Python for control purposes

Prof. Roberto Bucher

roberto.bucher.2812@gmail.com

July 25, 2024

Contents

List of Figures

Chapter 1

Introduction

1.1 Install the packages

The best way to install the control package, including the GUI pysimCoder, is to follow the method that can be found at this address([1])

`https://github.com/robertobucher/LinuxLabo`

Here it is possible to find a Makefile that can install all the required files in Ubuntu and Debian. More info are available at the github page of the pysimCoder project([2]):

`https://github.com/robertobucher/pysimCoder`

1.2 Video

Felipe Depine ([3]) is registering some videos about installation and use of the pysimCoder tool. The video are available at the Robots5 Youtube channel([4]).

Chapter 2

The Python Control System toolbox

2.1 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 implements common operations for the analysis and design of feedback control systems([5]).

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.

A complete description of the python control toolbox is available here:

`https://python-control.readthedocs.io/en/latest/`

In this chapter I introduce some basics of the Control system toolbox for Python. This tool is still in development. A more complete description of the toolbox is available here ([5])

2.2 Models

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

2.3 Continuous systems

2.4 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]]
```

2.5 Transfer function

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

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

In [3]: print(g)

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

2.6 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.

2.7 Discrete time systems

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

2.8 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
```

2.9 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
```

2.10 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.ctr_repl package implements the function **d2c** with the methods “zoh’, ”foh“and ”tustin“.

```
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 supsictrl.ctrl_repl import d2c

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

In [4]: gd =c2d(g,0.01)

In [5]: g2=d2c(gd)

In [6]: print(g)

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


In [7]: print(g2)

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

2.11 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.]]
```

2.12 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
```

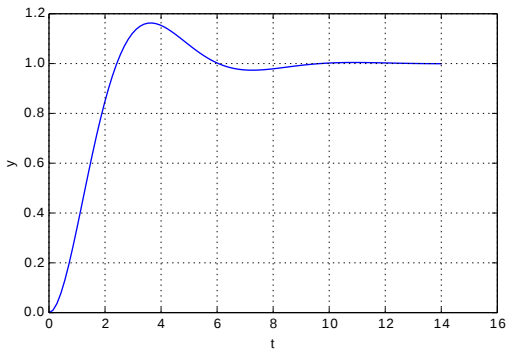
Chapter 3

System analysis

3.1 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 3.1, 3.2, 3.3, 3.4 and 3.5.

```
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')
```

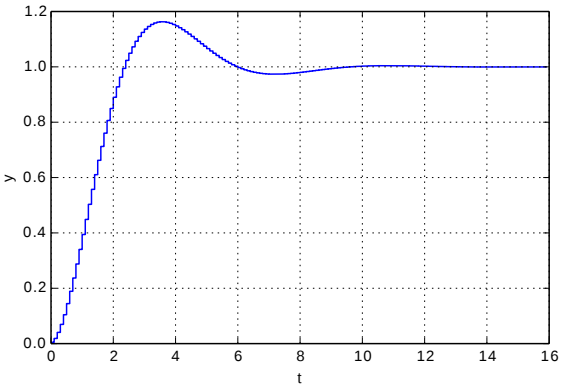


or alternatively

```
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 3.1: 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=np.arange(0,16,0.1)

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

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

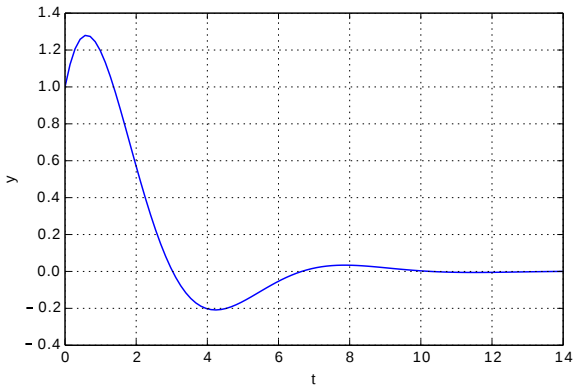


or alternatively

```
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]: gz=c2d(g,0.1)

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

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

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

Figure 3.2: 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')
```

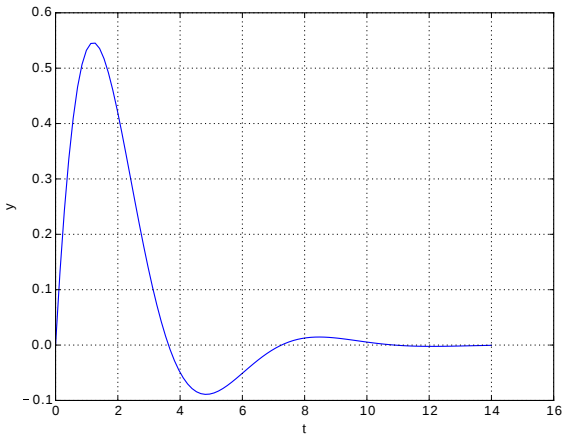


or alternatively

```
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 3.3: 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')
```

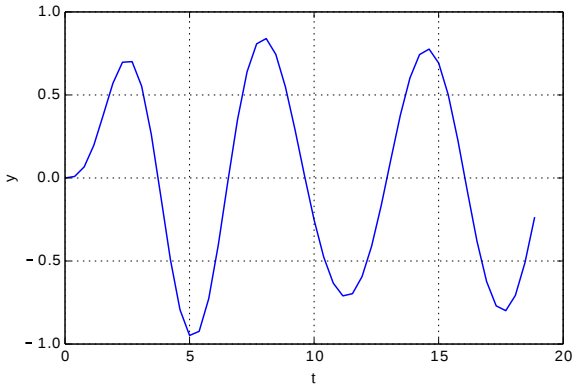


or alternatively

```
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 3.4: 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 3.5: Continuous time systems - Generic input

3.2 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 3.6, 3.7 and 3.8)

```
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);
```

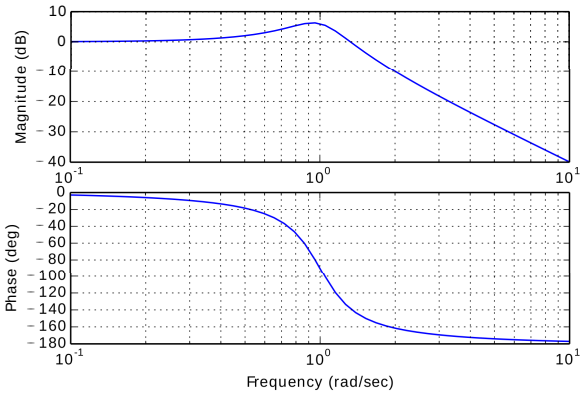


Figure 3.6: Bode plot

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,[1,2,1])
In [5]: nyquist(g), plt.grid()
```

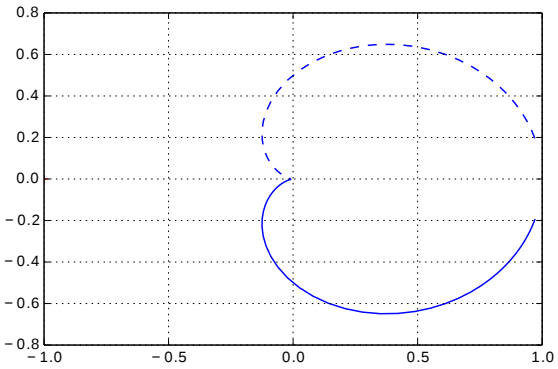


Figure 3.7: Nyquist plot

```
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)
```

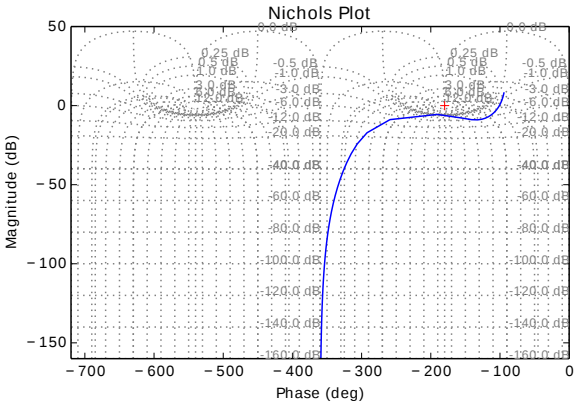


Figure 3.8: Nichols plot

```
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
```

3.3 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 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.
If in the jupyter shell you set the command **%matplotlib qt**, 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
```

```
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.])
```

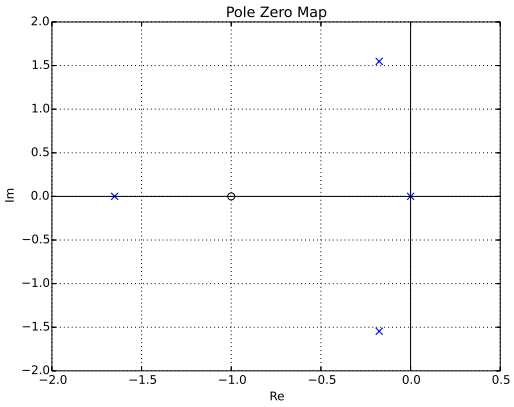


Figure 3.9: Poles and zeros

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

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

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

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);
```

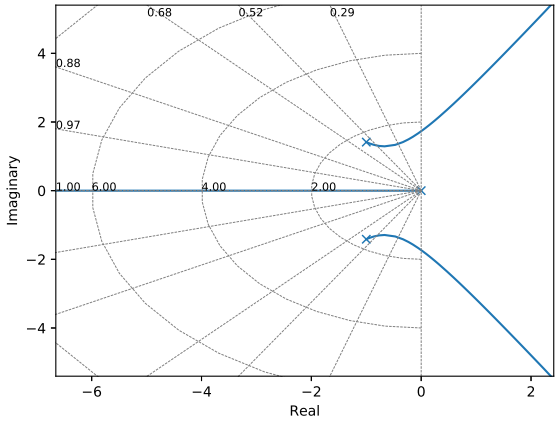


Figure 3.10: Root locus plot

Chapter 4

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 [6], [7], [8], [9], [10] and [11].

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.

4.1 Model of a DC motor (Lagrange method)

4.1.1 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).

4.1.2 Modules and constants

```
In [1]: from sympy import symbols, Matrix, pi
   ...: from sympy.physics.mechanics import *
   ...: import numpy as np
   ...:
   ...: # Modeling the system with Lagrange method
   ...:
   ...: # Signals
   ...: ph = dynamicsymbols('ph')     # motor angle
   ...: w  = dynamicsymbols('ph', 1)      # 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')
   ...:
```

4.1.3 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)
   ...:
```

4.1.4 Body and inertia of the load

```
In [3]: # Mechanics
   ...: Io = outer(Nr.x, Nr.x)
   ...:
   ...: InT = (J*Io, O)
   ...:
   ...: Last = RigidBody('Last', O, Nr, M, InT)
   ...: Last.potential_energy = 0
   ...:
```

4.1.5 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)]
```

4.1.6 Model

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

```
In [5]: # Lagrange model
   ...: L = Lagrangian(N, Last)      # Lagrange operator
   ...: LM = LagrangesMethod(L, [ph], forcelist = forces,
        frame = N)
   ...: LM.form_lagranges_equations()
   ...:
   ...: # symbolically linearize about arbitrary
        equilibrium
   ...: MM, linear_state_matrix, linear_input_matrix,
        inputs = LM.linearize(q_ind = [ph], qd_ind = [w])
   ...:
```

4.1.7 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]: A = MM.inv() * linear_state_matrix
   ...: B = MM.inv() * linear_input_matrix
   ...:
   ...: print(A)
   ...: print(B)
   ...:
Matrix([[0, 1], [0, -Dm/J]])
Matrix([[0], [kt/J]])
```

4.2 Model of a DC motor (Kane method)

4.2.1 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).

