https://www.halvorsen.blog

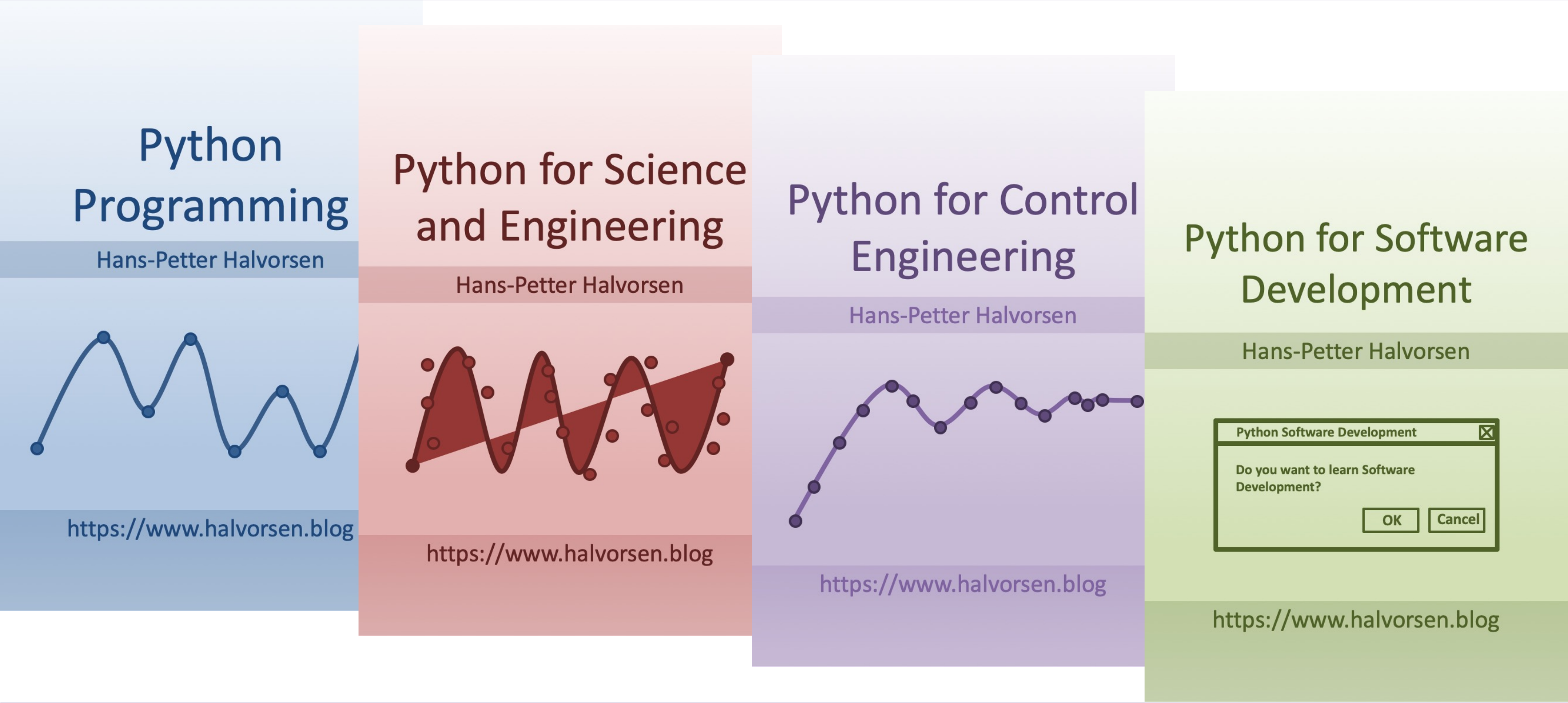
Python for Control Engineering

Hans-Petter Halvorsen

Free Textbook with lots of Practical Examples

https://www.halvorsen.blog/documents/programming/python/

Additional Python Resources



https://www.halvorsen.blog/documents/programming/python/

Contents

- Introduction to Control Engineering
- Python Libraries useful in Control Engineering Applications
  - NumPy, Matplotlib
  - SciPy (especially scipy.signal)
  - Python Control Systems Library (control)
- Python Examples
- Additional Tutorials/Videos/Topics

Additional Tutorials/Videos/Topics

This Tutorial is only the beginning. Some Examples of Tutorials that goes more in depth:

- Transfer Functions with Python
- State-space Models with Python
- Frequency Response with Python
- PID Control with Python
- Stability Analysis with Python
- Frequency Response Stability Analysis with Python
- Logging Measurement Data to File with Python
- Control System with Python
  - Exemplified using Small-scale Industrial Processes and Simulators
- DAQ Systems
- etc.

Videos available on YouTube

https://www.halvorsen.blog/documents/programming/python/

Python Libraries

Hans-Petter Halvorsen

NumPy, Matplotlib

- In addition to Python itself, the Python libraries NumPy, Matplotlib is typically needed in all kind of application
- If you have installed Python using the Anaconda distribution, these are already installed

SciPy.signal

- An alternative to The Python Control Systems Library is SciPy.signal, i.e. the Signal Module in the SciPy Library
- https://docs.scipy.org/doc/scipy/reference/signal.html

With SciPy.signal you can create Transfer Functions, State-space Models, you can simulate dynamic systems, do Frequency Response Analysis, including Bode plot, etc.

Continuous-time linear systems

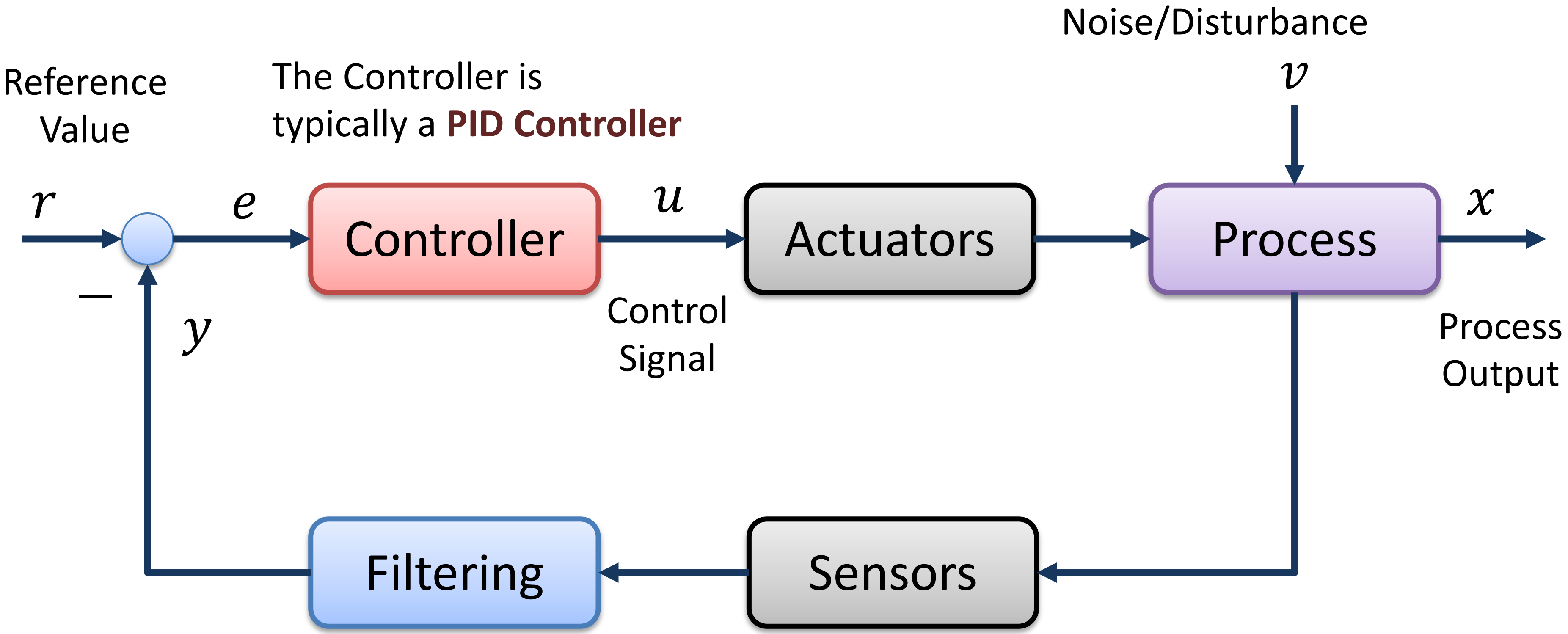
lti(*system)	Continuous-time linear time invariant system base class.
StateSpace(*system, **kwargs)	Linear Time Invariant system in state-space form.
TransferFunction(*system, **kwargs)	Linear Time Invariant system class in transfer function form.
ZerosPolesGain(*system, **kwargs)	Linear Time Invariant system class in zeros, poles, gain form.
lsim(system, U, T[, X0, interp])	Simulate output of a continuous-time linear system.
lsim2(system[, U, T, X0])	Simulate output of a continuous-time linear system, by using the ODE solver **scipy.integrate.odeint**.
impulse(system[, X0, T, N])	Impulse response of continuous-time system.
impulse2(system[, X0, T, N])	Impulse response of a single-input, continuous-time linear system.
step(system[, X0, T, N])	Step response of continuous-time system.
step2(system[, X0, T, N])	Step response of continuous-time system.
freqresp(system[, w, n])	Calculate the frequency response of a continuous-time system.
bode(system[, w, n])	Calculate Bode magnitude and phase data of a continuous-time system.

