2]: # Reference frame for the motor and Load
   ...: N = ReferenceFrame('N')
   ...:
   ...: O = Point('O')           # center of rotation
   ...: O.set_vel(N,0)
   ...:
   ...: # Reference frames for the rotating disk
   ...: Nr = N.orientnew('Nr', 'Axis',[ph, N.x])     #
       rotating reference (load)
   ...:
   ...: Nr.set_ang_vel(N, w*N.x)
   ...:
```

5.1.4 Body and inertia of the load

```
In [3]: # Mechanics
   ...: Io = J*outer(Nr.x, Nr.x)
   ...:
   ...: InT = (Io, O)
   ...:
   ...: B = RigidBody('B', O, Nr, M, InT)
   ...:
```

5.1.5 Forces and torques

In order to find the dynamic model of the plant we need some other definitions, in particular the relation between angle φ and angular velocity ω, the forces and torques applied to the system and a vector that contains the rigid bodies of the system.

```
In [4]: # Forces and torques
   ...: forces = [(Nr,(kt*I-Dm*w)*N.x)]
   ...:
   ...: kindiffs = [(ph.diff(t)-w)]
   ...:
   ...: bodies=[B]
   ...:
```

5.1.6 Model

Using the Kane's method is now possible to find the dynamic matrices related to the plant.

```
In [5]: # Model
   ...: KM = KanesMethod(N,q_ind=[ph],u_ind=[w],kd_eqs=
        kindiffs)
   ...: fr, frstar = KM.kanes_equations(forces,bodies)
   ...:
   ...: print fr
   ...: print frstar
   ...:
Matrix([[-Dm*w(t) + kt*I(t)]])
Matrix([[-J*Derivative(w(t), t)]])
```

5.1.7 State-space matrices

From the results of the Kane's model identification, we can now extract the matrices A and B of the state-space representation.

```
In [6]: # symbolically linearize about arbitrary
        equilibrium
   ...: linear_state_matrix, linear_input_matrix, inputs =
        KM.linearize()
   ...:
   ...: # set the the equilibrium point
   ...: eq_pt = [0, 0]
   ...: eq_dict = dict(zip([ph,w], eq_pt))
   ...:
   ...: f_A_lin = linear_state_matrix.subs(eq_dict)
   ...: f_B_lin = linear_input_matrix.subs(eq_dict)
   ...: m_mat = KM.mass_matrix_full.subs(eq_dict)
   ...:
   ...: # compute A and B matrices
   ...: A = np.matrix(m_mat.inv() * f_A_lin)
   ...: B = np.matrix(m_mat.inv() * f_B_lin)
```

```
In [6]: print A
   ...: print B
   ...:
[[0  1]
 [0 -Dm/J]]
[[0]
 [kt/J]]
```

5.2 Model of the inverted pendulum

The second example is represented by the classical inverted pendulum as shown in figure 11.

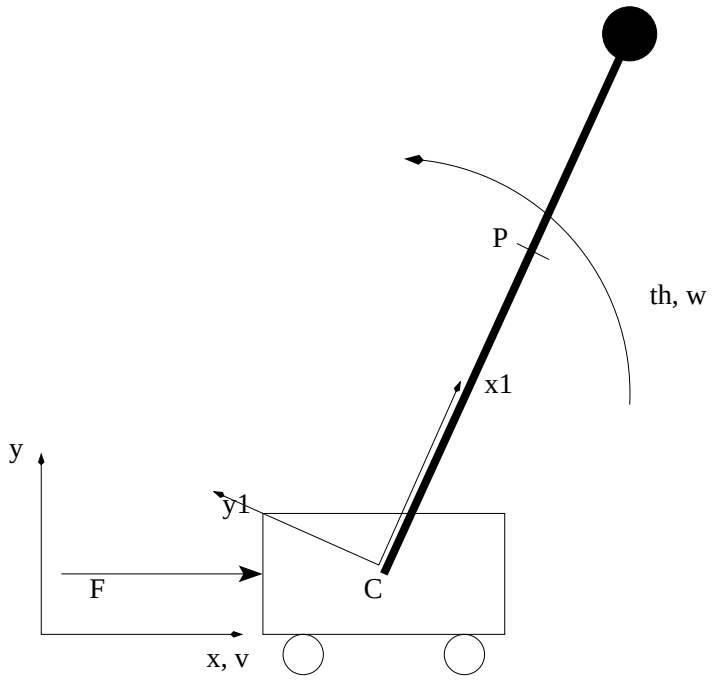


Figure 11: Inverted pendulum

The global reference frame is **Nf** (x, y) The point **P** is the center of mass of the pendulum. The car is moving with speed **v** and position **C**. The pole is rotating with the angle **th** and angular velocity **w**, In addition to the main coordinate frame **Nf** (x, y), we define a local body-fixed frame to the pendulum **Npend** (x_1, y_1).

5.2.1 Modules and constants

```
In [1]: from sympy import symbols, Matrix, pi
   ...: from sympy.physics.mechanics import *
   ...: import numpy as np
   ...:
   ...: # Modeling the system with Kane method
   ...:
   ...: # Signals
   ...: x, th  = dynamicsymbols('x th')
   ...: v, w = dynamicsymbols('v w')
   ...: F = dynamicsymbols('F')
  ...:
   ...: # Constants
   ...: d = symbols('d') # friction
   ...: m, r = symbols('m r')
   ...: M = symbols('M')
   ...: g, t = symbols('g t')
   ...: J = symbols('J')
   ...:
```

5.2.2 Frames - Car and pendulum

```
In [2]: # Frames and Coord. system
   ...:
   ...: # Car - reference x,y
   ...: Nf = ReferenceFrame('Nf')
   ...: C = Point('C')
   ...: C.set_vel(Nf, v*Nf.x)
   ...: Car = Particle('Car',C,M)
   ...:
   ...: # Pendulum - reference x1, y1
   ...: Npend = Nf.orientnew('Npend','Axis',[th,Nf.z])
   ...: Npend.set_ang_vel(Nf,w*Nf.z)
   ...:
   ...: P = C.locatenew('P',r*Npend.x)
   ...: P.v2pt_theory(C,Nf,Npend)
   ...: Pa = Particle('Pa', P, m)
   ...:
```

5.2.3 Points, bodies, masses and inertias

```
In [3]: I = outer (Nf.z, Nf.z)
   ...: Inertia_tuple = (J*I, P)
   ...: Bp = RigidBody('Bp', P, Npend, m, Inertia_tuple)
   ...:
```

5.2.4 Forces, frictions and gravity

```
In [4]: # Forces and torques
   ...: forces = [(C,F*Nf.x-d*v*Nf.x),(P,-m*g*Nf.y)]
   ...: frames = [Nf,Npend]
   ...: points = [C,P]
   ...:
   ...: kindiffs = [x.diff(t)-v, th.diff(t) - w]
   ...: particles = [Car,Bp]
   ...:
```

5.2.5 Final model and linearized state-space matrices

```
n [5]: # Model
   ...: KM = KanesMethod(Nf,q_ind=[x,th],u_ind=[v,w],
        kd_eqs=kindiffs)
   ...: fr,frstar = KM.kanes_equations(forces,particles)
   ...:
   ...: # Equilibrium point
   ...: eq_pt = [0, pi/2,0,0]
   ...: eq_dict = dict(zip([x,th,v,w], eq_pt))
   ...:
   ...: # symbolically linearize about arbitrary
        equilibrium
   ...: linear_state_matrix, linear_input_matrix, inputs =
        KM.linearize()
   ...:
   ...: # sub in the equilibrium point and the parameters
   ...: f_A_lin = linear_state_matrix.subs(eq_dict)
   ...: f_B_lin = linear_input_matrix.subs(eq_dict)
   ...: m_mat = KM.mass_matrix_full.subs(eq_dict)
   ...:
   ...: # compute A and B
   ...: A = m_mat.inv() * f_A_lin
   ...: B = m_mat.inv() * f_B_lin
   ...:
```

```
In [6]: A
Out[6]:
Matrix([
[0,   0,   1,  0],
[0,   0,   0,  1],
[0,    g*m**2*r**2/(J*M + J*m + M*m*r**2), -d*(m**2*r**2/((
    M + m)*(J*M + J*m
+ M*m*r**2)) + 1/(M + m)), 0],
[0, g*m*r*(M + m)/(J*M + J*m + M*m*r**2),
-d*m*r/(J*M + J*m + M*m*r**2),  0]])
```

