Introduction to the Python Control Package

Finn Aakre Haugen

18th July 2020

Contents

Preface

Welcome to this tutorial for the Python Control Package for analysis and design of dynamic systems in general and feedback control systems in particular. The package resembles the Control System Toolbox in MATLAB.

The package is developed at California Institute of Technology (Caltech), USA, by prof. Richard M. Murray and coworkers.

This tutorial is based on version 0.8.2 which was released on 17th April 2019.

This tutorial covers only some of the functions in the Python Control Package. However, these function are basic, and if you master these functions, you should be well prepared for using other functions in the package.

Most of the tutorial is about continuous-time models, i.e. transfer functions based on the Laplace transform and state space models based on differential equations. Discrete-time models are briefly covered in one chapter at the end of the tutorial. That coverage is brief because the basic functions for continuous-time models can be used also for discrete-time models, i.e. with the same syntax, however with the sampling time (period) as an extra input argument in the functions.

The programming environment used in this book is Spyder, which comes with the Anaconda distribution of Python.

The home page of this tutorial is on

http://techteach.no/python_control

On the home page, you can also find files used in the tutorial.

A simple introduction to Python and Spyder is provided by the free pdf book *Python Simply* available on http://techteach.no.

Finn Aakre Haugen, PhD
http://techteach.no/fh
finn@techteach.no
July 2020

Chapter 1

Introduction

1.1 Information about Python Control package on the Internet

The home page of the Python Control package is

https://pypi.org/project/control/

A complete list of the functions is available on

https://python-control.readthedocs.io/en/0.8.2/control.html

1.2 Installation of the Python Control package

You can install the package with the command

```
pip install control
```

executed e.g. at the Anaconda prompt (the Anaconda command window)[1].

1.3 Importing the Python Control package into Python

The following command (in Python) imports the Python Control package into Python:

```
import control
```

---

[1] In Windows: Start menu / Anaconda prompt.

1.4 Using arrays for numerical data

In Python, tuples, lists and arrays can be used to store numerical data. However, only arrays are practical for mathematical operations on the data, like addition and multiplication. Therefore, I use arrays as the numerical data type consistently in this book, even in cases where lists may be used.

To use arrays, you must import the numpy package. It has become a tradition to rename the numpy package as np. Thus, in the beginning of your programs, you should include the command

```
import numpy as np
```

Creation, manipulation and mathematical operation on arrays are described in detail e.g. in the book *Python Simply* (cf. the Preface).

Chapter 2

Transfer functions

This chapter is about Laplace transform based transfer functions, or simply s-transfer functions. Discrete time transfer functions, or z-transfer functions, are covered by Chapter 5.1.2.

2.1 How to create transfer functions

The control.tf() function is used to create transfer functions with the following syntax:

H = control.tf(num, den)

where H is the resulting transfer function (object). num (representing the numerator) and den (representing the denominator) are arrays where the elements are the coefficients of the s-polynomials in descending order from left to right.

Of course, you can use any other names than H, num, and den in your own programs.

To illustrate the syntax, assume that the transfer function is

$$H(s) = \frac{b_1 s + b_0}{a_1 s + a_0} \tag{2.1}$$

In this case,

num = np.array([b1, b0])

and

den = np.array([a1, a0])

where, of course, the values of b1, b0, a1 and a0 have been defined earlier.

Example 2.1 *Creating a transfer function*

In this example, we create the following transfer function:

$$H(s) = \frac{2}{5s+1} \tag{2.2}$$

The Python program 2.1 creates this transfer function. The code print('H(s) = ', H) is used to present the transfer function in the console (of Spyder).

http://techteach.no/python_control/programs/create_tf.py

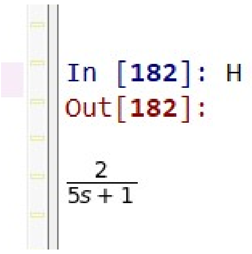
Listing 2.1: create_tf.py

```
import numpy as np
import control

# %% Creating the transfer function:
num = np.array([2])
den = np.array([5, 1])
H = control.tf(num, den)

# %% Displaying the transfer function:
print('H(s) =', H)
```

The result of the code above is shown as follows in the console:

```
H(s) =
  2
--------
5 s + 1
```

If you execute "H" (+ enter) in the Spyder console, the transfer function is more nicely displayed, see Figure 2.1.



```
In [182]: H
Out[182]:
```

$$\frac{2}{5s+1}$$

Figure 2.1: The transfer function nicely displayed with H (+ enter) executed in the console.

[End of Example 2.1]

2.2 Combinations of transfer functions

2.2.1 Series combination

Figure 2.2 illustrates a series combination of two transfer functions.

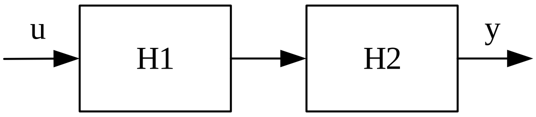


Figure 2.2: A series combination of two transfer functions, $H_1(s)$ and $H_2(s)$

The resulting transfer function is

$$\frac{y(s)}{u(s)} = H(s) = H_2(s)H_1(s) \tag{2.3}$$

If you are to calculate the combined transfer function manually using (2.3), the order of the factors in (2.3) is of no importance for SISO[1] transfer functions. But for MIMO[2] transfer functions, the order in (2.3) is crucial.

Whether SISO or MIMO, you create a series combination with the series() function of the Python Control package:

H = control.series(H1, H2)

Example 2.2 *Series combination of transfer functions*

Assume a series combination,

$$H(s) = H_1(s)H_2(s)$$

of the following two transfer functions:

$$H_1(s) = \frac{K_1}{s} \tag{2.4}$$

$$H_2(s) = \frac{K_2}{T_1 s + 1} \tag{2.5}$$

where $K_1 = 2$, $K_2 = 3$, and $T = 4$.

Manual calculation gives:

$$H(s) = \frac{K_1}{s} \cdot \frac{K_2}{Ts+1} = \frac{K_1 K_2}{Ts^2 + s} = \frac{6}{4s^2 + s}$$

Program 2.2 shows how the calculations can be done with the control.series() function.

---

[1] SISO = Single Input Single Output

[2] MIMO = Multiple Input Multiple Output

http://techteach.no/python_control/programs/series_tf.py

Listing 2.2: series_tf.py

```
# -*- coHding: utf-8 -*-
"""
Created on Sun Oct 27 01:50:36 2019

@author: Finn Haugen
"""

import numpy as np
import control

K1 = 2
K2 = 3
T = 4

num1 = np.array([K1])
den1 = np.array([1, 0])

num2 = np.array([K2])
den2 = np.array([T, 1])

H1 = control.tf(num1, den1)
H2 = control.tf(num2, den2)

H = control.series(H1, H2)

print(’H =’, H)
```

The result of the code above as shown in the console is:

```
H =
  6
------------
4 s^2 + s
```

[End of Example 2.2]

2.2.2 Parallel combination

Figure 2.3 illustrates a parallel combination of two transfer functions.

