Lab Units 11 & 12 Modeling Dynamic Systems Using Transfer Functions

- Overview and Components of the Robot Axis System
- Control Loop Interconnections and Feedback Mechanisms
- Transfer Functions for Dynamic System Analysis
- System Simulation and Input Response Types (Step, Ramp, Sinusoidal, Disturbances)
- Effects of PID Parameter Adjustments on System Stability and Performance

Overview of the Robot Axis System

A robotic axis is a complex electromechanical system that performs precise movements under varying load conditions. It consists of interconnected components working in harmony to achieve the desired performance.

Components and Their Roles

1. **Motor**:
   - **Function**: Converts electrical energy into mechanical energy to actuate the robotic arm.
   - **Dynamics**: Modeled as a first-order system with characteristics such as torque constant (K_t, back electromotive force (EMF), and electrical resistance.
   - **Key Parameters**:
     - Torque output (τ_m).
     - Rotational speed (ω_m).
2. **Gearbox**:
   - **Function**: Modifies the motor's speed and torque by a defined gear ratio (N).
   - **Dynamics**:
     - Reduces high-speed rotation from the motor to a slower, more powerful torque output.
     - Reflects the inertia of the load back to the motor.
   - **Key Parameters**:
     - Gear ratio (N).
     - Efficiency (η).
3. **PID Controller**:
   - **Function**: Ensures the system achieves precise and stable movements by minimizing the error between the desired and actual positions.
   - **Dynamics**:

- Implements proportional (K_p), integral (K_i), and derivative (K_d) control actions.

  - **Control Equation**:

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau) d\tau + K_d \frac{de(t)}{dt}$$

  where $e(t)$ is the error signal.

4. **Robot Arm**:
   - **Function**: Performs the physical movement and interacts with external forces.
   - **Dynamics**:
     - Introduces **variable torques** ($\tau_a(t)$) based on predefined laws (e.g., sinusoidal or step functions).
     - Acts as a load on the motor-gearbox system.
   - **Key Parameters**:
     - Arm inertia (J_a).
     - External torque law ($\tau_a(t)$).
5. **Sensor**:
   - **Function**: Measures the robotic axis's position, velocity, or torque and provides feedback to the PID controller.
   - **Dynamics**:
     - Often modeled as a first-order system with a time constant.
     - Assumes a linear response for simplicity in this lab work.

