State Feedback Controller Design using Python

Ing. Dipl. ETH Roberto Bucher

January 11, 2014

1 Install the files

The following files must be installed in order to run this example:

Python

Numpy from http://sourceforge.net/projects/numpy/files/

Scipy from http://sourceforge.net/projects/scipy/files/

Matplotlib from http://sourceforge.net/projects/matplotlib/

Slycot from https://github.com/avventi/Slycot

control-0.6c from http://sourceforge.net/projects/python-control/files/

yottalab.py from http://www.dti.supsi.ch/ bucher/python/index.html

RCPblk.py from http://www.dti.supsi.ch/ bucher/python/index.html

2 The plant

2.1 Identification

Figures 1 shows the real plant to be controlled and its driver.

Figure 1: Motor and driver
For state-feedback we need to have our plant in discrete state space form. We first model our plant as continuous time model, using a Laplace transform representation. Then we can transform the system in the discrete form. In order to find the continuous-time model we can for example identify our motor using a simple step response. We collect the response by a step of 500mA. Using the Newton laws we can find the mathematical model of our motor and driver:

\[ J \cdot \ddot{\varphi} = -D \cdot \dot{\varphi} + K_t \cdot I_a \]

where \( J \) is the motor+last inertia, \( D \) id the motor friction, \( K_t \) is the motor torque constant and \( I_a \) is the armature current.

The Laplace transform of the motor becomes

\[ G(s) = \frac{\frac{K_t}{J}}{s^2 + \frac{D}{J} \cdot s} = \frac{K}{s \cdot (s + \alpha)} \]

with \( K = K_t/J \) and \( \alpha = D/J \).

The time response of the motor can be found using the inverse Laplace transform of the motor which is

\[ y(t) = \mathcal{L}^{-1}[Y(s)] = \mathcal{L}^{-1}[\frac{500}{s} \cdot \frac{K}{s \cdot (s + \alpha)}] \]

\[ y(t) = \frac{500 \cdot K}{\alpha^2} + \frac{500 \cdot K}{\alpha} \cdot t + \frac{500 \cdot K}{\alpha^2} e^{-\alpha t} \]

First we must load all the python modules needed for identification and control:

```python
from yottalab import *
from scipy.optimize import leastsq
from scipy.signal import step2
import numpy as np
import scipy as sp
from control import *
from RCPblk import *
```

The values of \( K \) and \( \alpha \) can be found using the “leastsq” function in Python.

```python
# Motor response for least square identification
# Design script
fname='ID_MOT';
x = loadtxt(fname);

# Read and plot original data
t=x[:,0]
y=x[:,1]
plot(t,y)
title("Original data")
grid()
show()
```

Figure 2 shows the response of the system by a step of 500mA.

The value of \( K_t \) can be found in the motor data sheet: \( K_t = 126e6 \) - 6; then \( J = 0.001365039976 \) and \( D = 0.000351149780 \). From the data sheet we get the maximal motor current (1670mA) too. Now it is possible to proceed with the parameter identification of the motor. In particular we have to find the values of the of the total motor inertia \( J_m \) and the total motor friction \( D_m \).
The following script performs a simple least square identification of the two parameters.

```matlab
# Identify the motor transfer function
p0 = [1.0, 4.0]
Uo = 500;
y1 = y / Uo
pls = leastsq(residuals, p0, args=(y1, t))

# Motor parameters
Kt = 126.0e-6
Jm = Kt / plsq[0][0]
Dm = plsq[0][1] * Jm

# Transfer function
G(s) = \frac{\Phi(s)}{I_o(s)} = \frac{0.0965319}{0.6558342s + s^2}

# Compare step response and collected data
figure()
[Y, T] = step(g)
plot(T, Y)
hold
plot(t, y1)
title("Collected and simulated data")
grid()
```

The transfer function is

$$G(s) = \frac{\Phi(s)}{I_o(s)} = \frac{0.0965319}{0.6558342s + s^2}$$

We can compare the identified model and the collected data (see figure 3).
In order to proceed with a state feedback controller design, the plant is now represented in state-space form.

