BASIC OF CONTROL
(with Python)

Masahide TAMAKI

Contents

Chapter 1

INTRODUCTION

1.1 About this document

This document encapsulates very basic points I have learned about control systems. This also includes Python scripts to learn with practical application in mind, with specific references to [1].

I hope this document will be helpful for both beginners and my future endeavors in designing control mechanisms for suspension systems, main interferometers, and related applications.

1.2 What is control?

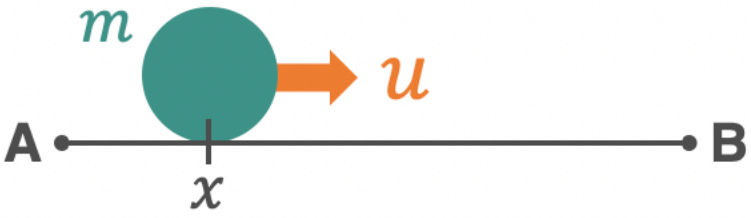

Figure 1.1: Example of control

As an example of control, take a look at Figure 1.1. Consider moving a ball stationary on the ground from point A to point B. Here, Newton's equation of motion is described by

$$m\ddot{x} = u. \tag{1.1}$$

This indicates that the acceleration $\ddot{x}$ is freely manipulated by the force u (control input). In this way, the act of applying a manipulation to transition the state of an object of interest to a desired target state is called "control".

As you know, control systems are ubiquitous in our environment. They are integral to the operation of everyday technologies such as trains and cars, as well as in the suspensions and interferometers of KAGRA.

1.2.1 Feedback control

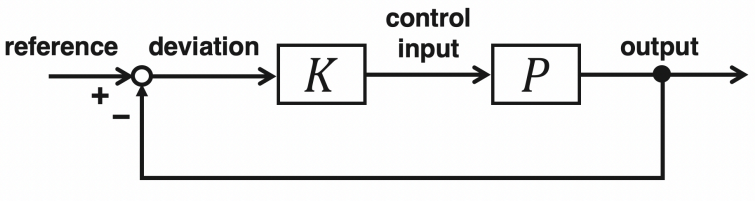

Figure 1.2: Block diagram

Feedback control simply means "looking at the current situation and considering the next control input".

This concept is illustrated in Figure 1.2. For instance, in suspension control, P represents the suspension mass, and K denotes the controller. The actuator signal corresponds to the control input, the mass position (or posture) to the output, and the desired position (or posture) to the reference. The difference between the reference and the output is referred to as the deviation. Most control systems in KAGRA, and control engineering broadly, feed back information that is negative to the output (negative feedback). This approach is adopted to reduce the deviation between the target value and the actual output.

Basically, the role of feedback control can be divided into the following three broad categories,

- Stabilization
- Reference tracking
- Disturbance suppression

1.2.2 Control system design concept

In feedback control, how to determine the control input is very important. In other words, it is necessary to consider what kind of control method to use and how to determine the

control parameters in accordance with the control target, which is control design.

The control design process is shown in Figure 1.3. First, a mathematical model is constructed that accurately represents (to some degree) the characteristics of the plant. Then, this model is used to determine the characteristics of the plant (e.g., how it behaves when a certain control input is applied). This allows us to determine the control input so that the plant behaves according to the reference.

The next step is to design a control law that determines the control inputs. This is to change the characteristics of the control target to desirable ones. Since "desirable" here depends on the control objective, the control law is designed to satisfy the control specification that formulates the control objective.

The last step is to implement it as the controller in a digital system, for example.

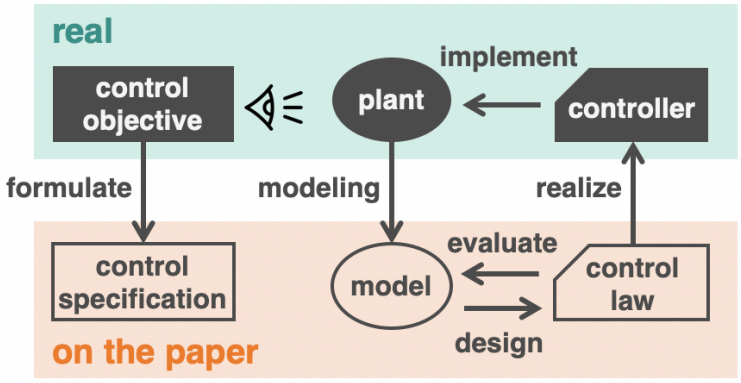

Figure 1.3: Control design

This is a bit abstract, so let's consider a concrete example. Consider the ball shown in Figure 1.1. For simplicity, the mass is assumed to be 1 ($m = 1$), and let $x = 0$ for point B.

Then the equation of motion is

$$\ddot{x}(t) = u(t). \tag{1.2}$$

Now consider feedback control

$$u(t) = -k_1 x(t) - k_2 \dot{x}(t), \tag{1.3}$$

which uses information about the ball's position and velocity to determine the force $u(t)$. Then the problem of determining the force on the ball boils down to the problem of designing k_1 and k_2.

Since the motion of the ball when using control law $u(t) = -k_1 x(t) - k_2 \dot{x}(t)$ is expressed as

$$\ddot{x}(t) + k_2 \dot{x}(t) + k_1 x(t) = 0. \tag{1.4}$$

This differential equation is solved to

$$x(t) = C_1 e^{\lambda_1 t} + C_2 e^{\lambda_2 t}, \tag{1.5}$$

where C_1 and C_2 are values determined from the initial position $x(0)$ and initial velocity $\dot{x}(0)$, and λ_1, λ_2 are the roots of the characteristic polynomial

$$p^2 + k_2 p + k_1 = 0. \tag{1.6}$$

Equation 1.5 is the expression for the behavior of the ball.

Now, the control objective here is to reach reference $x = 0$, so $x(t) \to 0$ $(t \to \infty)$ is all that is needed. Here, looking at Equation 1.5, we see that $x(t) \to 0$ $(t \to \infty)$ is achieved if λ_1 and λ_2 are negative. Also, the smaller they are, the faster they converge to 0. We can determine k_1 and k_2 by taking the above into account.

In this way, instead of vaguely searching for the control input $u(t)$, we can limit the form of $u(t)$ as a design condition, and then return to the problem of determining the parameters contained in $u(t)$. The parameters that achieve the control objective are then determined and a control law is created. After that, the control law is implemented as a controller, and the control objective is achieved.

1.3 Preparation of Python

1.3.1 Library

This subsection lists the Python libraries used in this document. If they are not installed on your machine, just install them with conda or pip and so on. The usage of Python is outside the scope of this document.

- Numpy
- Matplotlib
- Scipy
- Python-Control

1.3.2 Functions

The Python code described in Chapter 2 onward uses the following functions without specific mention, so if the code is compiled as is, it must be defined in your Python environment.

```
# for graph
def plot_set(fig_ax, *args):
    fig_ax.set_xlabel(args[0])
    fig_ax.set_ylabel(args[1])
    fig_ax.grid(ls=':')

    if len(args)==3:
        fig_ax.legend(loc=args[2])
```

Listing 1.1: For Graph

```
# for bode plot
def bodeplot_set(fig_ax, *args):
    fig_ax[0].grid(which='both',ls=':')
    fig_ax[0].set_ylabel('Gain [dB]')
    fig_ax[1].grid(which='both',ls=':')
    fig_ax[1].set_xlabel('$\\omega$ [rad/s]')
    fig_ax[1].set_ylabel('Phase [deg]')

    if len(args)>0:
        fig_ax[1].legend(loc=args[0])
    if len(args)>1:
        fig_ax[0].legend(loc=args[1])
```

Listing 1.2: For Bode Plot

Chapter 2

MODELING

In this chapter, we briefly summarize the representation by differential equations, transfer functions, state space equations, and block diagrams as basic items for modeling plant.

2.1 Representation of dynamic systems

Consider representing the general form of a dynamical system (a system in which past outputs are related to current inputs).

In a dynamic system in which input is u and output is y, if the output $y(t)$ at time t is determined by the input and output up to time t, the system is described by the following differential equation.

$$\begin{aligned} &\frac{\mathrm{d}^n}{\mathrm{d}t^n}y(t) + a_{n-1}\frac{\mathrm{d}^{n-1}}{\mathrm{d}t^{n-1}}y(t) + \cdots a_1\frac{\mathrm{d}}{\mathrm{d}t}y(t) + a_0 y(t) \\ &= b_m\frac{\mathrm{d}^m}{\mathrm{d}t^m}u(t) + b_{m-1}\frac{\mathrm{d}^{m-1}}{\mathrm{d}t^{m-1}}u(t) + \cdots b_1\frac{\mathrm{d}}{\mathrm{d}t}u(t) + b_0 u(t) \end{aligned} \tag{2.1}$$

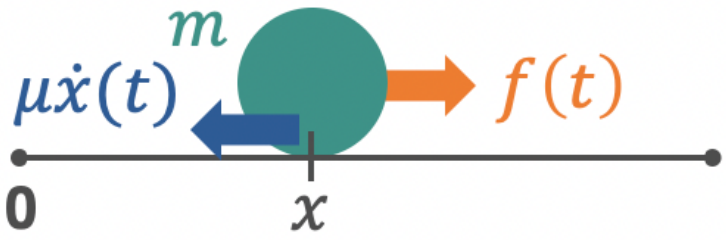

Figure 2.1: Ball movement

For example, consider moving a ball as shown in Figure 1.1. Now consider including friction (Fig. 2.1), and the equations of motion are as follows.

$$m\ddot{x}(t) = f(t) - \mu\dot{x}(t) \tag{2.2}$$

Here, we assume velocity as output $y(t) = \dot{x}(t)$, and force as input $u(t) = f(t)$, then we get shape of Eq 2.1

$$m\dot{y}(t) = u(t) - \mu y(t), \tag{2.3}$$

$$m\dot{y}(t) + \mu y(t) = u(t). \tag{2.4}$$

To know the behavior of the plant, we can find the solution of such a differential equation. However, the more complex the plant is, the more difficult it becomes to solve the differential equation because it is multi-level and higher-order. In such cases, even if the solution can be obtained, it is often not easy to analyze the behavior.

Therefore, instead of treating differential equations as they are, they are converted into a transfer function model in which the system is represented as a complex function, or a state space model in which the system is represented as a vector-valued first-order differential equation. The following sections summarize these models.

2.2 Transfer function

2.2.1 What is transfer function?

The transfer function is obtained by performing a Laplace transform

$$g(s) = \mathcal{L}[g(t)] \coloneqq \int_0^\infty g(\tau)e^{-s\tau}\mathrm{d}\tau, \tag{2.5}$$

on both sides of the differential equation expressed by the equation 2.1 with an initial value of 0 and taking the ratio of the input to the output.

Transfer Function

$$\mathcal{P}(s) = \frac{y(s)}{u(s)} = \frac{b_m s^m + b_{m-1}s^{m-1} + \cdots + b_1 s + b_0}{s^n + a_{n-1}s^{n-1} + \cdots + a_1 s + a_0}, \tag{2.6}$$

where $y(s) = \mathcal{L}[y(t)]$ and $u(s) = \mathcal{L}[u(t)]$.

TF $\mathcal{P}(s)$ represents the relation between system input u and output y, and system output is shown $y(s) = \mathcal{P}(s)u(s)$.

For example, let's consider the transfer dunction model 2.1. Since the differential equation for the ball is expressed by the equation 2.4, Laplace transforming it with an initial value of 0, i.e., $y(0) = 0$ and $\dot{y}(0) = 0$ gives

$$msy(s) + \mu y(s) = u(s), \tag{2.7}$$

$$(ms + \mu)y(s) = u(s). \tag{2.8}$$

So we finally get

$$\mathcal{P}(s) = \frac{y(s)}{u(s)} = \frac{1}{ms + \mu}. \tag{2.9}$$

2.2.2 Properness

In the transfer function model, it is called **strictly proper** when degree of the denominator polynomial d is greater than the degree of the numerator polynomial n ($d > n$). Also, if $d \geq n$, it is called **proper**, and if $d < n$, we call it **improper** (or non-proper).

Simply put, a proper system can be implemented in reality, while an improper system cannot be implemented in reality. We explain this a little more.

The transfer function was obtained by Laplace transformation of the differential equation 2.1. From this, if the order of the numerator of the transfer function is greater than that of the denominator, the time derivative of the input u will be included in the output y. For example, consider an improparative system

$$y(s) = \frac{s^2 + s + 1}{s + 1}u(s), \tag{2.10}$$

this can be transformed like

$$y(s) = su(s) + \frac{1}{s + 1}u(s). \tag{2.11}$$

In this equation, the first term on the right-hand side is the cause of improperness. Here, as mentioned in the previous subsection 2.2.1, the operation of multiplying by s in the s-domain is the operation of differential in the time domain. However, differential is defined

$$\dot{f}(t) \coloneqq \lim_{\Delta t \to 0} \frac{f(t + \Delta t) - f(t)}{\Delta t}, \tag{2.12}$$

and this indicates we need the future information $f(t + \Delta t)$. Unfortunately, it seems to be impossible to implement **exact** time differential in reality. I have not hear exciting news which tells someone succeed to know his or her future correctly, at least at the time I am writing this document. Anyway, this is the reason why we can't implement improper transfer function in real.

Nevertheless, there is no need to consider, for example, whether some transfer function measured in KAGRA is proper or improper. In the case of a measured transfer function, it is a real physical phenomenon, and therefore it is definitely proper.

On the other hand, if you design the controller for suspension or interferometer control in a digital system, there is the case where it is not porper, so you should care the degree of the denominator and numerator polynomials of the transfer function to be sure that it is proper.

By the way, the “D” in PID control, which appears in Chapter 4, means derivative. This means that it includes the derivative of the error (input to the controller), which makes it impossible to implement strictly. However, PID control is widely used in practice. This is because the derivative element is replaced by “a non-exact but almost derivative element” (e.g., an approximate calculation such as numerical differentiation by a program). Of course, this is not an exact derivative because it contains errors, but for practical use, it is almost always sufficient.

2.2.3 Python script

When writing transfer functions in Python, we can use `tf(num, den)`. Alternatively, the transfer function can be written after defining the variable s. If you want to extract the coefficients of the denominator and numerator polynomials, you can use methods such as `P.num` or `P.den`. In this description, the elements can be extracted using `tfdata` because the nested structure of the list makes it difficult to use in other programs.

```
# TF
from control.matlab import tf, tfdata

Np = [0, 1]  # Numerator polynomial (coefficient)
Dp = [1,2,3] # Denominator polynomial (coefficient)

P = tf(Np,Dp)

P1 = tf([0,1],[1,2,3]) # same as P

s = tf('s')
P2 = 1 / (s**2 + 2*s + 3) # same as P

print('P=',P)
print('P1=',P1)
print('P2=',P2)

print('denP =',P.den)          # example of extraction (array)

[[numP]],[[denP]] = tfdata(P) # example of extraction
print('denP =',denP)
```

Listing 2.1: Transfer Function

2.3 State-space equation

2.3.1 What is state-space equation?

The state-space equation is a first-order differential equation that uses matrices to represent multiple higher-order differential equations, and is very useful for describing multi-input and multi-output systems. It defines state variables and describes the relationship between input → state → output.

State-space equation

$$\dot{x}(t) = \boldsymbol{A}\boldsymbol{x}(t) + \boldsymbol{B}\boldsymbol{u}(t) \tag{2.13}$$

$$\boldsymbol{y}(t) = \boldsymbol{C}\boldsymbol{x}(t) + \boldsymbol{D}\boldsymbol{u}(t) \tag{2.14}$$

,where $\boldsymbol{x}$ is state variable, $\boldsymbol{u}$ is input、 $\boldsymbol{y}$ is output, and $\boldsymbol{A}$, $\boldsymbol{B}$, $\boldsymbol{C}$, $\boldsymbol{D}$ are constant matrices, and the equation 2.13 is called the state equation and 2.14 is called the output equation.

Here, the number of elements in the state variable $\boldsymbol{x}$ is the same as the number of initial conditions needed when solving the original differential equation. However, there are degrees of freedom in how the state variables are chosen. Therefore, there are an infinite number of state space models.

As an example of the state-space equation, consider the case of 2.1 as in last subsection 2.2.1. In this case, the equation of motion was $m\ddot{x}(t) + \mu\dot{x}(t) = f(t)$. So when we assume

$$\boldsymbol{x}(t) = \begin{bmatrix} x(t) \\ \dot{x}(t) \end{bmatrix}, \ u(t) = f(t), \ y(t) = z(t), \tag{2.15}$$

input equation becomes

$$\begin{aligned} \dot{\boldsymbol{x}} = \begin{bmatrix} \dot{x}(t) \\ \ddot{x}(t) \end{bmatrix} &= \begin{bmatrix} \dot{x}(t) \\ -\frac{\mu}{m}\dot{x}(t) + \frac{1}{m}u(t) \end{bmatrix} \\ &= \begin{bmatrix} 0 & 1 \\ 0 & -\frac{\mu}{m} \end{bmatrix} \begin{bmatrix} x(t) \\ \dot{x}(t) \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{1}{m} \end{bmatrix} u(t) \\ &= \begin{bmatrix} 0 & 1 \\ 0 & -\frac{\mu}{m} \end{bmatrix} \boldsymbol{x}(t) + \begin{bmatrix} 0 \\ \frac{1}{m} \end{bmatrix} u(t). \end{aligned} \tag{2.16}$$

On the other hand, output equation is

$$y(t) = \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} x(t) \\ \dot{x}(t) \end{bmatrix} = \begin{bmatrix} 1 & 0 \end{bmatrix} \boldsymbol{x}(t). \tag{2.17}$$

So we get

$$\boldsymbol{A} = \begin{bmatrix} 0 & 1 \\ 0 & -\frac{\mu}{m} \end{bmatrix}, \ \boldsymbol{B} = \begin{bmatrix} 0 \\ -\frac{1}{m} \end{bmatrix}, \ \boldsymbol{C} = \begin{bmatrix} 1 & 0 \end{bmatrix}, \ \boldsymbol{D} = 0. \tag{2.18}$$

In the following, unless otherwise mentioned, we consider a one-input, one-output system ($\boldsymbol{y} = y, \ \boldsymbol{u} = u$).

2.3.2 Derivation of the ss equation

Just in case, we note the derivation of the state-space equations 2.13 and 2.14. We assume $p = \frac{\mathrm{d}}{\mathrm{d}t}$ and

$$\begin{cases} A(p) = p^n + a_{n-1}p^{n-1} + \cdots + a_1 p + a_0 \\ B(p) = b_m p^m + b_{m-1}p^{m-1} + \cdots + b_1 p + b_0 \end{cases} \tag{2.19}$$

then equation 2.1 will be $A(p)y = B(p)u$. Here, we introduce new variance v and divide

$$A(p)v = u,\ \ y = B(p)v. \tag{2.20}$$

If we define n-order vector $\boldsymbol{x}$ as

$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \coloneqq \begin{bmatrix} v \\ pv \\ \vdots \\ p^{n-1}v \end{bmatrix}, \tag{2.21}$$

we get

$$\dot{\boldsymbol{x}} = p\boldsymbol{x} = \begin{bmatrix} pv \\ p^2v \\ \vdots \\ p^nv \end{bmatrix} = \begin{bmatrix} x_2 \\ x_3 \\ \vdots \\ -a_0x_1 - a_1x_2 - \cdots - a_{n-1}x_n + u \end{bmatrix}, \tag{2.22}$$

and

$$y = b_0v + b_1pv + \cdots + b_mp^mv = b_0x_1 + b_1x_2 + \cdots + b_mx_{m+1}. \tag{2.23}$$

If we rewrite these using the matrix representation,

$$\begin{cases} \dot{\boldsymbol{x}} = A\boldsymbol{x} + \boldsymbol{B}u \\ y = \boldsymbol{C}\boldsymbol{x} \end{cases} \tag{2.24}$$

where

$$\boldsymbol{A} = \begin{bmatrix} 0 & 1 & 0 & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & \ddots & \ddots & 0 \\ 0 & \cdots & \cdots & 0 & 1 \\ -a_0 & -a_1 & \cdots & \cdots & -a_{n-1} \end{bmatrix},\ \boldsymbol{B} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 0 \\ 1 \end{bmatrix},\ \boldsymbol{C} = \begin{bmatrix} b_0 & \cdots & b_m & 0 & \cdots & 0 \end{bmatrix}. \tag{2.25}$$

Note that this is the case where $m < n$, and $\boldsymbol{D}u$ is added where $m = n$.

2.3.3 Python script

When we describe state-space in Python, we use `ss(A, B, C, D)`. If you want to extract matrices $\boldsymbol{A}$, $\boldsymbol{B}$, $\boldsymbol{C}$, $\boldsymbol{D}$, just use `ssdata`.

```
# State Space
from control.matlab import ss, ssdata

A = [[0,1],[-1,-1]]
B = [[0],[1]]
C = [1,0]
D = [0]
P = ss(A,B,C,D)

```

```
sysA, sysB, sysC, sysD = ssdata(P) # example of extraction

print(P)
print("A=",sysA)
```

Listing 2.2: State-Space Equation

2.4 Relationship between TF and SS

2.4.1 Transformation between TF and SS

As we have seen, the differential equation 2.1 describing a dynamical system can be transformed into a transfer function or a state space model. In this section, we will discuss the transformations between these two representation.

First, by Laplace transformation both sides of the state-space equation $\dot{\boldsymbol{x}} = \boldsymbol{A}\boldsymbol{x} + \boldsymbol{B}u$ with an initial value of 0, we obtain $s\boldsymbol{x}(s) = \boldsymbol{A}\boldsymbol{x}(s) + \boldsymbol{B}u(s)$, which is $(s\boldsymbol{I} - \boldsymbol{A})\boldsymbol{x}(s) = \boldsymbol{B}u(s)$, that is

$$\boldsymbol{x}(s) = (s\boldsymbol{I} - \boldsymbol{A})^{-1}\boldsymbol{B}u(s) \tag{2.26}$$

Moreover, since $y(s) = \boldsymbol{C}\boldsymbol{x}(s) + \boldsymbol{D}u(s)$,

$$\mathcal{P}(s) = \boldsymbol{C}(s\boldsymbol{I} - \boldsymbol{A})^{-1}\boldsymbol{B} + \boldsymbol{D}. \tag{2.27}$$

Like this, for a state-space, which is defined by matrices $\boldsymbol{A}$, $\boldsymbol{B}$, $\boldsymbol{C}$, $\boldsymbol{D}$, the transfer function from input u to output y can be uniquely determined.

On the other hand, **The transformation from transfer function to state-space is not unique**. This is because there are countless state-space models, depending on how the state variables are defined. For example, for a regular matrix $\boldsymbol{T} \in \mathbb{R}^{n\times n}$, given the coordinate transformation $\bar{\boldsymbol{x}} = \boldsymbol{T}\boldsymbol{x}$, we obtain a state-space model

$$\mathcal{P} : \begin{cases} \dot{\bar{\boldsymbol{x}}}(t) = \bar{A}\bar{\boldsymbol{x}}(t) + \bar{\boldsymbol{B}}u(t) \\ \boldsymbol{y} = \bar{\boldsymbol{C}}\bar{\boldsymbol{x}} + \boldsymbol{D}u(t) \end{cases} \tag{2.28}$$

$$\bar{\boldsymbol{A}} = \boldsymbol{T}\boldsymbol{A}\boldsymbol{T}^{-1},\ \bar{\boldsymbol{B}} = \boldsymbol{T}\boldsymbol{B},\ \bar{\boldsymbol{C}} = \boldsymbol{C}\boldsymbol{T}^{-1} \tag{2.29}$$

with a new state variable $\bar{\boldsymbol{x}}$. This is called an equivalent transformation.

Therefore, controllable and observable canonical forms are used to convert transfer functions to state space, which will be described in Chapter 4 (maybe).

2.4.2 Python script

We use `tf2ss` or `ss2tf` to realize these transformation in Python. The controllability and observability in the code below will be discussed in detail in Chapter 4.

```python
# conversion
from control.matlab import tf, ss, tf2ss, ss2tf
from control import canonical_form

P = tf([0,1],[1,1,1])

Pss = tf2ss(P)   # TF to SS
Ptf = ss2tf(Pss) # SS to TF

Pr, Tr = canonical_form(Pss,form='reachable')  # reachable (=controllable if linear continuous-time system)
Po, To = canonical_form(Pss,form='observable') # observable

print(Pr)
print(Po)
```

Listing 2.3: Transformation between TF and SS

2.5 Block diagram

A block diagram is often used to describe a system. For example, a block diagram of a system $\mathcal{S}$ ($y = \mathcal{S}u$) with input u and output y is shown in Figure 2.2. In this case, the direction of the arrow corresponds to the direction of the signal flow.

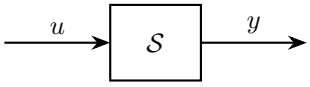

Figure 2.2: Block diagram

2.5.1 Series, parallel and feedback

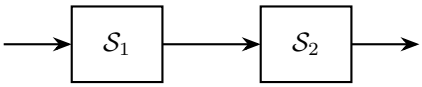

Figure 2.3: Series coupling

When two systems are arranged in series and the output of one system is combined to become the input of the other system as shown in Figure 2.3, this is called series coupling. For example, if $y = \mathcal{S}_1 u$ and $z = \mathcal{S}_2 y$ are series-coupled, $z = \mathcal{S}_2 \cdot \mathcal{S}_1 u$, so the entire system is

$$\mathcal{S} = \mathcal{S}_2 \cdot \mathcal{S}_1. \tag{2.30}$$

If $\mathcal{S}_1$ and $\mathcal{S}_2$ are linear and one-input and one-output, they can be reordered to form

$$\mathcal{S} = \mathcal{S}_1 \cdot \mathcal{S}_2, \tag{2.31}$$

but not if they are nonlinear or multi-input and multi-output.

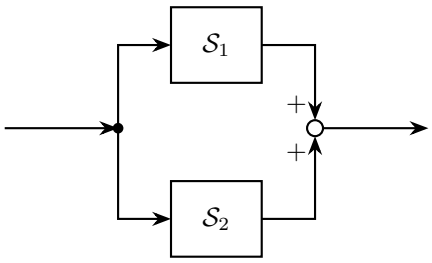

Figure 2.4: Parallel coupling

When two systems are placed in parallel and are coupled so that their outputs are added together with the inputs in common as shown in Figure 2.4, it is called parallel coupling. The black circles are draw points, indicating that the original and drawn signals are the same. On the other hand, the white circle is the additive point, which means that the two signals are added together. Here, $y_1 = \mathcal{S}_1 u$ and $y_2 = \mathcal{S}_2 u$ are combined in parallel, and y_1 and y_2 are added together, so $y = y_1 + y_2 = \mathcal{S}_1 u + \mathcal{S}_2 u = (\mathcal{S}_1 + \mathcal{S}_2)u$. Therefore, the entire system is

$$\mathcal{S} = \mathcal{S}_1 + \mathcal{S}_2. \tag{2.32}$$

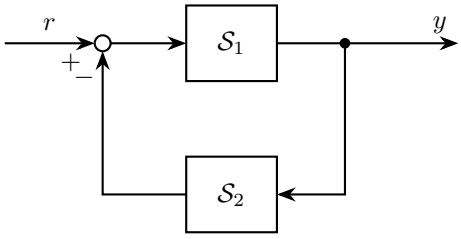

Figure 2.5: Feedback coupling

Furthermore, when two systems are placed side by side and their outputs are combined to become inputs to each other as shown in Figure 2.5, it is called feedback coupling. In the case of Figure 2.5,

$$y = \mathcal{S}_1 u = \mathcal{S}_1(r - z) = \mathcal{S}_1(r - \mathcal{S}_2 y) = \mathcal{S}_1 r - \mathcal{S}_1 \mathcal{S}_2 y \tag{2.33}$$

$$(1 + \mathcal{S}_1 \mathcal{S}_2) y = \mathcal{S}_1 r, \tag{2.34}$$

so we get

$$y = \frac{\mathcal{S}_1}{(1 + \mathcal{S}_1 \mathcal{S}_2)} r. \tag{2.35}$$

Therefore, the whole system becomes

$$\mathcal{S} = \frac{\mathcal{S}_1}{(1 + \mathcal{S}_1 \mathcal{S}_2)}. \tag{2.36}$$

2.5.2 Python script

The Python scripts for series coupling, parallel coupling, and feedback coupling are as follows. Note that since Python does not automatically commute, function `minreal` is used to make it an irreducible transfer function.

```
# Block Diagram
from control.matlab import tf, series, parallel, feedback

S1 = tf([0,1],[1,1])
S2 = tf([1,1],[1,1,1])

# series
Ss = series(S1, S2) # same as S = S2 * S1
print('Ss=', Ss.minreal()) #  irreducible form

# parallel
Sp = parallel(S1, S2) # same as S = S2 + S1
print('Sp=', Sp.minreal()) #  irreducible form

# feedback
Sf = feedback(S1, S2) # same as S = S1 / (1+ S1 * S2)
S = S1 / (1+ S1*S2)
print('Sf=', Sf.minreal()) #  irreducible form
```

Listing 2.4: Block Diagram

Chapter 3

BEHAVIOR OF PLANT

In this chapter, we will discuss how to examine the characteristics of a plant by adding inputs and observing the outputs. In this chapter and thereafter, the functions described in 1.3.2 are used.

3.1 Time response

In the following, we mainly consider the behavior of the output when a step input

$$u(t) = \begin{cases} 1 & (t \geq 0) \\ 0 & (t < 0) \end{cases} \tag{3.1}$$

is added (step response), and also consider first-order and second-order lag systems (most actual plant are higher-order lag systems, but can be approximated by first-order and second-order lag systems).

Also, as you can see in the following list, we can calculate the step response by using the function `step` in the form `y,t=step(sys,Td)` and so on. Note that `sys` and `Td` represent the transfer function or state-space model and simulation time respectively, and the model output response `y` and time `t` are the return values.

3.1.1 First-order lag system

As for ball movement shown Figure 2.1, we know its TF is

$$\mathcal{P} = \frac{1}{ms + \mu} = \frac{\frac{1}{\mu}}{1 + \frac{m}{\mu}s}. \tag{3.2}$$

Here, we assume $K = \frac{1}{\mu}$ and $T = \frac{m}{\mu}$, then we get

$$\mathcal{P}(s) = \frac{K}{1+Ts}. \tag{3.3}$$

A system expressed in this form is called **first-order lag system**, where K is called gain and T is called time constant. In particular, the time constant is a parameter that determines the quick response (speed of response).

3.1.1.1 Python script

For example, if we want to calculate the step response when $T = 0.5,\ K = 1$, we compile the code below, and get Figure 3.1.

```
# First Order Lag System
import matplotlib.pyplot as plt
import numpy as np
from control.matlab import tf, step

T, K = 0.5, 1       # time constant & gain
P1 = tf([0,K],[T,1]) # 1st order lag system

y, t = step(P1, np.arange(0,5,0.01)) # step response

fig, ax = plt.subplots()
ax.plot(t,y)
plot_set(ax,'t','y')

#plt.savefig('1st_order_lag',dpi=300)
```

Listing 3.1: Step response of 1st-order lag system

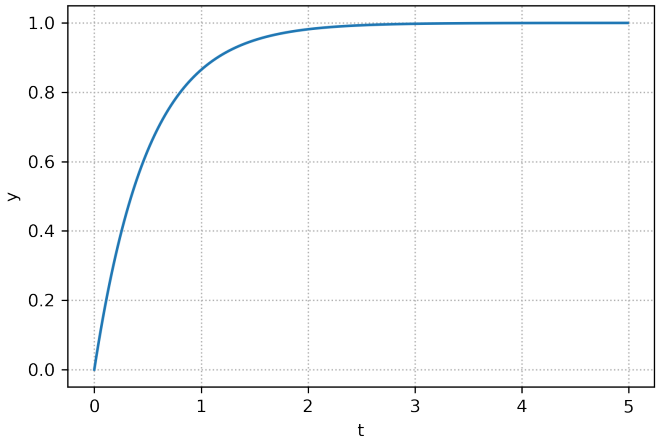

Figure 3.1: Step response of 1st-order lag system

Thus, the output starts from an initial value of 0 and gradually increases, reaching 1.0 in about 3 seconds. Also, $y = 0.632$ at $t = 0.5$, and this time is the time constant (the time when the output reaches 63.2% of its steady-state value).

In addition, to examine how the response changes when the time constant T is changed, the following code is executed to obtain Figure 3.2. This shows that the response becomes faster when the time constant is decreased.

```
# First Order Lag System - change T
fig, ax = plt.subplots()

Tp = [1,0.5,0.1] # change T

for i in range(len(Tp)):
    y,t = step(tf([0,K],[Tp[i],1]),np.arange(0,5,0.01))
    ax.plot(t,y,label=f'T={Tp[i]}')

plot_set(ax,'t','y','best')

#plt.savefig('1st_order_lag_changeT',dpi=300)
```

Listing 3.2: Step response of 1st-order lag system（change T）

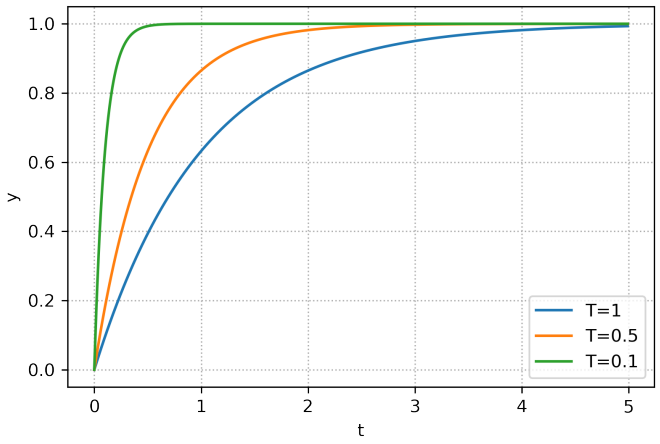

Figure 3.2: Step response of 1st-order lag system（change T）

On the other hand, when the following code with different gain K is executed, Figure 3.3 is obtained. From this, it can be seen that the steady-state value increases as the gain is increased.

```
# First Order Lag System - change K
fig, ax = plt.subplots()

Kp = [1,2,3] # change K

for i in range(len(Kp)):
    y,t = step(tf([0,Kp[i]],[T,1]),np.arange(0,5,0.01))
    ax.plot(t,y,label=f'K={Kp[i]}')

plot_set(ax,'t','y','best')

#plt.savefig('1st_order_lag_changeK',dpi=300)
```

Listing 3.3: 1st-order lag system（change K）

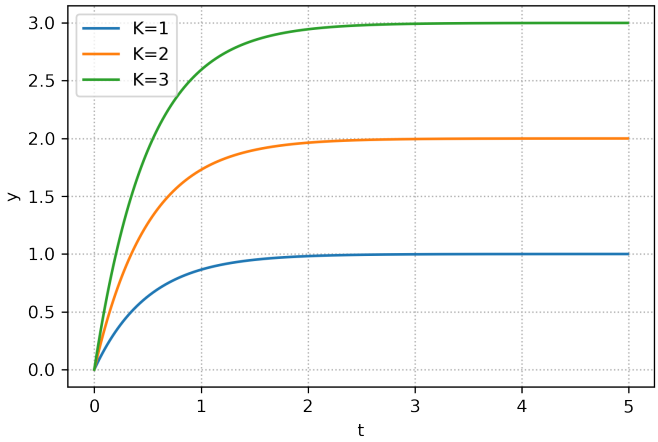

Figure 3.3: 1st-order lag system（change K）

To explain the above in relation to the motion of a ball, when a certain force is applied to a ball, the ball that starts moving will eventually move at a certain speed. At this time, the ball moves faster when its mass is smaller, corresponding to the smaller value of $T = \frac{m}{\mu}$. Also, the smaller the friction, the longer it takes for the ball to settle at a certain speed, and the greater the final speed (Figure 3.3). This corresponds to a larger $T = \frac{m}{\mu}$ and a larger $K = \frac{1}{\mu}$.

3.1.1.2 calculation of time response

The following is an explanation of the calculations performed in the Python code above. The output $y(s)$ of the plant $\mathcal{P}$ is represented by $y(s) = \mathcal{P}u(s)$. Considering the step input, $u(s) = \frac{1}{s}$, so the output is $y(s) = \frac{\mathcal{P}(s)}{s}$. Therefore, the inverse Laplace transform of this to obtain the time response is

$$y(t) = \mathcal{L}^{-1}[y(s)] = \mathcal{L}^{-1}\left[\mathcal{P}(s)\frac{1}{s}\right]. \tag{3.4}$$

For 1st-order lag system, since the output is

$$y(s) = \frac{K}{1+Ts}\frac{1}{s} = K\left(\frac{1}{s} - \frac{T}{1+Ts}\right) = K\left(\frac{1}{s} - \frac{1}{s+\frac{1}{T}},\right) \tag{3.5}$$

so the inverse Laplace transform of this is

$$y(t) = K\left(1 + e^{-\frac{1}{T}t}\right). \tag{3.6}$$

The plot of this is Figure 3.1 in next subsection.

3.1.2 Second-order lag system

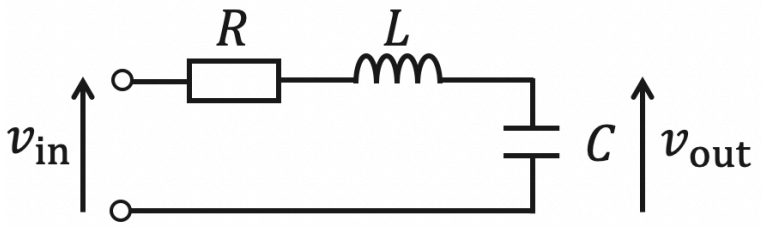

Figure 3.4: RLC circuits

As an example of a second-order lag system, consider the RLC circuit shown in Figure 3.4. Let v_{in} be the voltage applied to the circuit, $i(t)$ be the current flowing in the circuit, and $v_{\text{out}}(t)$ be the voltage at both ends of the capacitor, which is

$$v_{\text{in}}(t) = L\frac{\mathrm{d}}{\mathrm{d}t}i(t) + Ri(t) + \frac{1}{C}\int_0^t i(\tau)\mathrm{d}\tau, \tag{3.7}$$

from Ohm's law. If the output is

$$y(t) = v_{\text{out}}(t) = \frac{1}{C}\int_0^t i(\tau)\mathrm{d}\tau, \tag{3.8}$$

and the input is

$$u(t) = v_{\text{in}}(t), \tag{3.9}$$

then

$$LC\ddot{y}(t) + RC\dot{y}(t) + y(t) = u(t), \tag{3.10}$$

since $C\dot{y} = i(t)$. Laplace transforming this is

$$CLs^2y(s) + CRsy(s) + y(s) = u(s), \tag{3.11}$$

so TF from input to output is

$$\mathcal{P}(s) = \frac{y(s)}{u(s)} = \frac{1}{CLs^2 + CRs + 1}. \tag{3.12}$$

This can be transformed into

$$\mathcal{P}(s) = \frac{1}{CLs^2 + CRs + 1} = \frac{\frac{1}{CL}}{s^2 + \frac{R}{L}s + \frac{1}{CL}}, \tag{3.13}$$

and if we set

$$K = 1,\ \omega_n = \sqrt{\frac{1}{CL}},\ \zeta = \frac{R}{s}\sqrt{\frac{C}{L}} = \frac{R}{2L\omega_n}, \tag{3.14}$$

we get

$$\mathcal{P}(s) = \frac{K\omega_n^2}{s^2 + 2\zeta\omega_n s + \omega_n^2} \tag{3.15}$$

The system expressed in this form is called **second-order lag system**, where ζ is the damping coefficient and ω_n is the natural angular frequency.

3.1.2.1 Python script

If you want to examine the step response of a second-order lag system in Python, you can use the following code. Also, when you run this code, you will get Figure 3.5.

```
# Second Order Lag System
from control.matlab import tf, step

zeta, omega_n = 0.4, 5 # & attenuation coefficient eigen angular frequency
P2 = tf([0,omega_n**2],[1,2*zeta*omega_n,omega_n**2])
y, t = step(P2,np.arange(0,5,0.01))

fig, ax = plt.subplots()
ax.plot(t,y)
plot_set(ax,'t','y')

#plt.savefig('2nd_order_lag',dpi=300)
```

Listing 3.4: Step response of 2nd-order lag system

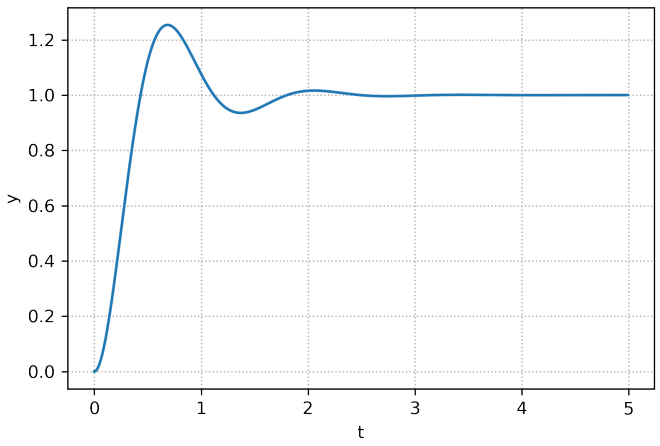

Figure 3.5: Step response of 2nd-order lag system

Thus, the output starts with an initial value of 0, gradually increases, takes a maximum value of $y_{\max}$ around $T = 0.685$, and then converges to 1. The difference between the maximum value and the final value ($1.25 - 1 = 0.25$ in this case) is called overshoot. Overshoot does not occur in the first-order lag system, but it may occur in some cases in the second-order lag system.

In addition, to examine how the behavior changes when the damping coefficient ζ changes, the following code is executed to obtain Figure 3.6.

```
# Second Order Lag System - change zeta
fig, ax = plt.subplots()

zetap = [1,0.7,0.4,0,-0.05] # change zeta
```

```
5
6  for i in range(len(zetap)):
7      P2 = tf([0,omega_n**2],[1,2*zetap[i]*omega_n,omega_n**2])
8      y,t = step(P2,np.arange(0,5,0.01))
9
10     ax.plot(t,y,label=f'$\zeta$={zetap[i]}')
11
12 ax.set_ylim(-1, 3)
13
14 plot_set(ax,'t','y','best')
15
16 #plt.savefig('2nd_order_lag_changeZeta',dpi=300)
```

Listing 3.5: Step response of 2nd-order lag system (change ζ)

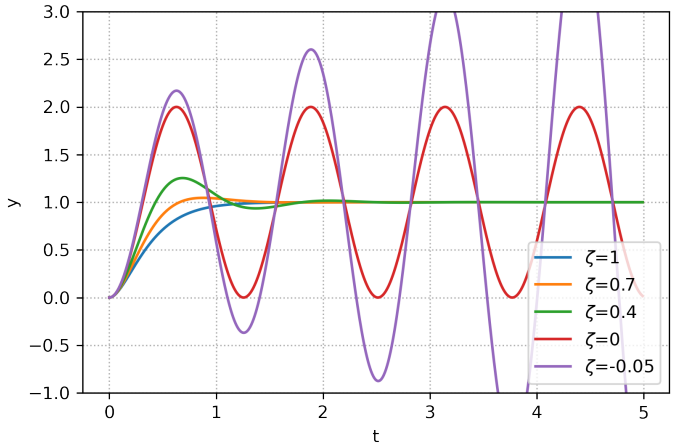

Figure 3.6: Step response of 2nd-order lag system (change ζ)

This indicates that the magnitude of the overshoot increases as the value of ζ decreases.

