Optimization-Based Control

Richard M. Murray
Control and Dynamical Systems
California Institute of Technology

Version v2.2c (29 Dec 2021)

Chapter One
Introduction

This chapter provides an introduction to the optimization-based framework that is used in throughout this supplement and also introduces the Python Control Systems Library (python-control), which implements all the functionality required for material presented in this supplement.

Prerequisites. Readers should be familiar with standard concepts in control theory, including input/output modeling, feedback interconnections, and the role of feedback in allowing the design of (closed loop) dynamics and providing robustness to uncertainty.

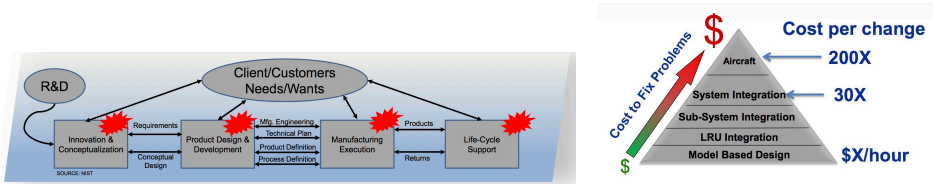
1.1 System and Control Design[1]

System design starts by developing an understanding of the system and its environment. It includes analysis of static and dynamic properties of the physical system and its sensors and actuators, bounds for safe operation, and characterization of the nature of the disturbances and the users of the system. There are a wide range of problems. Sometimes the process is given *a priori* and the task is to design a controller for a given process. In other cases the process and the controller are designed jointly. Co-design has many advantages because performance can be optimized. Sometimes it is an enabler, as was illustrated by the Wright Flyer, which was discussed in FBS2e Section 1.5. We quote from the 43rd Wilbur Wright Memorial Lecture by Charles Stark Draper [Dra55]:

> The Wright Brothers rejected the principle that aircraft should be made inherently so stable that the human pilot would only have to steer the vehicle, playing no part in stabilization. Instead they deliberately made their airplane with negative stability and depended on the human pilot to operate the movable surface controls so that the flying system—pilot and machine—would be stable. This resulted in increased maneuverability and controllability.

In design of modern control systems, the engineering workflow is broken down into phases to manage the complexity of overall system. Early phases of the design create a basic architecture for the system, with interaction

[1] The material in this section is drawn from FBS2e, Chapter 15 (online version).



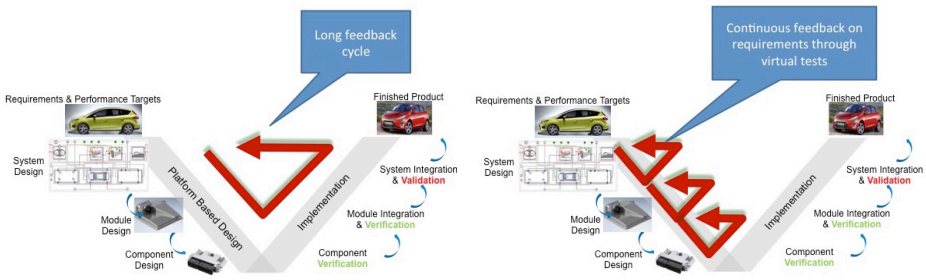
(a) Design Flow (b) Costs

Figure 1.1: Engineering design process. A typical design cycle is shown in (a) and (b) illustrates the costs of correcting faults or making design changes at different stages in the design process.

between subsystems that provide the main functionality of the system. For an aircraft, those subsystems might be the airframe (fuselage and wings), propulsion system (engines), the hydraulic system, the electric power system, the flight control system, and other critical components. For a cell phone, the subsystems might be the chassis, the display (including touch interface), the communications subsystem (5G, wifi), the audio system (speakers and microphone), the power system (batteries and charging), among others. In each case, a high level architecture is required that describes what the subsystems are responsible for and how they will interact. Each subsystem is itself comprised of a variety of components, which also have their own specifications and interfaces. The engineering workflow typically operates by carrying out a succession of refinements of the design from one level of abstraction down to the next, and then assembly the components of the design from the components to the subsystems to the overall system, with validation and testing at each stage.

Figure 1.1a shows a typical design process for a modern engineering system. Design is broken into phases such as research and development (R&D), conceptualization, development, manufacturing, and life-cycle support. One of the features of engineering complex systems is that it can be very costly to make corrections late in the product development cycle, since a substantial amount of engineering effort has already been carried out and may need to be redone. These costs are illustrated in Figure 1.1b. Notice the significant value in correcting faults early. Design of complex systems is a major effort where many people and groups are involved.