4.2.2 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
   ...:
```

4.2.3 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)
   ...:
```

4.2.4 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)
   ...:
```

4.2.5 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]
   ...:
```

4.2.6 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)]])
```

4.2.7 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
   ...: MM, linear_state_matrix, linear_input_matrix,
        inputs =
KM.linearize(new_method=True)
   ...:
   ...: # 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)
   ...: MM = MM.subs(eq_dict)
   ...:
   ...: # compute A and B matrices
   ...: A = MM.inv() * f_A_lin)
   ...: B = MM.inv() * f_B_lin)
```

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

4.3 Model of the inverted pendulum - Lagrange

The second example is represented by the classical inverted pendulum as shown in figure 4.1.

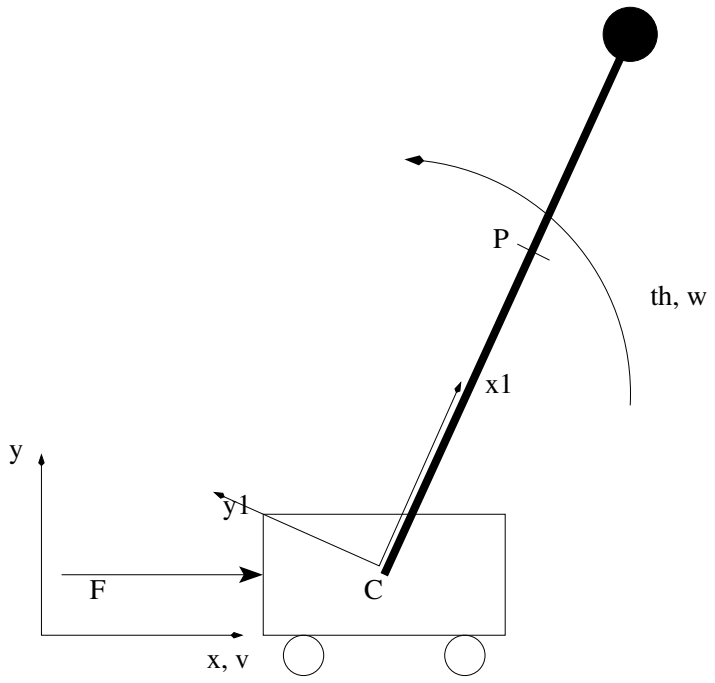


Figure 4.1: Inverted pendulum

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).

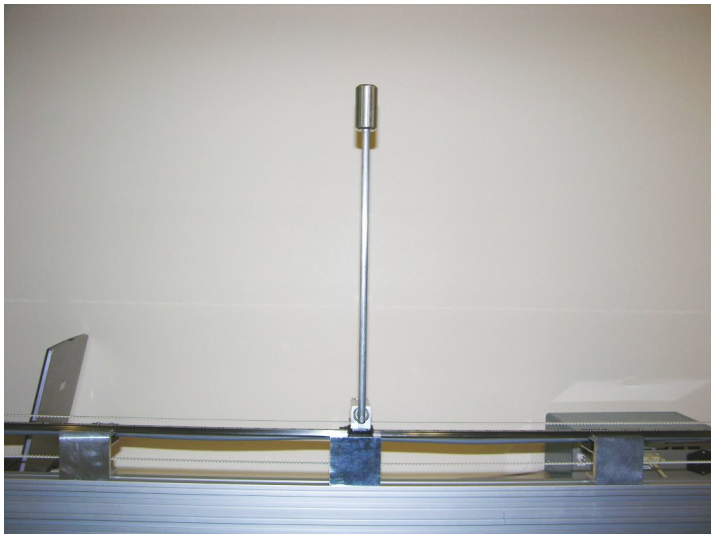


Figure 4.2: Inverted pendulum - Real plant

4.3.1 Modules and constants

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

4.3.2 Frames - Car and pendulum

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

4.3.3 Points, bodies, masses and inertias

```
In [3]: I = outer (Nf.z, Nf.z)
   ...: Inertia_tuple = (Ic*I, P)
   ...: Bp = RigidBody('Bp', P, A, m, Inertia_tuple)
   ...:
   ...: Bp.potential_energy = m*g*r*sin(th)
   ...: Car.potential_energy = 0
   ...:
```

4.3.4 Forces, frictions and gravity

```
In [4]: # Forces and torques
   ...: forces = [(C,F*Nf.x-d*v*Nf.x),(P,0*Nf.y)]
   ...:
```

4.3.5 Final model and linearized state-space matrices

```
In [5]: # Lagrange operator
   ...: L = Lagrangian(Nf, Car, Bp)
   ...:
   ...: # Lagrange model
   ...: LM = LagrangesMethod(L, [x, th], forcelist =
        forces, frame = Nf)
   ...: LM.form_lagranges_equations()
   ...:
   ...: # Equilibrium point
   ...: eq_pt = [0.0, pi/2,0.0,0.0]
   ...: eq_dict = dict(zip([x,th,v,w], eq_pt))
   ...:
   ...: # symbolically linearize about arbitrary
        equilibrium
   ...: MM, linear_state_matrix, linear_input_matrix,
        inputs = LM.linearize(q_ind = [x, th], qd_ind = [v,
        w])
   ...:
   ...: f_p_lin = linear_state_matrix.subs(eq_dict)
   ...: f_B_lin = linear_input_matrix.subs(eq_dict)
   ...:
   ...: MM = MM.subs(eq_dict)
   ...:
   ...: Atmp = MM.inv() * f_p_lin
   ...: Btmp = MM.inv() * f_B_lin
   ...:
```

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

```
In [7]: Btmp
Out[7]:
Matrix([
[                                                  0],
[                                                  0],
[(Ic + m*r**2)/(-m**2*r**2 + (Ic + m*r**2)*(M + m))],
[           m*r/(-m**2*r**2 + (Ic + m*r**2)*(M + m))]])
```

4.4 Model of the inverted pendulum - Kane

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).

4.4.1 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')
   ...:
```

4.4.2 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)
   ...:
```

4.4.3 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)
   ...:
```

4.4.4 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]
   ...:
```

4.4.5 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
   ...: MM, linear_state_matrix, linear_input_matrix,
        inputs =
KM.linearize(new_method=True)
   ...:
   ...: # 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)
   ...: MM = MM.subs(eq_dict)
   ...:
   ...: # compute A and B
   ...: A = MM.inv() * f_A_lin
   ...: B = MM.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

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

4.5 Model of the Ball-on-Wheel plant - Lagrange

A more complex plant is represented by the Ball-on-Wheel system of figure 4.3, where a ball must be maintened in the unstable equilibrium point on the top of a bike wheel.
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

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

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

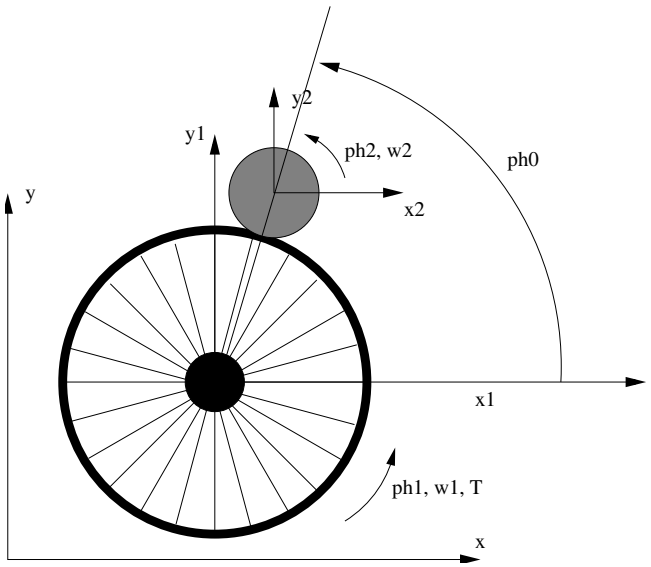


Figure 4.3: Ball-On-Wheel plant

4.5.1 Modules and constants

```
In [1]: from sympy import symbols, Matrix, pi, sin, cos
   ...: from sympy.physics.mechanics import *
   ...: # Lagrange Model of the system
   ...: # Index _b: angle between Wheel center and Ball CM
   ...: # Index _w: Wheel
   ...: # Index _roll: Ball
   ...:
   ...: # Dynamic symbols
   ...: phi_b, phi_w, phi_roll = dynamicsymbols('phi_b
       phi_w phi_roll')
   ...: w_b, w_w = dynamicsymbols('phi_b phi_w', 1)
   ...: w_roll = dynamicsymbols('w_roll')
   ...: T = dynamicsymbols('T')
   ...:
   ...: # Symbols
   ...: J_w, J_b = symbols('J_w J_b')
   ...: M_w, M_b = symbols('M_w M_b')
   ...: R_w, R_b = symbols('R_w R_b')
   ...: d_w      = symbols('d_w')
   ...: g        = symbols('g')
   ...: t        = symbols('t')
   ...:
```

4.5.2 Reference frames

```
In [2]: # Mechanical system
   ...: N = ReferenceFrame('N')
   ...:
   ...: O = Point('O')
   ...: O.set_vel(N,0)
   ...:
   ...: # Roll conditions
   ...: phi_roll = -(phi_w*R_w-phi_b*R_w)/R_b
   ...: w_roll = phi_roll.diff(t)
   ...:
   ...: # Rotating axes
   ...: # Ball rotation
   ...: # Wheel rotation
   ...: # Ball position
   ...: N_b = N.orientnew('N_b','Axis',[phi_b,N.y])
   ...: N_w = N.orientnew('N_w','Axis',[phi_w,N.y])
   ...: N_roll = N.orientnew('N_roll','Axis',[phi_roll,N.y
       ])
   ...:
   ...: N_w.set_ang_vel(N,w_w*N.y)
   ...: N_roll.set_ang_vel(N,w_roll*N.y)
   ...: N_b.set_ang_vel(N, w_b*N.y)
   ...:
```

4.5.3 Centers of mass of the ball

```
In [3]: # Ball Center of mass
   ...: CM2 = O.locatenew('CM2',(R_w+R_b)*N_b.z)
   ...: CM2.v2pt_theory(O,N,N_b)
   ...:
Out[3]: (R_b + R_w)*phi_b'*N_b.x
```

4.5.4 Masses and inertias

```
In [4]: # Inertia
   ...: Iy = outer(N.y,N.y)
   ...: In1T = (J_w*Iy, O)          # Wheel
   ...: In2T = (J_b*Iy, CM2)     # Ball
   ...:
   ...: # Bodies
   ...: B_w = RigidBody('B_w', O, N_w, M_w, In1T)
   ...: B_r = RigidBody('B_r', CM2, N_roll, M_b, In2T)
   ...:
   ...: B_r.potential_energy = (R_w+R_b)*M_b*g*sin(phi_b)
   ...: B_w.potential_energy = 0
   ...:
```