```
In [7]: B
Out[7]:
Matrix([
[
     0],
[
     0],
[m**2*r**2/((M + m)*(J*M + J*m + M*m*r**2)) + 1/(M + m)],
[                                  m*r/(J*M + J*m + M*m*r**2)]])
```

And the results can be written in a better form as

$$A = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & \frac{gm^2r^2}{JcM+Jcm+Mmr^2} & -\frac{d\left(Jc+mr^2\right)}{JcM+Jcm+Mmr^2} & 0 \\ 0 & \frac{gmr(M+m)}{JcM+Jcm+Mmr^2} & -\frac{dmr}{JcM+Jcm+Mmr^2} & 0 \end{bmatrix}$$

and

$$B = \begin{bmatrix} 0 \\ 0 \\ \frac{Jc+mr^2}{JcM+Jcm+Mmr^2} \\ \frac{mr}{JcM+Jcm+Mmr^2} \end{bmatrix}$$

5.3 Model of the Ball-on-Wheel plant

A more complex plant is represented by the Ball-on-Wheel system of figure 12, where a ball must be maintened in the unstable equilibrium point on the top of a bike wheel.

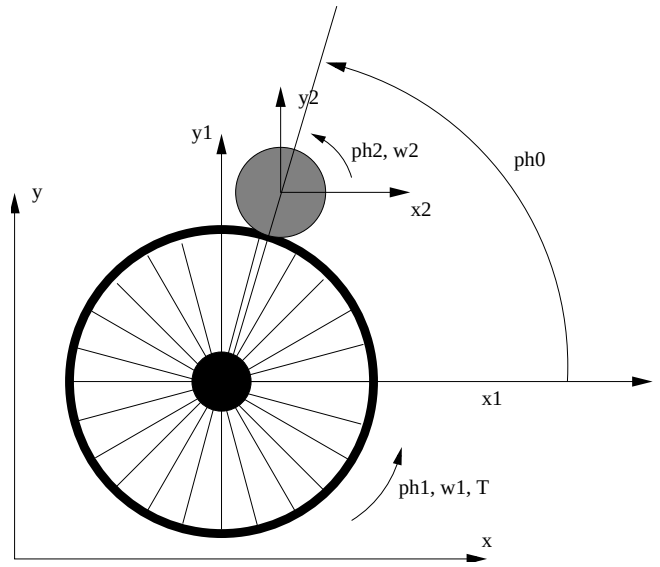


Figure 12: Ball-On-Wheel plant

In this system we have 4 reference frames. The frame **N** is the main reference frame, **N0** rotates with the line connecting the centers of mass of the wheel (**O**) and of the ball (**CM2**), **N1** (x_1, y_1) rotates with the wheel and **N2** (x_2, y_2) is body-fixed to the ball.

The radius of the wheel and of the ball are respectively R_1 and R_2. The non sliding condition is given by

$$(R_1 + R_2) \cdot ph_0 = R_1 \cdot ph_1 + R_2 \cdot ph_2$$

The input of the system is represented by the torque T applied to the wheel.

5.3.1 Modules and constants

```
In [1]: from sympy import symbols, Matrix, pi
   ...: from sympy.physics.mechanics import *
   ...: import numpy as np
   ...:
   ...: ph0, ph1, ph2 = dynamicsymbols('ph0 ph1 ph2')
   ...: w1, w2 = dynamicsymbols('w1 w2')
   ...:
   ...: T = dynamicsymbols('T')
   ...:
   ...: J1, J2 = symbols('J1 J2')
   ...: M1, M2 = symbols('M1 M2')
   ...: R1, R2 = symbols('R1 R2')
   ...: d1     = symbols('d1')
   ...: g      = symbols('g')
   ...: t      = symbols('t')
   ...:
```

5.3.2 Reference frames

```
In [2]: N = ReferenceFrame('N')
   ...:
   ...: O = Point('O')
   ...: O.set_vel(N,0)
   ...:
   ...: ph0 = (R1*ph1+R2*ph2)/(R1+R2)
   ...:
   ...: N0 = N.orientnew('N0','Axis',[ph0,N.z])
   ...: N1 = N.orientnew('N1','Axis',[ph1,N.z])
   ...: N2 = N.orientnew('N2','Axis',[ph2,N.z])
   ...: N1.set_ang_vel(N,w1*N.z)
   ...: N2.set_ang_vel(N,w2*N.z)
   ...:
```

5.3.3 Centers of mass of the ball

```
In [3]: CM2 = O.locatenew('CM2',(R1+R2)*N0.y)
   ...: CM2.v2pt_theory(O,N,N0)
   ...:
Out[3]: (-R1*ph1' - R2*ph2')*N0.x
```

5.3.4 Masses and inertias

```
In [4]: Iz = outer(N.z,N.z)
   ...: In1T = (J1*Iz, O)
   ...: In2T = (J2*Iz, CM2)
   ...:
   ...: B1 = RigidBody('B1', O, N1, M1, In1T)
   ...: B2 = RigidBody('B2', CM2, N2, M2, In2T)
   ...:
```

5.3.5 Forces and torques

```
In [5]: #forces = [(N1, (T-d1*w1)*N.z), (CM2,-M2*g*N.y)]
   ...: forces = [(N1, T*N.z), (CM2,-M2*g*N.y)]
   ...:
   ...: kindiffs = [ph1.diff(t)-w1,ph2.diff(t)-w2]
   ...:
```

5.3.6 Kane's model and linearized state-space matrices

```
In [6]: KM = KanesMethod(N,q_ind=[ph1, ph2],u_ind=[w1, w2
   ...: ],kd_eqs=kindiffs)
   ...: fr, frstar = KM.kanes_equations(forces,[B1, B2])
   ...:

In [7]: # Equilibrium point
   ...: eq_pt = [0, 0, 0, 0, 0]
   ...: eq_dict = dict(zip([ph1,ph2,w1,w2, T], eq_pt))
   ...:

In [8]: # symbolically linearize about arbitrary
   ...: equilibrium
   ...: linear_state_matrix, linear_input_matrix, inputs =
   ...:     KM.linearize()
   ...:
   ...: # sub in the equilibrium point and the parameters
   ...: f_A_lin = linear_state_matrix.subs(eq_dict)
   ...: f_B_lin = linear_input_matrix.subs(eq_dict)
   ...: m_mat = KM.mass_matrix_full.subs(eq_dict)
   ...:
   ...: # compute A and B
   ...: A = m_mat.inv() * f_A_lin
   ...: B = m_mat.inv() * f_B_lin
   ...:
```

```
In [9]: A
Out[9]:
Matrix([
[0, 0, 1, 0],
[0, 0, 0, 1],
[-M2**2*R1**2*R2**2*g/((R1 + R2)*(J1*J2 + J1*M2*R2**2 + J2
    *M2*R1**2)) +
M2*R1**2*g*(M2**2*R1**2*R2**2/((J1 + M2*R1**2)*(J1*J2 + J1
    *M2*R2**2 +
J2*M2*R1**2)) + 1/(J1 + M2*R1**2))/(R1 + R2), -M2**2*R1*R2
    **3*g/((R1 +
R2)*(J1*J2 + J1*M2*R2**2 + J2*M2*R1**2)) + M2*R1*R2*g*(M2
    **2*R1**2*R2**2/((J1 +
M2*R1**2)*(J1*J2 + J1*M2*R2**2 + J2*M2*R1**2)) + 1/(J1 +
    M2*R1**2))/(R1 + R2),
0, 0],
[                                                    -M2**2*R1**3*
    R2*g/((R1 + R2)*(J1*J2
+ J1*M2*R2**2 + J2*M2*R1**2)) + M2*R1*R2*g*(J1 + M2*R1**2)
    /((R1 + R2)*(J1*J2 +
J1*M2*R2**2 + J2*M2*R1**2)),
-M2**2*R1**2*R2**2*g/((R1 + R2)*(J1*J2 + J1*M2*R2**2 + J2*
    M2*R1**2)) +
M2*R2**2*g*(J1 + M2*R1**2)/((R1 + R2)*(J1*J2 + J1*M2*R2**2
     + J2*M2*R1**2)), 0,
0]])
```

```
In [10]: B
Out[10]:
Matrix([
[
              0],
[
              0],
[M2**2*R1**2*R2**2/((J1 + M2*R1**2)*(J1*J2 + J1*M2*R2**2 +
     J2*M2*R1**2)) +
1/(J1 + M2*R1**2)],
[                                                  -M2*R1*R2/(
    J1*J2 + J1*M2*R2**2 +
J2*M2*R1**2)]])
```

or as formula

$$A = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ \frac{J_2 M_2 R_1^2 g}{J_1 J_2 R_1 + J_1 J_2 R_2 + J_1 M_2 R_1 R_2^2 + J_1 M_2 R_2^3 + J_2 M_2 R_1^3 + J_2 M_2 R_1^2 R_2} & \frac{J_2 M_2 R_1 R_2 g}{J_1 J_2 R_1 + J_1 J_2 R_2 + J_1 M_2 R_1 R_2^2 + J_1 M_2 R_2^3 + J_2 M_2 R_1^3 + J_2 M_2 R_1^2 R_2} & 0 & 0 \\ \frac{J_1 M_2 R_1 R_2 g}{(R_1+R_2)\left(J_1 J_2 + J_1 M_2 R_2^2 + J_2 M_2 R_1^2\right)} & \frac{J_1 M_2 R_2^2 g}{(R_1+R_2)\left(J_1 J_2 + J_1 M_2 R_2^2 + J_2 M_2 R_1^2\right)} & 0 & 0 \end{bmatrix}$$

and

$$B = \begin{bmatrix} 0 \\ 0 \\ \frac{M_2^2 R_1^2 R_2^2}{\left(J_1 + M_2 R_1^2\right)\left(J_1 J_2 + J_1 M_2 R_2^2 + J_2 M_2 R_1^2\right)} + \frac{1}{J_1 + M_2 R_1^2} \\ -\frac{M_2 R_1 R_2}{J_1 J_2 + J_1 M_2 R_2^2 + J_2 M_2 R_1^2} \end{bmatrix}$$

6 Control design

6.1 PI+Lead design example

6.1.1 Define the system and the project specifications

In this first example we design a controller for a plant with the transfer function

$$G(s) = \frac{1}{s^2 + 6 \cdot s + 5}$$

The requirements for the control are

$$e_\infty = 0$$

for a step input

$$PM \geq 60^o$$

and

$$\omega_{gc} = 10 rad/s$$

The controller can be written in the form

$$C(s) = K \cdot \frac{1 + s \cdot T_i}{s \cdot T_i} \cdot \frac{1 + \alpha \cdot T_D \cdot s}{1 + s \cdot T_D}$$

with a PI and a lead part.
We have to design the controller and find the values of $\mathbf{T_i}$, α, $\mathbf{T_D}$ and $\mathbf{K}$. The full design is performed using the bode diagram.
After installing the required modules, we can define the plant transfer function and the requirements of the project.

