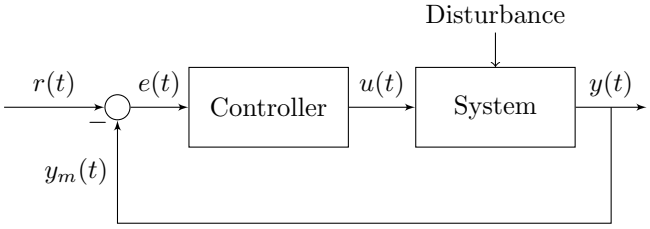
Modeling and Simulation Lecture Notes - 7

July 2023

Introduction to Control Systems

Definition of a Control System

A Control System is an arrangement of physical components connected in such a way to regulate or direct behavior of other systems. It is designed to manage, command, direct, or regulate the behavior of other devices or systems. Mathematically, it can be expressed in the form of differential equations.



In the diagram:

r(t) is the reference input to the system (the desired output), y(t) is the actual output of the system, e(t) = r(t) - y_m(t) is the error, the difference between the desired and actual output, u(t) is the control input to the system, which is designed to reduce the error, y_m(t) is the measured output, which may be different from the actual output due to noise or other disturbances.

Open-loop and Closed-loop systems

Open-loop Systems

In an open-loop control system, the control action from the controller is independent of the "process output", which is the process quantity being controlled. An example can be a washing machine. You specify a time, and the machine will run for that specific time regardless of whether the clothes are cleaned or not. Mathematically, if the input of the system is denoted as $u(t)$ and output as $y(t)$, open-loop system can be expressed as $y(t) = G(u(t))$ where G is the system's transfer function.

Closed-loop Systems

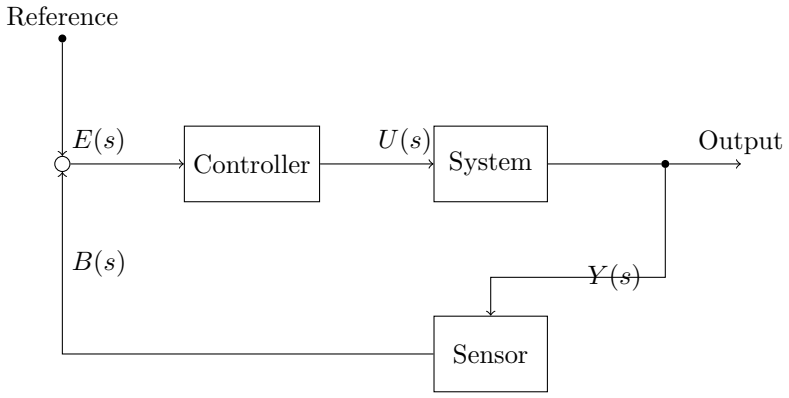
In a closed-loop control system, the control action from the controller is dependent on the process output. For instance, a thermostat-controlled heating system: the thermostat senses the temperature and controls the heater to maintain the temperature at a desired level. In a mathematical form, a closed-loop control system takes into account the feedback from the output to the input for control purposes. It can be expressed as $y(t) = G(u(t) + H(y(t)))$ where H represents the feedback transfer function.

Elements of a Control System

A typical control system consists of sensors, controllers, actuators, and the system plant. These elements form a feedback loop that enables precise control of the system dynamics. Below is an extended discussion of each component:

- **Sensors:** Sensors are devices used to measure various quantities, such as temperature, pressure, speed, and position, among others. The sensors continuously monitor these quantities and convert them into signals that can be read by the controller. In a home thermostat system, for instance, the sensor is a thermometer that measures the room's temperature.
- **Controller:** The controller is the brain of the control system. It receives the input signals from the sensors, which it then compares with the desired setpoints to compute the error signal (the difference between the desired and actual outputs). The controller uses this error signal to compute the control action based on a predefined control algorithm (PID, state feedback, etc.). In the thermostat system, the controller is the device that you set your desired temperature on. It calculates the difference between the desired and actual temperature and decides when to turn on or off the heating system.
- **Actuator:** The actuator is the device that applies the control action to the system. Depending on the control signal from the controller, the actuator could increase or decrease the system's input to drive the system towards the desired state. The actuator could be a motor, a valve, a pump, etc. In the thermostat example, the actuator could be an electric switch controlling the heater.
- **Plant:** The plant is the actual system to be controlled. It could be a physical system (like a drone or a vehicle), a chemical process, an electrical circuit, or any other system whose behavior we wish to control. In the thermostat example, the plant could be considered as the room whose temperature we want to control.