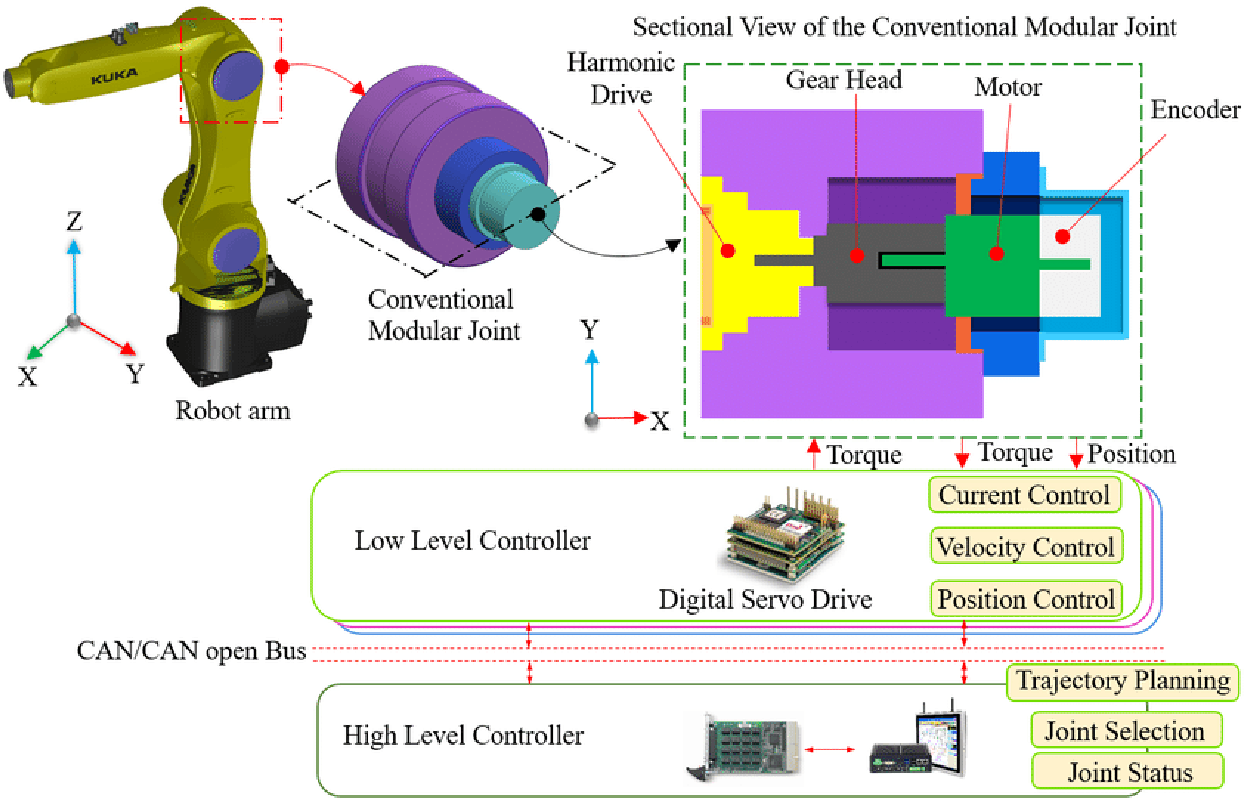
Control Loop Interconnections

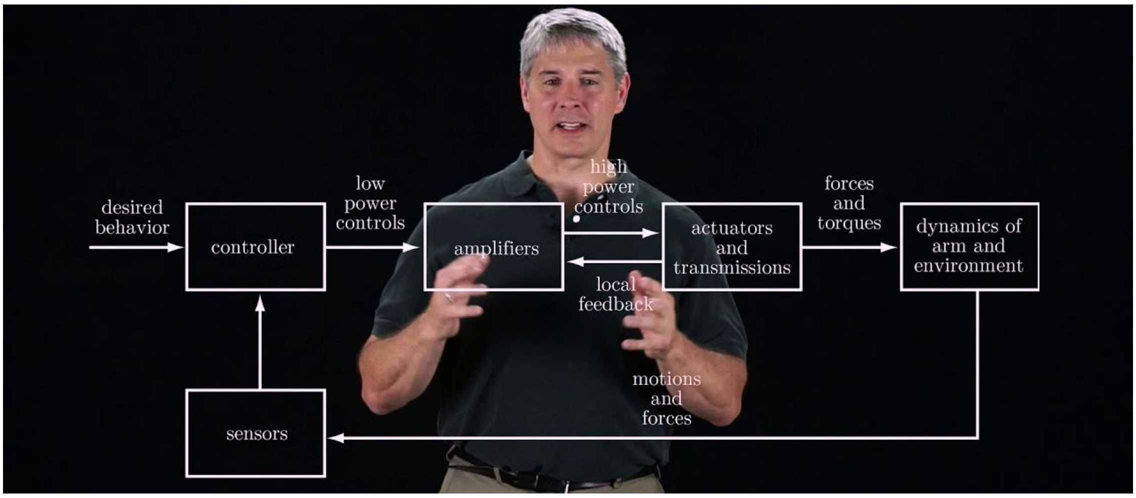
The robotic axis operates as a **closed-loop control system** with the following flow of information and energy:

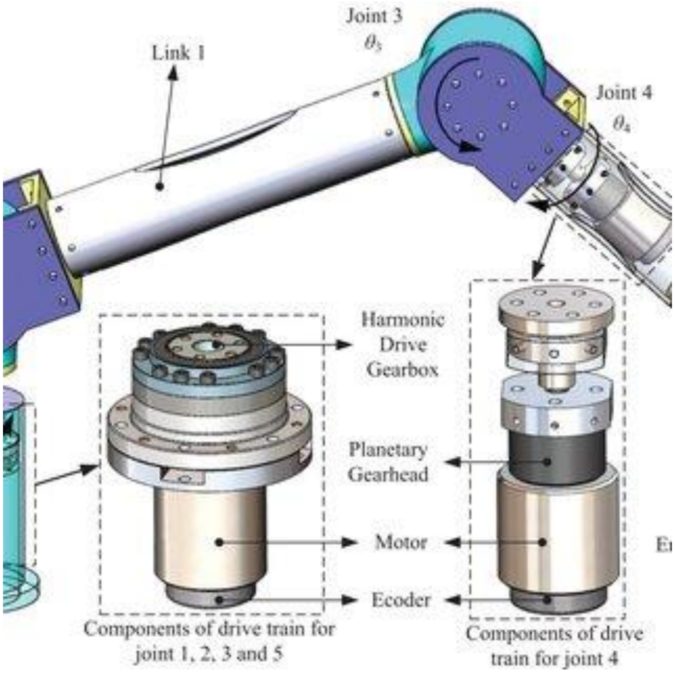
1. **Desired Input Signal** ($r(t)$):
   - The desired position or velocity of the robotic axis.
   - Serves as the reference for the PID controller.
2. **Controller Action**:
   - The PID controller processes the error signal ($e(t)=r(t)-y(t)$), where $y(t)$ is the actual output.

- Outputs a control signal ($u(t)$) to the motor.

3. **Motor and Gearbox**:
   - The motor converts the control signal ($u(t)$) into mechanical torque (τ_m).
   - The gearbox adjusts the torque and speed, transferring it to the robotic arm.
4. **Robot Arm and Disturbances**:
   - The arm executes movements while experiencing external disturbances and variable torques ($\tau_a(t)$).
   - The disturbances affect the system's stability and must be counteracted by the control system.
5. **Sensor Feedback**:
   - The sensor monitors the actual state of the robotic axis (e.g., position or velocity).
   - Sends feedback to the PID controller for continuous adjustment.







Mathematical Modeling of the Robot Axis System

1. Motor Dynamics

The motor converts electrical energy into mechanical energy. It can be modeled as a first-order system:

Equations of Motion:

1. **Electrical Dynamics**:

$$V_a(t) = I_a R_a + L_a \frac{dI_a}{dt} + e_b$$

where:

- $V_a(t)$: Applied voltage.
- I_a: Armature current.
- R_a, L_a: Resistance and inductance of the armature.
- $e_b = K_b \omega_m$: Back EMF, proportional to motor speed.

2. **Mechanical Dynamics**:

$$\tau_m = J_m \frac{d\omega_m}{dt} + b_m \omega_m$$

where:

- $\tau_m = K_t I_a$: Torque is proportional to armature current.
- J_m: Motor's moment of inertia.
- b_m: Viscous damping coefficient.

Transfer Function:

By combining electrical and mechanical dynamics:

$$G_m(s) = \frac{\omega_m(s)}{V_a(s)} = \frac{K_t}{(L_a J_m)s^2 + (R_a J_m + L_a b_m)s + (R_a b_m + K_t K_b)}$$

2. Gearbox Dynamics

The gearbox modifies speed and torque while reflecting the load inertia back to the motor.