A variety to methods have been developed for efficient design. The so-called *design V*, shown in Figure 1.2a, dates back to NASA's Apollo program [SC92] and is a common design pattern for both hardware and software. The left leg of the V illustrates the design process starting with requirements and ending with system, module, and component designs. The right leg of the V represents the implementation, starting with the components and ending with the finished process and its validation. There are many substeps in the design, they include functional requirements, architecture generation



(a) Classical Design V (b) Design V for Model Based Design

Figure 1.2: Design methodologies for complex systems. (a) The traditional design V. The left side of the V represents the decomposition of requirements and creation of system specifications. The right side represents the activities in implementation, including validation (building the right thing) and verification (building it right). Notice that validation and verification are performed late in the design process when all hardware is available. (b) A model-based design process where virtual validation is be made at many stages in the design process, shortening the feedback for validation.

and exploration, analysis, and optimization. Notice that validation is made only on the finished product.

The cost of faults or changes increase dramatically if they are discovered late in the development process or even worse when systems are in operation, as illustrated in Figure 1.1b. Model-based systems engineering can reduce the costs because models allow partial validation using models as virtual hardware at many steps in the development process, as illustrated in Figure 1.2b. When hardware and subsystems are built they can replace the corresponding models using hardware-in-the-loop simulations.

To perform verification efficiently it is necessary that requirements are expressed mathematically and checked automatically using models of the system and its environment, along with a variety of tools for analysis. *Regression analysis* can be used to ensure that changes in one part of a system do not create unexpected errors in other parts of the system. Efficient regression analysis requires robust system-level models and good scripting software that allows analyses to be performed automatically over many operating conditions with little to no human intervention. System-level models are also useful for *root cause analysis* by allowing errors to be reproduced, which is helpful to ensure that the real cause has been found.

There are strong interactions between the models and the analysis tools that are used; therefore, the models must satisfy the requirements of the algorithms for analysis and design. For example, when using Newton's method for solution of nonlinear equations and optimization, the models must be continuous and have continuous first (and sometimes second) derivatives. This property, which is called *smoothness*, is essential for algorithms to work

well. Lack of smoothness can be due to many factors: if-then-else statements, an actuator that saturates, or by careless modeling of fluid systems with reversing flows. Having tools that check if a given system model has functions with continuous first and second derivatives is valuable.

An alternative to the use of the traditional design V is the *agile development* model, which has been driven by software developers for products with short time to market, where requirements change and close interaction with customers is required. The method is characterized by the *Agile Manifesto* [BBvB$^+$01], which values individuals and interactions over processes and tools; working software over comprehensive documentation; customer collaboration over contract negotiation; and responding to change over following a plan. When choosing a design methodology it is also important to keep in mind that products involving hardware are more difficult to change than software.

Control system design is a subpart of system design that includes many activities, starting with requirements and system modeling and ending with implementation, testing, commissioning, operation, and upgrading. In between are the important steps of detailed modeling, architecture selection, analysis, design, and simulation. The V-model used in an iterative fashion is well suited to control design, particular if it is supported by a tool chain that admits a combination of modeling, control design, and simulation. Testing is done iteratively at every step of the design using models of different granularity as virtual systems. Hardware in the loop simulations are also used when they are available.

Today most control systems are implemented using computer control. Implementation then involves selection of hardware for signal conversion, communication, and computing. A block diagram of a system with computer control is shown in 1.3. The overall system consists of sensors, actuators, analog-to-digital and digital-to-analog converters, and computing elements. The filter before the A/D converter is necessary to ensure that high-frequency disturbances do not appear as low-frequency disturbances after sampling because of aliasing. The operations of the system are synchronized by a clock.

Real-time operating systems that coordinate sensing, actuation, and computing have to be selected, and algorithms that implement the control laws must be generated. The sampling period and the anti-alias filter must be chosen carefully. Since a computer can only do basic arithmetic, the control algorithms have to be represented as difference equations. They can be obtained by approximating differential equations, as was illustrated in FBS2e Section 8.5, but there are also design methods that automatically give controllers in the form of difference equations. Code can be generated automatically. It must also be ensured that computational delays and synchronization of algorithms do not create problems.

When the design is implemented and tested the system must be com-

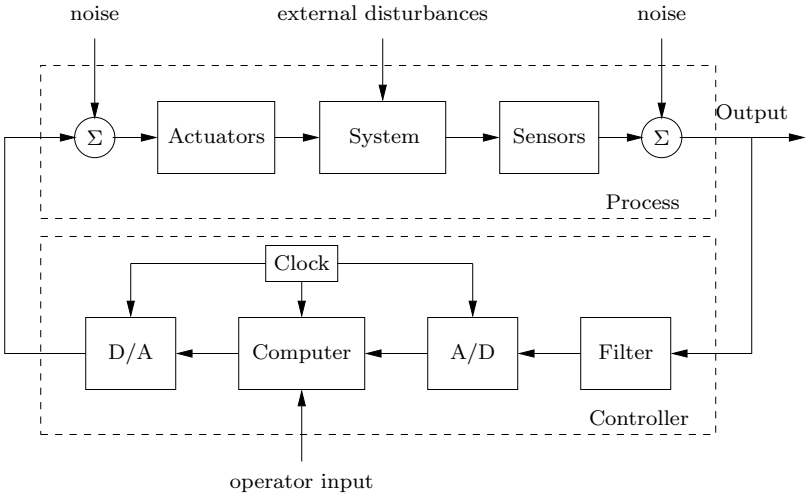


Figure 1.3: Schematic diagram of a control system with sensors, actuators, communications, computer, and interfaces.

missioned. This step may involve adjustment of controller parameters, and automatic tuning (discussed in FBS2e, Section 11.3) can be very beneficial at this stage. During operation it is important to monitor the behavior of the system to ensure that specifications are still satisfied. It may be necessary to upgrade the system when it has been operating. Specifications may also be modified due to operational experiences.