Furthermore, when $\zeta = 0$, it continues to oscillate and does not converge to a constant value, and when $\zeta = 0.05$, it is seen to diverge. Thus, ζ is a parameter that determines the damping property, converging without oscillation when $\zeta \geq 1$, converging with oscillation when $0 < \zeta < 1$, and diverging when $\zeta < 0$.

Next, considering the case where the natural angular frequency is varied, the following code is executed to obtain Figure 3.7.

```
# Second Order Lag System - change omega
fig, ax = plt.subplots()

omega_np = [1,5,10] # change omega

for i in range(len(omega_np)):
    P2 = tf([0,omega_np[i]**2],[1,2*zeta*omega_np[i],omega_np[i]**2])
    y,t = step(P2,np.arange(0,5,0.01))

    ax.plot(t,y,label=f'$\omega$={omega_np[i]}')

plot_set(ax,'t','y','best')

#plt.savefig('2nd_order_lag_changeOmega',dpi=300)
```

Listing 3.6: Step response of 2nd-order lag system (change ω_n)

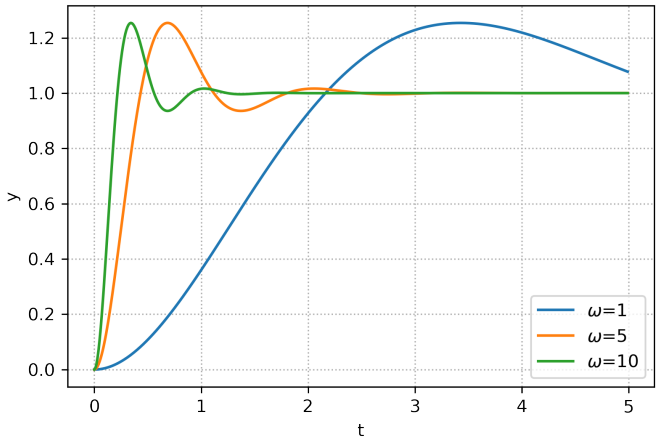

Figure 3.7: Step response of 2nd-order lag system (change ω_n)

From this, it can be seen that the response becomes faster as ω_n is increased. In other words, the natural angular frequency ω_n is a parameter that determines the quick response in the same way as the time constant T in the case of a first-order delay system.

To explain the above in relation to the RLC circuit, when a certain voltage is applied to the RLC circuit, current flows through the circuit and charge accumulates in the capacitor. Gradually, the current stops flowing and the voltage at both ends of the capacitor eventually converges to 1. However, when a coil is present, electromagnetic induction generates a counter electromotive force (in a direction that prevents the current from increasing or decreasing), and the greater the inductance of the coil, the stronger the magnitude of the electromotive force. As a result, the increase in the voltage at both ends of the capacitor may be delayed, or the voltage may temporarily exceed the steady-state value of 1. Here, since

$$\omega_n = \frac{1}{\sqrt{CL}}, \tag{3.16}$$

the delay of increase in the voltage at both ends of the capacitor corresponds to a slower rise as L increases. Also, since

$$\zeta = \frac{R\sqrt{C}}{2\sqrt{L}}, \tag{3.17}$$

the fact that the voltage temporarily exceeds the steady-state value of 1 corresponds to the fact that the response becomes more oscillatory as L increases.

3.1.2.2 Calculation of time response

Here, we explain the calculation performed in the Python code above: the step response of a second-order lag system is

$$y(s) = \frac{K\omega_n^2}{s^2 + 2\omega_n s + \omega_n^2}\frac{1}{s} = \frac{K\omega_n^2}{s(s+\omega_n)^2}, \tag{3.18}$$

when $\zeta = 1$, which, when partially fractionalized, becomes

$$y(s) = K\left(\frac{1}{s} - \frac{1}{s+\omega_n} - \frac{\omega_n}{(s+\omega_n)^2}\right). \tag{3.19}$$

By inverse Laplace transforming this, we can get

$$y(t) = K\left(1 - e^{-\omega_n t} - \omega_n t e^{-\omega_n t}\right). \tag{3.20}$$

From this we see that $y(0) = 0,\ y(\infty) = K$ (Figure 3.5). Also, since $y(t)$ approaches K exponentially, no overshoot occurs at this time.

On the other hand, when $\zeta \neq 1$, we can get

$$y(s) = \frac{K\omega_n^2}{s^2 + 2\zeta\omega_n s + \omega_n^2}\frac{1}{s} = \frac{K\omega_n^2}{s(s-p_1)(s-p_2)}, \tag{3.21}$$

where $p_1,\ p_2 = (-\zeta \pm \sqrt{\zeta^2 - 1})\omega_n$. This can be decomposed into partial fractions

$$y(s) = \frac{K\omega_n^2}{p_1 p_2}\left(\frac{1}{s} + \frac{p_2}{(p_1 - p_2)} - \frac{p_1}{(p_1 - p_2)(s - p_2)}\right) \tag{3.22}$$

so if we perdform inverse Laplace transformation, we get

$$y(t) = \frac{K\omega_n^2}{p_1 p_2}\left(1 + \frac{p_2}{p_1 - p_2}e^{p_1 t} - \frac{p_1}{p_1 - p_2}e^{p_2 t}\right). \tag{3.23}$$

Moreover, submitting $p_1 p_2 = \omega_n^2,\ p_1 - p_2 = 2\omega_n\sqrt{\zeta^2 - 1}$, we get

$$y(t) = K\left(1 - \frac{\zeta + \sqrt{\zeta^2 - 1}}{2\sqrt{\zeta^2 - 1}}e^{p_1 t} + \frac{\zeta - \sqrt{\zeta^2 - 1}}{2\sqrt{\zeta^2 - 1}}e^{p_2 t}\right). \tag{3.24}$$

Of these, $p_1, p_2 = (-\zeta \pm \sqrt{\zeta^2 - 1})\omega_n$ becomes negative real numbers when $\zeta > 1$, so $y(t)$ approaches exponentially to $y(\infty) = K$ and no overshoot occurs.

Since $p_1, p_2 = -\zeta\omega_n \pm j\omega_n\sqrt{1 - \zeta^2})$ when $\zeta < |1|$, Euler's formula $e^{j\theta} = \cos\theta + j\sin\theta$

$$y(t) = K\left(1 - e^{-\zeta\omega_n t}\cos\bar{\omega}_n t - \frac{1}{\sqrt{1-\zeta^2}}e^{-\zeta\omega_n t}\sin\bar{\omega}_n t\right), \tag{3.25}$$

where $\bar{\omega}_n = \omega_n\sqrt{1 - \zeta^2}$. In this case, the cos, sin causes oscillatory behavior. We can also see that the exponential part of the function eventually converges to K since it becomes zero over time.

Moreover, calculating the time derivative to find the maximum value of y, namely $y_{\max}$, is

$$\dot{y}(t) = \frac{K\omega_n}{\sqrt{1-\zeta^2}} e^{-\zeta\omega_n t} \sin \bar{\omega}_n t, \tag{3.26}$$

so we know that y takes the maximum value when

$$t = T_p = \frac{\pi}{\bar{\omega}_n} = \frac{\pi}{\omega_n\sqrt{1-\zeta^2}}. \tag{3.27}$$

The value at this time is

$$y_{\max} = y(T_p) = K(1 + e^{-\zeta\omega_n T_P}), \tag{3.28}$$

so the overshoot is

$$y_{\max} - y(\infty) = Ke^{-\zeta\omega_n T_P}. \tag{3.29}$$

At last, when $\zeta = 0$,

$$y(t) = K(1 - \cos\omega_n t). \tag{3.30}$$

Since the exponential element is eliminated, it becomes a sustained oscillation with amplitude K.

3.1.3 Time response in ss model

Let us consider the time response in a state-space model. Since the state-space model can take into account the effect of initial values on output, consider the behavior of $\dot{\boldsymbol{x}}(t) = \boldsymbol{A}\boldsymbol{x}(t)$ with input $u = 0$.

To obtain the initial value response, use the `initial` function, such as `x, t = initial(sys, Td, X0)`. Among the arguments, `sys, Td` is the same as in the `step` function, and `X0` is the initial state. This may be omitted if the initial state is 0. The return value `x,t` represents the response and time of the state, respectively.

Now, for the system given in

$$\boldsymbol{A} = \begin{bmatrix} 0 & 1 \\ -4 & -5 \end{bmatrix}, \ \boldsymbol{B} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \ \boldsymbol{C} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \ \boldsymbol{D} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \tag{3.31}$$

the following code is executed to examine the response, resulting in Figure 3.8.

```
# Response in SS model
import matplotlib.pyplot as plt
import numpy as np
from control.matlab import ss, step, initial, lsim

A = [[0,1],[-4,-5]]
B = [[0],[1]]
C = np.eye(2)
D = np.zeros([2,1])
P = ss(A, B, C, D)

Td = np.arange(0,5,0.01)
X0 = [-0.3,0.4]
x,t = initial(P,Td,X0)
```

```
15
16 fig, ax = plt.subplots()
17 ax.plot(t,x[:,0], label='$x_1$')
18 ax.plot(t,x[:,1], label='$x_2$')
19 plot_set(ax,'t','x','best')
20
21 #plt.savefig('response_in_SSmodel',dpi=300)
```

Listing 3.7: Time responce in ss model (initial response)

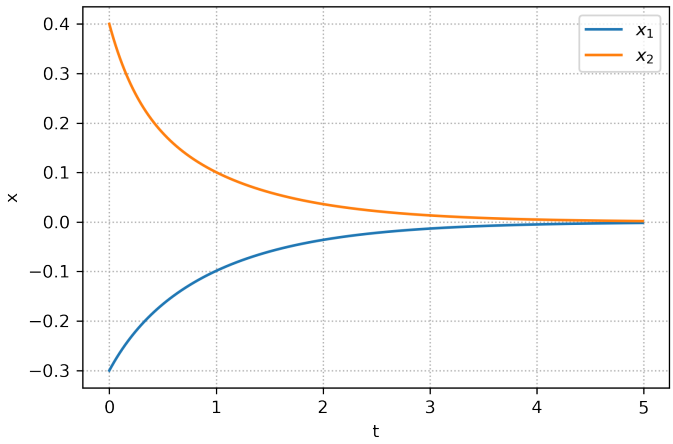

Figure 3.8: Time responce in ss model (initial response)

The two lines are there because state $\boldsymbol{x} = \begin{bmatrix} x_1 & x_2 \end{bmatrix}^\top$ is two-dimensional, starting from their respective initial values and eventually converging to 0 in both cases.

Since we now assume $u = 0$ for the input, the differential equation is $\dot{\boldsymbol{x}}(t) = \boldsymbol{A}\boldsymbol{x}(t)$, and we can find this solution. Then, Laplace transforming both sides of this equation yields

$$s\boldsymbol{x}(s) - \boldsymbol{x}(0) = \boldsymbol{A}\boldsymbol{x}(s). \tag{3.32}$$

Therefore, since it is

$$\boldsymbol{x}(s) = (s\boldsymbol{I} - \boldsymbol{A})^{-1}\boldsymbol{x}(0), \tag{3.33}$$

it becomes

$$\boldsymbol{x}(t) = \mathcal{L}^{-1}\left[(s\boldsymbol{I} - \boldsymbol{A})^{-1}\right]\boldsymbol{x}(0). \tag{3.34}$$

However, $\boldsymbol{I}$ is a unit matrix. If we rewrite this equation using the state transition matrix (matrix exponential function) $e^{\boldsymbol{A}t}$, the solution of the state equation with zero input is

$$\boldsymbol{x}(t) = e^{\boldsymbol{A}t}\boldsymbol{x}(0). \tag{3.35}$$

The state transition matrix can be calculated by hand, but that is tedious, so it can be obtained using Python as follows.

```
1 # Calculation of State Transition Matrix
2 import sympy as sp
3 import numpy as np
```

```
from scipy.linalg import expm

sp.init_printing()
s = sp.Symbol('s')
t = sp.Symbol('t', positive=True)

A = np.array([[0,1],[-4,-5]])

G = s*sp.eye(2)-A

exp_At = sp.inverse_laplace_transform(sp.simplify(G.inv()), s, t)

t = 5     # when you want to know the value at certain time
expm(A*5)
```

Listing 3.8: Calculation of state transition matrix

Next, consider the case where there is an input. In this case, the equation of state is $\dot{\boldsymbol{x}}(t) = \boldsymbol{A}\boldsymbol{x}(t) + \boldsymbol{B}u(t)$. Let $\boldsymbol{x}(t) = e^{\boldsymbol{A}t}\boldsymbol{z}(t)$ be a candidate solution ($\boldsymbol{z}(0) = \boldsymbol{x}(0)$). Here, the time derivative of $\boldsymbol{x}(t) = e^{\boldsymbol{A}t}\boldsymbol{z}(t)$ is

$$\dot{\boldsymbol{x}}(t) = \boldsymbol{A}e^{\boldsymbol{A}t}\boldsymbol{z}(t) + e^{\boldsymbol{A}t}\dot{\boldsymbol{z}}(t) = \boldsymbol{A}\boldsymbol{x}(t) + e^{\boldsymbol{A}t}\dot{\boldsymbol{z}}(t). \tag{3.36}$$

Therefore, to make this agree with the original differential equation, we only need to find $\boldsymbol{z}(t)$ that satisfies

$$e^{\boldsymbol{A}t}\dot{\boldsymbol{z}}(t) = \boldsymbol{B}u(t). \tag{3.37}$$

Therefore, integrating both sides of

$$\dot{\boldsymbol{z}}(t) = e^{-\boldsymbol{A}t}\boldsymbol{B}u(t), \tag{3.38}$$

yields

$$\boldsymbol{z}(t) = \boldsymbol{z}(0) + \int_0^t e^{-\boldsymbol{A}\tau}\boldsymbol{B}u(\tau)\mathrm{d}\tau. \tag{3.39}$$

Therefore, the solution of the equation of state with input is

$$\boldsymbol{x}(t) = e^{\boldsymbol{A}t}\boldsymbol{z}(t) = e^{\boldsymbol{A}t}\boldsymbol{x}(0) + \int_0^t e^{\boldsymbol{A}(t-\tau)}\boldsymbol{B}u(\tau)\mathrm{d}\tau. \tag{3.40}$$

The first term on the right side of this equation is called the zero input response and the second term is called the zero state response. For example, if the initial values are $\boldsymbol{x}(0) = \begin{bmatrix}0 & 0\end{bmatrix}^\top$ and the input is $u(t) = 1$ $(t \geq 0)$ (step response), the response (zero state response)

$$\boldsymbol{x}(t) = \int_0^t e^{\boldsymbol{A}(t-\tau)}\boldsymbol{B}\mathrm{d}\tau, \tag{3.41}$$

is considered and the following code is executed to obtain Figure 3.9.

```
# Zero State Response in SS model
Td = np.arange(0,5,0.01)
x, t = step(P, Td)

fig, ax = plt.subplots()
ax.plot(t,x[:,0], label='$x_1$')
ax.plot(t,x[:,1], label='$x_2$')
plot_set(ax,'t','x','best')

#plt.savefig('zero_state_responce',dpi=300)
```

Listing 3.9: zero state response

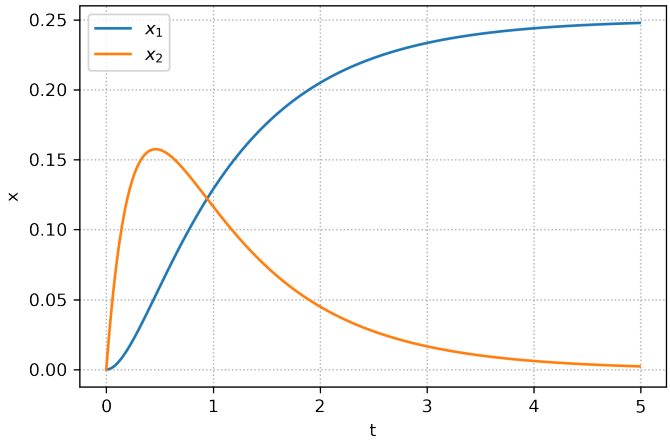

Figure 3.9: zero state response

If we want to check the time response of the expression 3.40, you can use the `lsim` function as shown in the code below, and obtain the figure 3.10 when executed.

```
# Time Responce of SS model
Ud = 1*(Td>=0) # step input
X0 = [-0.3, 0.4]

xst, t = step(P, Td)         # zero state response
xin, _ = initial(P, Td, X0) # zero input response
x, _, _ = lsim(P, Ud, Td, X0)

fig, ax = plt.subplots(1,2, figsize=(9, 3.4))
for i in [0, 1]:
    ax[i].plot(t, x[:,i], label='response')
    ax[i].plot(t, xst[:,i], label='zero state')
    ax[i].plot(t, xin[:,i], label='zero input')
    ax[i].set_ylim(-0.4, 0.6)


plot_set(ax[0], 't', '$x_1$')
plot_set(ax[1], 't', '$x_2$', 'best')
fig.tight_layout()

#plt.savefig('time_response_of_SSmodel',dpi=300)
```

Listing 3.10: Time response in ss model

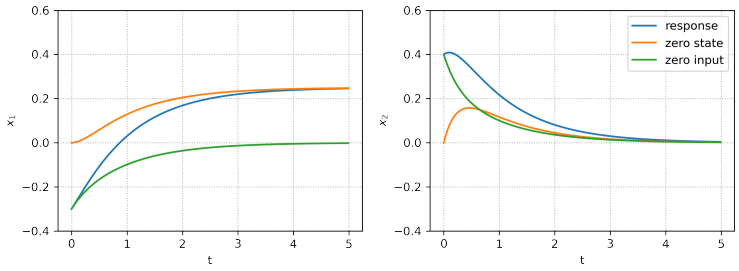

Figure 3.10: Time response in ss model

3.2 Stability and system behavior

3.2.1 Stability

When examining the step response of first-order and second-order lag systems, the output response sometimes diverged depending on how the parameters were chosen. This means that the system is unstable. In this section, we discuss the stability of the system in terms of input-output stability and asymptotic stability.

3.2.1.1 Input-output stability

Input-output stability (or Bounded-Input-Bounded-Output: BIBO stability) means that when a bounded signal is used as input, the output is also bounded. Note that a bounded signal, defined as

$$|u(t)| \leq M < \infty, \ \forall t, \ \exists M > 0, \tag{3.42}$$

, is a signal that does not diverge to infinity.

If this stability is not maintained, that is, if the system becomes unstable, it can be found by examining the poles of the transfer function (the roots of the denominator polynomial that $\mathcal{P}(s) = \infty$). For example, the following code shows the poles of each transfer function.

```
# Poles
from control.matlab import tf, tfdata

P1 = tf([0,1],[1,1])
P2 = tf([0,1],[-1,1])

for i in [P1, P2]:
    print('poles:',i.poles())
```

Listing 3.11: example of tf for stability

Then, the poles of $\mathcal{P}_1$ and $\mathcal{P}_2$ are $-1 + 0j$ and $+1 + 0j$, respectively. Of these, $\mathcal{P}_1$, whose real part of the pole is negative, is stable, while $\mathcal{P}_2$, whose real part of the pole is positive, is unstable. In general, it is known that

Poles and stability

The necessary and sufficient condition for a system to be input-output stable is **The real parts of all the poles of the master function are negative**. In other words, a system is input-output stable if the poles of the transfer function are in the left half-plane. A pole with a negative real part is called a stable pole, and one without a negative real part is called an unstable pole.

3.2.1.2 Asymptotic stability

In the previous subsection, we discussed the stability of the transfer function. On the other hand, the stability of the state-space model can be obtained by examining the eigenvalues

of the $\boldsymbol{A}$ matrix of the system.

$\boldsymbol{A}$matrix and stability

A necessary and sufficient condition for the system to be stable is that **the real parts of all eigenvalues of matrix $\boldsymbol{A}$ are negative**.

Stability here is defined as

$$\lim_{t\to\infty} \boldsymbol{x}(t) = 0, \tag{3.43}$$

which is called asymptotic stability. In particular, if the state-space model is asymptotically stable, the output is bounded for bounded inputs (input-output stable). However, depending on the properties of the matrices $\boldsymbol{B}, \boldsymbol{C}$, the transfer function $\boldsymbol{C}(s\boldsymbol{I}-\boldsymbol{A})^{-1}\boldsymbol{B}$ may have a common factor between the numerator and denominator polynomials, resulting in an irreducible division.

The fact that the system $\dot{\boldsymbol{x}} = \boldsymbol{A}\boldsymbol{x}$ is asymptotically stable is equivalent to the fact that for any matrix $\boldsymbol{Q} = \boldsymbol{Q}^\top > 0$, there exists a unique matrix $\boldsymbol{P} = \boldsymbol{P}^\top > 0$ satisfying Lyapunov's equation

$$\boldsymbol{P}\boldsymbol{A} + \boldsymbol{A}^\top\boldsymbol{P} = -\boldsymbol{Q}. \tag{3.44}$$

This is called Lyapunov's stability theorem. For example, executing

```
# Lyapunov Stable
from control.matlab import lyap

A = np.array([[0,1],[-4,-5]])

Q = np.eye(2)
P = lyap(A.T, Q)

np.linalg.eigvals(P)
```

Listing 3.12: Lyapunov's stability theorem

yields `array([1.1403882, 0.1096118])`, so $\boldsymbol{P}$ is positively defined, and from this we know that the system is asymptotically stable. Note that `lyap(M, Q)` is a function to find the solution of $\boldsymbol{M}\boldsymbol{P} + \boldsymbol{P}\boldsymbol{M}^\top = -\boldsymbol{Q}$.

Another way to visually show how the convergence is achieved when starting from arbitrary initial values is the phase portrait shown in Figure 3.11. This is obtained by the following code.

```
# Phase Portrait
import matplotlib.pyplot as plt
import numpy as np

w = 1.5 # step size
Y, X = np.mgrid[-w:w:100j, -w:w:100j]

A = np.array([[0,1],[-4,-5]])
s,v = np.linalg.eig(A) # eigen vector and eigen values

U = A[0,0]*X + A[0,1]*Y # calculation for component of xdot
V = A[1,0]*X + A[1,1]*Y # calculation for component of xdot

t = np.arange(-1.5,1.5,0.01)

fig, ax = plt.subplots()
```

```
17
18 ax.set_ylim(-1.5, 1.5)
19
20 ax.streamplot(X, Y, U, V, density=0.7, color='k')
21
22 if s.imag[0] == 0 and s.imag[1] == 0:
23     ax.plot(t, (v[1,0]/v[0,0])*t)
24     ax.plot(t, (v[1,1]/v[0,1])*t)
25
26 plot_set(ax, '$x_1$', '$x_2$')
27
28 #plt.savefig('phase_portrait',dpi=300)
```

Listing 3.13: Phase portrait

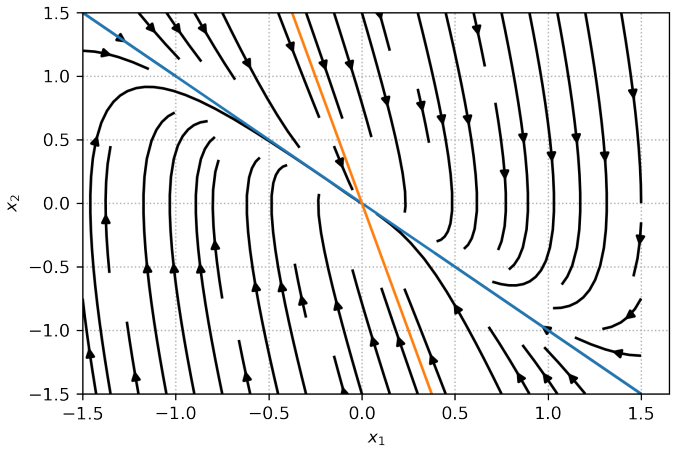

Figure 3.11: Phase portrait

In this figure, the blue and orange lines plot the eigenvectors corresponding to the eigenvalues -1 and -4 of the matrix $\boldsymbol{A}$, respectively. Eigenvectors are those that satisfy $\dot{\boldsymbol{x}} = \boldsymbol{A}\boldsymbol{x} = \lambda\boldsymbol{x}$, so once on the straight line, the state transitions along this line (called the invariant space).

3.2.2 Relationship between poles and system behavior

Having seen the relationship between the poles of the transfer function and the eigenvalues of the $\boldsymbol{A}$ matrix of the state-space model and stability, we now consider the relationship between them and the behavior of the system.

First, the further the pole is to the left of the left half-plane, i.e., to the negative side, the faster the response. For example, in the case of a first-order lag system, the pole is $s = -\frac{1}{T}$, corresponding to the fact that the smaller T is, the faster the response is. In the case of a second-order lag system, the pole is $s = -\zeta\omega_n \pm \omega_n\sqrt{\zeta^2 - 1}$, corresponding to a faster response when ω_n is larger.

Next, when the imaginary part of the pole is non-zero, it becomes oscillatory, and the larger the imaginary part, the faster the oscillation. For example, in the case of a first-order lag system, the poles are not oscillatory because they have only real parts. In the case of a second-order lag system, when the absolute value of ζ is less than 1, since $s = -\zeta\omega_n \pm j\omega_n\sqrt{1-\zeta^2}$, the closer ζ is to 0, the larger the imaginary part becomes and the more oscillatory it becomes.

3.3 Frequency response

When the transfer function is $\mathcal{P}(s)$, the output of the controlled object is expressed as $y(s) = \mathcal{P}(s)u(s)$, so the response of the input signal with $u(s) = 1$ is the transfer function. This signal $u(s) = 1$ is called an impulse input (this is a superfunction called Dirac's delta function $\delta(t)$, which has a magnitude of ∞ when $t = 0$ and is 0 at other times). In other words, the Laplace transform of the response when an impulse input is added is called the transfer function.

Therefore, if we want to examine the characteristics of a plant, you can add an impulse input and examine the response. In reality, however, it is difficult to add an impulse input. It is very difficult to add an input of infinite magnitude for a single moment. Therefore, the impulse input is expressed as a linear sum of different signals. For example, it can be viewed as a collection of multiple sine or cosine waves of different frequencies (mathematically speaking, this corresponds to a Fourier transform). The following code examines the output response when a sinusoidal signal is added.

```
# Response for sin input
import matplotlib.pyplot as plt
import numpy as np
from control.matlab import tf, lsim

fig, ax = plt.subplots(2,2,figsize=(9, 5))

zeta = 0.7
omega_n = 5
P = tf([0,omega_n**2],[1,2*zeta*omega_n,omega_n**2]) # second order lag system

freq = [2, 5, 10, 20]
Td = np.arange(0, 5, 0.01)

for i in range(2):
    for j in range(2):
        u = np.sin(freq[2*i+j]*Td)
        y, t, _ = lsim(P, u, Td, 0)

        ax[i,j].plot(t, u, label='u')
        ax[i,j].plot(t, y, label='y')
        plot_set(ax[i,j], 't', 'u, y')

ax[0,0].legend()

#plt.savefig('response_to_sin_input',dpi=300)
```

Listing 3.14: Response to sine wave input

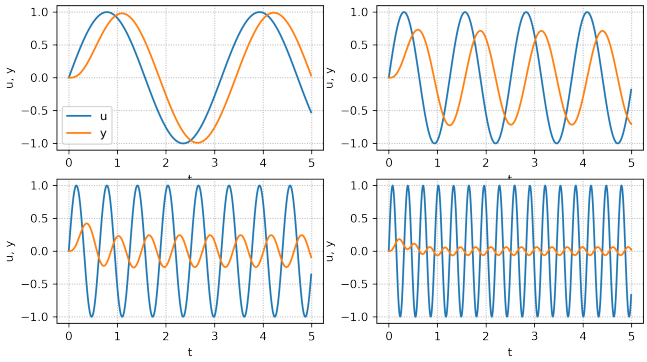

Figure 3.12: Response to sine wave input

From this, it can be seen that when a sinusoidal signal is added, the output also becomes a sinusoidal signal in steady state. Also, when the frequency of the sine wave is low, the amplitude of the input and output are almost the same (high gain), and as the frequency is increased, the amplitude becomes smaller (low gain). Furthermore, we can also see that the phase is delayed.

In summary, when the input signal is $u(t) = A\sin(\omega t)$, the output is $y(t) = B(\omega)\sin(\omega t + \phi(\omega))$ and the amplitude ratio $\frac{B(\omega)}{A}$ and phase $\phi(\omega)$ depend on the frequency.

Therefore, as shown in Figure 3.13, the gain (amplitude ratio of input/output signals) and phase are plotted against frequency to visually express the characteristics of the plant. For each frequency, the gain diagram shows the amplitude ratio in dB as $20\log_{10}\frac{B(\omega)}{A}$, and the phase diagram plots the phase [deg]. As described in the following section, the Bode diagram can be drawn in Python using the `bode` function.

Here we explain how the Bode diagram is drawn, i.e., the calculation of the frequency response. For example, when a sine wave $u(t) = \sin(\omega t)$ with an amplitude of 1 is input to a stable system, the output after a sufficient time is $y(t) = B(\omega)\sin(\omega t + \phi(\omega))$. In this case, $\mathcal{P}(j\omega)$ expressed as

$$\mathcal{P}(j\omega) = B(\omega)e^{j\phi(\omega)}, \tag{3.45}$$

using the amplitude $B(\omega)$ and phase ϕ is called the frequency transfer function. This is obtained by substituting $j\omega$ for s in the transfer function $\mathcal{P}(s)$ ($\mathcal{P}(s)$ and $\mathcal{P}(j\omega)$ are the Laplace transform and Fourier transform of the impulse response respectively). Since $\mathcal{P}(j\omega)$ is a complex function, when $\mathcal{P}(j\omega) = \alpha(\omega) + j\beta(\omega)$, its absolute value and declination angle are

$$|\mathcal{P}(j\omega)| = \sqrt{\alpha(\omega)^2 + \beta(\omega)^2},\ \angle\mathcal{P}(j\omega) = \tan^{-1}\left(\frac{\beta(\omega)}{\alpha(\omega)}\right). \tag{3.46}$$

The Bode diagram is a vector locus drawn by calculating these values when ω is varied.

3.3.1 First-order lag system

To draw a Bode diagram for a first-order lag system when the time constant T is varied, the following code is executed to obtain Fig. 3.13.

```
# Bode Plot for 1st order lag system
from control.matlab import tf, bode, logspace, mag2db

fig, ax = plt.subplots(2,1,figsize=(5, 5))

K = 1
T = [1, 0.5, 0.1]

for i in range(len(T)):
    P = tf([0,K],[T[i],1])
    mag, phase, w = bode(P, logspace(-2,2), plot=False)

    pltargs = {'label':f'T={T[i]}'}
    ax[0].semilogx(w, mag2db(mag), **pltargs)
    ax[1].semilogx(w, np.rad2deg(phase), **pltargs)

bodeplot_set(ax, 3, 3)

#plt.savefig('bodeplot_1st_order_lag',dpi=300)
```

Listing 3.15: Bode plot of first-order lag system

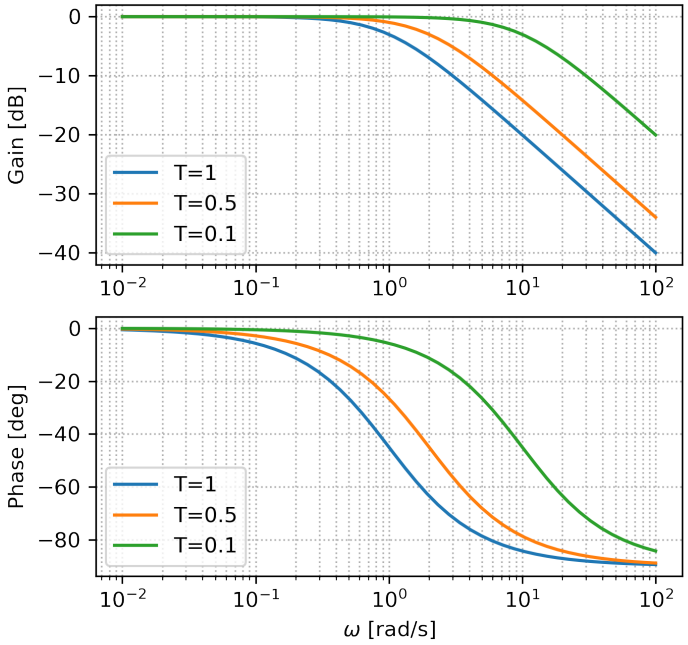

Figure 3.13: Bode plot of first-order lag system

The gain plot shows that the gain is around 0 dB in the low frequency range and decreases as the frequency increases. In other words, when the frequency of the input signal is low,

the amplitude of the output signal is almost equal to the amplitude of the input signal, and when the frequency is high, the amplitude of the output signal is small. Also, as the time constant decreases, the frequency at which the gain decreases increases.

The frequency transfer function of the first-order delay system is

$$\mathcal{P}(j\omega) = \frac{1}{1 + j\omega T}, \tag{3.47}$$

when $K = 1$. In this case, the gain and phase are

$$|\mathcal{P}(j\omega)| = \frac{1}{\sqrt{1 + (\omega T)^2}} \tag{3.48}$$

$$\angle\mathcal{P}(j\omega) = -\tan^{-1}\omega T \tag{3.49}$$

This mean that $|\mathcal{P}(j\omega)| = 1$, $\angle\mathcal{P}(j\omega) = 0$ at low frequency (small ω). This is graphed in Figure 3.13.

3.3.2 Second-order lag system

To draw Bode plots for a second-order lag system when the damping constant ζ and eigenangular frequency ω_n are varied, the following code is executed to obtain Figures 3.14 and 3.15.

```
# Bode Plot for 2nd order lag system
from control.matlab import tf, bode, logspace, mag2db

fig, ax = plt.subplots(2,1,figsize=(5, 5))

zeta = [1,0.7,0.4]
omega_n = 1

for i in range(len(zeta)):
    P = tf([0,omega_n**2],[1,2*zeta[i]*omega_n, omega_n**2])
    mag, phase, w = bode(P, logspace(-2,2), plot=False)

    pltargs = {'label':f'$\\zeta$={zeta[i]}'}
    ax[0].semilogx(w, mag2db(mag), **pltargs)
    ax[1].semilogx(w, np.rad2deg(phase), **pltargs)

bodeplot_set(ax, 3, 3)

#plt.savefig('bodeplot_2nd_order_lag_zeta',dpi=300)

fig, ax2 = plt.subplots(2,1,figsize=(5, 5))

zetap = 1
omega_np = [1,5,10]

for i in range(len(omega_np)):
    P = tf([0,omega_np[i]**2],[1,2*zetap*omega_np[i], omega_np[i]**2])
    mag, phase, w = bode(P, logspace(-2,2), plot=False)

    pltargs = {'label':f'$\\omega_n$={omega_np[i]}'}
    ax2[0].semilogx(w, mag2db(mag), **pltargs)
    ax2[1].semilogx(w, np.rad2deg(phase), **pltargs)

bodeplot_set(ax2, 3, 3)
```

```
35
36 #plt.savefig('bodeplot_2nd_order_lag_omega',dpi=300)
```

Listing 3.16: Bode plot of second-order lag system

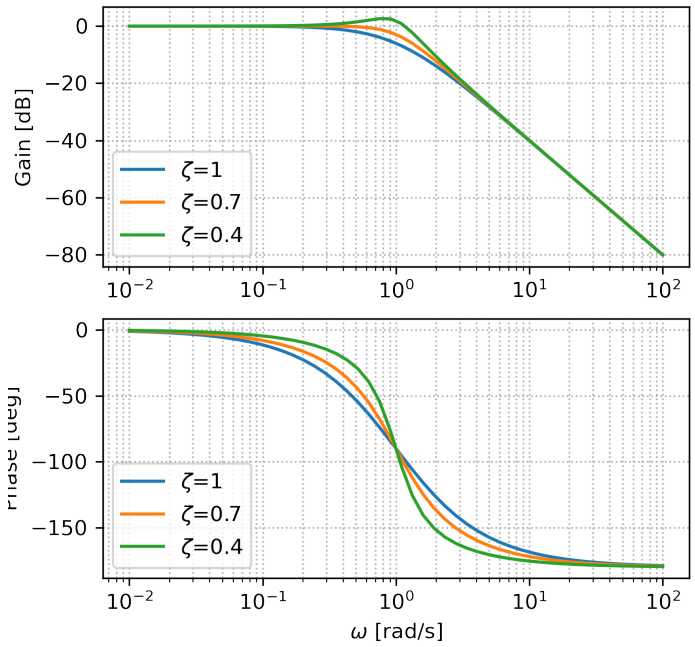

Figure 3.14: Bode plot of first-order lag system (change ζ)

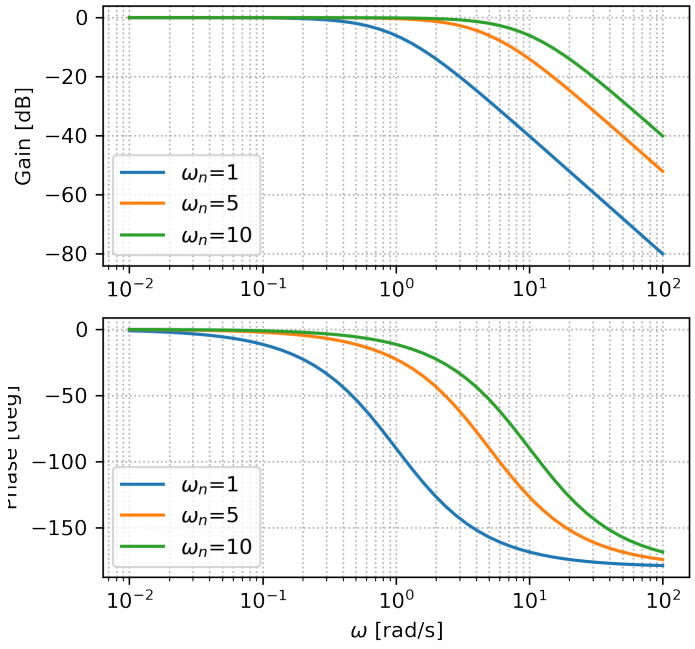

Figure 3.15: Bode plot of first-order lag system (change ω_n)

The figure shows that when ζ is varied, decreasing the value of ζ results in a portion of the signal that is larger than 0 dB. This means that the amplitude of the output signal is larger than that of the input signal, corresponding to the occurrence of overshoot. It can also be seen that as ω_n is decreased, the frequency at which the gain decreases becomes lower.

The frequenct transfer function of second-order lag system is

$$\mathcal{P}(j\omega) = \frac{\omega_n^2}{\omega_n^2 - \omega^2 + 2j\zeta\omega_n\omega}, \tag{3.50}$$

when $K = 1$, so

$$|\mathcal{P}(j\omega)| = \frac{\omega_n^2}{\sqrt{(\omega_n^2 - \omega^2)^2 + (2\zeta\omega_n\omega)^2}}, \tag{3.51}$$

$$\angle\mathcal{P}(j\omega) = -\tan^{-1}\frac{2\zeta\omega_n\omega}{\omega_n^2 - \omega^2}. \tag{3.52}$$

In this case, $|\mathcal{P}(j\omega)| = 1$ $(0\ dB)$ and $\angle\mathcal{P}(j\omega) = 0$ deg when the frequency is low (ω is sufficiently small). Also, when $\omega = \omega_n$, $|\mathcal{P}(j\omega)| = \frac{1}{2\zeta}$ and $\angle\mathcal{P}(j\omega) = -90$ deg. Furthermore, if the frequency is high (ω is large enough), $|\mathcal{P}(j\omega)| = 1/(\omega/\omega_n)^2$ $(0\ dB)$, $\angle\mathcal{P}(j\omega) = -180$ deg. Figures 3.14 and 3.15 show these graphs.

The maximum value of the gain is called the peak gain. Since the frequency is $((\omega_n^2 - \omega^2)^2 + (2\zeta\omega_n\omega)^2$, which is the minimum ω, if we find a solution to

$$\begin{aligned}\frac{\mathrm{d}}{\mathrm{d}\omega}\left\{(\omega_n^2-\omega^2)^2+(2\zeta\omega_n\omega)^2\right\} &= -4\omega(\omega_n^2-\omega_2)+8\zeta^2\omega_n^2\omega \\ &= 4\omega(\omega^2-\omega_n^2(1-2\zeta^2)) \\ &= 0,\end{aligned} \tag{3.53}$$

we get $\omega = 0, \pm\omega_n\sqrt{1-2\zeta^2}$. Therefore, when $0 \leq \zeta < \frac{1}{2}$, the value of $|\mathcal{P}(j\omega)|$ is the peak gain M_p since it takes the minimum value at $\omega = \omega_n\sqrt{1-2\zeta^2}$ and

$$\begin{aligned}|\mathcal{P}(j\omega)| &= \frac{\omega_n^2}{\sqrt{(\omega_n^2-\omega_n^2(1-2\zeta^2))^2+(2\zeta\omega_n^2\sqrt{1-2\zeta^2})^2}} \\ &= \frac{1}{\sqrt{(1-(1-2\zeta^2))^2+4\zeta^2(1-2\zeta^2)}} \\ &= \frac{1}{2\zeta\sqrt{1-\zeta^2}}\end{aligned} \tag{3.54}$$

On the other hand, when $\zeta \geq \frac{1}{\sqrt{2}}$, the minimum value is taken at $\omega = 0$ and $|\mathcal{P}(j\omega)| = 1$. In other words, the peak gain at this time is $M_p = 1$.

Chapter 4

SYSTEM DESIGN (CL)

4.1 Control specification for closed loop

In this chapter, we mainly consider to design closed loop (feedback loop) to plant $\mathcal{P}$ like Figure 4.1. In the following, $\mathcal{K}$ is the controller, r is the reference, u is the control input, d is the disturbance, y is the output, and e is the deviation, unless otherwise mentioned.

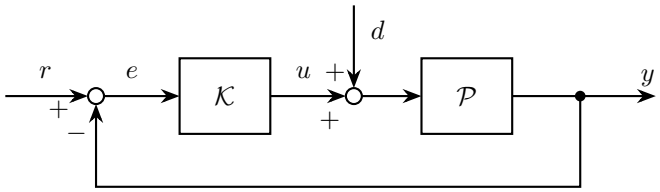

Figure 4.1: Closed loop

In this case, the target is to design $\mathcal{K}$ so that the closed-loop system has desirable characteristics. Here we explain the stability, time response, and frequency response characteristics which is the evaluation of this purpose.