The resulting transfer function is

$$\frac{y(s)}{u(s)} = H(s) = H_2(s) + H_1(s) \tag{2.6}$$

The control.parallell() function calculates parallel combinations:

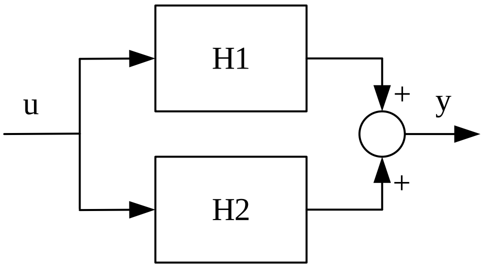


Figure 2.3: A parallel combination of two transfer functions, $H_1(s)$ and $H_2(s)$

H = control.parallel(H1, H2)

Example 2.3 *Parallel combination of transfer functions*

Given the transfer functions, $H_1(s)$ and $H_2(s)$, as in Example 2.2.

Manual calculation of their parallel combination gives[3]:

$$H(s) = \frac{2}{s} + \frac{3}{4s+1} = \frac{2(4s+1)+3s}{s(4s+1)} = \frac{11s+2}{4s^2+s}$$

Program 2.3 shows how the calculations can be done with the control.parallel() function.

http://techteach.no/python_control/programs/parallel_tf.py

Listing 2.3: parallel_tf.py

```
# -*- coHding: utf-8 -*-
"""
Created on Sun Oct 27 01:50:36 2019

@author: Finn Haugen
"""

import numpy as np
import control

K1 = 2
K2 = 3
T = 4

num1 = np.array([K1])
den1 = np.array([1, 0])

num2 = np.array([K2])
den2 = np.array([T, 1])
```

---

[3] For simplicity, I insert here the numbers directly instead of the symbolic parameters, but in general I recommend using symbolic parameters.

```
H1 = control.tf(num1, den1)
H2 = control.tf(num2, den2)

H = control.parallel(H1, H2)

print('H =', H)
```

The result of the code above as shown in the console is:

```
H =
  11 s + 2
------------
4 s^2 + s
```

[End of Example 2.3]

2.2.3 Feedback combination

Figure 2.4 illustrates a feedback combination of two transfer functions.

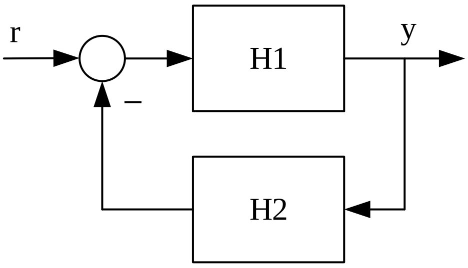


Figure 2.4: A feedback combination of two transfer functions, $H_1(s)$ and $H_2(s)$

The resulting transfer function, from r (reference) to y, which can be denoted the closed loop transfer function, can be calculated from the following expression defining y (for simplicity, I drop the argument s here):

$$y = H_1 \cdot (r - H_2 y) = H_1 r - H_1 H_2 y$$

which gives

$$y = \frac{H_1}{1 + H_1 H_2} r$$

Thus, the resulting transfer function is

$$\frac{y(s)}{r(s)} = H(s) = \frac{H_1(s)}{1 + H_1(s) H_2(s)} \tag{2.7}$$

The control.feedback() function calculates the resulting transfer function of a negative feedback combination:

$$\text{H} = \text{control.feedback(H1, H2, sign}=-1)$$

You may drop the argument sign $= -1$ if there is negative feedback since negative feedback is the default setting.

You must use sign $= 1$ if there is a positive feedback instead of a negative feedback in Figure 2.4.

In most cases – at least in feedback control systems – a negative feedback with $H_2(s) = 1$ in the feedback path is assumed. Then, $H_1()$ is the open loop transfer function, $L(s)$, and (2.7) becomes

$$\frac{y(s)}{r(s)} = H(s) = \frac{L(s)}{1 + L(s)} \tag{2.8}$$

In such cases, you can write

$$\text{H} = \text{control.feedback(L, 1)}$$

$L(s)$ may be the series combination (i.e. the product) of the controller, the process, the sensor, and the measurement filter:

$$L(s) = C(s) \cdot P(s) \cdot S(s) \cdot F(s) \tag{2.9}$$

Series combination using control.series() is described in Section 2.2.1.

Example 2.4 *The closed loop transfer function*

Given a negative feedback loop with the following open loop transfer function:

$$L(s) = \frac{2}{s} \tag{2.10}$$

Manual calculation of the closed loop transfer function with (2.8) gives

$$H(s) = \frac{L(s)}{1 + L(s)} = \frac{\frac{2}{s}}{1 + \frac{2}{s}} = \frac{2}{s + 2} \tag{2.11}$$

Program 2.4 shows how the calculations can be done with the control.feedback() function.

http://techteach.no/python_control/programs/feedback_tf.py

Listing 2.4: feedback_tf.py

```
# -*- coHding: utf-8 -*-
"""
Created on Sun Oct 27 01:50:36 2019

@author: Finn Haugen
```

```
"""

import numpy as np
import control

num = np.array([2])
den = np.array([1, 0])

L = control.tf(num, den)

H = control.feedback(L, 1)

print('H =', H)
```

The result:

```
H =
  2
---------
s + 2
```

[End of Example 2.4]

2.3 How to get the numerator and denominator of a transfer function

You can get (read) the numerator coefficients and denominator coefficients of a transfer function, say H, with the control.tfdata() function:

$$(\text{num\_list}, \text{den\_list}) = \text{control.tfdata(H)}$$

where num_list and den_list are *lists* (not arrays) containing the coefficients.

To convert the lists to arrays, you can use the np.array() function:

$$\text{num\_array} = \text{np.array(num\_list)}$$

and

$$\text{den\_array} = \text{np.array(den\_list)}$$

Example 2.5 *Getting the numerator and denominator of a transfer function*

See Program 2.5.

http://techteach.no/python_control/programs/get_tf_num_den.py

Listing 2.5: get_tf_num_den.py

```
import numpy as np
import control

# %% Creating a transfer function:
num = np.array([2])
den = np.array([5, 1])
H = control.tf(num, den)

# %% Getting the num and den coeffs as lists and then as arrays:
(num_list, den_list) = control.tfdata(H)
num_array = np.array(num_list)
den_array = np.array(den_list)

# %% Displaying the num and den arrays:
print('num_array =', num_array)
print('den_array =', den_array)
```

The result:

```
num_array = [[[2]]]
den_array = [[[5 1]]]
```

To "get rid of" the two inner pairs of square brackets, i.e. to reduce the dimensions of the arrays:

```
num_array = num_array[0,0,:]
den_array = den_array[0,0,:]
```

producing:

```
[2]
[5 1]
```

[End of Example 2.5]

2.4 Simulation with transfer functions

The function control.forced_response() is a function for simulation with transfer function and state space models. Here, we focus on simulation with transfer functions.

control.forced_response() can simulated with any user-defined input signal. Some alternative simulation functions assuming special input signals are:

- control.step_response()
- control.impulse_response()

- control.initial_response()

control.forced_response() may be used in any of these cases. Therefore, I limit the presentation in this document to the control.forced_response() function.

