Modeling and Simulation Lecture Notes - 8

July 2023

Design of Control Systems

- PID Controller: design and simulation
- Lead-Lag compensator design
- State feedback control
- Full-state feedback with integrator

Practical Applications of Control Systems

- Control system design for an Inverted Pendulum
- Cruise Control System in Automobiles
- Aircraft Pitch: System modeling and control

Design of Control Systems

PID Controller: Design and Simulation

A PID controller can be designed and simulated using the principles of feedback control. As previously discussed, the parameters K_p, K_i, and K_d of the controller are tuned to achieve the desired system response. We simulated a PID controller for a cruise control system and used it to adjust the car's speed to match a desired reference speed.

Lead Compensator

The lead compensator is designed to improve the transient response of a system. Before discussing the lead compensator, let's first define the relevant terms.

Transient Response

The transient response of a system refers to how the system responds to changes from one state to another. It is often quantified using measures such as rise time, peak time, settling time, and percent overshoot:

- *Rise time* is the time taken for the response to rise from 10% to 90% of its final value (for underdamped systems) or from 0% to 100% of its final value (for overdamped systems).
- *Peak time* is the time at which the response reaches its peak value for the first time.
- *Settling time* is the time taken for the response to stay within a certain percentage (typically 2
- *Percent overshoot* refers to how much the peak response is above the final value, as a percentage of the final value.

The lead compensator helps in improving these performance metrics by adding a zero and a pole to the open-loop transfer function such that the zero is closer to the imaginary axis than the pole, which results in a phase lead.

A lead compensator is essentially a filter that boosts the phase of the signal at high frequencies and has a transfer function of the form:

$$C(s) = K\frac{s/z + 1}{s/p + 1}$$

where K is the gain, z is the zero, and p is the pole of the compensator. The zero z is placed such that it is smaller than the pole p, i.e., $|z| < |p|$. We can define in the code:

```
num = [1, 2]
den = [1, 10]
compensator = control.TransferFunction(num, den)
```

This can be written in the standard form as $C(s) = (s/2 + 1)/(s/10 + 1)$. Here, the zero z is located at -2 and the pole p is located at -10. Since $|z| < |p|$, it's a lead compensator.

When the lead compensator is added, it modifies the root locus plot of the system. A root locus plot shows the possible locations of poles of a system as a system parameter (usually a gain) is varied. By adding the lead compensator (which has its pole p further to the left than its zero z), the root locus plot shifts to the left, indicating improved stability and faster transient response.

The impact of the lead compensator can be seen on the root locus plot of the system, where the compensator pulls the system poles to the left (towards more negative values). This results in a faster system response, reduced overshoot and improved stability. This impact is most noticeable when the compensator pole and zero are placed near the uncompensated system poles.

Unfortunately, with the Python Control Systems Library, generating root locus plots for systems with a lead compensator may not be straightforward. But you can visually analyze the effect of the lead compensator on the root locus plot and understand how it improves the system response.

In a real-world application, you might see lead compensators used in control systems where a faster response, less overshoot, or improved stability is desired. This could be in a wide range of industries, from automotive (for controlling engine speed) to electronics (for improving the response of a power supply), and more.

Lag Compensator

The lag compensator is designed to improve the steady-state response of a system. Let's define steady-state response.

In a lead compensator, $|z| < |p|$, while in a lag compensator $|z| > |p|$.

Steady-State Response

The steady-state response of a system refers to the behavior of the system after it has had sufficient time to react to a change in the input. It's essentially the long-term behavior of the system. For a stable system, the steady-state response to a step input is a constant value, to a ramp input is a ramp function, and so forth.

A key measure of steady-state response is the steady-state error, which is the difference between the actual output and the desired output as $t \to \infty$.

The lag compensator helps in reducing this steady-state error by adding a pole and a zero to the open-loop transfer function such that the pole is closer to the imaginary axis than the zero, which results in a phase lag.

State Feedback Control

State feedback control is a method where the controller uses the state of the system (rather than the output) to form the control signal. This allows the controller to effectively alter the dynamic characteristics of the system. A key challenge with state feedback control is that it requires full knowledge of the system's state, which may not always be practically available.

