Lab Units 13 & 14 Digital Twin for PID Controller Tuning (3D Printer Application)

- Introduction to PID Controllers and Their Role in 3D Printing
- Transfer Function Modeling of Heating Systems in 3D Printers
- Developing and Simulating a Digital Twin of a PID-Controlled System
- Manual and Automated PID Tuning Methods
- Advanced Optimization Techniques (e.g., Particle Swarm Optimization, Differential Evolution, Neuro-fuzzy)

Objective To understand how to use digital twin techniques to tune a PID controller for the extruder nozzle temperature and bed temperature in a 3D printer.

1. Introduction to PID Controllers and Applications

What is a PID Controller?

A **PID (Proportional-Integral-Derivative)** controller is a feedback control loop mechanism widely used in industrial systems. It continuously calculates the error between a desired setpoint (target value) and the actual output of a system, applying corrections through three components:

1. **Proportional (P):** Reacts to the current error.
2. **Integral (I):** Accounts for accumulated past errors.
3. **Derivative (D):** Predicts future errors based on the rate of change.

The Role of PID Controllers in 3D Printing

In 3D printers, PID controllers are crucial for maintaining consistent performance and ensuring high-quality outputs. **They are embedded in the printer's firmware** to regulate critical subsystems, ensuring the following:

Maintaining Precise Nozzle and Bed Temperature

The extruder nozzle and heated bed must operate within precise temperature ranges to:

- Melt the filament properly for extrusion.
- Ensure good adhesion of the first layer to the print bed.
- Prevent thermal inconsistencies that can lead to defects like warping or under-extrusion.