The syntax of control.forced_response() is:

(t, y, x) = control.forced_response(sys, t, u, X0)

where:

- Input arguments:
  - sys is the system to be simulated – a transfer function or a state space model.
  - t is the user-defined array of points of simulation time.
  - u is the user-defined array of values of the input signal of same length at the simulation time array.
  - X0 is the initial state. For SISO transfer functions, you can set X0 = 0.
- Output (return) arguments:
  - t is the returned array of time – the same as the input argument.
  - y is the returned array of output values.
  - x is the returned array of state values. For transfer functions, you are probably not interested in x, only in y.

To plot the simulated output (y above), and maybe the input (u above), you can use the plotting function in the matplotlib.pyplot module which requires import of this module. The common way to import the module is:

import matplotlib.pyplot as plt

Example 2.6 *Simulation with a transfer function*

We will simulate the response of the transfer function

$$\frac{y(s)}{u(s)} = \frac{2}{5s+1}$$

with the following conditions:

- Input u is a step of amplitude 4, with step time $t = 0$.
- Simulation start time is t0 = 0 sec.

- Simulation stop time is t1 = 20 sec.
- Simulation time step, or sampling time, is dt = 0.01 s.
- Initial state is 0.

Program 2.6 implements this simulation.

http://techteach.no/python_control/programs/sim_tf.py

Listing 2.6: sim_tf.py

```
# %% Import:
import numpy as np
import control
import matplotlib.pyplot as plt

# %% Creating model:
num = np.array([2])
den = np.array([5, 1])
H = control.tf(num, den)

# %% Defining signals:
t0 = 0
t1 = 20
dt = 0.01
nt = int(t1/dt) + 1 # Number of points of sim time
t = np.linspace(t0, t1, nt)
u = 2*np.ones(nt)

# %% Simulation:
(t, y, x) = control.forced_response(H, t, u, X0=0)

# %% Plotting:
plt.close('all')
fig_width_cm = 24
fig_height_cm = 18
plt.figure(1, figsize=(fig_width_cm/2.54, fig_height_cm/2.54))

plt.subplot(2, 1, 1)
plt.plot(t, y, 'blue')
#plt.xlabel('t [s]')
plt.grid()
plt.legend(labels=('y',))

plt.subplot(2, 1, 2)
plt.plot(t, u, 'green')
plt.xlabel('t [s]')
plt.grid()
plt.legend(labels=('u',))

plt.savefig('sim_tf.pdf')
```

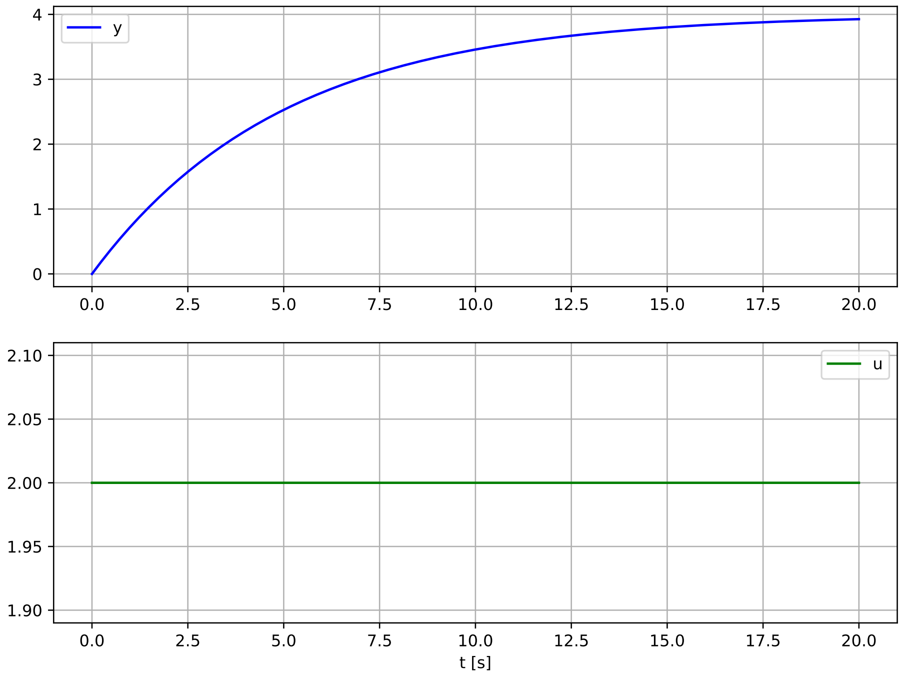


Figure 2.5: Plots of the output y and the input u.

Figure 2.5 shows plots of the output y and the input u.

[End of Example 2.6]

2.5 Poles and zeros of transfer functions

Poles and zeros of a transfer function, H, can be calculated and plotted in a cartesian diagram with

$$\text{(p, z)} = \text{control.pzmap(H)}$$

Example 2.7 *Poles and zeros of a transfer function*

Given the following transfer function:

$$H(s) = \frac{s+2}{s^2+4}$$

Manual calculations gives:

- Poles:

$$p_{1,2} = \pm 2j$$

- Zero:

$$z = -2$$

Program 2.7 calculates the poles and the zero and plots them with the control.pzmap() function. The plt.savefig() function is used to generate a pdf file of the diagram.

http://techteach.no/python_control/programs/poles_tf.py

Listing 2.7: poles_tf.py

```
import numpy as np
import control
import matplotlib.pyplot as plt

num = np.array([1, 2])
den = np.array([1, 0, 4])

H = control.tf(num, den)

(p, z) = control.pzmap(H)

print('poles =', p)
print('zeros =', z)

plt.savefig('poles_zeros.pdf')
```

The result:

```
poles = [-0.+2.j 0.-2.j]
zeros = [-2.]
```

Figure 2.6 shows the pole-zero plot.

[End of Example 2.7]

2.6 The Padé-approximation of a time delay

The transfer function of a time delay is

$$e^{-T_d s} \tag{2.12}$$

where T_d is the time delay. In the Python Control Package, there is no function to you can not define this s-transfer function (while this is straightforward for z-transfer functions, cf. Ch. 5.1.2). However, you can use the control.pade() function to generate an Padé-approximation of the time delay (2.12).