4.5.5 Forces and torques

```
In [5]: forces = [(N_roll, 0*N.y) , (N_w, T*N.y) ]
```

4.5.6 Lagrange's model and linearized state-space matrices

```
In [6]: # Lagrange operator
   ...: L = Lagrangian(N,  B_r,  B_w)
   ...:
   ...: # Lagrange model
   ...: LM = LagrangesMethod(L, [phi_b, phi_w], forcelist
       = forces,    frame = N)
   ...: LM.form_lagranges_equations()
   ...:
   ...: # Equilibrium point
   ...: eq_pt = [pi/2, 0, 0, 0]
   ...: eq_dict = dict(zip([phi_b, phi_w, w_b, w_w], eq_pt
       ))
   ...:
   ...: MM, linear_state_matrix, linear_input_matrix,
       inputs = LM.linearize(q_ind=[phi_b, phi_w], qd_ind
       = [w_b, w_w])
   ...:
   ...: f_p_lin = linear_state_matrix.subs(eq_dict)
   ...: f_B_lin = linear_input_matrix.subs(eq_dict)
   ...:
   ...: MM = MM.subs(eq_dict)
   ...:
   ...: Atmp = MM.inv() * f_p_lin
   ...: Btmp = MM.inv() * f_B_lin
   ...:
```

```
In [7]: Atmp
Out[7]:
Matrix([
[

     0,  0,  1,  0],
[

     0,  0,  0,  1],
[M_b*g*(R_b + R_w)*(J_b*R_w**2/R_b**2 + J_w)/(-J_b**2*R_w
     **4/R_b**4 + (J_b*R_w**2/R_b**2 + J_w)*(J_b*R_w**2/R_b
     **2 + M_b*(R_b + R_w)**2)),  0,  0,  0],
[       J_b*M_b*R_w**2*g*(R_b + R_w)/(R_b**2*(-J_b**2*R_w
     **4/R_b**4 + (J_b*R_w**2/R_b**2 + J_w)*(J_b*R_w**2/R_b
     **2 + M_b*(R_b + R_w)**2))),  0,  0,  0]])

In [8]: Btmp
Out[8]:
Matrix([
[

     0],
[

     0],
[                          J_b*R_w**2/(R_b**2*(-J_b**2*R_w**4/
     R_b**4 + (J_b*R_w**2/R_b**2 + J_w)*(J_b*R_w**2/R_b**2
     + M_b*(R_b + R_w)**2)))],
[(J_b*R_w**2/R_b**2 + M_b*(R_b + R_w)**2)/(-J_b**2*R_w**4/
     R_b**4 + (J_b*R_w**2/R_b**2 + J_w)*(J_b*R_w**2/R_b**2
     + M_b*(R_b + R_w)**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

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

4.6 Model of the Ball-on-Wheel plant - Kane

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

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

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

4.6.1 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')
   ...:
```

4.6.2 Reference frames

```
In [2]: N = ReferenceFrame('N')
   ...:
   ...: O = Point('O')
   ...: O.set_vel(N,0)
   ...:
   ...: ph0 = (R1*ph1+R2*ph2)/R1
   ...:
   ...: 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)
   ...:
```

4.6.3 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
```

4.6.4 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)
   ...:
```

4.6.5 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]
   ...:
```

4.6.6 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
   ...: MM, linear_state_matrix, linear_input_matrix,
   ...:     inputs =
KM.linearize(new_method=True)
   ...:
   ...: # 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)
   ...: MM = MM.subs(eq_dict)
   ...:
   ...: # compute A and B
   ...: A = MM.inv() * f_A_lin
   ...: B = MM.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)]])
```

Chapter 5

Control design

5.1 PI+Lead design example

5.1.1 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 supsisim.RCPblk import *

In [7]: from supsictrl.ctrl_utils import *

In [8]: from supsictrl.ctrl_repl import *

In [9]: g=tf([1],[1,6,5])

In [10]: bode(g,dB=True);

In [11]: legend(['G(s)'],prop={'size':10})
Out[11]:
(<matplotlib.axes.AxesSubplot at 0x7f85b5193550>,
 <matplotlib.legend.Legend at 0x7f85b47e6950>)

In [12]: wgc = 10          # Desired Bandwidth

In [13]: desiredPM = 60    # Desired Phase margin
```

Figure 5.1 shows the bode diagram of the plant.

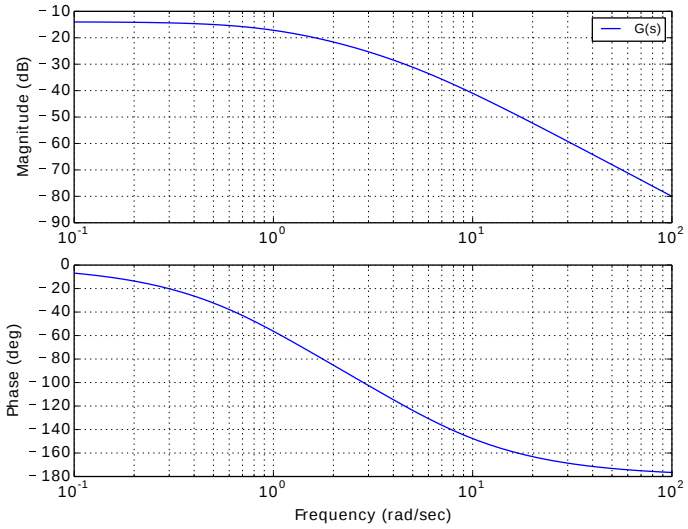


Figure 5.1: Bode diagram of the plant

5.1.2 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 [14]: # PI part

In [15]: Ti=0.15

In [16]: Gpi=tf([Ti,1],[Ti,0])

In [17]: print("PI part is:", Gpi)
PI part is:
0.15 s + 1
----------
  0.15 s


In [18]: figure()
Out[18]: <matplotlib.figure.Figure at 0x7f85b47eaa10>

In [19]: bode(g,dB=True,linestyle='dashed');

In [20]: bode(Gpi*g,dB=True);

In [21]: legend(['G(s)','Gpi(s)*G(s)'],prop={'size':10})
Out[21]:
(<matplotlib.axes.AxesSubplot at 0x7f85b4806250>,
 <matplotlib.legend.Legend at 0x7f85b4303850>)
```

Figure 5.2 shows the bode plot of the plant with and without the PI controller part.

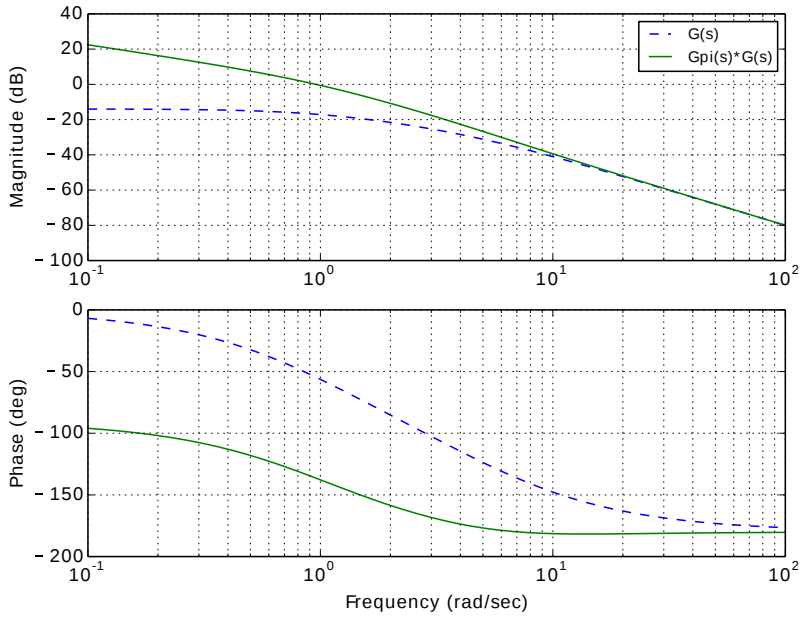


Figure 5.2: Bode diagram: G (dashed) and Gpi*G

5.1.3 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 [22]: mag,phase,omega = bode(Gpi*g,[wgc],Plot=False)

In [23]: ph = phase[0]

In [24]: if ph>=0:
    ...:     ph = phase[0]-360;
    ...:

In [2]: Phase = -180+desiredPM

In [26]: dPM = Phase-ph

In [27]: 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 [28]: # Lead part

In [29]: dPMrad = dPM/180*pi

In [30]: alfa = (1+sin(dPMrad))/(1-sin(dPMrad));

In [31]: print("Alpha is:", alfa)
Alpha is:  15.4073552425

In [32]: TD = 1/(sqrt(alfa)*wgc);

In [33]: Glead = tf([alfa*TD,1],[TD,1])

In [34]: print("Lead part is:", Glead)
Lead part is:
 0.3925 s + 1
--------------
0.02548 s + 1


In [35]: figure()
Out[35]: <matplotlib.figure.Figure at 0x7f85b43462d0>

In [36]: bode(g,dB=True,linestyle='dashed');

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

In [38]:
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 5.3 shows now the bode plot of the plant, the plant with the PI part and the plant with PI and Lead part

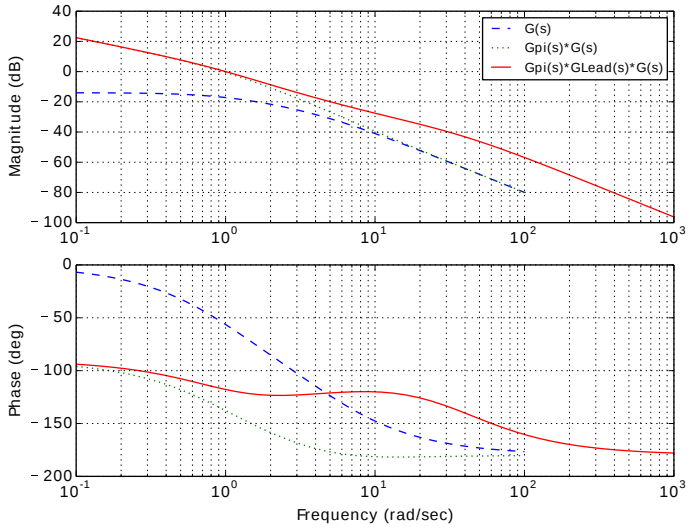


Figure 5.3: Bode diagram - G (dashed), Gpi*G (dotted) and Gpi*GLead*G

5.1.4 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}.

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

In [46]: bode(K*Gpi*Glead*g, dB=True);

In [47]:
legend(['G(s)','Gpi(s)*G(s)','Gpi(s)*GLead(s)*G(s)',
'K*Gpi(s)*GLead(s)*G(s)'],prop={'size':10})
Out[47]:
(<matplotlib.axes.AxesSubplot at 0x7f85b3a76690>,
 <matplotlib.legend.Legend at 0x7f85b33e6f90>)
```

In the figure 5.4 we see now that the gain plot has been translated up to get $0dB$ at the gain crossover frequency ω_{gc}.

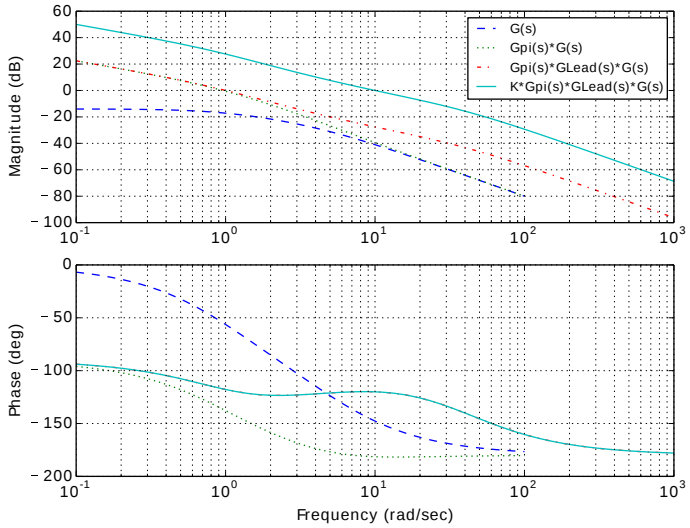


Figure 5.4: Bode diagram - G (dashed), Gpi*G (dotted), Gpi*GLead*G (dot-dashed) and K*Gpi*GLead*G

5.1.5 Simulation of the controlled system

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

```
In [48]: Contr = K*Gpi*Glead

In [49]: print("Full controller:", Contr)
Full controller:
1.402 s^2 + 12.92 s + 23.82
-----------------------------
   0.003821 s^2 + 0.15 s

In [50]: mag,phase,omega = bode(K*Gpi*Glead*g,[wgc],Plot=
     False)

In [51]: 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 [52]: gt=feedback(K*Gpi*Glead*g,1)

In [53]: t=linspace(0,1.5,300)

In [54]: y,t = step(gt,t)

In [55]: figure()
Out[55]: <matplotlib.figure.Figure at 0x7f85b3514290>

In [56]: plot(t,y), xlabel('t'), ylabel('y'), title('Step
     response of the
controlled plant')
Out[56]:
([<matplotlib.lines.Line2D at 0x7f85b34252d0>],