1.2 The Control System "Standard Model"

Feedback control appears across an enormous variety of applications and in various forms. Despite the wide range of applications and implementations, there is a common design pattern for most modern feedback control systems, as illustrated in Figure 1.4. The starting point for the control system is the process that we wish to control, which we model as an input/output system. The inputs to the process consist of both those inputs that the controller

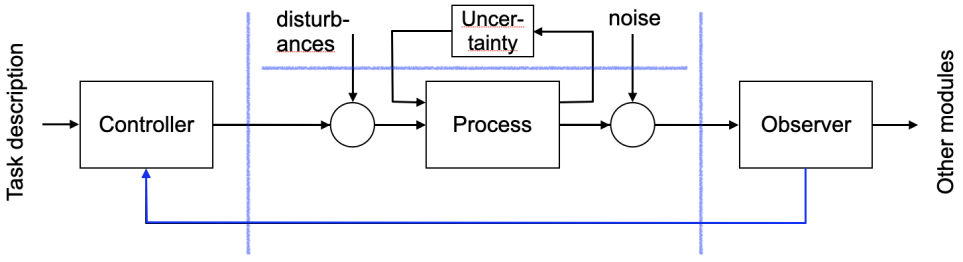


Figure 1.4: Control system standard model.

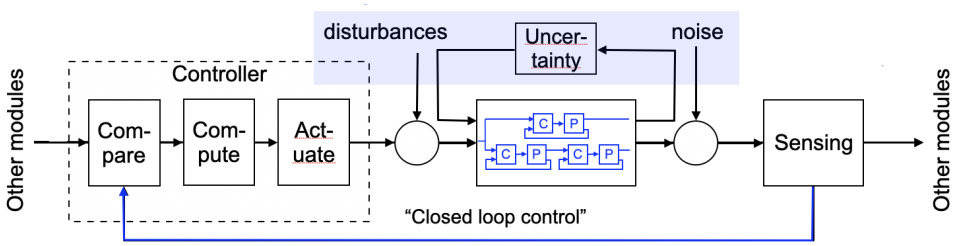


Figure 1.5: Nested control systems.

specifies as well as inputs that may come from external disturbances and uncertainties in the model. The outputs from the process, possibly corrupted by noise, are processed by an "observer", whose function is to estimate the state of the underlying process from the measured (and sometimes noisy) signals. Finally, the controller is responsible for taking some level of description of the task to be accomplished and generating those inputs that will cause the process to carry out the desired operation.

While this general diagram is likely familiar to anyone with experience in classical control theory (e.g., PID control of a linear dynamical system represented by its transfer function), it is perhaps useful to point out that this basic pattern is present in many systems that use different representations of the dynamics and uncertainty. For example, the process that we are controlling may be an infrastructure management system in which there are requests for resources that must be managed and balanced. Uber is one example of such a "control system", with the process consisting of the dynamics of individually driven Uber vehicles that can be dispatched based on observations of riders requesting transport. The models used for this type of process are likely to be based on stochastic queuing system models, but the basic pattern is still there. Other examples of control systems that match this pattern range from aircraft, to the supply chain, to your cell phone.

Another common feature of control systems is that the process itself may be a control system, so that we have a nested set of controllers, as illustrated in Figure 1.5. Note that in this view the inputs and outputs of the overall system are themselves coming from other modules. We have also expanded the view of the controller to include its three key functions: comparing the current and desired state of the process, computing the possible actions that can be taken to bring these closer, and then "actuating" the process being controlled via some appropriate command. This type of nested system could emerge, for example, if Uber vehicles were autonomous vehicles, where there is a control system in place for each car (which is itself a nested set of control systems, as we shall discuss in the next section).

Based on these observations, we define key elements of the "standard model" of a control system as follows:

Process. The process represents that system that we wish to control. The

inputs to the system include controller and environmental inputs, and the outputs to the system are the measurable variables.

Task Description. The task description is an input to the controller that describes the "task" to be performed. Depending on the type of system that is being controlled, the task description could be anything from a simple signal that should be tracked to a description of a complex task with cost functions and constraints.

Observer. The observer takes the outputs of the process and performs calculations to estimate the underlying state of the process and/or the environment. In some cases the observer may also make predictions about the future state of the system or the environment.