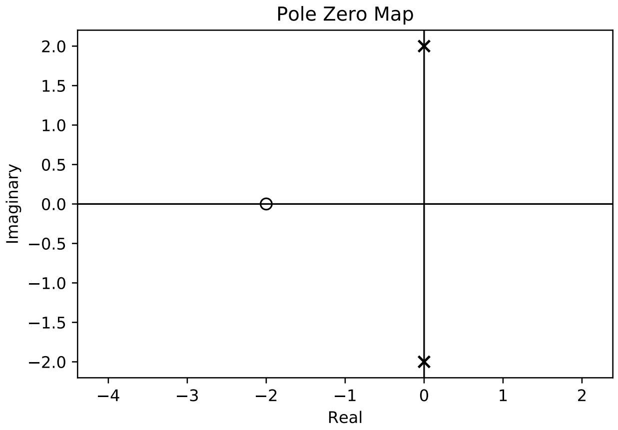


Figure 2.6: Pole-zero plot

Once you have a Padé-approximation of the time delay, you may use the control.series() function to combine it with the transfer function having no time delay:

$$H_{\text{with\_delay}}(s) = H_{\text{without\_delay}}(s) \cdot H_{\text{pade}}(s) \tag{2.13}$$

Example 2.8 *Padé-approximation*

Given the following transfer function with time constant of 10 s and *no* time delay:

$$H_{\text{without\_delay}}(s) = \frac{1}{10s + 1} \tag{2.14}$$

Assume that this transfer function is combined in series with a transfer function, $H_{\text{pade}}(s)$, of a 10th order Padé-apprioximation representing a time delay of 10 s. The resulting transfer function is:

$$H_{\text{with\_delay}}(s) = H_{\text{without\_delay}}(s) \cdot H_{\text{pade}}(s) = \frac{1}{10s + 1} \cdot H_{\text{pade}}(s) \tag{2.15}$$

Program 2.8 generates these transfer functions and simulated a step response of $H_{\text{with\_delay}}(s)$.

http://techteach.no/python_control/programs/pade_approx.py

Listing 2.8: pade_approx.py

```
import numpy as np
import control
import matplotlib.pyplot as plt

# %% Generating transfer function of Pade approx:
T_delay = 5
n_pade = 10
(num_pade, den_pade) = control.pade(T_delay, n_pade)
```

```
H_pade = control.tf(num_pade, den_pade)

# %% Generating transfer function without time delay:
num = np.array([1])
den = np.array([10, 1])
H_without_delay = control.tf(num, den)

# %% Generating transfer function with time delay:
H_with_delay = control.series(H_pade, H_without_delay)

# %% Simulation of step response:
t = np.linspace(0, 40, 100)
(t, y) = control.step_response(H_with_delay, t)
plt.plot(t, y)
plt.xlabel('t [s]')
plt.grid()

# %% Generating pdf file of the plotting figure:
plt.savefig('pade_approx.pdf')
```

Figure 2.7 shows the step response of $H_{\text{with\_delay}}(s)$.

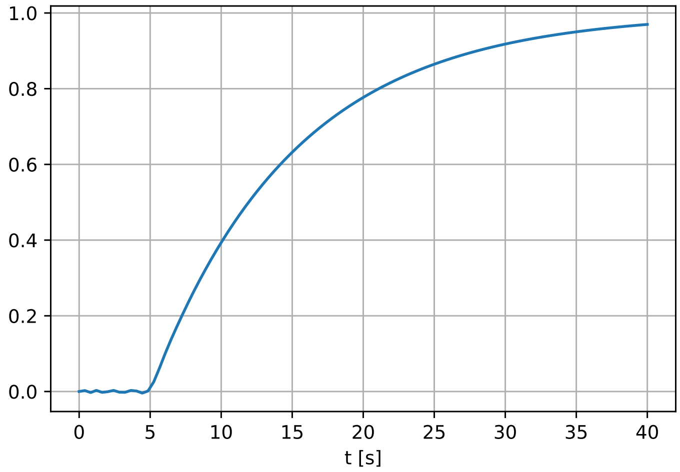


Figure 2.7: Step response of $H_{\text{with\_delay}}(s)$ where the time delay is approximated with a Padé-approximation.

[End of Example 2.8]

Chapter 3

Frequency response

3.1 Frequency response of transfer functions

The function control.bode_plot() generates frequency response data in terms of magnitude and phase. The function may also plot the data in a Bode diagram. However, in the following example, I have instead used the plt.plot() function to plot the data as this gives more freedom to configure the plot.

Example 3.1 *Frequency response*

A first order lowpass filter has the following transfer function:

$$H(s) = \frac{1}{\frac{s}{\omega_b} + 1} \tag{3.1}$$

where $\omega_b = 1$ rad/s, which is the bandwidth.

Program 3.1 generates and plots frequency response of $H(s)$ in terms of magnitude and phase.

http://techteach.no/python_control/programs/bode_plot_lowpass_filter.py

Listing 3.1: bode_plot_lowpass_filter.py

```
import numpy as np
import control
import matplotlib.pyplot as plt

# %% Generating Bode plot:

wb = 1  # Bandwidth [rad/s]
H = control.tf([1], [1/wb, 1])

w0 = 0.1
```

```
w1 = 10
dw = 0.001
nw = int((w1-w0)/dw) + 1 # Number of points of freq
w = np.linspace(w0, w1, nw)

(mag, phase_rad, w) = control.bode_plot(H, w)

# %% Plotting:

plt.close('all')
plt.figure(1, figsize=(12, 9))

plt.subplot(2, 1, 1)
plt.plot(np.log10(w), mag, 'blue')
#plt.xlabel('w [rad/s]')
plt.grid()
plt.legend(labels=('mag',))

plt.subplot(2, 1, 2)
plt.plot(np.log10(w), phase_rad*180/np.pi, 'green')
plt.xlabel('w [rad/s]')
plt.grid()
plt.legend(labels=('phase [deg]',))

# %% Generating pdf file of the plotting figure:

plt.savefig('bode_plot_filter.pdf')
```

Figure 3.1 shows the Bode plot. In the plot we can that bandwidth is indeed 1 rad/s (which is at 0 = log10(1) rad/s in the figure).

[End of Example 3.1]

3.2 Frequency response and stability analysis of feedback loops

Figure 3.2 shows a feedback loop with its loop transfer function, $L(s)$.

control.bode_plot()

We can use the function control.bode_plot() to calculate the magnitude and phase of L, and to plot the Bode plot of L.

The syntax of control.bode_plot() is:

$$\text{(mag, phase\_rad, w)} = \text{control.bode\_plot()}$$

Several input arguments can be set, cf. Example 3.2.