Key Relationships:

- **Speed-Torque Conversion**:

$$\omega_{out} = \frac{\omega_m}{N}, \quad \tau_{out} = N\tau_m$$

- **Inertia Reflection**:

$$J_{ref} = J_a N^2$$

Updated Motor Dynamics:

Incorporating the reflected inertia, the total effective inertia is:

$$J_{eff} = J_m + J_{ref}$$

The transfer function is updated as:

$$G_{motor+gearbox}(s) = \frac{\omega_{out}(s)}{V_a(s)} = \frac{K_t/N}{(L_a J_{eff})s^2 + (R_a J_{eff} + L_a b_m)s + (R_a b_m + K_t K_b)}$$

3. PID Controller

The PID controller regulates the motor to minimize error. Its transfer function is:

$$G_c(s) = K_p + \frac{K_i}{s} + K_d s$$

Simplified Form:

In the Laplace domain:

$$G_c(s) = \frac{K_d s^2 + K_p s + K_i}{s}$$

4. Robot Arm Dynamics

The robot arm introduces variable torque as a disturbance. Its dynamics include:

- **Disturbance Torque**:

$$\tau_a(t) = A\sin(\omega t) \quad \text{(or another predefined law)}.$$

- **Dynamics**: The torque introduces a disturbance to the motor-gearbox system:

$$\tau_{disturbance}(s) = \frac{\tau_a(s)}{J_a s^2 + b_a s}$$

where J_a is the arm inertia and b_a is the damping coefficient.

5. Sensor Dynamics

The sensor measures position or velocity and provides feedback.

Simple Model:

If modeled as a gain:

$$G_{sensor}(s) = K_s$$

Dynamic Model (if needed):

Include a first-order response:

$$G_{sensor}(s) = \frac{K_s}{\tau_s s + 1}$$

6. Combined Transfer Function

The complete system can be modeled as a cascade of subsystems and the feedback loop:

Open-Loop Transfer Function:

$$G_{open}(s) = G_c(s) \cdot G_{motor+gearbox}(s) \cdot G_{sensor}(s)$$

Closed-Loop Transfer Function:

With feedback, the closed-loop transfer function becomes:

$$G_{closed}(s) = \frac{G_{open}(s)}{1 + G_{open}(s)}$$

Disturbance Transfer Function:

The impact of disturbance torque is:

$$G_{disturbance}(s) = \frac{G_{motor+gearbox}(s)}{1 + G_{open}(s)}$$

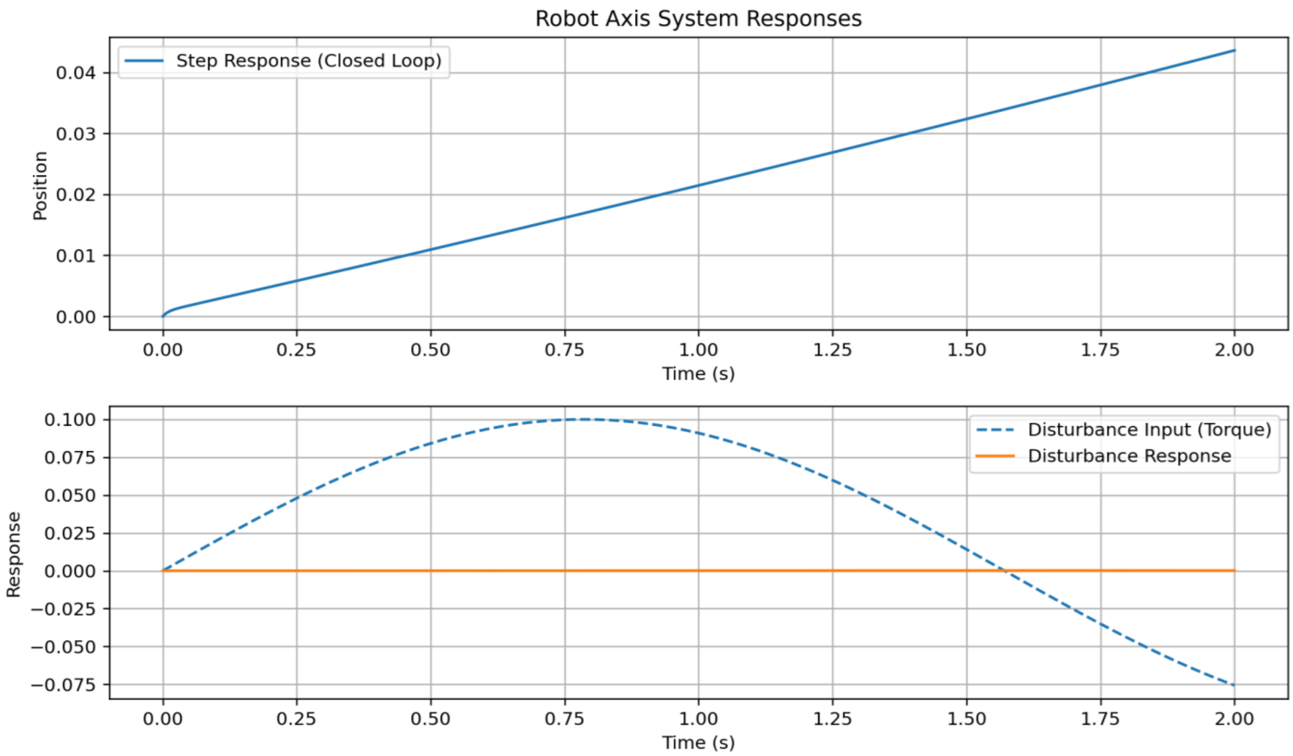
Final System Analysis

1. **Inputs**:
   - Desired position or velocity ($R(s)$).
   - Disturbance torque ($\tau_a(s)$).
2. **Outputs**:
   - Position or velocity of the robotic arm ($Y(s)$).
3. **Simulation Task**:
   - Simulate the response of $G_{closed}(s)$ to a step input.
   - Analyze the impact of $G_{disturbance}(s)$ for the predefined torque law.

```
import numpy as np
```

```python
import matplotlib.pyplot as plt
import control as ctrl

# Define system parameters
# Motor parameters
K_t = 0.1   # Torque constant
K_b = 0.1   # Back EMF constant
R_a = 1.0   # Armature resistance
L_a = 0.01  # Armature inductance
J_m = 0.01  # Motor inertia
b_m = 0.001 # Motor damping

# Gearbox parameters
N = 10      # Gear ratio
J_a = 0.05  # Arm inertia (robot arm)
b_a = 0.01  # Arm damping

# PID Controller parameters
K_p = 10.0  # Proportional gain
K_i = 1.0   # Integral gain
K_d = 0.5   # Derivative gain

# Sensor parameters
K_s = 1.0   # Sensor gain

# Disturbance torque parameters
A = 0.1     # Amplitude of sinusoidal disturbance
omega = 2.0 # Frequency of disturbance

# Effective inertia
J_eff = J_m + J_a * N**2

# Transfer function for the motor + gearbox
numerator_motor = [K_t / N]
denominator_motor = [
    L_a * J_eff,
    R_a * J_eff + L_a * b_m,
    R_a * b_m + K_t * K_b
]
G_motor = ctrl.TransferFunction(numerator_motor, denominator_motor)

# Transfer function for the PID controller
numerator_pid = [K_d, K_p, K_i]
denominator_pid = [1, 0]
G_pid = ctrl.TransferFunction(numerator_pid, denominator_pid)

# Transfer function for the sensor
G_sensor = ctrl.TransferFunction([K_s], [1])

# Open-loop transfer function
G_open = G_pid * G_motor * G_sensor

# Closed-loop transfer function
G_closed = ctrl.feedback(G_open, 1)

# Disturbance transfer function
G_disturbance = G_motor / (1 + G_open)

# Simulation: Step response for closed-loop system
time = np.linspace(0, 2, 500)  # 0 to 2 seconds, 500 points
time_out, response_closed = ctrl.step_response(G_closed, time)
```

```
# Simulation: Response to sinusoidal disturbance
disturbance_input = A * np.sin(omega * time)
time_out, response_disturbance = ctrl.forced_response(G_disturbance, time, disturbance_input)

# Plot the results
plt.figure(figsize=(10, 6))

# Plot closed-loop step response
plt.subplot(2, 1, 1)
plt.plot(time_out, response_closed, label='Step Response (Closed Loop)')
plt.title('Robot Axis System Responses')
plt.xlabel('Time (s)')
plt.ylabel('Position')
plt.legend()
plt.grid()

# Plot disturbance response
plt.subplot(2, 1, 2)
plt.plot(time, disturbance_input, label='Disturbance Input (Torque)', linestyle='dashed')
plt.plot(time_out, response_disturbance, label='Disturbance Response')
plt.xlabel('Time (s)')
plt.ylabel('Response')
plt.legend()
plt.grid()

plt.tight_layout()
plt.show()
```



How to Choose the Input Type

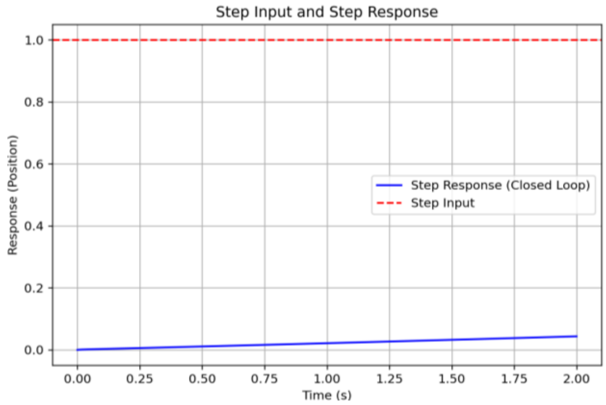
- Step Input: To analyze transient and steady-state behavior.
- Impulse Input: To study the natural dynamics and stability.
- Ramp Input: To test tracking ability for linearly changing signals.

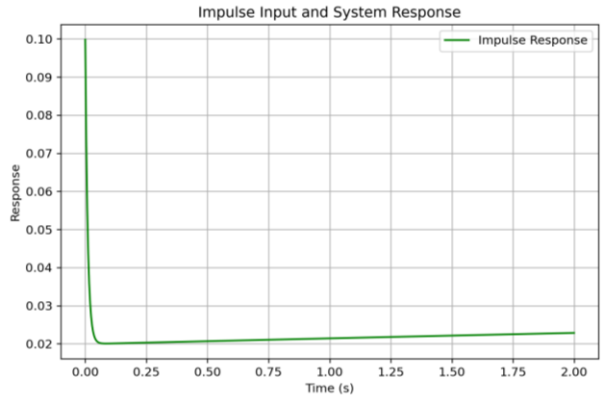
- Sinusoidal Input: To evaluate frequency response and resonance.
- Custom Input: For system-specific scenarios or complex real-world stimuli.