These components work together in a feedback loop to maintain the system at the desired state, compensating for any disturbances or changes in the system dynamics.



Examples of Control System Elements

To better understand the elements of a control system, let's take a look at a few practical examples:

- **Air Conditioning System:** In an air conditioning system, the *plant* is the room whose temperature we want to control. The *sensor* is a thermometer that measures the room's temperature. The *controller* is the thermostat device that you set your desired temperature on. It calculates the difference between the desired and actual temperature and decides when to turn on or off the cooling system. The *actuator* is the electric switch controlling the air conditioning system.
- **Cruise Control in a Car:** The cruise control system in a car is another good example. The *plant* is the car itself. The *sensor* is a speedometer that measures the car's actual speed. The *controller* is the cruise control unit that you set your desired speed on. It calculates the difference between the desired and actual speed and decides how much to press or release the gas pedal. The *actuator* is the system that controls the throttle opening based on the control signal from the controller.
- **Automatic Pilot in an Aircraft:** In an aircraft's automatic pilot system, the *plant* is the aircraft. The *sensors* include the gyroscope, altimeter, and other instruments that measure the aircraft's attitude, altitude, speed, etc. The *controller* is the autopilot computer that receives the desired state (altitude, speed, etc.) and computes the control signals based on the error signals. The *actuators* include the control surfaces (elevators, ailerons, rudder) and the engine throttle, which are controlled based on the signals from the controller.

These examples help us understand how different components of a control system work together in real-world applications to achieve the desired system behavior.

Transfer Function and Impulse Response Function

The *Transfer Function* of a system is a mathematical model in frequency domain that is used to represent the relation between the input and output of a Linear Time-Invariant system. For a single-input and single-output (SISO) system with input $u(t)$ and output $y(t)$, the transfer function $G(s)$ is defined as the ratio of the Laplace transform of the output $Y(s)$ to the Laplace transform of the input $U(s)$, under the assumption that all initial conditions are zero.

$$G(s) = \frac{Y(s)}{U(s)}$$

The *Impulse Response Function*, denoted as $g(t)$, is the output of a system when the input is a Dirac delta function. If $G(s)$ is the transfer function of a system, the impulse response function can be obtained by taking the inverse Laplace transform of $G(s)$.

Further Understanding of Transfer Function and Impulse Response Function

To gain a deeper understanding of these concepts, let's consider a simple first-order system, such as an RC (Resistor-Capacitor) circuit.

Modeling an RC Circuit

An RC circuit can be used as a simple model of a first-order system. It consists of a resistor and a capacitor in series. When a voltage $u(t)$ is applied to the circuit, the voltage across the capacitor $y(t)$ changes over time according to the differential equation

$$RC\frac{dy(t)}{dt} + y(t) = u(t)$$

where R is the resistance, C is the capacitance, and RC is the time constant of the system.

Transfer Function of an RC Circuit

Taking the Laplace transform of the above differential equation and assuming zero initial conditions, we get the transfer function of the RC circuit:

$$G(s) = \frac{Y(s)}{U(s)} = \frac{1}{RCs + 1}$$

This function shows how the output of the system (the capacitor voltage) responds to different frequencies of the input signal.

Impulse Response of an RC Circuit

The impulse response of the system can be obtained by taking the inverse Laplace transform of the transfer function:

$$g(t) = L^{-1}\{G(s)\} = e^{-t/RC}$$

for $t > 0$. This is the response of the capacitor voltage if the input voltage were a Dirac delta function, i.e., an impulse at time zero.

This example illustrates the basic process of obtaining a system's transfer function and impulse response, which are crucial in analyzing and designing control systems.