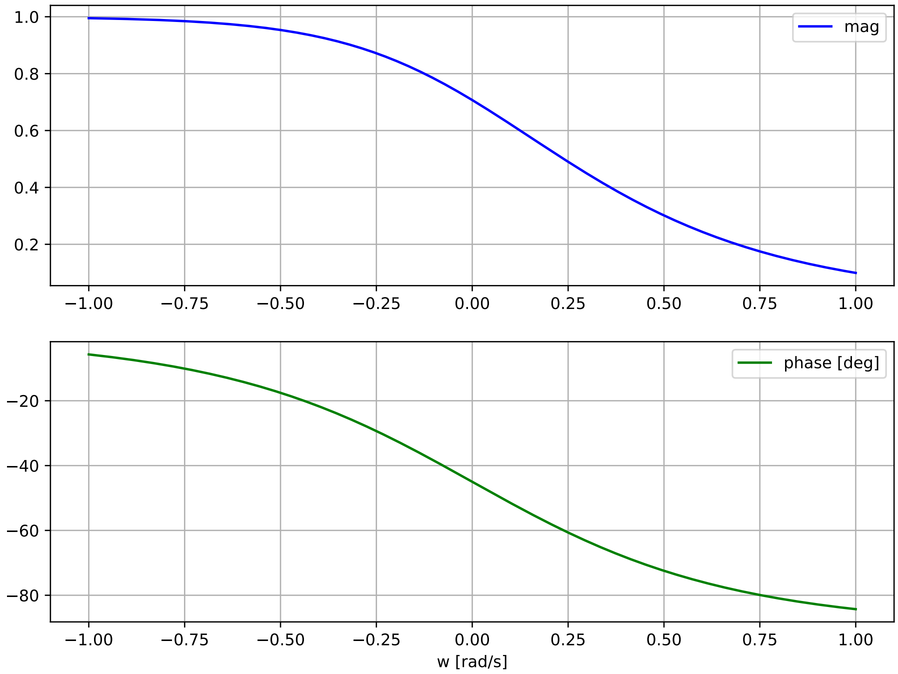


Figure 3.1: Bode plot

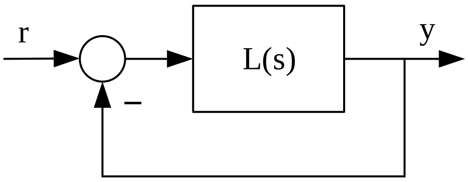


Figure 3.2: A feedback loop with its loop transfer function, $L(s)$

In addition to calculating the three return arguments above, control.bode_plot() can show the following analysis values in the plot:

- The amplitude cross-over frequency, ω_b [rad/s], which is also often regarded as the bandwidth of the feedback system.
- The phase cross-over frequency, ω_{180} [rad/s].
- The gain margin, GM, which is found at $\omega_{180} \equiv \omega_g$ [rad/s] (g for gain margin).
- The phase margin, PM, which is found at $\omega_b \equiv \omega_p$ [rad/s] (p for phase margin).

control.margin()

The control.bode_plot() does *not* return the above four analysis values to the workspace (although it shows them in the Bode plot). Fortunately, we can use the control.margin() function to calculate these analysis values. control.margin() can be used as follows:

$$\text{(GM, PM, wg, wp)} = \text{control.margin(L)}$$

where L is the loop transfer function, and the four return arguments are as in the list above. Note that GM has unit one; *not* dB, and that PM is in degrees.

Example 3.2 demonstrates the use of control.bode_plot() and control.margin().

Example 3.2 *Frequency response*

Given a control loop where the process to be controlled has the following transfer function (an integrator and two time constants in series):

$$P(s) = \frac{1}{(s+1)^2 s}$$

The controller is a P controller:

$$C(s) = K_c$$

where $K_c = 2$ is the controller gain.

The loop transfer function becomes:

$$L(s) = P(s) \cdot C(s) = \frac{K_c}{(s+1)^2 s} = \frac{K_c}{s^3 + 2s + s} \tag{3.2}$$

Program 3.2 generates and plots frequency response of $H(s)$, and shows the stability margins and the cross-over frequencies.

http://techteach.no/python_control/programs/bode_plot_with_stab_margins.py

Listing 3.2: bode_plot_with_stab_margins.py

```
import numpy as np
import control
import matplotlib.pyplot as plt

# %% Creating the loop transfer function:

Kp = 1
C = control.tf([Kp],[1])
P = control.tf([1], [1, 2, 1, 0])
L = control.series(C, P)

# %% Frequencies:

w0 = 0.1
```

```
w1 = 10
dw = 0.001
nw = int((w1-w0)/dw) + 1 # Number of points of freq
w = np.linspace(w0, w1, nw)

# %% Plotting:

plt.close('all')
plt.figure(1, figsize=(12, 9))
(mag, phase_rad, w) = control.bode_plot(
    L, w, dB=True, deg=True, margins=True)

# %% Calculating stability margins and crossover frequencies:

(GM, PM, wg, wp) = control.margin(L)

# %% Printing:

print('GM [1 (not dB)] =', f'{GM:.2f}')
print('PM [deg] =', f'{PM:.2f}')
print('wg [rad/s] =', f'{wg:.2f}')
print('wp [rad/s] =', f'{wp:.2f}')

# %% Generating pdf file of the plotting figure:

plt.savefig('bode_with_stab_margins.pdf')
```

Below are the results of control.margin() as shown in the console. The values are the same as shown in the Bode plot in Figure 3.3 (2 dB ≈ 6).

```
GM [1 (not dB)] = 2.00
PM [deg] = 21.39
wg [rad/s] = 1.00
wp [rad/s] = 0.68
```

[End of Example 3.2]

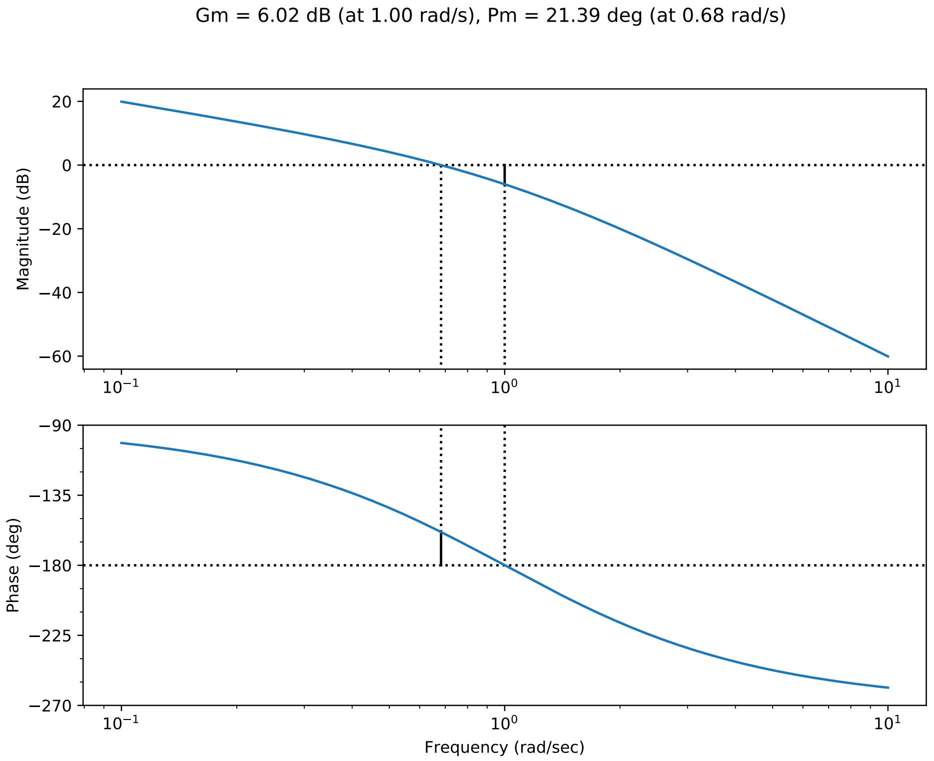


Figure 3.3: Bode plot including the stability margins and the crossover frequencies

Chapter 4

State space models

4.1 How to create state space models

The function control.ss() creates a *linear* state space model with the following form:

$$\dot{x} = Ax + Bu \tag{4.1}$$

$$y = Cx + Bu \tag{4.2}$$

where A, B, C, D are the model matrices.