Controller. The controller is responsible for determining what inputs should be applied to the system in order to carry out the desired task. It takes as inputs the description of the task as well as the output of the observer (often the estimated state and/or the state of the environment).

Disturbances. Disturbances represent exogenous inputs to the process dynamics that are not dependent on the dynamics of system or the controller. In Figure 1.4 the disturbances are modeled as being added to the inputs, but more general disturbances are also possible.

Noise. Noise represents exogenous inputs to the observer that corrupt the measurements of the system outputs. In Figure 1.4 the noise is modeled as being added to the inputs, but more general noise signals are also possible.

Uncertainty. This block represents uncertainty in the dynamics of the process or "reactive" uncertainty in the environment in which the process operates. We represent the environment as a feedback interconnection with the process to reflect the fact that the unmodeled dynamics and or environmental dynamics may depend on the state of the system.

Other Modules. Control systems are often connected with other modules of the overall system, in either a distributed, nested, or layered fashion. The type of interconnection can be through the controllers of the other modules, through physical interconnections, or both.

1.3 Layered Control Systems

A related view of a modern control system is as a "layered" control system in which we reason about the system at different layers of abstraction, as shown in Figure 1.6. To a large extent this is just a different view of the "nested" representation of a control system in Figure 1.5, but here we are more explicit about the different representations of the system. In this figure, the control system is described by four layers of abstraction, separated by horizontal lines.

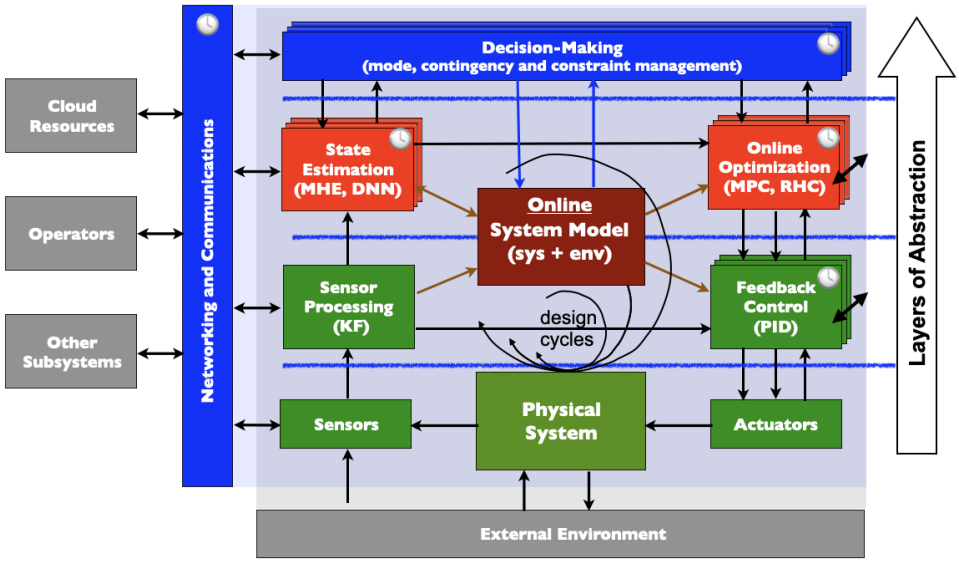


Figure 1.6: Layered control systems. The system consists of four layers: the physical layer (lowest), the feedback regulation layer (green), the trajectory generation layer (red), and the decision-making layer (blue). Networking and communications allows information to be transferred between the layers as well as with outside resources (left). The entire system interacts with the external environment, represented at the bottom of the figure.

The lowest layer is the physical layer, representing the physical process being control as well as the sensors and actuators. This layer is often described in terms of input/output dynamics that can be describe how the system evolves over time. The simplest (and one of the most common) representations is an ordinary differential equation model of the form

$$\frac{dx}{dt} = f(x, u, d), \qquad y = h(x, n),$$

where $x \in \mathbb{R}^n$ represents the state of the system, $u \in \mathbb{R}^m$ represents the inputs that can be commanded by the controller, $d \in \mathcal{D}$ represents disturbance signals that come from the external environment, $y \in \mathbb{R}^p$ represents the measured outputs o the system, and $n \in \mathcal{N}$ represents process or sensor noise. The design of the physical system will normally attempt to make sure that the region of the state space in which the system is able to operate (called the *operating envelope*) satisfies the needs of the user or customer. For an aircraft, for example, this might consist of specifications on the altitude, speed, and maneuverability of the physical system.