4.1.1 Stability

In a closed-loop system such as the one shown in Figure 4.1, the external inputs are r and d, and the outputs are y and u. Therefore, the stability of the closed-loop system must be considered in four ways: from r to y & u and from d to y & u (since a divergent control input cannot be implemented in reality, the response to u is also considered).

Since

$$y(s) = \mathcal{P}(s)(d(s) + u(s)) = \mathcal{P}(s)d(s) + \mathcal{P}(s)\mathcal{K}(s)(r(s) - y(s)), \tag{4.1}$$

outout y is

$$y(s) = \frac{\mathcal{P}(s)\mathcal{K}(s)}{1 + \mathcal{P}(s)\mathcal{K}(s)} r(s) + \frac{\mathcal{P}(s)}{1 + \mathcal{P}(s)\mathcal{K}(s)} d(s). \tag{4.2}$$

Also, since

$$u(s) = \mathcal{K}(s)(r(s) - y(s)) = \mathcal{K}(s)r(s) - \mathcal{P}(s)\mathcal{K}(s)(d(s) + u(s)), \tag{4.3}$$

control input u is

$$u(s) = \frac{\mathcal{K}(s)}{1 + \mathcal{P}(s)\mathcal{K}(s)} r(s) - \frac{\mathcal{P}(s)\mathcal{K}(s)}{1 + \mathcal{P}(s)\mathcal{K}(s)} d(s). \tag{4.4}$$

So, stability of feedback system is shown below:

Stability

A necessary and sufficient condition for the stability of a feedback control system is that all four transfer functions of

$$\begin{aligned} \mathcal{G}_{yr}(s) &= \frac{\mathcal{P}(s)\mathcal{K}(s)}{1 + \mathcal{P}(s)\mathcal{K}(s)} \\ \mathcal{G}_{yd}(s) &= \frac{\mathcal{P}(s)}{1 + \mathcal{P}(s)\mathcal{K}(s)} \\ \mathcal{G}_{ur}(s) &= \frac{\mathcal{K}(s)}{1 + \mathcal{P}(s)\mathcal{K}(s)} \\ \mathcal{G}_{ud}(s) &= -\frac{\mathcal{P}(s)\mathcal{K}(s)}{1 + \mathcal{P}(s)\mathcal{K}(s)} \end{aligned} \tag{4.5}$$

are input-output stable (3.2.1.1). This is called internal stability.

Also, when

$$\mathcal{P}(s) = \frac{N_{\mathcal{P}}(s)}{D_{\mathcal{P}}(s)}, \ \mathcal{K} = \frac{N_{\mathcal{K}}(s)}{D_{\mathcal{K}}(s)}, \tag{4.6}$$

the denominator polynomial of the four transfer functions is

$$\phi(s) = D_{\mathcal{P}}(s)D_{\mathcal{K}}(s) + N_{\mathcal{P}}(s)N_{\mathcal{K}}(s). \tag{4.7}$$

If the roots of this characteristic polynomial are stable, then the system is internally stable. Note that if there is no unstable pole-zero cancellation (i.e., the unstable pole of the control target cancels the unstable zero of the controller) between $\mathcal{P}(s)$ and $\mathcal{K}(s)$, the closed loop system is internally stable if $\mathcal{G}_{yr}(s)$ is input-output stable.

4.1.2 Time response

When designing a controller to make y reach r as fast as possible, evaluation items include the speed of response (quick response), the attenuation of oscillation (damping), and the magnitude of deviation (steady-state characteristics). In this section, we consider evaluating these in terms of time response.

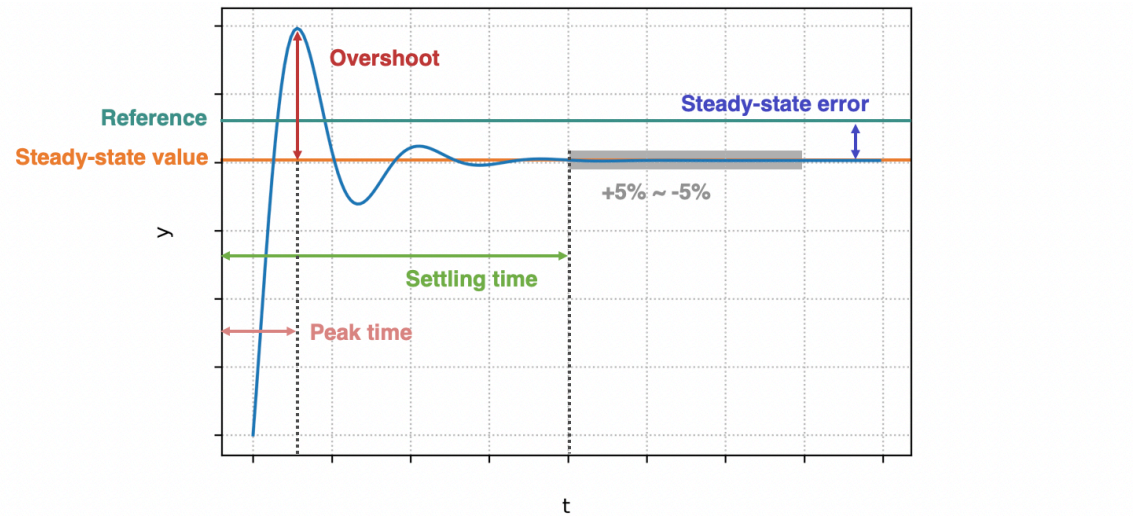

Figure 4.2: Time response

An example of the step response of the transfer function from r to y is shown in Fig. 4.2. The oscillating part is called the transient characteristic, which can be quantitatively evaluated in terms of rise time, settling time, overshoot, and overshoot time. Rise time refers to the time it takes for the step response to reach a value between 10% and 90% of the steady-state value y_∞, and settling time refers to the time it takes for the step response to settle within a certain range of steady-state values (for example, 5% settling time). Overshoot is the difference between the maximum value and the steady-state value when the step response exceeds the steady-state value, which is the value of $y_{\max} - y_\infty$, and overshoot time is the time until overshoot is reached. Of these, the rise time, settling time, and overshoot time are indicators for quick response, while the settling time and overshoot are indicators for attenuation.

In Figure 4.2, the steady-state characteristics are quantitatively evaluated by the steady-state deviation. This represents the difference between the target value and the steady-state value.

In Python, for example, `stepinfo(P, SettlingTimeThreshold=0.05)` gives the values of various performance indicators related to step response.

4.1.3 Frequency response

The quick-response, damping, or steady-state characteristics of a closed-loop system can also be evaluated by the frequency response.

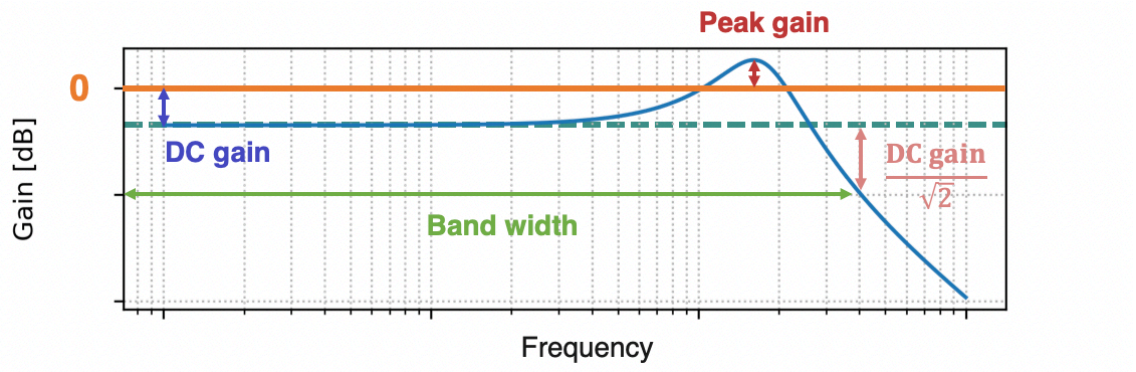

Figure 4.3: Frequency response

An example of a gain diagram for the transfer function from r to y is shown in Figure 4.3. In this diagram, the bandwidth and peak gain are indices for transient response characteristics. **The bandwidth is the frequency at which the gain becomes $\frac{1}{\sqrt{2}}$ (=3 dB lower) of the DC gain**, and the larger the bandwidth, the faster the response.

The peak gain is the maximum amplitude ratio of the input to the output signal,

$$M_p = \max_{\omega \geq 0} |\mathcal{G}(j\omega)|. \tag{4.8}$$

In the case of Figure 4.3, when an input near the peak gain frequency (resonance frequency) is applied, the output is larger than the input. This corresponds to an oscillatory time response, which is an indicator of attenuation.

The DC gain (low-frequency gain of $\omega \to 0$) is an indicator regarding the steady-state characteristics. For example, if the DC gain is 0 dB, the steady-state deviation is 0 in the step response.

4.1.4 Summary

Here, we summarize the control specification.

Stability : keep closed loop stable

Quick response : In the step response of $\mathcal{G}_{yr}$, make rise time and settling time small.
In the gain plot of $\mathcal{G}_{yr}$, make bandwidth ω_{bw} sufficiently large.

Damping : In the step response of $\mathcal{G}_{yr}$, make overshoot and settling time small.
In the gain plot of $\mathcal{G}_{yr}$, make the peak gain M_p small.

Steady-state characteristics : In the step response of $\mathcal{G}_{yr}$,
make the steady-state deviation small.
In the gain plot of $\mathcal{G}_{yr}$, keep the DC gain nearly 0 dB.

4.2 PID control

For $\mathcal{K}$ in the Fig. 4.1, we consider PID controller in this section. This is widely used controller, it consists of Proportional, Integral, and Derivative operations.

In PID control, the control input is determined by the linear sum of each operation with respect to the deviation (difference between target value and output). For example, consider angle control of a vertical drive arm as shown in Fig. 4.4, where the output $y(t)$ is the angle of the arm and the control input $u(t)$ is the torque to be applied.

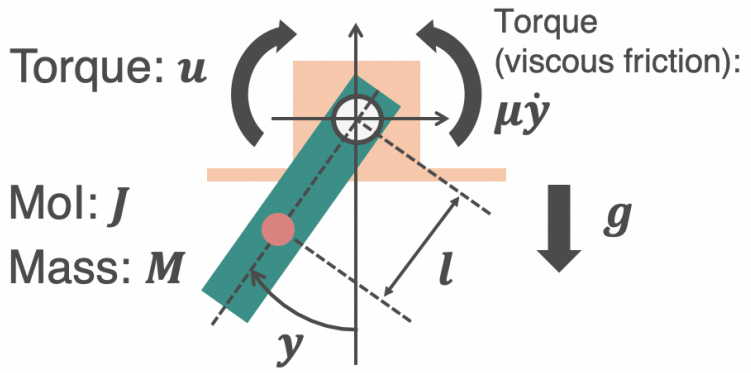

Figure 4.4: Vertical drive arm

In this case, we consider the difference $e(t) = r(t) - y(t)$ between the current angle and the target value. First, P control uses k_p times this deviation $e(t)$ as control input. However, with only this type of control, the control input and the torque due to gravity are in balance, and the target angle is never reached. Therefore, by adding the integral of the deviation to the control input, the control input that exceeds the torque due to gravity is generated. However, if the influence of P control or I control is made too large, the response may become oscillatory. Therefore, a derivative value of the deviation is added to the control input. This differential information predicts a little bit of the future for the arm's movement and prevents oscillatory behavior from occurring.

Since this alone is difficult to understand, we will consider it below by looking at individual Python codes and concrete diagrams. Note that if PID control is expressed as an expression, it is

$$u(t) = k_P e(t) + k_I \int_0^t e(\tau)\mathrm{d}\tau + k_D \dot{e}(t), \tag{4.9}$$

and when Laplace transformed, it is

$$u(t) = k_P e(s) + \frac{k_I}{s} e(s) + k_D s e(s) = \frac{k_D s^2 + k_P s + k_I}{s} e(s), \tag{4.10}$$

where k_P, k_I, and k_D are the proportional, integral, and derivative gains, respectively (Fig. 4.5).

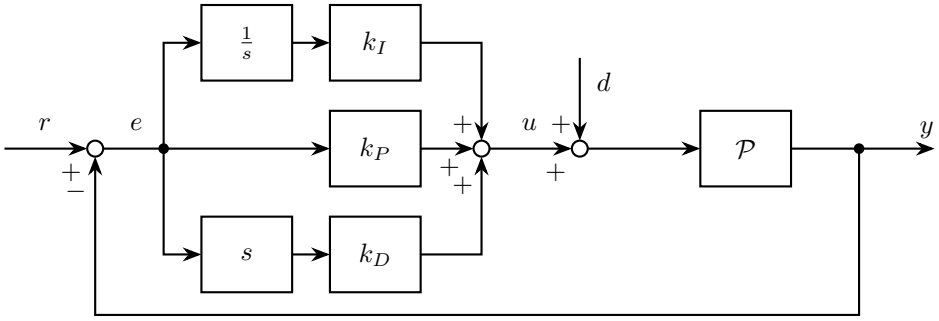

Figure 4.5: PID control

4.2.1 P control

P control is shown in Figure 4.6 with $\mathcal{K} = k_p$.

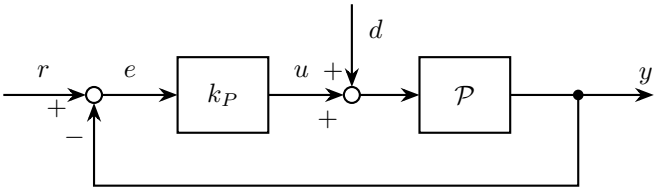

Figure 4.6: P control

We will consider the vertically driven arm in Figure 4.4 in the following sections, so we will derive the transfer function first. Since the transfer function describes a linear system, here we assume that the arm moves in the neighborhood of $y(t) = 0$ (linear approximation), the equation of motion is

$$J\ddot{y}(t) = u(t) - Mgly(t) - \mu\dot{y}(t), \tag{4.11}$$

which can be Laplace transformed and rearranged to

$$Js^2 y(s) + \mu s y(s) + Mgly(s) = u(s), \tag{4.12}$$

so the transfer function is

$$\mathcal{P} = \frac{y(s)}{u(s)} = \frac{1}{Js^2 + \mu s + Mgl}. \tag{4.13}$$

This model is described in Python as follows.

```
# Params for Vertical Drive Arm
from control.matlab import tf

g  = 9.81   # gravitational acceleration
l  = 0.2    # arm length
M  = 0.5    # arm mass
mu = 1.5e-2 # coefficient of viscous friction
J  = 1.0e-2 # moment of inertia

P = tf([0,1],[J,mu,M*g*l])

ref = 30 # reference angle
\label{list4.1}
```

Listing 4.1: Model of vertically driven arm

Here, the following code is executed to obtain the step response and Bode plot when the proportional gain k_p is varied, as shown in Figures 4.7 and 4.8.

```
# P Controller
## Step Responce
import matplotlib.pyplot as plt
import numpy as np
from control.matlab import tf, feedback, step, bode, logspace, mag2db

kp = (0.5, 1, 2) # P gain

fig, ax = plt.subplots()

for i in range(len(kp)):
    K = tf([0, kp[i]], [0, 1]) # P controller
    Gyr = feedback(P*K, 1)     # closed loop (see block diagram section in chap2)
    y,t = step(Gyr, np.arange(0, 2, 0.01))

    pltargs = {'label': f'$k_P$={kp[i]}'}
    ax.plot(t, y*ref, **pltargs)

ax.axhline(ref, color='k', linewidth=0.5)
plot_set(ax, 't', 'y', 'best')

#plt.savefig('Pcontroller_step',dpi=300)

## Bode Plot
fig, ax = plt.subplots(2,1)

for i in range(len(kp)):
    K = tf([0, kp[i]], [0, 1]) # P controller
    Gyr = feedback(P*K, 1)     # closed loop
    mag, phase, w = bode(Gyr, logspace(-1,2,1000), plot=False)

    pltargs = {'label': f'$k_P$={kp[i]}'}
    ax[0].semilogx(w, mag2db(mag), **pltargs)
    ax[1].semilogx(w, np.rad2deg(phase), **pltargs)

bodeplot_set(ax, 'lower left')

#plt.savefig('Pcontroller_bode',dpi=300)
```

Listing 4.2: P controller

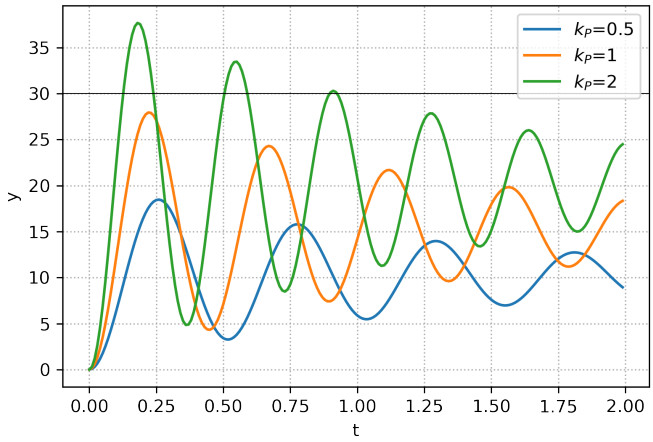

Figure 4.7: Step response of closed-loop system when using P control system

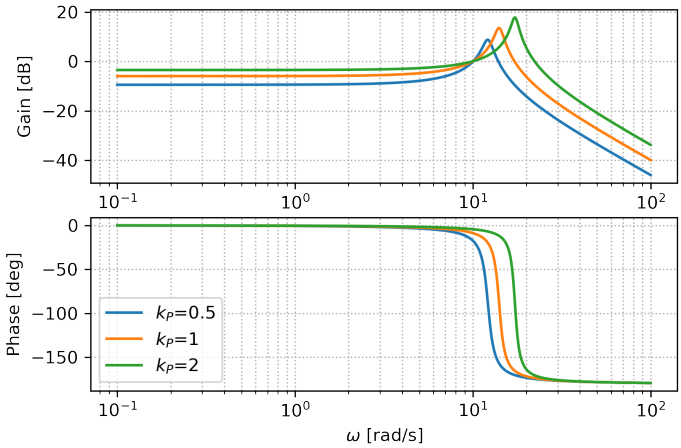

Figure 4.8: Bode plot of closed-loop system when using P control system

First, from Fig. 4.7, it can be seen that the output does not reach the target value of 30 deg in P control. However, by increasing the proportional gain, the difference between the target values becomes smaller, the rise time becomes faster, and the period of oscillation becomes shorter.

The Bode diagram in Figure 4.8 shows that increasing the proportional gain increases the low-frequency gain and the steady-state deviation of the step response becomes smaller. It also shows that the bandwidth and peak gains are also increased, which means that the response speed becomes faster but more oscillatory.

The above should also be confirmed in the formula. Since

$$\mathcal{P}(s) = \frac{1}{Js^2 + \mu s + Mgl}, \tag{4.14}$$

and

$$\mathcal{K}(s) = k_P, \tag{4.15}$$

closed loop system can be written

$$\mathcal{G}_{yr}(s) = \frac{\mathcal{P}(s)\mathcal{K}(s)}{1 + \mathcal{P}(s)\mathcal{K}(s)} = \frac{k_P}{Js^2 + \mu s + Mgl + k_P}. \tag{4.16}$$

If we map this to the standard form of the second-order lag system

$$\mathcal{G}_{yr}(s) = \frac{K\omega^2}{s^2 + 2\zeta\omega_n s + \omega_n^2}, \tag{4.17}$$

we get

$$\omega_n = \sqrt{\frac{Mgl + k_P}{J}},\ \zeta = \frac{\mu}{2\sqrt{J(Mgl + k_P)}},\ K = \frac{k_P}{Mgl + k_P}. \tag{4.18}$$

From this, it can be seen that the larger the proportional gain k_P is, the larger ω_n becomes, and thus the faster the response becomes. However, since ω_n is also included in the denominator of ζ, ζ becomes smaller at the same time, and the response becomes oscillatory. Therefore, it is not possible to improve both quick-response and damping at the same time. Also, from the final value theorem, $y(\infty) = \lim_{s\to 0} \mathcal{G}_{yr}(s) = K$, but since $K \neq 1$, the target value 1 is never reached. As k_P is increased, K approaches 1, but it cannot be infinitely large (input becomes excessive), so a steady deviation always remains in P control.

4.2.2 PD control

We saw that when k_P was increased in P control, the response became oscillatory. Therefore, an action of differentiation is added to reduce the oscillation. This is the PD control shown in Fig. 4.9, where $\mathcal{K} = k_D s + k_P$.

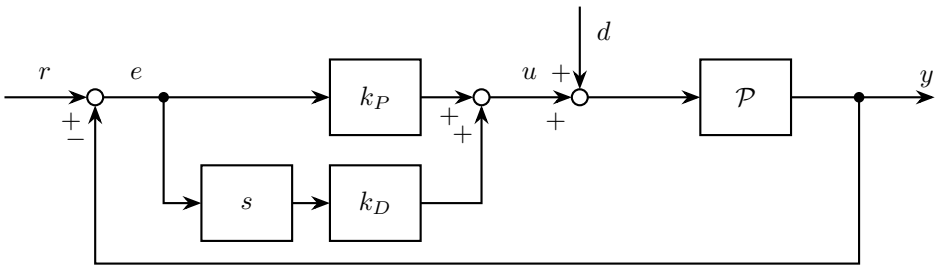

Figure 4.9: PD control

Execute the following code to obtain the step response and Bode diagram as shown in Figures 4.10 and 4.11.

```
# PD Controller

## Step Responce
```

```
kp = 2              # P gain
kd = (0, 0.1, 0.2) # D gain

fig, ax = plt.subplots()

for i in range(len(kd)):
    K = tf([kd[i], kp], [0, 1]) # PD controller
    Gyr = feedback(P*K, 1)      # closed loop
    y,t = step(Gyr, np.arange(0, 2, 0.01))

    pltargs = {'label': f'$k_D$={kd[i]}'}
    ax.plot(t, y*ref, **pltargs)

ax.axhline(ref, color='k', linewidth=0.5)
plot_set(ax, 't', 'y', 'best')

#plt.savefig('PDcontroller_step',dpi=300)

## Bode Plot
fig, ax = plt.subplots(2,1)

for i in range(len(kd)):
    K = tf([kd[i], kp], [0, 1]) # PD controller
    Gyr = feedback(P*K, 1)      # closed loop
    args = {'wrap_phase':True}
    mag, phase, w = bode(Gyr, logspace(-1,2,1000), plot=False, **args)

    pltargs = {'label': f'$k_D$={kd[i]}'}
    ax[0].semilogx(w, mag2db(mag), **pltargs)
    ax[1].semilogx(w, np.rad2deg(phase), **pltargs)

bodeplot_set(ax, 'lower left')

#plt.savefig('PDcontroller_bode',dpi=300)
```

Listing 4.3: PD controller

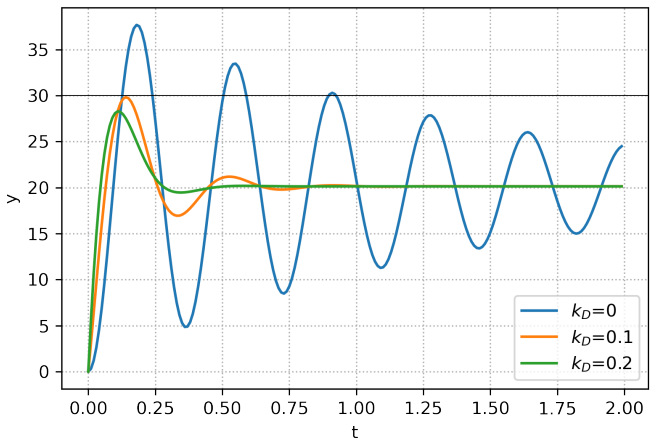

Figure 4.10: Step response of closed-loop system when using PD control system

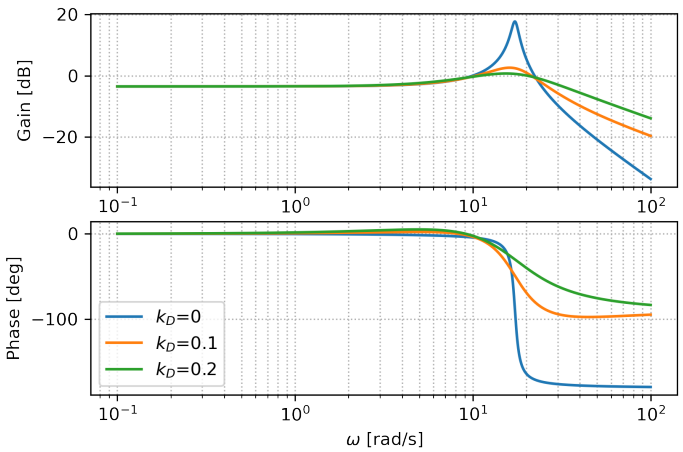

Figure 4.11: Bode plot of closed-loop system when using PD control system

First, from Fig. 4.10, it can be seen that the vibration can be suppressed by adding D control. However, the steady-state deviation does not become zero as in the case of P control.

The Bode plot in Fig. 4.11 shows that as the differential gain is increased, the peak gain becomes smaller and the bandwidth increases. This shows that oscillations are suppressed and the rise time becomes a little faster. However, the DC gain remains unchanged and the steady-state characteristics are not improved.

The above should also be confirmed in the formula. In the case of PD control,

$$\mathcal{K}(s) = k_P + k_D, \tag{4.19}$$

so the closed-loop system is

$$\mathcal{G}_{yr}(s) = \frac{\mathcal{P}(s)\mathcal{K}(s)}{1 + \mathcal{P}(s)\mathcal{K}(s)} = \frac{k_D s + k_P}{Js^2 + (\mu + k_D)s + Mgl + k_P}. \tag{4.20}$$

The poles of the closed-loop system are

$$s = \frac{-(\mu + k_D) \pm \sqrt{(\mu + k_D)^2 - 4J(Mgl + k_P)}}{2J}. \tag{4.21}$$

By adjusting k_D and k_P, the real and imaginary parts can be freely determined, respectively. This means that both quick-response and damping can be improved at the same time. However, since the value of $\mathcal{G}_{yr}(0)$ is the same as in the P control case, the steady-state characteristics do not change, and steady-state deviations always remain in the PD control case. Incidentally, D control uses the differential information of the deviation, but as mentioned in 2.2.2, the exact differential cannot be implemented as a controller. For example, in the case of PD control, the controller is $k_D s + k_P$, which is an improper. As it

is impossible to implement in the real world as it is, in practice, it is implemented in the form of

$$\mathcal{K}(s) = k_D \frac{s}{1 + T_{\mathrm{lp}} s} + k_P, \tag{4.22}$$

using an incomplete differentiator $\frac{s}{1+T_{\mathrm{lp}}s}$. This is a low-pass filter (first-order delay system) added to the derivative and is proprietary. Noise is amplified by the derivative, but the low-pass filter ($1/T_{\mathrm{lp}}$: cutoff frequency) reduces the effect of noise.

4.2.3 PID control

The PID control described at the beginning of this section (Fig. 4.5) incorporates the operation of integration to improve the steady-state characteristics. The following code is executed to obtain the step response when the integral gain is changed while the proportional gain and differential gain are fixed, as shown in Figures 4.12 and 4.13.

```
# PID Controller

## Step Responce
kp = 2          # P gain
ki = (0, 5, 10) # I gain
kd = 0.1        # D gain

fig, ax = plt.subplots()

for i in range(len(ki)):
    K = tf([kd, kp, ki[i]], [1, 0]) # PD controller
    Gyr = feedback(P*K, 1)          # closed loop
    y,t = step(Gyr, np.arange(0, 2, 0.01))

    pltargs = {'label': f'$k_I$={ki[i]}'}
    ax.plot(t, y*ref, **pltargs)

ax.axhline(ref, color='k', linewidth=0.5)
plot_set(ax, 't', 'y', 'best')

#plt.savefig('PIDcontroller_step',dpi=300)

## Bode Plot
fig, ax = plt.subplots(2,1)

for i in range(len(ki)):
    K = tf([kd, kp, ki[i]], [1, 0]) # PD controller
    Gyr = feedback(P*K, 1)          # closed loop
    args = {'wrap_phase':True}
    mag, phase, w = bode(Gyr, logspace(-1,2,1000), plot=False, **args)

    pltargs = {'label': f'$k_I$={ki[i]}'}
    ax[0].semilogx(w, mag2db(mag), **pltargs)
    ax[1].semilogx(w, np.rad2deg(phase), **pltargs)

bodeplot_set(ax, 'lower left')

#plt.savefig('PIDcontroller_bode',dpi=300)
```

Listing 4.4: PID controller

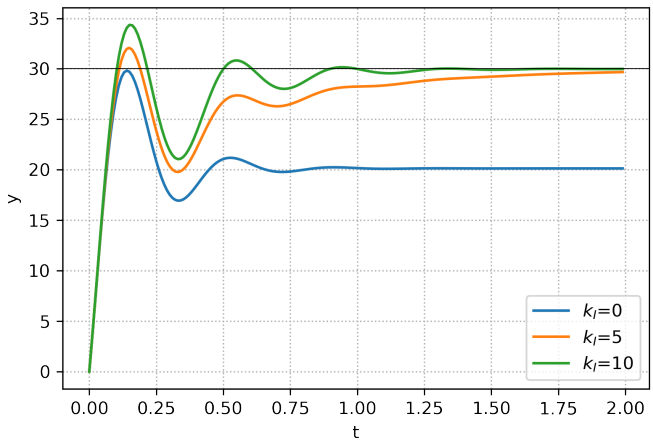

Figure 4.12: Step response of closed-loop system when using PID control system

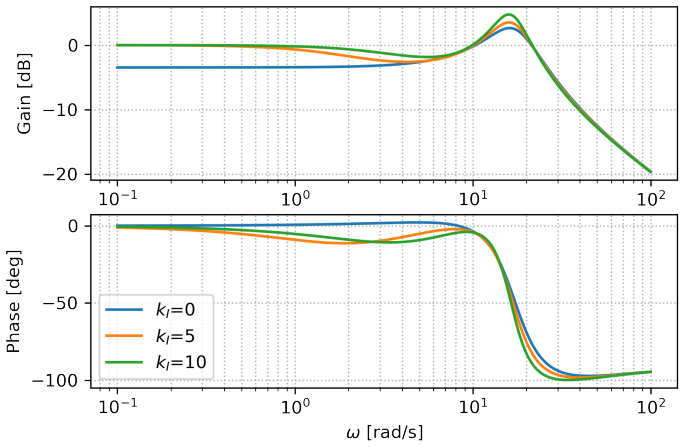

Figure 4.13: Bode plot of a closed-loop system when using a PID control system

First, from Figure 4.12, we can see that the steady-state deviation becomes zero by I control, and the larger k_I is, the more oscillatory it becomes.

Also, from the Bode diagram in Fig. 4.13, we can see that the DC gain is 0 dB and the peak gain is slightly larger. In other words, while the steady-state deviation is zero, the larger the gain, the more oscillatory it becomes.

The above is also confirmed by the equation: In the case of PD control, since

$$\mathcal{K}(s) = \frac{k_D s^2 + k_P s + k_I}{s}, \tag{4.23}$$

the closed-loop system is

$$\mathcal{G}_{yr}(s) = \frac{\mathcal{P}(s)\mathcal{K}(s)}{1 + \mathcal{P}(s)\mathcal{K}(s)} = \frac{k_D s^2 + k_P s + k_I}{Js^3 + (\mu + k_D)s^2 + (Mgl + k_P)s + k_I}. \tag{4.24}$$

In this case, $\mathcal{G}_{yr}(0) = 1$, which means that the steady-state deviation is zero. Also, the transfer function from the disturbance d to the output y is

$$\mathcal{G}_{yd}(s) = \frac{\mathcal{P}(s)}{1 + \mathcal{P}(s)\mathcal{K}(s)} = \frac{s}{Js^3 + (\mu + k_D)s^2 + (Mgl + k_P)s + k_I} \tag{4.25}$$

and $\mathcal{G}_{yr}(0) = 0$, which means that the steady-state deviation is 0 for a constant value disturbance (step-like disturbance). Note that the disturbance suppression performance of PID control can be found by executing the following code (Fig. 4.14, 4.15)

```
# PID Controller -- Disturbance Supression (Gyd)
## Step Responce
kp = 2          # P gain
ki = (0, 5, 10) # I gain
kd = 0.1        # D gain

fig, ax = plt.subplots()

for i in range(len(ki)):
    K = tf([kd, kp, ki[i]], [1, 0])
    Gyd = feedback(P, K)
    y, t = step(Gyd, np.arange(0, 2, 0.01))

    pltargs = {'label': f'$k_I$={ki[i]}'}
    ax.plot(t, y, **pltargs)

plot_set(ax, 't', 'y', 'best')

plt.savefig('PIDcontroller_Gyd_step',dpi=300)

## Bode Plot
fig, ax = plt.subplots(2, 1)

for i in range(len(ki)):
    K = tf([kd, kp, ki[i]], [1, 0])
    Gyd = feedback(P, K)
    args = {'wrap_phase':True}
    mag, phase, w = bode(Gyd, logspace(-1,2), plot=False, **args)

    pltargs = {'label': f'$k_I$={ki[i]}'}
    ax[0].semilogx(w, mag2db(mag), **pltargs)
    ax[1].semilogx(w, np.rad2deg(phase), **pltargs)

bodeplot_set(ax, 'best')

#plt.savefig('PIDcontroller_Gyd_bode',dpi=300)
```

Listing 4.5: PID controller

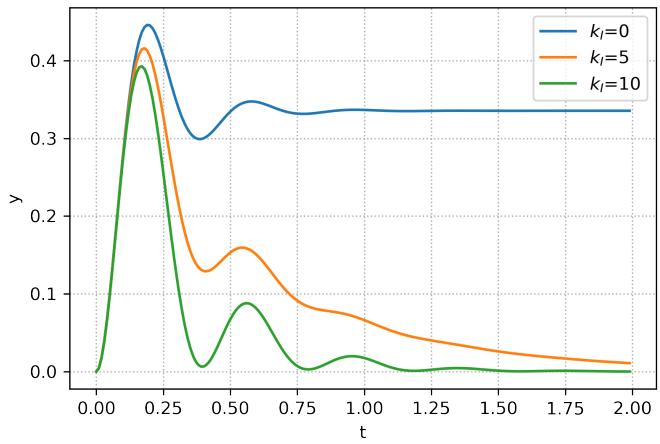

Figure 4.14: Disturbance suppression performance of PID control

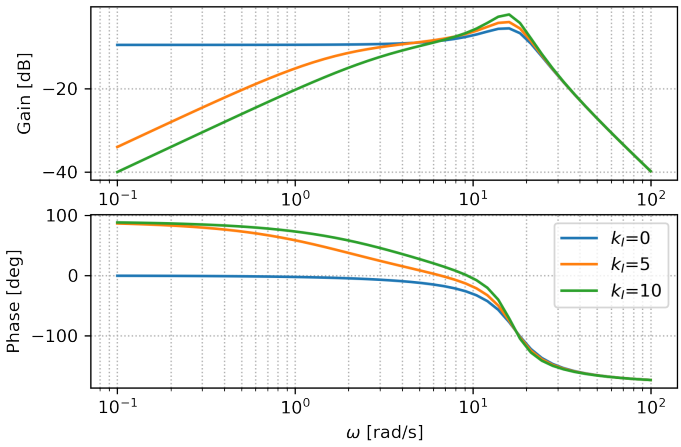

Figure 4.15: Disturbance suppression performance of PID control (Bode plot)

From this, increasing the integral gain causes the output to converge to zero, which means that the disturbance is suppressed, i.e., the desired response is obtained. The Bode diagram also shows that the low-frequency gain is $-\infty$ dB, which means that low-frequency disturbances are suppressed.

4.2.4 Improved PID control

When PID control is actually used, it is often in a modified form. For example, when the target value changes in a step-like manner in PID control, the control input $u(t)$ includes an impulse component generated by the differentiator. To avoid this, PI-D control is used

in which the differentiator acts only on the output as shown in Figure 4.16. The expression is

$$u(s) = k_P e(s) + \frac{k_I}{s} e(s) - k_D s y(s). \tag{4.26}$$

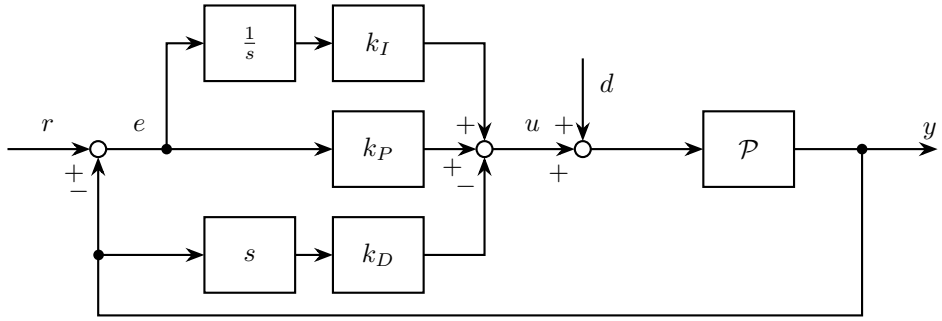

Figure 4.16: PI-D control

Since step-like signals are included in the input $u(s)$ in P control, the input may reflect k_P times the output instead of k_P times the deviation, as shown in Figure 4.17. This is called I-PD control and is expressed as

$$u(s) = -k_P y(s) + \frac{k_I}{s} e(s) - k_D s y(s). \tag{4.27}$$

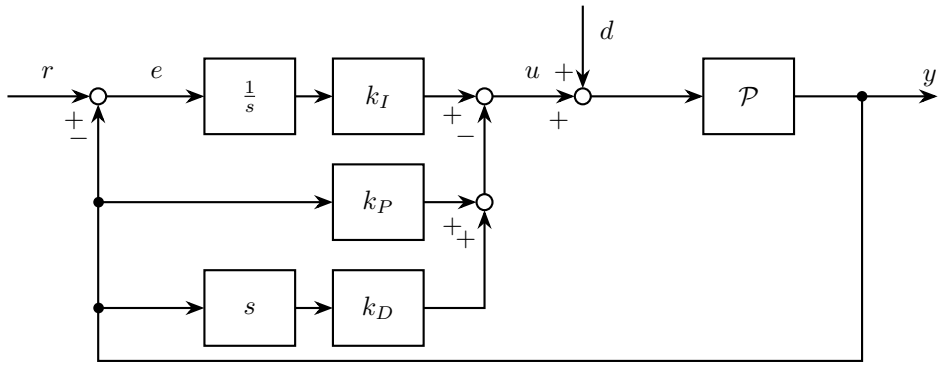

Figure 4.17: I-PD control

Now, in the case of PI-D control,

$$\begin{aligned} u(s) &= \frac{k_P s + k_I}{s} r(s) - \frac{k_D s^2 + k_P s + k_I}{s} y(s) \\ &= \frac{k_D s^2 + k_P s + k_I}{s} \left(\frac{k_P s + k_I}{k_D s^2 + k_P s + k_I} r(s) - y(s) \right) \\ &= \mathcal{K}_1(s)(\mathcal{K}_2(s) r(s) - y(s)) \end{aligned} \tag{4.28}$$

where

$$\mathcal{K}_1(s) = \frac{k_D s^2 + k_P s + k_I}{s}$$
$$\mathcal{K}_2(s) = \frac{k_P s + k_I}{k_D s^2 + k_P s + k_I} \tag{4.29}$$

and $\mathcal{K}_1(s)$ is a PID controller. Also, $\mathcal{K}_2(s)$ is a second-order lag system, which can be interpreted as shaping the target value r with a second-order filter and performing PID control using the shaped target value. In other words, PI-D control performs stabilization and disturbance suppression by feedback control using the PID controller $\mathcal{K}_1(s)$, and maintains target value response performance by feedforward control using the target value filter $\mathcal{K}_2$.

Note that $\mathcal{K}_1(s)$ is the same for I-PD control and is

$$\mathcal{K}_2(s) = \frac{k_I}{k_D s^2 + k_P s + k_I}. \tag{4.30}$$

This is called two-degree-of-freedom control with feedback control and feed-forward control added, and the improved PID control is called improved PID control. The following code can now be executed to compare PID control with PI-D control (Figure 4.18) and I-PD control (Figure 4.19) of the vertical drive arm.

```
# Improved PID
from control.matlab import tf, feedback, lsim

kp, ki, kd = 2, 10, 0.1
K1 = tf([kd, kp, ki],[1,0])     # PID
K2 = tf([kp, ki],[kd, kp, ki]) # PI-D
K3 = tf([0, ki],[kd, kp, ki])  # I-PD

Gyz = feedback(P*K1, 1) # TF from z to y

Td = np.arange(0, 2, 0.01)
r = 1*(Td>0)

z, t, _ = lsim(K2, r, Td, 0) # forming of referece with K2
z2, t2, _ = lsim(K3, r, Td, 0)

fig, ax = plt.subplots(1, 2,figsize=(8.0, 4.0))

y, _, _ = lsim(Gyz, r, Td, 0) # PID (z=r)
ax[0].plot(t, r*ref)
ax[1].plot(t, y*ref, label='PID')

y, _, _ = lsim(Gyz, z, Td, 0) # PI-D
ax[0].plot(t, z*ref)
ax[1].plot(t, y*ref, label='PI-D')

ax[1].axhline(ref, color='k', linewidth=0.5)

ax[1].set_ylim=(0,50)

plot_set(ax[0], 't', 'r')
plot_set(ax[1], 't', 'y', 'lower right')

#plt.savefig('Improved_PIDcontroller_PI-D',dpi=300)

fig, ax = plt.subplots(1, 2,figsize=(8.0, 4.0))

```

```
y, _, _ = lsim(Gyz, r, Td, 0) # PID (z=r)
ax[0].plot(t, r*ref)
ax[1].plot(t, y*ref, label='PID')

y, _, _ = lsim(Gyz, z2, Td, 0) # PI-D
ax[0].plot(t, z2*ref)
ax[1].plot(t, y*ref, label='I-PD')

ax[1].axhline(ref, color='k', linewidth=0.5)

ax[1].set_ylim=(0,50)

plot_set(ax[0], 't', 'r')
plot_set(ax[1], 't', 'y', 'lower right')

#plt.savefig('Improved_PIDcontroller_I-PD',dpi=300)
```

Listing 4.6: Improved PID control

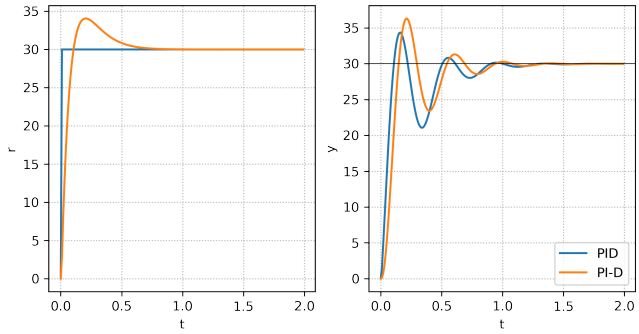

Figure 4.18: Comparison of PID control and PI-D control

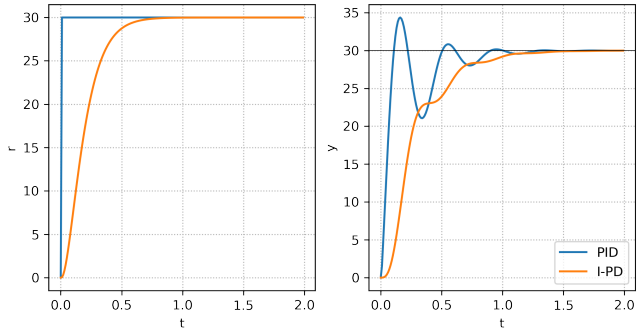

Figure 4.19: Comparison of PID control and I-PD control

The left figure in Fig. 4.18 is the signal z shaped by r and $\mathcal{K}_2(s)$, and the right figure is the output of the plant when it is taken as input. Although the target value is smoothed by the target value filter $\mathcal{K}_2$, the output of the control target is also more oscillatory than that of PID control because the target value is oscillatory.