In order to proceed with a state feedback controller design, the plant is now represented in state-space form.

\[
\begin{align*}
a &= \begin{bmatrix} 0 & 1 \\ 0 & -D_m/J_m \end{bmatrix} \\
b &= \begin{bmatrix} 0 \\ 1 \end{bmatrix} \\
c &= \begin{bmatrix} K_t/J_m & 0 \end{bmatrix}; \\
d &= [0];
\end{align*}
\]

\[\text{sysc} = \text{ss}(a, b, c, d) \quad \# \text{Continuous state space form}\]

### 3 State feedback controller design

#### 3.1 Controller with precompensation

First we want to design our controller as discrete one. We must convert our model as discrete time system.

\[\text{sysd} = \text{bb_c2d}(\text{sysc}, T_s, 'zoh') \quad \# \text{Get discrete state space form}\]

We have a 2. order plant. A state-feedback with precompensation need the placement of two poles in the discrete domain.

The result of the pole placement are the gains $k$ for each state of the plant.

\[
\begin{align*}
wn &= 8 \\
xi &= \frac{\sqrt{2}}{2}
\end{align*}
\]
```python
c1_poly=[1,2*xi*wn,wn**2]
c1_poles=sp.roots(c1_poly); # Desired continous poles
c1_polesd=sp.exp(c1_poles*Ts)  # Desired discrete poles
k=place(sysd.A,sysd.B,c1_polesd)
```

Now we can calculate the precompensation gain

```python
k_pre=1/bb_dcgain(sysct)[0,0]
bb_step(k_pre*sysct)
```

Figure 4 shows the result of the simulation.

![Figure 4: Step response of the controlled plant with precompensation](image)

### 3.2 Controller with integrator

If we desire to eliminate the steady-state error we can add an integrator at the input of the controller. In this case we get an additional state for the controller and we need an additional pole for the pole placement. The matrices for the pole placement must be modified with the new state.

```python
# State feedback with integral part
wn=8
xi=sqrt(2)/2
c1_p1=[1,2*xi*wn,wn**2]
c1_p2=[1,wn]
c1_poly=sp.polymul(c1_p1,c1_p2)
c1_poles=sp.roots(c1_poly);  # Desired continous poles
c1_polesd=sp.exp(c1_poles*Ts)  # Desired discrete poles
sz1=sp.shape(sysd.A);
sz2=sp.shape(sysd.B);
```
4 Observer

4.1 Basics

In our plant it is not possible to measure both the states of the plant. We have to estimate them using an observer. There are two possibilities to implement an observer:

- A full order observer
- A reduced order observer

The choice of the observer is completely independent on the choice of the controller. In the first case we estimate both the states of the plants and we use them to perform out state feedback. In the second case, we use the output of the plant to extract one of the states. Only one state is estimated in this case.

The advantage of this solution is that we have less calculation in our observer, but we have the disadvantage that the measured state can be affected by the measure noise.

In both case the main idea is to have the estimator with a dynamic quicker as the controlled plant. This can be achieved by placing the continuous poles of the observer more links that the poles of the controlled plants (the poles chosen for the pole placement).

4.2 Full order observer

In this case we must provide two poles for the pole placement of the observer.

```matlab
# Full order observer
p_oc=10*(cl_poles[1:3])
p_od=sp.exp(p_oc*Ts);
f_obs=full_obs(sysd,p_od)
```

4.3 Reduced order observer

In this case we have to provide only a pole for the unknown state which must be estimated.

```matlab
#Reduced order observer
p_oc=-10*max(abs(cl_poles))
p_od=sp.exp(p_oc*Ts);
T=[0,1]
r_obs=red_obs(sysd,T,[p_od])
```
5 Implementation - The compact form

5.1 Basics on compact form realization

Figure 5 shows the plant (yellow) with the observer (cyan), the integrator (green) and the state feedback (red).

One of the problems in the representation of figure 5 is that the control signal $u(t)$ depends on the output of the plant and of the control signal $u(t)$ itself. This implementation leads to an “algebraic loop” in the building of the final controller. A solution is to create a dynamic system from the “Controller-Observer-Integrator” part where the control signal $u(t)$ appears only as output. The resulting block has two inputs (reference signal $r(t)$ and plant output $y(t)$) and one output (control signal $u(t)$).

Figure 6 shows the system with the controller in compact form.