Python Control Systems Library

- The Python Control Systems Library (control) is a Python package that implements basic operations for analysis and design of feedback control systems.
- Existing MATLAB user? The functions and the features are very similar to the MATLAB Control Systems Toolbox.
- Python Control Systems Library Homepage: https://pypi.org/project/control
- Python Control Systems Library Documentation: https://python-control.readthedocs.io

Control Engineering

Control System



Control System

- r – Reference Value, SP (Set-point), SV (Set Value)
- y – Measurement Value (MV), Process Value (PV)
- e – Error between the reference value and the measurement value ($e = r - y$)
- v – Disturbance, makes it more complicated to control the process
- u - Control Signal from the Controller

The PID Algorithm

$$u(t) = K_p e + \frac{K_p}{T_i}\int_0^t e d\tau + K_p T_d \dot{e}$$

Where u is the controller output and e is the control error:

$$e(t) = r(t) - y(t)$$

r is the Reference Signal or Set-point

y is the Process value, i.e., the Measured value

Tuning Parameters:

K_p	Proportional Gain
T_i	Integral Time [sec.]
T_d	Derivative Time [sec.]

The PID Algorithm

$$u(t) = \underbrace{K_p e}_{\text{P}} + \underbrace{\frac{K_p}{T_i}\int_0^t e\,d\tau}_{\text{I}} + \underbrace{K_p T_d \dot{e}}_{\text{D}}$$

	Proportional Gain	Integral Time	Derivative Time
Tuning Parameters:	K_p	T_i	T_d

Python Examples

Hans-Petter Halvorsen

Dynamic Systems and Differential Equations

Hans-Petter Halvorsen

Dynamic Systems and Models

- The purpose with a Control System is to Control a Dynamic System, e.g., an industrial process, an airplane, a self-driven car, etc. (a Control System is “everywhere“).
- Typically, we start with a Mathematical model of such as Dynamic System
- The mathematical model of such a system can be
  - A Differential Equation
  - A Transfer Function
  - A State-space Model
- We use the Mathematical model to create a Simulator of the system

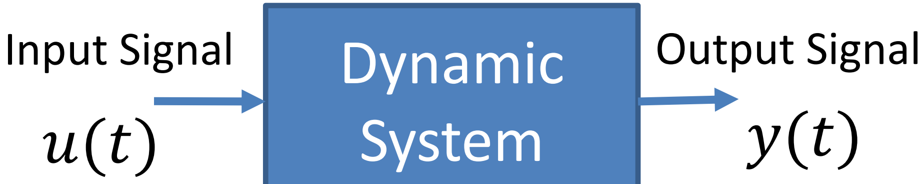
1.order Dynamic System

Assume the following general Differential Equation:

$$\dot{y} = ay + bu$$

or:

$$\boxed{\dot{y} = \frac{1}{T}(-y + Ku)}$$



Where $a = -\frac{1}{T}$ and $b = \frac{K}{T}$

Where K is the Gain and T is the Time constant

This differential equation represents a 1. order dynamic system

Assume $u(t)$ is a step (U), then we can find that the solution to the differential equation is:

$$y(t) = KU(1 - e^{-\frac{t}{T}})$$

(we use Laplace)

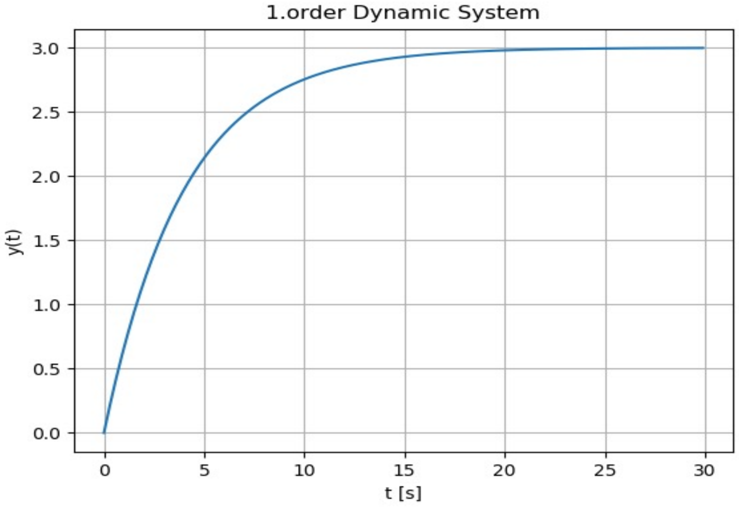
Python

We start by plotting the following:

$$y(t) = KU(1 - e^{-\frac{t}{T}})$$

In the Python code we can set:

$U = 1$
$K = 3$
$T = 4$




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

K = 3
T = 4
start = 0
stop = 30
increment = 0.1
t = np.arange(start,stop,increment)

y = K * (1-np.exp(-t/T))

plt.plot(t, y)
plt.title('1.order Dynamic System')
plt.xlabel('t [s]')
plt.ylabel('y(t) ')
plt.grid()
```

Comments

We have many different options when it comes to simulation a Dynamic System:

- We can solve the differential Equation(s) and then implement the the algebraic solution and plot it.
  - This solution may work for simple systems. For more complicated systems it may be difficult to solve the differential equation(s) by hand
- We can use one of the “built-in” ODE solvers in Python
- We can make a Discrete version of the system
- We can convert the differential equation(s) to Transfer Function(s)
- etc.

We will demonstrate and show examples of all these approaches

Python

Differential Equation:

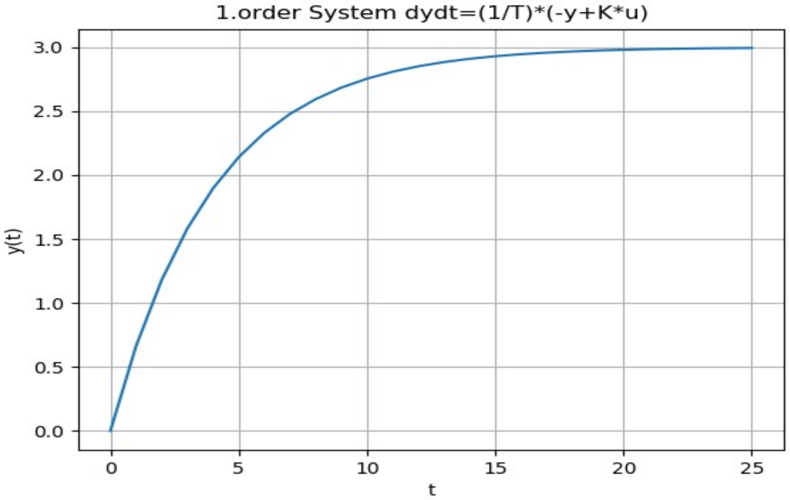
Using ODE Solver

$$\dot{y} = \frac{1}{T}(-y + Ku)$$

In the Python code we can set:

$K = 3$

$T = 4$




```python
import numpy as np
from scipy.integrate import odeint
import matplotlib.pyplot as plt

