L9-3: PID Control of Servomechanism

CDS 110/ChE 105, Winter 2024
Richard Murray

In this problem we will use a variety of methods to design proportional (P), proportional-integral (PI), and proportional-integral-derivative (PID) controllers for a cart pendulum system.

```python
In [1]:
import numpy as np
import matplotlib.pyplot as plt
from math import pi
try:
    import control as ct
    print("python-control", ct.__version__)
except ImportError:
    # Get the development version, which fixes a bug that affects the code bel
    !pip install git+https://github.com/python-control/python-control.git
import control as ct
python-control 0.10.0
```

System dynamics

Consider a simple mechanism for positioning a mechanical arm whose equations of motion are given by

$$J \ddot{\theta} = -b \dot{\theta} - kr \sin \theta + \tau_m,$$

which can be written in state space form as

$$\frac{d}{dt} \begin{bmatrix} \theta \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -kr \sin \theta / J - b \dot{\theta} / J \end{bmatrix} \begin{bmatrix} \dot{\theta} \\ \tau_m \end{bmatrix} + \begin{bmatrix} 0 \\ 1 / J \end{bmatrix} \tau_m.$$

The system consists of a spring loaded arm that is driven by a motor, as shown below.
The motor applies a torque that twists the arm against a linear spring and moves the end of the arm across a rotating platter. The input to the system is the motor torque $\tau_m$. The force exerted by the spring is a nonlinear function of the head position due to the way it is attached.

The system parameters are given by

$$k = 1, \quad J = 100, \quad b = 10, \quad r = 1, \quad l = 2, \quad \epsilon = 0.01.$$ 

and we assume that time is measured in msec and distance in cm. (The constants here are made up and don't necessarily reflect a real disk drive, though the units and time constants are motivated by computer disk drives.)

```python
# Parameter values
servomech_params = {
    'J': 100,  # Moment of inertial of the motor
    'b': 10,   # Angular damping of the arm
    'k': 1,    # Spring constant
}```
'r': 1,  # Location of spring contact on arm
'l': 2,  # Distance to the read head
'eps': 0.01,  # Magnitude of velocity-dependent perturbation

# State derivative
def servomech_update(t, x, u, params):
    # Extract the configuration and velocity variables from the state vector
    theta = x[0]  # Angular position of the disk drive arm
    thetadot = x[1]  # Angular velocity of the disk drive arm
    tau = u[0]  # Torque applied at the base of the arm

    # Get the parameter values
    J, b, k, r = map(params.get, ['J', 'b', 'k', 'r'])

    # Compute the angular acceleration
    dthetadot = 1 / J * (-b * thetadot - k * r * np.sin(theta) + tau)

    # Return the state update law
    return np.array([thetadot, dthetadot])

# System output (full state)
def servomech_output(t, x, u, params):
    l = params['l']
    return l * x[0]

# System dynamics
servomech = ct.nl.sy(s
    servomech_update, servomech_output, name='servomech',
    params=servomech_params,
    states=['theta', 'thdot'],
    outputs=['y'], inputs=['tau'])

In addition to the system dynamics, we assume there are actuator dynamics that limit the performance of the system. We take these as first order dynamics with saturation:

\[
\tau = \text{sat}\left(\frac{\alpha}{s + \alpha} u\right)
\]

In [3]:
actuator_params = {
    'umax': 5,  # Saturation limits
    'alpha': 10,  # Actuator time constant
}

def actuator_update(t, x, u, params):
    # Get parameter values
    alpha = params['alpha']
    umax = params['umax']

    # Clip the input
    u_clip = np.clip(u, -umax, umax)

    # Actuator dynamics
return -alpha * x + alpha * u_clip

actuator = ct.nlsys(
    actuator_update, None, params=actuator_params,
    inputs='u', outputs='tau', states=1, name='actuator')

system = ct.series(actuator, servomech)
if not system._system_name:
    system._system_name = 'system' # missing feature
print(system)

<InterconnectedSystem>: system
Inputs (1): ['u[0]']
Outputs (1): ['y[0]']
States (3): ['actuator_x[0]', 'servomech_theta_', 'servomech_thdot_']

Update: <function InterconnectedSystem.__init__.<locals>.updfcn at 0x140d111c0>
Output: <function InterconnectedSystem.__init__.<locals>.outfcn at 0x140d11260>

Linearization

To study the open loop dynamics of the system, we compute the linearization of the dynamics about the equilibrium point corresponding to $\theta_e = 15^\circ$.

In [4]:
   
   # Convert the equilibrium angle to radians
   theta_e = (15 / 180) * np.pi

   # Compute the input required to hold this position
   u_e = servomech.params['k'] * servomech.params['r'] * np.sin(theta_e)
   print("Equilibrium torque = %g" % u_e)

   # Linearize the system dynamics about the equilibrium point
   P = ct.tf(
       system.linearize([[0, theta_e, 0], u_e, copy_names=\True][0, 0]))
   P.name = 'P' # bug
   print(P, end="\n\n")
   
   ct.bode_plot(P)

Equilibrium torque = 0.258819
<TransferFunction>: P
Inputs (1): ['u[0]']
Outputs (1): ['y[0]']

0.2
---------------------------------
\s^3 + 10.1 \s^2 + 1.01 \s + 0.09659

Out[4]: array([[list([<matplotlib.lines.Line2D object at 0x140d2aab0>])],
       [list([<matplotlib.lines.Line2D object at 0x140d9fc20>])]],
       dtype=object)
# Colab: create a simplified form to get rid of warning messages