In [57]: grid()
```

The simulation of the controlled plant with a step input is shown in figure 5.5.

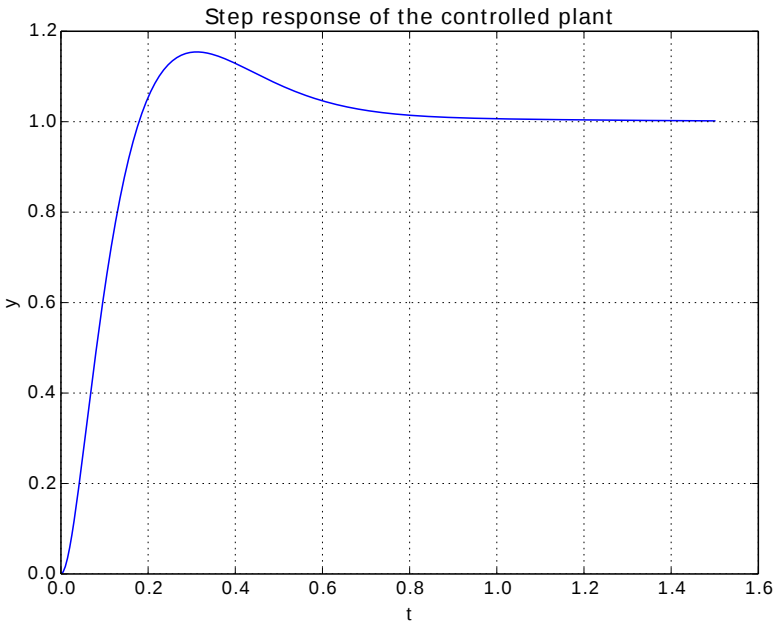


Figure 5.5: Step response of the controlled plant

5.2 Discrete-state feedback controller design

5.2.1 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".

5.2.2 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

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

5.2.3 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 supsisim.RCPblk import *
   ...: from supsictrl.ctrl_utils import *
   ...: from supsictrl.ctrl_repl import *
   ...:
```

5.2.4 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
   ...:
```

5.2.5 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
```

5.2.6 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 5.6)

```
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)
```

5.2.7 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

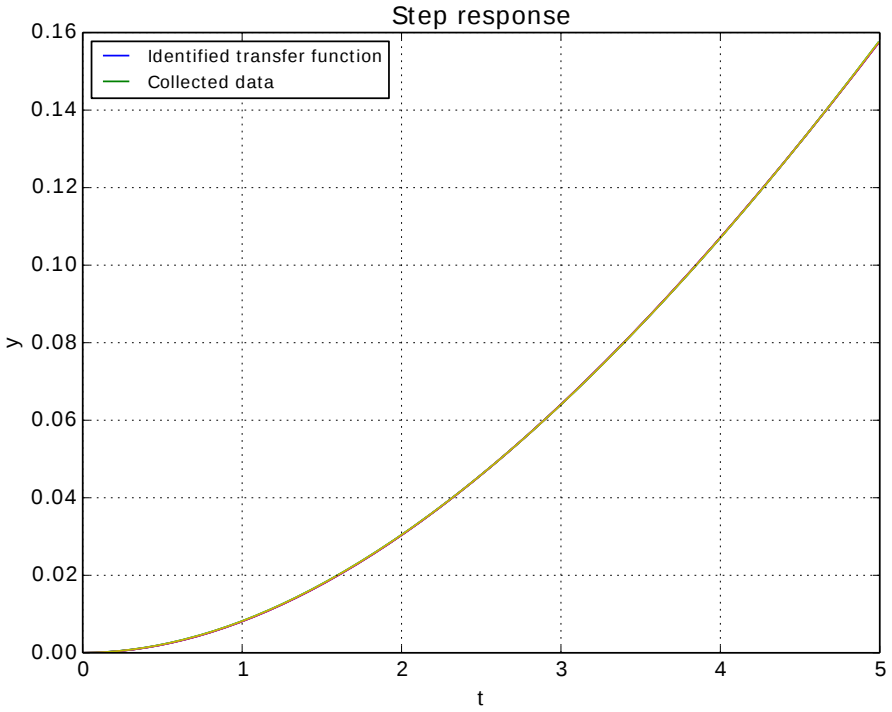


Figure 5.6: Step response and collected data

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

5.2.8 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)
```

5.2.9 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 pysimCoder 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])
```

5.2.10 Controller in compact form

The pysimCoder 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)
```

5.2.11 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])
```

5.2.12 Simulation of the controlled plant

The block diagram of the final controlled system is represented in figure 5.7.
It is not possible to simulate the resulting system in a Python shell because of:

- The controller is discrete and the plant is continuous. At present it is not possible to perform hybrid simulation usin the control package. In some cases we can substitute the plant with the discrete-time system and perform a discrete simulation. Hybrid simulation is possible using the pysimCoder application described in the next chapter.

- The block “CTRIN” has two inputs. The step function can only find the output from a single input.

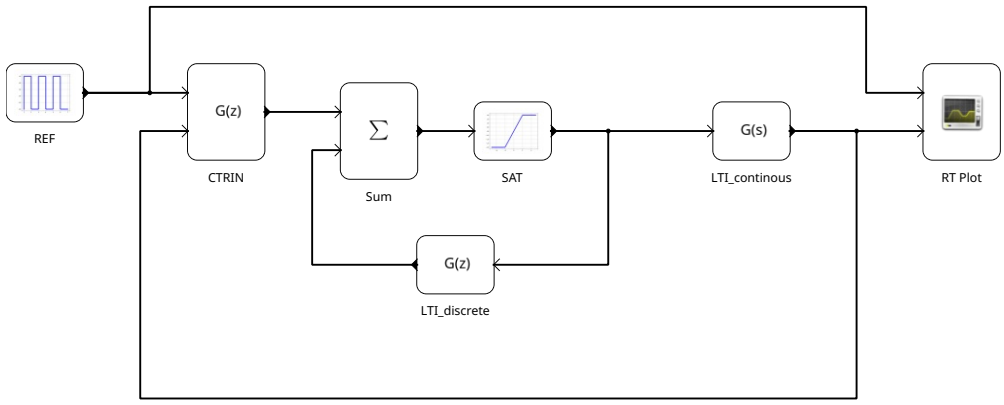


Figure 5.7: Block diagram of the controlled system

- The control toolbox can handle only linear system (and there is a saturation in the final system).

A possible method for the simulation of hybrid systems is described in chapter 6.

Chapter 6

Hybrid simulation and code generation

6.1 Basics

CACSD environments usually offer a graphical editor to perform the hybrid simulation (Matlab↔Simuli
Scioslab↔Scicos, Scilab↔xCos etc.).
The "pysimCoder.py" application should cover this task for the Python Control environment.
In the following we'll explain how it is possible, from the pysimCoder 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.

6.2 pysimCoder

6.2.1 The editor

The application "pysimCoder" is a block diagram editor to design schematics for simulation and code generation.
Starting points for the pysimCoder application were the PySimEd project ([12]) and the qtnodes-develop project ([13]).
PyEdit offers the most used blocks in control design. A little set of these blocks is shown in figure 6.1.

6.2.2 The first example

Using the editor we wont create the block diagram of figure 6.2.
We open a shell and we give the command

```
pysimCoder
```

The application opens 2 windows as shown in figure 6.3
The window on the left shows the library with the available blocks and on the right we have the diagram window. Now we can start to draw our block diagram.
From the library window we can choos the tab "input" and using "drag and drop" we can get the block "Step" and move it into the editor window. We can do the same operation with the "LTI continous" (from tab "linear") and the "Plot" (from tab "output") blocks.

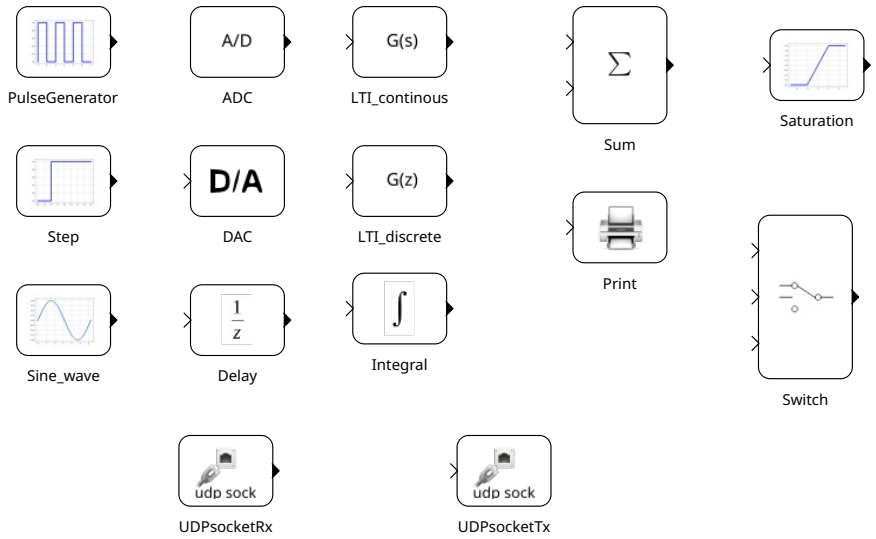


Figure 6.1: Some pysimCoder blocks for control design

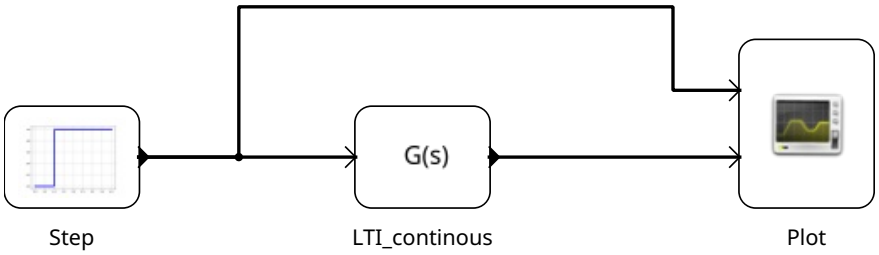


Figure 6.2: The first example

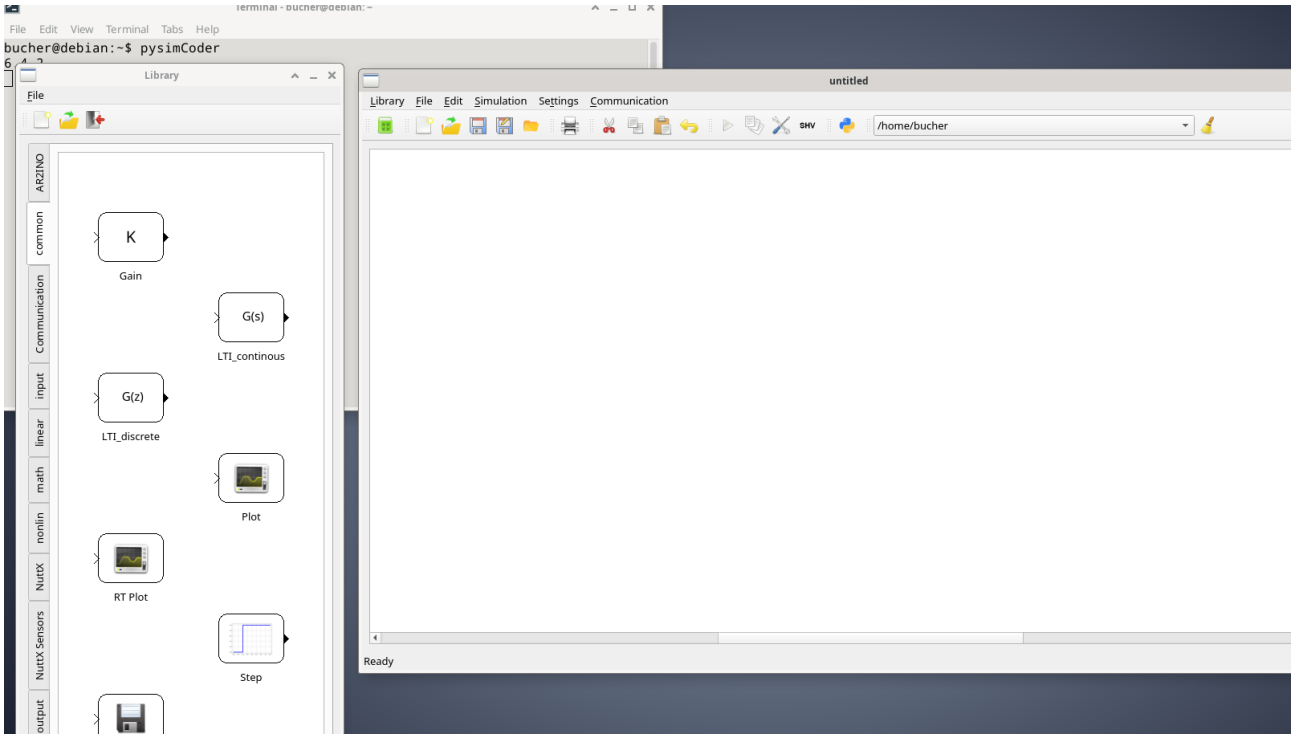


Figure 6.3: The pysimCoder environment

Now we should obtain the diagram shown in figure 6.4

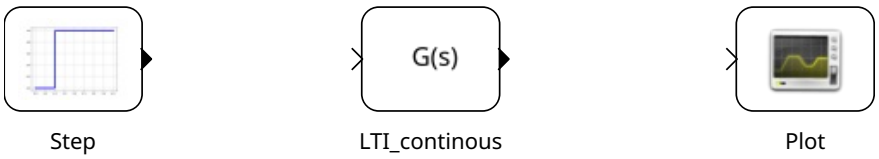


Figure 6.4: Result from the drag and drop operations

Before starting with the connection, we set some parameters to the blocks.