# Initialization
K = 3
T = 4
u = 1
tstart = 0
tstop = 25
increment = 1
y0 = 0
t = np.arange(tstart,tstop+1,increment)

# Function that returns dx/dt
def system1order(y, t, K, T, u):
    dydt = (1/T) * (-y + K*u)
    return dydt

# Solve ODE
x = odeint(system1order, y0, t, args=(K, T, u))
print(x)

# Plot the Results
plt.plot(t,x)
plt.title('1.order System dydt=(1/T)*(-y+K*u)')
plt.xlabel('t')
plt.ylabel('y(t)')
plt.grid()
plt.show()
```

Discretization

We start with the differential equation:

$$\boxed{\dot{y} = ay + bu}$$

We can use the **Euler forward method**:

$$\dot{y} \approx \frac{y_{k+1} - y_k}{T_s}$$

This gives:

$$\frac{y_{k+1} - y_k}{T_s} = ay_k + bu_k$$

Further we get:

$$y_{k+1} = y_k + T_s(ay_k + bu_k)$$

$$y_{k+1} = y_k + T_s a y_k + T_s b u_k$$

This gives the following discrete differential equation:

$$\boxed{y_{k+1} = (1 + T_s a)y_k + T_s b u_k}$$

Python

Let's simulate the discrete system:

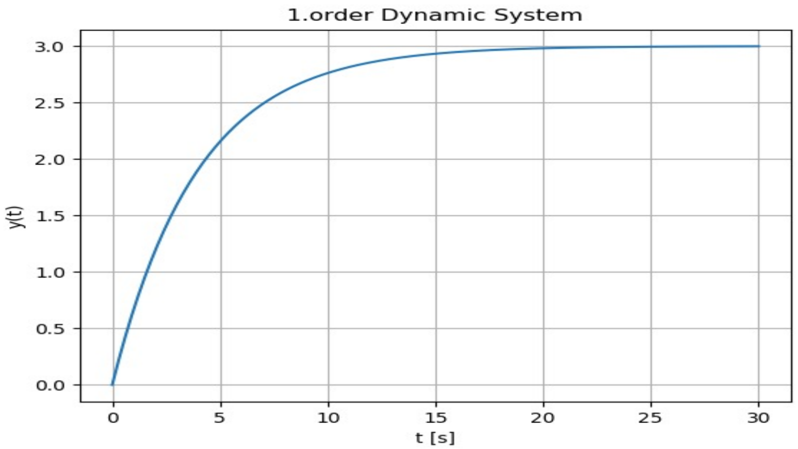
$$y_{k+1} = (1 + T_s a)y_k + T_s b u_k$$

Where $a = -\frac{1}{T}$ and $b = \frac{K}{T}$

In the Python code we can set:

$K = 3$

$T = 4$




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

# Model Parameters
K = 3
T = 4

a = -1/T
b = K/T

# Simulation Parameters
Ts = 0.1
Tstop = 30
uk = 1 # Step Response
yk = 0 # Initial Value
N = int(Tstop/Ts) # Simulation length
data = []
data.append(yk)

# Simulation
for k in range(N):
    yk1 = (1 + a*Ts) * yk  + Ts * b * uk
    yk = yk1
    data.append(yk1)

# Plot the Simulation Results
t = np.arange(0,Tstop+Ts,Ts)

plt.plot(t,data)
plt.title('1.order Dynamic System')
plt.xlabel('t [s]')
plt.ylabel('y(t)')
plt.grid()
```

Transfer Functions

Hans-Petter Halvorsen

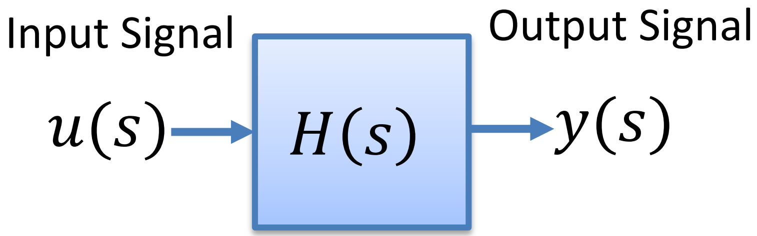
Transfer Functions

- Transfer functions are a model form based on the Laplace transform.
- Transfer functions are very useful in analysis and design of linear dynamic systems.
- You can create Transfer Functions both with SciPy.signal and the Python Control Systems Library

1.order Transfer Functions

A 1.order transfer function is given by:

$$H(s) = \frac{y(s)}{u(s)} = \frac{K}{Ts + 1}$$



Where K is the Gain and T is the Time constant

Differential Equation

$$\dot{y} = \frac{1}{T}(-y + Ku)$$

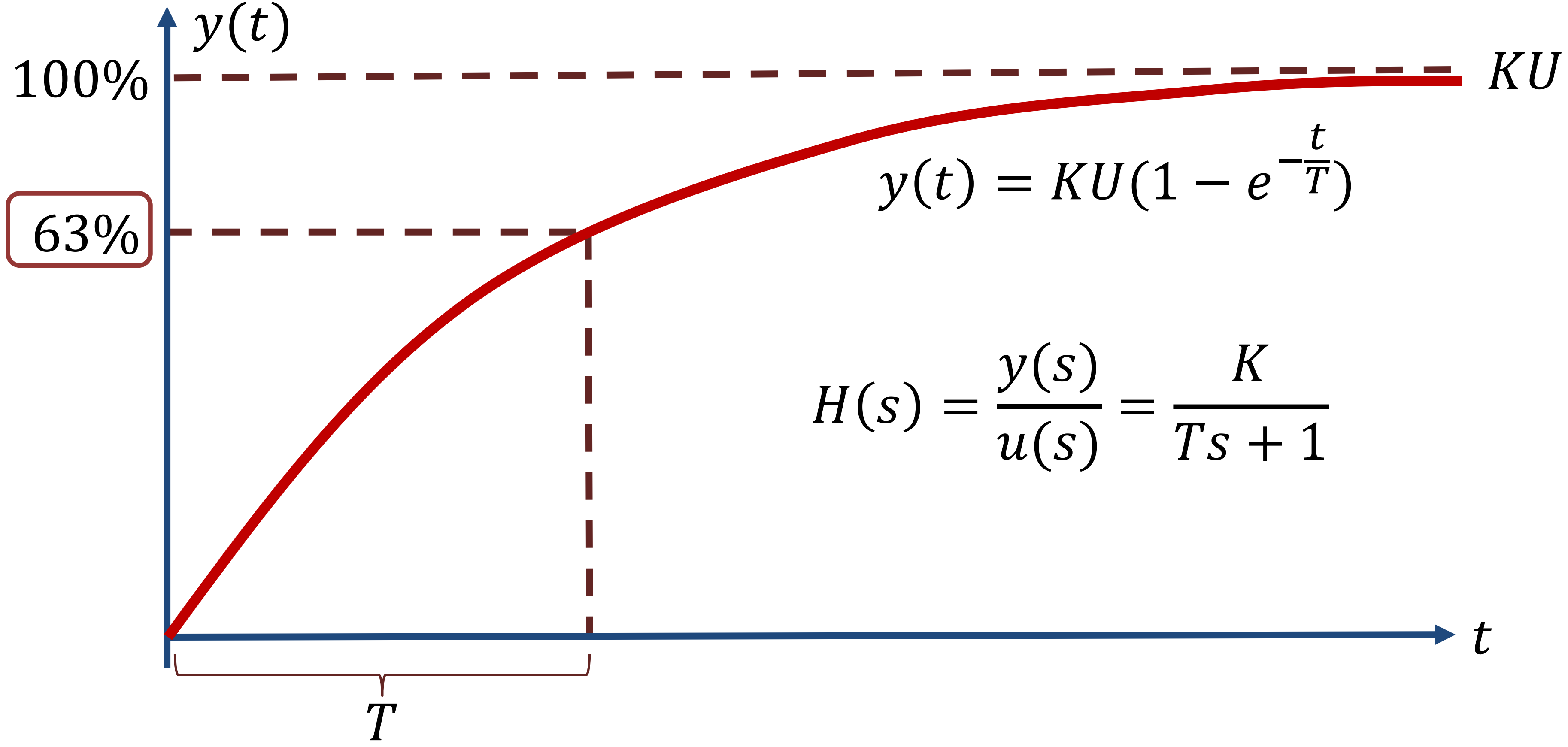
In the time domain we get the following equation (using Inverse Laplace):

$$y(t) = KU(1 - e^{-\frac{t}{T}})$$

(After a Step U for the unput signal $u(s)$)

