Lab Sheet:
Week 2 - Modeling of Dynamic Systems

Learning Objectives:

1. Understand transfer function representation and its operation in Python Control
2. Understand state space representation and converting its into transfer function and vice versa
3. Learn to use Python tools for system identification and parameter estimation
4. Learn about input signal of system

Task 1:
Transfer Function Representation

A transfer function is a mathematical representation of a system's input-output relationship. It is a ratio of the system's output to its input, and it is typically expressed in terms of Laplace transforms.

$$G(s) = \frac{num(s)}{den(s)} = \frac{a_0 s^m + a_1 s^{m-1} + ... + a_m}{b_0 s^n + b_1 s^{n-1} + ... + b_n}$$

The Python Control library provides a class called `control.TransferFunction()` that can be used to represent transfer functions. The `control.TransferFunction()` class takes two arguments: the numerator and denominator polynomials of the transfer function. To create a transfer function, use the `control.TransferFunction()` or `control.tf()` function:

```python
sys = control.tf(num, den)
sys = control.TransferFunction(num, den)
```

For example, the following code creates a transfer function object that represents a simple first-order system:

```python
import control

sys = control.TransferFunction([1], [1, 1])
print(sys)
```

The `sys` object can then be used to perform various operations on the transfer function, such as calculating its frequency response or stability.

Here is an example of how to use the `control.TransferFunction()` class to represent a transfer function:

```python
import control

sys = control.TransferFunction([1], [1, 1])
print(sys)
print(sys.num)
print(sys.den)
```

Task 2:
First Order System

A first order system is a system that can be represented by a transfer function of the form:

$$H(s) = \frac{K}{\tau s + 1}$$

where `K` is the gain of the system, `τ` is the time constant value for the first-order system.

For example, the following code creates a first order system with a gain of 2.5 and a time constant of 0.8 second:

```python
K = 2.5  # Gain
tau = 0.8  # Time constant

num = [K]
den = [tau, 1]
tf_first_order = control.TransferFunction(num, den)

print(tf_first_order)
```

Task 3:
Second Order System

A second order system is a system that can be represented by a transfer function of the form:

$$H(s) = \frac{K\omega_n^2}{s^2 + 2\zeta\omega_n s + \omega_n^2}$$

where `K` is the gain of the system, `ζ` is the damping coefficient, and `ω` is the natural frequency of the system.

For example, the following code creates a second order system with a gain of 1.5, a damping coefficient of 0.6, and a natural frequency of 2 rad/s:

```python
K = 1.5  # Gain
```

```
zeta = 0.6  # Damping ratio
omega_n = 2  # Natural frequency

num = [K * omega_n**2]
den = [1, 2 * zeta * omega_n, omega_n**2]
tf_second_order = control.TransferFunction(num, den)

print(tf_second_order)
```

Task 4:
Combination of Transfer Function

We will explore the operations of combining transfer functions in series, parallel, and feedback configurations using Python Control. These operations are fundamental in control systems engineering as they allow us to analyze and design complex control systems by combining simpler transfer functions.

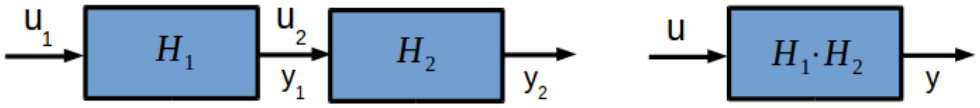
Before performing the operations, let's create some transfer function instances representing different subsystems:

```python
# First-order transfer function
K1 = 2.5
tau1 = 0.8
num1 = [K1]
den1 = [tau1, 1]
H1 = control.TransferFunction(num1, den1)

# Second-order transfer function
K2 = 1.5
zeta2 = 0.6
omega_n2 = 2
num2 = [K2 * omega_n2**2]
den2 = [1, 2 * zeta2 * omega_n2, omega_n2**2]
H2 = control.TransferFunction(num2, den2)
```

1. Combining Transfer Functions in Series
   In a series connection, the outputs of two systems are connected to the inputs of another system. The transfer function of the overall system is the product of the transfer functions of the individual systems.