```
In [1]: # Modules

In [2]: from matplotlib.pyplot import *

In [3]: from control import *

In [4]: from numpy import pi, linspace

In [5]: from scipy import sin, sqrt

In [6]: from supsictrl.yottalab import *

In [7]: g=tf([1],[1,6,5])

In [8]: bode(g,dB=True);

In [9]: subplot(211), legend(['G(s)'],prop={'size':10})
Out[9]:
(<matplotlib.axes.AxesSubplot at 0x7f85b5193550>,
 <matplotlib.legend.Legend at 0x7f85b47e6950>)

In [10]: wgc = 10          # Desired Bandwidth

In [11]: desiredPM = 60    # Desired Phase margin
```

Figure 13 shows the bode diagram of the plant.

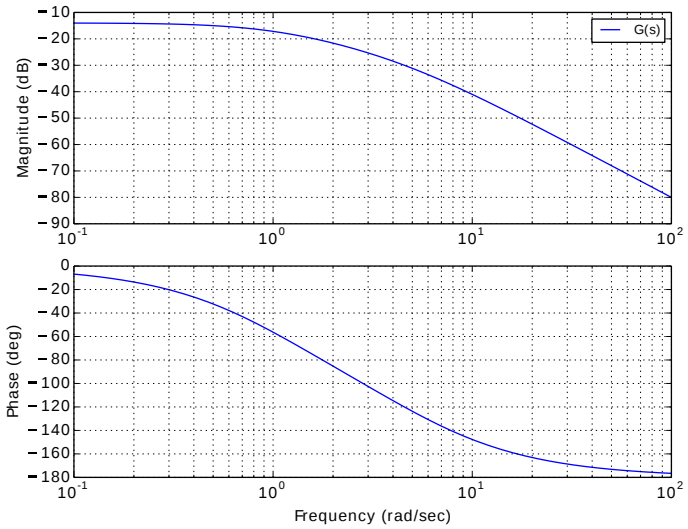


Figure 13: Bode diagram of the plant

6.1.2 PI part

Now we choose the integration time for the PI part of the controller. In this example we set

$$T_i = 0.15s$$

```
In [12]: # PI part

In [13]: Ti=0.15

In [14]: Gpi=tf([Ti,1],[Ti,0])

In [15]: print "PI part is:", Gpi
PI part is:
0.15 s + 1
-----------
  0.15 s


In [16]: figure()
Out[16]: <matplotlib.figure.Figure at 0x7f85b47eaa10>

In [17]: bode(g,dB=True,linestyle='dashed');

In [18]: hold
Out[18]: <function matplotlib.pyplot.hold>

In [19]: bode(Gpi*g,dB=True);

In [20]: subplot(211), legend((['G(s)','Gpi(s)*G(s)']),
    prop={'size':10})
Out[20]:
(<matplotlib.axes.AxesSubplot at 0x7f85b4806250>,
 <matplotlib.legend.Legend at 0x7f85b4303850>)
```

Figure 14 shows the bode plot of the plant with and without the PI controller part.

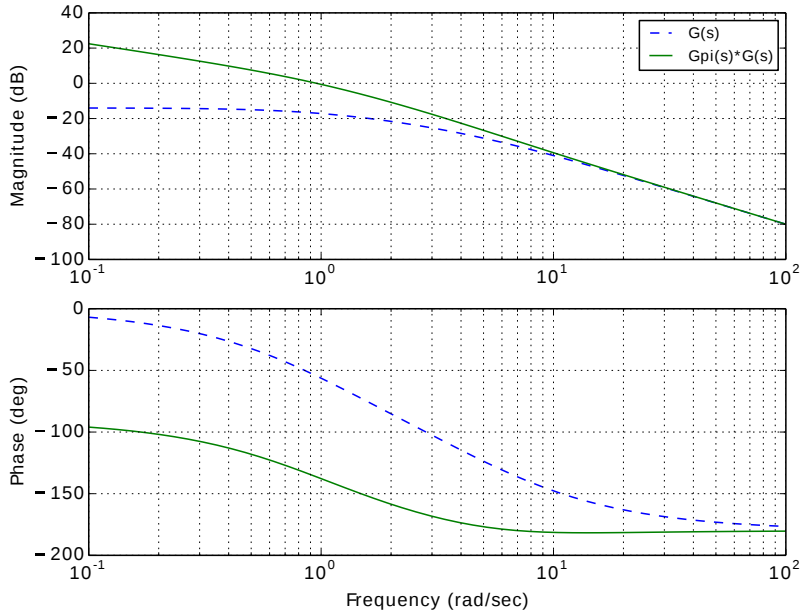


Figure 14: Bode diagram: G (dashed) and Gpi*G

6.1.3 Lead part

Now we can get the PM at the frequency ω_{gc} in order to calculate the additional phase contribution of the lead part of the controller.

```
In [21]: mag,phase,omega = bode(Gpi*g,[wgc],Plot=False)

In [22]: ph = phase[0]

In [23]: if ph>=0:
    ...:     ph = phase[0]-360;
    ...:

In [24]: Phase = -180+desiredPM

In [25]: dPM = Phase-ph

In [26]: print "Additional phase from Lead part: ", dPM
Additional phase from Lead part:  61.4144232114
```

Now it is possible to calculate the lead controller by finding the values of α and T_D.

```
In [27]: # Lead part

In [28]: dPMrad = dPM/180*pi

In [29]: alfa = (1+sin(dPMrad))/(1-sin(dPMrad));

In [30]: print "Alpha is: ", alfa
Alpha is:  15.4073552425

In [31]: TD = 1/(sqrt(alfa)*wgc);

In [32]: Glead = tf([alfa*TD,1],[TD,1])

In [33]: print "Lead part is: ", Glead
Lead part is:
 0.3925 s + 1
--------------
0.02548 s + 1


In [34]: figure()
Out[34]: <matplotlib.figure.Figure at 0x7f85b43462d0>

In [35]: bode(g,dB=True,linestyle='dashed');

In [36]: hold
Out[36]: <function matplotlib.pyplot.hold>

In [37]: bode(Gpi*Glead*g, dB=True);

In [38]: subplot(211),
legend((['G(s)','Gpi(s)*G(s)','Gpi(s)*GLead(s)*G(s)']),
    prop={'size':10})
Out[38]:
(<matplotlib.axes.AxesSubplot at 0x7f85b43736d0>,
 <matplotlib.legend.Legend at 0x7f85b3b1f450>)
```

Figure 15 shows now the bode plot of the plant, the plant with the PI part and the plant with PI and Lead part

6.1.4 Controller Gain

The last step is to find the amplification K of the controller which move up the bode gain plot in order to obtain the required crossover frequency ω_{gc}.

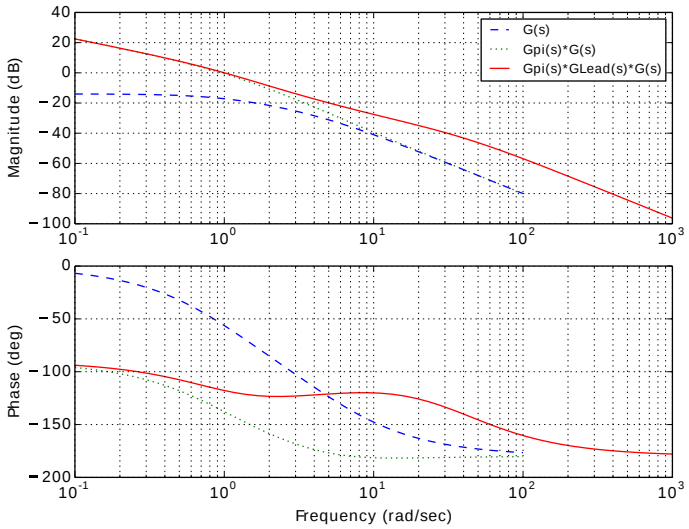


Figure 15: Bode diagram - G (dashed), Gpi*G (dotted) and Gpi*GLead*G

```
In [39]: mag,phase,omega = bode(Gpi*Glead*g,[wgc],Plot=
    False)

In [40]: print "Phase at wgc is:", phase[0]
Phase at wgc is:  -120.0

In [41]: K=1/mag[0]

In [42]: print "Gain to have MAG at gwc 0dB:", K
Gain to have MAG at gwc 0dB:  23.8177769548

In [43]: figure()
Out[43]: <matplotlib.figure.Figure at 0x7f85b3a703d0>

In [44]: bode(g,dB=True,linestyle='dashed');