Figure 4.19 shows that the oscillation of the signal shaped by $\mathcal{K}_2(s)$ is suppressed, which also suppresses the oscillation of the output of the plant. The difference between PI-D control and I-PD control is that $\mathcal{K}_2(s)$ includes or does not include zero points, respectively.

4.3 Gain tuning

4.3.1 Ultimate sensitivity method

One method for tuning PID gain is the ultimate sensitivity method. While this method has the disadvantage that the actual device must be operated near the limit of instability (ultimate sensitivity), it has the advantage that it does not require a model of the control target.

In the ultimate sensitivity method, P control is first configured and the proportional gain k_P is increased. As the oscillation increases and a sustained oscillation is generated, the proportional gain k_{P0} and the period of the sustained oscillation T_0 at that time are investigated. Note that in an ideal first-order or second-order lag system, even if the proportional gain is increased, sustained oscillation does not occur, but in an actual system, there is dead time, and sustained oscillation is caused (which becomes unstable when the gain is increased).

Then, the proportional gain k_P, integral time T_I, and derivative time T_D are determined using Table 4.1 and Table 4.2. Here, PID control is assumed to be

$$u(t) = k_P \left(e(t) + \frac{1}{T_I} \int_0^t e(\tau) \mathrm{d}\tau + T_D \frac{\mathrm{d}}{\mathrm{d}t} e(t) \right), \tag{4.31}$$

and $k_I = \frac{k_P}{T_I}$, $k_D = k_P T_D$.

	k_P	T_I	T_D
P control	$0.5k_{P0}$		
PI control	$0.45k_{P0}$	$0.83T_0$	
PID control	$0.6k_{P0}$	$0.5T_0$	$0.125T_0$

Table 4.1: Ultimate sensitivity method

	k_P	T_I	T_D
No Overshoot	$0.2k_{P0}$	$0.5T_0$	$0.33T_0$

Table 4.2: Improvement of table 4.1

As an example, let us assume that the vertical drive arm (second-order lag system) of list 4.1 has a minute dead time as the plant. Also, a first-order Padé approximation (a rational function approximation of the dimensionless system e^{-hs}) is used, and the dead

time is assumed to be 0.005. Applying P control with an appropriate proportional gain of $k_{P0} = 2.9$ as in the following code, we obtain Figure 4.20. Furthermore, Figure 4.21 is the result of tuning the PID gain using the period of duration.

```
# Gain tuning with Ultimate Sensitivity Method

from control.matlab import tf, pade, feedback, step

num_delay, den_delay = pade(0.005, 1) # dead time (Pad\UTF{00E9} approximant)
Pdelay = P * tf(num_delay, den_delay)

kp0 = 2.9                    # P gain
K = tf([0, kp0], [0,1])      # P controller
Gyr = feedback(Pdelay*K, 1) # closed loop
y, t = step(Gyr, np.arange(0, 2, 0.01))

fig, ax = plt.subplots()
ax.plot(t, y*ref)
ax.axhline(ref, color='k', linewidth=0.5)
plot_set(ax, 't', 'y')

#plt.savefig('sustained_oscillation',dpi=300)

kp = [0, 0]
ki = [0, 0]
kd = [0, 0]
Rule = ['', '']

T0 = 0.3 # this value is periodic time of the sustained oscillation above

Rule[0] = 'Classic' # ultimate sensitivity method
kp[0] = 0.6 * kp0
ki[0] = kp[0] / (0.5 * T0)
kd[0] = kp[0] * (0.125 * T0)

Rule[1] = 'No Overshoot' # improved ultimate sensitivity method
kp[1] = 0.2 * kp0
ki[1] = kp[1] / (0.5 * T0)
kd[1] = kp[1] * (0.33 * T0)

fig, ax = plt.subplots()

for i in range(2):
    K = tf([kd[i], kp[i], ki[i]], [1,0])
    Gyr = feedback(Pdelay*K, 1)
    y, t = step(Gyr, np.arange(0, 2, 0.01))

    ax.plot(t, y*ref, label=Rule[i])

    print(Rule[i])
    print('kP=', kp[i])
    print('kI=', ki[i])
    print('kD=', kd[i])
    print('-------------')

ax.axhline(ref, color='k', linewidth=0.5)
plot_set(ax, 't', 'y', 'best')

#plt.savefig('gain_tuning',dpi=300)
```

Listing 4.7: Gain tuning

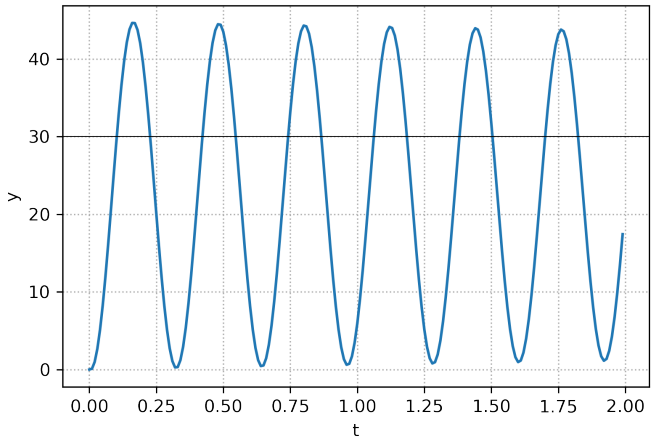

Figure 4.20: Sustained vibration

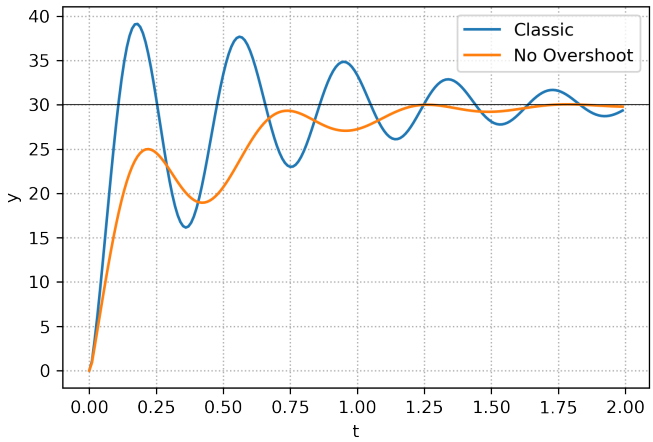

Figure 4.21: The result of gain tuning

Thus, it can be seen that the ultimate sensitivity method produces reasonably good results and that the improved version does not have overshoot.

4.3.2 Model matching

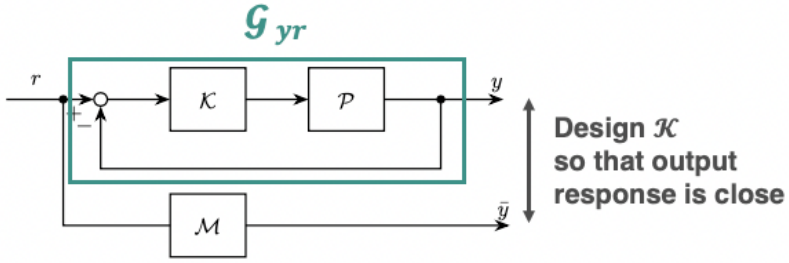

Figure 4.22: Model matching

The model matching method, as shown in Figure 4.22, is to provide an appropriate normative model $\mathcal{M}(s)$ and match (or approach) the transfer function $\mathcal{G}_{yr}(s)$ from the target value r to the output y to it. Binomial coefficient standard forms and Butterworth standard forms are often used for normative models. For example, in the case of a second-order system, $\zeta = 1$ and $\zeta = \frac{1}{\sqrt{2}}$ in

$$\mathcal{M}(s) = \frac{\omega_n^2}{s^2 + 2\zeta\omega_n s + \omega_n^2}, \tag{4.32}$$

respectively (Fig. 4.23).

In the case of the third-order system, $(\alpha_1, \alpha_2) = (3, 3)$, $(\alpha_1, \alpha_2) = (2, 2)$, and $(\alpha_1, \alpha_2) = (2.15, 1.75)$ for

$$\mathcal{M}(s) = \frac{\omega_n^3}{s^3 + \alpha_2\zeta\omega_n s^2 + \alpha_1\omega_n^2 s + \omega_n^3}, \tag{4.33}$$

are binomial coefficient standard form, Butterworth standard form, and ITAE minimum standard form (which approximately minimizes the time weighted integral of the absolute value of the deviation) respectively (Figure 4.24).

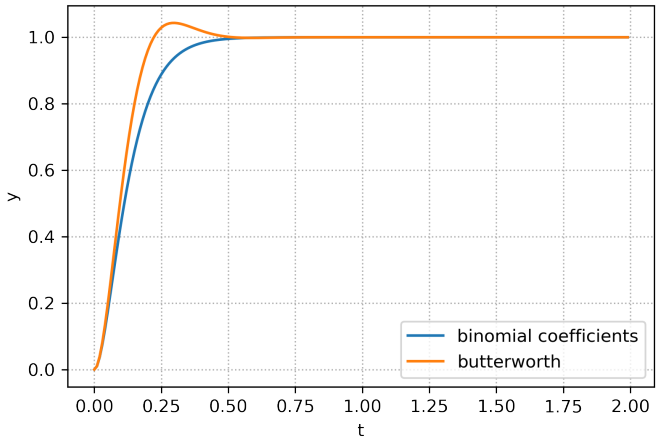

Figure 4.23: nominal model for second-order lag system

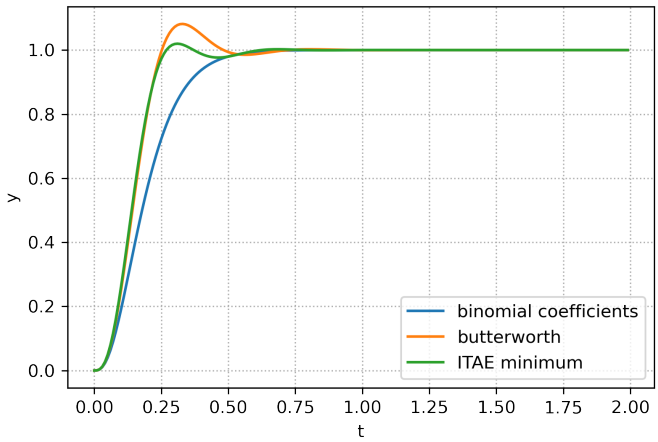

Figure 4.24: nominal model for third-order lag system

The model matching method finds the transfer function $\mathcal{G}_{yr}(s)$ from r to y and calculates the Maclaurin expansion of $\frac{1}{\mathcal{G}_{yr}(s)}$ and $\frac{1}{\mathcal{M}(s)}$. The PID gains are determined to match the lower-order terms in that order.

For example, consider the model matching of a PI-D control system for a vertical drive arm. The transfer function from r to y is

$$\mathcal{G}_{yr}(s) = \frac{k_P s + k_I}{Js^3 + (\mu + k_D)s^2 + (Mgl + k_P)s + k_I}, \tag{4.34}$$

which is matched to the quadratic normative model $\mathcal{M}$ in 4.32. To find the Maclaurin expansion of $\frac{1}{\mathcal{G}_{yr}(s)}$, we can run the following code.

```
# Maclaurin Expansion
import sympy as sp

s = sp.Symbol('s')
kp, kd, ki = sp.symbols('k_p k_d k_i')
Mgl, mu, J = sp.symbols('Mgl mu J')
sp.init_printing()

G = (kp*s + ki) / (J*s**3 + (mu+kd)*s**2 + (Mgl+kp)*s + ki)
sp.series(1/G, s, 0, 4)
```

Listing 4.8: Maclaurin expansion

From this, we get

$$1+s^2\left(-\frac{Mglk_P}{k_I^2}+\frac{k_D}{k_I}+\frac{\mu}{k_I}\right)+s^3\left(\frac{J}{k_I}+\frac{Mglk_P^2}{k_I^3}+\frac{k_Dk_P}{k_I^2}-\frac{k_P\mu}{k_I^2}\right)+\frac{Mgls}{k_I}+O(s^4), \quad (4.35)$$

so

$$\frac{1}{\mathcal{G}_{yr}(s)}=1+\frac{Mgl}{k_I}s+\left(\frac{\mu+k_D}{k_I}-Mgl\frac{k_P}{k_I^2}\right)s^2+\left(\frac{J}{k_I}-\frac{k_P(\mu+k_D)}{k_I^2}+Mgl\frac{k_P^2}{k_I^3}\right)s^3+\cdots. \quad (4.36)$$

On the other hand, $\frac{1}{\mathcal{M}(s)}$ will be

$$\frac{1}{\mathcal{M}(s)}=1+\frac{2\zeta}{\omega_n}s+\frac{1}{\omega_n^2}s^2. \quad (4.37)$$

Next, k_P, k_I, and k_D are obtained using the following code to match the first, second, and third order terms of s in $\frac{1}{\mathcal{G}_{yr}(s)}$ and $\frac{1}{\mathcal{M}(s)}$.

```
# Model Matching
import sympy as sp

z, wn = sp.symbols('zeta omega_n')
kp, kd, ki = sp.symbols('k_p k_d k_i')
Mgl, mu, J = sp.symbols('Mgl mu J')
sp.init_printing()

f1 = Mgl/ki - 2*z/wn
f2 = (mu+kd)/ki - Mgl*kp/(ki**2) - 1/(wn**2)
f3 = J/ki - kp*(mu+kd)/(ki**2) + Mgl*kp**2/(ki**3)

sp.solve([f1, f2, f3],[kp, kd, ki])
```

Listing 4.9: Model mathcing

As a result, we get

$$\left(J\omega_n^2,\ 2J\omega_n\zeta+\frac{Mgl}{2\omega_n\zeta}-\mu,\ \frac{Mgl\omega_n}{2\zeta}\right). \quad (4.38)$$

From this,

$$k_P=\omega_n^2J,\ k_I=\frac{\omega_nMgl}{2\zeta},\ k_D=2\zeta\omega_nJ+\frac{Mgl}{2\zeta\omega_n}-\mu. \quad (4.39)$$

Using this, the following code is executed to obtain Figure 4.25.

```
# Check of Gain Tuning with Model Matching
from control.matlab import tf, step

```

```
omega_n = 15
zeta = 1/np.sqrt(2) # butterworth
Msys = tf([0,omega_n**2],[1,2*zeta*omega_n,omega_n**2]) # normative model

g  = 9.81   # gravitational acceleration
l  = 0.2    # arm length
M  = 0.5    # arm mass
mu = 1.5e-2 # coefficient of viscous friction
J  = 1.0e-2 # moment of inertia

## model matching
kp = J*omega_n**2
ki = M*g*l*omega_n/(2*zeta)
kd = 2*J*omega_n*zeta + M*g*l/(2*omega_n*zeta) - mu
Gyr = tf([kp, ki],[J, mu+kd, M*g*l+kp, ki])

yM, tM = step(Msys, np.arange(0, 2, 0.01))
y, t = step(Gyr, np.arange(0, 2, 0.01))

fig, ax = plt.subplots()
ax.plot(tM, yM*ref, label='M')
ax.plot(t, y*ref, label='Gyr', linestyle='--')
plot_set(ax, 't', 'y', 'best')

#plt.savefig('gain_tuning_with_model_mathcing',dpi=300)
```

Listing 4.10: Gain tuning with model matching

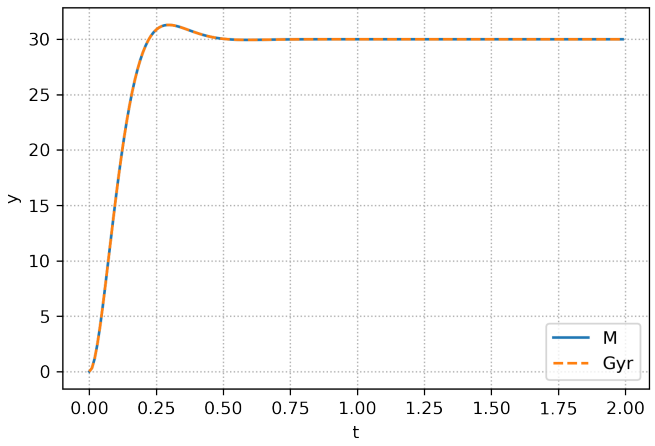

Figure 4.25: Gain tuning with model matching

This shows that the response of the closed-loop system $\mathcal{G}_{yr}(s)$ perfectly matches the response of the normative model $\mathcal{M}(s)$.

4.4 State feedback control

This time, a controller is designed for the system described by the state-space model $\dot{\boldsymbol{x}} = \boldsymbol{A}\boldsymbol{x} + \boldsymbol{B}u$. Here, all states are assumed to be observable by sensors, etc., and the observed

information is used to determine the control input. Also, consider a state feedback

$$u = \boldsymbol{F}\boldsymbol{x}, \tag{4.40}$$

as shown in Figure 4.26.

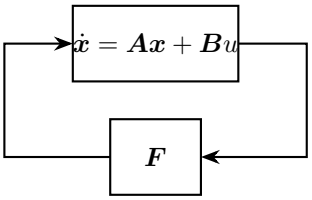

Figure 4.26: State feedback control

This uses information on the state $\boldsymbol{x}$ to determine the control input u. In this case, there are two methods for designing feedback gain $\boldsymbol{F}$: the pole placement method and the optimal regulator.

4.4.1 Pole placement

When state feedback control $u = \boldsymbol{F}\boldsymbol{x}$ is applied to the system $\dot{\boldsymbol{x}} = \boldsymbol{A}\boldsymbol{x}+\boldsymbol{B}u$, the closed-loop system becomes

$$\dot{\boldsymbol{x}} = (\boldsymbol{A} + \boldsymbol{B}\boldsymbol{F})\boldsymbol{x}. \tag{4.41}$$

Since the system is stable if the real parts of all eigenvalues of the matrix $\boldsymbol{A} + \boldsymbol{B}\boldsymbol{F}$ are negative, $\boldsymbol{F}$ is designed to be so.

In the pole assignment method, the eigenvalues of $\boldsymbol{A} + \boldsymbol{B}\boldsymbol{F}$ with negative real parts are first specified for the number of states. In the case of Python, you can use `F = -acker(A, B, p)` to obtain $\boldsymbol{F}$ so that the eigenvalues of $\boldsymbol{A}+\boldsymbol{B}\boldsymbol{F}$ are the ones you specified. The reason for the negative sign is that the return value is $\boldsymbol{F}$ such that the eigenvalues of $\boldsymbol{A} - \boldsymbol{B}\boldsymbol{F}$ are the specified poles.

Based on the above, the following code can be executed to perform state feedback control using the pole placement method, and Figure 4.27 is obtained.

```
# State Feedback Control -- pole placement method
from control.matlab import ss, acker, initial

A = '0 1; -4 5'
B = '0; 1'
C = '1 0; 0 1'
D = '0; 0'
P = ss(A, B, C, D)

regulator_poles = [-1, -1]
F = -acker(P.A, P.B, regulator_poles)

Acl = P.A + P.B@F
Pfb = ss(Acl, P.B, P.C, P.D)

Td = np.arange(0, 5, 0.01)
X0 = [-0.3, 0.4]
x, t = initial(Pfb, Td, X0)
```

```
19
20 fig, ax = plt.subplots()
21 ax.plot(t, x[:,0], label='$x_1$')
22 ax.plot(t, x[:,1], label='$x_2$')
23 plot_set(ax, 't', 'x', 'best')
24
25 #plt.savefig('SFC_PolePlacement',dpi=300)
```

Listing 4.11: Pole placement

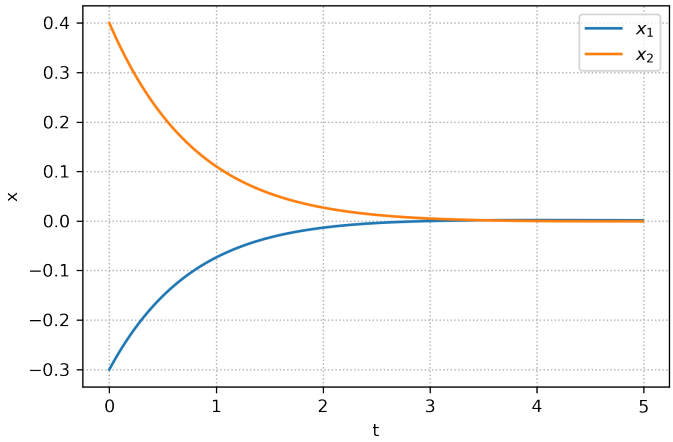

Figure 4.27: Pole placement

This shows that the state $\boldsymbol{x}$ converges to 0.

4.4.2 Controllability and observability

In designing of state feedback gains by pole placement, the system must be controllable (the controller can be freely designed). In general, controllability is very important when constructing a control system, so we discuss it here. In addition, observability, which is necessary for the design of observers that will (probably) appear later, is often explained as a set, so it is described here. Note that we assume $u = \boldsymbol{u}$ for this section, since the case other than one input is also valid.

4.4.2.1 Controllability

A system is said to be controllable if the input $\boldsymbol{u}(t)$ can be transferred from any initial value $\boldsymbol{x}(0)$ to any $\boldsymbol{x}(t_z)$ in any finite time t_z by control, otherwise it is said to be uncontrollable. Note that controllability is determined only by the matrices $\boldsymbol{A}$ and $\boldsymbol{B}$ (independent of output). In addition, the necessary and sufficient conditions for being controllable are as follows.

Necessary and sufficient conditions for the system to be controllable

It is to be full row rank

$$\text{rank } \boldsymbol{U}_c = n, \tag{4.42}$$

where

$$\boldsymbol{U}_c = \begin{bmatrix} \boldsymbol{B} & \boldsymbol{AB} & \boldsymbol{A^2B} & \cdots & \boldsymbol{A}^{n-1}\boldsymbol{B} \end{bmatrix}. \tag{4.43}$$

Here, $\boldsymbol{U}_c(n \times (nm))$ is called controllability matrix.

First, let's consider the necessary conditions.

$$e^{-\boldsymbol{A}t_z}\boldsymbol{x}(t_z) - \boldsymbol{x}(0) = \int_0^{t_z} e^{-\boldsymbol{A}\tau}\boldsymbol{B}\boldsymbol{u}\tau)\mathrm{d}\tau, \tag{4.44}$$

is a solution to the state equation. Assume that there exists an arbitrary initial state $\boldsymbol{x}(0)$, a finite time t_z, $\boldsymbol{u}(t)$ for the state $\boldsymbol{x}(t_z)$. Note that $e^{-\boldsymbol{A}\tau}$ is a state transition matrix,

$$e^{-\boldsymbol{A}\tau} = \boldsymbol{I} - \boldsymbol{A}\tau + \frac{1}{2!}\boldsymbol{A}^2\tau^2 + \cdots + \frac{1}{n!}\boldsymbol{A}^n(-\tau)^n + \cdots. \tag{4.45}$$

Using Cayley-Hamilton's theorem

$$\boldsymbol{A}^n + a_n\boldsymbol{A}^{n-1} + \cdots + a_2\boldsymbol{A} + a_1\boldsymbol{I} = \boldsymbol{0}, \tag{4.46}$$

for the matrix $\boldsymbol{A}$, the state transition matrix $e^{-\boldsymbol{A}\tau}$ can be expressed as an $n-1$ degree polynomial

$$e^{-\boldsymbol{A}\tau} = q_0(\tau)\boldsymbol{I} + q_1(\tau)\boldsymbol{A} + \cdots + q_{n-1}(\tau)\boldsymbol{A}^{n-1}, \tag{4.47}$$

of the matrix $\boldsymbol{A}$. So, we get

$$\begin{aligned} e^{-\boldsymbol{A}t_z}\boldsymbol{x}(t_z) - \boldsymbol{x}(0) &= \int_0^{t_z} \left\{q_0(\tau)\boldsymbol{I} + q_1(\tau)\boldsymbol{A} + \cdots + q_{n-1}(\tau)\boldsymbol{A}^{n-1}\right\}\boldsymbol{B}\boldsymbol{u}(\tau)\mathrm{d}\tau \\ &= \begin{bmatrix} \boldsymbol{B} & \boldsymbol{AB} & \cdots & \boldsymbol{A}^{n-1}\boldsymbol{B} \end{bmatrix} \int_0^{t_z} \begin{bmatrix} q_0(\tau) \\ q_1(\tau) \\ \vdots \\ q_{n-1}(\tau) \end{bmatrix} \boldsymbol{u}(\tau)\mathrm{d}\tau \\ &= \boldsymbol{U}_c \int_0^{t_z} \begin{bmatrix} q_0(\tau) \\ q_1(\tau) \\ \vdots \\ q_{n-1}(\tau) \end{bmatrix} \boldsymbol{u}(\tau)\mathrm{d}\tau. \end{aligned} \tag{4.48}$$

For there to be $u(t)$ for any initial state $\boldsymbol{x}(0)$, finite time t_z, and state $\boldsymbol{x}(t_z)$, there must be at least n linearly independent nm column vectors in $\boldsymbol{U}_c$. That is,

$$\text{rank } \boldsymbol{U}_c = n. \tag{4.49}$$

This is necessary condition.

On the other hand, sufficiency, i.e., rank$\boldsymbol{U}_c = n$, holds when there exists an arbitrary initial state $\boldsymbol{x}(0)$, a finite time t_z, and an input $\boldsymbol{u}(t)$ for the state $\boldsymbol{x}(t_z)$. Let

$$\boldsymbol{W}_c(t) \equiv \int_0^t e^{-\boldsymbol{A}\tau}\boldsymbol{B}\boldsymbol{B}^T e^{-\boldsymbol{A}^T\tau}\mathrm{d}\tau \tag{4.50}$$

be the controllable Gram matrix (controllable Gramian). If this matrix is regular, then by determining the input as

$$\boldsymbol{u}(t) = -\boldsymbol{B}^T e^{-\boldsymbol{A}^T t}\boldsymbol{W}_c(t_z)^{-1}\left[\boldsymbol{x}(0) - e^{-\boldsymbol{A}t_z}\boldsymbol{x}(t_z)\right], \tag{4.51}$$

for any initial state $\boldsymbol{x}(0)$, finite time t_z, and state $\boldsymbol{x}(t_z)$, we can make the state at time t_z like $\boldsymbol{x}(t_z)$, as

$$\begin{aligned} e^{\boldsymbol{A}t_z}\boldsymbol{x}(0) + \int_0^{t_z} e^{\boldsymbol{A}(t_z-\tau)}\boldsymbol{B}\boldsymbol{u}(\tau)\mathrm{d}\tau &= e^{\boldsymbol{A}t_z}\left\{\boldsymbol{x}(0) + \int_0^{t_z} e^{-\boldsymbol{A}\tau}\boldsymbol{B}(-\boldsymbol{B}e^{-\boldsymbol{A}^T\tau})\mathrm{d}\tau\boldsymbol{W}_c(t_z)^{-1}\left[\boldsymbol{x}(0) - e^{-\boldsymbol{A}t_z}\boldsymbol{x}(t_z)\right]\right. \\ &= e^{\boldsymbol{A}t_z}\left\{\boldsymbol{x}(0) - \boldsymbol{W}_c(t_z)\boldsymbol{W}_c(t_z)^{-1}\left[\boldsymbol{x}(0) - e^{-\boldsymbol{A}t_z}\boldsymbol{x}(t_z)\right]\right\} \\ &= \boldsymbol{x}(t_z) \end{aligned} \tag{4.52}$$

from the solution equation of the state equation. Therefore, we can show that $\boldsymbol{W}_c(t)$ is regular.

Assuming that the matrix $\boldsymbol{W}_c(t)$ is not regular for some t, there exists a vector $\boldsymbol{y} \neq \boldsymbol{0}$ such that

$$\boldsymbol{y}^T\boldsymbol{W}_c(t)\boldsymbol{y} = 0, \tag{4.53}$$

and since

$$0 = \boldsymbol{y}^T\boldsymbol{W}_c(t)\boldsymbol{y} = \int_0^t \boldsymbol{y}^T e^{-\boldsymbol{A}\tau}\boldsymbol{B}\boldsymbol{B}^T e^{-\boldsymbol{A}^T\tau}\boldsymbol{y}\mathrm{d}\tau = \int_0^t \left|\boldsymbol{y}^T e^{-\boldsymbol{A}\tau}\boldsymbol{B}\right|\mathrm{d}\tau, \tag{4.54}$$

so

$$\boldsymbol{y}^T e^{-\boldsymbol{A}\tau}\boldsymbol{B} = \boldsymbol{0} \tag{4.55}$$

holds for any τ. Therefore, we have

$$\boldsymbol{y}^T\boldsymbol{B} = \boldsymbol{0}, \tag{4.56}$$

with $\tau = 0$, and

$$\boldsymbol{y}^T\boldsymbol{A}\boldsymbol{B} = \boldsymbol{0}, \tag{4.57}$$

when $\boldsymbol{y}^T e^{-\boldsymbol{A}\tau}\boldsymbol{B}$ is differentiated by τ and

$$\boldsymbol{y}^T\boldsymbol{B} = \boldsymbol{y}^T\boldsymbol{A}\boldsymbol{B} = \boldsymbol{y}^T\boldsymbol{A}^2\boldsymbol{B} = \cdots = \boldsymbol{y}^T\boldsymbol{A}^{n-1}\boldsymbol{B} = \boldsymbol{0}, \tag{4.58}$$

repeating this differentiation. Therefore,

$$\boldsymbol{y}^T\begin{bmatrix}\boldsymbol{B} & \boldsymbol{A}\boldsymbol{B} & \cdots & \boldsymbol{A}^{n-1}\boldsymbol{B}\end{bmatrix} = \boldsymbol{0} \tag{4.59}$$

are obtained, which contradicts the assumption that rank$\boldsymbol{U}_c = n$. Therefore, the matrix $\boldsymbol{W}_c(t)$ is regular for any t (sufficient condition).

4.4.2.2 Observability

A system is said to be observable when the components of all state variables of the system at the start of observation can be known by observing the output of the system for a finite time, and is said to be unobservable when this is not the case. The system is observable if the initial state $\boldsymbol{x}(0)$ can be uniquely obtained from the input $\boldsymbol{u}(t)$ and output $\boldsymbol{y}(t)$ from the start of observation to the finite time t_z. If the system is observable, the transition of all states $\boldsymbol{x}(t)$ in a finite time interval $0 \leq t \leq t_z$ can be calculated from the transition of input $\boldsymbol{u}(t)$ and output $\boldsymbol{y}(t)$ in that time interval. Note that observability assumes that the inputs are known and is determined only by the matrices $\boldsymbol{A}$ and $\boldsymbol{C}$ (not by $\boldsymbol{B}$ or $\boldsymbol{D}$). In addition, the necessary and sufficient conditions for being observable are as follows.

Necessary and sufficient conditions for the system to be observable

$$\text{rank } \boldsymbol{U}_o = n, \tag{4.60}$$

for an observability matrix$\boldsymbol{U}_o((ln) \times n)$

$$\boldsymbol{U}_o = \begin{bmatrix} \boldsymbol{C} \\ \boldsymbol{CA} \\ \boldsymbol{CA^2} \\ \vdots \\ \boldsymbol{CA}^{n-1} \end{bmatrix} \tag{4.61}$$

i.e., it has column full rank

Consider sufficiency conditions. Assume that the system is not observable, i.e., the initial state $\boldsymbol{x}(0)$ cannot be uniquely obtained from the input $\boldsymbol{u}(t)$ and output $\boldsymbol{y}(t)$ from the start of observation to a finite time t_z. Let $\boldsymbol{x}_A(0)$、$\boldsymbol{x}_B(0)$ $(\boldsymbol{x}_A(0) \neq \boldsymbol{x}_B(0))$ be the initial state obtained at this time, then

$$\boldsymbol{y}(t) = \boldsymbol{C}e^{\boldsymbol{A}t}\boldsymbol{x}_A(0) + \boldsymbol{C}\int_0^t e^{\boldsymbol{A}(t-\tau)}\boldsymbol{B}\boldsymbol{u}(\tau)\mathrm{d}\tau + \boldsymbol{D}\boldsymbol{u}(t) \tag{4.62}$$

$$\boldsymbol{y}(t) = \boldsymbol{C}e^{\boldsymbol{A}t}\boldsymbol{x}_B(0) + \boldsymbol{C}\int_0^t e^{\boldsymbol{A}(t-\tau)}\boldsymbol{B}\boldsymbol{u}(\tau)\mathrm{d}\tau + \boldsymbol{D}\boldsymbol{u}(t) \tag{4.63}$$

is valid. Since

$$\boldsymbol{C}e^{\boldsymbol{A}t}\left\{\boldsymbol{x}_A(0) - \boldsymbol{x}_B(0)\right\} = \boldsymbol{0} \tag{4.64}$$

from these, $\boldsymbol{z} \equiv \boldsymbol{x}_A(0) - \boldsymbol{x}_B(0)$, then

$$\boldsymbol{C}e^{\boldsymbol{A}t}\boldsymbol{z} = \boldsymbol{0} \tag{4.65}$$

is true for any t. Therefore,

$$\boldsymbol{C}\boldsymbol{z} = \boldsymbol{0} \tag{4.66}$$

with $t = 0$, and

$$\boldsymbol{C}\boldsymbol{A}\boldsymbol{z} = \boldsymbol{0} \tag{4.67}$$

when $\boldsymbol{C}e^{\boldsymbol{A}t}\boldsymbol{z}$ is differentiated by t, and

$$\boldsymbol{Cz} = \boldsymbol{CAz} = \boldsymbol{CA^2z} = \cdots = \boldsymbol{CA^{n-1}z} = \boldsymbol{0}, \tag{4.68}$$

repeated this differentiation are true. So, we get

$$\begin{bmatrix} \boldsymbol{C} \\ \boldsymbol{CA} \\ \vdots \\ \boldsymbol{CA^{n-1}} \end{bmatrix} \boldsymbol{z} = \boldsymbol{M}_o\boldsymbol{z} = \boldsymbol{0}. \tag{4.69}$$

Here, since $\boldsymbol{x}_A(0) \neq \boldsymbol{x}_B(0)$, we know $\boldsymbol{z} \neq \boldsymbol{0}$, but it must be

$$\mathrm{rank}\boldsymbol{M}_o < n, \tag{4.70}$$

for it to be true (sufficient condition). Necessary condition can be traced backwards.

The following code can be used to check controllability and observability in Python.

```
from control.matlab import ss, ctrb, obsv

A = '0 1; -4 5'
B = '0; 1'
C = '1 0'
P = ss(A, B, C, 0)

Uc = ctrb(P.A, P.B)
Uo = obsv(P.A, P.C)

print('Uc=\n',Uc)
print('det(Uc)=', np.linalg.det(Uc))
print('rank(Uc)=', np.linalg.matrix_rank(Uc))

print('Uo=\n',Uo)
print('det(Uo)=', np.linalg.det(Uo))
print('rank(Uo)=', np.linalg.matrix_rank(Uo))
```

Listing 4.12: Controllability and observability

4.4.3 Optimal regulator

Although the pole placement method can be used to determine the state feedback gain, there are some problems such as

- Increasing the real part of the eigenvalues to the negative side speeds up the response, while increasing the feedback gain $\boldsymbol{F}$ and increasing the input.
- Some state variables may appear with a large amplitude of swing.

, so the selection of poles can be difficult. Therefore, we consider a state feedback gain that minimizes these values by setting evaluation indices related to the dynamic characteristics of the system and input energy (LQ optimal control problem).

LQ optimal control problem

The controller that minimizes the quadratic form of the evaluation function

$$J = \int_0^\infty \boldsymbol{x}(t)\boldsymbol{Q}\boldsymbol{x}(t) + u(t)^\top \boldsymbol{R}u(t)\mathrm{d}t \tag{4.71}$$

for $\boldsymbol{Q} = \boldsymbol{Q}^\top > 0$ and $\boldsymbol{R} = \boldsymbol{R}^\top > 0$ is obtained in the form $u = \boldsymbol{F}_{\mathrm{opt}}\boldsymbol{x}$ where $\boldsymbol{F}_{\mathrm{opt}}$ is the value

$$\boldsymbol{F}_{\mathrm{opt}} = -\boldsymbol{R}^{-1}\boldsymbol{B}^\top \boldsymbol{P}. \tag{4.72}$$

However, $\boldsymbol{P} = \boldsymbol{P}^\top > 0$ satisfies Riccati equation

$$\boldsymbol{A}^\top \boldsymbol{P} + \boldsymbol{P}\boldsymbol{A} - \boldsymbol{P}\boldsymbol{B}\boldsymbol{R}^{-1}\boldsymbol{B}^\top \boldsymbol{P} + \boldsymbol{Q} = 0. \tag{4.73}$$

It is the only positive definite symmetric solution. Also, the minimum value of J is $\boldsymbol{x}(0)^\top \boldsymbol{P}\boldsymbol{x}(0)$.

The state feedback control obtained by optimizing the evaluation function is called the optimal regulator. In the optimal control problem, the input u from initial time to infinite time that minimizes the evaluation function J is sought, and the optimal control input is given in the form of a state feedback control. Also, $\boldsymbol{Q}$ is often set in a diagonal matrix like

$$\boldsymbol{Q} = \begin{bmatrix} q_1 & 0 \\ 0 & q_2 \end{bmatrix} \tag{4.74}$$

At this time,

$$\boldsymbol{x}(t)^\top \boldsymbol{Q}\boldsymbol{x}(t) = q_1 x_1(t)^2 + q_2 x_2(t)^2, \tag{4.75}$$

so x_1 should be set so that $q_1 > q_2$ if x_2 is to converge to 0 more quickly. On the other hand, $\boldsymbol{R}$ becomes a scalar when u is one input, and a larger $\boldsymbol{R}$ gives a feedback gain $\boldsymbol{F}$ where the input does not become too large. to design an optimal regulator in Python, simply run the following code, which will produce Figure 4.28 is obtained.

```
# Optimal Regulator
from control.matlab import lqr, care

Q = np.diag([100, 1])
R = 1

F, X, E = lqr(P.A, P.B, Q, R)
F = -F

print('-- Feedback Gain --')
print(F)
print(-(1/R)*P.B.T@X)
print('-- Poles of Closed Loop')
print(E)
print(np.linalg.eigvals(P.A+P.B@F))

Acl = P.A + P.B*F
Pfb = ss(Acl, P.B, P.C, P.D)

Td = np.arange(0, 5, 0.01)
x0 = [-0.3, 0.4]
x, t = initial(Pfb, Td, x0)
```

```
23
24 fig, ax = plt.subplots()
25 ax.plot(t, x[:, 0], label = '$x_1$')
26 ax.plot(t, x[:, 1], label = '$x_2$')
27 plot_set(ax, 't', 'x', 'best')
28
29 #plt.savefig('SFC_OptimalRegulator',dpi=300)
```

Listing 4.13: Optimal regulator

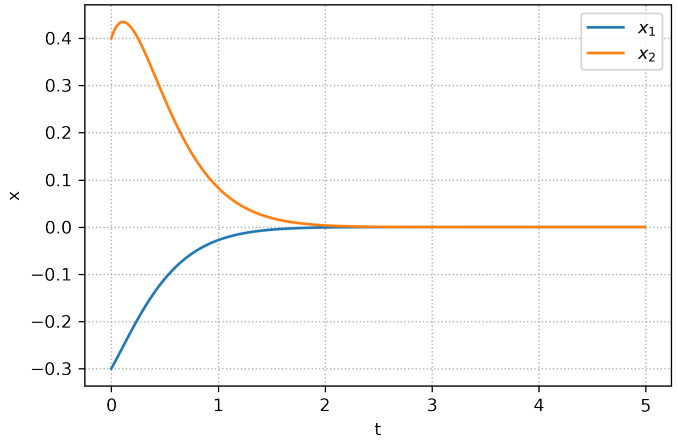

Figure 4.28: State feedback control with optimal regulator

4.4.4 Integral servo system

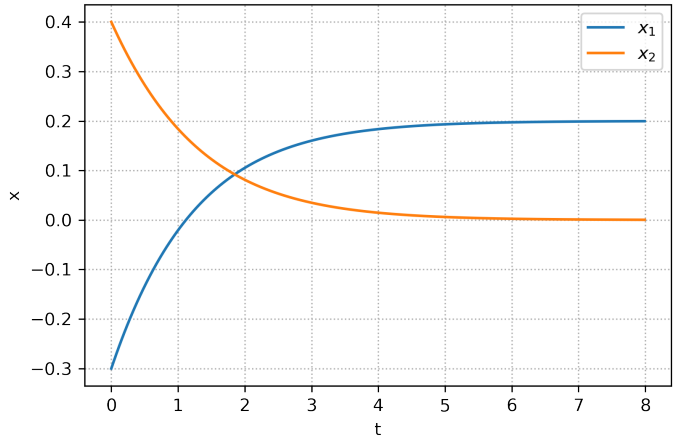

Figure 4.29: State feedback control in the presence of constant disturbance

When a constant disturbance d is added to the control symmetry, the state does not converge to zero (Fig. 4.29). To solve this problem, an integral servo system is sought.