   For example, consider two systems with transfer functions $H_1(s)$and $H_2(s)$. The transfer function of the overall system is given by:

$$H(s) = H_1(s) * H_2(s)$$



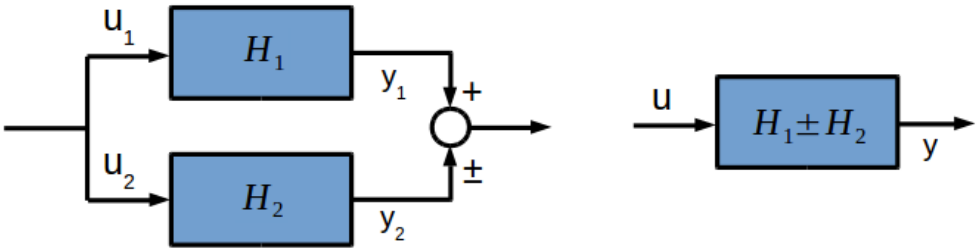
To combine transfer functions in series, we use the `control.series()` function:

```python
tf_series = control.series(H1, H2)

print("Transfer Function in Series:")
print(tf_series)
```

2. Combining Transfer Functions in Parallel
   In a parallel connection, the inputs of two systems are connected to the same output. The transfer function of the overall system is the sum of the transfer functions of the individual systems.

   For example, consider two systems with transfer functions $H_1(s)$and $H_2(s)$. The transfer function of the overall system is given by:

$$H(s) = H_1(s) + H_2(s)$$



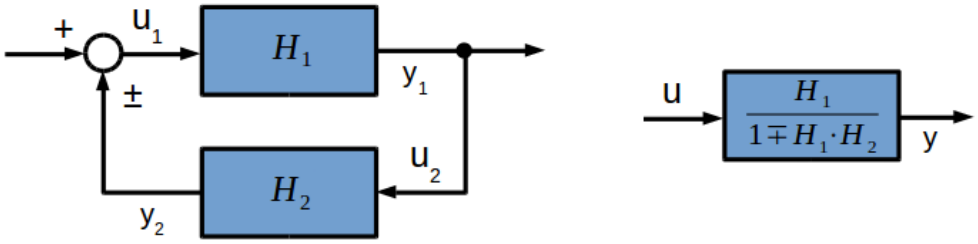
To combine transfer functions in parallel, we use the `control.parallel()` function:

```python
tf_parallel = control.parallel(H1, H2)

print("\nTransfer Function in Parallel:")
print(tf_parallel)
```

3. Creating a Feedback Control System
   In a feedback connection, the output of a system is fed back to its input. The transfer function of the overall system is the product of the transfer function of the system and the feedback gain.

   For example, consider a system with transfer functions $H_1(s)$and $H_2(s)$. The transfer function of the overall system is given by:

$$H(s) = \frac{H_1(s)}{1 + H_1(s) \cdot H_2(s)}$$



To create a feedback control system, we use the `control.feedback()` function:

```python
tf_feedback = control.feedback(H1, H2)

print("\nTransfer Function in Feedback:")
print(tf_feedback)
```

Task 5:
State space representation

State space modeling is a mathematical technique widely used in control systems engineering to represent the behavior of dynamic systems. It provides a systematic and elegant way to describe the evolution of a system over time using a set of first-order differential equations. The state space representation is particularly valuable for analyzing complex systems, designing control algorithms, and performing various control system analyses.

The state-space representation consists of two main components: state equations and output equations.

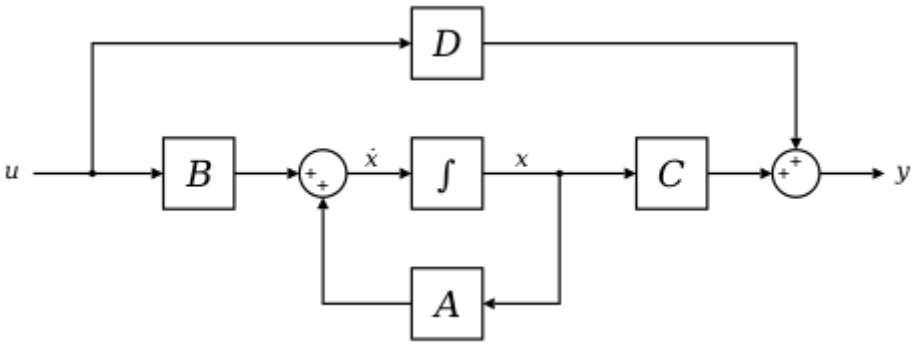
- State Equations:
  The state equations represent the evolution of the system's state variables over time. The state variables are a set of variables that describe the internal state of the system and are necessary to fully determine the system's future behavior. They capture the essential information about the system at any given time. The state equations are typically represented in matrix form as follows:
  $$\dot{x} = Ax + Bu$$
  Where:
  - x is the state vector, containing n state variables ($n \times 1$matrix).
  - A is the state matrix ($n \times n$), which defines the relationships between the state variables and their time derivatives.
  - B is the input matrix ($n \times m$), which relates the inputs to the state variables.
  - u is the input vector, containing m input variables ($m \times 1$ matrix).