In [45]: hold
Out[45]: <function matplotlib.pyplot.hold>

In [46]: bode(Gpi*Glead*g, dB=True,linestyle='-.');

In [47]: bode(K*Gpi*Glead*g, dB=True);

In [48]:
subplot(211),legend(['G(s)','Gpi(s)*G(s)','Gpi(s)*GLead(s
    )*G(s)',
'K*Gpi(s)*GLead(s)*G(s)']),prop={'size':10})
Out[48]:
(<matplotlib.axes.AxesSubplot at 0x7f85b3a76690>,
 <matplotlib.legend.Legend at 0x7f85b33e6f90>)
```

In the figure 16 we see now that the gain plot has been translated up to get $0dB$ at the gain crossover frequency ω_{gc}.

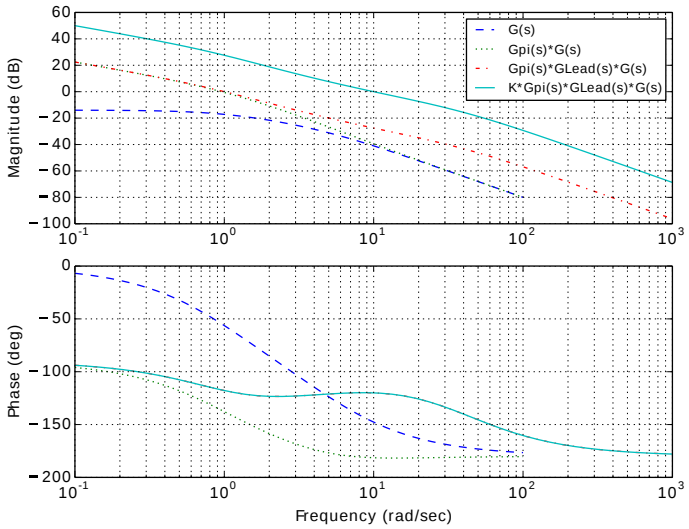


Figure 16: Bode diagram - G (dashed), Gpi*G (dotted), Gpi*GLead*G (dot-dashed) and K*Gpi*GLead*G

6.1.5 Simulation of the controlled system

Now it is possible to simulate the controlled system after closing the loop.

```
In [49]: Contr = K*Gpi*Glead

In [50]: print "Full controller:", Contr
Full controller:
1.402 s^2 + 12.92 s + 23.82
---------------------------
   0.003821 s^2 + 0.15 s

In [51]: mag,phase,omega = bode(K*Gpi*Glead*g,[wgc],Plot=
     False)

In [52]: print "Data at wgc - wgc:", omega[0], "Magnitude
     :",mag[0], "Phase:
",phase[0]
Data at wgc - wgc:  10 Magnitude:  1.0 Phase:  -120.0

In [53]: gt=feedback(K*Gpi*Glead*g,1)

In [54]: t=linspace(0,1.5,300)

In [55]: y,t = step(gt,t)

In [56]: figure()
Out[56]: <matplotlib.figure.Figure at 0x7f85b3514290>

In [57]: plot(t,y), xlabel('t'), ylabel('y'), title('Step
     response of the
controlled plant')
Out[57]:
([<matplotlib.lines.Line2D at 0x7f85b34252d0>],

In [58]: grid()
```

The simulation of the controlled plant with a step input is shown in figure 17.

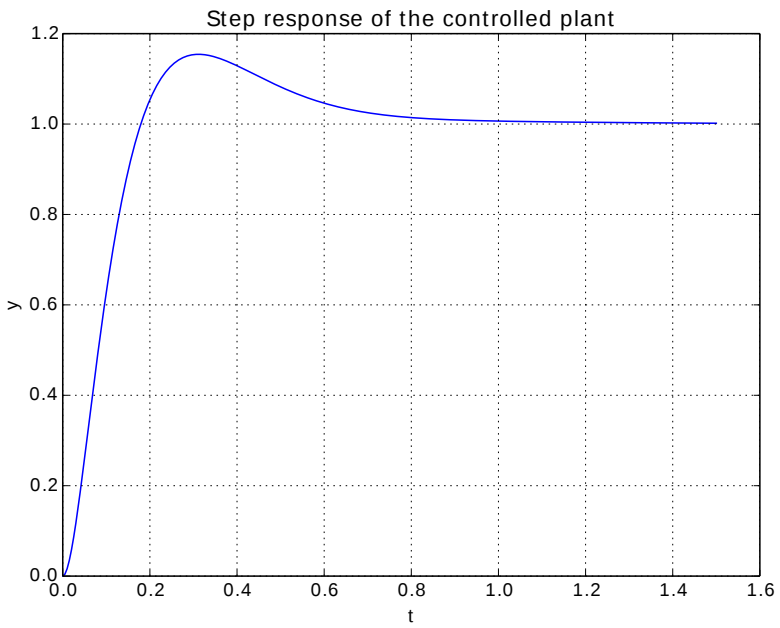


Figure 17: Step response of the controlled plant

6.2 Discrete-state feedback controller design

6.2.1 Plant and project specifications

In this example we design a discrete-state feedback controller for a DC servo motor.
We want to have a controlled system with a maximum of 4% overshooting and an error $e_\infty = 0$ with a step input. In addition we desire a bandwidth of the controlled system of at least 6 rad/s.
The step response of the motor with the current input of $I_{in} = 500mA$) has been saved into the file "MOT".

6.2.2 Motor parameters identification

We try to find the parameters of the srvo motor using a least square identification from the collected data.
The transfer function of the DC motor from input current $I(s)$ to output angle $\Phi(s)$ can be represented as

$$G(s) = \frac{\Phi(s)}{I_{in}(s)} = \frac{K_t/J}{s^2 + s \cdot D/J}$$

6.2.3 Required modules

```
In [1]: from scipy.optimize import leastsq

In [2]: from scipy.signal import step2

In [3]: import numpy as np

In [4]: import scipy as sp

In [5]: from control import *

In [6]: from control.Matlab import *

In [7]: from supsictrl.yottalab import *
```

6.2.4 Function for least square identification

We define now the function **residuals** which returns the error between the collected and the simulated data. Using this function we can try to minimize the error using a least square approach.

```
In [8]: # Motor response for least square identification

In [9]: def residuals(p, y, t):
   ...:     [k,alpha] = p
   ...:     g = tf(k,[1,alpha,0])
   ...:     Y,T = step(g,t)
   ...:     err=y-Y
   ...:     return err
   ...:
```

6.2.5 Parameter identification

We load the collected data to perform the parameter identification of the numerator $K = K_t/J$ and the denominator value $\alpha = D/J$.

```
In [10]: # Identify motor

In [11]: x = np.loadtxt('MOT');

In [12]: t = x[:,0]

In [13]: y = x[:,2]

In [14]: Io = 1000

In [15]: y1 = y/Io
```

```
In [16]: p0 = [1,4]

In [17]: plsq = leastsq(residuals, p0, args=(y1, t))

In [18]: kt = 0.0000382            # Motor torque constant

In [19]: Jm=kt/plsq[0][0]          # Motor Inertia

In [20]: Dm=plsq[0][1]*Jm          # Motor friction

In [21]: g=tf([kt/Jm],[1,Dm/Jm,0])  # Transfer function
```

6.2.6 Check of the identified parameters

The next step is to check how good our parameters have been identified by comparing the simulated function with the measured data (see figure 18)

```
In [22]: Y,T = step(g,t)

In [23]: plot(T,Y,t,y1), legend(('Identified transfer
    function','Collected
data'),prop={'size':10},loc=2), xlabel('t'), ylabel('y'),
    title('Step
response')), grid()
Out[23]:
([<matplotlib.lines.Line2D at 0x7fb9a1b6b590>,
  <matplotlib.lines.Line2D at 0x7fb9a1b6b710>],
 <matplotlib.legend.Legend at 0x7fb9a1b6bb10>,
 <matplotlib.text.Text at 0x7fb9a3cec310>,
 <matplotlib.text.Text at 0x7fb9a1b8b910>,
 <matplotlib.text.Text at 0x7fb9a1b3cbd0>,
 None)
```

6.2.7 Continuous and discrete model

For the state controller design we need to model our motor in the state-space form. We define the continuous-state and the discrete-state space model

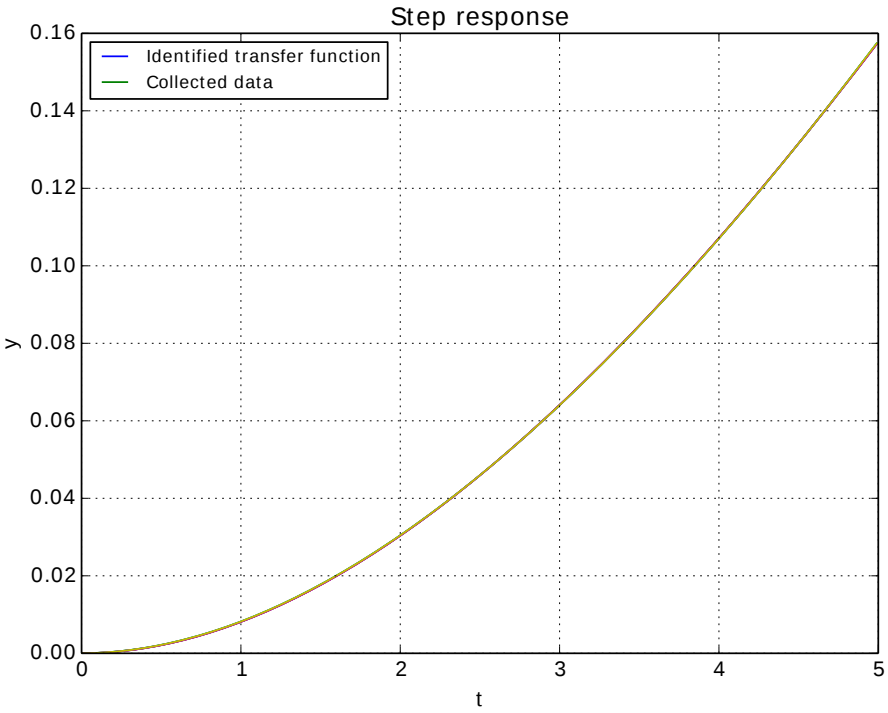


Figure 18: Step response and collected data