```python
if len(P.num[0][0]) > 1:
    P = ct.tf(P.num[0][0][2], P.den, name='P') # Fix up conditioning
print(P)
ct.bode_plot(P)

<TransferFunction>: P
Inputs (1): ['u[0]']
Outputs (1): ['y[0]']

0.2

-----------------------------

s^3 + 10.1 s^2 + 1.01 s + 0.09659
```

Out[5]:
```
array([[<matplotlib.lines.Line2D object at 0x14157b260>],
        [<matplotlib.lines.Line2D object at 0x14157b590>]]),
dtype=object)```
Ziegler-Nichols tuning

(a) Step response method

<table>
<thead>
<tr>
<th>Type</th>
<th>$k_p$</th>
<th>$T_i$</th>
<th>$T_d$</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>$1/a$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PI</td>
<td>$0.9/a$</td>
<td>$\tau/0.3$</td>
<td></td>
</tr>
<tr>
<td>PID</td>
<td>$1.2/a$</td>
<td>$\tau/0.5$</td>
<td>$0.5\tau$</td>
</tr>
</tbody>
</table>

In [6]:  
```python
# Plot the step response
resp = ct.step_response(P)
resp.plot()

# Find the point of the steepest slope
slope = np.diff(resp.outputs) / np.diff(resp.time)
mxi = np.argmax(slope)
mx_time = resp.time[mxi]
```
mx_out = resp.outputs[mxi]
plt.plot(resp.time[mxi], resp.outputs[mxi], 'ro')

# Draw a line going through the point of max slope
mx_slope = slope[mxi]
timepts = np.linspace(0, mx_time*2)
plt.plot(timepts, mx_out + mx_slope * (timepts - mx_time), 'r-')

# Solve for the Ziegler-Nichols parameters
a = -(mx_out - mx_slope * mx_time)  # Find the value of the line at t = 0
tau = a / mx_slope  # Solve a + mx_slope * tau = 0
print(f"{a=}, {tau=}")

a=0.43326263676972254, tau=3.9297636758958916

Step response for P

In [7]:
s = ct.tf('s')

# Proportional controller
kp = 1/a
ctrl_zn_P = kp

# PI controller
kp = 0.9/a
Ti = tau/0.3; ki = kp/Ti
ctrl_zn_PI = kp + ki / s

# PID controller
kp = 1.2/a
Ti = tau/0.5; ki = kp/Ti
\[ Td = 0.5 \times \tau; \quad kd = kp \times Td \]
\[ \text{ctrl}_z \text{n}_\text{PID} = kp + ki / \tau + kd \times \tau \]

```
print(control_zn_PID)
```