Introduction to Python Control Systems Library

The Python Control Systems Library, also known as python-control, is a powerful open-source library that provides tools for system modeling, analysis, and design, primarily for control systems. It features a simple syntax and builds upon the familiar scientific computing libraries NumPy and SciPy, making it intuitive for users experienced with Python's scientific stack.

To install the Python Control Systems Library, use pip:

```
pip install control
```

Creating a System Model

To create a system model, we typically use the TransferFunction or StateSpace classes, depending on the representation we want to use for our system. Both of these classes are defined in the python-control package.

Transfer Function Representation

A transfer function can be created using the TransferFunction class. This class accepts two arrays: one for the coefficients of the numerator and one for the coefficients of the denominator.

For example, to create a transfer function $G(s) = \frac{s+3}{s^2+3s+2}$, you can use the following code:

```
import control
num = [1, 3] # Coefficients for s + 3
den = [1, 3, 2] # Coefficients for s^2 + 3s + 2
G = control.TransferFunction(num, den)
```

State-Space Representation

A state-space model can be created using the StateSpace class. This class accepts four arrays representing the state-space matrices A, B, C, and D.

For example, to create a state-space model for a system with matrices A = [[0, 1], [-2, -3]], B = [[0], [1]], C = [[1, 0]], and D = [[0]], you can use the following code:

```
import control
A = [[0, 1], [-2, -3]]
B = [[0], [1]]
C = [[1, 0]]
D = [[0]]
sys = control.StateSpace(A, B, C, D)
```

Linear Time-Invariant (LTI) Systems

The Python Control Systems Library assumes that all systems are Linear Time-Invariant (LTI). This is a common assumption in control systems theory. LTI systems are characterized by two properties:

- Linearity: The principle of superposition applies. This means that the response to a sum of inputs is equal to the sum of the responses to the individual inputs.
- Time-invariance: The behavior of the system does not change over time. This means that if the input signal is shifted in time, the output will also be shifted by the same amount.

Both the TransferFunction and StateSpace classes represent LTI systems.

Transfer Functions in Python

As we have discussed, the python-control library allows us to create transfer function models of LTI systems. Beyond creating these models, the library also provides numerous functions for working with them. For example, we can use the step function to compute the step response of the system, or the bode function to plot the Bode plot of the system.

For instance, to compute and plot the step response of a system, we can use the following code:

```
import control
import matplotlib.pyplot as plt

num = [1, 3]
den = [1, 3, 2]
G = control.TransferFunction(num, den)
```

```
T, yout = control.step_response(G)

plt.figure()
plt.plot(T, yout)
plt.xlabel('Time [s]')
plt.ylabel('Amplitude')
plt.title('Step response')
plt.grid(True)
plt.show()
```

Interconnecting Systems

The python-control library also provides functions to interconnect systems. This includes series, parallel, and feedback interconnections.

For example, to connect two systems in series (meaning the output of the first system is the input to the second system), we can use the series function:

```
import control

#Define two systems
num1 = [1]
den1 = [1, 1]
sys1 = control.TransferFunction(num1, den1)

num2 = [1]
den2 = [1, 2]
sys2 = control.TransferFunction(num2, den2)

#Connect in series
sys_series = control.series(sys1, sys2)
```

Similar functions exist for parallel and feedback interconnections (control.parallel and control.feedback, respectively).

Creating a Transfer Function

Consider a system with the transfer function $H(s) = \frac{s+3}{s^2+3s+2}$. The coefficients of the numerator and denominator polynomials represent this system, which we can define in Python as follows:

```
import control

# Numerator coefficients
num = [1, 3]
```

```
# Denominator coefficients
den = [1, 3, 2]

# Creating the Transfer Function
G = control.TransferFunction(num, den)
```

Plotting the Step Response

After defining the Transfer Function, we can simulate the step response. The step response of a system is the output when the input is a unit step function. We compute and plot the step response as follows:

```
import matplotlib.pyplot as plt

# Time array for the simulation
T = np.linspace(0, 10, 1000)

# Compute step response
T, yout = control.step_response(G, T)

# Create the plot
plt.figure()
plt.plot(T, yout)
plt.title('Step Response')
plt.xlabel('Time (s)')
plt.ylabel('Amplitude')
plt.grid(True)
plt.show()
```

This will create a plot of the system's step response, showing how the system would evolve over time if the input changed abruptly from 0 to 1 at the initial time.