This system integrates the difference between the output y and the target value r and adds it to the input, denoted

$$u(t) = \boldsymbol{F}\boldsymbol{x}(t) + G\int_0^t (r - y(\tau))\mathrm{d}\tau. \tag{4.76}$$

Now consider the case where a constant disturbance d is added to the control target, i.e.,

$$\begin{cases} \dot{\boldsymbol{x}}(t) = \boldsymbol{A}\boldsymbol{x}(t) + \boldsymbol{B}(u(t) + d) \\ y(t) = \boldsymbol{C}\boldsymbol{x}(t) \end{cases} \tag{4.77}$$

If the state of the integrator is w, then $\dot{w}(t) = r - y(t) = r - \boldsymbol{C}\boldsymbol{x}(t)$. Using this to expand the control system and setting $r = 0$ yields

$$\begin{cases} \begin{bmatrix} \dot{\boldsymbol{x}}(t) \\ \dot{w}(t) \end{bmatrix} = \begin{bmatrix} \boldsymbol{A} & 0 \\ -\boldsymbol{C} & 0 \end{bmatrix} \begin{bmatrix} \boldsymbol{x}(t) \\ w(t) \end{bmatrix} + \begin{bmatrix} \boldsymbol{B} \\ 0 \end{bmatrix} u(t) + \begin{bmatrix} \boldsymbol{B} \\ 0 \end{bmatrix} d \\ y(t) = \begin{bmatrix} \boldsymbol{C} & 0 \end{bmatrix} \begin{bmatrix} \dot{\boldsymbol{x}}(t) \\ \dot{w}(t) \end{bmatrix} \end{cases} \tag{4.78}$$

Or, if

$$\boldsymbol{x}_e = \begin{bmatrix} \dot{\boldsymbol{x}}(t) \\ \dot{w}(t) \end{bmatrix},\ \boldsymbol{A}_e = \begin{bmatrix} \boldsymbol{A} & 0 \\ -\boldsymbol{C} & 0 \end{bmatrix},\ \boldsymbol{B}_e = \begin{bmatrix} \boldsymbol{B} \\ 0 \end{bmatrix},\ \boldsymbol{C}_e = \begin{bmatrix} \boldsymbol{C} & 0 \end{bmatrix} \tag{4.79}$$

, it becomes

$$\begin{cases} \dot{\boldsymbol{x}}_e(t) = \boldsymbol{A}_e\boldsymbol{x}_e(t) + \boldsymbol{B}_e(u(t) + d) \\ y(t) = \boldsymbol{C}_e\boldsymbol{x}_e(t) \end{cases} \tag{4.80}$$

Since the control law at this time is

$$u(t) = \begin{bmatrix} \boldsymbol{F} & G \end{bmatrix} \begin{bmatrix} \boldsymbol{x}(t) \\ w(t) \end{bmatrix}, \tag{4.81}$$

$\boldsymbol{F}_e = \begin{bmatrix} \boldsymbol{F} & G \end{bmatrix}$ is the state feedback gain for the expanded system (the design method is the same as for ordinary state feedback control). This is used in the following code, which, when executed, yields Figure 4.30.

```
# Integral Servo System
from control.matlab import ss, acker, lsim

A = '0 1; -4 5'
B = '0; 1'
C = '1 0; 0 1'
D = '0; 0'
P = ss(A, B, C, D)

Pole = [-1, -1]
F = -acker(P.A, P.B, Pole)
Acl = P.A + P.B@F
Pfb = ss(Acl, P.B, P.C, P.D) # stabilized with state feedback

Td = np.arange(0,8,0.01)
Ud = 0.2 * (Td>=0)
```

```
x, t, _ = lsim(Pfb, Ud, Td, [-0.3, 0.4])

fig, ax = plt.subplots()
ax.plot(t, x[:,0], label='$x_1$')
ax.plot(t, x[:,1], label='$x_2$')

plot_set(ax, 't', 'x', 'best')

#plt.savefig('FB_notConvergenceZero',dpi=300)

A2 = '0 1; -4 5'
B2 = '0; 1'
C2 = '1 0'
D2 = '0'
P2 = ss(A2, B2, C2, D2)

Ae = np.block([[P2.A, np.zeros((2,1))], [-P2.C, 0]])
Be = np.block([[P2.B], [0]])
Ce = np.block([P2.C ,0])

Pole2 = [-1, -1, -5]
Fe = -acker(Ae, Be, Pole2)

Acl2 = Ae + Be@Fe
Pfb2 = ss(Acl2, Be, np.eye(3), np.zeros((3,1)))

x2, t2, _ = lsim(Pfb2, Ud, Td, [-0.3, 0.4, 0])

fig, ax = plt.subplots()
ax.plot(t2, x2[:,0], label='$x_1$')
ax.plot(t2, x2[:,1], label='$x_2$')

plot_set(ax, 't', 'x', 'best')

#plt.savefig('integral_servo_system',dpi=300)
```

Listing 4.14: Integral servo system

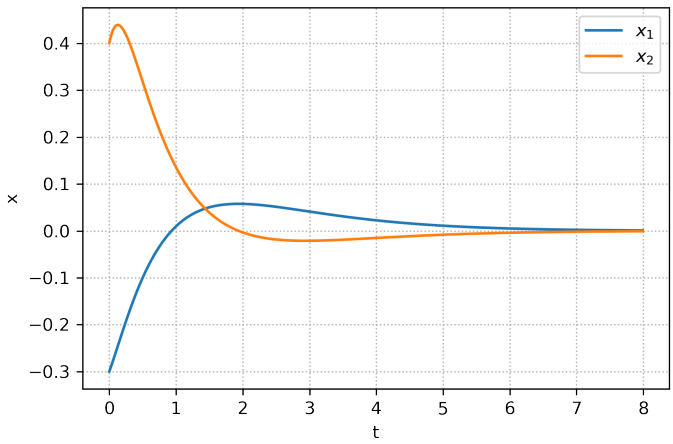

Figure 4.30: Integral servo system

This shows that the state converges to 0 for a constant value disturbance. Also, by constructing an integral servo system, the output y can be made to follow a constant target value r.

Chapter 5

SYSTEM DESIGN (OL)

5.1 Control specification for open loop

In chapter 4, we considered transfer function from reference to output

$$\mathcal{G}_{yr} = \frac{\mathcal{P}(s)\mathcal{K}}{1 + \mathcal{P}(s)\mathcal{K}} \tag{5.1}$$

in closed loop system. However, this is nonlinear with respect to the controller and the controlled object, so it is difficult to see at a glance how a change in $\mathcal{K}(s)$ or uncertainty in $\mathcal{P}(s)$ changes the characteristics of the closed loop.

Therefore, the controller is designed by cutting the loop and looking at the characteristics of the open loop

$$\mathcal{H}(s) = \mathcal{P}(s)\mathcal{K}(s). \tag{5.2}$$

Since this is linear with respect to the controller and the control target, we can immediately determine how to change $\mathcal{K}(s)$ to satisfy the control specification and how much the uncertainty of $\mathcal{P}(s)$ affects it. For example, even if the model $\mathcal{P}(s)$ of the control target is not known, control design is possible if the frequency response of $\mathcal{H}(s)$ is known. In addition, the frequency characteristics (gain and phase) of multiple elements coupled in series can be obtained by adding up the frequency characteristics of each element, and controller design is often easier than in closed-loop applications.

Therefore, in this chapter, the design specifications for closed-loop systems described in Chapter 4 are rewritten for open-loop systems. Unless otherwise stated, $\mathcal{H}(s) = \mathcal{P}(s)\mathcal{K}(s)$ is stable and there is no zero cancellation of unstable poles between $\mathcal{P}(s)$ and $\mathcal{K}(s)$.

5.1.1 Stability

The stability of the open loop is determined using the **Nyquist Stability Discriminant Method**. This is approximately as follows.

> Nyquist Stability Discriminant Method
>
> When $\mathcal{H}(s) = \mathcal{P}(s)\mathcal{K}(s)$ is stable, at its frequency response, the open-loop system is stable if the phase crossing frequency ω_{pc} is greater than the gain crossing frequency ω_{gc}.

Here, the phase crossing frequency ω_{pc} is the frequency where $\angle\mathcal{H}(j\omega) = -180\mathrm{deg} = \pi\mathrm{rad}$, and the gain crossing frequency ω_{gc} is the frequency whee $|\mathcal{H}(j\omega)| = 1$.

For example, let us apply a sin signal $u(t) = \sin\omega_{\mathrm{pc}}$ with frequency ω_{pc} to a stable system $\mathcal{H}(s)$. If the phase is delayed by 180deg, or π rad, the steady-state output of the system is

$$y(t) = |\mathcal{H}(j\omega_{\mathrm{pc}})|\sin(\omega_{\mathrm{pc}}t - \pi) = -|\mathcal{H}(j\omega_{\mathrm{pc}})|\sin(\omega_{\mathrm{pc}}t). \tag{5.3}$$

When a new input is added to the system as

$$u(t) = \sin\omega_{\mathrm{pc}}t + |\mathcal{H}(j\omega_{\mathrm{pc}})|\sin(\omega_{\mathrm{pc}}t) = (1 + |\mathcal{H}(j\omega_{\mathrm{pc}})|)\sin\omega_{\mathrm{pc}}t \tag{5.4}$$

by negative feedback, the output is

$$y(t) = -(|\mathcal{H}(j\omega_{\mathrm{pc}})| + |\mathcal{H}(j\omega_{\mathrm{pc}})|^2)\sin(\omega_{\mathrm{pc}}t). \tag{5.5}$$

Strictly speaking, the output is negatively feedbacked at each time when a closed loop is constructed, however, for simplicity, we consider here that the feedback loop is connected only when a new input is generated using the steady-state output. Then, by repeating this, the amplitude of the input signal becomes

$$1 + |\mathcal{H}(j\omega_{\mathrm{pc}})| + |\mathcal{H}(j\omega_{\mathrm{pc}})|^2 + \cdots \tag{5.6}$$

If $|\mathcal{H}(j\omega_{\mathrm{pc}})| \geq 1$, the input and output signals diverge, and if $|\mathcal{H}(j\omega_{\mathrm{pc}})| < 1$, then

$$1 + |\mathcal{H}(j\omega_{\mathrm{pc}})| + |\mathcal{H}(j\omega_{\mathrm{pc}})|^2 + \cdots = \frac{1}{1 - |\mathcal{H}(j\omega_{\mathrm{pc}})|} \tag{5.7}$$

B and that the input and output converge to some bounded value.

To consider a concrete example, execute the following code to obtain figures 5.1 and 5.2.

```
# Change in amplitude of output with a sine wave input
import matplotlib.pyplot as plt
import numpy as np

from control.matlab import tf, lsim, margin

P = tf([0,1], [1, 1, 1.5, 1])
P2 = tf([0,1], [1, 2, 2, 1])
_, _, wpc, _ = margin(P) # phase crossover frequency
_, _, wpc2, _ = margin(P2) # phase crossover frequency

```

```
t = np.arange(0, 30, 0.1)
u = np.sin(wpc*t)
y = np.zeros_like(t)

t2 = np.arange(0, 30, 0.1)
u2 = np.sin(wpc*t2)
y2 = np.zeros_like(t2)

fig, ax = plt.subplots(2,2,figsize=(8.0, 6.0))

for i in range(2):
    for j in range(2):
        u = np.sin(wpc*t) - y # negative feedback
        y, t, _ = lsim(P, u, t, 0)

        ax[i,j].plot(t, u, label='u')
        ax[i,j].plot(t, y, label='y')
        plot_set(ax[i,j], 't', 'u, y', 'lower left')

fig.tight_layout

#plt.savefig('change_in_amp_divergence.png', dpi=300)


fig, ax = plt.subplots(2,2,figsize=(8.0, 6.0))

for i in range(2):
    for j in range(2):
        u2 = np.sin(wpc2*t2) - y2 # negative feedback
        y2, t2, _ = lsim(P2, u2, t2, 0)

        ax[i,j].plot(t2, u2, label='u')
        ax[i,j].plot(t2, y2, label='y')
        plot_set(ax[i,j], 't', 'u, y', 'lower left')

fig.tight_layout

#plt.savefig('change_in_amp_convergence.png', dpi=300)
```

Listing 5.1: Amplitude change of output due to sin wave input

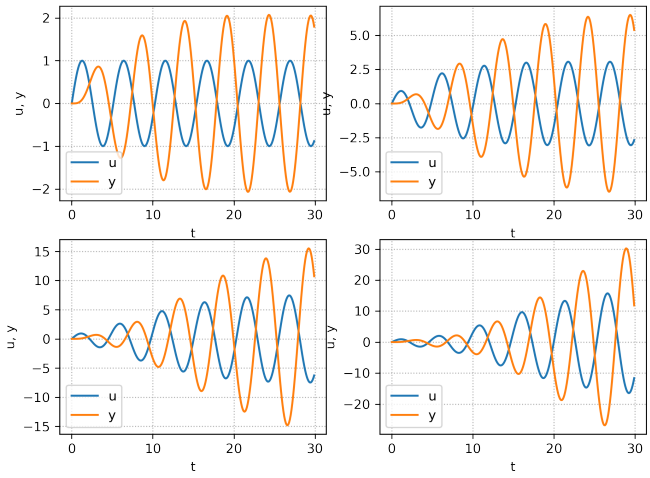

Figure 5.1: Amplitude of output diverges

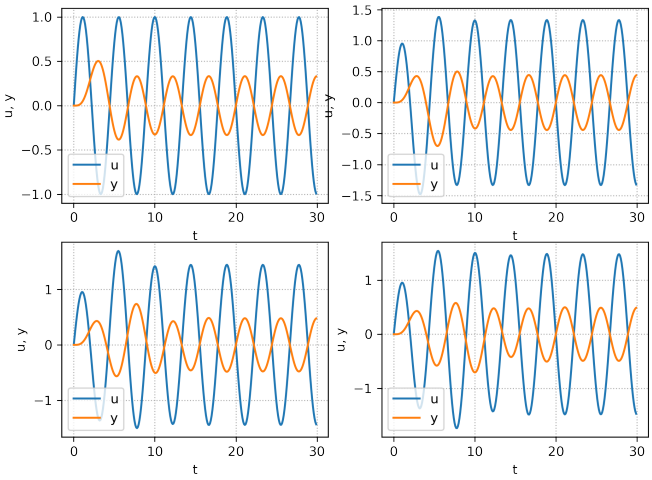

Figure 5.2: Amplitude of output converges

First, Figure 5.1 shows that the phases of the input and output signals are 180 deg out of phase. In addition, the figure can be viewed as follows: y in the upper left block is used to generate u in the upper right block, resulting in $u = y + \sin \omega_{\mathrm{pc}} t$, y in the upper right block is used to generate u in the lower left block, and so on. In this case, the amplitude of the output signal (orange) is larger than the amplitude of the input signal (blue), and we can see that the output amplitude diverges. In other words, in such a case, the system becomes unstable when a closed-loop system is set up.

Therefore, the case where the control target is changed to `P2 = tf([0,1], [1, 2, 2, 2, 1]` is shown in Figure 5.2. Although the phase is shifted 180deg as before, the amplitude of the output signal is smaller than the amplitude of the input signal, and the output amplitude converges to a certain value without diverging as the input is updated. In such a case, the system is stable.

Now, the closed loop becomes stable when $|\mathcal{H}(j\omega_{\mathrm{pc}})| < 1$ at a frequency ω_{pc} where the phase is delayed by 180deg. Also, when the control target is a physical system, in most cases $\mathcal{H}(j\omega_{\mathrm{pc}})| \to 0$ $(\omega \to \infty)$, so $\omega_{\mathrm{pc}} > \omega_{\mathrm{gc}}$ is stable, which is called Nyquist's stability discriminant.

A visual representation of this is called a Nyquist diagram. This is a plot of $|\mathcal{H}(j\omega)|$ when ω is changed from $-\infty$ to ∞ on the complex number plane. And the gain $|\mathcal{H}(j\omega_{\mathrm{pc}})|$ when the locus intersects the real axis is less than 1, or in simple terms, if the Nyquist diagram is to the right of the point $(-1, j0)$, it is stable, otherwise it is unstable. The Nyquist diagram can be drawn by drawing

$$\mathcal{H}(j\omega) = \alpha(\omega) + j\beta(\omega) \tag{5.8}$$

A with ω varying from $-\infty$ to ∞ in Python as follows.

```python
# Nyquist Diagram
from control.matlab import tf, nyquist, logspace

fig, ax = plt.subplots(1,2)

P = tf([0, 1], [1, 1, 1.5, 1]) # unstable
x, y, _ = nyquist(P, logspace(-3, 5, 1000), plot=False)
ax[0].plot(x, y)
ax[0].plot(x, -y)
ax[0].scatter(-1, 0)
plot_set(ax[0], 'Re', 'Im')

P2 = tf([0, 1], [1, 2, 2, 1]) # stable
x, y, _ = nyquist(P2, logspace(-3, 5, 1000), plot=False)
ax[1].plot(x, y)
ax[1].plot(x, -y)
ax[1].scatter(-1, 0)
plot_set(ax[1], 'Re', 'Im')

fig.tight_layout()

#plt.savefig('nyquist.png', dpi=300)
```

Listing 5.2: Nyquist diagram

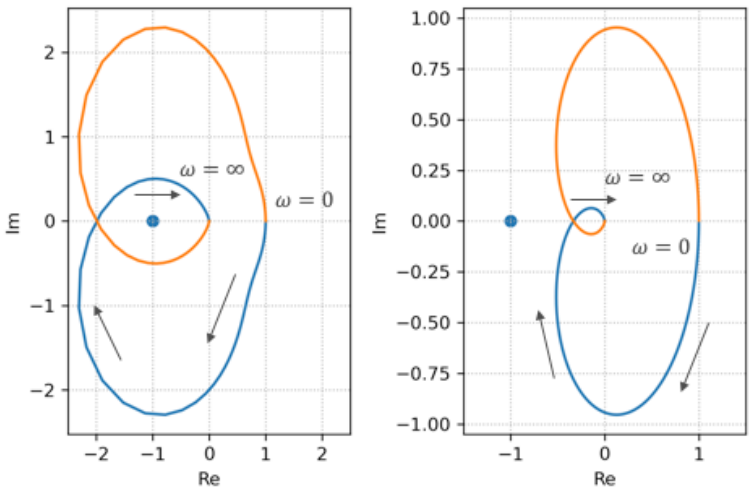

Figure 5.3: Nyquist diagram

In the left figure 5.3, when ω is changed from 0 to ∞, the closed-loop system is unstable because the trajectory goes around the left side of the point $(-1, j0)$. On the other hand, in the figure on the right, the trajectory goes around the right side of the point $(-1, j0)$, so the closed-loop system is stable.

In addition, when considered in correspondence with the Bode diagram, Figure 5.4 is shown. Since $\mathcal{H} = \mathcal{P}\mathcal{K}$, increasing the gain of the controller $\mathcal{K}$ increases the gain of $\mathcal{H}$ and the gain diagram moves up in parallel. Then ω_{gc} increases, but the phase does not change.

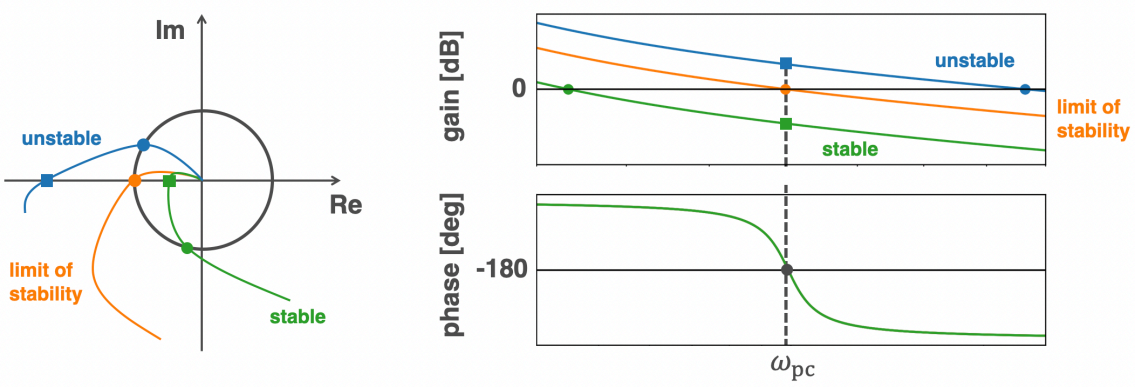

Figure 5.4: Open loop TF

5.1.2 Quick-response

First, let us discuss the phase margin $\phi_{\rm pm}$, which will appear frequently from this point onward. This is the degree to which the phase $\angle\mathcal{H}(j\omega_{\rm gc})$ of the open loop $\mathcal{H}$ at the gain crossing frequency $\omega_{\rm gc}$ is lead from -180deg and is expressed as

$$\phi_{\rm pm} = 180 + \angle\mathcal{H}(j\omega_{\rm gc}), \tag{5.9}$$

(Figure **??**). In other words, it represents how far $\mathcal{H}(j\omega_{\rm gc})$ is from the point $(-1, j0)$. The larger the phase margin, the less likely it is to become unstable even if the parameter being controlled fluctuates.

There is also a gain margin $G_{\rm gm}$, which indicates how far $\mathcal{H}(j\omega_{\rm pc})$ is from the point $(-1, j0)$ at the phase crossing frequency $\omega_{\rm pc}$. In other words, it means how many times $|\mathcal{H}(j\omega_{\rm pc})|$ is equal to 1, and it takes a larger value the further it is from the point $(-1, j0)$ and the closer it is to the origin. If $\rho = \frac{1}{|\mathcal{H}(j\omega_{\rm pc})|}$, the gain margin is

$$G_{\rm gm} = 20\log_{10}\rho. \tag{5.10}$$

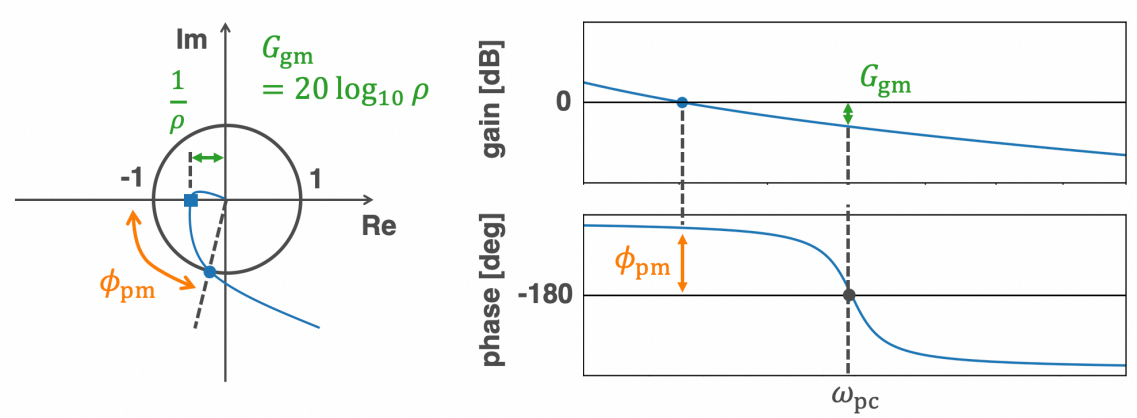

Figure 5.5: Phase margin · Gain margin

Now, in Chapter 5, we confirmed that the larger the bandwidth $\omega_{\rm bw}$ is, the better the quick response is in a closed loop. For example, if $|\mathcal{G}_{yr}(0)| = 1$, the bandwidth of $\mathcal{G}_{yr}$ is the frequency such that $|\mathcal{G}_{yr}(j\omega_{\rm bw})| = \frac{1}{\sqrt{2}}$.

On the other hand, for the open-loop system $\mathcal{H}(j\omega)$, we know $|\mathcal{H}(j\omega_{\rm gc})| = 1$, so

$$|\mathcal{G}_{yr}(j\omega_{\rm gc})| = \frac{1}{|1 + \mathcal{H}(j\omega_{\rm gc})|}, \tag{5.11}$$

and from the picture on the left side of Figure 5.5,

$$|1 + \mathcal{H}(j\omega_{\rm gc})| = 2\sin\frac{\phi_{\rm pm}}{2}, \tag{5.12}$$

so

$$|\mathcal{G}_{yr}(j\omega_{\rm gc})| = \frac{1}{2\sin\frac{\phi_{\rm pm}}{2}}. \tag{5.13}$$

Therefore, when $\phi_{\rm pm} = 90$deg, it is

$$|\mathcal{G}_{yr}(j\omega_{\rm gc})| = \frac{1}{\sqrt{2}}, \tag{5.14}$$

and $\omega_{\rm gc} = \omega_{\rm bw}$. On the other hand, when $\phi_{\rm pm} < 90$deg, it is

$$|\mathcal{G}_{yr}(j\omega_{\rm gc})| > \frac{1}{\sqrt{2}}, \tag{5.15}$$

so $\omega_{\rm gc} < \omega_{\rm bw}$.

From the above, $\phi_{\rm pm} \leq 90$, $\omega_{\rm gc} \leq \omega_{\rm bw}$. Therefore, if the gain crossing frequency $\omega_{\rm gc}$ of the open-loop system $\mathcal{H}$ is increased, bandwidth of the closed-loop system $\omega_{\rm bw}$ becomes larger, resulting in a faster response.

5.1.3 Damping

Next, consider the damping property. In a closed-loop system, the smaller the peak gain M_p of $\mathcal{G}_{yr}$, the better the damping (Chapter 5). The peak gain of $\mathcal{G}_{yr}$ is

$$M_p = \max_{\omega} |\mathcal{G}_{yr}(j\omega)|. \tag{5.16}$$

From this and equation 5.13,

$$M_p > |\mathcal{G}_{yr}(j\omega_{\rm gc})| = \frac{1}{2\sin\frac{\phi_{\rm pm}}{2}} \tag{5.17}$$

is obtained. From this, if the phase margin $\phi_{\rm pm}$ is small, $|\mathcal{G}_{yr}(j\omega_{\rm gc})|$ becomes large, and it is easy to become oscillatory. In other words, to improve the damping property, the phase margin $\phi_{\rm pm}$ should be increased.

5.1.4 Steady-state properties

Since the transfer function $\mathcal{G}_{er}(s)$ from the reference r to the deviation e is

$$\mathcal{G}_{er}(s) = \frac{1}{1 + \mathcal{H}(s)}, \tag{5.18}$$

the steady-state deviation from the step target value is

$$e(\infty) = \frac{1}{1 + \mathcal{H}(0)} \tag{5.19}$$

Therefore, to reduce the steady-state deviation, increase the low-frequency gain

$$\lim_{\omega \to 0} |\mathcal{H}(j\omega)| \tag{5.20}$$

of the open-loop system $\mathcal{H}$, that is, increase the DC gain $|\mathcal{H}(0)|$.

When $\mathcal{K}$ includes an integrator, $|\mathcal{H}(0)| = \infty$, and the step target value is followed without steady-state deviation (type 1 control system). More generally, when the number of integrators is i, it is called an i-type control system. When the Laplace transform of the reference is $\frac{1}{s^i}$, the steady-state deviation becomes 0 for an i-type control system.

5.1.5 Summary

Here, we summarize the design specifications for open-loop systems.

Stability : Keep gain crossing frequency ω_{gc}<phase crossing frequency ω_{pc}.

Quick-response : Make the gain cross frequency ω_{gc} as large as possible.

Damping : Make the phase margin ϕ_{pm} large.

Steady-state properties : Increase low-frequency gain (set $|\mathcal{H}(0)| = \infty$ for DC gain)

If you want to find gain margin, phase margin, phase crossing frequency, and gain crossing frequency in Python, you can use the `margin` function and write `gm, pm, wpc, wgc = margin(sys)`. To obtain the gain and phase for a specific frequency, use the `freqresp` function and write `mag, phase, w = freqresp(sys, omega)`. Specific code examples are shown in the following sections.

5.2 PID control (open-loop characteristics)

As in Chapter 5, the following vertical drive arm model is considered to confirm the open-loop characteristics of PID control.

```
# Params for Vertical Drive Arm
from control.matlab import tf

g  = 9.81   # gravitational acceleration
l  = 0.2    # arm length
M  = 0.5    # arm mass
mu = 1.5e-2 # coefficient of viscous friction
J  = 1.0e-2 # moment of inertia

P = tf([0,1],[J,mu,M*g*l])

ref = 30 # reference angle
```

Listing 5.3: Vertical drive arm model

5.2.1 P control

When considering the proportional gain of $\mathcal{K}(s) = k_P$, the following code confirms what happens to the characteristics of the open-loop transfer function $\mathcal{H}(s) = \mathcal{P}(s)\mathcal{K}(s)$. This yields Figure 5.6.

```
# P Control (OL)
import matplotlib.pyplot as plt
import numpy as np
from control.matlab import tf, bode, logspace, mag2db, margin, feedback, step

kp = (0.5, 1, 2)

fig, ax = plt.subplots(2,1)

for i in range(len(kp)):
    K = tf([0, kp[i]], [0,1]) # P controller
    H = P * K # open loop

    mag, phase, w = bode(H, logspace(-1,2,1000), plot=False)
    pltargs = {'label':f'$k_P$={kp[i]}'}
    ax[0].semilogx(w, mag2db(mag), **pltargs)
    ax[1].semilogx(w, np.rad2deg(phase), **pltargs)

    gm, pm, wpc, wgc = margin(H)
    ax[0].scatter(wgc, 0)
    print('kP=', kp[i])
    print('(GM, PM, wpc, wgc)')
    print(margin(H))
    print('----------------')

bodeplot_set(ax, 3)

#plt.savefig('Pcontrol_ol.png', dpi=300)
```

Listing 5.4: P control (open-loop)

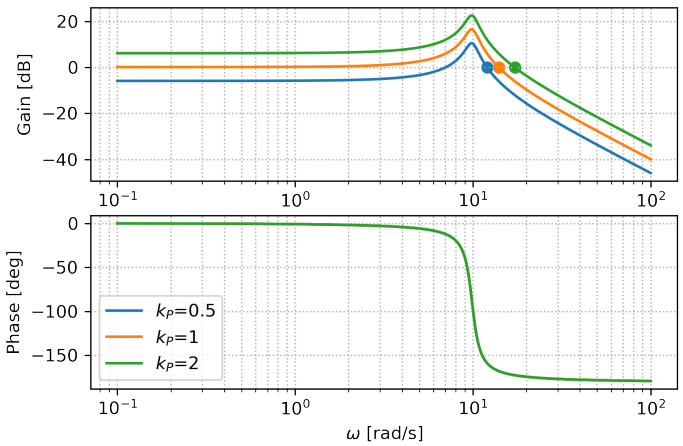

Figure 5.6: Bode plot for P control (open-loop)

The circles in Figure 5.6 indicate the gain crossing frequency. From this figure, it can be seen that the gain crossing frequency increases as the proportional gain k_P is increased. On the other hand, it also shows that the phase margin becomes smaller as the proportional gain k_P is increased. In other words, increasing the proportional gain makes the response faster, but the response becomes oscillatory. The low-frequency gain also increases along with the proportional gain k_P, but because it is not ∞, the steady-state deviation remains. This is consistent with the results in Figure 4.7.

5.2.2 PI control

Next, for PI control, we examine the open-loop transfer function characteristics when the proportional gain k_P of $\mathcal{K}(s) = k_P + \frac{k_I}{s}$ is fixed and the integral gain k_I is varied. Figures 5.7 and 5.8 are obtained by executing the following code.

```
# PI Control (OL)
import matplotlib.pyplot as plt
import numpy as np
from control.matlab import tf, bode, logspace, mag2db, margin, feedback, step

kp = 2
ki = (0, 5, 10)

fig, ax = plt.subplots(2,1)

for i in range(len(ki)):
    K = tf([kp, ki[i]], [1,0]) # PI controller
    H = P * K # open loop

    mag, phase, w = bode(H, logspace(-1,2,1000), plot=False)
    pltargs = {'label':f'$k_I$={ki[i]}'}
    ax[0].semilogx(w, mag2db(mag), **pltargs)
    ax[1].semilogx(w, np.rad2deg(phase), **pltargs)

    gm, pm, wpc, wgc = margin(H)
```

```
    ax[0].scatter(wgc, 0)
    print('kP=', kp, ', kI=', ki[i])
    print('(GM, PM, wpc, wgc)')
    print(margin(H))
    print('-----------------')

bodeplot_set(ax, 3)

#plt.savefig('PIcontrol_ol.png', dpi=300)

fig, ax = plt.subplots()

for i in range(len(ki)):
    K = tf([kp, ki[i]], [1,0])
    Gyr = feedback(P*K, 1) # closed loop
    y, t = step(Gyr, np.arange(0, 2, 0.01)) # step responce

    pltargs = {'label':f'$k_I$={ki[i]}'}
    ax.plot(t, y*ref, **pltargs)

ax.axhline(ref, color='k', linewidth=1)
plot_set(ax, 't', 'y', 'best')

#plt.savefig('StepResponse_PI.png', dpi=300)
```

Listing 5.5: PI control (open-loop)

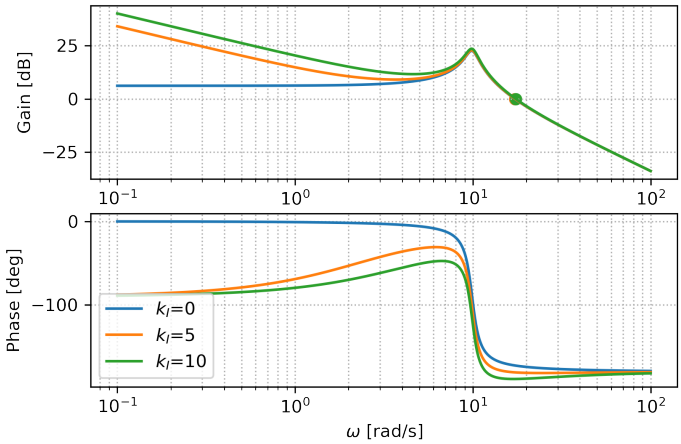

Figure 5.7: Bode plot for PI control (open-loop)

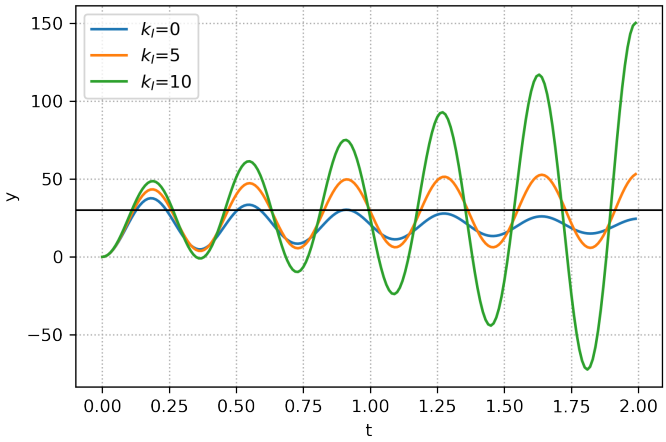

Figure 5.8: Step response of closed-loop system with PI control

First, from Figure 5.7, it can be seen that the low-frequency gain increases as the integral gain k_I is increased. Also, since the DC gain is ∞, the steady-state deviation from the step target value is 0.

On the other hand, as the integral gain k_I is increased, the phase margin becomes smaller, eventually falling below 0 deg and $\omega_{\rm pc} > \omega_{\rm gc}$ is no longer valid. This means that the closed-loop system becomes unstable. In other words, increasing the integral gain k_I can reduce the steady-state deviation, but the response becomes oscillatory and unstable. This is illustrated in Fig. 5.8. The step response of the closed-loop system $\mathcal{G}_{yr}(s)$ becomes unstable as the integral gain k_I increases.

5.2.3 PID control

Finally, consider the PID control $\mathcal{K}(s) = k_P + \frac{k_I}{s} + k_D s$ with D control added, and also examine the open-loop characteristics when the differential gain k_D is varied with the proportional gain k_P and integral gain k_I fixed. Execute the following code to obtain Figures 5.9 and 5.10.

```
# PID Control (OL)
import matplotlib.pyplot as plt
import numpy as np
from control.matlab import tf, bode, logspace, mag2db, margin, feedback, step

kp = 2
ki = 5
kd = (0, 0.1, 0.2)

fig, ax = plt.subplots(2,1)

for i in range(len(kd)):
    K = tf([kd[i], kp, ki], [1,0]) # PID controller
    H = P * K # open loop

    mag, phase, w = bode(H, logspace(-1,2,1000), plot=False)
```

```
    pltargs = {'label':f'$k_D$={kd[i]}'}
    ax[0].semilogx(w, mag2db(mag), **pltargs)
    ax[1].semilogx(w, np.rad2deg(phase), **pltargs)

    gm, pm, wpc, wgc = margin(H)
    ax[0].scatter(wgc, 0)
    print('kP=', kp, ', kI=', ki, ', kD=', kd[i])
    print('(GM, PM, wpc, wgc)')
    print(margin(H))
    print('-----------------')

bodeplot_set(ax, 3)

#plt.savefig('PIDcontrol_ol.png', dpi=300)

fig, ax = plt.subplots()

for i in range(len(kd)):
    K = tf([kd[i], kp, ki], [1,0])
    Gyr = feedback(P*K, 1) # closed loop
    y, t = step(Gyr, np.arange(0, 2, 0.01)) # step responce

    pltargs = {'label':f'$k_D$={kd[i]}'}
    ax.plot(t, y*ref, **pltargs)

ax.axhline(ref, color='k', linewidth=1)
plot_set(ax, 't', 'y', 'best')

#plt.savefig('StepResponse_PID.png', dpi=300)
```

Listing 5.6: PID control (open-loop)

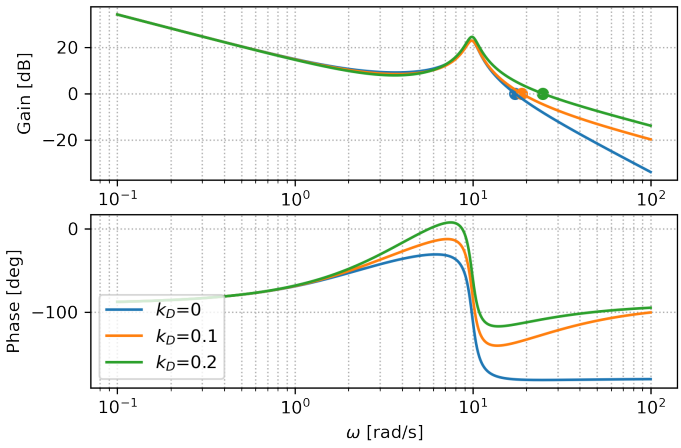

Figure 5.9: Bode plot for PID control (open-loop)

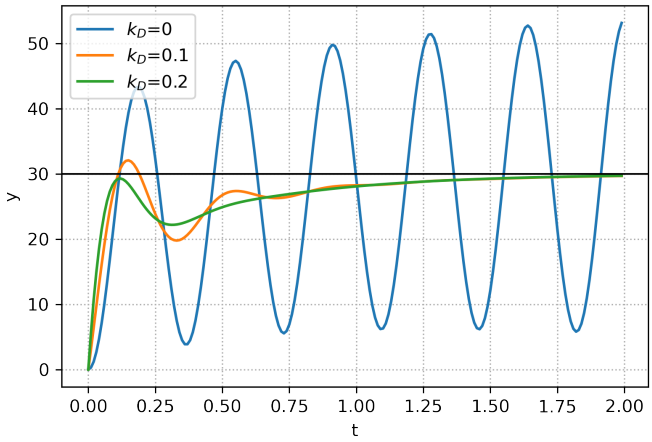

Figure 5.10: Step response of closed-loop system with PID control

First, from Figure 5.9, we can see that increasing the differential gain k_D increases the phase margin and prevents instability. In addition, the addition of differential gain k_D does not change the low-frequency gain. This indicates that the addition of D control reduces oscillation, but does not improve the steady-state characteristics. This can also be seen from the step response of the closed-loop system (Figure 5.10).

5.2.4 Summary

To summarize the above, the performance of the two controllers

- P control : $k_P = 1,\ k_I = 0,\ k_D = 0$
- PID control : $k_P = 2,\ k_I = 5,\ k_D = 0.1$

are compared using the following code to obtain Figures 5.11, 5.12 and 5.13.

```
# Comparison
kp = (1, 2)
ki = (0, 5)
kd = (0, 0.1)
Label = ('Before','After')

fig, ax = plt.subplots(2, 1)

for i in range(2):
    K = tf([kd[i], kp[i], ki[i]], [1,0])
    H = P * K

    mag, phase, w = bode(H, logspace(-1,2,1000), plot=False)
    pltargs = {'label':Label[i]}
    ax[0].semilogx(w, mag2db(mag), **pltargs)
    ax[1].semilogx(w, np.rad2deg(phase), **pltargs)

    gm, pm, wpc, wgc = margin(H)
    ax[0].scatter(wgc, 0)

bodeplot_set(ax, 3)
```

```
22
23 #plt.savefig('Comparison_OL.png', dpi=300)
24
25 fig, ax = plt.subplots(2, 1)
26
27 for i in range(2):
28     K = tf([kd[i], kp[i], ki[i]], [1,0])
29     Gyr = feedback(P*K, 1)
30
31     mag, phase, w = bode(Gyr, logspace(-1,2,1000), plot=False)
32     pltargs = {'label':Label[i]}
33     ax[0].semilogx(w, mag2db(mag), **pltargs)
34     ax[1].semilogx(w, np.rad2deg(phase), **pltargs)
35
36 bodeplot_set(ax, 3)
37
38 #plt.savefig('Comparison_CL.png', dpi=300)
39
40 fig, ax = plt.subplots()
41
42 for i in range(2):
43     K = tf([kd[i], kp[i], ki[i]], [1,0])
44     Gyr = feedback(P*K, 1)
45     y, t = step(Gyr, np.arange(0, 2, 0.01)) # step responce
46
47     pltargs = {'label':f'$k_D$={kd[i]}'}
48     ax.plot(t, y*ref, **pltargs)
49
50 ax.axhline(ref, color='k', linewidth=1)
51 plot_set(ax, 't', 'y', 'best')
52
53 #plt.savefig('Comparison_StepResponse.png', dpi=300)
```

Listing 5.7: Comparison

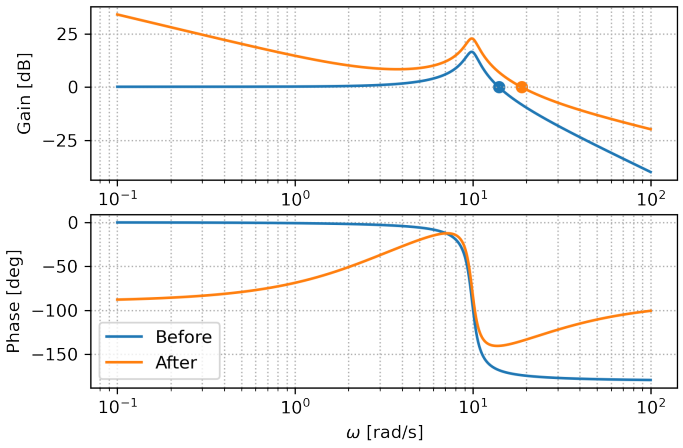

Figure 5.11: Comparison of Bode diagram of P control and PID control (open loop)

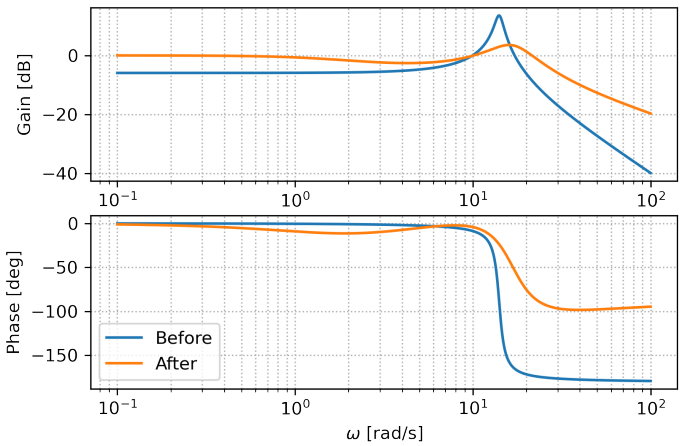

Figure 5.12: Comparison of Bode diagram of P control and PID control (closed loop)

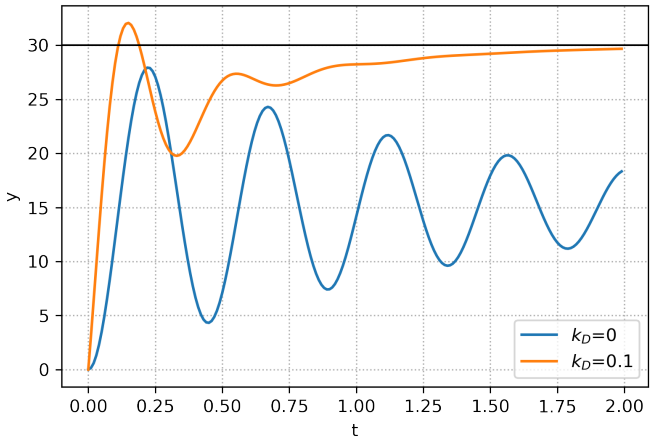

Figure 5.13: Comparison of step responses of P-control and PID-control (closed-loop)

First, from Fig. 5.11, it can be seen that by using PID control, the open-loop system can be tuned to meet the design specifications (stability: maintain $\omega_{\mathrm{gc}} < \omega_{\mathrm{pc}}$, quick response: make ω_{gc} as large as possible, and damping property: (increase the phase margin ϕ_{pm}, steady-state characteristics: increase the low-frequency gain).