We ca find the Transfer function from the Differential Equation using Laplace

1.order – Step Response



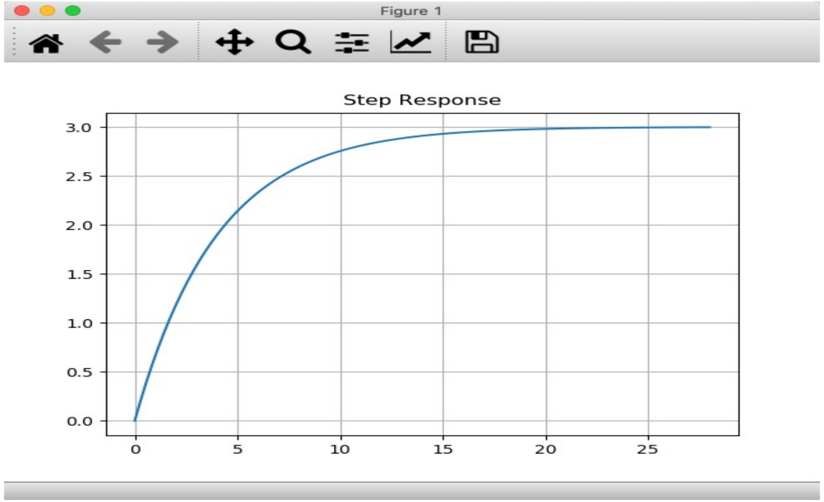
$$y(t) = KU(1 - e^{-\frac{t}{T}})$$

$$H(s) = \frac{y(s)}{u(s)} = \frac{K}{Ts + 1}$$

Python

Transfer Function:

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




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

K = 3
T = 4
num = np.array([K])
den = np.array([T , 1])

H = control.tf(num , den)
print ('H(s) =', H)

t, y = control.step_response(H)

plt.plot(t,y)
plt.title("Step Response")
plt.grid()
```

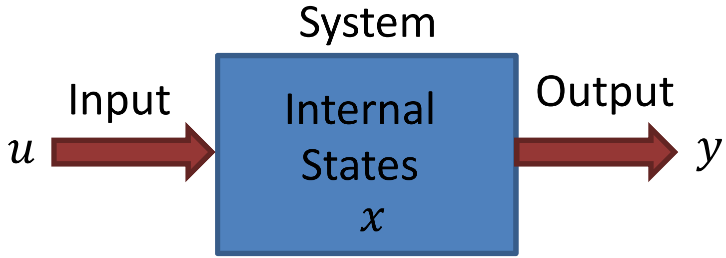
State-space Models

Hans-Petter Halvorsen

State-space Models

A general State-space Model is given by:

$$\boxed{\begin{aligned} \dot{x} &= Ax + Bu \\ y &= Cx + Du \end{aligned}}$$



Note that $\dot{x}$ is the same as $\frac{dx}{dt}$

A, B, C and D are matrices

$x, \dot{x}, u, y$ are vectors

- A state-space model is a structured form or representation of a set of differential equations. State-space models are very useful in Control theory and design. The differential equations are converted in matrices and vectors.
- You can create State.space Models both with SciPy.signal and the Python Control Systems Library

Basic Example

Given the following System:

$$\dot{x}_1 = x_2$$
$$\dot{x}_2 = -x_2 + u$$
$$y = x_1$$

We want to put the equations on the following form:

$$\dot{x} = Ax + Bu$$
$$y = Cx + Du$$

This gives the following State-space Model:

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} u$$

$$y = \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

Where:

$$A = \begin{bmatrix} 0 & 1 \\ 0 & -1 \end{bmatrix} \quad B = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

$$C = \begin{bmatrix} 1 & 0 \end{bmatrix} \quad D = \begin{bmatrix} 0 \end{bmatrix}$$

$$\dot{x} = \begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} \quad x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

Python

We have the differential equations:

$$\dot{x}_1 = \frac{1}{T}(-x_1 + Ku)$$

$$\dot{x}_2 = 0$$

$$y = x_1$$

The State-space Model becomes:

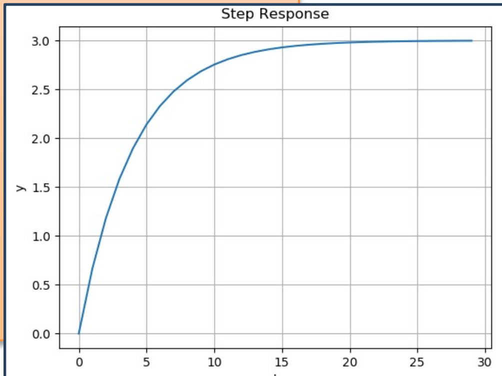
$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} -\frac{1}{T} & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} \frac{K}{T} \\ 0 \end{bmatrix} u$$

$$y = \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

Here we use the following function:

```
t, y = sig.step(sys, x0, t)
```

```
import scipy.signal as sig
import matplotlib.pyplot as plt
import numpy as np

#Simulation Parameters
x0 = [0,0]

start = 0
stop = 30
step = 1
t = np.arange(start,stop,step)

K = 3
T = 4

# State-space Model
A = [[-1/T, 0],
     [0, 0]]
B = [[K/T],
     [0]]
C = [[1, 0]]
D = 0

sys = sig.StateSpace(A, B, C, D)

# Step Response
t, y = sig.step(sys, x0, t)

# Plotting
plt.plot(t, y)
plt.title("Step Response")
plt.xlabel("t")
plt.ylabel("y")
plt.grid()
plt.show()
```



Python

State-space Model:

$$\begin{bmatrix}\dot{x}_1\\ \dot{x}_2\end{bmatrix} = \begin{bmatrix}-\frac{1}{T} & 0\\ 0 & 0\end{bmatrix}\begin{bmatrix}x_1\\ x_2\end{bmatrix} + \begin{bmatrix}\frac{K}{T}\\ 0\end{bmatrix}u$$

$$y = \begin{bmatrix}1 & 0\end{bmatrix}\begin{bmatrix}x_1\\ x_2\end{bmatrix}$$

We want to find the Transfer Function:

$$H(s) = \frac{y(s)}{u(s)}$$

```
TransferFunctionContinuous(
array([0.75, 0.  ]),
array([1.  , 0.25, 0.  ]),
dt: None)
```

Python give us the following:

$$H(s) = \frac{0.75}{s + 0.25}$$

Which is the same as

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

```
import scipy.signal as sig
import matplotlib.pyplot as plt
import numpy as np

#Simulation Parameters
x0 = [0,0]
start = 0; stop = 30; step = 1
t = np.arange(start,stop,step)
K = 3; T = 4

# State-space Model
A = [[-1/T, 0],
     [0, 0]]
B = [[K/T],
     [0]]
C = [[1, 0]]
D = 0

sys = sig.StateSpace(A, B, C, D)

H = sys.to_tf()

print(H)