We can consider a simple mass-spring-damper system, a common physical system that can be controlled.

Let's say we have a 1 kg mass attached to a spring with stiffness 1 N/m, and a damper with damping coefficient 0.2 Ns/m. We can model this as a second-order system where the state vector is [position, velocity]. Our control input is a force applied to the mass.

First, let's define the system. In state-space form, the equations of motion are:

$$dx/dt = [v, -x - 0.2v + u]$$

We'll use the control systems library in Python to simulate and control this system.

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

# Define the system matrices
A = np.array([[0, 1], [-1, -0.2]])
B = np.array([[0], [1]])
C = np.array([1, 0])
D = 0

# Create a state-space system
sys = control.StateSpace(A, B, C, D)

# Define the state feedback gain
K = np.array([2, 3])

# Form the closed-loop system with state feedback
sys_cl = control.StateSpace(A - B @ K, B, C, D)

# Simulate the response to an initial condition of x = 1, v = 0
T, yout = control.step_response(sys_cl, T=np.linspace(0, 10, 1000), X0=[1, 0])

# Plot the response
plt.figure()
plt.plot(T, yout)
plt.xlabel('Time (s)')
plt.ylabel('Position (m)')
plt.title('Response of Mass-Spring-Damper System with State Feedback')
plt.grid(True)
```