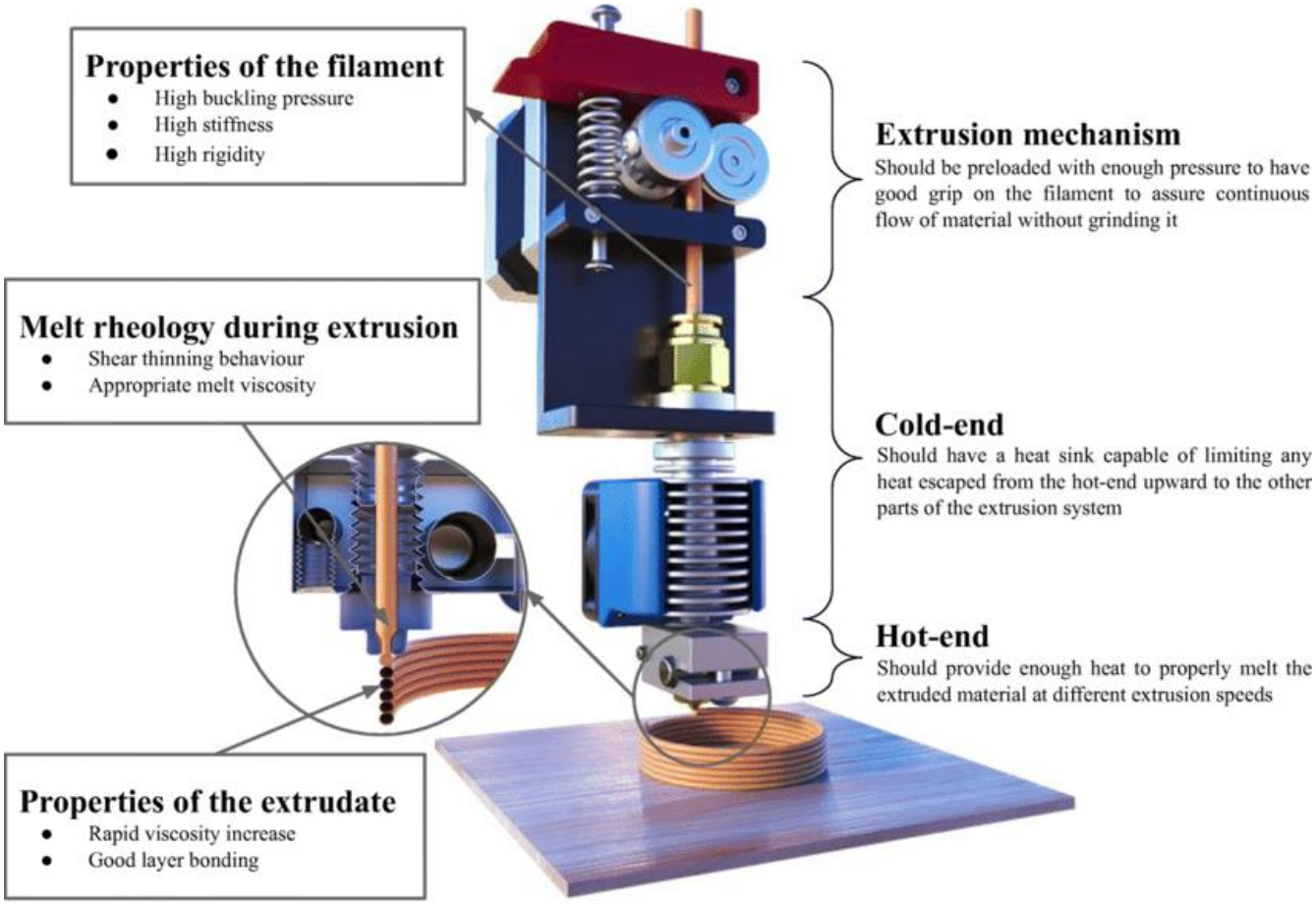
Controlling Axis Movements

- The X, Y, and Z axes and the extruder motor require precise positioning to create accurate prints.
- PID controllers ensure smooth and precise movements of the stepper motors, minimizing vibrations and positional errors.

Applications of PID Controllers in 3D Printing

Extruder Nozzle Heating System

The **extruder nozzle heating system** uses a PID controller to maintain a stable temperature, which is critical for melting filament.



The PID loop:

- Measures the nozzle temperature using a **thermistor** or **thermocouple**.
- Adjusts the power supplied to the heating element based on the temperature error.
- Ensures a consistent temperature despite fluctuations in ambient conditions or heat loss during extrusion.

Importance:

- Prevents filament clogging due to insufficient heating.
- Avoids overheating that could degrade the filament material.
- Provides stable extrusion for smooth layer deposition.

Heated Bed for Consistent Adhesion

The **heated bed** is another critical component where PID control is applied. It ensures a stable surface temperature to improve adhesion and reduce warping. The PID controller:

- Monitors the bed temperature using a thermistor.
- Adjusts the power to the bed heater to maintain the target temperature.
- Responds to external factors like airflow or cooling fan effects that can cause temperature drops.

Importance:

- Ensures proper adhesion of the first layer, crucial for print success.
- Prevents warping of larger parts by maintaining uniform temperature distribution across the bed.

How PID Controllers Work in These Applications

The firmware of 3D printers, such as **Marlin**, includes PID control algorithms for heating. Key parameters to tune are:

1. **K_p**: Proportional gain to address immediate errors.
2. **K_i**: Integral gain to eliminate residual error over time.
3. **K_d**: Derivative gain to dampen oscillations and prevent overshoot.

The PID loop operates in real time to adjust the heater's power output:

- If the temperature is below the setpoint, the heater power is increased.
- If the temperature is above the setpoint, the heater power is decreased or turned off.

Common Challenges and Solutions

1. **Overshoot and Oscillations**:

- When K_p is too high, the temperature may overshoot and oscillate around the setpoint.
- Solution: Adjust K_d to dampen the response.

2. **Slow Response**:

- If K_i is too low, the system takes longer to reach the setpoint.
- Solution: Increase K_i to address accumulated errors more aggressively.

3. **Environmental Factors**:

- Ambient temperature changes or drafts can affect stability.
- Solution: Optimize PID parameters and use enclosures to isolate the printer.

Why PID Tuning Matters for 3D Printing

- A poorly tuned PID controller can result in inconsistent print quality due to temperature fluctuations or delayed responses.

- Proper PID tuning ensures:
    - Smooth extrusion.
    - Strong layer adhesion.
    - Reduced defects like warping, under-extrusion, or stringing.

By maintaining precise temperature and smooth movement, PID controllers are essential for the reliability and efficiency of 3D printers.

2. Understanding the Transfer Function of a 3D Printer's Heating System

Dynamics of the Extruder and Heated Bed

In a 3D printer, the heating systems (extruder and heated bed) are critical for maintaining proper temperatures. These systems can be modeled using control theory principles to understand and predict their behavior under varying conditions.

Heat Transfer Mechanisms

Input Power vs. Thermal Dissipation

1. **Input Power**:
    - The heater applies electrical power (P_{in}) to increase the temperature.
    - Power is directly proportional to the voltage and current supplied to the heater ($P=V \times I$).
2. **Thermal Dissipation**:
    - Heat loss occurs due to conduction, convection, and radiation.
        - **Conduction**: Heat transfer through the material of the nozzle or heated bed.
        - **Convection**: Heat lost to the surrounding air, influenced by ambient temperature and airflow.
        - **Radiation**: Heat emitted from the surface, proportional to the temperature difference between the system and its environment.
3. **Thermal Dynamics**:
    - The temperature T(t) increases as heat input exceeds dissipation and stabilizes when the system reaches thermal equilibrium.

Simplified First-Order or Second-Order System Modeling

Why Use First-Order Models?

- For most practical applications, the heating dynamics of the extruder nozzle and heated bed can be approximated by a **first-order system**. This simplification assumes:

- A single dominant thermal mass.
- A linear relationship between input power and temperature response.