- Output Equations:

The output equations describe the relationship between the system's state variables and the measured or observed outputs of the system. The outputs are typically variables of interest that we want to control or analyze. The output equations are represented as follows:

$$y = Cx + Du$$

Where:
y is the output vector, containing p output variables ($p \times 1$matrix).
C is the output matrix ($p \times n$), which relates the state variables to the outputs.
D is the feedthrough matrix ($p \times m$), which relates the inputs directly to the outputs.



To represent a system in state-space form using the Python Control library, you can use the `control.StateSpace()` class. The state-space representation is a mathematical model that describes a system in terms of its state variables, input, and output equations. Here's how you can create a state-space representation in Python Control:

```python
import control

A = [[-2, 0], [1, -1]]
B = [[1], [0]]
C = [[1, 0]]
D = [[0]]

sys_ss = control.StateSpace(A, B, C, D)

print("State-Space Representation:")
print(sys_ss)
```

Task 6:
Conversion between Transfer Function and State Space

In control systems engineering, it is common to work with both transfer function and state space representations of dynamic systems. Python Control provides convenient functions to

convert between these representations. In this tutorial, we will learn how to convert a transfer function to a state space model and vice versa using Python Control.

Let's create a transfer function and a state space model to use in the conversion process:

```python
import control

num = [2]
den = [1, 3, 2]
tf = control.TransferFunction(num, den)
print("Transfer Function:")
print(tf)

A = [[-3, -2], [1, 0]]
B = [[-1], [0]]
C = [[0, -2]]
D = [[0]]
ss = control.StateSpace(A, B, C, D)
print("State Space Representation:")
print(ss)
```

This code will first define a transfer function with the numerator and denominator polynomials num and den. Then, it will convert the transfer function to state space using the `control.tf2ss()` function. The state space representation will be printed to the console. Finally, the state space representation will be converted back to a transfer function using the `control.ss2tf()`function. The transfer function representation will also be printed to the console.

The `control.tf2ss()` and `control.ss2tf()` functions in Python Control are very efficient and easy to use. They can be used to convert between transfer function and state space representations of control systems. To convert a transfer function to a state space model, use the `control.tf2ss()` method on the transfer function:

```python
ss_from_tf = control.tf2ss(tf)
print("State Space Representation from Transfer Function:")
print(ss_from_tf)
```

To convert a state space model to a transfer function, use the `control.ss2tf()` function:

```python
tf_from_ss = control.ss2tf(ss)
print("Transfer Function from State Space Representation:")
print(tf_from_ss)
```

Task 7:
Control System Input Signal

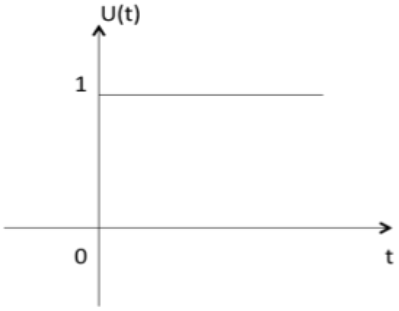
In control systems and signal processing, step, ramp, and sinusoidal signals are fundamental input signals used to analyze and understand the behavior of systems. Each signal has unique characteristics and is valuable for various applications. Here's a brief explanation of each signal:

In control systems engineering, analyzing the response of a system to different input signals is a crucial aspect of understanding its behavior and performance. Python Control provides a convenient way to generate and apply various input signals to control systems for simulation and analysis. In this tutorial, we will explore how to create and apply different types of control system input signals using Python Control.

1. Step Signal
   A step input is a common type of control system input signal, where the input changes abruptly from zero to a constant value at a specified time. The step signal, also known as the unit step function, is a simple and commonly used input signal in control systems. It is defined as:

$$u(t) = 0, for\ t < 0$$
$$u(t) = 1, for\ t >= 0$$

   The step signal has an abrupt change at $t = 0$, transitioning from 0 to 1. It represents an immediate change or activation in the system's behavior, often used to study transient responses and system stability.