Figure 5: State feedback with integrator and observer

Figure 6: State feedback in compact form
5.2 Controller with precompensation

For this case we need the following values:

- Observer \( obs \).
- State feedback gains in the vector \( K \)

\[
\text{contr}=\text{comp}\_\text{form}\( \text{sysd} , r\_\text{obs} , k \)
\]

5.3 Controller with integrator

For this case we need the following values:

- Observer \( obs \).
- State feedback gains \( K \) with the gains including \( Ke \).
- The vector \( q \) to choose which state is used for the integral error of the controller (not needed if the plant has a single output, default is \([1]\)).

\[
\text{contr\_I}=\text{comp}\_\text{form}\_\text{i}(\text{sys} , r\_\text{obs} , \text{kint} , \text{Ts})
\]

No we can built the controlled system which includes the controller (with integrator), the system and the output feedback.

The system can be simulated (step response).

\[
\text{sysct}=\text{sysctr}(\text{sys} , \text{contr}\_\text{I})
\]
\[
\text{dstep(\text{sysct} , \text{Tf}=4)}
\]

Figure 7 shows the response of the controlled motor.

5.4 Anti windup

We can generate the two statespace systems for the antiwindup with a provided function “set\_aw”

\[
[\text{gss\_in} , \text{gss\_out}]=\text{set\_aw}(\text{contr}\_\text{I} , [0 , 0])
\]

6 Hybrid simulation and Rapid Controller Prototyping

At present it is not possible to generate block diagram with python to perform hybrid simulation as for example in Simulink.

I implemented a very simple text editor with few blocks (the most used) in order to generate block diagram sequences under Linux OS.

From this textual description, a “C” program is generated. The generated code can be used to perform simulations or to be integrated into a microcontroller or a RT system.

The result is a quite simple Rypid Controller Prototyping environment implemented into a new library called “RCPblk.py”.

We can generate the code to simulate the complete system with feedback controller, integral part and anti-windup.
Figure 7: Response of the controlled system

```python
# square Input
sq1 = squareBlk(1, 1, 8, 4, 0, 0)

# plant (continuous)
plant = cssBlk(5, 6, sysc, [0, 0])

# anti-windup input
crin = dssBlk([1, 6], 2, gss_in, [0, 0])

# anti-windup feedback
crout = dssBlk([5, 3], gss_out, [0, 0])

# sum block
sum1 = sumBlk([2, 3], 4, [1, 1])

# Saturation (for anti-windup)
sat1 = saturBlk(4, 5, 500, -500)

# print results
prnt = printBlk([1, 6])

# generate the "C" code and the Makefile
fname = 'motorAW'
genCode(fname, 6, Ts, [sq1, plant, crin, crout, sum1, sat1, prnt])
genMake(fname)
```

Figure 8 shows the result of the simulation

A Block defined for RCP

```python
def stepBlk(pout, initTime, Val):
    """Create STEP block"
    # Step input signal
    # Call: stepBlk(pout, initTime, Val)
    # Parameters
    # pout: connected output port
    # initTime: step Time
    # Val: Value at step Time
```
```python
def sineBlk(pout, Amp, Freq, Phase, Bias, Delay):
    '''Create SINUS block
    Sine input signal
    Call: sineBlk(pout, Amp, Freq, Phase, Bias, Delay)
    Parameters
    pout: connected output port
    Amp: Signal Amplitude
    Freq: Signal Freq
    Phase: Signal Phase
    Bias: Signal Bias
    Delay: Signal Delay
    Returns
    _____blk: RCPblk
    '''

def squareBlk(pout, Amp, Period, Width, Bias, Delay):
    '''Create SQUARE block
    square input signal
    Call: squareBlk(pout, Amp, Period, Width, Bias, Delay)
    '''
```

Figure 8: Simulation of the full system
Parameters

pout: connected output port
Amp: Signal Amplitude
Period: Signal Period
Width: Signal Width
Bias: Signal Bias
Delay: Signal Delay

Returns

blk: RCPblk

---

def printBlk(pin):
    """ Create PRINT block
    """
    print output data
    Call: printBlk(pin)

    Parameters
    --------
    pin: connected input ports

    Returns
    -------
    blk: RCPblk

    ---

def sumBlk(pin, pout, Gains):
    """ Create SUM block
    """
    Sum input signal multiplied by gains
    Call: sumBlk(pin, pout, Gains)

    Parameters
    --------
    pin: connected_input_ports
    pout: connected_output_port
    Gains: input gains