TransferFunction: sys[19]
Inputs (1): ['u[0]']
Outputs (1): ['y[0]']

\[
5.442 \ s^2 + 2.77 \ s + 0.3524
\]
---------------------------------------
\[ s \]

```
# Compute the closed loop systems and plots the step and frequency responses.

clsys_zn_P = control.feedback(P * control_zn_P)
clsys_zn_P.name = 'P'

clsys_zn_PI = control.feedback(P * control_zn_PI)
clsys_zn_PI.name = 'PI'

clsys_zn_PID = control.feedback(P * control_zn_PID)
clsys_zn_PID.name = 'PID'

# Plot the step responses
resp.sysname = 'open_loop'
resp.plot(color='k')

stepresp_zn_P = control.step_response(clsys_zn_P)
stepresp_zn_P.plot(color='b')

stepresp_zn_PI = control.step_response(clsys_zn_PI)
stepresp_zn_PI.plot(color='r')

stepresp_zn_PID = control.step_response(clsys_zn_PID)
stepresp_zn_PID.plot(color='g')
plt.legend()
plt.figure()
control.bode_plot([clsys_zn_P, clsys_zn_PI, clsys_zn_PID]);
```
Loop shaping

In [9]:

# Design parameters
Td = 1  # Set to gain crossover frequency
Ti = Td * 10  # Set to low frequency region
kp = 500  # Tune to get desired bandwidth

# Updated gains
kp = 150
Ti = Td * 5; kp = 150

# Compute controller parameters
ki = kp/Ti
kd = kp * Td

# Controller transfer function
ctrl_shape = kp + ki / s + kd * s
ctrl_shape.name = 'C'

# Frequency response (open loop) – use this to help tune your design
ltf_shape = P * ctrl_shape
ltf_shape.name = 'L'

cf.frequency_response([P, ctrl_shape]).plot()
cf.frequency_response(ltf_shape).plot(margins=True);
# Compute the closed loop systems and plot the step response
# and Nyquist plot (to make sure margins look OK)

# Create the closed loop systems
clsys_shape = ct.feedback(P * ctrl_shape)
clsys_shape.name = 'loopshape'

# Step response
plt.subplot(2, 1, 1)
stepresp_shape = ct.step_response(clsys_shape)
stepresp_shape.plot(color='b')
plt.plot([0, stepresp_shape.time[-1]], [1, 1], 'k--')

# Compare to the ZN controller
ax = plt.subplot(2, 1, 2)
ct.step_response(clsys_shape, stepresp_zn_PID.time).plot(
    color='b', ax=np.array([[ax]])
stepresp_zn_PID.plot(color='g', ax=np.array([[ax]]))
ax.plot([0, stepresp_shape.time[-1]], [1, 1], 'k--')

# Nyquist plot
plt.figure()
ct.nyquist([ltf_shape])
Gang of Four

In [11]: `ct.gangof4(P, ctrl.shape)`

Out[11]: `array([[list([<matplotlib.lines.Line2D object at 0x142e665d0>]),
           list([<matplotlib.lines.Line2D object at 0x142e31910>])],
          [list([<matplotlib.lines.Line2D object at 0x142e322d0>]),
           list([<matplotlib.lines.Line2D object at 0x142f49670>])]],
         dtype=object)`
Anti-windup

We now implement the full PID controller with anti-windup and derivative filtering:

I. Review from Wed

Low pass filter

The low pass filtered derivative has transfer function

\[ G(s) = \frac{a s}{s + a}. \]
This can be implemented using the differential equation

\[
\dot{\xi} = -a\xi + ay, \quad \eta = -a\xi + ay.
\]

```
In [12]:
ctrl_params = {'kaw': 5 * ki, 'a': 10/Td}

def ctrl_update(t, x, u, params):
    # Get the parameter values
    kaw = params['kaw']
    a = params['a']
    umax_ctrl = params.get('umax_ctrl', actuator.params['umax'])

    # Extract the signals into more familiar variable names
    r, y = u[0], u[1]
    z = x[0]  # integral error
    xi = x[1]  # filtered derivative

    # Compute the controller components
    u_prop = kp * (r - y)
    u_int = z
    ydt_f = -a * xi + a * (-y)
    u_der = kd * ydt_f

    # Compute the commanded and saturated outputs
    u_cmd = u_prop + u_int + u_der
    u_sat = np.clip(u_cmd, -umax_ctrl, umax_ctrl)

    dz = ki * (r - y) + kaw * (u_sat - u_cmd)
    dxi = -a * xi + a * (-y)
    return np.array([dz, dxi])

def ctrl_output(t, x, u, params):
    # Get the parameter values
    kaw = params['kaw']
    a = params['a']
    umax_ctrl = params.get('umax_ctrl', params['umax'])

    # Extract the signals into more familiar variable names
    r, y = u[0], u[1]
    z = x[0]  # integral error
    xi = x[1]  # filtered derivative

    # Compute the controller components
    u_prop = kp * (r - y)
    u_int = z
    ydt_f = -a * xi + a * (-y)
    u_der = kd * ydt_f

    # Compute the commanded and saturated outputs
    u_cmd = u_prop + u_int + u_der
    u_sat = np.clip(u_cmd, -umax_ctrl, umax_ctrl)

    return u_cmd

ctrl = ct.nlinsys(
```
ctrl_update, ctrl_output, name='ctrl', params=ctrl_params, inputs=['r', 'y'], outputs=['u'], states=2