   Let's create a step signal:

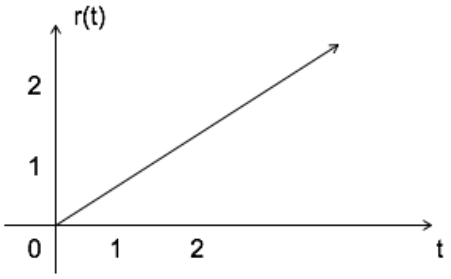
```python
import numpy as np
import matplotlib.pyplot as plt

t = np.linspace(0, 5, 1000)

step_signal = np.ones_like(t)
step_signal[t < 1] = 0

plt.plot(t, step_signal)
plt.xlabel('Time (seconds)')
plt.ylabel('Amplitude')
plt.title('Step Signal')
plt.grid(True)
plt.show()
```

2. Ramp Signal

A ramp input is another common control system input signal, where the input increases linearly with time. The ramp signal is another frequently used input signal in control systems and signal processing. It is defined as a linearly increasing function with time:

$$r(t) = 0, for t < 0$$
$$r(t) = t, for t >= 0$$

The ramp signal starts from 0 and continuously increases with time. It is used to analyze the steady-state response of systems and is particularly relevant in studying the response of integrators and systems with proportional control.



Let's create a ramp signal:

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

t = np.linspace(0, 5, 1000)

ramp_signal = t

plt.plot(t, ramp_signal)
plt.xlabel('Time (seconds)')
plt.ylabel('Amplitude')
plt.title('Ramp Signal')
plt.grid(True)
plt.show()
```

3. Sinusoidal Signal

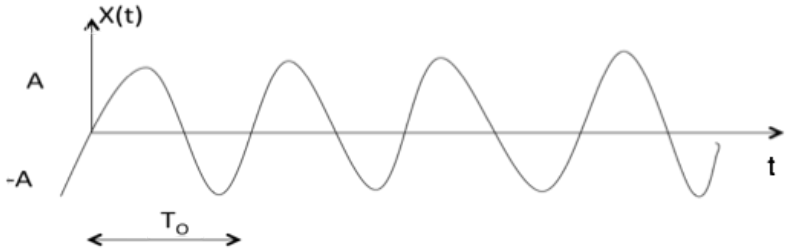
   A sinusoidal input is often used to analyze the frequency response of a control system. The sinusoidal signal is a periodic waveform represented by the sine or cosine function. It is defined as:

$$x(t) = A \times sin(2\pi ft + \phi)$$

   Where:

   - A is the amplitude of the sinusoid.
   - f is the frequency of the sinusoid in Hertz (Hz).
   - t is time.
   - ϕ is the phase shift.

   Sinusoidal signals are crucial for frequency response analysis and studying system dynamics in the frequency domain. They are used to analyze resonance, frequency-dependent behaviors, and system stability.

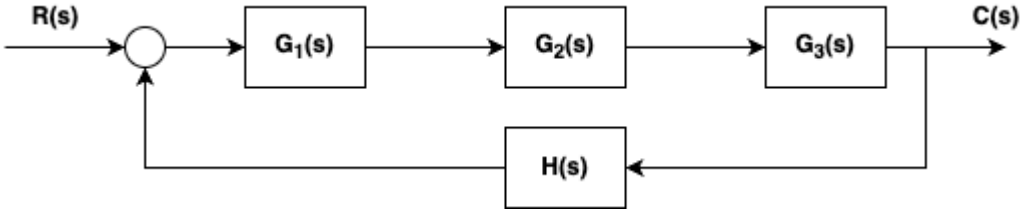


Let's create a sinusoidal signal:

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

t = np.linspace(0, 5, 1000)

sinus_signal = np.sin(2 * np.pi * t)

plt.plot(t, sinus_signal)
plt.xlabel('Time (seconds)')
plt.ylabel('Amplitude')
plt.title('Sinusoidal Signal')
plt.grid(True)
plt.show()
```

Assignment:

1. Write the python control code for the second order system transfer function below
   $H(s) = \frac{1}{s^2 + 0.2s + 1}$
2. Reduce the block diagram below using the code in python control