```
In [24]: # Design Controller Motor 1

In [25]: a=[[0,1],[0,-Dm/Jm]]

In [26]: b=[[0],[1]]

In [27]: c=[[kt/Jm,0]];

In [28]: d=[0];

In [29]: sysc=ss(a,b,c,d)                    # Continuous
     state-space form

In [30]: Ts=0.01                             # Sampling time

In [31]: sys = c2d(sysc,Ts,'zoh')            # Discrete ss
     form
```

6.2.8 Controller design

For the controller we set a bandwidth to 6 rad/s with a damping factor of $\xi = \sqrt{2}/2$.

```
In [32]: # Control system design

In [33]: print rank(ctrb(sys.A,sys.B))==2     #
    Controllability check
True

In [34]: # State feedback with integral part

In [35]: wn=6

In [36]: xi=np.sqrt(2)/2

In [37]: angle = np.arccos(xi)
```

We add a discrete integral part to eliminate the steady state error and we obtain an additional state for the error between reference and output signal. The two matrices Φ and Γ required by the pole placement routine must be extended with the additional state.

```
In [38]: cl_poles = -wn*array([1, exp(1j*angle), exp(-1j*
    angle)] ) # three poles

In [39]: cl_polesd=sp.exp(cl_poles*Ts)     # Desired
    discrete poles

In [40]: sz1=sp.shape(sys.A);

In [41]: sz2=sp.shape(sys.B);

In [42]: # Add discrete integrator for steady state zero
    error

In [43]: Phi_f=np.vstack((sys.A,-sys.C*Ts))

In [44]: Phi_f=np.hstack((Phi_f,[[0],[0],[1]]))

In [45]: G_f=np.vstack((sys.B,zeros((1,1))))

In [46]: k=place(Phi_f,G_f,cl_polesd)
```

6.2.9 Observer design

Now we can implement the observer: in this example we choose a reduced-order observer and we can use the function provided by the yottalab module to obtain it.

```
In [47]: #Reduced order observer

In [48]: print rank(obsv(sys.A,sys.C))==2      #
     Observability check
True

In [49]: p_oc=-10*max(abs(cl_poles))

In [50]: p_od=sp.exp(p_oc*Ts);

In [51]: T=[0,1]

In [52]: r_obs=red_obs(sys,T,[p_od])
```

6.2.10 Controller in compact form

The yottalab function **comp_form_i** allows to integrate the controller gains and the observer into an unique block.

```
In [53]: # Controller + integral + observer in compact
     form

In [54]: contr_I=comp_form_i(sys,r_obs,k)
```

6.2.11 Anti windup

The last operation consists in dividing the controller into an input part and a feedback part in order to realize the anti-windup mechanism and considering the saturation block.

```
In [55]: # Anti windup

In [56]: [gss_in, gss_out]=set_aw(contr_I,[0,0])
```

6.2.12 Simulation of the controlled plant

The block diagram of the final controlled system is represented in figure 19.

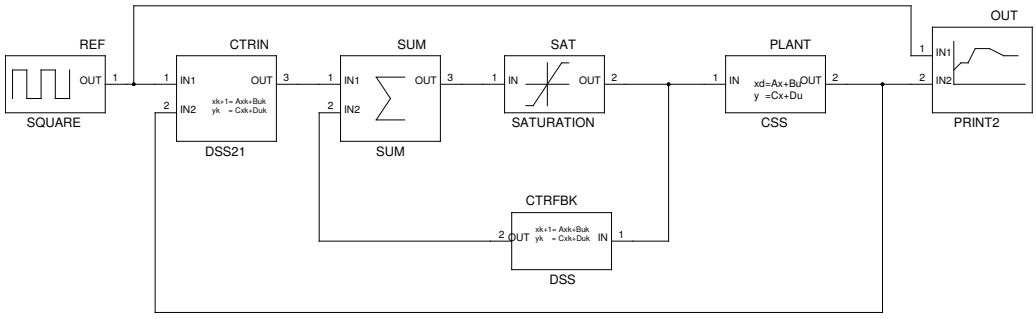


Figure 19: Block diagram of the controlled system

It is not possible to simulate the resulting system in Python because of:

- The controller is discrete and the plant is continuous. At present it is not possible to perform hybrid simulation. In some cases we can substitute the plant with the discrete-time system and perform a discrete simulation.
- The block "CTRIN" has two inputs. The step function can only find the output from a single input.
- The control toolbox can handle only linear system (and there is a saturation in the final system).

7 Hybrid simulation and code generation

CACSD environments usually offer a graphical editor to perform the hybrid simulation (Matlab↔Simulink, Scioslab↔Scicos, Scilab↔xCos etc.). At present no tool like Simulink is available for Python.
In order to perform hybrid simulations, different open source tools have been tested. After this analysis we choose the gEda tool suite [21]. The gEda suite contains a set of applications used for electronic design and it is released under the GNU Public License [22]. The gEDA tool suite offers the possibility to easy create and integrate new blocks. In particular we have the possibility to create a set of new blocks related to the control design. A little set of these blocks is shown in figure 20.

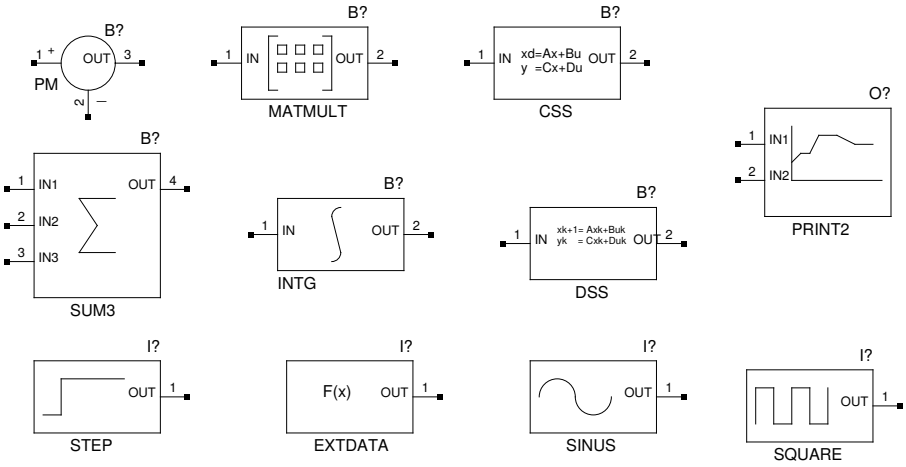


Figure 20: Some gEDA blocks for control design

In the following we'll explain how it is possible, from the gEDA schematics, to generate code for the hybrid simulation. Code for the RT controller can be generated in the same way: users should only replace the mathematical model of the plant with the blocks interfacing the sensors and the actuators of the real system.

8 Code generation

8.1 Basics

Each element of a block diagram is defined with two functions:

The interface function that describes how the block must be drawn in the block diagram

The Implementation function that contains the code to be executed to perform the tasks related with this block.

In addition we need to know all the nodes connected to the inputs and to the outputs of each block.

8.2 Interface functions

Each block is defined as a **symbol** in the gEDA schematics and stored into a gEDA library. This library is loaded at the start of the gEDA schematic application. The symbol file contains the instructions required to draw the symbol and the default parameters of the block. Using **drag and drop**, symbols can be easily inserted in the block diagram and connected to other elements.
Each block must be renamed with a unique name, and its parameters can be modified directly in the **gschem** application or using the python program **pyAttrib**.

8.3 The implementation functions

In a schematic, each block can be described with the functions (1) for continuous-time systems or (2) for discrete-time systems.

$$\begin{aligned} \mathbf{y} &= \mathbf{g}(\mathbf{x}, \mathbf{u}, t) \\ \dot{\mathbf{x}} &= \mathbf{f}(\mathbf{x}, \mathbf{u}, t) \end{aligned} \tag{1}$$

$$\begin{aligned} \mathbf{y}_k &= \mathbf{g}(\mathbf{x}_k, \mathbf{u}_k, k) \\ \mathbf{x}_{k+1} &= \mathbf{f}(\mathbf{x}_k, \mathbf{u}_k, k) \end{aligned} \tag{2}$$

The $\mathbf{g}(\ldots)$ function represents the static part of the block. This function is used to read inputs, read sensors, write actuators or update the outputs of the block.
The second function ($\mathbf{f}(\ldots)$) is only required if the block has internal states, and it is only used by dynamic systems. In addition, each block implements two other functions, one for the block initialization and one to cleanly terminate it.
All these functions are programmed as C-files, compiled and archived into a library.

8.4 I/O connections

The gEDA tool suite offers an application called **gnetlist**, which is used to generate net lists for different targets. In particular, we generate here a net list for the **spice** simulator. The resulting net list describes in particular the nodes connected to the inputs and outputs of each block.
After this first setup it is possible to translate the block diagram into a list of elements of the class **RCPblk** provided by the **suspictrl** package. This class contains all the information required for the code generation and can be expanded in the future to handle additional fields (ex. the type of the I/O signals: int, double etc.).
This class contains the following fields:

fcn: the name of the C-Function to be used to handle this block

pin: an array containing the id of the input nodes

pout: an array containing the id of the output nodes

nx: the number of internal states (continuous or discrete)

uy: a flag which indicates a direct dependency between input and output signals (feed-through flag).

realPar: an array containing the real parameters of the block

intPar: an array containing the integer parameters of the block

str: a string related to the block

For example, the diagram of figure 21

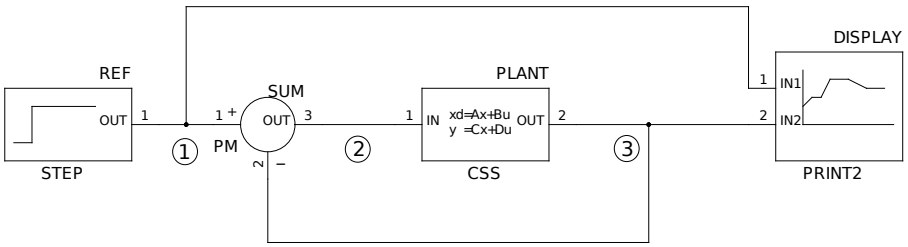


Figure 21: Simple block diagram

is translated into the following net list