The syntax of control.ss() is:

S = control.ss(A, B, C, D)

where S is the resulting state space model, and the matrices A, B, C, D are in the form of 2D arrays in Python. (Actually, they may be of the list data type, but I recommend using arrays, cf. Section 1.4.)

Example 4.1 *Creating a state space model*

Figure 4.1 shows a mass-spring-damper-system.

z is position. F is applied force. d is damping constant. k is spring constant. Newton's 2. Law gives the following mathematical model:

$$m\ddot{z}(t) = F(t) - d\dot{z}(t) - kz(t) \tag{4.3}$$

Let us define the following state variables:

- Position:

$$x_1 = z$$

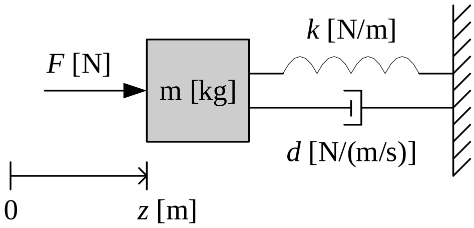


Figure 4.1: Mass-spring-damper system

- Speed:

$$x_2 = \dot{z} = \dot{x}_1$$

Let us define the position x_1 as the output variable:

$$y = x_1$$

Eq. (4.3) can now be expressed with the following equivalent state space model:

$$\underbrace{\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix}}_{\dot{x}} = \underbrace{\begin{bmatrix} 0 & 1 \\ -\frac{k}{m} & -\frac{d}{m} \end{bmatrix}}_{A} \underbrace{\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}}_{x} + \underbrace{\begin{bmatrix} 0 \\ \frac{1}{m} \end{bmatrix}}_{B} F \tag{4.4}$$

$$y = \underbrace{\begin{bmatrix} 1 & 0 \end{bmatrix}}_{C} \underbrace{\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}}_{x} + \underbrace{\begin{bmatrix} 0 \end{bmatrix}}_{D} F \tag{4.5}$$

Assume following parameter values:

$$m = 10 \text{ kg}$$

$$k = 4 \text{ N/m}$$

$$d = 2 \text{ N/(m/s)}$$

Program 4.1 creates the above state space model with the control.ss() function.

http://techteach.no/python_control/programs/create_ss.py

Listing 4.1: create_ss.py

```
import numpy as np
import control

m = 10  # [kg]
k = 4  # [N/m]
d = 2  # [N/(m/s)]

# %% System matrices as 2D arrays:
A = np.array([[0, 1], [-k/m, -d/m]])
B = np.array([[0], [1/m]])
```

```
C = np.array([[1, 0]])
D = np.array([[0]])

# %% Creating the state space model:
S = control.ss(A, B, C, D)

# %% Displaying S:
print('S =', S)
```

The results as shown in the console of Spyder:

```
A = [[ 0.  1.]
 [-0.4 -0.2]]

B = [[0. ]
 [0.1]]

C = [[1. 0.]]

D = [[0.]]
```

4.2 How to get the model matrices of a state space model

You can get (read) the model matrices of a given state space model, say S, with the control.ssdata() function:

(A_list, B_list, C_list, D_list) = control.ssdata(S)

where the matrices are in the form of *lists* (not arrays).

To convert the lists to arrays, you can use the np.array() function, e.g.

A_array = np.array(A_list)

Example 4.2 *Getting the model matrices of a given state space model*

Program 4.2 creates a state space model and gets its matrices with the control.ssdata() function.

http://techteach.no/python_control/programs/get_ss_matrices.py

Listing 4.2: get_ss_matrices.py

```
import numpy as np
import control
```

```
# %% Creating a state space model:
A = np.array([[0, 1], [2, 3]])
B = np.array([[4], [5]])
C = np.array([[6, 7]])
D = np.array([[8]])

S = control.ss(A, B, C, D)

# %% Getting the model matrices as lists and then as arrays:
(A_list, B_list, C_list, D_list) = control.ssdata(S)
A_array = np.array(A_list)
B_array = np.array(B_list)
C_array = np.array(C_list)
D_array = np.array(D_list)

# %% Displaying the matrices as arrays:
print('A_array =', A_array)
print('B_array =', B_array)
print('C_array =', C_array)
print('D_array =', D_array)
```

The results as shown in the console:

```
A_array = [[0. 1.] [2. 3.]]
B_array = [[4.] [5.]]
C_array = [[6. 7.]]
D_array = [[8.]]
```

[End of Example 4.2]

4.3 Simulation with state space models

Simulation with state space models can be done with the control.forced_response() function, cf. Section 2.4, with the following syntax:

The syntax of control.forced_response() is:

(t, y, x) = control.forced_response(sys, t, u, x0)

where sys is the state space model, cf. Section 4.1.

Example 4.3 *Simulation with a state space model*

The program shown below runs a simulation with the state space model presented in Example 4.1 with the following conditions:

- Force (input signal) F is a step of amplitude 10 N, with step time $t = 0$.
- Simulation start time: t0 = 0 s.
- Simulation stop time: t1 = 50 s.
- Simulation time step, or sampling time: dt = 0.01 s.
- Initial states: $x_{1,0}$= 1 m, $x_{2,0}$ = 0 m/s.

Program 4.3 implements the simulation.

http://techteach.no/python_control/programs/sim_ss.py

Listing 4.3: sim_ss.py

```
# %% Import:

import numpy as np
import control
import matplotlib.pyplot as plt

# %% Model parameters:

m = 10  # [kg]
k = 4  # [N/m]
d = 2  # [N/(m/s)]

# %% System matrices as 2D arrays:

A = np.array([[0, 1], [-k/m, -d/m]])
B = np.array([[0], [1/m]])
C = np.array([[1, 0]])
D = np.array([[0]])

# %% Creating the state space model:

S = control.ss(A, B, C, D)

# %% Defining signals:

t0 = 0  # [s]
t1 = 50  # [s]
dt = 0.01  # [s]
nt = int(t1/dt) + 1 # Number of points of sim time
t = np.linspace(t0, t1, nt)
F = 10*np.ones(nt)  # [N]

# %% Initial state:

x1_0 = 1  # [m]
x2_0 = 0  # [m/s]
x0 = np.array([x1_0, x2_0])
```

```
# %% Simulation:

(t, y, x) = control.forced_response(S, t, F, x0)

# %% Extracting individual states:

x1 = x[0,:]
x2 = x[1,:]

# %% Plotting:

plt.close('all')
plt.figure(1, figsize=(12, 9))

plt.subplot(3, 1, 1)
plt.plot(t, x1, 'blue')
plt.grid()
plt.legend(labels=('x1 [m]',))

plt.subplot(3, 1, 2)
plt.plot(t, x2, 'green')
plt.grid()
plt.legend(labels=('x2 [m/s]',))

plt.subplot(3, 1, 3)
plt.plot(t, F, 'red')
plt.grid()
plt.legend(labels=('F [N]',))
plt.xlabel('t [s]')

# %% Generating pdf file of the plotting figure:

plt.savefig('sim_ss.pdf')
```

Figure 4.2 shows the simulated signals.