The next layer is the *feedback regulation* layer (sometimes also called the "inner loop") in which we we use feedback control to track a reference trajectory. This layer commonly represents the abstractions used in classical control theory, where we have a reference input r that we wish to track while

at the same time attenuating disturbances d and avoiding amplification of process or sensor noise n. The system and controller at this level might be represented by transfer function $P(s)$ and $C(s)$ and our specification might be on various input/output transfer functions such as the Gang of Four (see FBS2e. Section 12.1):

$$S = \frac{1}{1+PC} \quad \text{sensitivity function} \qquad PS = \frac{P}{1+PC} \quad \text{load (or input) sensitivity function}$$

$$T = \frac{PC}{1+PC} \quad \text{complementary sensitivity function} \qquad CS = \frac{C}{1+PC} \quad \text{noise (or output) sensitivity function}$$

A typical specification for design at this layer of abstraction might be a weighted sensitivity function, such as

$$\| |W_1 S| + |W_2 T| \|_\infty < 1.$$

The feedback regulation phase of design will also often compensate for the effects of unmodeled dynamics, traditionally done by the specification of gain, phase, and stability margins.

This layer also carries out some level of sensor processing to try to minimize the effects of noise. In classical control design the sensor processing is often integrated into the controller design process (for example by imposing some amount of high frequency rolloff), but many modern control systems will use Kalman filtering to process signals and also perform sensor fusion. Kalman filtering is described in more detail in Chapter 6.

Continuing up our abstraction hierarchy, the next layer of abstraction is the *trajectory generation* layer (sometimes also called the "outer loop"). In this layer we attempt to find trajectories for the system that satisfy a commanded task, such as moving the system from one operating point to another while satisfying constraints on the inputs and states. At this layer, we assume that the effects of noise, disturbances, and unmodeled dynamics have been taken care of at lower levels but nonlinearities and constraints are explicitly accounted for. Thus we might use a model of the form

$$\frac{dx}{dt} = f(x,u), \qquad g(x,u) \leq 0,$$

where $g : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}^k$ is a nonlinear function describing constraints on inputs and states. Our control objective might be to optimize according to a cost function of the form

$$J(x,u) = \int_0^T L(x,u)\,dt + V\big(x(T)\big),$$

where $L(x,u)$ represents the integrated cost along the trajectory and $V(x)$ represents the terminal cost (e.g., it should be small near the final operating point that we seek to reach). We will study this problem and its variants in

Chapters 2 and 3.

As in the case of the feedback regulation layer, the trajectory generation layer also has a "observer" function, labeled as "state estimation" in Figure 1.6. The details of this observer depend on the application, but could represent additional sensor processing that is required for trajectory generation or sensing of the environment for the purpose of satisfying specifications relative to that environment. The latter case is particularly common in applications such as autonomous vehicles, where the state estimation often includes perception and prediction tasks that are used to identify other agents in the environment, their type (e.g., pedestrian, bicycle, car, truck), and their predicted trajectory. This information will be used by the trajectory generation algorithm to avoid collisions or to maintain the proper position relative to those other agents. The applications of Kalman filtering and sensor fusion to problems at this layer are considered in Chapters 6 and 7.

The highest layer of abstraction in Figure 1.6 is the *decision-making* layer (sometimes also called the "supervisory" layer). At this layer we often reason over discrete events and logical relations. For example, we may care about discrete modes of behavior, which could correspond to different phases of operation (takeoff, cruise, landing) or different environment assumptions (highway driving, city streets, parking lot). This layer can also be used to reason over discrete (as opposed to continuous) decisions that we must make (stop, go, turn left, turn right). This final layer is not explicitly part of the material covered in this book; a brief discussion of the design problem at this layer can be found in FBS2e, Section 15.3.

In a full system design, the three control layers that we depict here may in fact include additional layers within them, or be divided up slightly differently. Similarly, the physical layer may consist of system that themselves have internal control loops running, potentially at multiple layers of abstraction. And the system may be networked to other agents and information systems that provide information and constraints on system operation. Thus, our system is a combination of nested, layered, distributed systems, all operating together.

Another important element of modern control systems is their distributed and interconnected nature. Much of this is already presented in the layered control structure described above, but there can also be "external" interactions. On the left side of Figure 1.6 are a set of blocks that represent some of the elements that can connected through networked information channels. These can include cloud resources (such as computing or databases), operators (humans or automated), and interactions with other systems and subsystems. The increased capability and capacity of networking and communications is one of the drivers of complexity in modern control systems and has created both new opportunities and new challenges.

Finally, we note the effect of the environment, represented in Figure 1.6

as a block at the bottom of the diagram. This block represents many things, including noise, disturbances, unmodeled dynamics of the process, and the dynamics of other systems with which our system is interacting. It is the uncertainty represented in this catchall block that is driving the need for feedback control, and the impact of these different types of uncertainty appears in each level of our controller design.

1.4 The Python Control Systems Library (python-control)[2]

The Python Control Systems Library (python-control) is a Python package that implements basic operations for analysis and design of feedback control systems. The package was created in 2009, shortly after the publication of the first edition of *Feedback Systems*. The initial goal of the project was to implement the operations needed to carry out all the examples in the book. A primary motivation for the creation of the python-control library was the need for open-source control design software built on the Python general-purpose programming language. The "scientific stack" of NumPy, SciPy, and Matplotlib provide fast and efficient array operations, linear algebra and other numerical functions, and plotting capabilities to Python users. Python-control has benefited from this foundation, using, e.g., optimization routines from SciPy in its optimal control methods, and Matplotlib for Bode diagrams.