```
* Spice netlister for gnetlist
DISPLAY 1 3 nin:2|nout:0|printBlk
PLANT 2 3 nin:1|nout:1|cssBlk|System: tf(1,[1,1,1])|
    Initial
                    conditions: 0
SUM 1 3 2 nin:2|nout:1|sumBlk|Gains: [1,-1]
REF 1 nin:0|nout:1|stepBlk|Step Time: 1|Step Value: 1
.END
```

The block **PLANT** has one input connected to node ② and one output connected to node ③, it is a continuous transfer function (cssBlk, $1/(s+1)$) with zero initial conditions. The **SUM** block has 2 inputs connected to node ① and ③, one output connected to node ② and performs a subtraction of the input signals (Gains: [1,-1]), etc.
This resulting net list is parsed and translated into 4 objects of the class **RCPblk**:

```
DISPLAY = printBlk([1, 3])
PLANT = cssBlk([2],[3], tf(1,[1,1,1]), 0)
SUM = sumBlk([1, 3],[2], [1,-1])
REF = stepBlk([1], 1, 1)

blks = [DISPLAY, PLANT, SUM, REF]
```

8.5 Translating the block list into C-code

8.5.1 Finding the right execution sequence

Before starting with the translation of the block diagram into C.code, we need to find the correct sequence of execution of the blocks. This task can be performed by analizing the *uy* flag of the block object. When in a block the *uy* flag is set to 1, we need the output of the blocks connected at his input before starting to update his output. This means that we have to generate a dependency tree of all the blocks and then we must rearrange the order of the block list for code generation.
In linear blocks for examples, the *uy* flag is set if the D matrix is not null.
In the blockdiagram of figure 21, the **SUM** and the **DISPLAY** blocks require to know their inputs before update their outputs.

```
In [5]: NrOfNodes = 3

In [6]: ordered_list = detBlkSeq(NrOfNodes, blks)

In [7]: for n in ordered_list:
   ...:     print n
   ...:
Function            : css
Input ports         : [2]
Outputs ports       : [3]
Nr. of states       : [2 0]
Relation u->y       : 0
Real parameters     : [[ 0.  0. -1.  1. -1. -1.
                         0.  0. -1.  0.  0.  0.]]
Integer parameters : [ 2  1  1  1  5  7  9 10]
String Parameter    :

Function            : step
Input ports         : []
Outputs ports       : [1]
Nr. of states       : [0 0]
Relation u->y       : 0
Real parameters     : [1 1]
Integer parameters : []
String Parameter    :

Function            : print
Input ports         : [1 3]
Outputs ports       : []
Nr. of states       : [0 0]
Relation u->y       : 1
Real parameters     : []
Integer parameters : []
String Parameter    :

Function            : sum
Input ports         : [1 3]
Outputs ports       : [2]
Nr. of states       : [0 0]
Relation u->y       : 1
Real parameters     : [ 1 -1]
Integer parameters : []
String Parameter    :
```

If the block diagram contains algebraic loops it is not possible to find a solution for the **detBlkSeq** function and an error is raised.

8.5.2 Generating the C-code

Starting from the ordered list of blocks, it is possible to generate C-code.
The code contains 3 functions:

- The initialization function
- The termination function
- The periodic task

8.5.3 The init function

In this function each block is translated into a python_block structure defined as follows:

```
typedef struct {
  int nin;              /* Number of inputs */
  int nout;             /* Number of outputs */
  int *nx;              /* Cont. and Discr states */
  void **u;             /* inputs */
  void **y;             /* outputs */
  double *realPar;      /* Real parameters */
  int *intPar;          /* Int parameters */
  char * str;           /* String */
  void * ptrPar;        /* Generic pointer */
}python_block;
```

The nodes of the block diagram are defined as "double" variables and the inputs and outputs of the blocks are defined as vectors of pointers to them.

```
...
/* Nodes */
static double Node_1[] = {0.0};
static double Node_2[] = {0.0};
static double Node_3[] = {0.0};

/* Input and outputs */
static void *inptr_0[]  = {0};
static void *outptr_0[] = {0};
static void *outptr_1[] = {0};
static void *inptr_2[]  = {0,0};
static void *inptr_3[]  = {0,0};
static void *outptr_3[] = {0};
...
  inptr_0[0]  = (void *) Node_2;
  outptr_0[0] = (void *) Node_3;
..
  block_test[0].nin  = 1;
  block_test[0].nout = 1;
  block_test[0].nx   = nx_0;
  block_test[0].u    = inptr_0;
  block_test[0].y    = outptr_0;
...
```

After this initialization phase, the implementation functions of the blocks are called with the flag **INIT**.

```
  css(INIT, &block_test[0]);
  step(INIT, &block_test[1]);
  print(INIT, &block_test[2]);
  sum(INIT, &block_test[3]);
```

8.5.4 The termination function

This procedure calls the implementation functions of the blocks with the flag **END**.

8.5.5 The ISR function

This procedure represents the periodic task of the RT execution. First of all, the implementation functions are called with the flag **OUT**, in order to perform the output update of each blocks. As a

second step, the implementation functions of the block containing internal states ($nx \neq 0$) are called with the flag **STUPD** (state update).

```
  ...
  css(OUT, &block_test[0]);
  step(OUT, &block_test[1]);
  print(OUT, &block_test[2]);
  sum(OUT, &block_test[3]);
  ...
  css(OUT, &block_test[0]);
  css(STUPD, &block_test[0]);
  ...
```

8.6 The main file

The core of the RT execution is represented by the "python_main_rt.c" file. During the RT execution, the main procedure starts a high priority thread for handling the RT behavior of the system. The following main file, for example, is used to launch the executable in a Linux preempt_rt environment.

```
void *rt_task(void *p)
{
  ...
  param.sched_priority = prio;
  if(sched_setscheduler(0, SCHED_FIFO, &param)==-1){
    perror("sched_setscheduler failed");
    exit(-1);
  }

  ...
  double Tsamp = NAME(MODEL,_get_tsamp)();

  ...
  NAME(MODEL,_init)();

  while(!end){
   /* wait untill next shot */
    clock_nanosleep(CLOCK_MONOTONIC,
                    TIMER_ABSTIME, &t, NULL);