[End of Example 4.3]

4.4 From state space model to transfer function

The function control.ss2tf() derives a transfer function from a given state space model. The syntax is:

$$\text{H} = \text{control.ss2tf(S)}$$

where H is the transfer function and S is the state space model.

Example 4.4 *From state space model to transfer function*

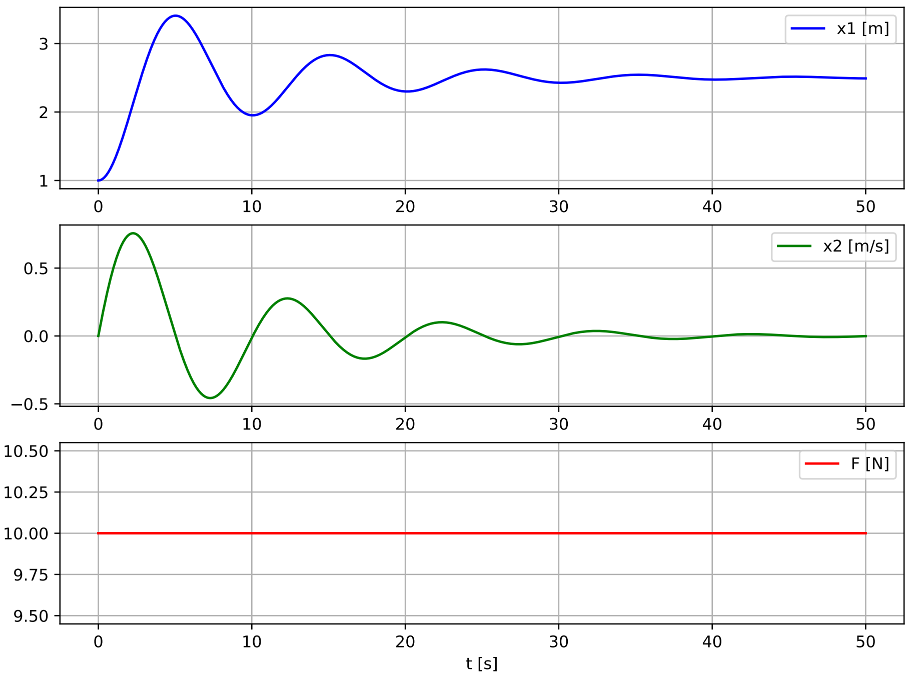


Figure 4.2: Plots of the simulated signals of the mass-spring-damper system.

In Example 4.1 a state space model of a mass-spring-damper system is created with the control.ss() function. The program shown below derives the following two transfer functions from this model:

- The transfer function, H_1, from force F to position x_1. To obtain H_1, the output matrix use is set as
$$C = [1,\ 0]$$
- The transfer function, H_2, from force F to position x_2. To obtain H_2, the output matrix is set as
$$C = [0,\ 1]$$

Program 4.4 derives the two transfer functions from a state space model.

http://techteach.no/python_control/programs/from_ss_to_tf.py

Listing 4.4: from_ss_to_tf.py

```
import numpy as np
import control

# %% Model params:
```

```
m = 10  # [kg]
k = 4  # [N/m]
d = 2  # [N/(m/s)]

# %% System matrices as 2D arrays:

A = np.array([[0, 1], [-k/m, -d/m]])
B = np.array([[0], [1/m]])
D = np.array([[0]])

# %% Creating the state space model with x1 as output:

C1 = np.array([[1, 0]])
S1 = control.ss(A, B, C1, D)

# %% Deriving transfer function H1 from S1:

H1 = control.ss2tf(S1)

# %% Displaying H1:

print(’H1 =’, H1)

# %% Creating the state space model with x2 as output:

C2 = np.array([[0, 1]])
S2 = control.ss(A, B, C2, D)

# %% Deriving transfer function H2 from S2:

H2 = control.ss2tf(S2)

# %% Displaying H1:

print(’H2 =’, H2)
```

The result of the code above, as shown in the console of Spyder, is shown below. The very small numbers – virtually zeros – in the numerators of H1 and H2 are due to numerical inaccuracies in the control.ss2tf() function.

```
H1 =
    0.1
---------------------
s^2 + 0.2 s + 0.4

H2 =
0.1 s + 1.665e-16
---------------------
s^2 + 0.2 s + 0.4
```

[End of Example 4.4]

Chapter 5

Discrete-time models

5.1 Transfer functions

5.1.1 Introduction

Many functions in the Python Control Package are used in the same way for discrete-time transfer functions, or z-transfer functions, as for continuous-time transfer function, or s-transfer function, except that for z-transfer functions, you must include the sampling time Ts as an additional parameter. For example, to create a z-transfer function, the control.tf() is used in this way:

H_d = control.tf(num_d, den_d, Ts)

where Ts is the sampling time. H_d is the resulting z-transfer function.

Thus, the descriptions in Ch. 2 gives you a basis for using these functions for z-transfer functions as well. Therefore, the descriptions are not repeated here. Still there are some specialities related to z-transfer function, and they are presented in the subsequent sections.

5.1.2 How to create transfer functions

The control.tf() function is used to create z-transfer functions with the following syntax:

H = control.tf(num, den, Ts)

where H is the resulting transfer function (object). num (representing the numerator) and den (representing the denominator) are arrays where the elements are the coefficients of the z-polynomials of the numerator and denominator, respectively, in descending order from left to right, with positive exponentials of z. Ts is the sampling time (time step).

Note that control.tf() assumes *positive* exponents of z. Here is one example of such a transfer function:

$$H(z) = \frac{0.1z}{z-1} \tag{5.1}$$

(which is used in Example 5.1). However, in e.g. signal processing, we may see negative exponents in transfer functions. $H(z)$ given by (5.1) and written in terms of negative exponents of z, are:

$$H(z) = \frac{0.1}{1-z^{-1}} \tag{5.2}$$

(5.1) and () are equivalent. But, in the Python Control Package, we must use only positive exponents of z in transfer functions.