The python-control package provides the functionality required to implement all of the techniques described in this supplement. This section provides a brief overview of the python-control package, with the intent of indicating the calling structure of the code and including a few simple examples. More detailed examples are given in subsequent chapters, and more detailed documentation is available at `http://python-control.org`.

Package Structure and Basic Functionality

The python-control package implements an inheritance hierarchy of dynamical system objects. For the most part, when two systems are combined in some way through a mathematical operation, one will be promoted to the type that is the highest of the two. Arranged in order from most to least general, they are:

- `InputOutputSystem`: Input/output system that may be nonlinear and time-varying
  - `InterconnectedSystem`: Interconnected I/O system consisting of multiple subsystems
  - `NonlinearIOSystem`: Nonlinear I/O system

[2]The material in this section is drawn from [FGM+21].

Table 1.1: Sample functions available in the python-control package.

Frequency domain analysis:

`sys(x)`	Evaluate frequency response of an LTI system at complex frequenc(ies) x
`sys.frequency_response()`	Evaluate frequency response of an LTI system at real angular frequenc(ies) omega
`stability_margins()`	Calculate stability margins and associated crossover frequencies
`bode_plot()`	Bode plot for a system
`nyquist_plot()`	Nyquist plot for a system
`gangof4_plot()`	Plot the "Gang of 4" transfer functions for a system
`nichols_plot()`	Nichols plot for a system

Time domain analysis:

`forced_response()`	Simulated response of a linear system to a general input
`impulse_response()`	Compute the impulse response for a linear system
`initial_response()`	Initial condition response of a linear system
`step_response()`	Compute the step response for a linear system
`step_info()`	Compute step response characteristics
`phase_plot()`	Phase plot for 2D dynamical systems

Other analysis functions and methods:

`sys.dcgain()`	Return the zero-frequency (or DC) gain of an LTI system
`sys.pole()`	Compute poles of an LTI system
`sys.zero()`	Compute zeros of an LTI system
`sys.damp()`	Compute natural frequency and damping ratio of LTI system poles
`pzmap()`	Plot a pole/zero map for a linear system
`root_locus()`	Root locus plot
`sisotool()`	Sisotool style collection of plots inspired by MATLAB

Synthesis tools:

`acker()`	Pole placement using the Ackermann method
`h2syn()`	H_2 control synthesis for plant P
`hinfsyn()`	H_∞ control synthesis for plant P
`lqr()`	Linear quadratic regulator design
`lqe()`	Linear quadratic estimator design (Kalman filter) for continuous-time systems
`mixsyn()`	Mixed-sensitivity H-infinity synthesis
`place()`	Place closed-loop poles

  - `LinearICSystem`: Linear interconnected I/O systems
  - `LinearIOSystem`: Linear I/O system

- `LTI`: Linear, time-invariant system

  - `FrequencyResponseData`: Frequency response data systems
  - `StateSpace`: State space systems
  - `TransferFunction`: Transfer functions

Each can be either discrete-time, that is, $x(k+1) = f(x(k), u(k))$; $y(k) = g(x(k), u(k))$ or continuous time, that is, $\dot{x} = f(x, u)$; $y = g(x, u)$. A discrete-time system is created by specifying a nonzero 'timebase' dt when the system is constructed:

- $dt = 0$: continuous time system (default)
- $dt > 0$: discrete time system with sampling period dt
- $dt =$ `True`: discrete time with unspecified sampling period
- $dt =$ `None`: no timebase specified

Linear, time-invariant systems can be interconnected using mathematical operations `+`, `-`, `*`, and `/`, as well as the domain-specific functions `feedback`, `parallel` (`+`), and `series` (`*`). Some important functions for `LTI` systems and their descriptions are given in Table 1.1. Other categories of tools that are available include model simplification and reduction tools, matrix computations (Lyapunov and Riccati equations), and a variety of system creation, interconnection and conversion tools. A MATLAB compatibility layer is provided that has functions and calling conventions that are equivalent to their MATLAB counterparts, e.g. `tf, ss, step, impulse, bode, margin, nyquist` and so on. A complete list is available at `http://python-control.org`.

Linear Systems Example

To illustrate the use of the package, we present an example of the design of an inner/outer loop control architecture for the planar vertical takeoff and landing (PVTOL) example in FBS2e, Example 12.9. A slightly different version of this example is available in the python-control GitHub repository.