    ...
    /* periodic task */
    NAME(MODEL,_isr)(T);
    ...
  }
  NAME(MODEL,_end)();
}
```

8.7 The pyCodeGen application

A python application (pyCodeGen) allows to manage the full development (Figure 22), from the schematic to the C-code generation.
The user should provide a schematic file , a template makefile (for simulation or RT execution) and an optional python initialization script. The script can be used to design the controller and to define the variables (like a Matlab file for a Simulink model).

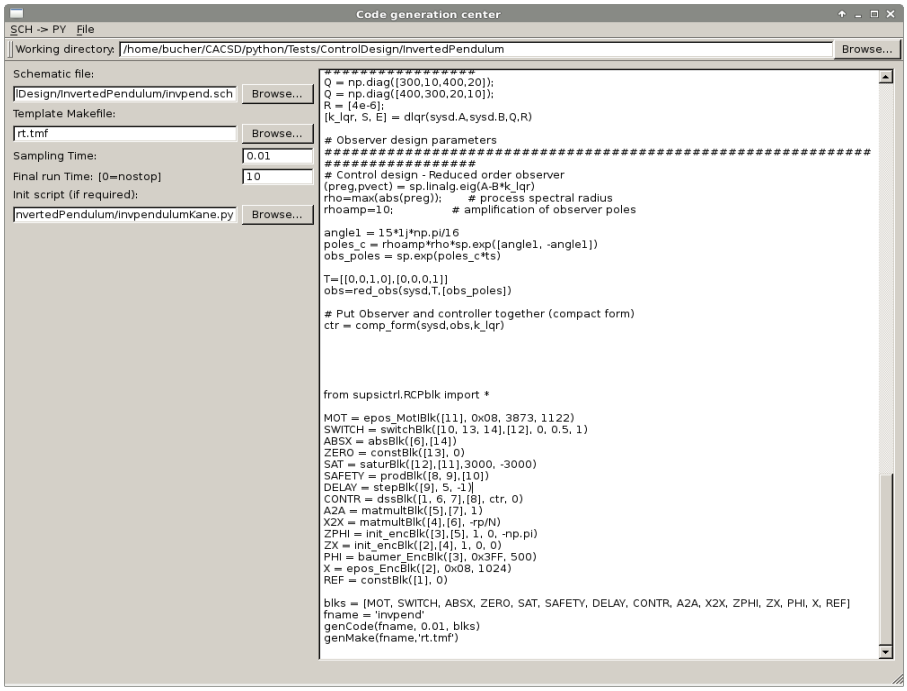


Figure 22: The pyCodeGen environment

9 Example

9.1 The plant

One of the educational plants available at the SUPSI laboratory is the system shown in figure 23.

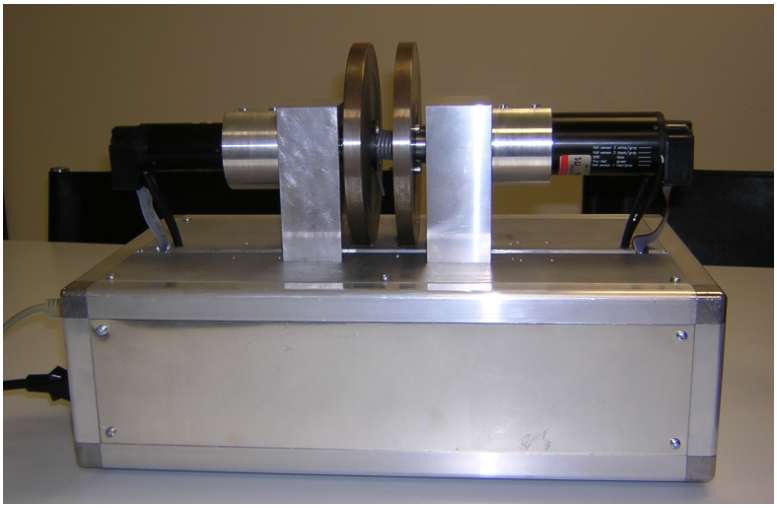


Figure 23: The disks and spring plant

Two disks are connected by a spring. The goal for the students is to control the angle of the disk on the right by applying an appropriate torque to the disk on the left.
The physical model of this plant can be directly calculated in python using for example the **sympy** toolbox. Sympy can deliver a symbolic description of the system and through a python *dictionary* it is possible to easily obtain the numerical matrices of the state-space representation of the plant.

```
In [4]: A
Out[4]:
matrix([[0, 0, 1, 0],
        [0, 0, 0, 1],
        [-c/J1, -c/J1, (-d - d1)/J1, -d/J1],
        [-c/J2, -c/J2, -d/J2, (-d - d2)/J2]])

In [5]: B1

Out[5]:
matrix([[0, 0],
        [0, 0],
        [kt1/J1, 0],
        [0, kt2/J2]])

In [6]: B = B1[:,0]

In [7]: C
Out[7]: [[1, 0, 0, 0], [0, 1, 0, 0]]

In [8]: C2
Out[8]: [0, 1, 0, 0]

In [9]: D
Out[9]: [[0], [0]]

In [10]: D2
Out[10]: [0]
```

The control system toolbox and the additional "yottalab.py" package contain all the functions required for the design of the controller. In this case we design a discrete-state feedback controller with integral part for eliminating steady-state errors. The states are estimated with a reduced-order observer. In addition, an anti-windup mechanism has been implemented. The sampling time is set to 10 ms.
The yottalab module offers 3 functions that facilitate the controller design:

- The function **red_obs**(sys, T, poles) which implements the reduced-order observer for the system **sys**, using the submatrix **T** (required to obtain the estimator C-matrix and the desired state-estimator poles **poles**.

$$P = [C; T] \rightarrow C^* = C \cdot P^{-1} = [I_q, O_{(n-q)}]$$

- The function **comp_form_i**(sys,obs,K,Cy) that transforms the observer **obs** with the state-feedback gains **K** and the integrator part into a single dynamic block with the reference signal and the two positions φ_1 and φ_2 as inputs and the control current I_1 as output. The vector **Cy** is used to select φ_2 as the output signal that is compared with the reference signal for generating the steady-state error for the integral part of the controller.

- The function **set_aw**(sys,poles) that transforms the previous controller ($Contr(s) = N(s)/D(s)$) in an input state-space system and a feedback state-space system, implementing the anti-windup mechanism. The vector **poles** contains the desired poles of the two new systems ($D_{new}(s)$) (see figure 24).

$$sys_{in}(s) = \frac{N(s)}{D_{new}(s)}$$

$$sys_{fbk}(s) = 1 - \frac{D(s)}{D_{new}(s)}$$

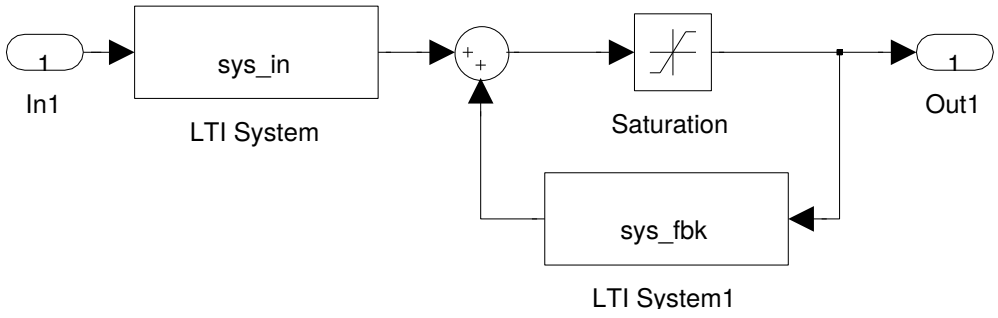


Figure 24: Anti windup

9.2 The plant model

```
# Sampling time
ts = 10e-3

gss1 = ss(A,B,C,D)
gss = ss(A,B,C2,D2)
gz = c2d(gss,ts,'zoh')
```

9.3 Controller design

```
# Control design
wn = 10
xi1 = np.sqrt(2)/2
xi2 = 0.85

cl_p1 = [1,2*xi1*wn,wn**2]
cl_p2 = [1,2*xi2*wn,wn**2]
cl_p3 = [1,wn]
cl_poly1 = sp.polymul(cl_p1,cl_p2)
cl_poly = sp.polymul(cl_poly1,cl_p3)
cl_poles = sp.roots(cl_poly)      # Desired continuous
    poles
cl_polesd = sp.exp(cl_poles*ts)  # Desired discrete poles

# Add discrete integrator for steady state zero error
Phi_f = np.vstack((gz.A,-gz.C*ts))
Phi_f = np.hstack((Phi_f,[[0],[0],[0],[0],[1]]))
G_f = np.vstack((gz.B,zeros((1,1))))

# Pole placement
k = placep(Phi_f,G_f,cl_polesd)
```

9.4 Observer design

```
# Observer design - reduced order observer
poli_o = 5*cl_poles[0:2]
poli_oz = sp.exp(poli_o*ts)

disks = ss(A,B,C,D)
disksz = StateSpace(gz.A,gz.B,C,D,ts)
T = [[0,0,1,0],[0,0,0,1]]