# Step Response
t, y = sig.step(H, x0, t)

# Plotting
plt.plot(t, y)
plt.title("Step Response")
plt.xlabel("t"); plt.ylabel("y")
plt.grid()
plt.show()
```

Frequency Response

Frequency Response

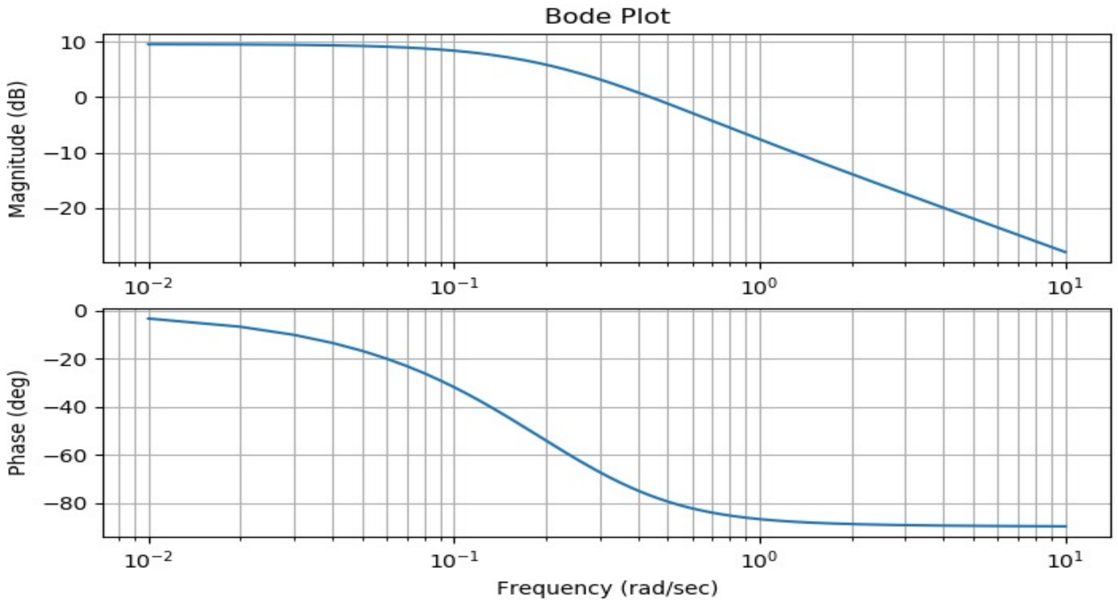
- The Frequency Response is an important tool for Analysis and Design of signal filters and for analysis and design of Control Systems
- The frequency response can be found from a transfer function model
- The Bode diagram gives a simple Graphical overview of the Frequency Response for a given system
- The Bode Diagram is tool for Analyzing the Stability properties of the Control System.

Python

SciPy.signal

Transfer Function Example:

$$H(s) = \frac{3(2s+1)}{(3s+1)(5s+1)}$$



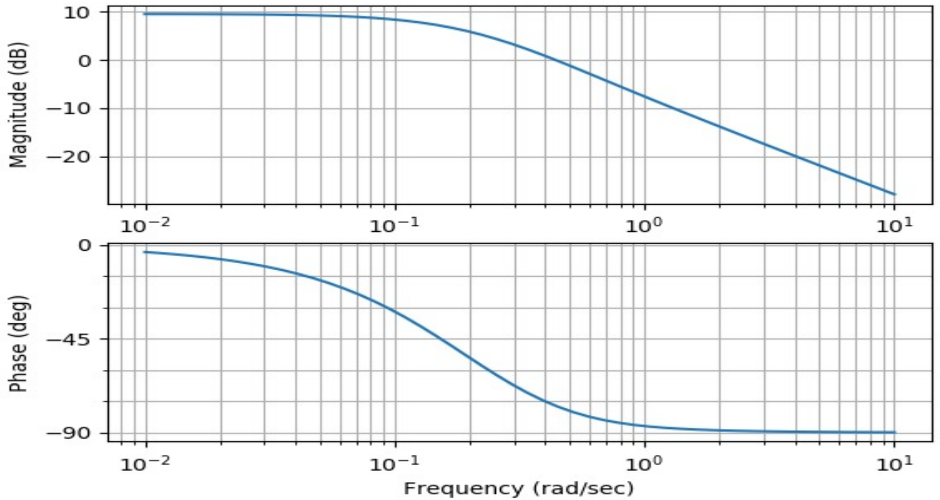

```
import numpy as np
import scipy.signal as signal
import matplotlib.pyplot as plt

# Define Transfer Function
num1 = np.array([3])
num2 = np.array([2, 1])
num = np.convolve(num1, num2)

den1 = np.array([3, 1])
den2 = np.array([5, 1])
den = np.convolve(den1, den2)

H = signal.TransferFunction(num, den)
print ('H(s) =', H)

# Frequencies
w_start = 0.01
w_stop = 10
step = 0.01
N = int ((w_stop-w_start )/step) + 1
w = np.linspace (w_start , w_stop , N)

# Bode Plot
w, mag, phase = signal.bode(H, w)

plt.figure()
plt.subplot (2, 1, 1)
plt.semilogx(w, mag)    # Bode Magnitude Plot
plt.title("Bode Plot")
plt.grid(b=None, which='major', axis='both')
plt.grid(b=None, which='minor', axis='both')
plt.ylabel("Magnitude (dB)")

plt.subplot (2, 1, 2)
plt.semilogx(w, phase)  # Bode Phase plot
plt.grid(b=None, which='major', axis='both')
plt.grid(b=None, which='minor', axis='both')
plt.ylabel("Phase (deg)")
plt.xlabel("Frequency (rad/sec)")
plt.show()
```

Python

Python Control Systems Library

Transfer Function Example:

$$H(s) = \frac{3(2s+1)}{(3s+1)(5s+1)}$$



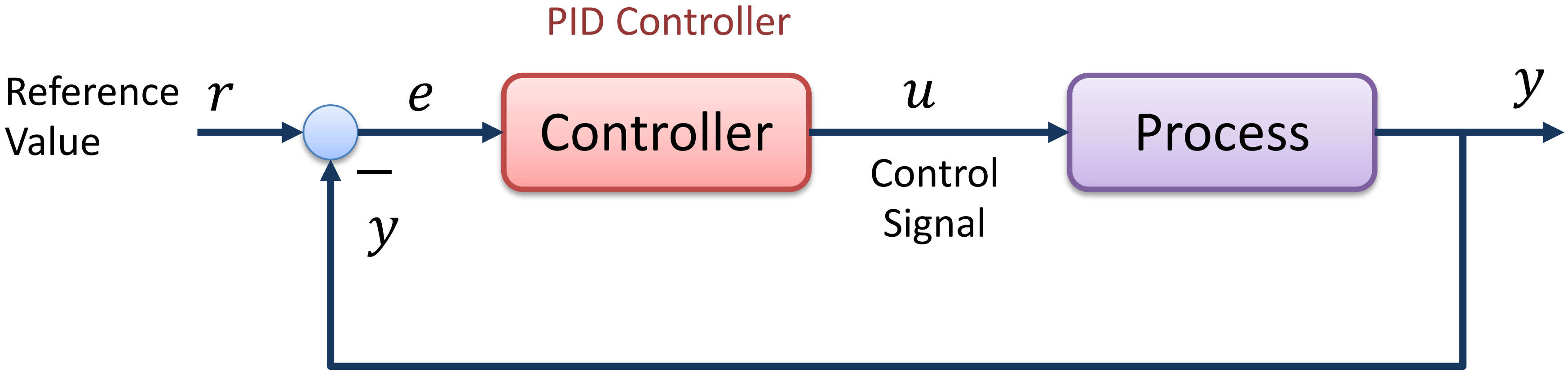

```
import numpy as np
import control

# Define Transfer Function
num1 = np.array([3])
num2 = np.array([2, 1])
num = np.convolve(num1, num2)

den1 = np.array([3, 1])
den2 = np.array([5, 1])
den = np.convolve(den1, den2)

H = control.tf(num, den)
print ('H(s) =', H)