    Returns
    -------
    blk: RCPblk

    ---

def saturBlk(pin, pout, satP, satN):
    """ Create SATURATION block
    """
    Saturation of the input signal
def dssBlk ( pin , pout , sys , X0 = [] ) :
    """ Create DSS block
    Discrete state space block
    Call : dssBlk ( pin , pout , sys , X0 )
    Parameters
    ________________________
    pin : connected input ports
    pout : connected output ports
    sys : Discrete system in SS form
    X0 : Initial conditions
    Returns
    __________
    blk : RCPblk
    """

def cssBlk ( pin , pout , sys , X0 = [] ) :
    """ Create CSS block
    Continuous state space block
    Call : cssBlk ( pin , pout , sys , X0 )
    Parameters
    ________________________
    pin : connected input ports
    pout : connected output ports
    sys : Discrete system in SS form
    X0 : Initial conditions
    Returns
    __________
    blk : RCPblk
    """

def matmultBlk ( pin , pout , Gains ) :
    """ Create MXMULT block
    Call : matmultBlk ( pin , pout , Gains )
Matrix multiplication of the input signals

Call: matmultBlk(pin, pout, Gains)

Parameters

---

- **pin**: connected input ports
- **pout**: connected output ports
- **Gains**: Matrix with gains

Returns

---

**blk**: RCPblk

```
def constBlk(pout, val):
    """ Create CONST block
    """
    Constant value as input
    Call: constBlk(pout, val)

Parameters

---

- **pout**: connected output ports
- **val**: Const Value

Returns

---

**blk**: RCPblk

```
def absBlk(pin, pout):
    """ Create ABS block
    """
    Absolute Value of the input signal
    Call: absBlk(pin, pout)

Parameters

---

- **pin**: connected input ports
- **pout**: connected output ports

Returns

---

**blk**: RCPblk

```
def prodBlk(pin, pout):
    """ Create PROD block
    """
    Multiply input signals

Parameters

---

- **pin**: connected input ports
- **pout**: connected output ports

Returns

---

**blk**: RCPblk

```
def prodBlk(pin, pout):
    """ Call: prodBlk(pin, pout) """

    Parameters
    ----------
    pin: connected input ports
    pout: connected output port

    Returns
    -------
    blk: RCPblk

    """

def extdataBlk(pout, length, fname):
    """ Create EXTDATA block
    Get input data from an external file
    Call: extdataBlk(pout, length, fname)
    """

    Parameters
    ----------
    pout: connected output port
    length: number of points in file
    fname: Filename with data

    Returns
    -------
    blk: RCPblk

    """

def epos_MotIBlk(pin, ID, irq, propGain, intGain):
    """ Create EPOS MOTI block
    Torque control of an maxon epos driver
    Call: epos_MotIBlk(pin, ID, irq)
    """

    Parameters
    ----------
    pin: connected input port
    ID: CAN node ID
    irq: CAN interrupt for receiving data

    Returns
    -------
    blk: RCPblk

    """

def epos_EncBlk(pout, ID, irq, res):
    """ Create EPOS ENC block
    Get encoder value from an maxon epos driver
    """

    Parameters
    ----------
    pout: connected output port
    ID: CAN node ID
    irq: CAN interrupt for receiving data
    res: Resolution

    Returns
    -------
    blk: RCPblk

    """
```python
def maxon_MotBlk(pin, ID, irq, propGain, intGain):
    """ Create MAXON MOT block
    Maxon driver for torque control
    """
    Call : maxon_MotBlk(pin, ID, irq)

    Parameters
    --------
    pin : connected input port
    ID : CAN node ID
    irq : CAN interrupt for receiving data

    Returns
    -------
    blk : RCPblk

```

```python
def maxon_EncBlk(pout, ID, irq, res):
    """ Create MAXON ENC block
    Maxon driver for encoder
    """
    Call : maxon_EncBlk(pout, ID, irq, res)

    Parameters
    --------
    pout : connected output port
    ID : CAN node ID
    irq : CAN interrupt for receiving data
    res : encoder resolution

    Returns
    -------
    blk : RCPblk

```

```python
def epos_EncBlk(pout, ID, irq, res):
    Parameters
    -----------
    pout : connected output port
    ID : CAN node ID
    irq : CAN interrupt for receiving data
    res : encoder resolution

    Returns
    -------
    blk : RCPblk

```
```