```
plt.show()
```

In this code:

We start by importing the necessary libraries - numpy for numerical calculations, matplotlib for plotting, and control for control system functions.

Then we define the matrices A, B, C, and D. These are the matrices that define the state-space representation of the system:

A is the state matrix, which describes how the state of the system evolves over time.

B is the input matrix, which describes how the system's state is influenced by the input.

C is the output matrix, which maps the state of the system to the output.

D is the direct transmission matrix, which maps the input directly to the output.

The state-space system is then created using the control.StateSpace() function, passing in the matrices A, B, C, and D.

We define the state feedback gain K. This matrix determines how the state of the system is fed back to the input to control the system.

The closed-loop system is formed by subtracting the product of the input matrix B and the feedback gain K from the state matrix A. This is based on the principle of state feedback, where the control input is formed by feeding back the system state.

A step response simulation is then performed using the control.step_response() function. The initial conditions are specified as X0=[1, 0], representing a system that starts at position 1 and velocity 0. The function returns the time points T and the system output yout at these times.

Finally, we plot the system response using matplotlib's plotting functions.

What is Stat Space?

In control theory, a system can be represented in the state space form, which provides a convenient and compact way to model and analyze systems with multiple inputs and outputs.

With state space, the internal state of the system is explicitly accounted for. The 'state' of the system is the minimal set of variables (known as 'state variables') such that the knowledge of these variables at $t = t_0$ (initial time), along with the input in the interval $t \geq t_0$, completely determines the behavior of the system for $t > t_0$.

A state space representation of a system consists of a system of first-order differential equations. For an nth order system, there will be n state variables and n first-order differential equations.

The generic form of a state space representation is as follows:

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

where:

- $x(t)$ is the state vector (a vector of state variables)
- $u(t)$ is the input vector
- $y(t)$ is the output vector
- $\dot{x}(t)$ is the derivative of the state vector
- A, B, C, and D are matrices that define the system. The matrix A is called the state matrix, B is the input matrix, C is the output matrix, and D is the feedforward matrix.

This representation is very flexible and can model multiple-input multiple-output (MIMO) systems, time-varying systems, nonlinear systems, etc. It also provides a starting point for many control design techniques, like pole placement, optimal control, etc.

When you see `control.StateSpace` in Python, it is used to create a state space system model. This function takes the matrices A, B, C, and D (which define the system) as arguments, and returns a StateSpace object, which can be used for analysis or design purposes.

Full-State Feedback with Integrator

In some systems, it may be desirable to eliminate steady-state error in response to a step reference input. This can be achieved by augmenting the system with an integrator and designing a full-state feedback controller for the augmented system. This allows the system to achieve zero steady-state error in response to a step input, at the cost of increasing the system order by one.

System Augmentation

In full-state feedback, an important design objective can be to eliminate steady-state error, particularly in response to a step reference input. While this can be achieved by implementing an integrator in the controller, doing so may not always be feasible due to the increased complexity and the risk of introducing instability.

However, one workaround is to augment the system with an integrator. Augmentation is a technique where we add additional dynamics to the system to achieve a specific design objective. In this case, we augment the system with an integrator to eliminate the steady-state error.

Mathematically, if the original system is described by the state-space equations:

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

The augmented system with an integrator becomes:

$$\begin{bmatrix} \dot{x} \\ \dot{z} \end{bmatrix} = \begin{bmatrix} A & 0 \\ -C & 0 \end{bmatrix} \begin{bmatrix} x \\ z \end{bmatrix} + \begin{bmatrix} B \\ 0 \end{bmatrix} u$$
$$y = \begin{bmatrix} C & 0 \end{bmatrix} \begin{bmatrix} x \\ z \end{bmatrix} + Du$$

where z is the integrator state variable and the integrator itself has been added as an additional state to the system.

Designing the Controller

With the augmented system, we can now design a full-state feedback controller for it. This essentially means we find a gain matrix that allows us to control not just the original system states, but also the integrator state. This enables us to effectively control the steady-state error of the system.

The control law in this case would be of the form:

$$u = -K \begin{bmatrix} x \\ y \end{bmatrix}$$

where K is the state-feedback gain matrix. This control law would ensure that the system has zero steady-state error in response to a step input.

Considerations

While this method allows us to control the steady-state error, it's important to note that it increases the order of the system by one. This might complicate the controller design and make the system potentially harder to control. Furthermore, it requires the system to be completely controllable and observable in order to guarantee that the controller can place the system poles at the desired locations.

Overall, full-state feedback with an integrator is a powerful control design method, but it requires careful design and implementation to ensure stability and satisfactory performance.

Practical Applications of Control Systems

Control System Design for an Inverted Pendulum

An inverted pendulum is a classic problem in control theory. The objective is to design a controller that can balance a pendulum upright by applying forces

at its base. This problem is challenging because the pendulum is inherently unstable in its upright position. This system is often used as a benchmark for testing new control techniques and algorithms.

An inverted pendulum system consists of a hinge, a pole, and a cart that moves along a track. The hinge connects the cart and the pole. When the pole is balanced upright, any slight disturbance will cause the pole to start falling. The objective of the controller is to move the cart back and forth on the track to keep the pole balanced upright.

The equations of motion of the system can be derived from Newton's second law and are typically nonlinear. However, for small deviations from the upright position, the system can be linearized and approximated as a set of linear differential equations.

An inverted pendulum system is often used to illustrate various important concepts in control theory:

- **Unstable system:** In the absence of control, the pole will not stay upright. This makes the inverted pendulum a good example of an unstable system.
- **Non-minimum phase system:** Due to the physics of the problem, when we try to move the pole in one direction, it initially moves in the opposite direction. This makes the system non-minimum phase.
- **MIMO system:** The system has multiple inputs (forces on the cart) and multiple outputs (position and angle).

To design a controller for this system, various techniques can be used, such as PID control, state feedback control, optimal control, etc. The design of the controller is usually done based on the linearized system model, but the performance of the controller needs to be tested on the original nonlinear system.

In practical applications, the inverted pendulum problem serves as a simplified model for many real-world control problems, such as balancing a two-wheeled robot, stabilizing an aircraft during flight, or a rocket during launch.

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

# Parameters for the inverted pendulum
m = 0.5  # mass of the pendulum
M = 0.5  # mass of the cart
L = 0.5  # length of the pendulum
```

```
g = 9.81  # gravitational acceleration
d = 4.0  # damping factor

# Linearized system matrices
A = np.array([[0, 1, 0, 0],
              [0, -d/M, -(g*m)/M, 0],
              [0, 0, 0, 1],
              [0, -d/(M*L), -(g*(M+m))/(M*L), 0]])
B = np.array([0, 1/M, 0, 1/(M*L)]).reshape(-1,1)
C = np.array([1, 0, 0, 0])  # we're only measuring the cart position
D = np.array([0])

# Create a state-space model
sys = control.ss(A, B, C, D)

# Choose a controller gain (try different values)
K = np.array([1, 1, 1, 1]).reshape(-1,1)

# Form the closed-loop system
sys_cl = control.ss(A - B @ K.T, B, C, D)

# Simulate the response to an initial condition (try different values)
t, y = control.step_response(sys_cl, T=np.linspace(0, 10, 1000))

# Plot the response
plt.plot(t, y)
plt.xlabel('Time (s)')
plt.ylabel('Cart position (m)')
plt.title('Step response of the closed-loop system')
plt.grid()
plt.show()
```