Modeling and Simulation of a PID Controller using Python

A PID controller is a commonly used type of feedback controller and is widely used in various engineering fields.

PID Control is a type of feedback control system. It stands for Proportional-Integral-Derivative Control. The controller tries to minimize the error between the desired output (reference signal) and the measured output of the system.

- *Proportional Control (P)*: This is the part of the control action that is proportional to the current error. If the error is large, the control action

is large, and vice versa. The proportionality constant is known as the proportional gain (K_p).

This is like when you look at your speedometer and see that you're going 50 mph, but you want to be going 60 mph. You press harder on the gas pedal to go faster. If you're going way too slow or way too fast, you'll press or release the gas pedal a lot. This "proportional" part means the more wrong we are, the harder we try to make it right.

- *Integral Control (I)*: This part of the control action is proportional to both the magnitude and the duration of the error. It sums up (integrates) the error over time. This action is useful to eliminate the steady-state error. The integral action is determined by the integral gain (K_i).

  Let's say there's a small hill or some wind, and even though you're pressing the gas pedal the same amount as before, now you're only going 58 mph. This is where the integral part comes in. It's like if you notice that you've been below your desired speed for a while, you might start pressing the gas pedal a little more, even if you're only a little bit below the desired speed. The integral part pays attention to how long we've been wrong.

- *Derivative Control (D)*: This is the part of the control action that is proportional to the rate of change of the error. It anticipates the future behavior of the error. By looking at the rate at which the error is changing, the derivative control can provide the system with damping and improve the stability. The derivative action is determined by the derivative gain (K_d).

  Now, think about what happens when you're coming up on a big hill. You don't want to start up the hill too fast and then have to slow down a lot, and you don't want to start up the hill too slow and then have to speed up a lot. The derivative part is like seeing the hill coming up and starting to press the gas pedal a little bit more to prepare. The derivative part is about noticing how things are changing and getting ready for what's coming next.

  So in summary, the PID controller is like a really good driver. It pays attention to how fast we're going (P), how long we've been going the wrong speed (I), and what things look like they're going to happen soon (D).

- A PID controller combines these three actions to get a control signal. The mathematical expression for a PID controller is given as:

$$u(t) = K_p e(t) + K_i \int e(t)dt + K_d \frac{de(t)}{dt}$$

where $u(t)$ is the control signal, $e(t)$ is the error signal (difference between the reference and the output), and t is time.

The mathematical expression of a PID controller is:

$$u(t) = K_p e(t) + K_i \int e(t)dt + K_d \frac{de(t)}{dt}$$

where: - $u(t)$ is the controller output, - $e(t)$ is the control error (the difference between the desired and the actual output), - K_p, K_i, K_d are the proportional, integral, and derivative gains, respectively.

We can implement a simple simulation of a PID controller in Python. Here, we will consider a first-order system as the plant to be controlled, described by the differential equation:

$$\frac{dx}{dt} = -x + u$$

The control goal is to adjust the input u so that the system output x tracks a desired reference signal.

We can implement this system in Python using the scipy.integrate.odeint function:

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

# PID parameters
Kp = 1.0   # proportional gain
Ki = 1.0   # integral gain
Kd = 0.0   # derivative gain

# reference signal
ref = lambda t: 1.0   # step reference at t=0

# system to be controlled
def plant(x, t, u):
    dxdt = -x + u
    return dxdt

# PID controller
```

```
def pidController(t, e, e_prev, int_e, Kp, Ki, Kd):
    diff_e = e - e_prev
    control = Kp*e + Ki*int_e + Kd*diff_e
    return control, e, int_e + e