- Souble click with the mouse on the block "LTI continous". In the dialog windows set the System to **tf(1,[1,1])**
- Click the right mouse on the LTI continous block". In the new menu choose "Change Name" and rename it as **Plant**.
- Click the right mouse on the Plot block. In the new menu choose "Block I/Os" and set the number of inputs to **2**.

Figure 6.5 shows the new diagram.
Now we can proceed with the connections.

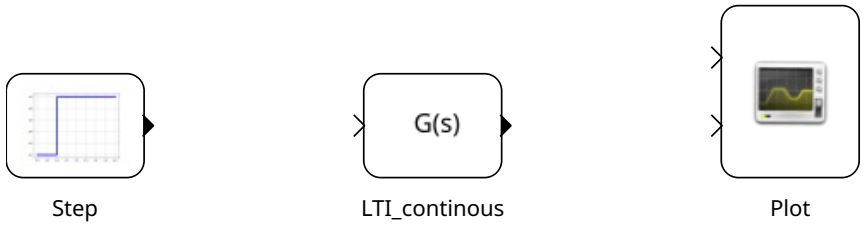


Figure 6.5: Result after parametrization

- Move the mouse on the output of the block "Step": the mouse pointer should become a "cross". Click and release the left mouse button.
- Now we can move the mouse to the input of the block "Plant": the mouse pointer should become a "cross". Click and release the left mouse button.
- Do the same operation from the output of the block "Plant" to the second input of the block "Plot"
- Now move to the node (the little circle) between the "Step" and the block "Plant": the mouse pointer should become a "cross". Click and release the left mouse button.
- move the mouse up, click, and continue to move left the mouse. Left of the position of the block "Plot", click and release again the left mouse button and then finish the connection on the first input of the block "Plot" (click and release the left mouse button)

You should obtain the diagram of figure 6.2
Now we are able to simulate the diagram.

- From the menu "Simulation" choose "Simulate" or click on the button "Simulate" on the toolbar (the button with the triangle).
- Double click with the mouse on the block "Plot" to get the graphical output of the simulation (see figure 6.6).

6.2.3 Some remarks

- the simulation result (Plot) **is available only after the simulation**. Please be sure to restart the simulation before opening the plot result. The simualtion creates a file with the name of the block in "/tmp" folder: this file is overwritten by every new simulation.
- For the simulation, the application creates and compile a C-executable. The sources are written in the folder "xxxxxx_gen", where "xxxxx" is the name of the diagram.

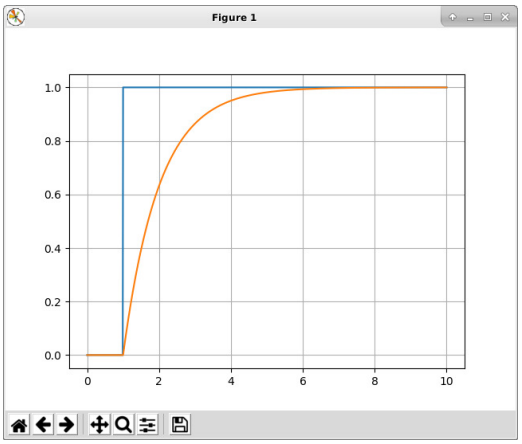


Figure 6.6: Result (plot) of the simulation

6.2.4 Defining new blocks

The user can define new blocks and integrate them into the pysimCoder application.
Two applications help the user todefine a new block.

- defBlocks
- xblk2Blk

The first application (defBlocks) is used to generate the ".xblk" file, with the default values of the block, by simply filling the different fields and adding the parameters on the bottom (seee figure 6.7).
The parameters in the window represent:

Library is the name of the "tab" window in the pysimCoder library

Name is the name of the block which appears under the block in the editor

Icon is the name of the icon file (located under "resources/blocks/Icons" without the extension (".svg")

Function is the name of the ".py" block which translates the block into the RCPBlk class objects (see code generation)

Inputs : number of the input ports

Outputs : number of the output ports

input settable is a flag which indicates if the number of input ports can be changed or not

output settable is a flag which indicates if the number of output ports can be changed or not

Bottom window is a grid which contains the parameters of the block (Label+default value).

Help contains an help about the block and the fields.