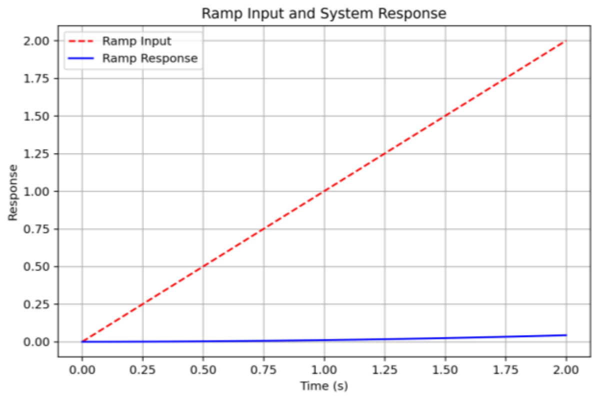
```
import numpy as np
import matplotlib.pyplot as plt
import control as ctrl

# Define system parameters
# Motor parameters
K_t = 0.1   # Torque constant
K_b = 0.1   # Back EMF constant
R_a = 1.0   # Armature resistance
L_a = 0.01  # Armature inductance
J_m = 0.01  # Motor inertia
b_m = 0.001 # Motor damping

# Gearbox parameters
N = 10      # Gear ratio
J_a = 0.05  # Arm inertia (robot arm)
b_a = 0.01  # Arm damping

# PID Controller parameters
K_p = 10.0  # Proportional gain
K_i = 1.0   # Integral gain
K_d = 0.5   # Derivative gain

# Sensor parameters
K_s = 1.0   # Sensor gain

# Effective inertia
J_eff = J_m + J_a * N**2

# Transfer function for the motor + gearbox
numerator_motor = [K_t / N]
denominator_motor = [
    L_a * J_eff,
    R_a * J_eff + L_a * b_m,
    R_a * b_m + K_t * K_b
]
G_motor = ctrl.TransferFunction(numerator_motor, denominator_motor)

# Transfer function for the PID controller
numerator_pid = [K_d, K_p, K_i]
denominator_pid = [1, 0]
G_pid = ctrl.TransferFunction(numerator_pid, denominator_pid)

# Transfer function for the sensor
G_sensor = ctrl.TransferFunction([K_s], [1])

# Open-loop transfer function
G_open = G_pid * G_motor * G_sensor

# Closed-loop transfer function
G_closed = ctrl.feedback(G_open, 1)

# Simulation: Step response for closed-loop system
time = np.linspace(0, 2, 500)  # 0 to 2 seconds, 500 points
time_out, response_closed = ctrl.step_response(G_closed, time)

# Plot the step input and response
plt.figure(figsize=(8, 5))
```

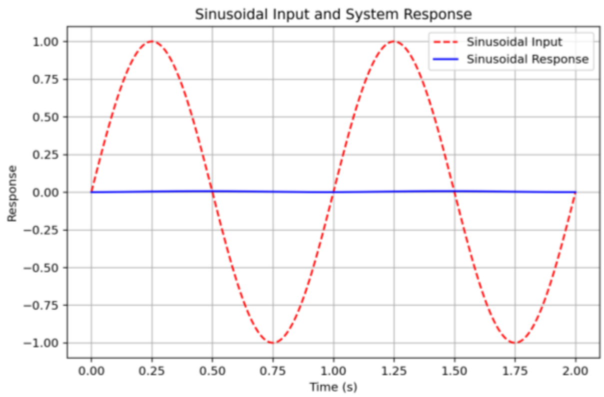
```
plt.plot(time_out, response_closed, label='Step Response (Closed Loop)', color='b')
plt.axhline(y=1, color='r', linestyle='--', label='Step Input')
plt.title('Step Input and Step Response')
plt.xlabel('Time (s)')
plt.ylabel('Response (Position)')
plt.legend()
plt.grid()
plt.show()

time_out, response_impulse = ctrl.impulse_response(G_closed, time)

plt.figure(figsize=(8, 5))
plt.plot(time_out, response_impulse, label='Impulse Response', color='g')
plt.title('Impulse Input and System Response')
plt.xlabel('Time (s)')
plt.ylabel('Response')
plt.legend()
plt.grid()
plt.show()

# Ramp input signal
ramp_input = time  # Linearly increasing input with time
time_out, response_ramp = ctrl.forced_response(G_closed, time, ramp_input)

# Plot ramp input and response
plt.figure(figsize=(8, 5))
plt.plot(time, ramp_input, label='Ramp Input', linestyle='dashed', color='r')
plt.plot(time_out, response_ramp, label='Ramp Response', color='b')
plt.title('Ramp Input and System Response')
plt.xlabel('Time (s)')
plt.ylabel('Response')
plt.legend()
plt.grid()
plt.show()

# Sinusoidal input signal
freq = 1.0  # Frequency of the sine wave (Hz)
sine_input = np.sin(2 * np.pi * freq * time)  # Sinusoidal input
time_out, response_sine = ctrl.forced_response(G_closed, time, sine_input)

# Plot sinusoidal input and response
plt.figure(figsize=(8, 5))
plt.plot(time, sine_input, label='Sinusoidal Input', linestyle='dashed', color='r')
plt.plot(time_out, response_sine, label='Sinusoidal Response', color='b')
plt.title('Sinusoidal Input and System Response')
plt.xlabel('Time (s)')
plt.ylabel('Response')
plt.legend()
plt.grid()
plt.show()

# Exponential input signal
exponential_input = np.exp(-time)  # Exponentially decaying input
time_out, response_exponential = ctrl.forced_response(G_closed, time, exponential_input)

# Plot exponential input and response
plt.figure(figsize=(8, 5))
plt.plot(time, exponential_input, label='Exponential Input', linestyle='dashed', color='r')
plt.plot(time_out, response_exponential, label='Exponential Response', color='b')
plt.title('Exponential Input and System Response')
plt.xlabel('Time (s)')
plt.ylabel('Response')
```

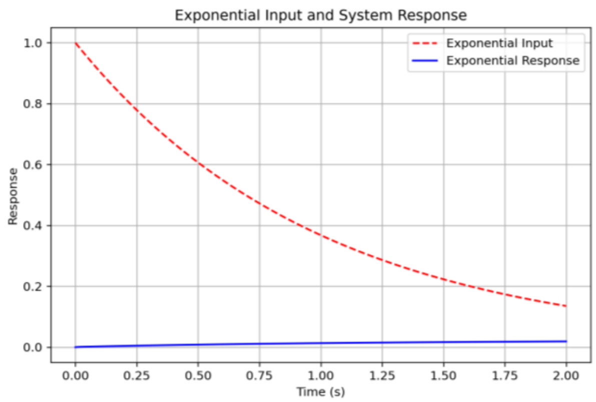
```python
plt.legend()
plt.grid()
plt.show()

from scipy.signal import square

# Square wave input signal
freq = 1.0  # Frequency of the square wave (Hz)
square_input = square(2 * np.pi * freq * time)  # Square wave signal
time_out, response_square = ctrl.forced_response(G_closed, time, square_input)

# Plot square wave input and response
plt.figure(figsize=(8, 5))
plt.plot(time, square_input, label='Square Wave Input', linestyle='dashed', color='r')
plt.plot(time_out, response_square, label='Square Wave Response', color='b')
plt.title('Square Wave Input and System Response')
plt.xlabel('Time (s)')
plt.ylabel('Response')
plt.legend()
plt.grid()
plt.show()
```