# initial conditions
x0 = 0.0
e_prev = 0
int_e = 0

# time array
t = np.linspace(0.0, 10.0, 1000)
dt = t[1] - t[0]

# variables to store
x = np.zeros(t.shape)
control = np.zeros(t.shape)
e = np.zeros(t.shape)

# simulation
for i in range(len(t)):
    x[i] = odeint(plant, x0, [0, dt], args=(control[max(0, i-1)],))[-1]
    e[i] = ref(t[i]) - x[i]
    control[i], e_prev, int_e = pidController(t[i], e[i], e_prev, int_e, Kp, Ki, Kd)
    x0 = x[i]

# plot response
plt.figure()
plt.subplot(2, 1, 1)
plt.plot(t, x, label='x')
plt.plot(t, [ref(ti) for ti in t], label='ref') # Modified line
plt.legend()
plt.subplot(2, 1, 2)
plt.plot(t, control, label='control')
plt.legend()
plt.show()
```

This Python code implements a PID controller for a first-order system. The PID controller adjusts the input u so that the system output x follows a desired reference signal. The controller parameters K_p, K_i, and K_d can be adjusted to change the controller's behavior.

Setting PID parameters: The script begins by setting the proportional gain (Kp), integral gain (Ki), and derivative gain (Kd). These values can be adjusted to tune the PID controller's performance.

Defining the reference signal: The reference signal is a function that represents the desired output of the system. In this script, the reference signal ref is a constant function that always returns 1.0.

Defining the plant and PID controller: The plant is the system that we want to control, defined by the differential equation in the function plant. The PID controller is defined by the function pidController, which calculates the control input u based on the current error e, the previous error e_prev, and the integral of the error int_e.

Initializing variables: The initial conditions and arrays for storing the output of the system x, the control input control, and the error e are initialized.

Simulating the system: The script then runs a for-loop over the time array t to simulate the system. For each time step, it uses the odeint function to solve the plant's differential equation, calculates the current error, and uses the PID controller to calculate the control input for the next time step.

Plotting the results: Finally, the script plots the system output x, the reference signal ref, and the control input control as functions of time.

Cruise Control Example

In this example, the cruise control system increases the throttle position (presses the accelerator pedal) to increase the car's speed to the desired level. When the desired speed changes at t = 5.0 seconds, the controller adjusts the throttle position to achieve the new desired speed. The controller will adjust the throttle position based on the difference between the actual and desired speed (the "error") and the integral and derivative of the error.

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

# PID parameters
Kp = 2.0  # proportional gain
Ki = 1.0  # integral gain
Kd = 0.1  # derivative gain

# reference signal (desired speed, changes at t=5.0 seconds)
def ref(t):
    if t < 5.0:
        return 20.0  # 20 m/s ~ 72 km/h
    else:
        return 25.0  # 25 m/s ~ 90 km/h
```

```
# system to be controlled (car's speed)
def plant(v, t, u):
    dvdt = -v + u  # simple first-order model
    return dvdt

# PID controller
def pidController(t, e, e_prev, int_e, Kp, Ki, Kd):
    diff_e = e - e_prev
    control = Kp*e + Ki*int_e + Kd*diff_e
    return control, e, int_e + e

# initial conditions
v0 = 0.0  # start from rest
e_prev = 0
int_e = 0

# time array
t = np.linspace(0.0, 20.0, 1000)
dt = t[1] - t[0]

# variables to store
v = np.zeros(t.shape)
control = np.zeros(t.shape)
e = np.zeros(t.shape)

# simulation
for i in range(len(t)):
    v[i] = odeint(plant, v0, [0, dt], args=(control[max(0, i-1)],))[-1]
    e[i] = ref(t[i]) - v[i]
    control[i], e_prev, int_e = pidController(t[i], e[i], e_prev, int_e, Kp, Ki, Kd)
    v0 = v[i]

# plot response
plt.figure()
plt.subplot(2, 1, 1)
plt.plot(t, v, label='Speed (v)')
plt.plot(t, [ref(ti) for ti in t], label='Desired speed (ref)')
plt.legend()
plt.ylabel('Speed (m/s)')
plt.subplot(2, 1, 2)
plt.plot(t, control, label='Throttle position (control)')
plt.legend()
plt.ylabel('Throttle position')
plt.xlabel('Time (s)')
plt.show()
```

The reference signal ref(t) is the speed set by the driver, which changes over time. For example, at time t = 5.0 seconds, the driver sets a new speed. The plant (system) model plant(v, t, u) is a simplified car's speed model, where v is the speed, u is the throttle position (control signal), and the equation -v + u is a first-order model where a positive throttle position u increases speed, and speed naturally decays over time due to friction.