# Bode Plot
control.bode(H, dB=True)
```

PID Control

Hans-Petter Halvorsen

Control System

The purpose with a Control System is to Control a Dynamic System, e.g., an industrial process, an airplane, a self-driven car, etc. (a Control System is “everywhere“).



PID

- The PID Controller is the most used controller today
- It is easy to understand and implement
- There are few Tuning Parameters

The PID Algorithm

$$u(t) = K_p e + \frac{K_p}{T_i}\int_0^t e\,d\tau + K_p T_d \dot{e}$$

Where u is the controller output and e is the control error:

$$e(t) = r(t) - y(t)$$

r is the Reference Signal or Set-point

y is the Process value, i.e., the Measured value

Tuning Parameters:

K_p	Proportional Gain
T_i	Integral Time $[sec.]$
T_d	Derivative Time $[sec.]$

Discrete PI Controller

We start with the continuous PI Controller:

$$u(t) = K_p e + \frac{K_p}{T_i}\int_0^t e d\tau$$

We derive both sides in order to remove the Integral:

$$\dot{u} = K_p \dot{e} + \frac{K_p}{T_i} e$$

We can use the Euler Backward Discretization method:

$$\dot{x} \approx \frac{x(k) - x(k-1)}{T_s}$$

Where T_s is the Sampling Time

Then we get:

$$\frac{u_k - u_{k-1}}{T_s} = K_p \frac{e_k - e_{k-1}}{T_s} + \frac{K_p}{T_i} e_k$$

Finally, we get:

$$u_k = u_{k-1} + K_p(e_k - e_{k-1}) + \frac{K_p}{T_i} T_s e_k$$

Where $e_k = r_k - y_k$

Control System Simulations

PI Controller:

$$u(t) = K_p e + \frac{K_p}{T_i}\int_0^t e d\tau$$

Discrete Version (Ready to implement in Python):

$$e_k = r_k - y_k$$

$$u_k = u_{k-1} + K_p(e_k - e_{k-1}) + \frac{K_p}{T_i}T_s e_k$$

Process (1.order system):

$$\dot{y} = ay + bu$$

Discrete Version (Ready to implement in Python):

$$y_{k+1} = (1 + T_s a)y_k + T_s b u_k$$

Where $a = -\frac{1}{T}$ and $b = \frac{K}{T}$

In the Python code we can set $K = 3$ and $T = 4$

Python

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

# Model Parameters
K = 3
T = 4
a = -(1/T)
b = K/T

# Simulation Parameters
Ts = 0.1 # Sampling Time
Tstop = 20 # End of Simulation Time
N = int(Tstop/Ts) # Simulation length
y = np.zeros(N+2) # Initialization the Tout vector
y[0] = 0 # Initial Vaue

# PI Controller Settings
Kp = 0.5
Ti = 5

r = 5 # Reference value
e = np.zeros(N+2) # Initialization
u = np.zeros(N+2) # Initialization

# Simulation
for k in range(N+1):
    e[k] = r - y[k]
    u[k] = u[k-1] + Kp*(e[k] - e[k-1]) + (Kp/Ti)*Ts*e[k]
    y[k+1] = (1+Ts*a)*y[k] + Ts*b*u[k]

# Plot the Simulation Results
t = np.arange(0,Tstop+2*Ts,Ts) #Create the Time Series
```

```
# Plot Process Value
plt.figure(1)
plt.plot(t,y)

# Formatting the appearance of the Plot
plt.title('Control of Dynamic System')
plt.xlabel('t [s]')
plt.ylabel('y')
plt.grid()
xmin = 0
xmax = Tstop
ymin = 0
ymax = 8
plt.axis([xmin, xmax, ymin, ymax])
plt.show()

# Plot Control Signal
plt.figure(2)
plt.plot(t,u)

# Formatting the appearance of the Plot
plt.title('Control Signal')
plt.xlabel('t [s]')
plt.ylabel('u [V]')
plt.grid()
```

Python

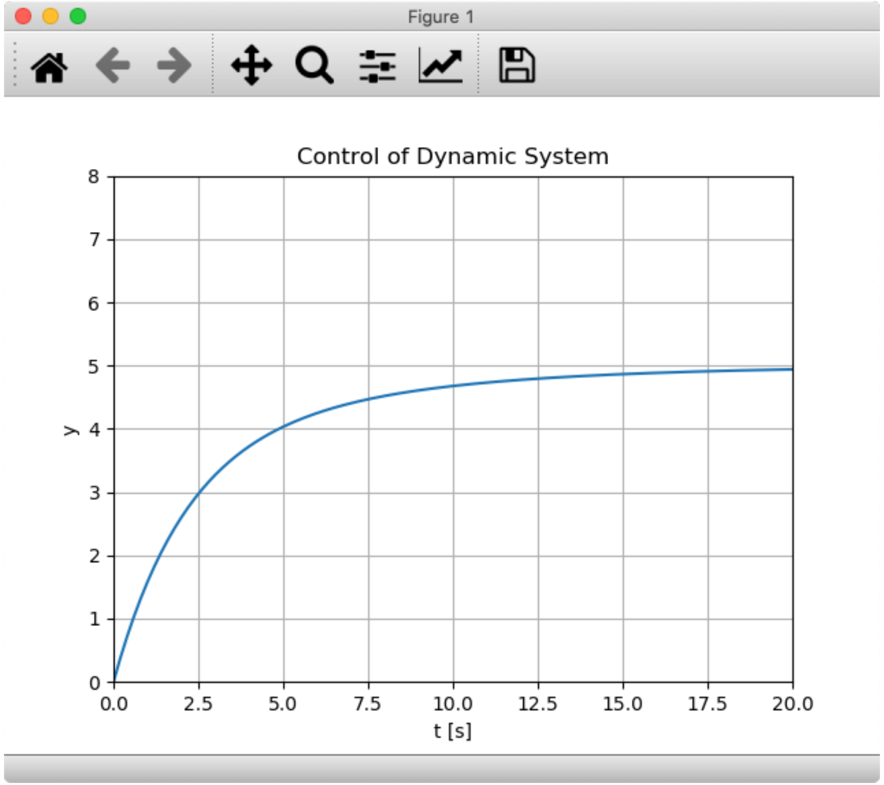


Figure 1
Control of Dynamic System
y
t [s]
0
1
2
3
4
5
6
7
8
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0

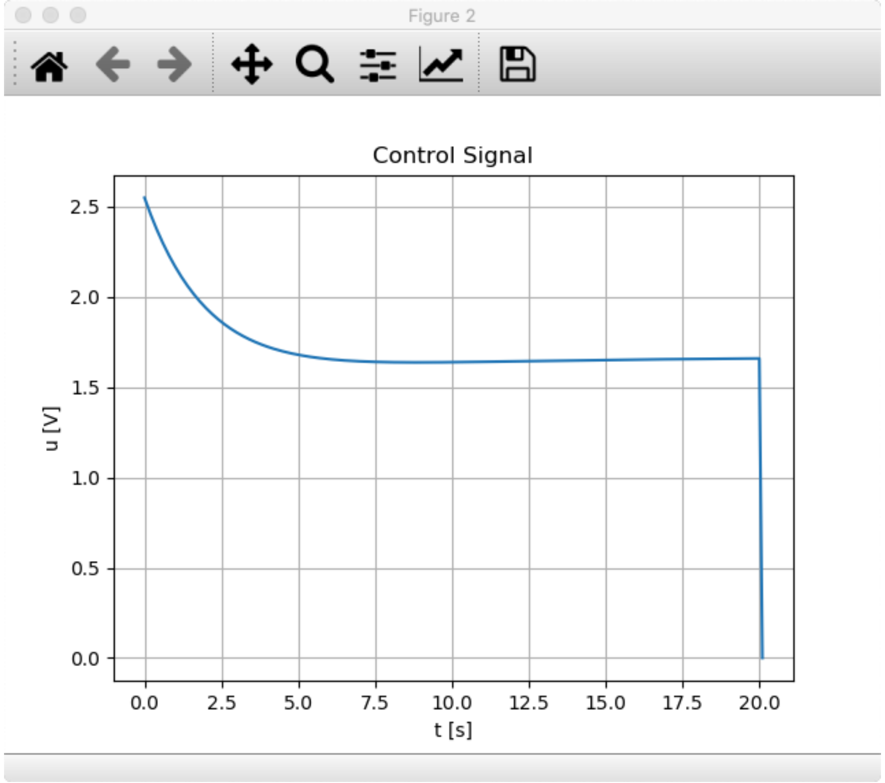
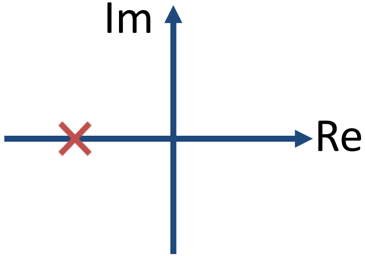
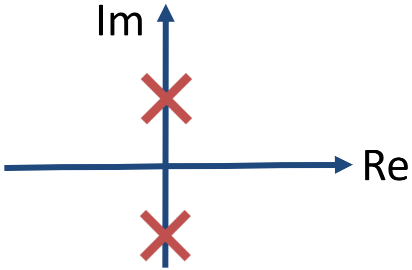
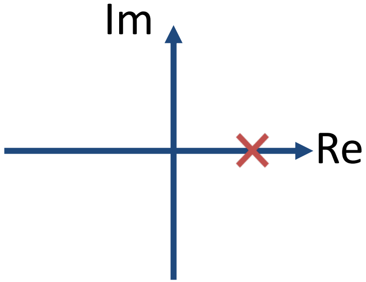
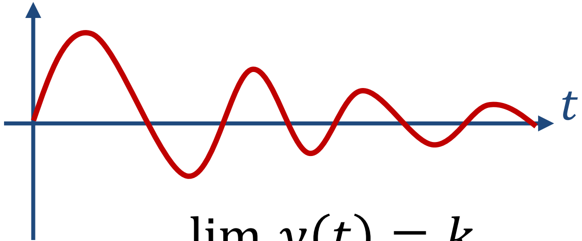
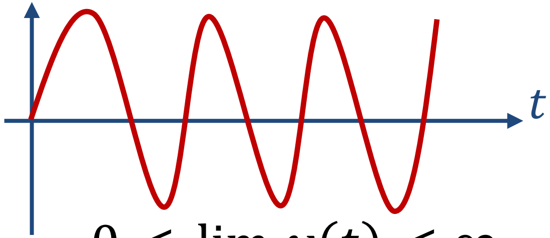
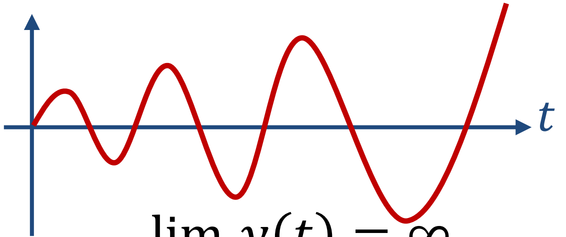