Example 5.1 *Creating a z-transfer function*

Given the following transfer function[1]:

$$H(z) = \frac{0.1z}{z-1} \tag{5.3}$$

Program 5.1 creates $H(z)$. The code print(’H(z) = ’, H) is used to present the transfer function in the console (of Spyder).

http://techteach.no/python_control/programs/create_tf_z.py

Listing 5.1: create_tf_z.py

```
import numpy as np
import control

# %% Creating the z-transfer function:
Ts = 0.1
num = np.array([0.1, 0])
den = np.array([1, -1])
H = control.tf(num, den, Ts)

# %% Displaying the transfer function:
print(’H(z) =’, H)
```

The result as shown in the console:

```
H(z) =
  0.1 z
--------
z - 1

dt = 0.1
```

[End of Example 5.1]

---

[1] This is the transfer function of an integrator based on the Euler Backward method of discretization: $y_k = y_{k-1} + Ts \cdot u_k$with sampling time $Ts = 0.1$ s.

5.1.3 Discretizing an s-transfer function

The control.sample_system() function can be used to discretize given continuous-time models, including s-transfer functions:

sys_disc = control.sample_system(sys_cont, Ts, method='zoh')

where:

- sys_cont is the continuous-time model – a transfer function, or a state space model.
- Ts is the sampling time.
- The discretization method is 'zoh' (zero order hold) by default, but you can alternatively use 'matched' or 'tustin'. (No other methods are supported.)
- sys_disc is the resulting discrete-time model – a transfer function, or a state space model.

Example 5.2 *Discretizing an s-transfer function*

Given the following s-transfer function:

$$H_c(s) = \frac{3}{2s+1} \tag{5.4}$$

Program 5.2 discretizes this transfer function using the zoh method with sampling time 0.1 s.

http://techteach.no/python_control/programs/discretize_tf.py

Listing 5.2: discretize_tf.py

```
import numpy as np
import control

# %% Creating the s-transfer function:
num_cont = np.array([3])
den_cont = np.array([2, 1])
H_cont = control.tf(num_cont, den_cont)

# %% Discretizing:
Ts = 0.1
H_disc = control.sample_system(H_cont, Ts, method='zoh')

# %% Displaying the z-transfer function:
print('H_disc(z) =', H_disc)
```

The result as shown in the console:

```
H_disc(z) =
  0.1463
-----------
z - 0.9512

dt = 0.1
```

[End of Example 5.2]

5.1.4 Exact representation of a time delay with a z-transfer function

In Section 2.6 we saw how to use the control.pade() function to generate a transfer function which is an Padé-approximation of the true transfer function of the time delay, $e^{-T_d s}$. As alternative to the Padé-approximation, you can generate an exact representation of the time delay in terms of a z-transfer function.

The z-transfer function of a time delay is:

$$H_d(z) = \frac{1}{z^{n_d}} \tag{5.5}$$

where

$$n_d = \frac{T_d}{T_s} \tag{5.6}$$

Example 5.3 *Creating a z-transfer function of a time delay*

Assume the time delay is

$$T_d = 5\,\mathrm{s}$$

and the sampling time is

$$T_s = 0.1\,\mathrm{s}$$

So, the transfer function of the time delay becomes

$$H_{\mathrm{delay}}(z) = \frac{1}{z^{n_d}}$$

with

$$n_d = \frac{T_d}{T_s} = \frac{5}{0.1} = 50$$

Python program 5.3 creates $H_{\mathrm{delay}}(z)$, which represents this time delay exactly. The program also simulates the step response of $H_{\mathrm{delay}}(z)$.[2]

http://techteach.no/python_control/programs/time_delay_hz.py

---

[2] For some reason, the returned simulation array, y, becomes a 2D array. I turn it into a 1D array with y = y[0,:] for the plotting.

Listing 5.3: time_delay_hz.py

```python
import numpy as np
import control
import matplotlib.pyplot as plt

# %% Generating a z-transfer function of a time delay:

Ts = 0.1
Td = 5
nd = int(Td/Ts)
denom_tf = np.append([1], np.zeros(nd))
H_delay = control.tf([1], denom_tf, Ts)

# %% Displaying the z-transfer function:

print('H_delay(z) =', H_delay)

# %% Sim of step response of time delay transfer function:

t = np.arange(0, 10+Ts, Ts)
(t, y) = control.step_response(H_delay, t)
y = y[0,:] # Turning 2D array into 1D array for plotting
plt.plot(t, y)
plt.xlabel('t [s]')
plt.grid()

# %% Generating pdf file of the plotting figure:

plt.savefig('step_response_hz_time_delay.pdf')
```

The result as shown in the console:

```
H_delay(z) =
  1
--------
z^50

dt = 0.1
```

Figure 5.1 shows the step response of $H_{\mathrm{delay}}(z)$.

[End of Example 5.3]

5.2 Frequency response

Frequency response analysis of z-transfer functions is accomplished with the same functions as for s-transfer function. Therefore, I assume it is sufficient that I refer you to Ch. 3.

However, note the following comment in the manual of the Python Control Package: “If a discrete time model is given, the frequency response is plotted along the upper branch of

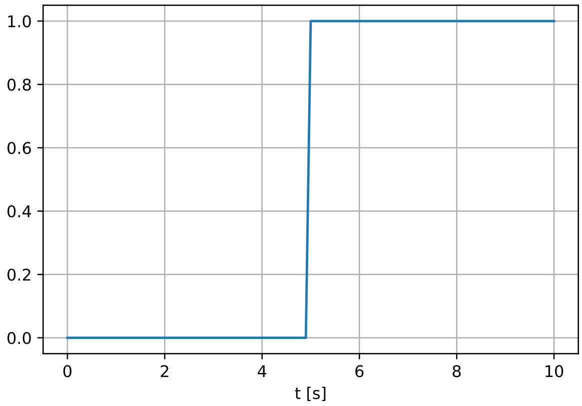


Figure 5.1: The step response of $H_{\text{delay}}(z)$

the unit circle, using the mapping z = exp(j omega dt) where omega ranges from 0 to pi/dt and dt is the discrete timebase. If not timebase is specified (dt = True), dt is set to 1.”

5.3 State space models

In the Python Control Package, discrete-time linear state space models have the following form:

$$x_{k+1} = A_d x_k + B_d u_k \tag{5.7}$$

$$y_k = C_d x_k + B u_{dk} \tag{5.8}$$

where A_d, B_d, C_d, D_d are the model matrices.

Many functions in the package are used in the same way for both discrete-time linear state space models and for continuous-time state space models, except that for discrete-time state space models, you must include the sampling time Ts as an additional parameter. For example, to create a discrete-time state space model, the control.ss() is used in this way:

S_d = control.ss(A_d, B_d, C_d, D_d, Ts)

where Ts is the sampling time. S_d is the resulting discrete time state space model.

Thus, the descriptions in Ch. 4 gives you a basis for using these functions for continuous-time state space models as well. Therefore, the descriptions are not repeated here.