Figure 5.12 also shows that the closed-loop system is designed to meet the design specifications (fast response: make the bandwidth ω_{bw} sufficiently large, attenuation: make the peak gain M_p small, and steady-state characteristics: make the DC gain 0 dB).

Furthermore, Fig. 5.13 shows the step response of the closed-loop system, which indicates that the system is able to follow the target value quickly while suppressing vibration by tuning with PID control.

5.3 Phase lead and lag compensation

In the PID control we have seen so far, the proportional, integral, and derivative elements were connected in parallel. Now consider gain compensation, phase lag compensation, and phase lead compensation connected in series as shown in Figure 5.14. In general, the series connection is easier to design prospectively because the gain and phase diagrams of the system are expressed by adding up those of each element.

Figure 5.14: Series compensation

5.3.1 Phase lag compensation

The phase delay compensation is represented as

$$\mathcal{K}_1(s) = \alpha \frac{T_1 s + 1}{\alpha T_1 s + 1} \ (\alpha > 1) \tag{5.21}$$

For example, a Bode plot with $\alpha = 10,\ T_1 = 0.1$ using the following code is shown in Fig. 5.15.

```
# Phase-Lag Compensation
import matplotlib.pyplot as plt
import numpy as np
from control.matlab import tf, bode, logspace, mag2db, margin, feedback, step

alpha = 10
T1 = 0.1
K1 = tf([alpha*T1, alpha], [alpha*T1, 1])
mag, phase, w = bode(K1, logspace(-2,3), plot=False)

fig, ax = plt.subplots(2, 1)
ax[0].semilogx(w, mag2db(mag))
ax[1].semilogx(w, np.rad2deg(phase))
bodeplot_set(ax)

#plt.savefig('phase-lag_compensation.png', dpi=300)
```

Listing 5.8: Phase lag compensation

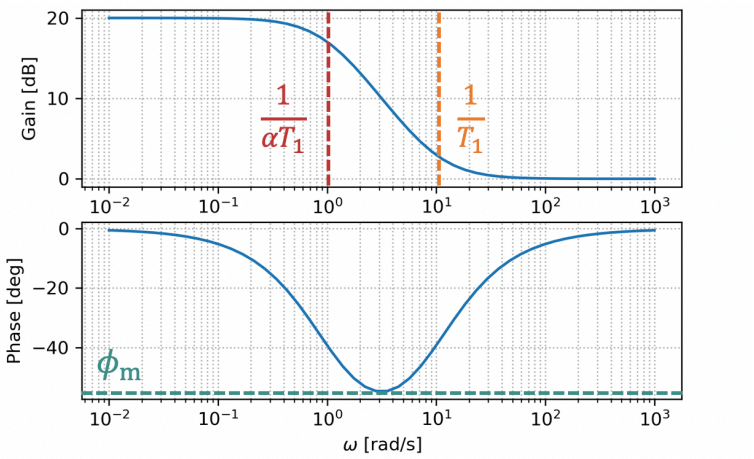

Figure 5.15: Phase lag compensation

Thus, it can be seen that increasing the low-frequency gain (by $20\log_{10}\alpha$) can improve the steady-state characteristics. However, in this case, the phase is delayed in the band of $\frac{1}{\alpha T_1} \sim \frac{1}{T_1}$. Also, the phase will be delayed by a maximum of ϕ_m at ω_m. However,

$$\omega_m = \frac{1}{T_1\sqrt{\alpha}},\ \ \phi_m = \sin^{-1}\frac{1-\alpha}{1+\alpha}. \tag{5.22}$$

In addition, if $\alpha \to \infty$ is used in the phase lag compensation formula 5.21, the performance is approximated as $K_1(s) = 1 + \frac{1}{T_1 s}$, which is close to PI control.

The design procedure using this phase lag compensation is as follows.

1. Determine α to meet the specification for steady-state deviation, taking into account that the low-frequency gain will increase by $20\log_{10}\alpha$ [dB].
2. Choose T_1 so that $\omega = \frac{1}{T_1}$ is less than one-tenth of the design value of the gain crossing frequency so that the phase lag does not degrade stability.

5.3.2 Phase lead compensation

The phase lead compensation is represented as

$$\mathcal{K}_2(s) = \frac{T_2 s + 1}{\beta T_2 s + 1}\ (\beta < 1). \tag{5.23}$$

For example, a Bode diagram with $\beta = 0.1,\ T_2 = 1$ using the following code is shown in Fig. 5.16.

```
# Phase-Lead Compensation
import matplotlib.pyplot as plt
import numpy as np
from control.matlab import tf, bode, logspace, mag2db, margin, feedback, step

beta = 0.1
```

```
T2 = 1
K2 = tf([T2, 1], [beta*T2, 1])
mag, phase, w = bode(K2, logspace(-2,3), plot=False, wrap_Phase=True)

fig, ax = plt.subplots(2, 1)
ax[0].semilogx(w, mag2db(mag))
ax[1].semilogx(w, np.rad2deg(phase))
bodeplot_set(ax)

#plt.savefig('phase-lead_compensation.png', dpi=300)
```

Listing 5.9: Phase lead compensation

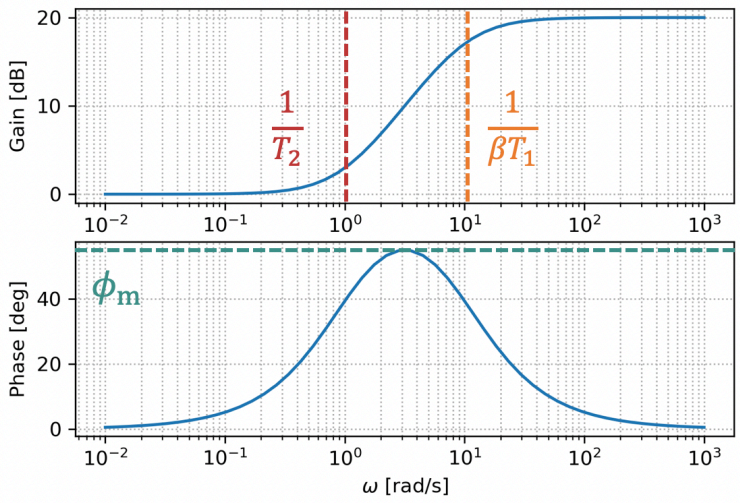

Figure 5.16: Phase lead compensation

This phase lead increases the phase margin and improves attenuation. In addition, it can be seen that the high-frequency gain is increased, which improves the quick response. Note that the phase advances by a maximum of ϕ_m at ω_m. However, it is

$$\omega_m = \frac{1}{T_2\sqrt{\beta}},\ \ \phi_m = \sin^{-1}\frac{1-\beta}{1+\beta}. \tag{5.24}$$

If $\beta \to \infty$ is used in the phase lead compensation formula 5.23, the performance is approximated as $K_2(s) = 1 + T_2 s$, which is close to PD control.

The design procedure using this phase lead compensation is as follows:

1. Evaluate the phase margin $\tilde{\phi}_{\rm pm}$ of the open-loop system before combining $\mathcal{K}_2$ and $\bar{\phi} = \phi_{\rm pm} - \tilde{\phi}_{\rm pm}$ for the target $\phi_{\rm pm}$ is calculated. Then, determine β so that $\phi_m = \bar{\phi}$.
2. Determine T_2 so that the frequency you want to set as the final gain crossing frequency is ω_m.

5.3.3 Control system design for vertical drive arm

As an example, let us design a controller $\mathcal{K}(s) = k\mathcal{K}_1(s)\mathcal{K}_1(s)$ consisting of gain compensation, phase lag compensation, and phase lead compensation for a vertical drive arm. As design specifications, the gain crossing frequency is set to 40 rad/s and the phase margin to 60 dB, aiming to keep the steady-state deviation as small as possible.

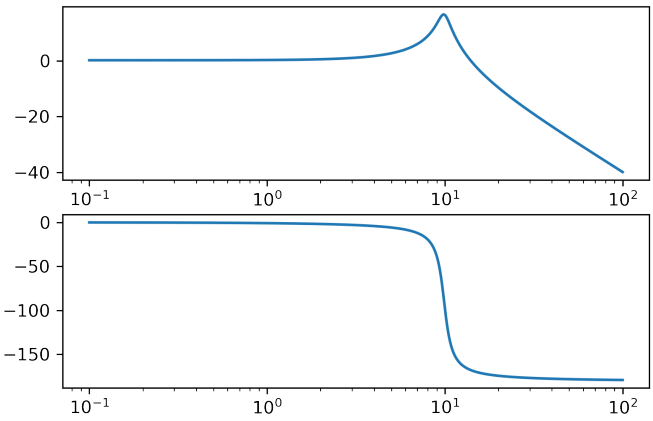

Figure 5.17: Bode plot for vertical drive arm

First, the Bode plot of the plant is drawn as shown in Fig. 5.17. Since the DC gain is 0 dB, a steady-state deviation remains even if the feedback control system is built as it is. Therefore, to reduce the steady-state deviation, the low-frequency gain is increased by phase lag compensation.

Here, $\alpha = 20$ is used, and the frequency at which the gain is increased should be such that the value of $\frac{1}{T_1}$ is about one-tenth of the final gain crossing frequency. In the current case, $T_1 = 0.25$, since 40 rad/s is the design value. Therefore, the value is

$$\mathcal{K}_1(s) = \frac{5s + 20}{5s + 1} \tag{5.25}$$

Here, if the code below is executed, the Bode plot of the open-loop system $\mathcal{H}_1(s) = \mathcal{P}(s)\mathcal{K}_1(s)$ will be as shown in Fig. 5.18.

```
# Phase-Lag Compensation for Vertical Drive Arm

from control.matlab import tf, bode, logspace, mag2db, margin, freqresp

alpha = 20
T1 = 0.25
K1 = tf([alpha*T1, alpha], [alpha*T1, 1])
print('K1=', K1)
```

```
9
10 H1 = P * K1
11 mag, phase, w = bode(H1, logspace(-1,2,1000), plot=False)
12
13 fig, ax = plt.subplots(2, 1)
14 ax[0].semilogx(w, mag2db(mag))
15 ax[1].semilogx(w, np.rad2deg(phase))
16 ax[1].set_ylim(-200, 2)
17 bodeplot_set(ax)
18
19 #plt.savefig('phase-lag_compensation_VDA.png', dpi=300)
20
21 [mag], [phase], _ = freqresp(H1, [40])
22 phaseH1at40 = np.rad2deg(phase)
23 print('----------------------')
24 print('phase at 40 rad/s =', phaseH1at40-360)
```

Listing 5.10: Phase lag compensation (vertical drive arm)

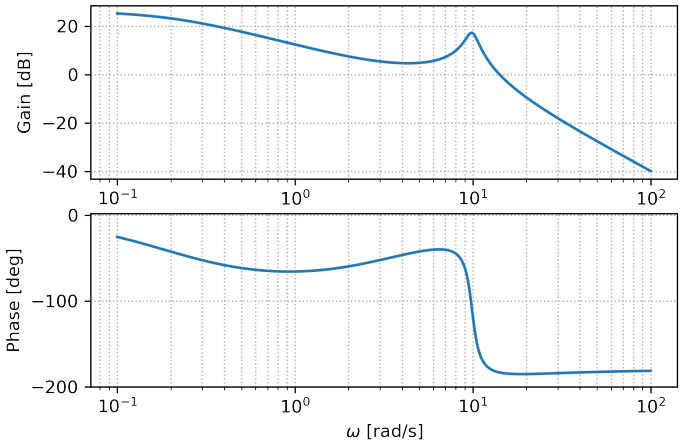

Figure 5.18: Phase lag compensation (vertical drive arm)

From this, it can be seen that the low-frequency gain is increasing.

If the phase margin is left as it is, the phase margin will fall below 60 deg (the phase margin at 40 rad/s is about -3 deg), so it is necessary to advance the phase by about 63 deg. Therefore, $\phi_m = 63\text{deg}$ and β is calculated as in

$$\beta = \frac{1 - \sin\phi_m}{1 + \sin\phi_m}. \tag{5.26}$$

Also, let T_2 be

$$T_2 = \frac{1}{\omega_m\sqrt{\beta}}. \tag{5.27}$$

From this, the phase lead compensation is

$$\mathcal{K}_2(s) = \frac{0.1047s + 1}{0.005971s + 1}. \tag{5.28}$$

At this point, executing the code below, the Bode plot of the open-loop system $\mathcal{H}_2(s) = \mathcal{P}(s)\mathcal{K}_1(s)\mathcal{K}_2(s)$ will be as shown in Fig. 5.19.

```
# Phase-Lead Compensation for Vertical Drive Arm

from control.matlab import tf, bode, logspace, mag2db, margin, freqresp

phim = (60 - (180 + phaseH1at40)) * np.pi / 180
beta = (1 - np.sin(phim)) / (1 + np.sin(phim))

T2 = 1/40/np.sqrt(beta)
K2 = tf([T2, 1], [beta*T2, 1])
print('K2=', K2)

fig, ax = plt.subplots(2, 1)
H2 = P * K1 * K2
mag, phase, w = bode(H2, logspace(-1,2,1000), plot=False)
ax[0].semilogx(w, mag2db(mag))
ax[1].semilogx(w, np.rad2deg(phase))
ax[1].set_ylim(-200, 2)
bodeplot_set(ax)

#plt.savefig('phase-lead_compensation_VDA.png', dpi=300)

[mag], [phase], _ = freqresp(H2, [40])
magH2at40 = mag
phaseH2at40 = np.rad2deg(phase)
print('----------------------')
print('gain at 40 rad/s =', mag2db(magH2at40))
print('phase at 40 rad/s =', phaseH2at40)
```

Listing 5.11: Phase lead compensation (vertical drive arm)

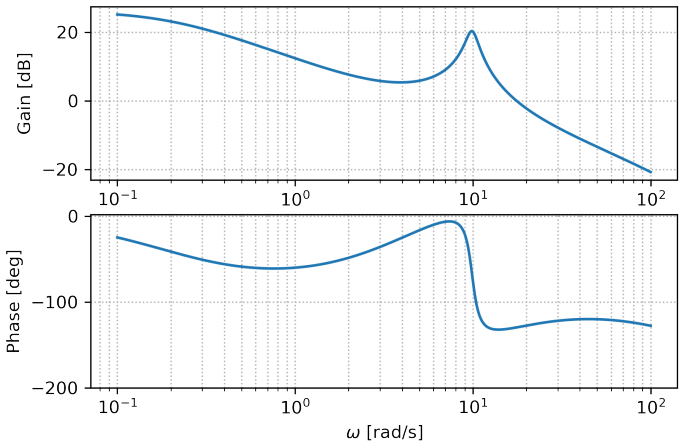

Figure 5.19: Phase lead compensation (vertical drive arm)

From this, the phase is advanced around 40 rad/s to -120deg.

Finally, the gain at 40 rad/s is set to 0 dB by gain compensation; since the gain at 40 rad/s is about -11.06 dB, it is moved up by this amount. Therefore, as shown in the code below, $k =$`1/magH2at40` is used as gain compensation to obtain Fig. 5.20.

```
# Gain Compensation for Vertical Drive Arm

k = 1 / magH2at40
print('k=', k)
```

```
5
6  H = P * k * K1 * K2
7
8  fig, ax = plt.subplots(2,1)
9  mag, phase, w = bode(P, logspace(-1,2,1000), plot=False) # before
10 ax[0].semilogx(w, mag2db(mag), label='P')
11 ax[1].semilogx(w, np.rad2deg(phase), label='P')
12 gm, pm, wpc, wgc = margin(P)
13 ax[0].scatter(wgc, 0)
14
15 mag, phase, w = bode(H, logspace(-1,2,1000), plot=False) # after
16 ax[0].semilogx(w, mag2db(mag), label='H')
17 ax[1].semilogx(w, np.rad2deg(phase), label='H')
18 gm, pm, wpc, wgc = margin(H)
19 ax[0].scatter(wgc, 0)
20
21 bodeplot_set(ax, 3)
22
23 print('-----------------------')
24 print('(GM, PM, wpc, wgc)')
25 print(margin(H))
26
27 #plt.savefig('HandP_VDA.png')
```

Listing 5.12: Gain compensation (vertical drive arm)

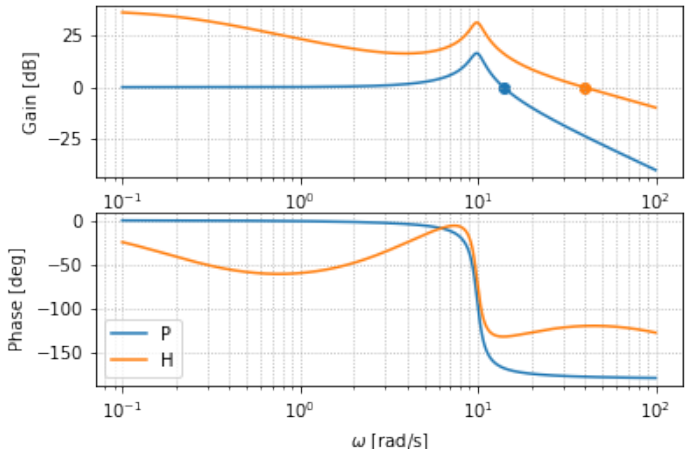

Figure 5.20: Gain compensation (vertical drive arm)

This shows that the control design meets the specifications. Therefore, finally, we also check the characteristics of the closed-loop system. The transfer function from the target value r to the output y is

$$\mathcal{G}_{yr} = \frac{\mathcal{H}}{1+\mathcal{H}}, \tag{5.29}$$

and executing the following code yields the step response shown in Fig. 5.21.

```
1  # Gain Compensation for Vertical Drive Arm (Step Responce)
2
3  from control.matlab import feedback, step
4
5  fig, ax = plt.subplots()
6
```

```
Gyr_P = feedback(P, 1)
y, t = step(Gyr_P, np.arange(0,2,0.01))
ax.plot(t, y*ref, label='before')

Gyr_H = feedback(H, 1)
y, t = step(Gyr_H, np.arange(0,2,0.01))
ax.plot(t, y*ref, label='after')

ax.axhline(ref, color='k', linewidth=1)
plot_set(ax, 't', 'y', 'best')

#plt.savefig('step_VDA.png')
```

Listing 5.13: Step response (vertical drive arm)

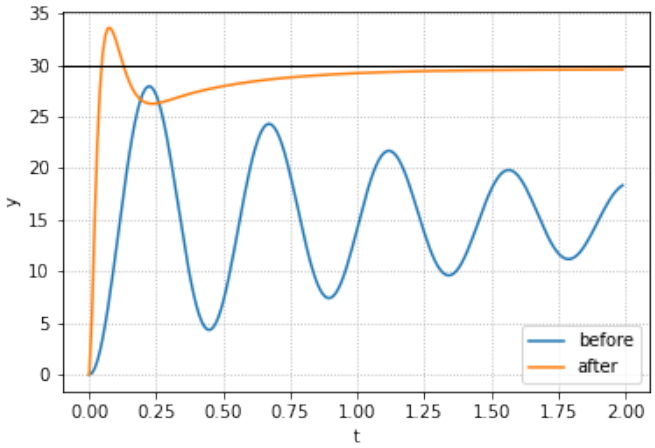

Figure 5.21: Gain compensation (step response for vertical drive arm)

This shows that the oscillations and steady-state deviations are reduced, as they quickly reach near the target value. In addition, the following code will produce a Bode diagram as shown in Fig. 5.22

```
# Closed-Loop for VDA

fig, ax = plt.subplots(2, 1)

mag, phase, w = bode(Gyr_P, logspace(-1,2,1000), plot=False)
ax[0].semilogx(w, mag2db(mag), label='before')
ax[1].semilogx(w, np.rad2deg(phase), label='before')

mag, phase, w = bode(Gyr_H, logspace(-1,2,1000), plot=False)
ax[0].semilogx(w, mag2db(mag), label='before')
ax[1].semilogx(w, np.rad2deg(phase), label='after')

bodeplot_set(ax, 3)
#plt.savefig('CL_VDA.png')
```

Listing 5.14: Boded plot (vertical drive arm)

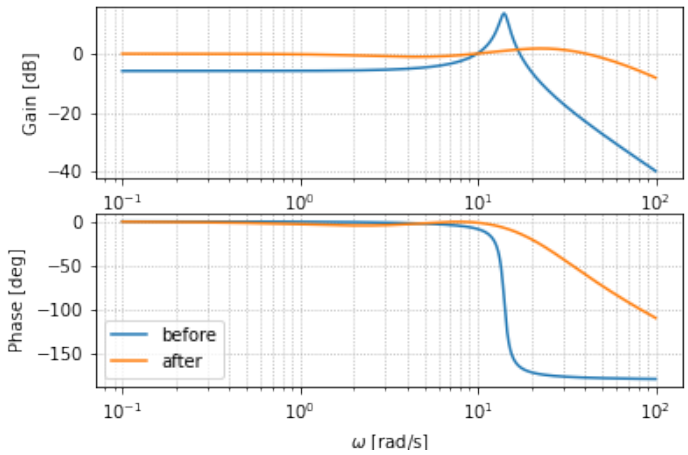

Figure 5.22: Gain compensation (Bode plot for vertical drive arm)

The DC gain was less than 0 dB and the peak gain was small before the design, but the phase lag/lead compensation increased the DC gain to around 0 dB, suppressed the peak gain, and increased the bandwidth.

CHAPTER 6

ADVANCED CONTROL

For example, when a user wants to build a state feedback control system, sometimes it may be difficult to determine the state by sensors for a variety of reasons, such as physical constraints that prevent the placement of sensors in the mechanism. In other cases, parameters in the model may be uncertain, and in reality, not everything will work. As an example of how to deal with these problems, this chapter describes the basics of observers, robust control, and model predictive control. Finally, we discuss discretization for implementing the designed controller in a digital device.

6.1 Observer

In the design of control systems using transfer function models, such as PID control and phase lead/lag compensation described in the previous chapters, the "output" of the control target is fed back. On the other hand, state feedback control using a state-space model feeds back the "state" of the control target.

However, there are cases where a sensor cannot be placed in the first place due to mechanical problems with the control target, or where not all states can be acquired even if a sensor is used. In such cases, state feedback cannot be provided, and the user is left with the choice of giving up. In such cases, we can estimate the internal state $\boldsymbol{x}$ from the known input u and output y using an observer like the one shown in Figure 6.1.

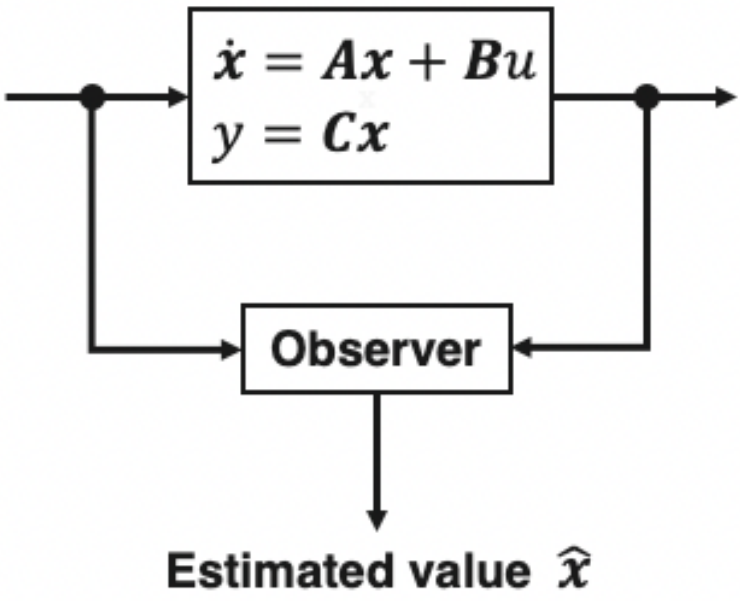

Figure 6.1: Observer

6.1.1 Full-order state observer

There are various types of observers, but the full-order state observer represented by

$$\dot{\hat{\boldsymbol{x}}}(t) = \boldsymbol{A}\hat{\boldsymbol{x}}(t) + \boldsymbol{B}u(t) - \boldsymbol{L}(y(t) - \boldsymbol{C}\hat{\boldsymbol{x}}(t)) \tag{6.1}$$

is often used. where $\hat{\boldsymbol{x}}$ is the estimated value, the first and second terms on the right side are the model of the control target, and the third term is the feedback of the difference between the actually observed output y of the control target and the estimated output $\boldsymbol{C}\hat{\boldsymbol{x}}$ by the observer. Note that $\boldsymbol{L}$ is the design parameter, which is called the observer gain.

Now, substituting $y = \boldsymbol{C}\boldsymbol{x}$ into equation 6.1 and rearranging, we obtain

$$\dot{\hat{\boldsymbol{x}}}(t) = (\boldsymbol{A} + \boldsymbol{LC})\hat{x}(t) + \boldsymbol{B}u(t) - \boldsymbol{LC}\boldsymbol{x}(t). \tag{6.2}$$

Let $\boldsymbol{e} = \boldsymbol{x}(t) - \hat{x}(t)$ be the estimation error, since

$$\dot{e}(t) = (\boldsymbol{A} + \boldsymbol{LC})\boldsymbol{e}(t), \tag{6.3}$$

so if $\boldsymbol{A} + \boldsymbol{LC}$ is stable, then $\boldsymbol{e}(t) \to 0$ $(t \to \infty)$, i.e., $\hat{\boldsymbol{x}}(t) \to \boldsymbol{x}(t)$ $(t \to \infty)$, and the estimated value $\hat{\boldsymbol{x}}$ follows the state $\boldsymbol{x}$ to be controlled.

Basically, the design of the observer gain $\boldsymbol{L}$ uses pole placement, similar to the design of the state feedback gain $\boldsymbol{F}$ described in Chapter 4. If the system is observable, the poles can be placed at arbitrary locations, so $\boldsymbol{L}$ is determined so that the eigenvalues of $\boldsymbol{A} + \boldsymbol{LC}$ are at the specified poles.

In the following, the plant is as follow list.

```python
# Plant
from control.matlab import ss, acker, lsim

A = '0 1; -4 -5'
B = '0; 1'
C = '1 0'
D = '0'

P = ss(A, B, C, D)
```

Listing 6.1: Plant for chap7

Observer gains are designed with $-10 \pm 5j$ as the observer pole, utilizing the dual relationship between controllability and observability. To put it simply, it is equivalent to say that $(\boldsymbol{C}, \boldsymbol{A})$ is observable and $(\boldsymbol{A}^\top, \boldsymbol{C}^\top)$ is controllable. Based on this, we can consider $(\boldsymbol{A} + \boldsymbol{LC})^\top = \boldsymbol{A}^\top + \boldsymbol{C}^\top \boldsymbol{L}^\top$ and calculate $\boldsymbol{L}^\top$ from $\boldsymbol{A}^\top$ と $\boldsymbol{C}^\top$.

If the observer is implemented in the form

$$\dot{\hat{\boldsymbol{x}}}(t) = (\boldsymbol{A} + \boldsymbol{LC})\hat{x}(t) + \begin{bmatrix} \boldsymbol{B} & -\boldsymbol{L} \end{bmatrix} \begin{bmatrix} u(t) \\ y(t) \end{bmatrix} \tag{6.4}$$

state estimation can be simulated with the following code, yielding Figure 6.2.

```python
# State Estimation with Observer
import matplotlib.pyplot as plt
import numpy as np

observer_poles = [-10+5j, -10-5j]

L = -acker(P.A.T, P.C.T, observer_poles).T # observer gain

fig, ax = plt.subplots(1,2, figsize=(12,4))

## true behavior
G = ss(P.A, P.B, np.eye(2), [[0],[0]])
Td = np.arange(0, 3, 0.01)

u = np.zeros_like(Td) # input
X0 = [-1, 0.5]        # initial state
x, t, _ = lsim(G, u, Td, X0)
ax[0].plot(t, x[:, 0], label='${x}_1$')
ax[1].plot(t, x[:, 1], label='${x}_2$')

y = x[:, 0] # output


## observer
Obs = ss(P.A+L@P.C, np.c_[P.B, -L], np.eye(2), [[0,0],[0,0]])
xhat, t, _ = lsim(Obs, np.c_[u, y], Td, [0, 0])
ax[0].plot(t, xhat[:, 0], label='$\hat{x}_1$')
ax[1].plot(t, xhat[:, 1], label='$\hat{x}_2$')

for i in [0, 1]:
    plot_set(ax[i], 't', '', 'best')
    ax[i].set_xlim([0, 3])

ax[0].set_ylabel('$x_1, \hat{x}_1$')
ax[1].set_ylabel('$x_2, \hat{x}_2$')

#plt.savefig('StateEstimation_with_Observer.png', dpi=300)
```

Listing 6.2: State estimation with observer

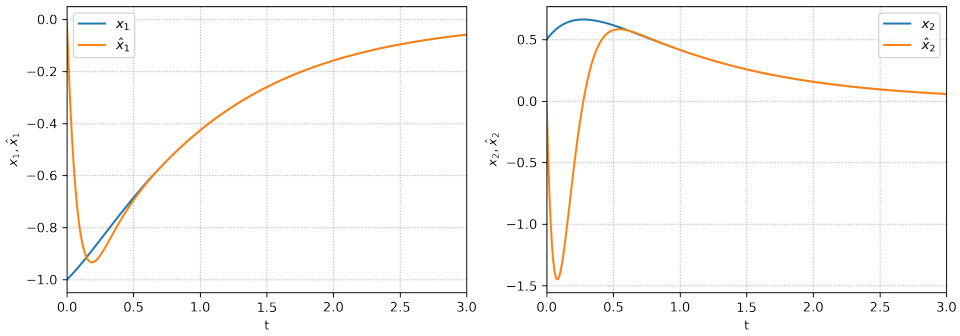

Figure 6.2: State estimation with observer

From Figure 6.2, we can see that $\hat{\boldsymbol{x}}$ follows $\boldsymbol{x}$ as time passes. This indicates that the observer can estimate the state.

In addition, based on the principle of separation, the observer and state feedback can be designed independently. Therefore, as shown in Fig. 6.3, the state feedback $u(t) = \boldsymbol{F}\hat{\boldsymbol{x}}(t)$ can be applied using $\hat{x}(t)$ estimated by the observer.

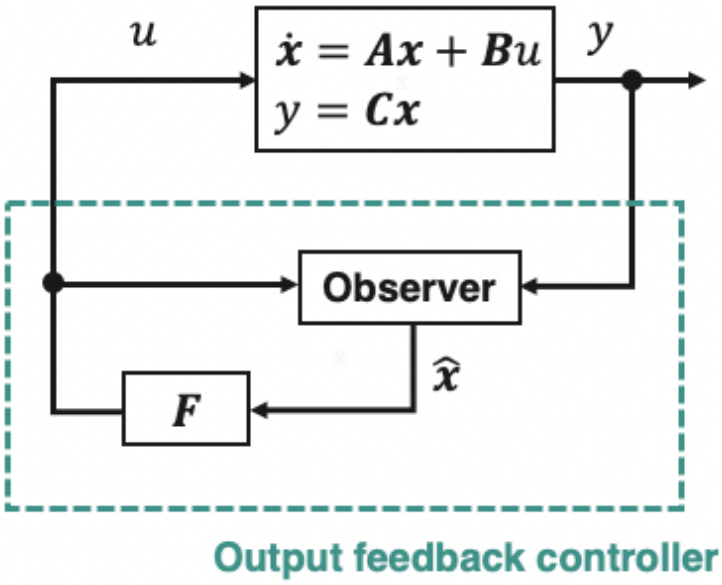

Figure 6.3: Block diagram of output feedback control

The controller is then

$$\mathcal{K} = \begin{cases} \dot{\hat{\boldsymbol{x}}}(t) = (\boldsymbol{A} + \boldsymbol{BF} + \boldsymbol{LC})\hat{\boldsymbol{x}}(t) - \boldsymbol{L}y(t) \\ u(t) = \boldsymbol{F}\hat{\boldsymbol{x}} \end{cases} \tag{6.5}$$

and the control input $u(t)$ can be calculated from the output $y(t)$ to be controlled. Also,

Laplace transforming it yields

$$\mathcal{K}(s) = -\boldsymbol{F}(s\boldsymbol{I} - (\boldsymbol{A} + \boldsymbol{B}\boldsymbol{F} + \boldsymbol{L}\boldsymbol{C}))^{-1}\boldsymbol{L}. \tag{6.6}$$

Furthermore, executing the code below for this output feedback control yields Figure 6.4.

```
# Output Feedback Control
from control.matlab import tf, feedback, initial

## design of state feedback gain
regulator_poles = [-5+2j, -5-2j]
F = -acker(P.A, P.B, regulator_poles)

## design of observer gain
observer_poles= [-10+5j, -10-5j]
L = -acker(P.A.T, P.C.T, observer_poles).T

## output feedback (observer + state feedback)
K = ss(P.A+P.B@F+L@P.C, -L, F, 0)
print('K:\n', K)
print('------------')
print('K(s)=', tf(K))

Gfb = feedback(P, K, sign=1)

fig, ax = plt.subplots()
Td = np.arange(0, 3, 0.01)

y, t = initial(P, Td, [-1, 0.5]) # without output feedback
ax.plot(t, y, label='w/o output feedback')

y, t = initial(Gfb, Td, [-1, 0.5, 0, 0]) # with output feedback
ax.plot(t, y, label='with output feedback')

plot_set(ax, 't', 'y', 'best')

#plt.savefig('OutputFeedbackController.png', dpi=300)
```

Listing 6.3: Output feedback control

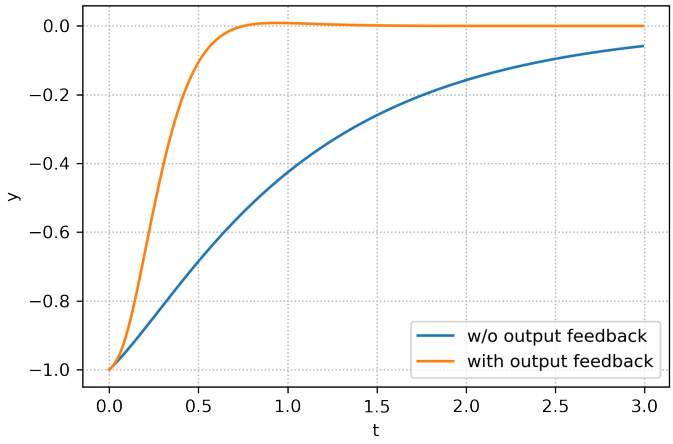

Figure 6.4: Output feedback control

From this, it can be seen that the response is improved by using both observer and state feedback.

6.1.2 Disturbance observer

If a constant-valued disturbance is added to the output y, the correct state cannot be estimated by the observer, as shown in Figure 6.5.

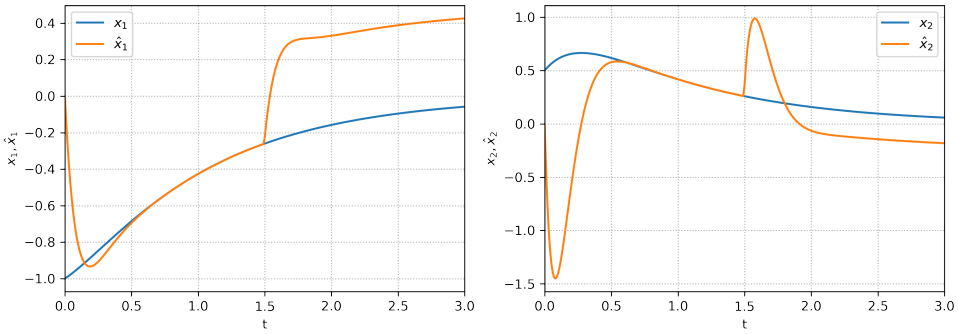

Figure 6.5: State estimation in the presence of disturbances

Therefore, we use a disturbance observer that estimates not only the state but also the disturbance. For example, if a disturbance d is added to the output y, then

$$\begin{cases} \dot{\boldsymbol{x}}(t) = \boldsymbol{A}\boldsymbol{x}(t) + \boldsymbol{B}u(t) \\ y(t) = \boldsymbol{C}\boldsymbol{x}(t) + d(t) \end{cases} \tag{6.7}$$

Since $\dot{d}(t) = 0$ in the case of a constant-valued disturbance, the state is expanded to

$$\begin{cases} \begin{bmatrix} \dot{\boldsymbol{x}}(t) \\ \dot{d}(t) \end{bmatrix} = \begin{bmatrix} \boldsymbol{A} & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} \boldsymbol{x}(t) \\ d(t) \end{bmatrix} + \begin{bmatrix} \boldsymbol{B} \\ 0 \end{bmatrix} u(t) \\ y(t) = \begin{bmatrix} \boldsymbol{C} & 1 \end{bmatrix} \begin{bmatrix} \boldsymbol{x}(t) \\ d(t) \end{bmatrix} \end{cases} \tag{6.8}$$

Here, constructing the observer as

$$\boldsymbol{x}_e = \begin{bmatrix} \boldsymbol{x}(t) \\ d(t) \end{bmatrix}, \ \boldsymbol{A}_e = \begin{bmatrix} \boldsymbol{A} & 0 \\ 0 & 0 \end{bmatrix}, \ \boldsymbol{B}_e = \begin{bmatrix} \boldsymbol{B} \\ 0 \end{bmatrix}, \ \boldsymbol{C}_e = \begin{bmatrix} \boldsymbol{C} & 1 \end{bmatrix} \tag{6.9}$$

yields

$$\dot{\hat{\boldsymbol{x}}}_e = \boldsymbol{A}_e\hat{\boldsymbol{x}}_e(t) + \boldsymbol{B}_e u(t) + \boldsymbol{L}_e(y(t) - \boldsymbol{C}_e\hat{\boldsymbol{x}}_e(t)) \tag{6.10}$$

(the design of the observer gain $\boldsymbol{L}_e$ is the same as in the normal case). Executing the code below for this, we obtain Figure 6.6 (Figure 6.5 is also obtained).

```
# Disturbance Observer

## without DOB
fig, ax = plt.subplots(1, 2, figsize=(12,4))

Td = np.arange(0, 3, 0.01)
X0 = [-1, 0.5]
d = 0.5*(Td>=1.5) # step disturbance

x, t = initial(G, Td, X0)
ax[0].plot(t, x[:, 0], label='${x}_1$')
ax[1].plot(t, x[:, 1], label='${x}_2$')

u = np.zeros_like(Td) # input
y = x[:, 0]+d         # output
xhat, t, _ = lsim(Obs, np.c_[u,y], Td, [0, 0])
ax[0].plot(t, xhat[:, 0], label='$\hat{x}_1$')
ax[1].plot(t, xhat[:, 1], label='$\hat{x}_2$')

for i in [0, 1]:
    plot_set(ax[i], 't', '', 'best')
    ax[i].set_xlim([0, 3])

ax[0].set_ylabel('$x_1, \hat{x}_1$')
ax[1].set_ylabel('$x_2, \hat{x}_2$')

#plt.savefig('withoutDOB.png', dpi=300)


## with DOB
fig, ax = plt.subplots(1, 2, figsize=(12,4))

Dobserver_poles = [-10+5j, -10-5j, -3]

Abar = np.block([[P.A, np.zeros([2,1])], [np.zeros((1,3))]])
Bbar = np.block([[P.B], [0]])
Cbar = np.block([P.C, 1])

Lbar = -acker(Abar.T, Cbar.T, Dobserver_poles).T

Aob = Abar + Lbar@Cbar
Bob = np.block([Bbar, -Lbar])

DObs = ss(Aob, Bob, np.eye(3), np.zeros([3,2]))

x, t = initial(G, Td, X0)
ax[0].plot(t, x[:, 0], label='${x}_1$')
ax[1].plot(t, x[:, 1], label='${x}_2$')

xhat, t, _ = lsim(DObs, np.c_[u,y], Td, [0, 0, 0])
ax[0].plot(t, xhat[:, 0], label='$\hat{x}_1$')
ax[1].plot(t, xhat[:, 1], label='$\hat{x}_2$')

for i in [0, 1]:
    plot_set(ax[i], 't', '', 'best')
    ax[i].set_xlim([0, 3])

ax[0].set_ylabel('$x_1, \hat{x}_1$')
ax[1].set_ylabel('$x_2, \hat{x}_2$')

#plt.savefig('withDOB.png', dpi=300)
```

Listing 6.4: Disturbance observer

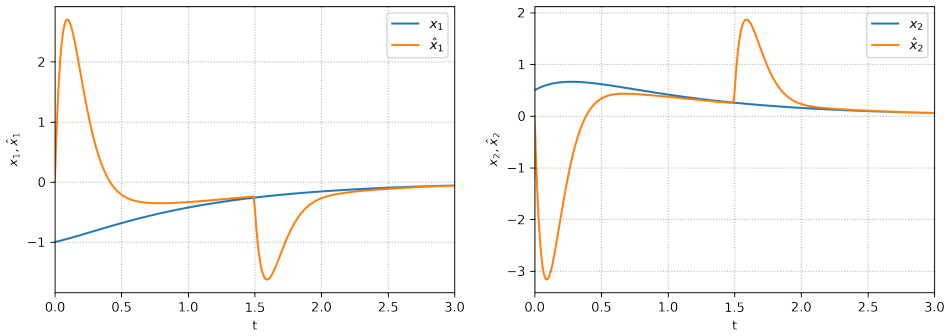

Figure 6.6: Disturbance observer

6.1.3 Stationary Kalman filter

Since the observer estimates the internal state from the output of the plant, it may not be able to estimate correctly if the output is ridden with noise. In such a case, assume that the noise is white and design an observer gain $\boldsymbol{L}$ that minimizes the mean square value of the estimation error.