We begin by initializing the Python environment with the packages that we will use in the example:

```
# pvtol-nested.py - inner/outer design for
#   vectored thrust aircraft
# RMM, 5 Sep 2009 (updated 11 May 2021)
#
# This file works through a control design and
# analysis for the planar vertical takeoff and
# landing (PVTOL) aircraft in Astrom and Murray.

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

We next define the system that we plan to control:

```
# System parameters
m = 4               # mass of aircraft
J = 0.0475          # inertia around pitch axis
r = 0.25            # distance to center of force
g = 9.8             # gravitational constant
c = 0.05            # damping factor (estimated)

# Transfer functions for dynamics
Pi = ct.tf([r], [J, 0, 0])  # inner loop (roll)
Po = ct.tf([1], [m, c, 0])  # outer loop (posn)
```

The control design is performed by using a lead compensator to control the inner loop (roll axis):

```
# Inner loop control design
#
# Controller for the pitch dynamics: the goal is
# to have a fast response so that we can use this
# as a simplified process for the lateral dynamics

# Design a simple lead controller for the system
k_i, a_i, b_i = 200, 2, 50
Ci = k_i * ct.tf([1, a_i], [1, b_i])
Li = Pi * Ci
```

We can now analyze the results by plotting the frequency response as well as the Gang of 4:

```
# Loop transfer function Bode plot, with margins
plt.figure(); ct.bode_plot(Li, margins=True)
plt.savefig('pvtol-inner-ltf.pdf')

# Make sure inner loop specification is met
plt.figure(); ct.gangof4_plot(Pi, Ci)
plt.savefig('pvtol-gangof4.pdf')
```

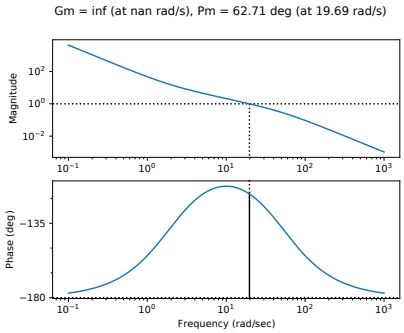
Figures 1.7a and b show the outputs from these commands.

The outer loop (lateral position) is designed using a second lead compensator, using the roll angle as the input:

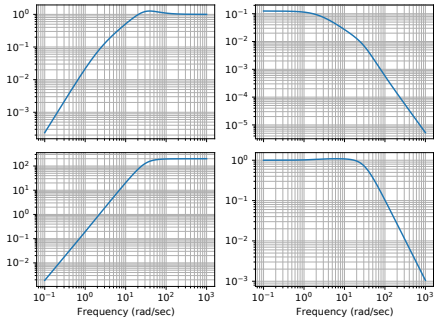
```
# Design lateral control system (lead compensator)
a_o, b_o, k_o = 0.3, 10, 2
Co = -k_o * ct.tf([1, a_o], [1, b_o])
Lo = -m * g * Po * Co

# Compute real outer-loop loop transfer function
L = Co * Hi * Po
```

We can analyze the results using Bode plots, Nyquist plots and time domain simulations:

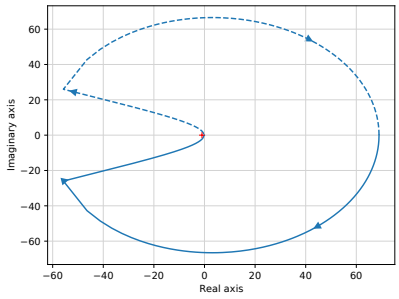
```
# Compute stability margins
gm, pm, wgc, wpc = ct.margin(L)
```



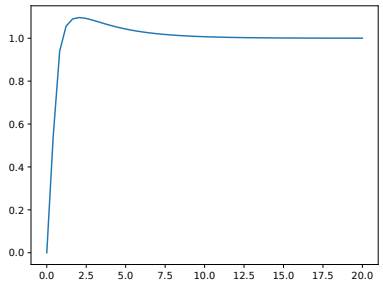
(a) Inner loop, with margins



(b) Gang of 4 for inner loop



(c) Nyquist plot for full system



(d) Step response for full system

Figure 1.7: Sample outputs for PVTOL example.

```
# Check to make sure that the specification is met
plt.figure(); ct.gangof4_plot(-m * g * Po, Co)

# Nyquist plot for complete design
plt.figure(); ct.nyquist_plot(L)
plt.savefig('pvtol-nyquist.pdf')

# Step response
t, y = ct.step_response(T, np.linspace(0, 20))
plt.figure(); plt.plot(t, y)
plt.savefig('pvtol-step.pdf')
```

Figures 1.7c and d show the outputs from the `nyquist_plot` and `step_response` commands (note that the `step_response` command only computes the response, unlike MATLAB, which also plots the response).

Input/output systems

Python-control supports the notion of an input/output system in a manner that is similar to the MATLAB "S-function" implementation. Input/output

systems can be combined using standard block diagram manipulation functions (including overloaded operators), simulated to obtain input/output and initial condition responses, and linearized about an operating point to obtain a new linear system that is both an input/output and an LTI system.

An input/output system is defined as a dynamical system that has a system state as well as inputs and outputs (either inputs or states can be empty). The dynamics of the system can be in continuous or discrete time. To simulate an input/output system, the `input_output_response()` function is used:

```
t, y = input_output_response(io_sys, T, U, X0, params)
```

Here, the variable `T` is an array of times and the variable `U` is the corresponding inputs at those times. The output will be evaluated at those times, though the NumPy `interp` function can be used to interpolate inputs at a finer timescale, if desired.

An input/output system can be linearized around an equilibrium point to obtain a state space linear system. The `find_eqpt()` function can be used to obtain an equilibrium point and the `linearize()` function to linearize about that equilibrium point:

```
xeq, ueq = find_eqpt(io_sys, X0, U0)
ss_sys = linearize(io_sys, xeq, ueq)
```

The resulting `ss_sys` object is a `LinearIOSystem` object, which is both an I/O system and an LTI system, allowing it to be used for further operations available to either class.

Input/output systems can be created from state space LTI systems by using the `LinearIOSystem` class:

```
io_sys = LinearIOSystem(ss_sys)
```

Nonlinear input/output systems can be created using the NonlinearIOSystem class, which requires the definition of an update function (for the right-hand side of the differential or difference equation) and output function (computes the outputs from the state):

```
io_sys = NonlinearIOSystem(
    updfcn, outfcn, inputs=M, outputs=P, states=N)
```

More complex input/output systems can be constructed by using the `interconnect()` function, which allows a collection of input/output subsystems to be combined with internal connections between the subsystems and a set of overall system inputs and outputs that link to the subsystems:

```
steering = ct.interconnect(
    [plant, controller], name='system',
    connections=[['controller.e', '-plant.y']],
    inplist=['controller.e'], inputs='r',
    outlist=['plant.y'], outputs='y')
```

In addition to explicit interconnections, signals can also be interconnected automatically using signal names by simply omitting the `connections` parameter.

Interconnected systems can also be created using block diagram manipulations such as the `series()`, `parallel()`, and `feedback()` functions. The `InputOutputSystem` class also supports various algebraic operations such as `*` (series interconnection) and `+` (parallel interconnection).

Exercises

1.1 (Basics of python-control) Consider a control system with

$$P(s) = \frac{b}{(s+a)^2}, \qquad C(s) = \frac{k_\mathrm{p} s + k_\mathrm{i}}{s},$$

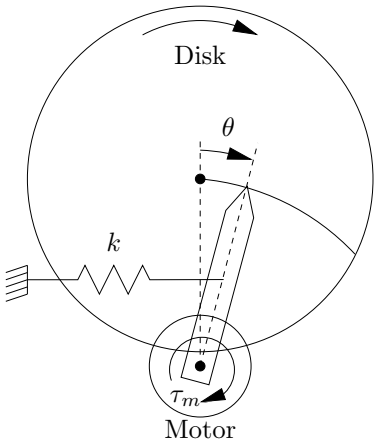
and set $a = b = 1$ and $k_\mathrm{p} = 1$, $k_\mathrm{i} = 0.1$. Using the Python Control Systems Library (python-control), do the following:

(a) Plot the step response of the closed loop system and compute the rise time, settling time, and steady state error.

(b) Plot the frequency response of the open loop system (Bode plot) and compute the gain margin, phase margins, and bandwidth of the system.

(c) Plot the Nyquist plot of the system and compute the stability margin (smallest distance to the -1 point).

1.2 (I/O systems using python-control) Consider a simple mechanism for positioning a mechanical arm and the and the associated equations of motion:



$$J\ddot{\theta} = -b\dot{\theta} - kr\sin\theta + \tau_m$$
$$\dot{\tau}_m = -a(\tau_m - u)$$

The system consists of a spring-loaded arm that is driven by a motor. The motor applies a force against the spring and pulls the tip across a rotating platter. The input to the system is the desired motor torque, u. In the diagram above, the force exerted by the spring is a nonlinear function of the head position due to the way it is attached. Take the system parameters to be

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

Starting with the template Jupyter notebook posted on the course website, create a Jupyter notebook that documents the following operations:

(a) Compute the linearization of the dynamics about the equilibrium point corresponding to $\theta_e = 15°$.

(b) Plot the step response of the linearized, open-loop system and compute the rise time and settling time.

(c) Design a state feedback controller for the system that stabilizes the system about $\theta_e = 45°$ and sets the closed loop eigenvalues to $\lambda_{1,2} = -1 \pm \sqrt{3}i$. Plot the step response for the closed loop system and compute the rise time, settling time, and steady state error.

(d) Compute the transfer function H_{yu} for the open system around the equilibrium point and sketch the frequency response of the open loop system.

(e) Design a frequency domain compensator that provides tracking with less than 10% error up to 1 rad/sec and has a phase margin of at least 45°. Demonstrate that your controller meets these requirements by showing Bode, Nyquist, and step response plots, and compute the rise time, settling time, and steady state error for the system using your controller design.

(f) Create simulations of the full nonlinear system with the linear controllers designed in parts (c) and (e) and plot the response of the system from an initial position of 0 mm at $t = 0$, to 0.4 mm at $t = 30$ ms, to 1.2 mm at $t = 90$ ms, to 0.8 mm at $t = 120$ ms.