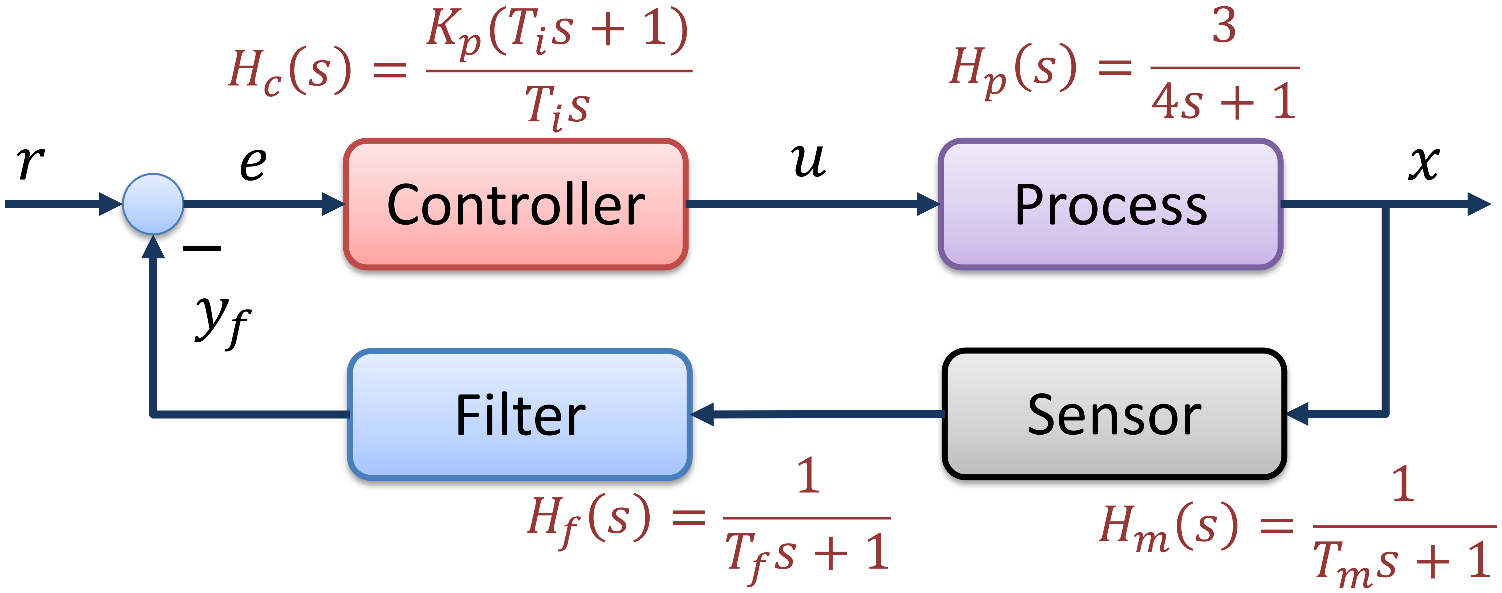
Figure 2
Control Signal
u [V]
t [s]
0.0
0.5
1.0
1.5
2.0
2.5
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0

Stability Analysis

Stability Analysis

	Asymptotically Stable System	Marginally Stable System	Unstable System
Poles:			
Step Response:	 $\lim_{t\to\infty} y(t) = k$	 $0 < \lim_{t\to\infty} y(t) < \infty$	 $\lim_{t\to\infty} y(t) = \infty$
Frequency Response:	$\omega_c < \omega_{180}$	$\omega_c = \omega_{180}$	$\omega_c > \omega_{180}$

Stability Analysis Example



In Stability Analysis we use the following Transfer Functions:

Loop Transfer Function: $\boxed{L(s) = H_c(s)H_p(s)H_m(s)H_f(s)}$

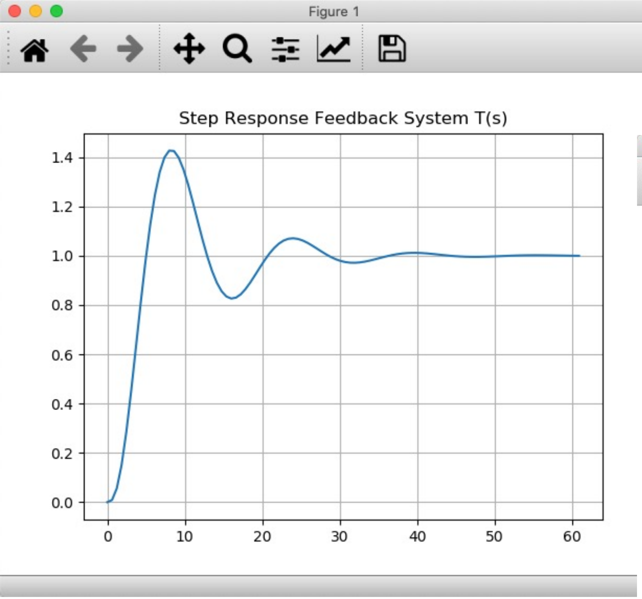
Tracking Transfer Function: $\boxed{T(s) = \frac{y(s)}{r(s)} = \frac{L(s)}{1+L(s)}}$

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

# Transfer Function Process
K = 3; T = 4
num_p = np.array ([K])
den_p = np.array ([T , 1])
Hp = control.tf(num_p , den_p)
print ('Hp(s) =', Hp)

# Transfer Function PI Controller
Kp = 0.4
Ti = 2
num_c = np.array ([Kp*Ti, Kp])
den_c = np.array ([Ti , 0])
Hc = control.tf(num_c, den_c)
print ('Hc(s) =', Hc)

# Transfer Function Measurement
Tm = 1
num_m = np.array ([1])
den_m = np.array ([Tm , 1])
Hm = control.tf(num_m , den_m)
print ('Hm(s) =', Hm)

# Transfer Function Lowpass Filter
Tf = 1
num_f = np.array ([1])
den_f = np.array ([Tf , 1])
Hf = control.tf(num_f , den_f)
print ('Hf(s) =', Hf)

# The Loop Transfer function
L = control.series(Hc, Hp, Hf, Hm)
print ('L(s) =', L)
```

```python
# Tracking transfer function
T = control.feedback(L,1)
print ('T(s) =', T)

# Step Response Feedback System (Tracking System)
t, y = control.step_response(T)
plt.figure(1)
plt.plot(t,y)
plt.title("Step Response Feedback System T(s)")
plt.grid()

# Bode Diagram with Stability Margins
plt.figure(2)
control.bode(L, dB=True, deg=True, margins=True)

# Poles and Zeros
control.pzmap(T)
p = control.pole(T)
z = control.zero(T)
print("poles = ", p)

# Calculating stability margins and crossover frequencies
gm , pm , w180 , wc = control.margin(L)

# Convert gm to Decibel
gmdb = 20 * np.log10(gm)

print("wc =", f'{wc:.2f}', "rad/s")
print("w180 =", f'{w180:.2f}', "rad/s")

print("GM =", f'{gm:.2f}')
print("GM =", f'{gmdb:.2f}', "dB")
print("PM =", f'{pm:.2f}', "deg")