Here, we consider

$$\begin{cases} \dot{\boldsymbol{x}}(t) = \boldsymbol{A}\boldsymbol{x}(t) + \boldsymbol{B}u(t) + \boldsymbol{v}(t) \\ y(t) = \boldsymbol{C}\boldsymbol{x}(t) + w(t) \end{cases} \tag{6.11}$$

as the plant. However, $\boldsymbol{v}$ is the system noise and w is the observed noise (both are white noise). Also, their mean and variance are

$$\begin{cases} \mathbb{E}[\boldsymbol{v}(t)] = 0, \quad \mathbb{E}[\boldsymbol{v}(t)\boldsymbol{v}(\tau)^\top] = \boldsymbol{Q}\delta(t-\tau) \\ \mathbb{E}[w(t)] = 0, \quad \mathbb{E}[w(t)w(\tau)^\top] = \boldsymbol{R}\delta(t-\tau) \end{cases} \tag{6.12}$$

and $\boldsymbol{v}$ and w are independent of each other.

The following results are known, and the observer designed in this way is called a stationary Kalman filter.

Stationary Kalman filter

For $\boldsymbol{Q} > 0$, $\boldsymbol{R} > 0$, the observer (stationary Kalman filter) that minimizes the mean square value

$$J = \mathbb{E}[(\boldsymbol{x}(t) - \hat{\boldsymbol{x}}(t))^\top (\boldsymbol{x}(t) - \hat{\boldsymbol{x}}(t))] \tag{6.13}$$

of the estimation error is

$$\dot{\hat{\boldsymbol{x}}}(t) = \boldsymbol{A}\hat{\boldsymbol{x}}(t) + \boldsymbol{B}u(t) - \boldsymbol{L}_{\text{opt}}(y(t) - \boldsymbol{C}\hat{\boldsymbol{x}}(t)) \tag{6.14}$$

and the value of $\boldsymbol{L}_{\text{opt}}$ is

$$\boldsymbol{L}_{\text{opt}} = -\boldsymbol{P}\boldsymbol{C}^\top \boldsymbol{R}^{-1} \tag{6.15}$$

However, $\boldsymbol{P} = \boldsymbol{P}^\top$ is the only positive definite symmetric solution that satisfies the Riccati equation

$$\boldsymbol{A}\boldsymbol{P} + \boldsymbol{P}\boldsymbol{A}^\top + \boldsymbol{Q} - \boldsymbol{P}\boldsymbol{C}^\top \boldsymbol{R}^{-1}\boldsymbol{C}\boldsymbol{P} = 0. \tag{6.16}$$

The stationary Kalman filter is designed in Python as `L, P, E = lqe(A, Bv, C, QN, RN)`, etc. Here, the argument `A, Bv, C` is the system $\dot{\boldsymbol{x}} = \boldsymbol{A}\boldsymbol{x} + \boldsymbol{B}u\boldsymbol{v} + \boldsymbol{B}_v$, $y = \boldsymbol{C}\boldsymbol{v} + w$ matrices $\boldsymbol{A}$, $\boldsymbol{B}_v$ and $\boldsymbol{C}$, where `QN` and `RN` are the covariance of the noise. The return values `L, P, E` are the observer gains, the solution of the Riccati equation, and the closed-loop poles (= eigenvalues of $\boldsymbol{A} - \boldsymbol{L}\boldsymbol{C}$), respectively.

In the following, we design a stationary Kalman filter for

$$\begin{cases} \begin{bmatrix} \dot{\boldsymbol{x}}(t) \\ \dot{d}(t) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -4 & -5 \end{bmatrix} \boldsymbol{x}(t) + \begin{bmatrix} 0 \\ 1 \end{bmatrix} (u(t) + v(t)) \\ y(t) = \begin{bmatrix} 1 & 0 \end{bmatrix} \boldsymbol{x}(t) + w(t) \end{cases} \tag{6.17}$$

If we run the code below with the covariance of the system noise and the observed noise both set to 1, we obtain Figure 6.7.

```
# Stationary Kalman Filter
from control.matlab import ss, lqe, lsim

QN, RN = 1, 1 # covariance
L, _, _ = lqe(P.A, P.B, P.C, QN, RN)
L = -L
KObs = ss(P.A+L@P.C, np.block([P.B, -L]), np.eye(2), np.zeros([2,2]))

Td = np.arange(0, 6, 0.01)
u = 0.5 * np.sin(6*Td) + 0.5 * np.cos(8*Td)
w = np.random.normal(loc=0, scale=np.sqrt(QN), size=len(Td)) # system noise
v = np.random.normal(loc=0, scale=np.sqrt(RN), size=len(Td)) # measurement noise

X0 = [-0.3, 0.2]
sys = ss(P.A, P.B, np.eye(2), np.zeros([2,1]))
_, t, xorg = lsim(sys, u, Td, X0) # state without noise
_, t, x = lsim(sys, u+w, Td, X0)  # state with noise
y = x[:, 0]+ v                    # output with measurement noise

fig, ax = plt.subplots(2, 2, figsize=(12,5))
ax[0,1].plot(t, y, label='$y$')
ax[1,0].plot(t, xorg[:,0], label='$x_1$')
ax[1,1].plot(t, xorg[:,1], label='$x_2$')
ax[0,0].plot(t, u+w, label='$u+w$')
```

```
25
26 xhat, t, _ = lsim(KObs, np.c_[u, y], Td, [0, 0]) # state estimation with Kalman filter
27 ax[1,0].plot(t, xhat[:,0], label='$\hat{x}_1$')
28 ax[1,1].plot(t, xhat[:,1], label='$\hat{x}_2$')
29 ax[0,0].plot(t, u, label='$u$')
30
31 for i in [0,1]:
32     for j in [0,1]:
33         plot_set(ax[i,j], '$t$', '', 'best')
34         ax[i,j].set_xlim([0,6])
35
36 ax[0,1].set_ylabel('$y$')
37 ax[0,0].set_ylabel('$u$')
38 ax[1,0].set_ylabel('$x_1$')
39 ax[1,1].set_ylabel('$x_2$')
40
41 #plt.savefig('StationaryKalmanFilter.png', dpi=300)
```

Listing 6.5: Stationary Kalman filter

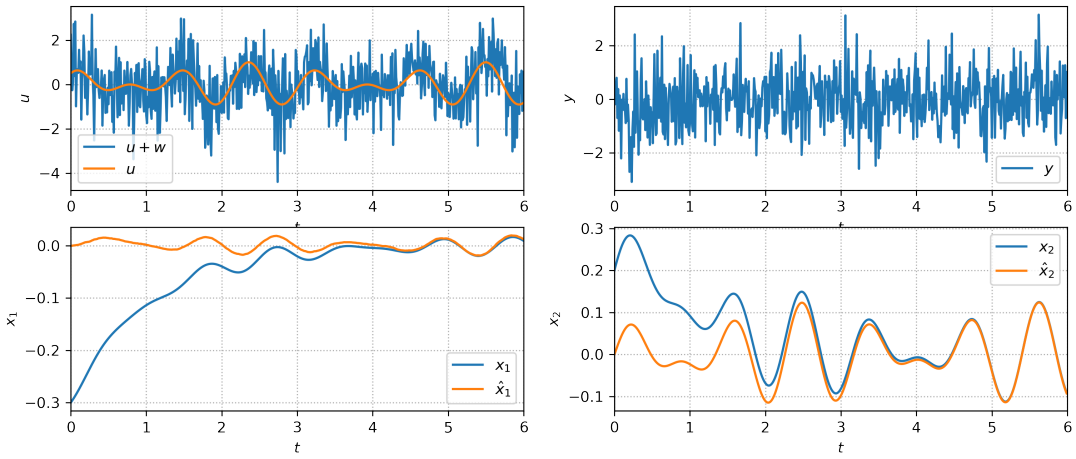

Figure 6.7: Stationary Kalman filter

The upper part of Figure 6.7 shows the control input and output with noise input to the Kalman filter, and the lower part shows the state estimated from them. This figure shows that the state can be estimated by using the Kalman filter even in the presence of noise.

6.2 Robust control

6.2.1 About robust Control

Up to this point, a control system has been constructed using a mathematical model, but this mathematical model is not a perfect representation of the real control target, i.e., the model includes uncertainties (such as errors in the model parameters, effects of disturbances, or nonlinearity that is not taken into account). Therefore, it is important

to design the control considering the model uncertainty. In this section, we discuss the fundamentals of robust control as an example.

In the following, the transfer function model of the control object will be referred to as the nominal model to distinguish it from the real control object. The real control target is denoted as

$$\mathcal{P}_r(s) = (1 + \Delta(s)\mathcal{W}_\mathcal{T}(s))\mathcal{P}(s) \tag{6.18}$$

(example of multiplicative error). Here, $\mathcal{W}_\mathcal{T}(s)$ is a certain stable transfer function and $\Delta(s)$ is a stable transfer function whose magnitude is less than or equal to 1, representing uncertainty. In this case, consider the set

$$\mathbb{P} = \{\mathcal{P}_r(s)(s)|\mathcal{P}_r(s) = (1 + \Delta(s)\mathcal{W}_\mathcal{T}(s))\mathcal{P}(s),\ |\Delta(j\omega)| \leq 1, \forall\omega\} \tag{6.19}$$

of $\tilde{\mathcal{P}}(s)$. In this case, $\Delta(s)\mathcal{W}_\mathcal{T}(s)$ represents the uncertainty. The value of $|\Delta(j\omega)|$ is between 0 and 1 for each frequency, and $|\mathcal{W}_\mathcal{T}(j\omega)|$ determines the magnitude of uncertainty for each frequency. Therefore, $\mathcal{W}_\mathcal{T}$ is called the frequency weight function and $\Delta(s)\mathcal{W}_\mathcal{T}(s)$ is called multiplicative uncertainty.

As an example of a control object involving multiplicative uncertainty, consider an example with a vertical drive arm. In this case, executing the following code yields Figure 6.8.

```
# Multiplicative Uncertainty
from control.matlab import tf, bode, logspace, mag2db, ss2tf, step, feedback

g = 9.81
l = 0.2
M = 0.5
mu = 1.5e-2
J = 1.0e-2

Pn = tf([0, 1],[J, mu, M*g*l])

## uncertainty
delta = np.arange(-1, 1, 0.1)
WT = tf([10, 0], [1, 150])

fig, ax = plt.subplots(1, 2, figsize=(15,5))

for i in range(len(delta)):
    P = (1 + WT*delta[i])* Pn # plant with uncertainty
    mag, _, w = bode(P, logspace(-3,3), plot=False)
    ax[0].semilogx(w, mag2db(mag))

    DT = (P - Pn) / Pn # multiplicative uncertainty
    mag, _, w = bode(DT, logspace(-3,3), plot=False)
    ax[1].semilogx(w, mag2db(mag))

mag, _, w = bode(Pn, logspace(-3,3), plot=False)
ax[0].semilogx(w, mag2db(mag), color='k', lw=1.5)
mag, _, w = bode(WT, logspace(-3,3), plot=False)
ax[1].semilogx(w, mag2db(mag), color='k', lw=1.5)

bodeplot_set(ax)
ax[0].set_xlabel('$\omega$ [rad/s]')
ax[0].set_ylabel('Gain of $P$ [dB]')
ax[0].set_xlabel('$\omega$ [rad/s]')
ax[0].set_ylim([-100, 30])
ax[1].set_ylim([-100, 30])
```

```
38
39 #plt.savefig('MultiplicativeUncertainty.png', dpi=300)
```

Listing 6.6: Multiplicative uncertainty

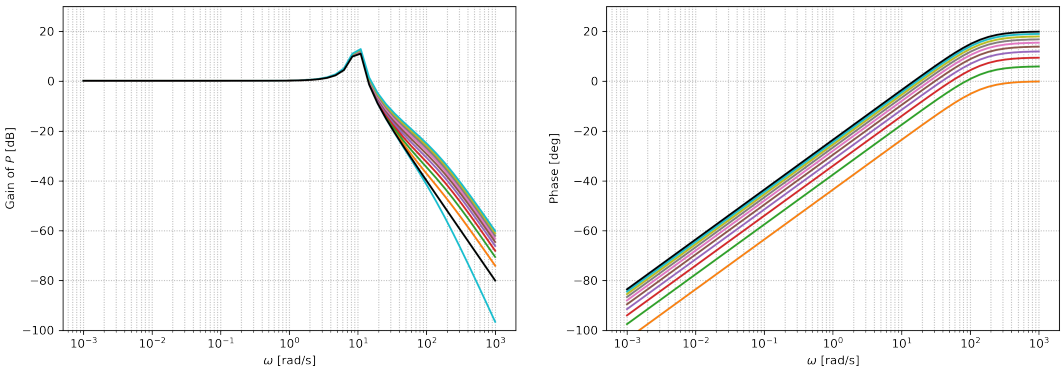

Figure 6.8: Multiplicative uncertainty

On the right is a gain diagram of the uncertainty $\Delta(s)\mathcal{W}_T(s)$, which shows that the high-frequency gain varies. This means that the response to the high-frequency component of the input signal varies, suggesting that when the controller gain is increased to improve quick response, for example, the desired response may not be obtained, i.e., it may become unstable. In the following, we first summarize the basic issues and then design a controller $\mathcal{K}$ that guarantees the internal stability of the feedback system for all $\tilde{\mathcal{P}}(s)$ in the set $\mathbb{P}$ (robust stabilization problem). We will also consider the mixed sensitivity problem and actually design a robust controller in Python.

6.2.2 Summary of basics

6.2.2.1 $\mathcal{H}_\infty$ norm

The control to keep $\mathcal{H}_\infty$ norm below a certain value is called $\mathcal{H}_\infty$ control, and here we discuss the basics of this control. Incidentally, $\mathcal{H}$ is an acronym for Hardy space. A Hardy space is a space consisting of regular functions satisfying certain properties on an open unit disk or upper half-plane. The Hardy space of order ∞ for an open unit disc $\mathcal{H}_\infty$ is defined by a vector space consisting of bounded regular functions on the disc. In a case like the one we are considering now and in plain language without fear of misunderstanding, this means that the closed loop is stable.

Now, the $\mathcal{H}_\infty$ norm is defined as

$$\|\mathcal{G}(s)\|_\infty = \sup_\omega |G(j\omega)| \tag{6.20}$$

for the closed-loop transfer function $\mathcal{G}(s)$ in the case of one-input one-output (SISO) control. Note that $\sup_\omega$ can be read as the maximum value $\max_\omega$. But there is the case where the

gain takes the maximum value only in $\omega \to \infty$ (although it cannot be realized), so strictly speaking $\sup_\omega$.

Anyway, in the case of SISO, the $\mathcal{H}_\infty$ **norm is the maximum value of the gain of the closed-loop transfer function**.

$$y = \mathcal{G}(s)u \tag{6.21}$$

for input u and output y, and the $\mathcal{H}_\infty$ norm satisfies

$$\|\mathcal{G}(s)\|_\infty = \sup_u \frac{\int_{-\infty}^{\infty} \|y(t)\|^2 \mathrm{d}t}{\int_{-\infty}^{\infty} \|u(t)\|^2 \mathrm{d}t}. \tag{6.22}$$

Furthermore, it is known that it satisfies

$$\|\mathcal{G}(s)\|_\infty = \sup_u \frac{\text{output power}}{\text{input power}}. \tag{6.23}$$

In other words, $\|\mathcal{G}(s)\|_\infty$ indicates **worst input to worst (maximum) output magnitude**, and it is sufficient to design the controller so that this is less than a certain value.

On the other hand, in the case of multiple-input multiple-output (MIMO) control, the transfer function is not a scalar but a matrix $\boldsymbol{\mathcal{G}}(s)$, which is an extension of the SISO case,

$$\|\boldsymbol{\mathcal{G}}(s)\|_\infty = \sup_\omega \bar{\sigma}\{\boldsymbol{\mathcal{G}}(j\omega)\}, \tag{6.24}$$

where $\bar{\sigma}\{\boldsymbol{\mathcal{G}}(j\omega)\}$ is the maximum singular value of $\boldsymbol{\mathcal{G}}(j\omega)$. The maximum singular value of matrix $\boldsymbol{\mathcal{G}}$ is the non-negative square rootmaximum of eigenvalue of $\boldsymbol{\mathcal{G}}^*\boldsymbol{\mathcal{G}}$, i.e. $\sqrt{\lambda_{\max}(\boldsymbol{\mathcal{G}}^*\boldsymbol{\mathcal{G}})}$ when the adjoint matrix of $\boldsymbol{\mathcal{G}}$ is $\boldsymbol{\mathcal{G}}^*$. Note that $\boldsymbol{\mathcal{G}}^*\boldsymbol{\mathcal{G}}$ is a Gram matrix, and it is well known that all eigenvalues are non-negative real numbers since a Gram matrix is a semipositive definite Hermitian matrix.

6.2.2.2 Sensitivity function

Now consider the control system shown in Figure 6.9.

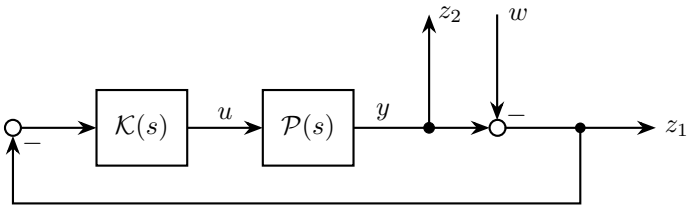

Figure 6.9: Example of control system

In addition,

$$\begin{aligned} \mathcal{P}(s) &: \text{Plant} \\ \mathcal{K}(s) &: \text{Controller} \\ u &: \text{input} \\ y &: \text{output} \\ w &: \text{external input} \end{aligned}$$

If w is the reference of the output y to be controlled in Figure 6.9, the objective of control is $y \to w$. In other words, if $z_1 = y - w \to 0$ (to reduce the deviation between the reference and the output), the output y can follow the reference w.

Here, the transfer function from w to z_1 is

$$z_1 = -\frac{1}{1 + \mathcal{P}(s)\mathcal{K}(s)} w. \tag{6.25}$$

If we write

$$\mathcal{S}(s) = \frac{1}{1 + \mathcal{P}(s)\mathcal{K}(s)}, \tag{6.26}$$

then

$$z_1 = -\mathcal{S}(s)w. \tag{6.27}$$

From this, if the gain of $\|\mathcal{S}(s)\|_\infty$ is small, $Z_1 = y - w$ will be small. In other words, control to make the output y follow the reference w can be achieved by making $\|\mathcal{S}(s)\|_\infty (= \| - \mathcal{S}(s)\|_\infty)$ as small as possible. This $\mathcal{S}(s)$ is called **sensitivity function**.

When evaluating a control system focusing on tracking characteristics, it is sufficient to select

$$J = \|\mathcal{S}(s)\|_\infty \tag{6.28}$$

as the evaluation function and minimize it, but it is impossible to make $\mathcal{S}(s)$ small over the entire frequency band. To make $\mathcal{S}(s)$ small over the entire frequency band means that the output of the control target follows signals of all frequencies, but in the case of damping control of a suspension system, for example, there is no need to control at several kHz, and it is also scary to think of a pendulum that can swing at such frequencies. It is scary to think of a pendulum that can swing at such a frequency.

Therefore, we can build a control system in which the output follows only the target signal in a part of the frequency band (basically low frequency), so that the gain of the sensitivity function $\mathcal{S}(s)$) becomes smaller in this band.

To obtain such a sensitivity function, consider a control specification using a frequency weighting transfer function $\mathcal{W}_\mathcal{S}(s)$ as expressed in the following equation.

$$\|\mathcal{W}_\mathcal{S}(s)\mathcal{S}(s)\|_\infty < 1 \tag{6.29}$$

This can be thought of as a problem of making the $\mathcal{H}_\infty$ norm of the transfer function $-\mathcal{W}_\mathcal{S}(s)\mathcal{S}(s)$ from w to z less than 1 in Figure 6.9 ($\because \|\mathcal{W}_\mathcal{S}(s)\mathcal{S}(s)\|_\infty = \| - \mathcal{W}_\mathcal{S}(s)\mathcal{S}(s)\|_\infty$).

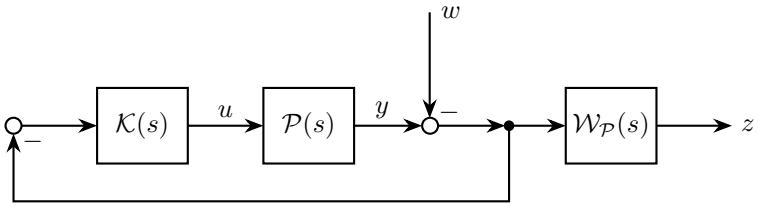

Figure 6.10: Frequency weighting transfer function

Assuming that equation 6.29 can be realized, from the definition of the $\mathcal{H}_\infty$ norm 6.20,

$$|\mathcal{W}_{\mathcal{S}}(j\omega)\mathcal{S}(j\omega)| < 1, \quad \forall\omega \tag{6.30}$$

Namely,

$$|\mathcal{S}(j\omega)| < \frac{1}{|\mathcal{W}_{\mathcal{S}}(j\omega)|}, \quad \forall\omega. \tag{6.31}$$

This is depicted in the Bode plot in Fig. 6.11, where the sensitivity function $\mathcal{S}(s)$ is shaped to be less than $1/\mathcal{W}_{\mathcal{S}}(s)$ using the frequency weight transfer function. Also, at frequencies where $|\mathcal{W}_{\mathcal{S}}(j\omega)|$ is sufficiently large ($/|\mathcal{W}_{\mathcal{S}}(j\omega)|$ is sufficiently small), $|\mathcal{S}(j\omega)| \approx 0$. For a reference signal in such a frequency band, $Z_1 = y - w$ is small. In other words, the output y follows the target value w.

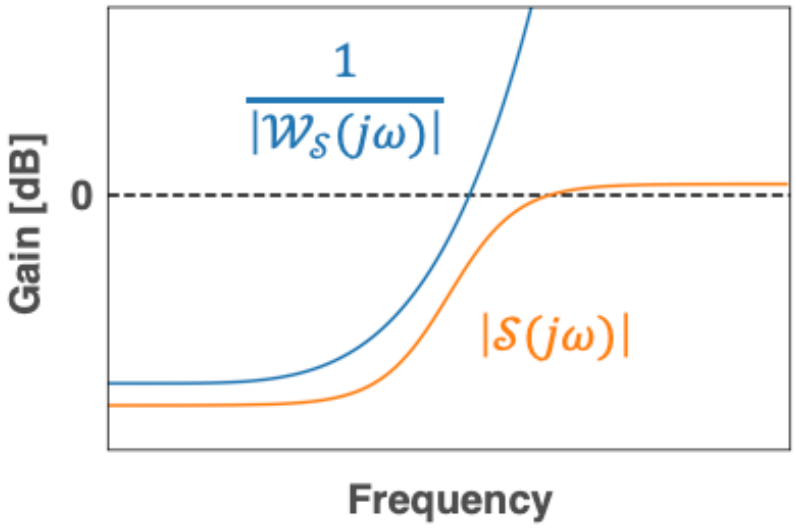

Figure 6.11: Shaping of sensitivity functions

Thus, the frequency response of the sensitivity function $\mathcal{S}(s)$ can be shaped using the frequency weighting function $\mathcal{W}_{\mathcal{S}}(s)$.

By the way, we haven't talked about phase at all for a while, but there is not much of a problem. In classical control, controllers are basically designed with sufficient phase and gain margins for the transfer function $\mathcal{P}(s)\mathcal{K}(s)$, which is defined for the open-loop system. Then, the sensitivity function $\mathcal{S}(s)$ of the closed-loop system is reduced by adjusting the

phase and gain margins in the open-loop system. On the other hand, the concept of phase margin and gain margin is unnecessary here because the sensitivity function $\mathcal{S}(s)$, which is the characteristic of the transfer function of the closed-loop system, is adjusted directly.

6.2.3 Robust stabilization problem

As mentioned at the beginning of this section, when actually constructing a control system, a nominal model $\mathcal{P}(s)$ is constructed for an actual plant $\mathcal{P}_r(s)$, and a controller $\mathcal{K}(s)$ is designed based on it. At this time, an error

$$\Delta_a(s) = P_r(s) - P(s) \tag{6.32}$$

exists between the control target $\mathcal{P}_r(s)$ and the nominal model $\mathcal{P}(s)$. Therefore, the controller $\mathcal{K}(s)$ should be designed to stabilize both the nominal model $\mathcal{P}(s)$ and the actual control target $\mathcal{P}_r(s)$. Such a controller is called a robust controller (i.e., a controller that can be stabilized even in the presence of model errors).

Here, the actual transfer function of the control target is $\mathcal{P}_r(s)$ and the nominal model transfer function (to be known) is $\mathcal{P}(s)$, and the additive error $\Delta_a(s)$ is used to denote

$$\mathcal{P}_r(s) = \mathcal{P}(s) + \Delta_a(s). \tag{6.33}$$

Also, assume that the number of unstable poles in $\mathcal{P}_r(s)$ and $\mathcal{P}(s)$ are equal and that $\Delta_a(s)$ is bounded by the known stable transfer function $\mathcal{W}_{\mathcal{A}}(s)$ in size as

$$|\Delta_a(j\omega)| < |\mathcal{W}_{\mathcal{A}}(j\omega)|, \quad \forall\omega \tag{6.34}$$

(which would be expressed as $\bar{\sigma}\{\Delta_a(j\omega)\} < |\mathcal{W}_{\mathcal{A}}(j\omega)|$ in MIMO model).

The robust stabilization problem is to design a controller $\mathcal{K}(s)$ such that the closed-loop system is stable for all plant $\mathcal{P}_r(s)$ expressed in equation 6.33 by any error $\Delta_a(s)$ satisfying equation 6.34.

In designing a controller $\mathcal{K}(s)$ that stabilizes Figure 6.9, if the additive error $\Delta_a(s)$ is considered, the actual closed-loop system is as shown in Figure 6.12.

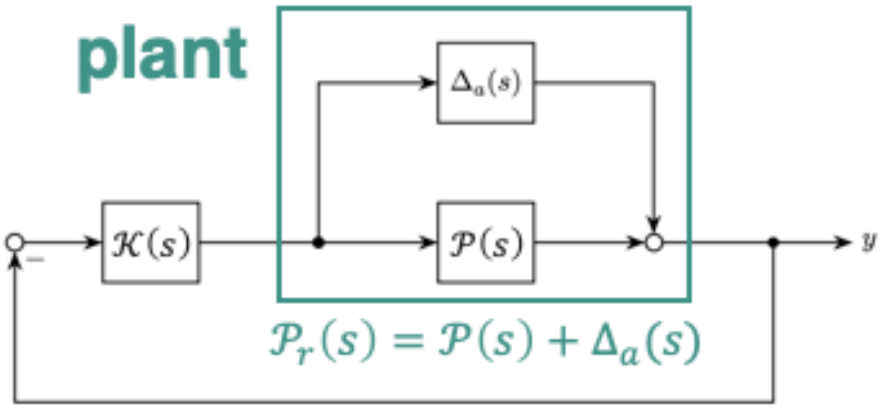

Figure 6.12: Closed-loop system with additive error

From Nyquist's stability theorem, if

$$\left\| \mathcal{W}_{\mathcal{A}}(s) \frac{\mathcal{K}(s)}{1 + \mathcal{P}(s)\mathcal{K}(s)} \right\|_{\infty} < 1, \tag{6.35}$$

the closed-loop system in Figure 6.12 is stable regardless of $\Delta_a(s)$.

As mentioned above, **Robust stabilization against additive errors is the problem of designing the controller $\mathcal{K}(s)$ to satisfy equation 6.35 using the $\mathcal{H}_\infty$ norm.**

Since the transfer function from w to z in Figure 6.13 is

$$-\frac{\mathcal{W}_{\mathcal{A}}(s)\mathcal{K}(s)}{1 + \mathcal{P}(s)\mathcal{K}(s)}, \tag{6.36}$$

robust stabilization against additive error can be considered to be the problem of reducing the $\mathcal{H}_\infty$ norm of this transfer function to less than 1. In Figure 6.13, w is the signal leaving $\Delta_a(s)$ in Figure 6.12, and z is the signal entering $\Delta_a(s)$ multiplied by the frequency weighting transfer function $\mathcal{W}_{\mathcal{A}}$.

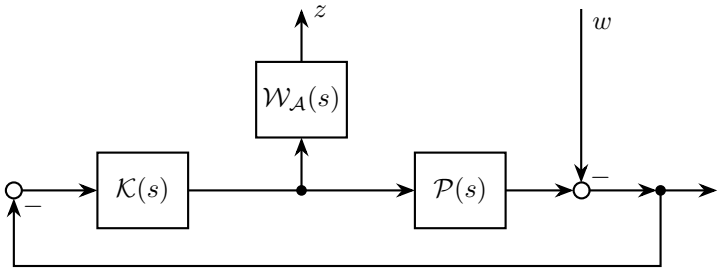

Figure 6.13: Robust stabilization problem for additive errors

Now, up to this point, we have considered the error between the actual plant and the nominal model as additive, but as we saw at the beginning of the section, it is also possible to consider multiplicative errors.

$$\mathcal{P}_r(s) = (1 + \Delta_t(s))\mathcal{P}(s) \tag{6.37}$$

Also, as in the additive error case, the number of unstable poles in $\mathcal{P}_r(s)$ and $\mathcal{P}(s)$ are equal, and the multiplicative error $\Delta_t(s)$ is bounded by the size of the stable transfer function $\mathcal{W}_{\mathcal{T}}$, as in A

$$|\Delta_t(j\omega)| < |\mathcal{W}_{\mathcal{T}}(j\omega)|, \quad \forall\omega. \tag{6.38}$$

In designing a controller $\mathcal{K}(s)$ that stabilizes Figure 6.9, if the multiplicative error $\Delta_t(s)$ is considered, the actual closed-loop system is as shown in Figure 6.14.

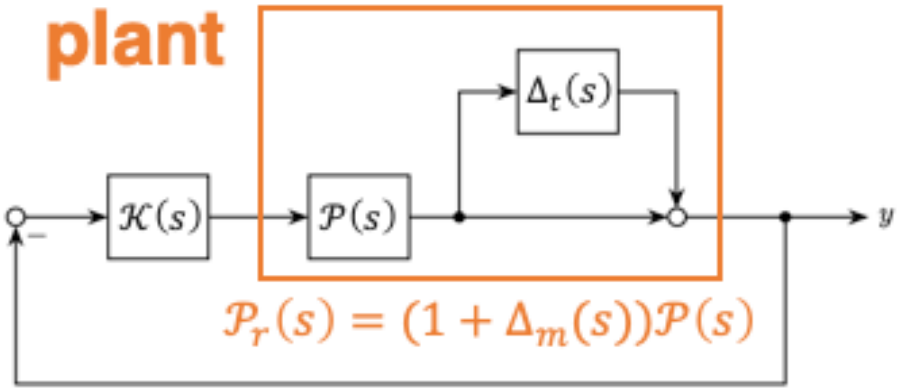

Figure 6.14: Closed-loop system when multiplicative error is considered

From Nyquist's stability theorem, if

$$\left\| \mathcal{W}_{\mathcal{T}}(s) \frac{\mathcal{P}(s)\mathcal{K}(s)}{1 + \mathcal{P}(s)\mathcal{K}(s)} \right\|_{\infty} < 1, \tag{6.39}$$

then the closed-loop system in Figure 6.14 is stable, regardless of $\Delta_t(s)$.

where

$$\mathcal{T}(s) \equiv \frac{\mathcal{P}(s)\mathcal{K}(s)}{1 + \mathcal{P}(s)\mathcal{K}(s)} \tag{6.40}$$

is defined to be

$$\|\mathcal{W}_{\mathcal{T}}(s)\mathcal{T}(s)\|_{\infty} < 1 \tag{6.41}$$

and **Robust stabilization against multiplicative error is the problem of designing the controller $\mathcal{K}(s)$ to satisfy equation 6.41 using the $\mathcal{H}_\infty$ norm**. Note that this $\mathcal{T}(s)$ is called the **complementary sensitivity function** and is the transfer function from w to z_2 in Figure 6.9.

Also, since the transfer function from w to z in Figure 6.15 is

$$-\mathcal{W}_{\mathcal{T}}(s)\mathcal{T}(s), \tag{6.42}$$

robust stabilization against multiplicative error is considered to be a problem of making the $\mathcal{H}_\infty$ norm of this transfer function less than 1. In Figure 6.15, w is the signal leaving $\Delta_t(s)$ in Figure 6.14, and z is the signal entering $\Delta_t(s)$ multiplied by the frequency weighting transfer function $\mathcal{W}_{\mathcal{T}}(s)$. which is z.

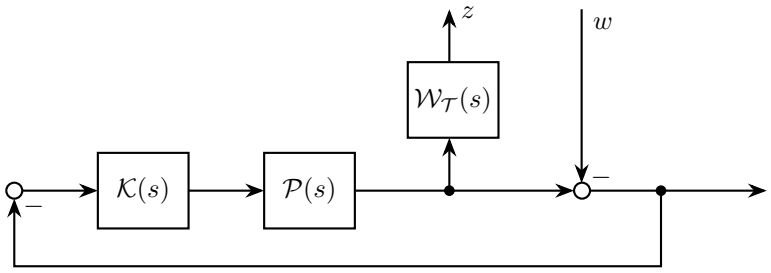

Figure 6.15: Robust stabilization problem for multiplicative errors

6.2.4 Mixed sensitivity problem

From the above, it can be seen that We can design the controller $\mathcal{K}(s)$ so that
Focus on target tracking characteristics $\cdots \|\mathcal{W}_\mathcal{S}(s)\mathcal{S}(s)\|_\infty < 1$
Focus on robust stability $\|\mathcal{W}_\mathcal{T}(s)\mathcal{T}(s)\|_\infty < 1$

In general, since both target tracking and robust stability are important characteristics in control, we want to design a control system that takes both into account. However, it is difficult to find a controller that satisfies both of these separately, so we consider a mixed sensitivity problem that minimizes γ in the following equation.

$$\left\| \begin{pmatrix} \mathcal{W}_\mathcal{S}(s)\mathcal{S}(s) \\ \mathcal{W}_\mathcal{T}(s)\mathcal{T}(s) \end{pmatrix} \right\|_\infty < \gamma \tag{6.43}$$

If we can find a controller $\mathcal{K}(s)$ that satisfies this, then even in the case of MIMO systems, from property

$$\|\mathcal{G}_i(s)\|_\infty \leq \left\| \begin{pmatrix} \mathcal{G}_1(s) & \mathcal{G}_2(s) \\ \mathcal{G}_3(s) & \mathcal{G}_4(s) \end{pmatrix} \right\|_\infty, \quad i = 1, 2, 3, 4 \tag{6.44}$$

of the $\mathcal{H}_\infty$ norm, it becomes

$$\|\mathcal{W}_\mathcal{S}(s)\mathcal{S}(s)\|_\infty < 1, \ \|\mathcal{W}_\mathcal{T}(s)\mathcal{T}(s)\|_\infty < 1 \tag{6.45}$$

and satisfies the desired property. Also, since the transfer function from w to $z = (z_1, \ z_2)^\top$ in Fig. 6.16 is

$$\begin{pmatrix} -\mathcal{W}_\mathcal{S}(s)\mathcal{S}(s) \\ \mathcal{W}_\mathcal{T}(s)\mathcal{T}(s) \end{pmatrix} \tag{6.46}$$

and

$$\left\| \begin{pmatrix} -\mathcal{W}_\mathcal{S}(s)\mathcal{S}(s) \\ \mathcal{W}_\mathcal{T}(s)\mathcal{T}(s) \end{pmatrix} \right\|_\infty = \left\| \begin{pmatrix} \mathcal{W}_\mathcal{S}(s)\mathcal{S}(s) \\ \mathcal{W}_\mathcal{T}(s)\mathcal{T}(s) \end{pmatrix} \right\|_\infty \tag{6.47}$$

the mixed sensitivity problem can be considered as a problem in which the $\mathcal{H}_\infty$ norm of the transfer function from w to $z = (z_1, \ z_2)^\top$ in Fig. 6.16 is less than 1. Here, w in the figure 6.16 is the reference input when considering the sensitivity function, and is a virtual signal for evaluating the error when considering the complementary sensitivity function. Note that the meaning of the signal changes depending on the viewpoint in $\mathcal{H}_\infty$ control.

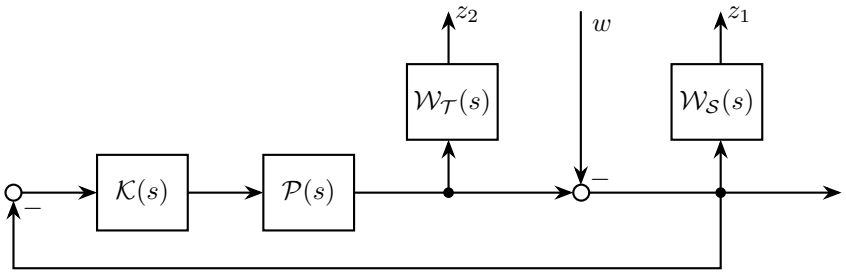

Figure 6.16: Mixed sensitivity problem

Furthermore, from the definition of sensitivity function $\mathcal{S}(s)$ and complementary sensitivity function $\mathcal{T}(s)$, it is

$$\mathcal{S}(s) + \mathcal{T}(s) = 1, \tag{6.48}$$

and $\mathcal{S}(s)$ and $\mathcal{T}(s)$ cannot be reduced simultaneously at the same frequency. Here,

$$|\mathcal{S}(j\omega)| < \frac{1}{|\mathcal{W}_\mathcal{S}(j\omega)|} \tag{6.49}$$

$$|\mathcal{T}(j\omega)| < \frac{1}{|\mathcal{W}_\mathcal{T}(j\omega)|} \tag{6.50}$$

is true when the equation 6.45 is satisfied. From this, if $\mathcal{W}_\mathcal{S}(j\omega) \gg 1$ at a certain frequency ω, then $\mathcal{T}(j\omega) \approx 0$, but at this time $\mathcal{T}(j\omega) \approx 1$, so $|\mathcal{W}_\mathcal{T}(j\omega)| \ll 1$. In other words, the frequency bands of $\mathcal{W}_\mathcal{S}(s)$ and $\mathcal{W}_\mathcal{T}(s)$ must be separated.

In general, the multiplicative error $\Delta_t(j\omega)$ is often large at high frequency bands, so $|\mathcal{W}_\mathcal{T}(j\omega)|$ must be large at high frequency, but can be small at low frequency. On the other hand, since it is often sufficient to achieve target tracking characteristics at low frequencies, $|\mathcal{W}_\mathcal{S}(j\omega)|$ can be large at low frequencies and small at high frequencies. Therefore, when considering the mixed sensitivity problem, the frequency weight function is often set as shown in Figure 6.17.

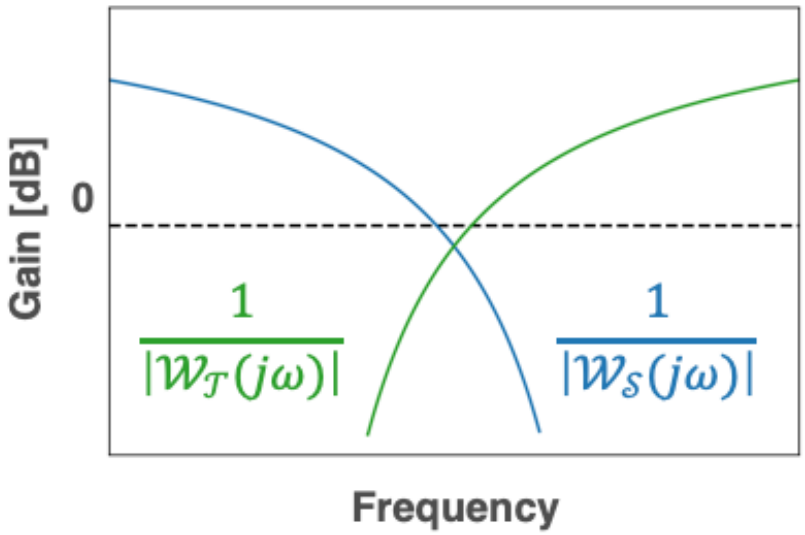

Figure 6.17: Frequency weighting transfer function in mixed sensitivity problems

6.2.5 Design of robust control in Python

Based on the above, we describe an example of robust control design using Python. Here, the mixed sensitivity problem (Eq. 6.43) is solved with the frequency weighting transfer function as

$$\mathcal{W}_S(s) = \frac{1}{(s+0.5)} \tag{6.51}$$

$$\mathcal{W}_T(s) = \frac{10s}{(s+150)} \tag{6.52}$$

In Python, we can use `mixsyn` as in `K, cl, info = mixsyn(sys, w1, w2, w3)`. Note that the expression numbers `sys` denote the nominal model, and `w1` and `w3` denote $\mathcal{W}_S(s)$ and $\mathcal{W}_T(s)$, respectively (`w2` is assumed to be 1 here). The controller `K` that minimizes the γ of equation 6.43 is now obtained, and its value is stored in the return value `info`.