In this code, we are simulating the control of an inverted pendulum. The following steps are taken:

1. **Define parameters:** The physical constants of the problem are defined: the mass of the pendulum (m), the mass of the cart (M), the length of the pendulum (L), the acceleration due to gravity (g), and a damping factor (d).

   ```
   m = 0.5 # mass of the pendulum
   M = 0.5 # mass of the cart
   L = 0.5 # length of the pendulum
   g = 9.81 # gravitational acceleration
   d = 4.0 # damping factor
   ```

2. **Linearization:** The dynamics of the inverted pendulum are nonlinear. However, to facilitate the control design, the system is linearized around the unstable equilibrium point (the upright position). The state of the system is defined as $[x, \dot{x}, \theta, \dot{\theta}]^T$ where x is the position of the cart, $\dot{x}$ is its velocity, θ is the angle of the pendulum, and $\dot{\theta}$ its angular velocity.

```
A = np.array([[0, 1, 0, 0],
[0, -d/M, -(gm)/M, 0],
[0, 0, 0, 1],
[0, -d/(ML), -(g*(M+m))/(ML), 0]])
B = np.array([0, 1/M, 0, 1/(ML)]).reshape(-1,1)
C = np.array([1, 0, 0, 0]) # we're only measuring the cart position
D = np.array([0])
```

3. **State-space representation:** The dynamics are represented in state-space form. The matrices A, B, C, and D represent the dynamics, the input, the output, and the direct transmission matrix, respectively. The Python control library is used to create a state-space object.

```
sys = control.ss(A, B, C, D)
```

4. **Controller design:** A full state feedback controller is designed. This means that the control input is a linear combination of the states. The gains of this linear combination are defined in the vector K. These gains have to be chosen carefully. In this case, they are chosen arbitrarily. The real system's closed-loop dynamics are given by $(A - BK)$.

```
K = np.array([1, 1, 1, 1]).reshape(-1,1)
sys_cl = control.ss(A - B @ K.T, B, C, D)
```

5. **Simulation:** The response of the closed-loop system to a step input is simulated. The control.step_response function is used for this purpose, which simulates the response of the system over a specified time interval.

```
t, y = control.step_response(sys_cl, T=np.linspace(0, 10, 1000))
```

6. **Plotting:** The response is plotted against time.

```
plt.plot(t, y)
plt.xlabel('Time (s)')
plt.ylabel('Cart position (m)')
plt.title('Step response of the closed-loopsystem')
plt.grid()
plt.show()
```

In conclusion, this script is a simple example of how control theory is applied to stabilize a nonlinear and unstable system. The inverted pendulum is a classic benchmark in control theory because it is a relatively simple system, yet it captures many of the challenges found in real-world control problems: nonlinearity, instability, and the need for feedback control. Despite its simplicity, the inverted pendulum is similar to many practical systems. For example, the techniques used to control an inverted pendulum can also be applied to stabilize a rocket during take-off, a process that is often likened to balancing a stick on your hand.

Note: In control systems design, a major objective is to make the system stable, i.e., the system's output remains bounded for all bounded inputs. For an unstable system like the inverted pendulum, this means that the system must be "stabilized" by the controller, which needs to balance the pole upright and keep it there in spite of disturbances.