clsys = ct.interconnect(
    [servomech, actuator, ctrl], name='clsys',
    inputs=['r'], outputs=['y', 'tau'])
print(clsys)

<InterconnectedSystem>: clsys
Inputs (1): ['r']
Outputs (2): ['y', 'tau']
States (5): ['servomech_theta_', 'servomech_thdot_', 'actuator_x[0]', 'ctrl_x[0]', 'ctrl_x[1]']

Update: <function InterconnectedSystem.__init__.<locals>.updfcn at 0x14296e7a0>
Output: <function InterconnectedSystem.__init__.<locals>.outfcn at 0x142e60040>

# Plot the step responses for the following cases:
# 'linear': the original linear step response (no actuation limits)
# 'clipped': PID controller with input limits, but not anti-windup
# 'anti-windup': PID controller with anti-windup compensation

# Use more time points to get smoother response curves
timepts = np.linspace(0, 2*stepresp_shape.time[-1], 500)

# Compute the response for the individual cases
stepsize = theta_e / 2
resp_ln = ct.input_output_response(clsys, timepts, stepsize, params={'umax': np.inf, 'kaw': 0, 'a': 1e3})
resp_cl = ct.input_output_response(clsys, timepts, stepsize, params={'umax': 5, 'kaw': 0, 'a': 100})
resp_aw = ct.input_output_response(clsys, timepts, stepsize, params={'umax': 5, 'kaw': 2*ki, 'a': 100})

# Plot the time responses in a single plot
out = ct.time_response_plot(resp_ln, color='b', plot_inputs=False)
ct.time_response_plot(resp_cl, color='r', plot_inputs=False)
ct.time_response_plot(resp_aw, color='g', plot_inputs=False)

# Annotations
axs = ct.get_plot_axes(out)
axs[0, 0].legend(['linear', 'clipped', 'anti-windup'])
axs[0, 0].plot([0, timepts[-1]], [stepsize, stepsize], 'k--')
axs[1, 0].plot([0, timepts[-1]], [0, 0], 'k--')
axs[1, 0].set_ylim([-5, 15])
axs[1, 0].legend([]);
The response of the anti-windup compensator is very sluggish, indicating that we may be setting $k_{aw}$ too high.

In [14]:

```python
resp_aw = ct.input_output_response(  
    # clsys, timepts, stepsize, params={'umax': 5, 'kaw': 1, 'a': 100})  
    clsys, timepts, stepsize, params={'umax': 5, 'kaw': 0.05 * ki, 'a': 100})

# Plot the time responses in a single plot
out = ct.time_response_plot(resp_ln, color='b', plot_inputs=False)
ct.time_response_plot(resp_cl, color='r', plot_inputs=False)
ct.time_response_plot(resp_aw, color='g', plot_inputs=False)

# Annotations
axs = ct.get_plot_axes(out)
axs[0, 0].legend(['linear', 'clipped', 'anti-windup'])
axs[0, 0].plot([0, timepts[-1]], [stepsize, stepsize], 'k--')
axs[1, 0].plot([0, timepts[-1]], [0, 0], 'k--')
axs[1, 0].set_ylim([-5, 15])
axs[1, 0].legend([]);
```
This gives a much better response, though the value of $k_{aw}$ falls well outside the range of [2, 10]. One reason for this is that $k_{aw}$ acts on the inputs, $\tau$, which are roughly 100 larger than the size of the outputs, $y$, as seen in the above plots.

We can now see if this affects the Gang of Four in the expected way:

In [15]:
```python
C = ctrl.linearize([0, 0], 0, params=resp_aw.params)[0, 1]
ct.gangof4(P, C);
```
Note that in the transfer function from $r$ to $u$ (which is the same as the transfer function from $e$ to $u$, the high frequency gain is now bounded. (We could make it go back down by using a second order filter.)