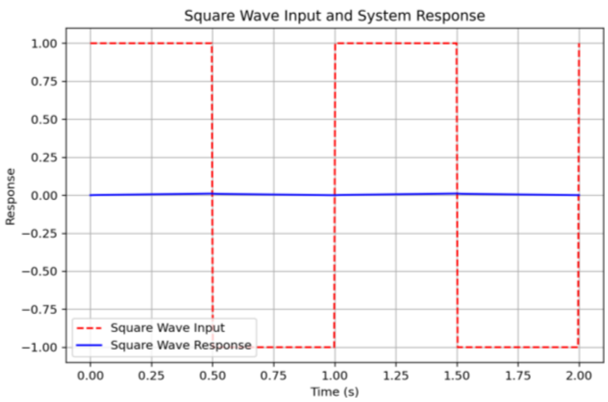












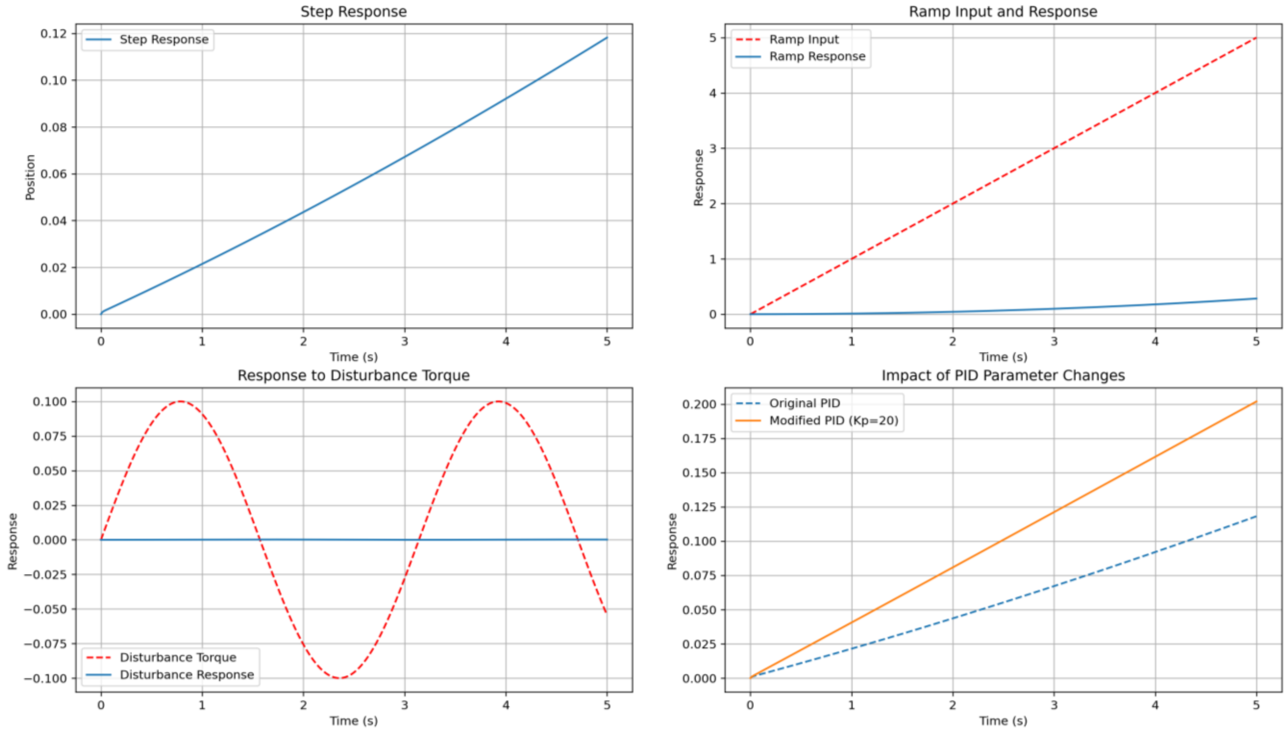
Simulating the Model

- Use a simulation tool (you can choose from MATLAB Simulink, Python, or others).
- Steps:
  - Build the system diagram.
  - Implement each component with appropriate parameters.
  - Add the disturbance torque $\tau(t)$.
  - Apply a step or ramp input to observe system behavior.
- Simulate the system's response to:
  - Disturbance torques.
  - PID parameter changes.