The PID controller is responsible for adjusting the throttle position (accelerator pedal position) to achieve the desired speed.

Analysis of Control Systems

Control systems analysis is essential to understand the behavior of the system and its response to different inputs or disturbances. Various analysis techniques provide information about the stability, transient response, and frequency response of the system. This section introduces several fundamental methods for control systems analysis.

0.1 Step Response, Impulse Response, and Frequency Response

- The *step response* of a system is its output when subjected to a step input signal. This response provides insights into the system's stability and performance characteristics such as rise time, settling time, and overshoot.
- The *impulse response* of a system is its output when subjected to an impulse input signal. The impulse response contains all the information about the system and can be used to determine its response to any input signal using convolution.
- The *frequency response* of a system is the steady-state response of the system to sinusoidal input signals as a function of frequency. The frequency response provides information about the magnitude and phase of the system at different frequencies.

Let's consider the cruise control system that we've discussed before as a real-life example to understand step response, impulse response, and frequency response. We've already looked at the step response. Now, let's discuss the impulse response and frequency response using Python.

Below is the Python code for computing the impulse response and frequency response of our cruise control system modeled as a first-order system.

```
import numpy as np
```

```
from scipy import signal
import matplotlib.pyplot as plt

# System parameters
tau = 5.0  # time constant
K = 2.0    # gain

# Create a linear system representation
system = signal.lti([K], [tau, 1])

# Compute and plot impulse response
t_impulse, y_impulse = signal.impulse(system)
plt.figure()
plt.plot(t_impulse, y_impulse)
plt.title('Impulse response')
plt.xlabel('Time [s]')
plt.ylabel('Response')
plt.grid(True)

# Compute and plot frequency response
frequencies, mag, phase = signal.bode(system)
plt.figure()
plt.subplot(2, 1, 1)
plt.semilogx(frequencies, mag)    # Bode magnitude plot
plt.title('Frequency response')
plt.xlabel('Frequency [Hz]')
plt.ylabel('Magnitude [dB]')
plt.grid(True)
plt.subplot(2, 1, 2)
plt.semilogx(frequencies, phase)  # Bode phase plot
plt.xlabel('Frequency [Hz]')
plt.ylabel('Phase [degrees]')
plt.grid(True)
plt.tight_layout()
plt.show()
```

The impulse response represents the car's speed response to a sudden and brief burst of throttle, such as you might see if you momentarily floored the gas pedal. This is less realistic in a physical sense, but it's important in control theory because the impulse response fully characterizes the system's dynamics.

The frequency response represents the car's speed response to a sinusoidal throttle input at different frequencies. For instance, the magnitude response at a frequency of 0.1 Hz tells you how much the car's speed would oscillate in response to a throttle input that oscillates every 10 seconds. The phase

response tells you the phase shift between the throttle oscillation and the speed oscillation.

Stability Analysis: Poles and Zeros

The stability of a system is analyzed by examining the location of the poles of its transfer function.

- A system is stable if all its poles lie in the left half of the complex plane (for continuous-time systems) or inside the unit circle (for discrete-time systems).
- The zeros of the transfer function can affect the transient and steady-state response of the system.

Root Locus Plot

The root locus plot provides a graphical representation of the movement of the poles of a system as a system parameter (usually a gain) is varied. It is a powerful tool for understanding the effects of parameter changes on the system's stability and transient response.