# Reduced order observer
r_obs = red_obs(disksz ,T, poli_oz)

# Controller and observer in the same matrix - Compact
    form
contr_I = comp_form_i(disksz,r_obs,k,[0,1])

# Implement anti windup
[gss_in,gss_out] = set_aw(contr_I,[0.1,0.1,0.1])
```

9.5 Simulation

We can perform the simulation of the discrete-time controller with the continuous-time mathematic plant model using the block diagram of figure 25

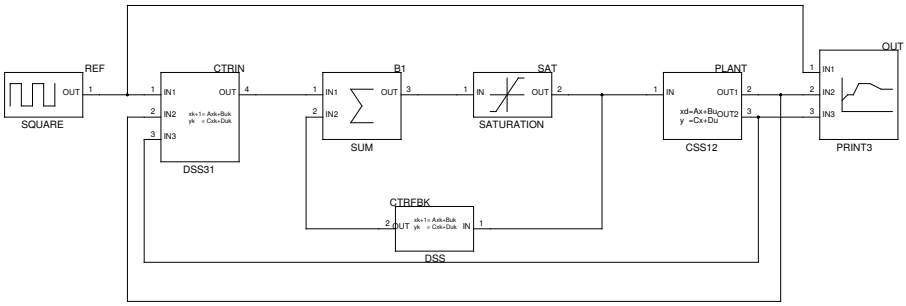


Figure 25: Block diagram for the simulation

The plant is represented by a continuous-time state-space block with 1 input and 2 outputs. The controller implements the state-feedback gains and the state observer and it has been split into a CTRIN block and a CTRFBK block in order to implement the anti-windup mechanism.
We can now generate the code for the simulation and launch the generated executable. The template makefile for this executable is **sim.tmf**.
The plots resulting from the simulation are shown in figure 26.

9.6 Real-time controller

In order to generate the RT controller for the real plant, we first have to substitute the plant with the interfaces for sensors and actuators using blocks that send and receive CAN message using a USB dongle of Peak System. The template makefile for this system is now **rt.tmf**, that allows to generate code with real-time behaviour.
The block diagram for the real-time controller is represented in figure 27.
The motor position can be plotted in python at the end of the execution (see figure 28).

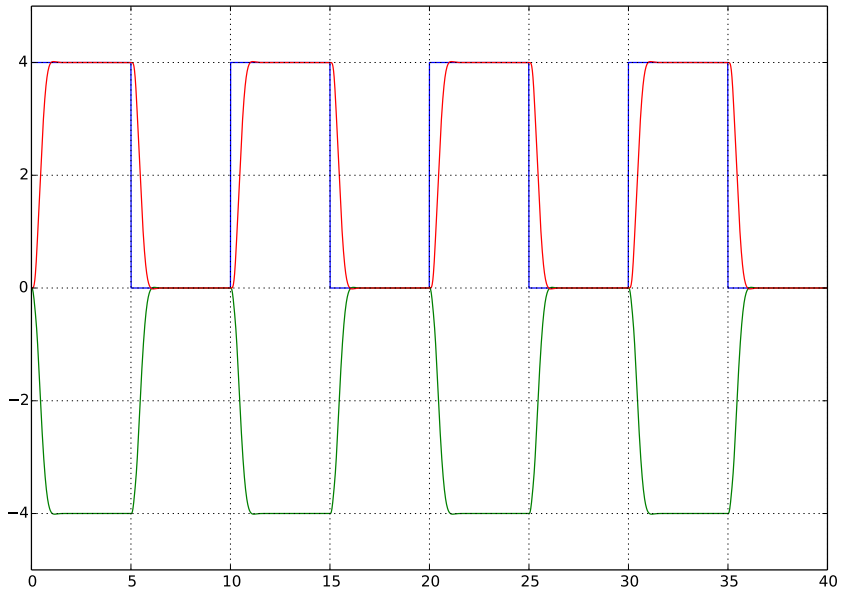


Figure 26: Simulation of the plant

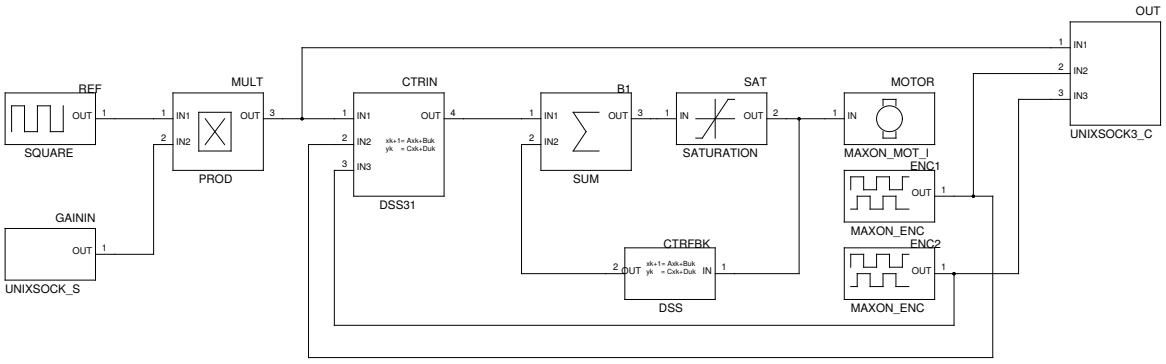


Figure 27: Block diagram for the RT implementation

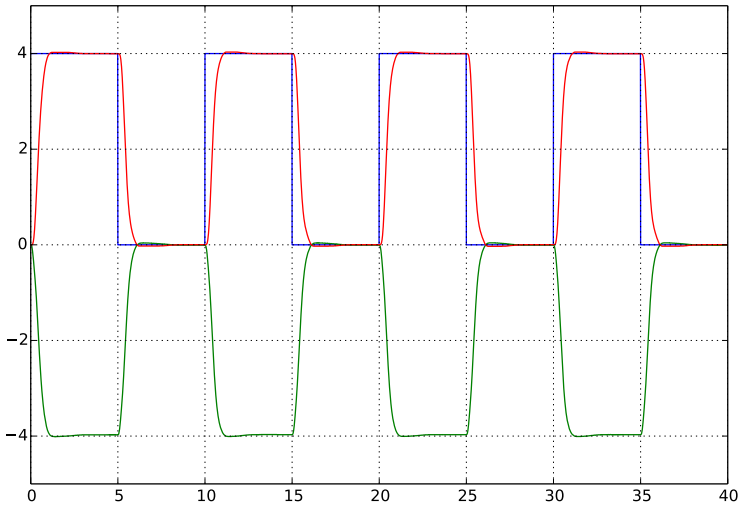


Figure 28: RT execution

10 Future work

As shown in this document, basic design of controllers with Python is possible. The set of available functions is not yet complete, but everybody can add the own contributions to increase the usability of the project, as we did with the **supsictrl** package.
The gEDA suite is not the right tool to perform hybrid simulation and code generation. A new graphical editor should be implemented to solve some problems related to this tool:

- At present, no check during the edit phase is performed: it is possible for example to link two outputs of blocks together without rising an error. It is possible to find such error only in the code generation phase.
- It is not possible to dynamically increase or decrease the number of inputs or outputs of a block. We must propose the same block with different I/O in the block library.

Despite of this (minor) problem, the gEDA suite demonstrates that it is possible, with a little effort, to translate a block diagram into C-code.
At present only a simple Runga-Kutta algorithm for numerical integration has been implemented: in the future, the toolbox must contain more integration algorithms for system simulation. A solution could be the integration of the generated code (with only discrete blocks) into another open source simulation tool like for example **Modelica** [23].

References

[1] Richard Murray and Carl Åström. Feedback Systems. [Online]. Available: http://www.cds.caltech.edu/ murray/books/AM08/pdf/am08-complete_28Sep12.pdf

[2] VirtualBox. [Online]. Available: https://www.virtualbox.org

[3] Download Anaconda. [Online]. Available: http://continuum.io/downloads

[4] Obtaining NumPy and SciPy libraries. [Online]. Available: http://www.scipy.org/scipylib/download.html

[5] Python Control toolbox. [Online]. Available: https://github.com/python-control/python-control

[6] Slycot. [Online]. Available: https://github.com/jgoppert/Slycot

[7] pyControlDistro.tgz. [Online]. Available: http://robertobucher.dti.supsi.ch/python/

[8] Slycot Master - 0.1.0. [Online]. Available: http://www.lfd.uci.edu/ gohlke/pythonlibs/#slycot

[9] NumPy for Matlab Users. [Online]. Available: http://wiki.scipy.org/NumPy_for_Matlab_Users

[10] David J. Pine. Introduction to Python for Science. [Online]. Available: https://github.com/djpine/pyman

[11] Tentative NumPy Tutorial. [Online]. Available: http://wiki.scipy.org/Tentative_NumPy_Tutorial

[12] SciPy Tutorial. [Online]. Available: http://docs.scipy.org/doc/scipy-0.14.0/reference/tutorial/index.html

[13] Matplotlib. [Online]. Available: http://matplotlib.org/

[14] SymPy Tutorial. [Online]. Available: http://docs.sympy.org/dev/tutorial/index.html

[15] Kanes Method in Physics/Mechanics. [Online]. Available: http://docs.sympy.org/0.7.5/modules/physics/mechanics/kane.html

[16] Kanes Method and Lagranges Method (Docstrings). [Online]. Available: http://docs.sympy.org/latest/modules/physics/mechanics/api/kane_lagrange.html

[17] P. C. M. . T. R. Kane. Motion Variables Leading to Efficient Equations of Motions. [Online]. Available: http://www2.mae.ufl.edu/ fregly/PDFs/efficient_generalized_speeds.pdf

[18] A Brief Synopsis of Kanes Method. [Online]. Available: www.cs.cmu.edu/ delucr/kane.doc

[19] L. A. Sandino1, M. Bejar2, and A. Ollero1. Tutorial for the application of Kanes Method to model a small-size helicopter. [Online]. Available: http://grvc.us.es/publica/congresosint/documentos/Sandino_RED-UAS_Sevilla2011.pdf

[20] A. Purushotham1 and M. J.Anjeneyulu. Kanes Method for Robotic Arm Dynamics: a Novel Approach. [Online]. Available: http://www.iosrjournals.org/iosr-jmce/papers/vol6-issue4/B0640713.pdf

[21] gEDA homepage. [Online]. Available: http://www.geda-project.org/

[22] GNU General Public License. [Online]. Available: http://www.gnu.org/copyleft/gpl.html

[23] OpenModelica. [Online]. Available: https://openmodelica.org/