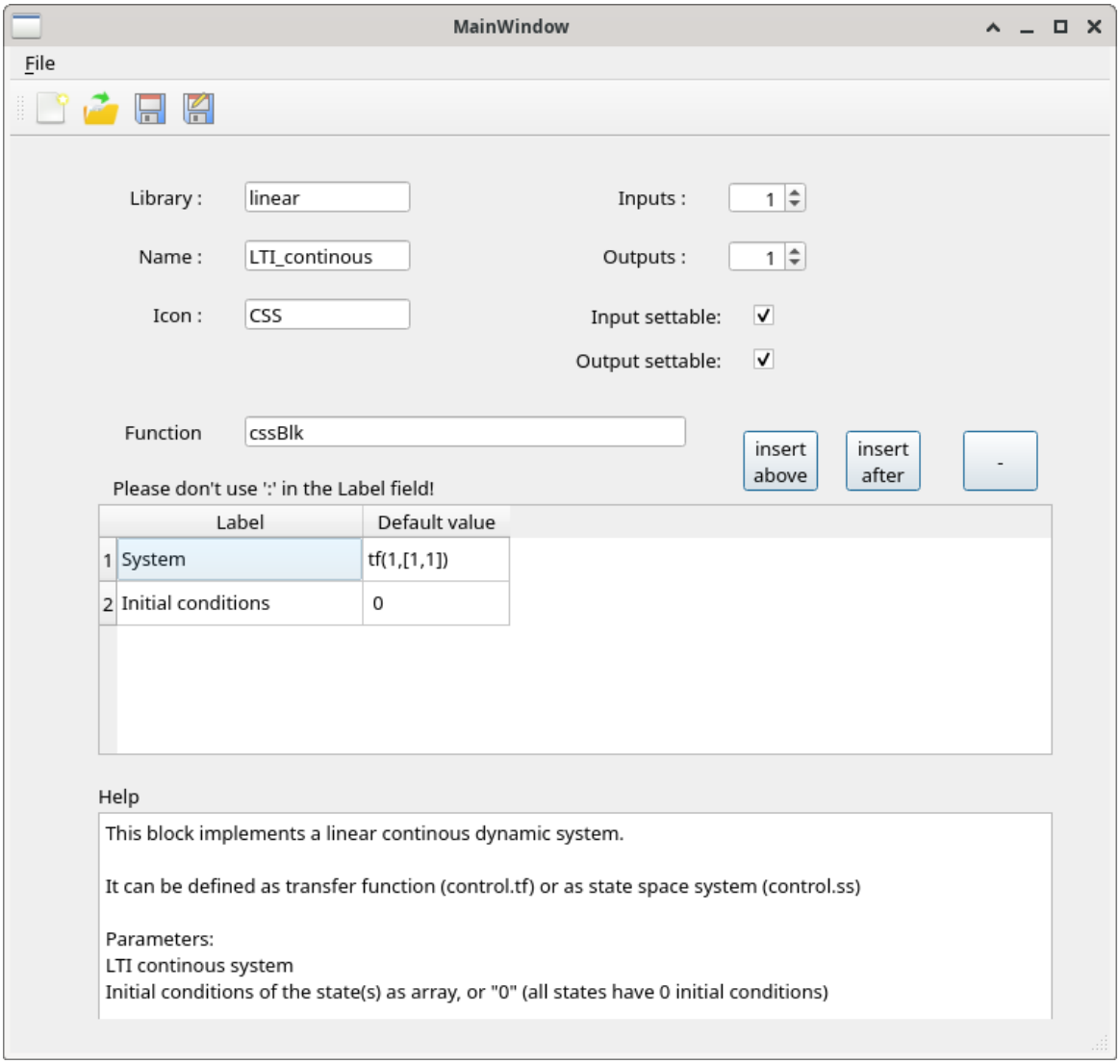


Figure 6.7: The “defBlocks” application

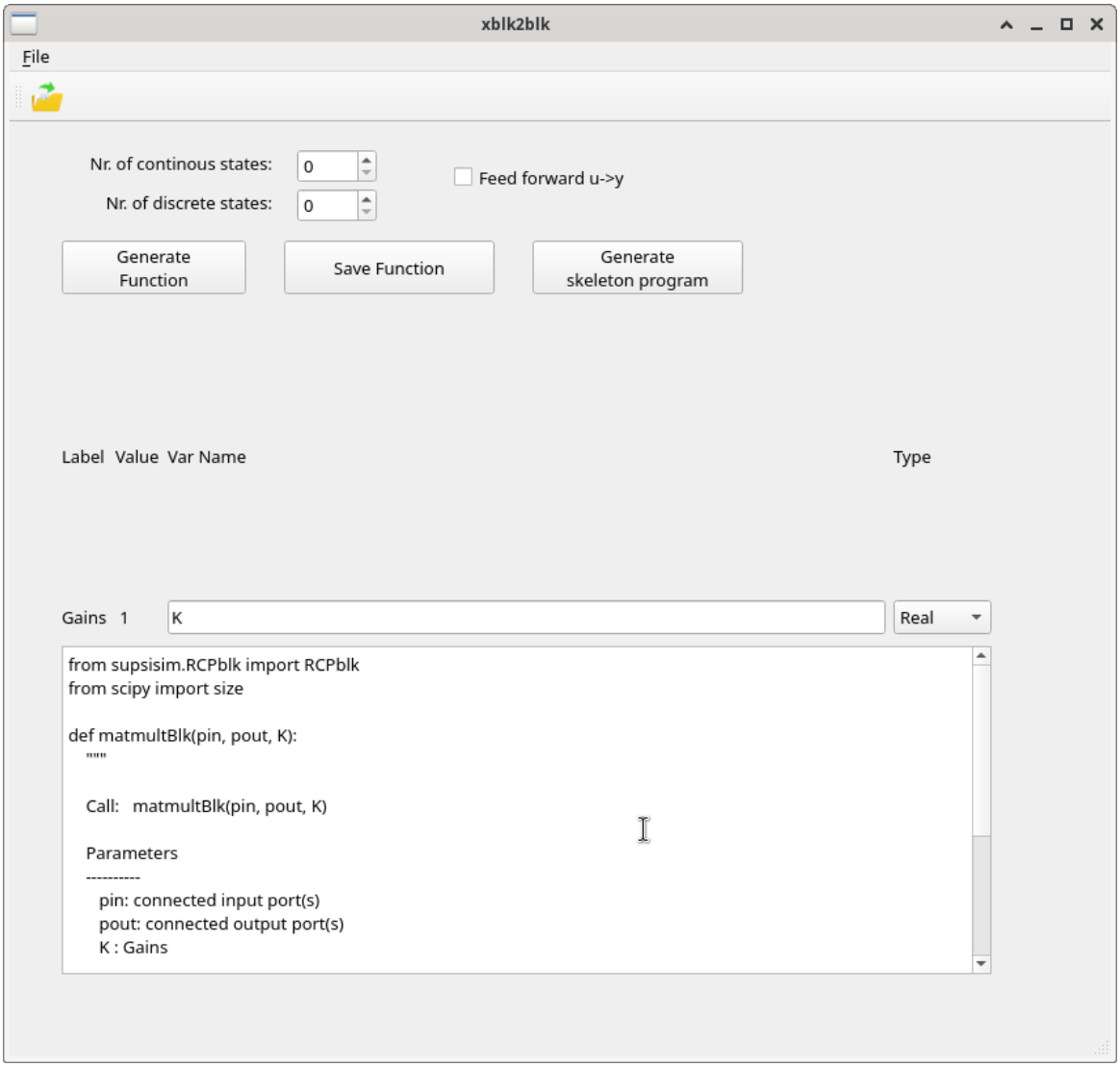


Figure 6.8: The "xblk2Blk" application

The “Save” or “Save as” operation generates the “.xblk” file. This file must be placed under “resources/blocks/blocks”
The second step is to call the application “xblk2Blk” (see figure 6.8).
After opening the “.xblk” file, it is possible to set a name and a type of each parameters of the block.
These informations are used to generate the “.py” which can be modified and saved and the ‘*.c* skeleton, which should be modified for the specific block tasks.
The “.py” file must be moved in the folder “resources/blocks/rcpBlk”, the “*.c* file must be edited and stored under ”CodeGen/XXX/devices“ where ”XXX“ represents the specific target.

6.3 Special libraries and blocks

6.3.1 The ”tab“ of the library

All the blocks are available in different ”tabs“ on the left of the library panel. The ”Common“ tab is a special library that can be personalized from the user whit his more used blocks.
The application ”configLibs“ allow to choose the library that must be shown in the library panel.

6.4 The editor window

6.4.1 The toolbar

The application offers set of operations in the toolbar as shown in the figure 6.9.

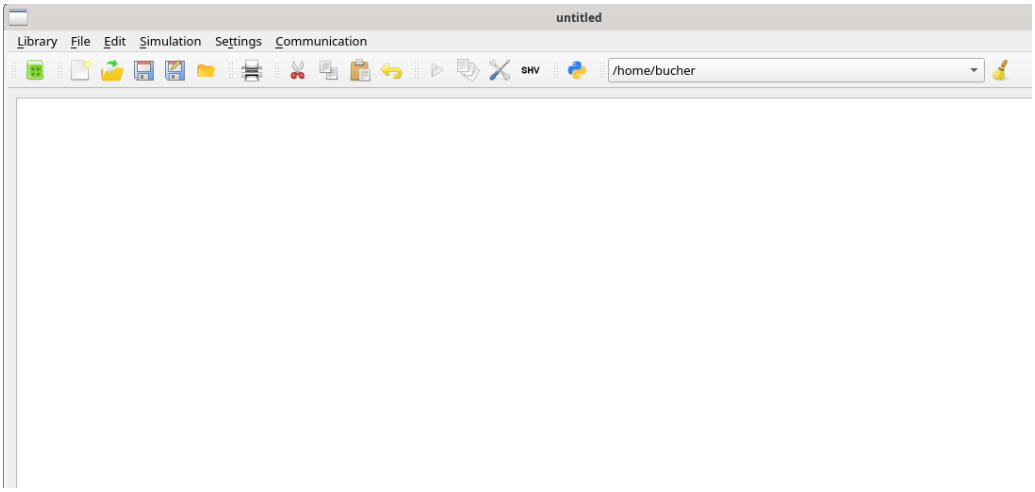


Figure 6.9: The pysimCoder application

6.4.2 Operations with the right mouse button

Depending on the position of the mouse, clicking and releasing the right mouse button leads to different behaviours.

6.4.3 Operations with the right mouse button on a block

Clicking with the right mouse button on a block opens a popup menu with the following commands:

Block I/Os to modify (if possible) the number of input and output ports of the block

Flip block Flip left/right the block

Change name Each block in the diagram must have **a unique name**

Block parameters to modify the parameters: this operation is available with a double click tool

Clone block to get a copy of the selected block

6.4.4 Operations with the right mouse button on multiple selected blocks

Clocking with the right mouse button on a blocck between multiple selected blocks allows to generate a subsystem.

6.4.5 Operations with the right mouse button on a connection

Moving the mouse on a connection, change the pointer to a pointing hand and by clicking with the right mouse button a popup menu is opened with the following commands:

Start connection Insert a node and start a new connection

Delete connection deletes the pointed connection

6.4.6 Behaviour of the left mouse button by drawing a connection

Clicking the left mouse button by drawing a connection starts a new segment of this connection.

6.4.7 Behaviour of the right mouse button by drawing a connection

Clicking the right mouse button by drawing a connection abort the connection.

6.5 Basic editor operations

6.5.1 Inserting a block

Get a block from a library and drag it into the main window.

6.5.2 Connecting blocks

It is possible to connect blocks with usual operations:

- Starting from an output port of a block and moving to an input port of another block. Mouse button can be released or not during this operation.
- Starting from an input port of a block and moving to a connection or an aoutput port of a block.
- Starting from a connection (after clicking with the right mouse button and choosing "Add connection"),

6.5.3 Deleting a block

- Move to a block and click with the right mouse button.
- Choose the submenu "delete"

It is also possible to select a block and use the "DEL" key.

Chapter 7

Simulation and Code generation

Each element of a block diagram is defined with three (or in special cases four) 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.

The translation of the block into the RCPblk class described in the RCPblk.py module.

If required, a particular dlg function to implement a special dialog box for the block parameters.

7.1 Interface functions

Each block is defined into a file with extension ".xblk", stored in the "resources/blocks/blocks" folder. The file is defined as a Python dictionary:

```
{
  "lib": "math",
  "name": "Sum",
  "ip": 2,
  "op": 1,
  "stin": 1,
  "stout": 0,
  "icon": "SUM",
  "params": "sumBlk|Gains: [1,1]",
  "help": "This block get the weighted sum of the input signals.\n\nIt can have more than 2 inputs.\n"
}
```

using the following fields:

"lib" the name of the tab for the block library (example "tab":"linear")

"name" the default name of the block

"ip" number of inputs

"op" number of outputs

"stin" flag which indicates if the number of inputs can be modified

“stout” flag which indicates if the number of outputs can be modified

“icon” the name of the “.svg” file with the icon of the block

“param” the parameters of the block

The first string in the param field is used as name of the Python function used to prepare the block to be translated into C-Code.
The block libraries are loaded after launching the pysimCoder application as shown in figure 7.1

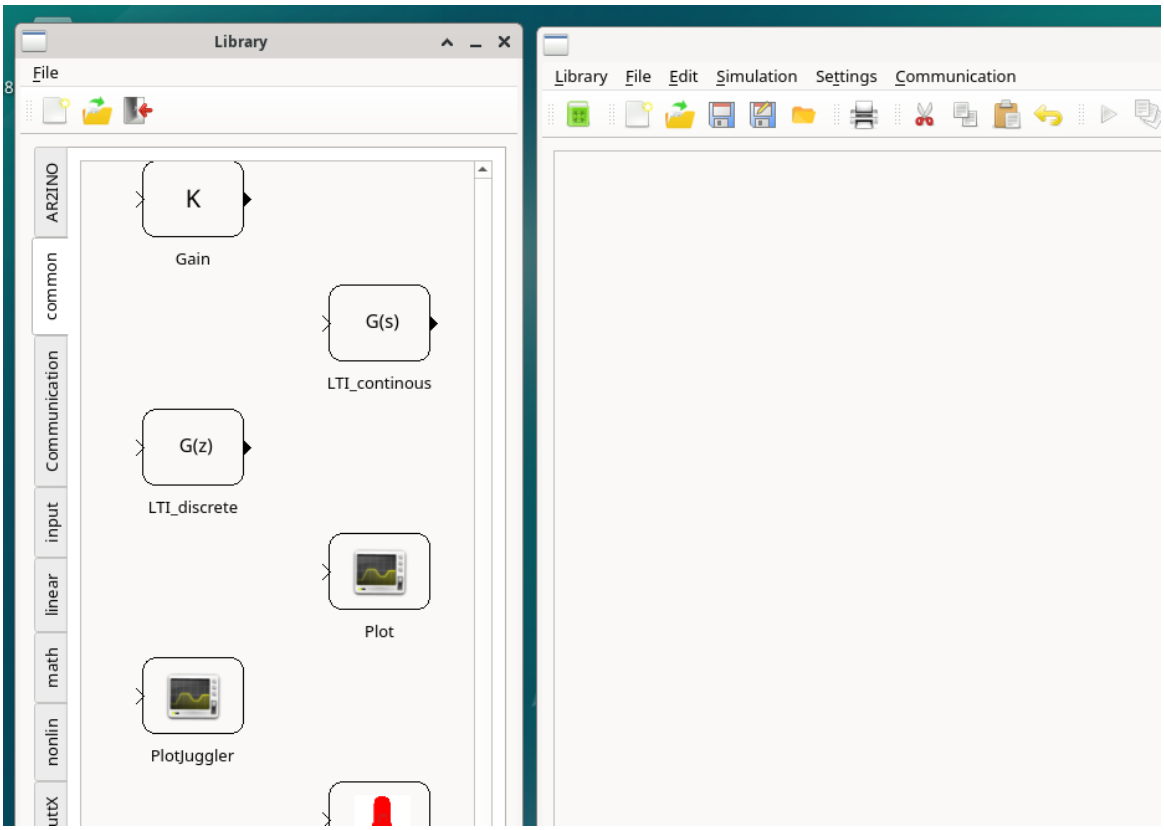


Figure 7.1: Window with the block libraries

Each block must be renamed with a unique name (popup menu “Change name”), and its parameters can be modified directly in the pysimCoder application with a double click.

7.2 The implementation functions

In a schematic, each block can be described with the functions (7.1) for continuous-time systems or (7.2) for discrete-time systems.

$$\begin{aligned} \mathbf{y} &= \mathbf{g}(\mathbf{x}, \mathbf{u}, t) \\ \dot{\mathbf{x}} &= \mathbf{f}(\mathbf{x}, \mathbf{u}, t) \end{aligned} \tag{7.1}$$

$$\begin{aligned} \mathbf{y}_k &= \mathbf{g}(\mathbf{x}_k, \mathbf{u}_k, k) \\ \mathbf{x}_{k+1} &= \mathbf{f}(\mathbf{x}_k, \mathbf{u}_k, k) \end{aligned} \tag{7.2}$$

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.

7.3 Translating the block into the RCPblk class

Before generating the C-Code, each block in the diagram must be translated into an element of the RCPblk class (see section 7.7 for more details). For each block, the corresponding function (the name is given by the 1. string in the parameters line) must exists and should be declared with the required parameters. This function is responsible to fill all the RCPblk fields.

7.4 Special dialog box for the block parameters

Usually, the graphic editor build a simple dialog box to enter the block parameters. In this dialoh, a "HELP" button open a MessageBox showing the block specific help text.
In special cases, it is possible to write a special function to enter the parameters.. In this case, the user should provide this function in the RCPDlg.py file. The name of this function is built using the first string of the parameter line, by subsistuting the las 3 letters "Blk" with "Dlg". This new function must receive as input:

- Numper of inputs
- Number of outputs
- The parameters line

This function returns a modified parameters line. Anexample is the "PlotDlg" function in the file "toolbox/supsisim/src/RCPGDlg.py".