# Find when Sysem is Marginally Stable (Kritical Gain -  Kc)
Kc = Kp*gm
print("Kc =", f'{Kc:.2f}')
```

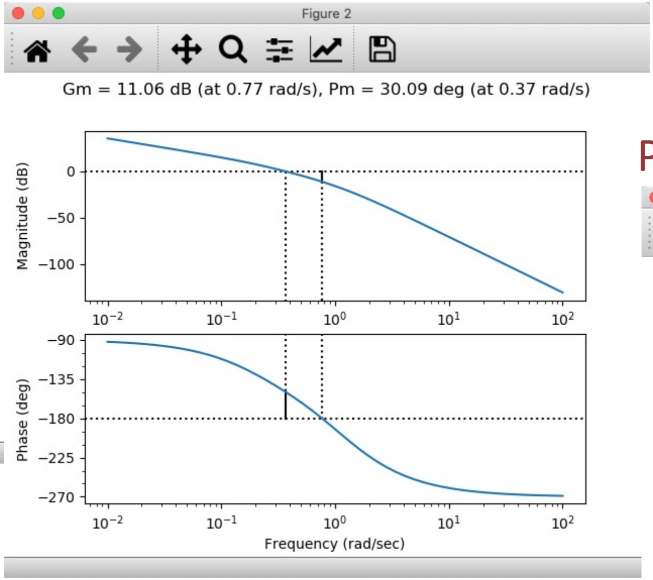
$K_p = 0.4$
$T_i = 2s$

Results



Step Response

Frequency Response

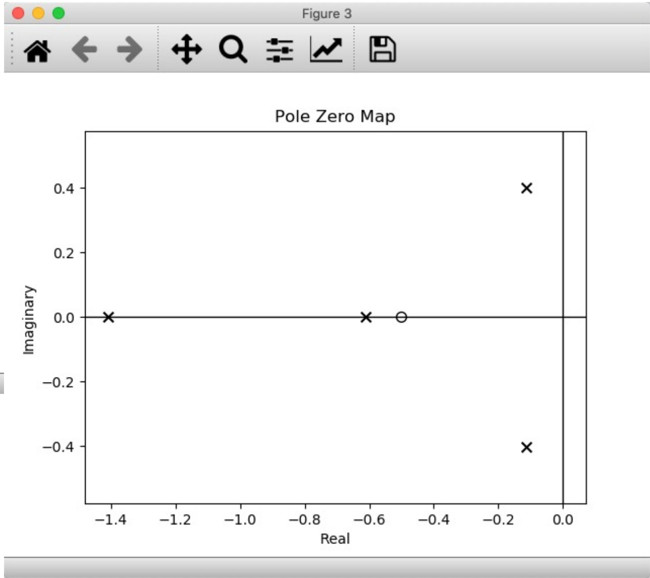


Gain Margin (GM): $\Delta K \approx 11. dB$

Phase Margin (PM): $\varphi \approx 30°$

This means that we can increase K_p a bit without problem

Poles



As you see we have an Asymptotically Stable System

The Critical Gain is $K_c = K_p \times \Delta K = 1.43$

Conclusions

We have an Asymptotically Stable System when $K_p < K_c$

- We have Poles in the left half plane
- $\lim_{t \to \infty} y(t) = 1$ (Good Tracking)
- $\omega_c < \omega_{180}$

We have a Marginally Stable System when $K_p = K_c$

- We have Poles on the Imaginary Axis
- $0 < \lim_{t \to \infty} y(t) < \infty$
- $\omega_c = \omega_{180}$

We have an Unstable System when $K_p > K_c$

- We have Poles in the right half plane
- $\lim_{t \to \infty} y(t) = \infty$
- $\omega_c > \omega_{180}$

Additional Tutorials/Videos/Topics

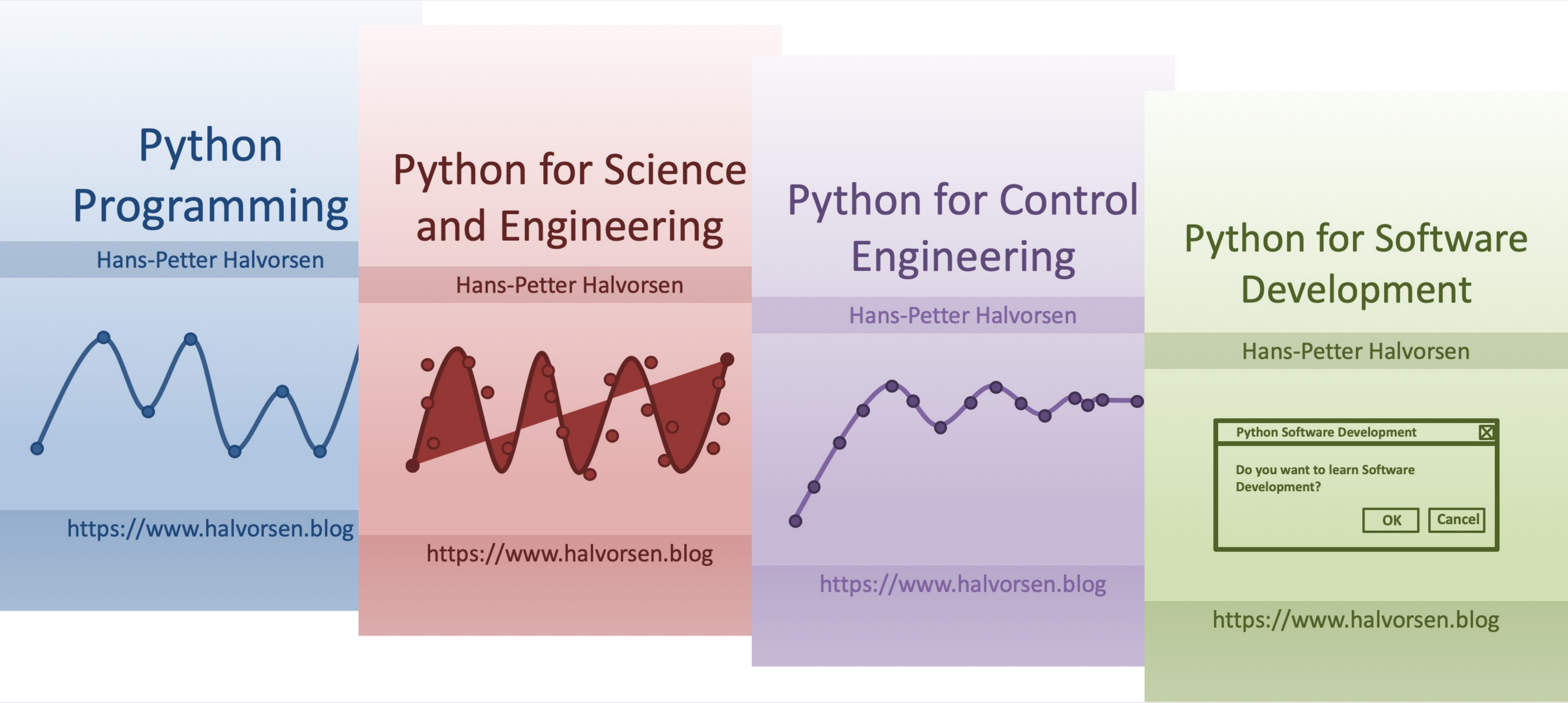
Want to learn more? Some Examples:

- Transfer Functions with Python
- State-space Models with Python
- Frequency Response with Python
- PID Control with Python
- Stability Analysis with Python
- Frequency Response Stability Analysis with Python
- Logging Measurement Data to File with Python
- Control System with Python – Exemplified using Small-scale Industrial Processes and Simulators
- DAQ Systems
- etc.

Videos available on YouTube

https://www.halvorsen.blog/documents/programming/python/

Additional Python Resources



https://www.halvorsen.blog/documents/programming/python/

Hans-Petter Halvorsen

University of South-Eastern Norway

www.usn.no

E-mail: hans.p.halvorsen@usn.no

Web: https://www.halvorsen.blog