$G_1(s) = \frac{1}{(s+1)}$

$G2(s) = \frac{(s+2)}{(s^2+2s+1)}$

$G3(s) = \frac{1}{(s+3)}$

$H(s) = 2$

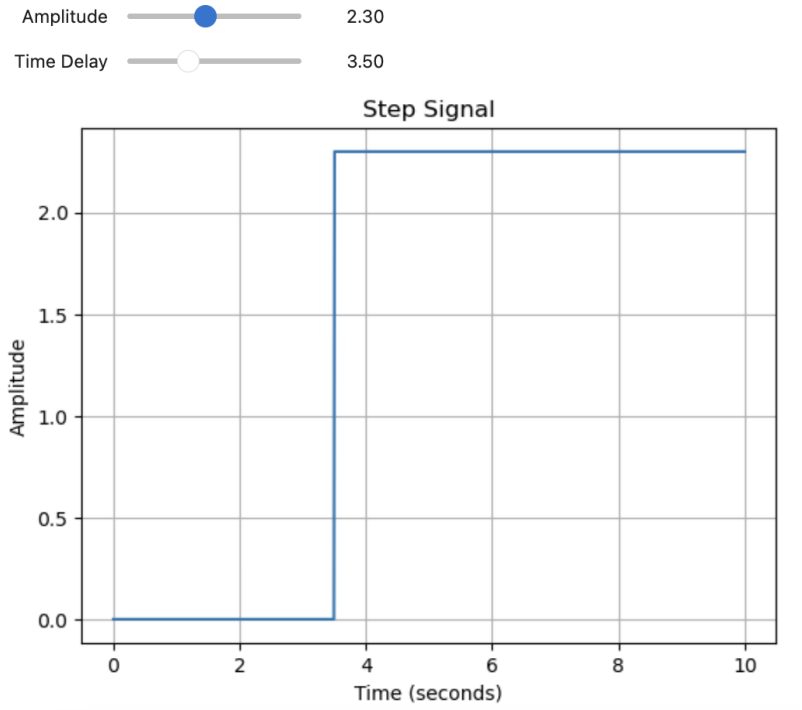
Result

$$\frac{C(s)}{R(s)} = \frac{s+2}{s^4+6s^3+12s^2+12s+7}$$

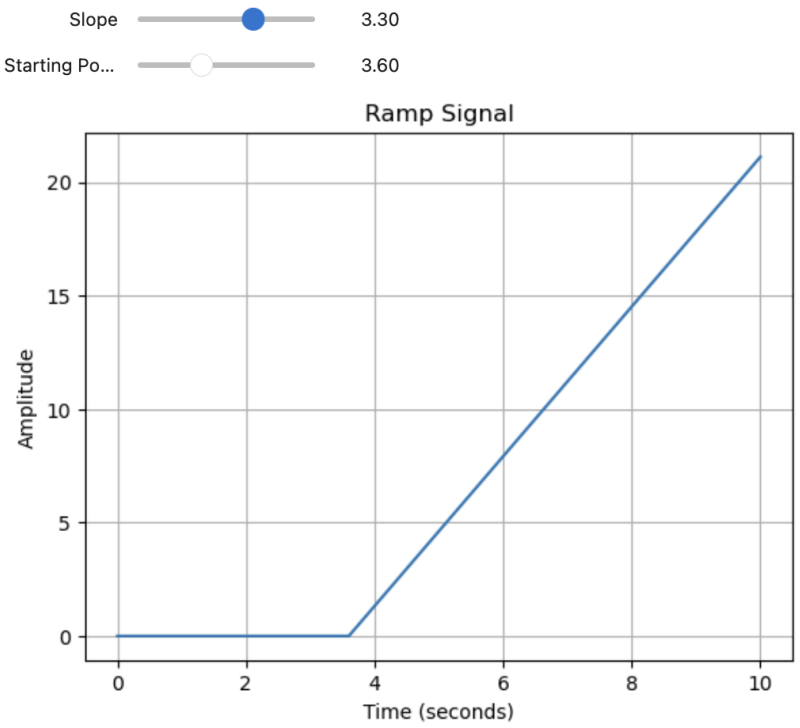
3. Convert the transfer function in number 1 into a state space representation.

```
State Space Representation from Transfer Function:
A = [[-0.2 -1. ]
     [ 1.   0. ]]

B = [[-1.]
     [ 0.]]

C = [[ 0. -1.]]

D = [[0.]]
```

4. Generate the step signal with adjustable amplitude and time delay. aPlot the step signal using interactive Jupyter Lab widgets.



5. Generate the ramp signal with adjustable slope and starting point. Plot the ramp signal using interactive Jupyter Lab widgets.


Slope
3.30
Starting Po...
3.60
Ramp Signal
Amplitude
Time (seconds)
0
5
10
15
20
0
2
4
6
8
10