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

# Define system parameters
# Motor parameters
K_t = 0.1   # Torque constant
K_b = 0.1   # Back EMF constant
R_a = 1.0   # Armature resistance
L_a = 0.01  # Armature inductance
J_m = 0.01  # Motor inertia
b_m = 0.001 # Motor damping

# Gearbox parameters
N = 10      # Gear ratio
J_a = 0.05  # Arm inertia (robot arm)
b_a = 0.01  # Arm damping

# PID Controller parameters
K_p = 10.0  # Proportional gain
K_i = 1.0   # Integral gain
K_d = 0.5   # Derivative gain

# Disturbance torque parameters
A = 0.1     # Amplitude of sinusoidal disturbance
omega = 2.0 # Frequency of disturbance

# Effective inertia
J_eff = J_m + J_a * N**2

# Transfer function for the motor + gearbox
numerator_motor = [K_t / N]
denominator_motor = [
    L_a * J_eff,
    R_a * J_eff + L_a * b_m,
    R_a * b_m + K_t * K_b
]
G_motor = ctrl.TransferFunction(numerator_motor, denominator_motor)

# Transfer function for the PID controller
numerator_pid = [K_d, K_p, K_i]
```

```python
denominator_pid = [1, 0]
G_pid = ctrl.TransferFunction(numerator_pid, denominator_pid)

# Open-loop transfer function
G_open = G_pid * G_motor

# Closed-loop transfer function
G_closed = ctrl.feedback(G_open, 1)

# Simulation setup
time = np.linspace(0, 5, 500)  # 0 to 5 seconds, 500 points

# Step Input
time_step, response_step = ctrl.step_response(G_closed, time)

# Ramp Input
ramp_input = time  # Linearly increasing input
time_ramp, response_ramp = ctrl.forced_response(G_closed, time, ramp_input)

# Sinusoidal Disturbance Torque
disturbance_torque = A * np.sin(omega * time)
G_disturbance = G_motor / (1 + G_open)
time_dist, response_disturbance = ctrl.forced_response(G_disturbance, time, disturbance_torque)

# PID Parameter Changes (Example: Increase K_p)
K_p_new = 20.0  # New proportional gain
G_pid_new = ctrl.TransferFunction([K_d, K_p_new, K_i], [1, 0])
G_open_new = G_pid_new * G_motor
G_closed_new = ctrl.feedback(G_open_new, 1)
time_pid, response_pid = ctrl.step_response(G_closed_new, time)

# Plotting the results
plt.figure(figsize=(12, 10))

# Step Response
plt.subplot(2, 2, 1)
plt.plot(time_step, response_step, label='Step Response')
plt.title('Step Response')
plt.xlabel('Time (s)')
plt.ylabel('Position')
plt.legend()
plt.grid()

# Ramp Response
plt.subplot(2, 2, 2)
plt.plot(time, ramp_input, label='Ramp Input', linestyle='dashed', color='r')
plt.plot(time_ramp, response_ramp, label='Ramp Response')
plt.title('Ramp Input and Response')
plt.xlabel('Time (s)')
plt.ylabel('Response')
plt.legend()
plt.grid()

# Disturbance Response
plt.subplot(2, 2, 3)
plt.plot(time, disturbance_torque, label='Disturbance Torque', linestyle='dashed', color='r')
plt.plot(time_dist, response_disturbance, label='Disturbance Response')
plt.title('Response to Disturbance Torque')
plt.xlabel('Time (s)')
plt.ylabel('Response')
plt.legend()
plt.grid()
```

```python
# Step Response with Modified PID
plt.subplot(2, 2, 4)
plt.plot(time_step, response_step, label='Original PID', linestyle='dashed')
plt.plot(time_pid, response_pid, label='Modified PID (Kp=20)')
plt.title('Impact of PID Parameter Changes')
plt.xlabel('Time (s)')
plt.ylabel('Response')
plt.legend()
plt.grid()

plt.tight_layout()
plt.show()
```



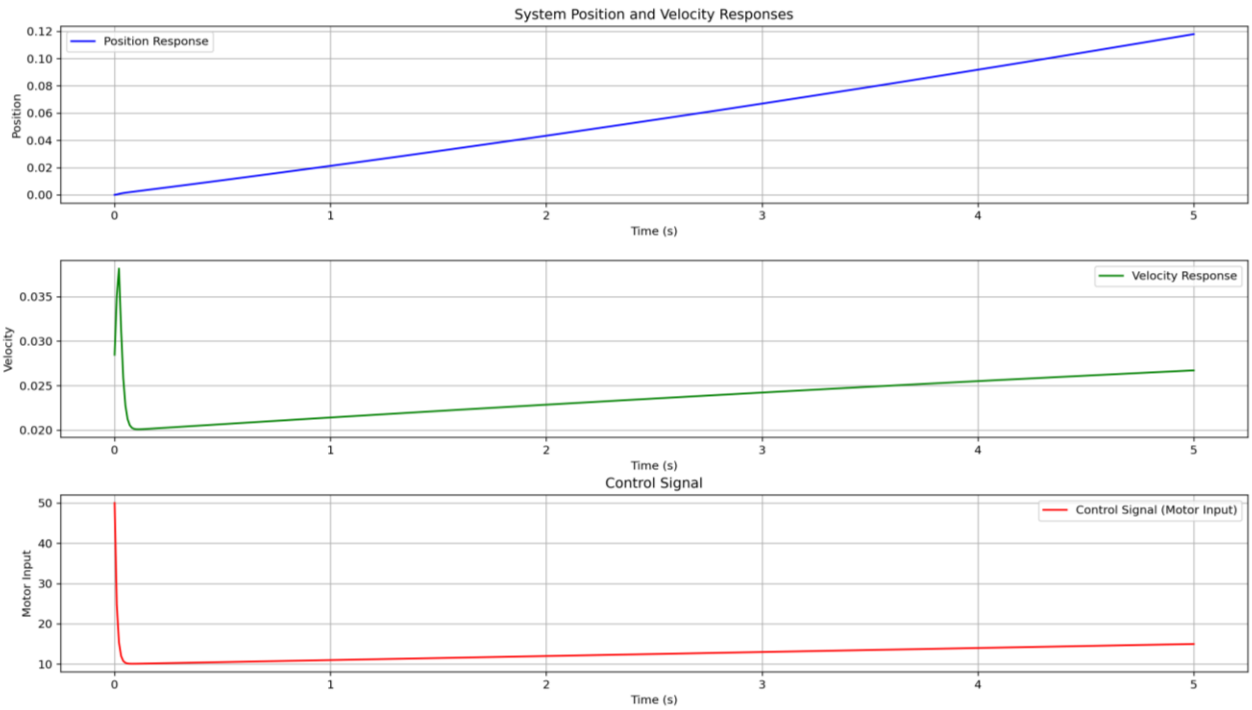
Analyzing Results

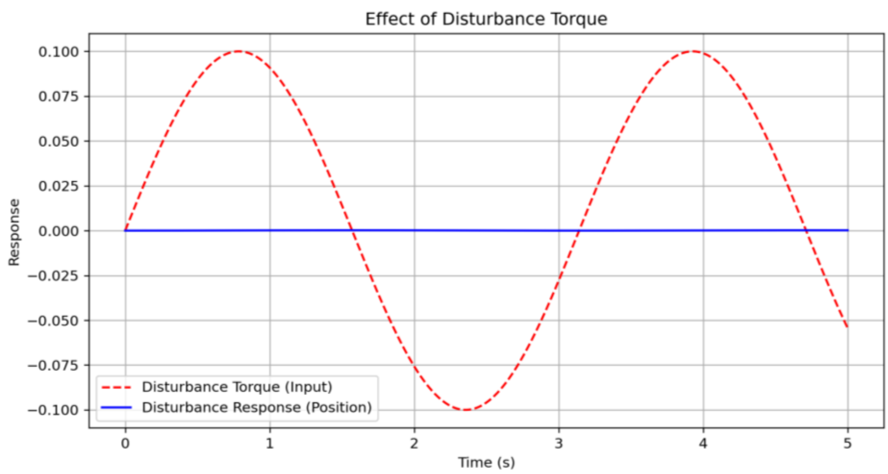
- Plot:
  - System response (position, velocity).
  - Control signal (motor input).
  - Effect of disturbance torque.
- Discuss:
  - How different PID gains affect system stability.
  - The impact of the gearbox ratio on performance.
  - The influence of the predefined torque law.