7.5 Example

We can show with an example what happens with a block in the different phases from block to RCPblk class.
The "Pulse generator" input block is stored in the "PulseGenerator.xblk" file with the following infos

```
{
  "lib": "input",
  "name": "PulseGenerator",
  "ip": 0,
  "op": 1,
  "stin": 0,
```

```
  "stout": 0,
  "icon": "SQUARE",
  "params": "squareBlk|Amplitude: 1: double|Period: 4: double|Width: 2: double|Bias:
  "help": "This block implements a Pulse input signal\n\nParameters:\nAmplitude\nPeri
}
```

The block has no inputs, 1 ouput, the I/O are not modifiable (settable=0).
After a double click on the block, the "params" field is parsed and translated into the the dialog box shown in figure 7.2.

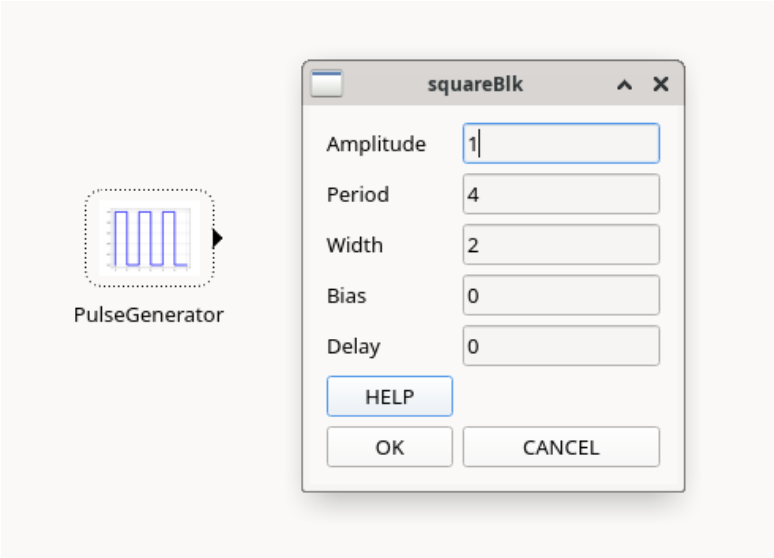


Figure 7.2: Dialog box for the Pulse generator block

By generating the element of the class RCPblk, the function "squareBlk" is called with the following parameters:

```
SQUARE = squareBlk(pout, Amp, Period, Width, Bias, Delay)
```

where

pout is the matrix with the id of the inputs (connections)

Amp is the signal amplitude

Period is the period of the signal

width is the duration where the signal has value "Amp-bias"

bias is an offset for the signal

delay represent the time wenn the signal start

The function translate the block into the following object of the RCPblk class

```
Function           : square
Input ports        : []
Output ports      : [2]
Nr. of states      : [0 0]
Relation u->y      : 0
Real parameters    : [ 1  4  2  0 0]
Integer parameters : []
```

7.6 The parameters for the code generation

Before clicking on the "code generation" tool on the toolbar, the user should fill some parameters in a dialog box (see figure 7.3).

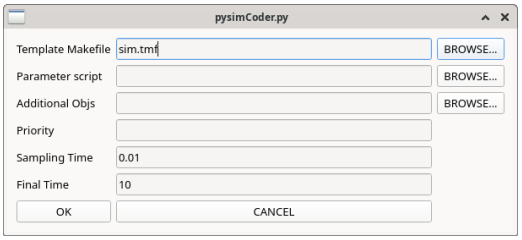


Figure 7.3: Dialog for code generation

In this dialog it is possible to choose the "template makefile" for simulation or real-time execution, the sampling time of the system and some additional libraries, required by special blocks.

7.7 Translating the diagram into elements of the RCPdlg class

After this first setup it is possible to translate the block diagram into a list of elements of the class **RCPblk** provided by the **suspisim** package. This class contains all the information required for the code generation.
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 in figure 7.4 is translated into the following code

```
from supsisim.RCPgen import *
from control import *

LTI_continous_0 = cssBlk([3],[1], tf(1,[1,1]), 0)
Print_1 = printBlk([2,1])
Step_2 = stepBlk([2], 1, 0, 1)
Sub_3 = sumBlk([2,1],[3], [1,-1])

# [...]

blks = [LTI_continous_0,Print_1,Step_2,Sub_3,]

fname = 'step'
os.chdir("./step_gen")
genCode(fname, 0.01, blks)
genMake(fname, 'sim.tmf', addObj = '')

# [...]

import os
os.system("make clean")
os.system("make")
os.chdir("..")
```

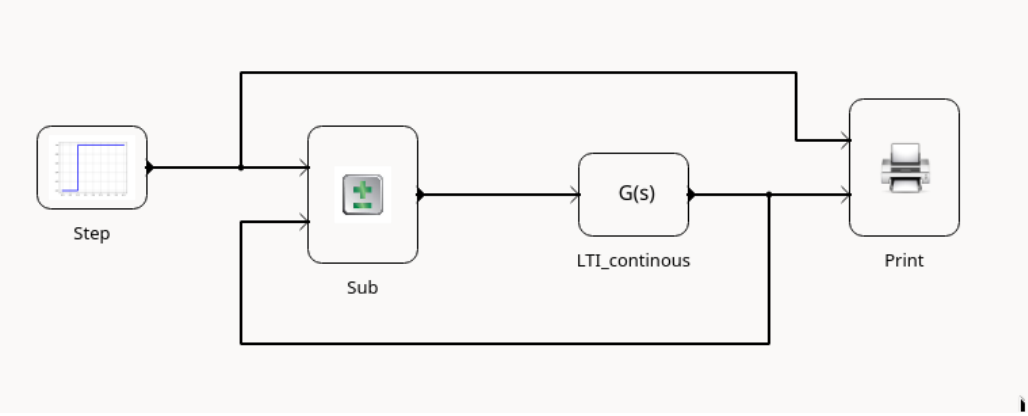


Figure 7.4: Simple block diagram

The block **CSS** 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 **PM** block has 2 inputs connected to node ① and ③, one output connected to node ② and performs a subtraction of the output from the input signals.

7.8 Translating the block list into C-code

7.8.1 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 7.4, the **PM** and the **PRINT** blocks require to know their inputs before update their outputs.

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

The next code paragrapg shows the right sequence of block execution, after the ordering algorithm. This is probably the only difficult task in code generation!

This is the list before ordering:

```
In [3]: for el in blks:
   ...:     print(el.name)
   ...:
LTI_continous_0
Print_1
Step_2
Sub_3
```

and this is the ordered list

```
In [4]: for el in ordered_list:
   ...:     print(el.name)
   ...:
LTI_continous_0
Step_2
Print_1
Sub_3
```

The complete list of the ordered block is consequently:

```
NrOfNodes = 3

ordered_list = detBlkSeq(NrOfNodes,blks)

for el in ordered_list:
    print(el)

Block Name          : LTI_continous_0
Function            : css
System path         : /LTI_continous
Input ports         : [3]
Output ports        : [1]
Input dimensions    : [1.]
Output dimensions   : [1.]
Nr. of states       : [1 0]
Relation u->y       : 0
Real parameters     : [[ 0. -1.  1.  1.  0.  0.]]
Names of real parameters : []
Integer parameters : [1 1 1 1 2 3 4 5]
Names of integer parameters : []
String Parameter    :

Block Name          : Step_2
Function            : step
System path         : /Step
Input ports         : []
Output ports        : [2]
Input dimensions    : []
Output dimensions   : [1.]
Nr. of states       : [0 0]
Relation u->y       : 0
Real parameters     : [1 0 1]
Names of real parameters : ['Step-Time', 'Initial-Value',
    'Final-Value']
Integer parameters : []
Names of integer parameters : []
String Parameter    :

Block Name          : Print_1
Function            : print
System path         : /Print
Input ports         : [2 1]
Output ports        : []
Input dimensions    : [1. 1.]
Output dimensions   : []
Nr. of states       : [0 0]
Relation u->y       : 1
Real parameters     : []
Names of real parameters : []
Integer parameters : []
Names of integer parameters : []
String Parameter    :

Block Name          : Sub_3
Function            : sum
System path         : /Sub
Input ports         : [2 1]
Output ports        : [3]
Input dimensions    : [1. 1.]
Output dimensions   : [1.]
Nr. of states       : [0 0]
Relation u->y       : 1
Real parameters     : [ 1 -1]
Names of real parameters : []
Integer parameters : []
Names of integer parameters : []
String Parameter    :
```

7.8.2 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

7.8.3 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 * dimIn;          /* Port signal dimension */
  int * dimOut;         /* Port signal dimension */
  int *nx;              /* Cont. and Discr states */
  void **u;             /* inputs */
  void **y;             /* outputs */
  double *realPar;      /* Real parameters */
  int realParNum;       /* Number of real parameters */
  int *intPar;          /* Int parameters */
  int intParNum;        /* Number of int parameters */
  char * str;           /* String */
  void * ptrPar;        /* Generic pointer */
  char **realParNames; /* Names of real parameters */
  char **intParNames;  /* Names of integer parameter */
} 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[]  = {&Node_3};
static void *outptr_0[] = {&Node_1};
static void *outptr_1[] = {&Node_2};
static void *inptr_2[]  = {&Node_2,&Node_1};
static void *inptr_3[]  = {&Node_2,&Node_1};
static void *outptr_3[] = {&Node_3};
...
  block_step[0].nin  = 1;
  block_step[0].nout = 1;
  block_step[0].nx   = nx_0;
  block_step[0].u    = inptr_0;
  block_step[0].y    = outptr_0;
...
```

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

```
/* Set initial outputs */

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

7.8.4 The termination function

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

7.8.5 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(CG_OUT, &block_step[0]);
  step(CG_OUT, &block_step[1]);
  print(CG_OUT, &block_step[2]);
  sum(CG_OUT, &block_step[3]);


  h = step_get_tsamp()/10;

  block_step[0].realPar[0] = h;
  for(i=0;i<10;i++){
    css(CG_OUT, &block_step[0]);
    css(CG_STUPD, &block_step[0]);
  }
  ...
```

7.9 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)();
}
```

Chapter 8

Example

8.1 The plant

One of the educational plants available at the SUPSI laboratory is the system shown in figure 8.1. This example is located in to the "pycontrol/Tests/ControlDesign/DisksAndSpring" folder,

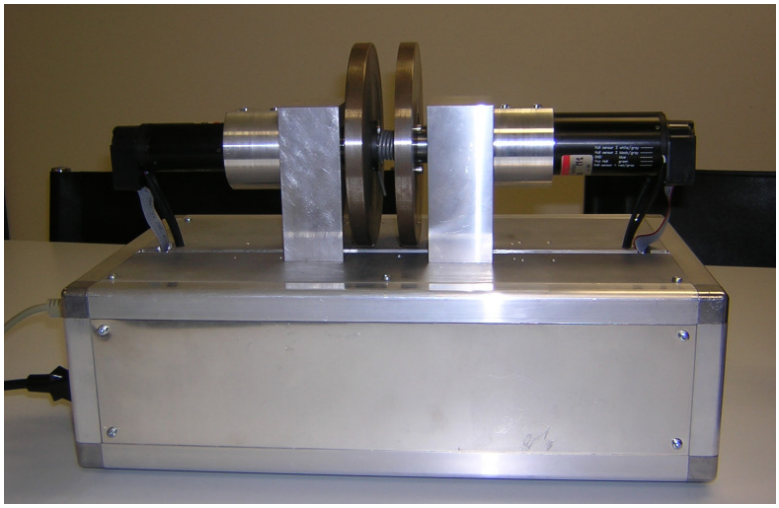


Figure 8.1: The disks and spring plant

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 [1]: # Real plants parameters
   ...: # Motor 1
   ...: jm1 = 0.0000085  # inertia [kg*m2]
   ...: kt1 = 0.0000382  # Torque constant
   ...: d1 = 0.0002953  # Damp
   ...:
   ...: # Last motor 1
   ...: rho_ac = 7900  # density [g/m3]
   ...: rv1 = 0.065     # radius [m]
   ...: hv1 = 0.01      # thickness [m]
   ...: mv1 = ((rho_ac*(rv1**2))*np.pi)*hv1   # mass [kg]
   ...: jv1 = (mv1*(rv1**2))/2   # inertia [kg*m2]
   ...: J1 = jv1+jm1   # total inertia[kg*m2]
   ...:
```