Bode and Nyquist Plots

- The Bode plot provides a graphical representation of the frequency response of a system. It consists of two plots: the magnitude plot (which shows the magnitude of the frequency response as a function of frequency) and the phase plot (which shows the phase of the frequency response as a function of frequency).
- The Nyquist plot is another graphical representation of the frequency response. It is a parametric plot of the complex-valued frequency response, with the real part of the response on the x-axis and the imaginary part on the y-axis. The Nyquist plot provides a powerful method for assessing the stability of a system.

Root Locus, Bode, and Nyquist Plots of a Cruise Control System

Consider the cruise control system that we've modeled as a first-order system with a time constant of $\tau = 5.0$ seconds and a gain of $K = 2.0$. We can examine how the system behaves under different conditions using the root locus, Bode, and Nyquist plots.

Python's control systems library, control, provides functions for creating these plots. Here's the Python code for generating these plots:

```
import control
import matplotlib.pyplot as plt

# System parameters
tau = 5.0  # time constant
K = 2.0    # gain

# Create a transfer function representation of the system
system = control.TransferFunction([K], [tau, 1])

# Create the root locus plot
plt.figure()
control.root_locus(system)
plt.title('Root Locus')

# Create the Bode plot
plt.figure()
mag, phase, omega = control.bode(system, dB=True)

# Create the Nyquist plot
plt.figure()
control.nyquist_plot(system, omega=[0.0001, 10])
plt.title('Nyquist Plot')

plt.show()
```

In the root locus plot, you can see how the pole of the system (marked with an 'x') moves along the real axis as the gain K increases from 0 to infinity. This provides insight into how the system's stability and transient response would change with different gains.

The Bode plot shows the magnitude and phase of the frequency response. The magnitude plot shows how effectively the system can track sinusoidal inputs of different frequencies, while the phase plot shows the phase lag between the input and output signals at these frequencies.

The Nyquist plot provides a different view of the frequency response, showing the real and imaginary parts of the frequency response together. This plot can be used to assess system stability using the Nyquist stability criterion.

Analysis of Control Systems Examples

This section provides brief examples of various control systems analysis methods introduced earlier.

Step Response, Impulse Response, and Frequency Response Examples

The examples for these topics are usually performed using simulation tools. The input signal (step, impulse, or sinusoidal) is applied to the system's mathematical model, and the output response is observed. Here is a simple example for a step response of a first-order system.

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

# First order system: H(s) = 1 / (s + 1)
sys = signal.TransferFunction([1], [1, 1])
t, y = signal.step(sys)

plt.figure()
plt.plot(t, y)
plt.title('Step response for 1st order system')
plt.xlabel('Time')
plt.ylabel('Amplitude')
plt.grid(True)
plt.show()
```

Stability Analysis: Poles and Zeros Example

The poles and zeros of a system can be computed from the coefficients of the numerator and denominator of the transfer function. For example, for the first-order system defined above, the pole is at -1 (making it a stable system) and there are no zeros.

Root Locus Plot Example

The root locus plot can be created using control system analysis software or libraries. For a simple example, consider a system with a transfer function $G(s) = K/(s(s+1)(s+2))$, where K is a variable gain.

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

sys = signal.TransferFunction([1], [1, 3, 2, 0])
```

```
r, k = signal.root_locus(sys)

plt.figure()
plt.plot(r.real, r.imag, '.')
plt.title('Root locus plot of the system')
plt.xlabel('Re')
plt.ylabel('Im')
plt.grid(True)
plt.show()
```

Bode and Nyquist Plots Example

The Bode and Nyquist plots can also be created using control system analysis software or libraries. For example, for the first-order system defined above, the Bode plot can be created as follows:

```
import matplotlib.pyplot as plt
from scipy import signal

sys = signal.TransferFunction([1], [1, 1])
w, mag, phase = signal.bode(sys)

plt.figure()
plt.semilogx(w, mag)  # Bode magnitude plot
plt.figure()
plt.semilogx(w, phase)  # Bode phase plot
plt.show()
```

For the Nyquist plot:

```
import matplotlib.pyplot as plt
from scipy import signal

sys = signal.TransferFunction([1], [1, 1])
w, h = signal.freqresp(sys)

plt.figure()
plt.plot(h.real, h.imag)  # Nyquist plot
plt.show()
```