Now, executing the following code, we find that $\gamma = 0.9527651218302327 < 1$, and also obtain figures 6.18 and 6.19.

```
# Robust Controller
from control import mixsyn

WS = tf([0, 1], [1, 1, 0.25]) # sensitivity function
WU = tf(1,1)
WT = tf([10, 0], [1, 150])    # complementary sensitivity function

## mix sensitivity problem
K, _, info = mixsyn(Pn, w1=WS, w2=WU, w3=WT)
print('K =', ss2tf(K))
print('gamma =', info[0])

fig, ax = plt.subplots(1, 2, figsize=(15,5))

## sensitivity function
```

```
Ssys = feedback(1, Pn*K)
mag, _, w = bode(Ssys, logspace(-3,3), plot=False)
ax[0].semilogx(w, mag2db(mag), ls='-', label='$S$')
mag, _, w = bode(1/WS, logspace(-3,3), plot=False)
ax[0].semilogx(w, mag2db(mag), ls='-', label='$1/W_S$')

## complementary sensitivity function
Tsys = feedback(Pn*K, 1)
mag, _, w = bode(Tsys, logspace(-3,3), plot=False)
ax[1].semilogx(w, mag2db(mag), ls='-', label='$T$')
mag, _, w = bode(1/WT, logspace(-3,3), plot=False)
ax[1].semilogx(w, mag2db(mag), ls='-', label='$1/W_T$')

for i in range(2):
    ax[i].set_ylim(-40, 40)
    ax[i].legend()
    ax[i].grid(which='both', ls=':')
    ax[i].set_xlabel('$\omega$ [rad/s]')
    ax[i].set_ylabel('Gain [dB]')

#plt.savefig('Sensitivity_and_ComplementarySensitivity_function.png', dpi=300)

## robust controller
fig, ax = plt.subplots(figsize=(7, 5))
ref = 30 # reference angle

## performance for the model with uncertainty
for i in range(len(delta)):
    P = (1+WT*delta[i])*Pn
    Gyr = feedback(P*K, 1)
    y, t = step(Gyr, np.arange(0, 5, 0.01))
    ax.plot(t, y*ref)


## performance for nominal model
Gyr = feedback(Pn*K, 1)
y, t = step(Gyr, np.arange(0, 5, 0.01))
ax.plot(t, y*ref, lw=1.5, color='k')
ax.set_xlim([0, 3])
plot_set(ax, 't', 'y')

#plt.savefig('RobustController.png', dpi=300)
```

Listing 6.7: Robust control

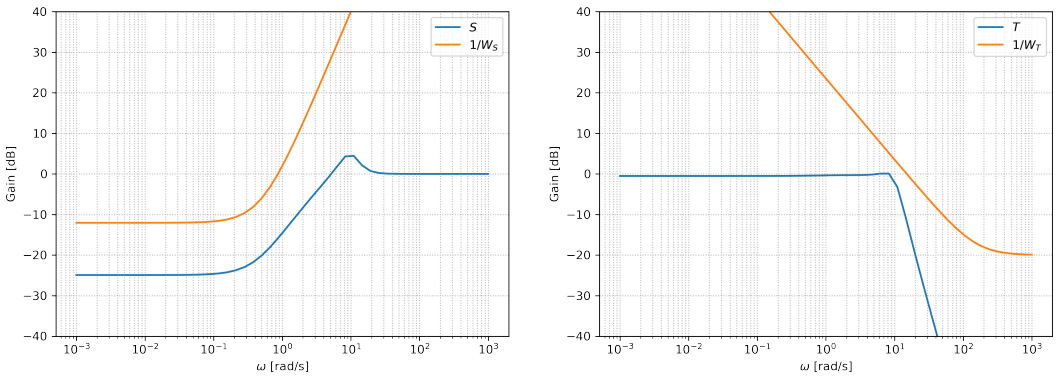

Figure 6.18: Sensitivity functions and complementary sensitivity functions

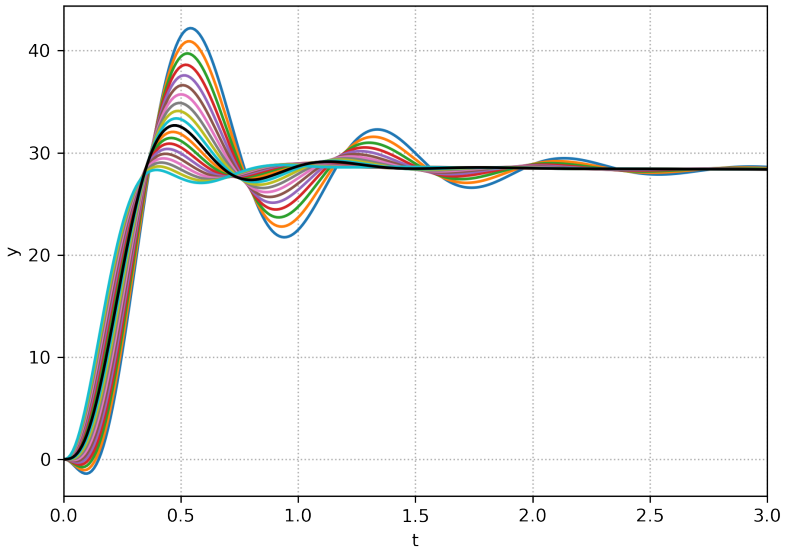

Figure 6.19: Robust control

From Figure 6.18, we can see that the sensitivity function and the complementary sensitivity function are below the gain diagram of $\frac{1}{W_S}$ and $\frac{1}{W_T}$, respectively.

Figure 6.19 shows the step response when the designed controller is used. From this, it can be seen that the response follows the target value promptly and that the response does not change much even with uncertainty.

6.2.6 Solution of $\mathcal{H}_\infty$ control

Regarding the $\mathcal{H}_\infty$ control using the $\mathcal{H}_\infty$ norm, we will discuss its solution in a little more detail, just to be sure.

6.2.6.1 Standard problem

Various control specifications are possible, but here, we define a standard problem to consider them all together. As mentioned above, the control specification of $\mathcal{H}_\infty$ control is defined by the $\mathcal{H}_\infty$ norm of the transfer function from input w to output z in a closed loop system. Therefore, the standard problem is shown in Figure 6.20.

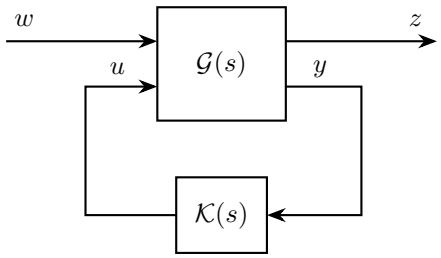

Figure 6.20: Standard problem

In the figure, $\mathcal{G}(s)$ is called the generalized plant, where the inputs are the external input w and the operating quantity u and the outputs are the control quantity z and the observed output y. The state space is assumed to be given by

$$\dot{x} = Ax + B_1 w + B_2 w \tag{6.53}$$
$$z = C_1 x + D_{11} w + D_{12} u \tag{6.54}$$
$$y = C_2 x + D_{21} w \tag{6.55}$$

Also, the controller $\mathcal{K}(s)$ is represented by the transfer function from the observed output y to the manipulated quantity u

$$u = \mathcal{K}(s) y. \tag{6.56}$$

Now, the control objective here is to stabilize the system and minimize (less than 1) the $\mathcal{H}_\infty$ norm of the transfer function $\mathcal{G}_{zw}(s)$ from the external input w to the control quantity z

$$\|\mathcal{G}_{zw}(s)\|_\infty < 1. \tag{6.57}$$

6.2.6.2 Generalized plant in mixed sensitivity problems

As an example, let us solve the mixed sensitivity problem in equation 6.43 using the standard problem.

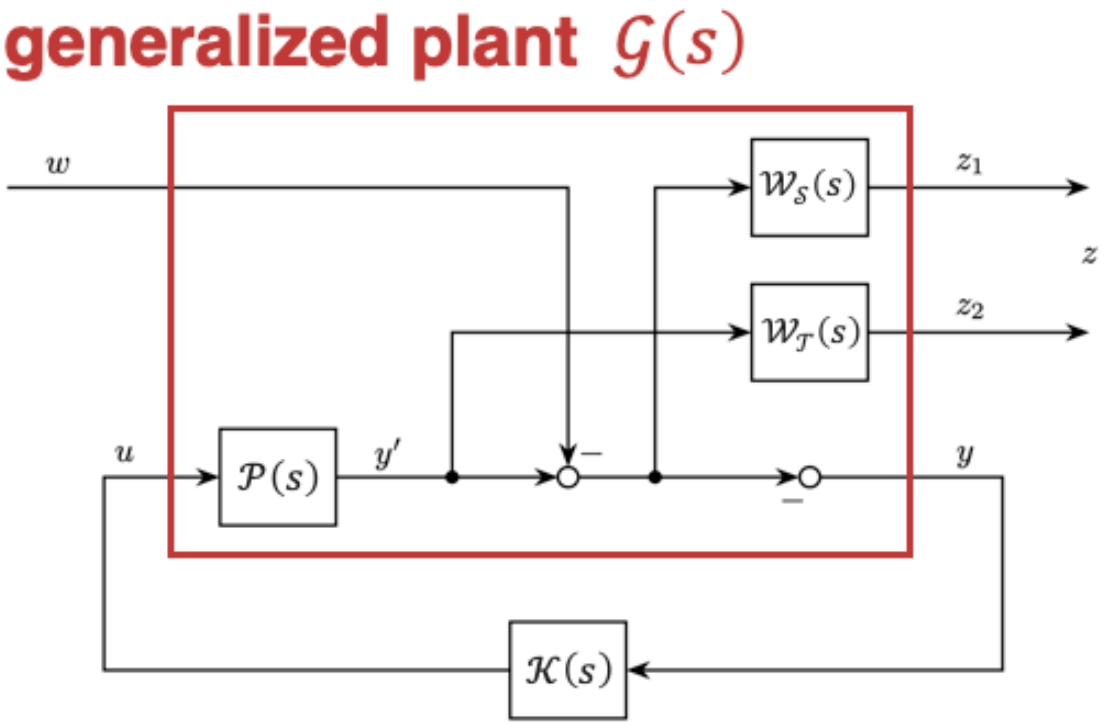

Figure 6.21: Solving mixed sensitivity problems using standard problem

If we transform the block diagram, we can see that the systems in Figures 6.16 and 6.21 are equivalent. Therefore, the mixed sensitivity problem in Equation 6.43 is to find the controller $\mathcal{K}(s)$ that makes the $\mathcal{H}_\infty$ norm of $\mathcal{G}_{zw}(s)$ less than 1.

To distinguish it from the observed output y of the generalized plant, let the output of the plant $\mathcal{P}(s)$ be y', and the state equation representation of $\mathcal{P}(s)$ can be

$$\dot{x} = Ax + Bu \tag{6.58}$$
$$y' = Cx \tag{6.59}$$

At this time, since the input to the frequency weighting transfer function $\mathcal{W}_\mathcal{S}(s)$ is $(y' - w)$, the equation of state representation of $\mathcal{W}_\mathcal{S}$ is

$$\dot{x}_{\mathcal{W}_S} = A_{\mathcal{W}_S} x_{\mathcal{W}_S} + B_{\mathcal{W}_S}(y' - w) \tag{6.60}$$
$$z_1 = C_{\mathcal{W}_S} x_{\mathcal{W}_S} + D_{\mathcal{W}_S}(y' - w), \tag{6.61}$$

and $\mathcal{W}_\mathcal{T}$ is denoted by

$$\dot{x}_{\mathcal{W}_T} = A_{\mathcal{W}_T} x_{\mathcal{W}_T} + B_{\mathcal{W}_T} y' \tag{6.62}$$
$$z_2 = C_{\mathcal{W}_T} x_{\mathcal{W}_T} + D_{\mathcal{W}_T} y'. \tag{6.63}$$

Using the fact that the observed output of the generalized plant is $y = -(y' - w)$ and

$y' = Cx$, the equation of state of the generalized plant is obtained as

$$\frac{\mathrm{d}}{\mathrm{d}t}\begin{pmatrix} x \\ x_{\mathcal{W}_S} \\ x_{\mathcal{W}_T} \end{pmatrix} = \begin{pmatrix} A & 0 & 0 \\ B_{\mathcal{W}_S}C & A_{\mathcal{W}_S} & 0 \\ B_{\mathcal{W}_T}C & 0 & A_{\mathcal{W}_T} \end{pmatrix}\begin{pmatrix} x \\ x_{\mathcal{W}_S} \\ x_{\mathcal{W}_T} \end{pmatrix} + \begin{pmatrix} 0 \\ -B_{\mathcal{W}_S} \\ 0 \end{pmatrix} w + \begin{pmatrix} B \\ 0 \\ 0 \end{pmatrix} u \tag{6.64}$$

$$\begin{pmatrix} z_1 \\ z_2 \end{pmatrix} = \begin{pmatrix} D_{\mathcal{W}_S}C & C_{\mathcal{W}_S} & 0 \\ D_{\mathcal{W}_T}C & 0 & C_{\mathcal{W}_T} \end{pmatrix}\begin{pmatrix} x \\ x_{\mathcal{W}_S} \\ x_{\mathcal{W}_T} \end{pmatrix} + \begin{pmatrix} -D_{\mathcal{W}_S} \\ 0 \end{pmatrix} w + \begin{pmatrix} 0 \\ 0 \end{pmatrix} u \tag{6.65}$$

$$y = \begin{pmatrix} -C & 0 & 0 \end{pmatrix}\begin{pmatrix} x \\ x_{\mathcal{W}_S} \\ x_{\mathcal{W}_T} \end{pmatrix} + Iw. \tag{6.66}$$

By solving the standard problem using this generalized plant, we can design a controller $\mathcal{K}(s)$ that satisfies the mixed sensitivity problem.

6.2.6.3 How to solve standard problem

The standard problem is solved by solving the two Riccati equations

$$\begin{aligned} &XA + A^\top X + \frac{1}{\gamma^2}XB_1B_1^\top X \\ &\quad - (C_1^\top D_{12} + XB_2)(D_{12}^\top D_{12})^{-1}(D_{12}^\top C_1 + B_2^\top X) + C_1^\top C_1 = 0 \end{aligned} \tag{6.67}$$

$$\begin{aligned} &YA^\top + AY + \frac{1}{\gamma^2}YC_1^\top C_1 Y \\ &\quad - (B_1D_{21}^\top + YC_2^\top)(D_{21}D_{21}^\top)^{-1}(D_{21}B_1^\top + C_2Y) + B_1B_1^\top = 0 \end{aligned} \tag{6.68}$$

for the generalized plant, checking to see if there exists a controller that satisfies equation 6.57, and if so, finding that controller. The assumption here is

1. (A, B_2) is stable
2. (C_2, A) is detectable
3. D_{12} is column full rank
4. D_{21} is row full rank
5. $\begin{pmatrix} A - j\omega I & B_2 \\ C_1 & D_{12} \end{pmatrix}$ is column full rank
6. $\begin{pmatrix} A - j\omega I & B_1 \\ C_2 & D_{21} \end{pmatrix}$ is row full rank
7. $D_{11} = 0,\ D_{22} = 0$

Of these, 1 and 2 are necessary to stabilize the control system, and 3∼6 is (roughly) valid if the poles and zeros of the actual control object and weight transfer function are not on the imaginary axis (on the origin). Also, the solution would be complicated without the condition 7.

Now, after solving the Riccati equations 6.67 and 6.68, select the solution that satisfies condition

- $X = X^\top \geq 0$ (quasi-definite)
- $Y = Y^\top \geq 0$ (quasi-definite)
- $\rho(XY) < \gamma^2$ ($\rho(A) = \max|\lambda_i(A)|$: Maximum eigenvalue)

When X, Y is thus obtained, the $\mathcal{H}_\infty$ controller is constructed as

$$\frac{\mathrm{d}\hat{x}}{\mathrm{d}t} = \hat{A}\hat{x} + \hat{B}y \tag{6.69}$$

$$u = F\hat{x} \tag{6.70}$$

However,

$$\begin{aligned}
\hat{A} &= A + \gamma^{-2}B_1B_1^TX + B_2F + ZL_y(C_2 + \gamma^{-2}D_{21}B_1^TX) && (6.71)\\
\hat{B} &= -ZL_y && (6.72)\\
F &= -(D_{12}^TD_{12})^{-1}(D_{12}^TC_1 + B_2^TX) && (6.73)\\
L_y &= -(B_1D_{21}^T + YC_2^T)(D_{21}D_{21}^T)^{-1} && (6.74)\\
Z &= (I - \gamma^{-2}YX)^{-1} && (6.75)
\end{aligned}$$

This solution is called the central solution and is commonly used.

From this central solution 6.69 and 6.70, the $\mathcal{H}_\infty$ controller is

$$\mathcal{K}(s) = F(sI - \hat{A})^{-1}\hat{B} \tag{6.76}$$

6.3 Optimal control

6.3.1 What is optimal control

An optimal control problem is the problem of finding the control input that minimizes the evaluation function, given a mathematical model of the system expressed in terms of differential equations, constraints, and an evaluation function.

The basic optimal control is formulated as follows

Optimal control

The equation of state $\dot{\boldsymbol{x}} = f(\boldsymbol{x}, \boldsymbol{u}(t))$ $(\boldsymbol{x} \in \mathbb{R}^n,\ \boldsymbol{u} \in \mathbb{R}^m)$ and initial conditions $\boldsymbol{x}(0) = \boldsymbol{x}_0$, equality constraints $h_i(\boldsymbol{x}, \boldsymbol{u}) = 0,\ (i = 1, 2, ...q)$, inequality constraint $g_i(\boldsymbol{x}, \boldsymbol{u}) \leq 0,\ (i = 1, 2, ...k)$, find the one that minimizes the evaluation function

$$J = \int_0^T L(\boldsymbol{x}(t), \boldsymbol{u}(t))\mathrm{d}t + V(\boldsymbol{x}(T)) \tag{6.77}$$

for $\boldsymbol{x}(t),\ \boldsymbol{u}(t)$.

The first term of the evaluation function is the cost related to the behavior of the system, such as the speed of response and the magnitude of the input. On the other hand, the second term is the cost related to the end state, where T is the end time. Equality and inequality constraints are used to set the upper and lower bounds of the input, the value of the termination state, and so on. In particular, those with $T = \infty$ are called infinite-time optimal control problems, and those with $T < \infty$ are called finite-time optimal control problems.

Among optimal control problems, those that minimize the evaluation function

$$J = \int_0^T (\boldsymbol{x}(t)^\top \boldsymbol{Q}\boldsymbol{x}(t) + \boldsymbol{u}(t)^\top \boldsymbol{R}\boldsymbol{u}(t))\mathrm{d}t + \boldsymbol{x}(T)^\top \boldsymbol{Q}_f \boldsymbol{x}(T)) \tag{6.78}$$

in quadratic form for a dynamic system $\dot{\boldsymbol{x}} = \boldsymbol{A}\boldsymbol{x} + \boldsymbol{B}\boldsymbol{u}$ are especially called LQ optimal control problems. In the LQ optimal control problem, the case where $T = \infty$ is the optimal regulator (see 4.4.3).

As a simple example, let us consider the following case of system

$$\dot{x}(t) = x(t) + u(t),\ x(0) = 1 \tag{6.79}$$

given the constraints $-4 \leq u(t) \leq 4$ on the inputs, $x(T) = 0$ at the end time of the state, and the evaluation function

$$J = \int_0^T 50x(t)^2 + 0.1u(t)^2\mathrm{d}t \tag{6.80}$$

to find the control input $u(t)$ $(0 \leq t \leq T = 0.5)$ that minimizes J.

For this, use the `quadratic_cost` function to set the evaluation function, and use the `input_range_constraint` and `state_range_constraint` functions to describe the constraint conditions. The `solve_opc` function is used to solve the optimal control problem. Based on the above, execute the following code to obtain Figure 6.22.

```
# Optimal Control
import control as ct
import control.optimal as obc

P = ct.ss(1, 1, 1, 0)  # system
sys = ct.ss2io(P)      # ss to input/output model

xf = 0  # reference state / terminal state
uf = 0  # reference input
Q = 50  # weight for state
```

```
R = 0.1 # weight for input

cost = obc.quadratic_cost(sys, Q, R, x0=xf, u0=uf)  # cost function

umin, umax = -4, 4
constraints = [obc.input_range_constraint(sys, [umin], [umax])]
terminal = [obc.state_range_constraint(sys, [xf], [xf])]

Td = np.arange(0, 0.5, 0.01)  # simulation time
x0 = 1  # initial state

result = obc.solve_ocp(sys, Td, x0, cost=cost, constraints=constraints, terminal_constraints=
    terminal)

resp = ct.input_output_response(sys, Td, result.inputs, x0)

fig, ax = plt.subplots(1, 2, figsize=(15,5))
ax[0].plot(resp.time, resp.outputs)  # output
ax[1].plot(resp.time, resp.inputs)   # input
plot_set(ax[0], 't', 'x')
plot_set(ax[1], 't', 'u')

#plt.savefig('OptimalController.png', dpi=300)
```

Listing 6.8: Finite-time optimum control

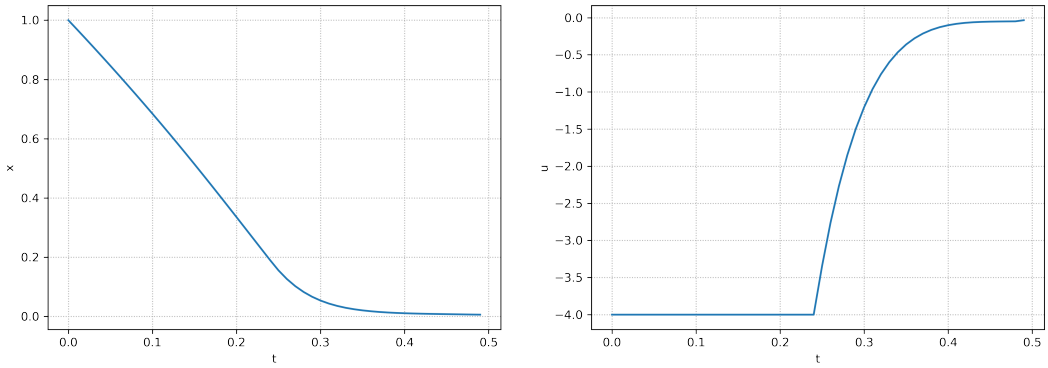

Figure 6.22: Finite-time optimum control

From this, it can be seen that the state x reaches 0 at time $t = 0.5$ while satisfying the input constraints, and the solution to the optimal control problem is obtained. It should be noted here that since the control target is unstable, it will not reach $x = 0$ without applying appropriate inputs.

6.3.2 Model predictive control

For example, when considering the case of controlling the anti-vibration suspension system in KAGRA, ground vibration changes from moment to moment, so the optimal filter also changes from moment to moment. Therefore, the ground vibration from the present moment to a short time in the future is grasped, and the filter is determined using that

condition. By repeating this process, the optimal control filter can always be used. This method is called model predictive control.

In other words, model predictive control uses a model to predict the motion up to a finite time in the future and solves a finite-time optimal control problem to determine the current input. The evaluation function in model predictive control is

$$J = \int_t^{t+N} L(\boldsymbol{x}(\tau), \boldsymbol{u}(\tau))\mathrm{d}\tau + V(\boldsymbol{x}(t+N)). \tag{6.81}$$

The optimal control problem is solved with $\boldsymbol{x}(t)$ as the initial state at each time. Then, we can repeat the process using only the initial value $\boldsymbol{u}(t)$ of the obtained control input $\boldsymbol{u}(\tau)$ $(t \leq \tau \leq t+N)$ as the actual control input $\cdots\cdots$. Note that N, which determines the evaluation interval, is called the prediction horizon.

Model predictive control does not necessarily make the closed-loop system stable because the evaluation interval is finite. However, since the control input is obtained by moving the evaluation interval, feedback control can be performed continuously. Another advantage of model predictive control is that it can handle various control targets and various problems by setting evaluation functions and constraint conditions.

The following describes an example of executing Model Predictive Control (MPC) in Python. Here, we target a second-order discrete-time model (see 6.4), and the evaluation function is quadratic. The Model Predictive Controller is defined using the `create_mpc_iosystem` function, with a prediction horizon of $N = 5$. Finally, the feedback control system composed of the plant and the MPC is obtained using the `interconnect` function, and the time response is calculated.

Based on the above, executing the following code will yield Figure 6.23.

```
# Model Predictive Control
import control as ct
import control.optimal as obc

## plant
A = [[0, 1], [-4, -5]]
B = [[0], [1]]
C = np.eye(2)
D = np.zeros([2,1])
P = ct.ss(A, B, C, D)

Pd = ct.c2d(P, 0.1, name='plant')
sys = ct.ss2io(Pd)


## cost function
xf = [0, 0]  # reference state
uf = 0       # reference input

Q = np.diag([100, 1]) # weight on state
R = 0.1               # weight on input

cost = obc.quadratic_cost(sys, Q, R, x0=xf, u0=uf)  # cost function

xmin, xmax = -1.5, 1.5
umin, umax = -8, 8
constraints = [obc.input_range_constraint(sys, [umin], [umax]), obc.state_range_constraint(sys, [
    xmin, xmin], [xmax, xmax])]

```

```
29
30 ## design of MPC
31 N = 5  # predictive horizon
32 ctrl = obc.create_mpc_iosystem(sys, np.arange(0, N)*0.1, cost, constraints, name='controller')
33
34
35 ## definition of closed loop
36 loop = ct.interconnect(
37     [sys, ctrl],   # connect plant and controller
38     connections=[  # signal connection
39       ['plant.u[0]', 'controller.u[0]'],
40       ['controller.x[0]', 'plant.y[0]'],
41       ['controller.x[1]', 'plant.y[1]']
42     ],
43     outlist=['plant.y[0]', 'plant.y[1]', 'controller.u[0]']
44 )
45
46
47 ## simulation
48 Td = np.arange(0, 2.1, 0.1)  # simulation time
49 X0 = [1, 0, 0, 0, 0, 0, 0]   # initial state ([1, 0]) + initial value of predictive horizon
50 resp = ct.input_output_response(loop, Td, 0, X0)
51
52 fig, ax = plt.subplots(1, 2, figsize=(15,5))
53 ax[0].plot(resp.time, resp.outputs[0], label='x1')  # state
54 ax[0].plot(resp.time, resp.outputs[1], label='x2')  # state
55 ax[1].plot(resp.time, resp.outputs[2])              # control input
56
57 ax[0].hlines(xmin, 0, 2, colors='black', linestyle='dashed')
58 ax[0].hlines(xmax, 0, 2, colors='black', linestyle='dashed')
59 ax[1].hlines(umin, 0, 2, colors='black', linestyle='dashed')
60 ax[1].hlines(umax, 0, 2, colors='black', linestyle='dashed')
61
62 plot_set(ax[0], 't', 'x', 'best')
63 plot_set(ax[1], 't', 'u')
64
65 #plt.savefig('MPC.png', dpi=300)
```

Listing 6.9: Model predictive control

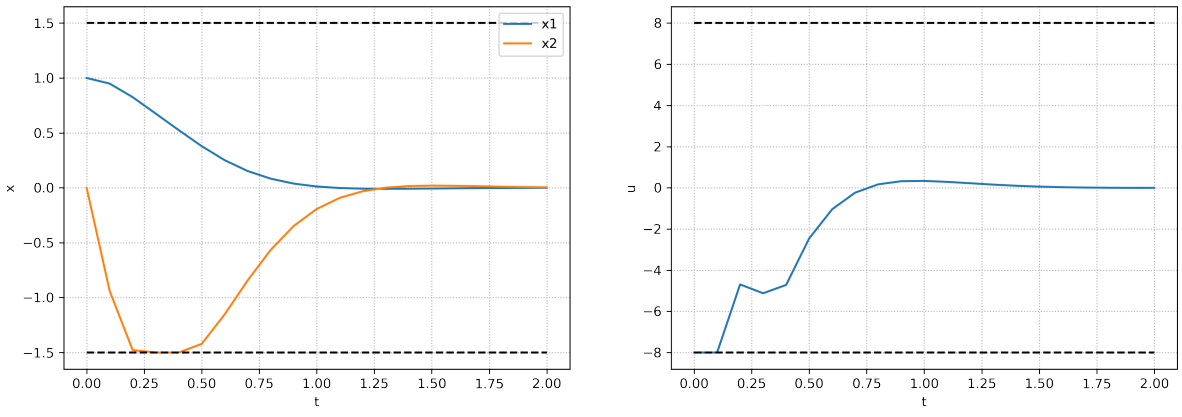

Figure 6.23: Model predictive control

6.4 Digital implementation

6.4.1 Regarding discretization

To implement the designed controller in a digital system, it is necessary to convert the controller, represented by continuous-time differential equations, into discrete-time difference equations (**discretization**).

Since the output from the plant is a continuous-time signal, it is sampled at regular intervals by an ideal sampler. The control input is determined by the discretized controller, but the discrete-time signal cannot be directly used as the input to the plant, so it is converted into a continuous-time signal through a hold circuit.

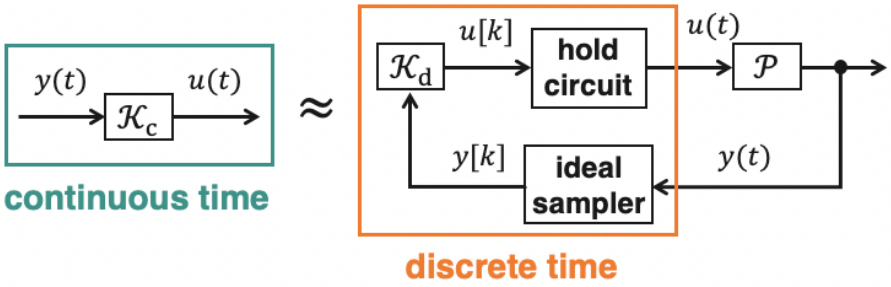

Figure 6.24: Image of discretization

The following considers methods for converting a continuous-time system

$$\mathcal{K}_{\mathrm{c}} : \begin{cases} \dot{\boldsymbol{x}}(t) = \boldsymbol{A}_{\mathrm{c}}\boldsymbol{x}(t) + \boldsymbol{B}_{\mathrm{c}}y(t) \\ u(t) = \boldsymbol{C}_{\mathrm{c}}\boldsymbol{x}(t) + \boldsymbol{D}_{\mathrm{c}}y(t) \end{cases} \tag{6.82}$$

into a discrete-time system

$$\mathcal{K}_{\mathrm{d}} : \begin{cases} \boldsymbol{x}[k+1] = \boldsymbol{A}_{\mathrm{d}}x[k] + \boldsymbol{B}_{\mathrm{d}}y[k] \\ u[k] = \boldsymbol{C}_{\mathrm{d}}x[k] + \boldsymbol{D}_{\mathrm{d}}y[k] \end{cases} \tag{6.83}$$

specifically focusing on discretization using zero-order hold and discretization using bilinear transformation.

6.4.1.1 Discretization using zero-order hold

In discretization using zero-order hold, the hold circuit, continuous-time system, and sampler are considered as a unified system, and their behavior is represented as a discrete-time system.

When the sample time is t_s, the relationship between the parameters of the continuous-time system $\mathcal{K}_\mathrm{c}$ and the discrete-time system $\mathcal{K}_\mathrm{d}$ is given by

$$\boldsymbol{A}_\mathrm{d} = e^{\boldsymbol{A}_\mathrm{c} t_s},\ \boldsymbol{B}_\mathrm{d} = \int_0^{t_s} e^{\boldsymbol{A}_\mathrm{c} t} \mathrm{d}t \boldsymbol{B}_\mathrm{c},\ \boldsymbol{C}_\mathrm{d} = \boldsymbol{C}_\mathrm{c},\ \boldsymbol{D}_\mathrm{d} = \boldsymbol{D}_\mathrm{c}. \tag{6.84}$$

The step response of a discrete-time system obtained by discretization using zero-order hold matches the step response of the original continuous-time system at the sample points, and thus it is also called a step-invariant transformation.

6.4.1.2 Discretization using bilinear transformation

In discretization using bilinear transformation (Tustin ' s transformation), the behavior of the continuous-time system is approximately represented by a discrete-time system.

When the sample time is t_s, the relationship between the parameters of the continuous-time system$\mathcal{K}_\mathrm{c}$ and the discrete-time system $\mathcal{K}_\mathrm{d}$ is

$$\begin{aligned} &\boldsymbol{A}_\mathrm{d} = \left(\boldsymbol{I} + \frac{t_s}{2}\boldsymbol{A}_\mathrm{c}\right)\left(\boldsymbol{I} - \frac{t_s}{2}\boldsymbol{A}_\mathrm{c}\right)^{-1},\ \boldsymbol{B}_\mathrm{d} = \frac{t_s}{2}\left(\boldsymbol{I} - \frac{t_s}{2}\boldsymbol{A}_\mathrm{c}\right)^{-1}\boldsymbol{B}_\mathrm{c}, \\ &\boldsymbol{C}_\mathrm{d} = \boldsymbol{C}_\mathrm{c}(\boldsymbol{A}_\mathrm{d} + \boldsymbol{I}),\ \boldsymbol{D}_\mathrm{d} = \boldsymbol{B}_\mathrm{d}\boldsymbol{C}_\mathrm{c} + \boldsymbol{D}_\mathrm{c} \end{aligned} \tag{6.85}$$

In discretization using bilinear transformation, the stability and phase characteristics of the system are preserved. Additionally, the frequency response matches at $\omega = 0$. This indicates that $\mathcal{K}_\mathrm{c}(0) = \mathcal{K}_\mathrm{d}(1)$.

6.4.2 Methods for discretization in Python

In Python, discretization can be performed using the `c2d` function, such as `sysd = c2d(sys, ts, method)`. You can specify the discretization method by setting the `method` argument, for example, `method='zoh'`.

Based on this, executing the following code will yield Figures 6.25 and 6.26.

```
# Continuous to Discrete Time System
from control.matlab import tf, c2d, step, lsim

P = tf([0, 1], [0.5, 1])
print(P)

ts = 0.2  # sampling rate

Pd1 = c2d(P, ts, method='zoh')  # 0th order hold
print('Discrete Time Sysyem (zoh)', Pd1)

Pd2 = c2d(P, ts, method='tustin')  # 1st order hold
print('Discrete Time Sysyem (tustin)', Pd2)

## step responce
fig, ax = plt.subplots(1, 2, figsize=(15,5))

Tc = np.arange(0, 3, 0.01)
y, t = step(P, Tc)  # continuous time system
ax[0].plot(t, y)
```

```
ax[1].plot(t, y)


T = np.arange(0, 3, ts)
y, t = step(Pd1, T)  # discrete time system (0th order hold)
ax[0].plot(t, y, ls='', marker='o', label='zoh', c='#ff7f0e')

y, t = step(Pd2, T)  # discrete time system (1st order hold)
ax[1].plot(t, y, ls='', marker='o', label='tustin', c='#2ca02c')

ax[0].legend()
ax[1].legend()

#plt.savefig('TimeResponse_of_DiscreteTimeSystem_step.png', dpi=300)


## apply input
fig, ax = plt.subplots(1, 2, figsize=(15,5))

Tc = np.arange(0, 3, 0.01)
Uc = 0.5 * np.sin(6*Tc) + 0.5 * np.cos(8*Tc)
y, t, x0 = lsim(P, Uc, Tc)  # continuous time system
ax[0].plot(t, y)
ax[1].plot(t, y)

T = np.arange(0, 3, ts)
U = 0.5 * np.sin(6*T) + 0.5 * np.cos(8*T)
y, t, x0 = lsim(Pd1, U, T)  # discrete time system (0th order hold)
ax[0].plot(t, y, ls='', marker='o', label='zoh', c='#ff7f0e')

y, t, x0 = lsim(Pd2, U, T)  # discrete time system (1st order hold)
ax[1].plot(t, y, ls='', marker='o', label='tustin', c='#2ca02c')

ax[0].legend()
ax[1].legend()

#plt.savefig('TimeResponse_of_DiscreteTimeSystem.png', dpi=300)
```

Listing 6.10: Time response of discrete-time systems

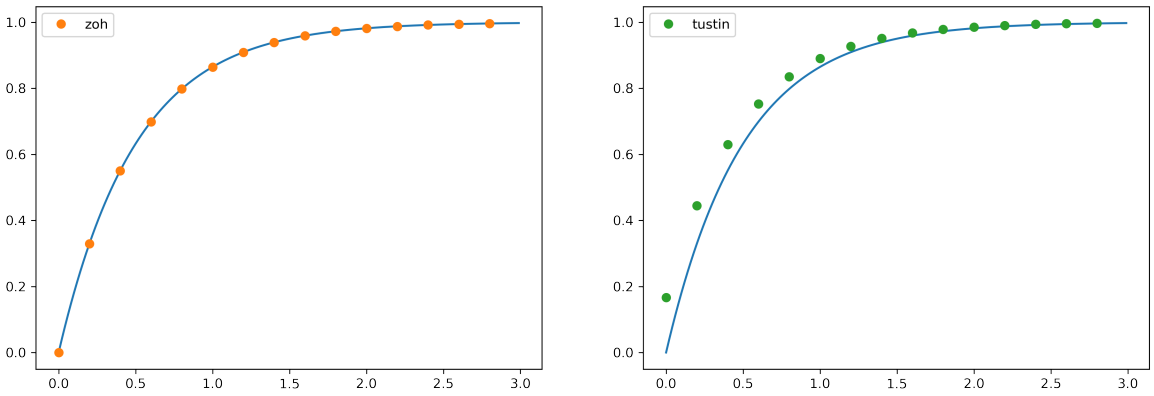

Figure 6.25: Time response of discrete-time systems (step response)

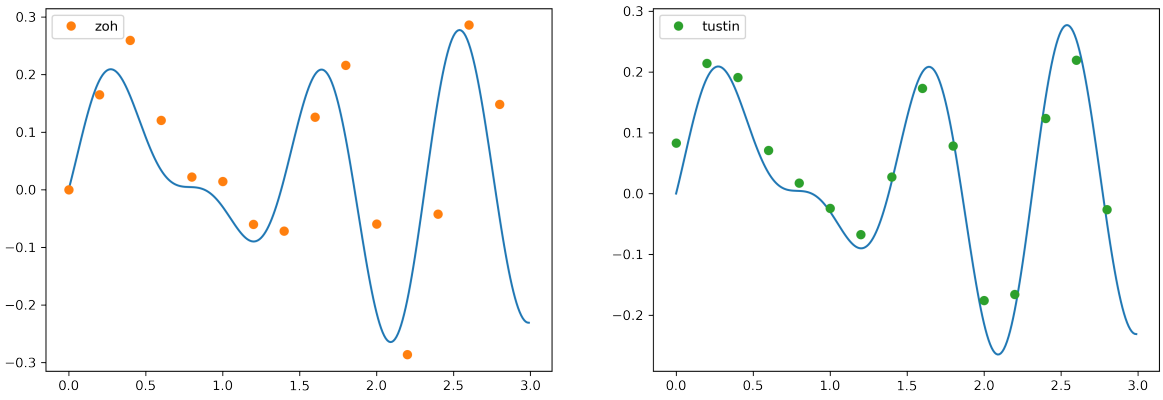

Figure 6.26: Time response of discrete-time systems

The left side represents discretization using zero-order hold, and the right side represents discretization using bilinear transformation. From Figure 6.25, it can be seen that discretization using zero-order hold better preserves the characteristics of the continuous-time system in terms of the step response. On the other hand, Figure 6.26 shows that when the input $u(t) = 0.5\sin(6t) + 0.5\cos(8t)$ is applied, discretization using bilinear transformation is closer to the response of the continuous system.

Additionally, to examine the frequency characteristics of the discretized model, executing the following code will yield Figure 6.27.

```
# Bode Plot for Discrete Time System
from control.matlab import bode, logspace, linspace, mag2db

fig, ax = plt.subplots(2, 1, figsize=(7, 7))

## continuous time system
mag, phase, w = bode(P, logspace(-2,2), plot=False)
ax[0].semilogx(w, mag2db(mag), label='continuous')
ax[1].semilogx(w, np.rad2deg(phase), label='continuous')

## discrete time system (0th order hold)
mag, phase, w = bode(Pd1, linspace(0.01, np.pi/ts-0.001, 1000), plot=False)
ax[0].semilogx(w, mag2db(mag), label='zoh')
ax[1].semilogx(w, np.rad2deg(phase), label='zoh')

## discrete time system (1st order hold)
mag, phase, w = bode(Pd2, linspace(0.01, np.pi/ts-0.001, 1000), plot=False)
ax[0].semilogx(w, mag2db(mag), label='tustin')
ax[1].semilogx(w, np.rad2deg(phase), label='tustin')

## Nyquist frequency
ax[0].axvline(np.pi/ts, lw=0.5, c='k')
ax[1].axvline(np.pi/ts, lw=0.5, c='k')

bodeplot_set(ax, 3)

#plt.savefig('BodePlot_for_DiscreteTimeSystem.png', dpi=300)
```

Listing 6.11: Frequency characteristics of discrete-time systems

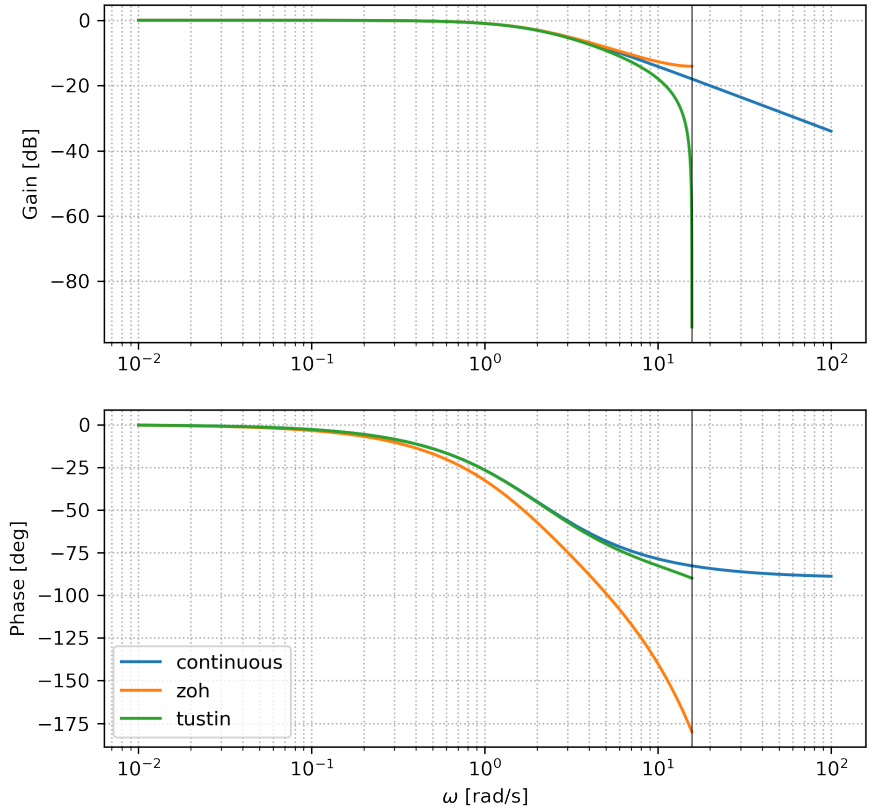

Figure 6.27: Frequency characteristics of discrete-time systems

From Figure 6.27, it can be observed that at low frequencies, the characteristics of the continuous-time and discrete-time systems are almost the same, but at high frequencies, the characteristics differ. In particular, the phase characteristics of the system discretized using zero-order hold are significantly different from those of the continuous-time system, whereas the phase characteristics of the system discretized using bilinear transformation are closer to those of the continuous-time system.

Bibliography

[1] 南祐樹『Python による制御工学入門』オーム社（2019）.

[2] 美多勉『$\mathcal{H}_\infty$ 制御』 昭晃堂（1994）.