Cruise Control System in Automobiles

We have previously discussed the application of PID controllers in a cruise control system for automobiles. The control objective is to maintain a desired speed by automatically adjusting the throttle position. This requires a controller that can handle variations in load (such as changes in road gradient and wind resistance) and reject disturbances.

Aircraft Pitch: System Modeling and Control

The pitch control of an aircraft is a significant control system design problem. The objective is to control the pitch angle (the angle the aircraft makes with the horizontal plane) to achieve desired flight conditions. This is typically achieved by controlling the elevator deflection angle. A pitch control system must be able to handle non-linear aerodynamic forces and moments, and the dynamics of the system can change significantly over the flight envelope.

Project Samples

1. **Simple Pendulum Simulation:**

   The simple pendulum, a mass hanging from a frictionless pivot that can swing in a plane, is a classic problem in dynamics and control. The equations of motion are given by a second-order nonlinear ordinary differential equation. Derive the equations of motion, linearize them around the vertical position, and simulate the pendulum motion over time for given initial conditions. Plot the pendulum position as a function of time and compare it with the solution assuming small angle oscillations.

2. **Population Dynamics:**

   Consider a simple ecological system consisting of two species: a predator and its prey. The interaction between the species can be described by the Lotka-Volterra equations. Simulate the system over time for a given set of initial conditions. Plot the populations of predators and prey as functions of time. Examine the impact of changes in the parameters representing the birth rate, death rate, and the interaction between the two species.

3. **SIR Model:**

   Simulate an outbreak using the SIR (Susceptible, Infectious, Recovered) model, with given parameters representing the disease transmission rate and recovery rate. Analyze the impact of these parameters on the progression of the outbreak. Plot the susceptible, infectious, and recovered populations as functions of time.

4. **Control of a DC Motor:**

   DC motors are common in many applications and understanding their control can be very educational. Model a DC motor taking into account parameters like resistance, inductance, and the back electromotive force constant. Design a PID controller to meet specifications such as rise time, settling time, and overshoot. Simulate the system's step response. Analyze the impact of each of the PID parameters on the system response.

5. **Mass-Spring-Damper System:**

   The mass-spring-damper system is a classical mechanical model representing many physical systems, such as vehicle suspension. The system consists of a mass (M), a spring with spring constant (K), and a damper with damping coefficient (B). Derive the system equations, simulate the system response for a given set of initial conditions, and analyze the effects of varying the mass, spring constant, and damping coefficient.

6. **Temperature Control:**

   Consider a simple system where a room is heated by an electric heater, and the temperature of the room is controlled by a thermostat. This system can be described by a first-order differential equation. Develop a

model for this system, simulate it for a step input in desired temperature, and design a controller (such as a PID controller) to maintain the room temperature close to the desired temperature.

7. **Queuing Theory:**

   Consider a bank with a single server where customers arrive randomly. The inter-arrival times and service times can be modeled as exponential random variables. This system can be described by a birth-death process, a special case of continuous-time Markov chain. Simulate this queuing system and compute performance measures like the average number of customers in the system, the average time a customer spends in the system, etc.

8. **Traffic Flow Simulation:**

   Create a simple traffic flow model on a single-lane road with a specified speed limit. Assume vehicles enter the road at one end and exit at the other, with Poisson distributed inter-arrival times. The distance between each vehicle and the one ahead follows a specified car-following model. Simulate this system over a specified time period and plot the vehicle positions as a function of time. Analyze the impact of parameters like vehicle arrival rate, speed limit, and safe following distance on the traffic flow.

9. **Spread of Rumors or Information:**

   This project involves the use of network theory and simple probability to simulate how information or rumors spread across a social network. Assume that the network is represented by a graph, where each node is a person and each edge represents a connection between two people. A person who knows the information (or rumor) will share it with each connected neighbor with a certain probability during each time step. Simulate this information spread over the network and plot the number of informed persons as a function of time. Analyze how changing the information sharing probability affects the speed and extent of the information spread.

Each task should include:

- A clear explanation of the model and its assumptions.
- The mathematical equations representing the model.
- The Python code used to simulate the model.
- Graphical visualizations of the simulation results.
- An analysis of the simulation results, including the impact of varying model parameters.