```python
import numpy as np
```

```python
import matplotlib.pyplot as plt
import control as ctrl

# System Parameters
K_t = 0.1   # Torque constant
K_b = 0.1   # Back EMF constant
R_a = 1.0   # Armature resistance
L_a = 0.01  # Armature inductance
J_m = 0.01  # Motor inertia
b_m = 0.001 # Motor damping
N = 10      # Gear ratio
J_a = 0.05  # Arm inertia
b_a = 0.01  # Arm damping
K_p = 10.0  # Proportional gain
K_i = 1.0   # Integral gain
K_d = 0.5   # Derivative gain
A = 0.1     # Disturbance amplitude
omega = 2.0 # Disturbance frequency

# Effective inertia
J_eff = J_m + J_a * N**2

# Motor + Gearbox Transfer Function
numerator_motor = [K_t / N]
denominator_motor = [L_a * J_eff, R_a * J_eff + L_a * b_m, R_a * b_m + K_t * K_b]
G_motor = ctrl.TransferFunction(numerator_motor, denominator_motor)

# PID Controller Transfer Function (made proper)
epsilon = 0.01  # Small time constant
numerator_pid = [K_d, K_p, K_i]
denominator_pid = [epsilon, 1, 0]
G_pid = ctrl.TransferFunction(numerator_pid, denominator_pid)

# Open-loop and Closed-loop Transfer Functions
G_open = G_pid * G_motor
G_closed = ctrl.feedback(G_open, 1)

# Simulation Setup
time = np.linspace(0, 5, 500)  # 0 to 5 seconds, 500 points

# 1. Step Response
time_step, response_step = ctrl.step_response(G_closed, time)

# 2. Ramp Input Response
ramp_input = time  # Linearly increasing input
time_ramp, response_ramp = ctrl.forced_response(G_closed, time, ramp_input)

# 3. Control Signal (Motor Input)
step_input = np.ones_like(time)  # Step input of magnitude 1
_, control_signal = ctrl.forced_response(G_pid, time, step_input)

# 4. Disturbance Torque Response
disturbance_torque = A * np.sin(omega * time)
G_disturbance = G_motor / (1 + G_open)
time_dist, response_disturbance = ctrl.forced_response(G_disturbance, time, disturbance_torque)

# 5. Velocity Response (Derivative of Position)
velocity_response = np.gradient(response_step, time)

# Plotting Results
plt.figure(figsize=(14, 10))
```

```python
# Plot Step Response
plt.subplot(3, 1, 1)
plt.plot(time_step, response_step, label='Position Response', color='b')
plt.title('System Position and Velocity Responses')
plt.xlabel('Time (s)')
plt.ylabel('Position')
plt.legend()
plt.grid()

# Plot Velocity Response
plt.subplot(3, 1, 2)
plt.plot(time_step, velocity_response, label='Velocity Response', color='g')
plt.xlabel('Time (s)')
plt.ylabel('Velocity')
plt.legend()
plt.grid()

# Plot Control Signal
plt.subplot(3, 1, 3)
plt.plot(time, control_signal, label='Control Signal (Motor Input)', color='r')
plt.title('Control Signal')
plt.xlabel('Time (s)')
plt.ylabel('Motor Input')
plt.legend()
plt.grid()

plt.tight_layout()
plt.show()

# Plot Disturbance Torque Effect
plt.figure(figsize=(10, 5))
plt.plot(time, disturbance_torque, label='Disturbance Torque (Input)', linestyle='dashed', color='r')
plt.plot(time_dist, response_disturbance, label='Disturbance Response (Position)', color='b')
plt.title('Effect of Disturbance Torque')
plt.xlabel('Time (s)')
plt.ylabel('Response')
plt.legend()
plt.grid()
plt.show()
```





Individual Work

- Modify the predefined torque law (e.g., replace sinusoidal with random impulses).
- Adjust PID gains to optimize system behavior.
- Experiment with different gearbox ratios.
- Document findings in a short report.