```
In [2]: # Motor 2
   ...: jm2 = 0.000003   # inertia [kg*m2]
   ...: kt2 = 0.0000205  # Torque constant
   ...: d2 = 0.0004001 # damp
   ...:
   ...: # Last motor 2
   ...: rho_ac = 7900   # density [kg/m3]
   ...: rv2 = 0.065     # radius [m]
   ...: hv2 = 0.01      # thickness [m]
   ...: mv2 = ((rho_ac*(rv2**2))*np.pi)*hv2   # mass [kg]
   ...: jv2 = (mv2*(rv2**2))/2                # inertia [kg*
        m2]
   ...: J2 = jv2+jm2                       # total inertia [km
        *m2]
   ...:
```

```
In [3]: # Spring
   ...: d = 0.0027836   # damp
   ...: c = 0.4797954   # spring factor
   ...:
```

```
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 "pysimCoder.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 pysimCoder 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 8.2).

$$sys\_in(s) = \frac{N(s)}{D_{new}(s)}$$

$$sys\_fbk(s) = 1 - \frac{D(s)}{D_{new}(s)}$$

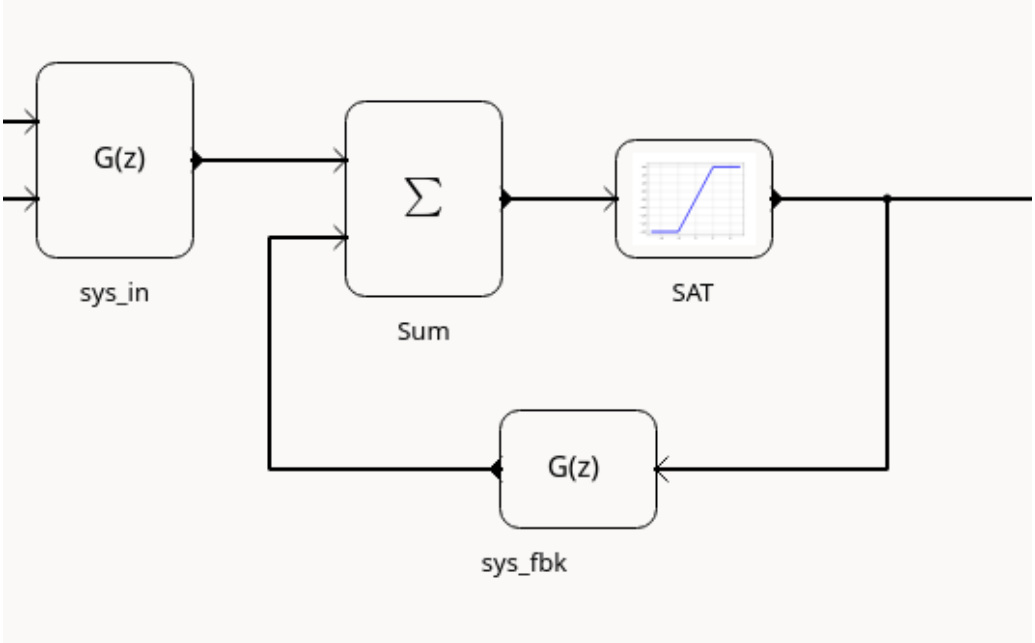


Figure 8.2: Anti windup

8.2 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')
```

8.3 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)
```

8.4 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])
```

8.5 Simulation

We can perform the simulation of the discrete-time controller with the continuous-time mathematic plant model using the block diagram of figure 8.3
This diagram is stored as “disks_sim.dgm” in the folder.
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.

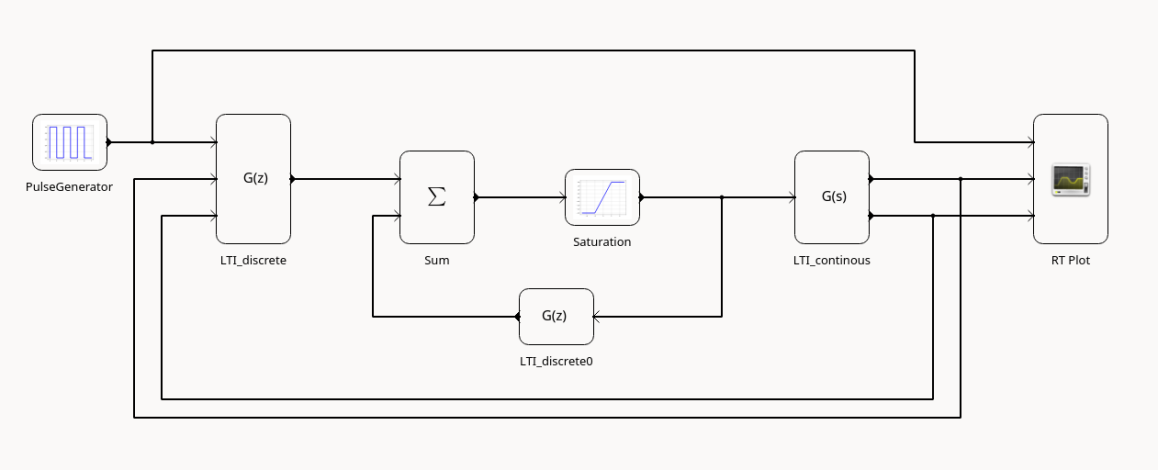


Figure 8.3: Block diagram for the simulation

Now we can launch the simulation with the command “Simulate” from the toolbar or from the menu.
A double click on the ‘block “Plot” show the result of the simulation (see figure 8.4)

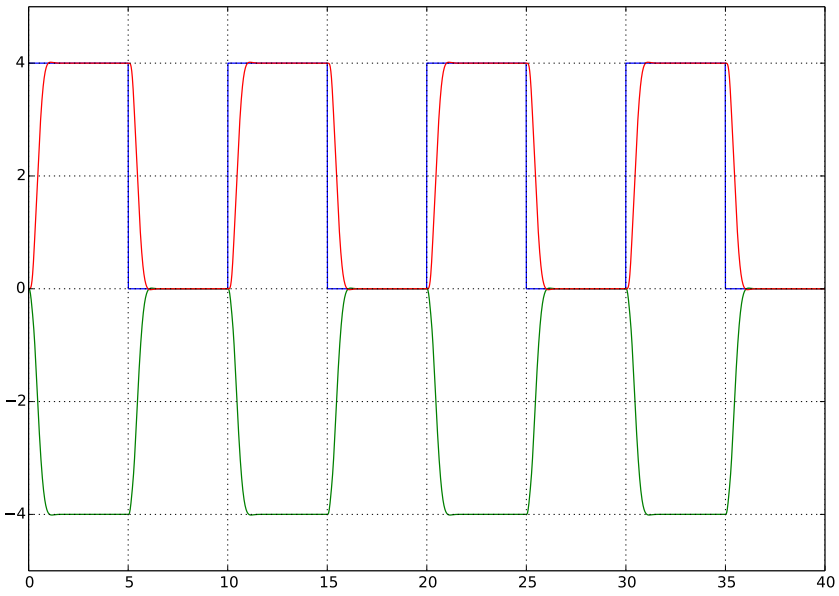


Figure 8.4: Simulation of the plant

8.6 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 8.5.

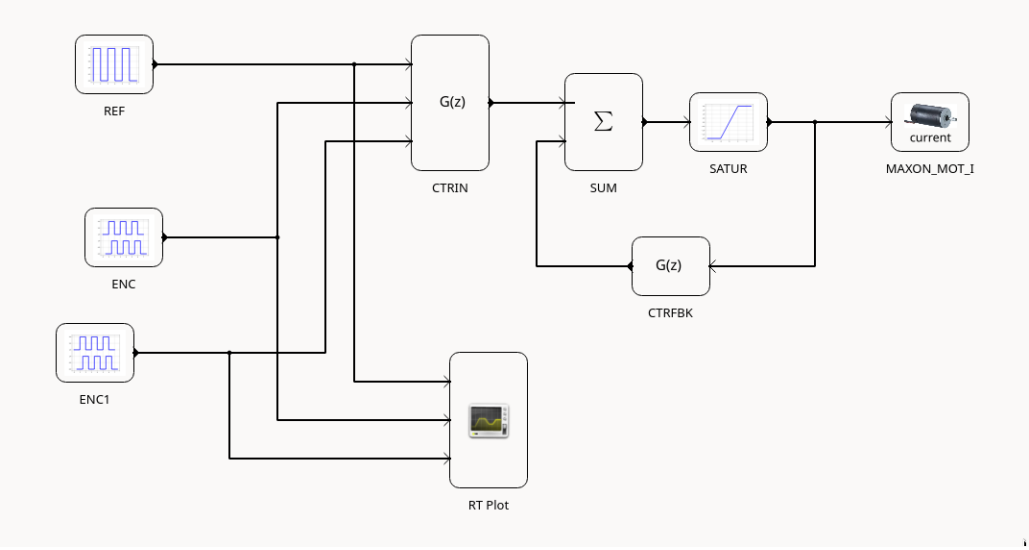


Figure 8.5: Block diagram for the RT implementation

The motor position can be plotted in python at the end of the execution (see figure 8.6).

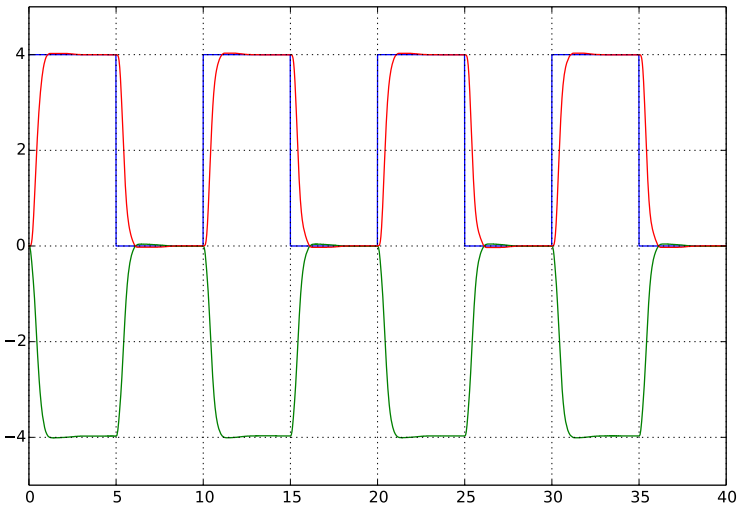


Figure 8.6: RT execution

Bibliography

[1] (2018) LinuxLabo. [Online]. Available: https://github.com/robertobucher/LinuxLabo

[2] (2018) pysimCoder. [Online]. Available: https://github.com/robertobucher/pysimCoder

[3] Electro-Mechanical Breadboard. [Online]. Available: https://www.robots5.com

[4] Robots5 LLC. [Online]. Available: https://www.youtube.com/@robots5

[5] Python Control Systems Toolbox. [Online]. Available: https://control-toolbox.readthedocs.io/en/latest

[6] Kane's Method in Physics/Mechanics. [Online]. Available: http://docs.sympy.org/0.7.5/modules/physics/mechanics/kane.html

[7] Kane's Method and Lagrange's Method (Docstrings). [Online]. Available: http://docs.sympy.org/latest/modules/physics/mechanics/api/kane_lagrange.html

[8] 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

[9] A Brief Synopsis of Kane's Method. [Online]. Available: www.cs.cmu.edu/ delucr/kane.doc

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

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

[12] PySimEd. [Online]. Available: http://www.kiwiki.info/index.php/PySimEd

[13] A port of qnodeseditor to PySide. [Online]. Available: https://github.com/cb109/qtnodes