System Assumptions:

- The heater has a finite thermal capacity.
- The thermal resistance and capacitance are constant over the operating range.

First-Order Model Representation

The temperature response T(t) of the heating system can be described by:

$$\tau \frac{dT(t)}{dt} + T(t) = K \cdot P_{in}(t)$$

where:

- τ: Time constant (how quickly the system responds to changes in input power).
- K: System gain (temperature change per unit power input).
- $P_{in}(t)$: Input power.

Transfer Function of the Heating System

The transfer function H(s) is the Laplace transform representation of the system's behavior, relating the output temperature T(s) to the input power $P_{in}(s)$. For a first-order system, it is expressed as:

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

Parameters:

- **System Gain (K)**:
  - Defines the maximum achievable temperature change per unit of power input.
  - Example: For an extruder, K = 10°C/W
- **Time Constant (τ)**:
  - Represents the time it takes for the temperature to reach approximately 63% of its final value after a step change in input power.
  - Example: For a heated bed, τ=30 s.

Interpretation of H(s):

- At steady state (s=0):

$$H(0) = \frac{K}{1} = K$$

This indicates the system gain.

At high frequencies (s→∞):

$$H(s) \to 0$$

The system cannot respond instantaneously to rapid changes in input.

Example: Extruder Heating System

Suppose:

- K = 5°C/W
- τ=10 s

The transfer function for the extruder would be:

$$H(s) = \frac{5}{10s + 1}$$

If a step input of 10 W is applied, the temperature response T(t) is:

$$T(t) = K \cdot P_{in} \cdot \left(1 - e^{-\frac{t}{\tau}}\right)$$

$$T(t) = 5 \cdot 10 \cdot \left(1 - e^{-\frac{t}{10}}\right)$$

Second-order System for Advanced Models

For more complex scenarios, such as accounting for thermal inertia or additional heat sources, a second-order model may be used:

$$H(s) = \frac{K}{\tau^2 s^2 + 2\zeta\tau s + 1}$$

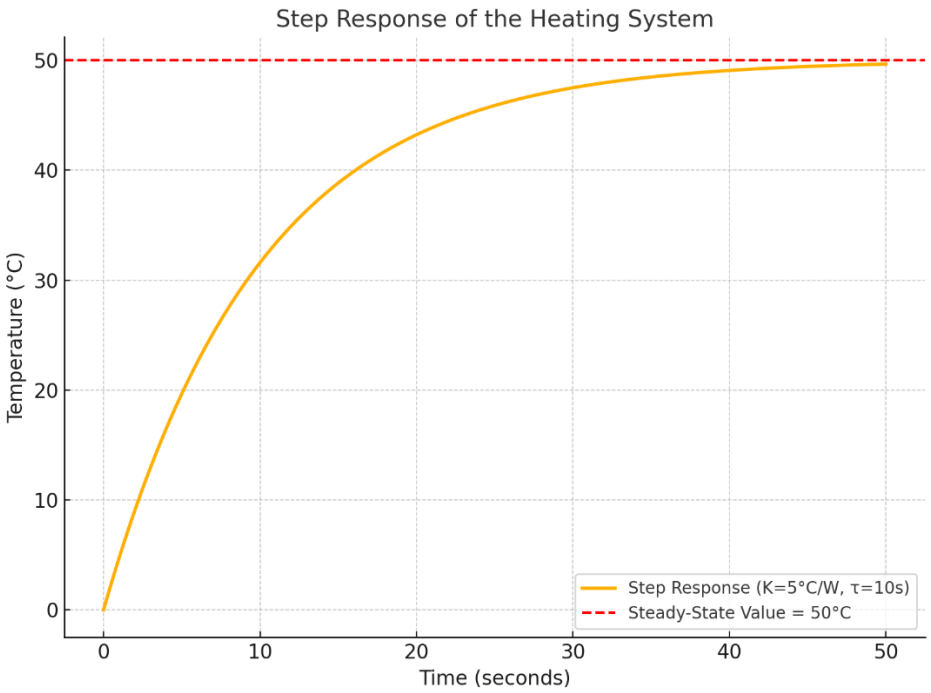
where:

- ζ: Damping ratio, indicating how oscillatory the response is.
- τ: Dominant time constant.

Practical Implications

Understanding and modeling the heating system's transfer function allows for:

- **PID Tuning**: Optimize K_p, K_i, K_d to achieve faster response times with minimal overshoot.
- **Simulation**: Predict system behavior before real-world implementation.
- **Diagnostics**: Identify and address inefficiencies in the heating system.



3. Creating a Digital Twin for the Heating System

- Develop a **digital twin** using Python:
  - Input: Power supplied to the heater.
  - Output: Temperature of the extruder nozzle or heated bed.
- Steps:
  - Model the heating dynamics with the given transfer function.
  - Simulate the system's behavior with different power inputs.

Input: Power levels of 5W, 10W, and 15W.

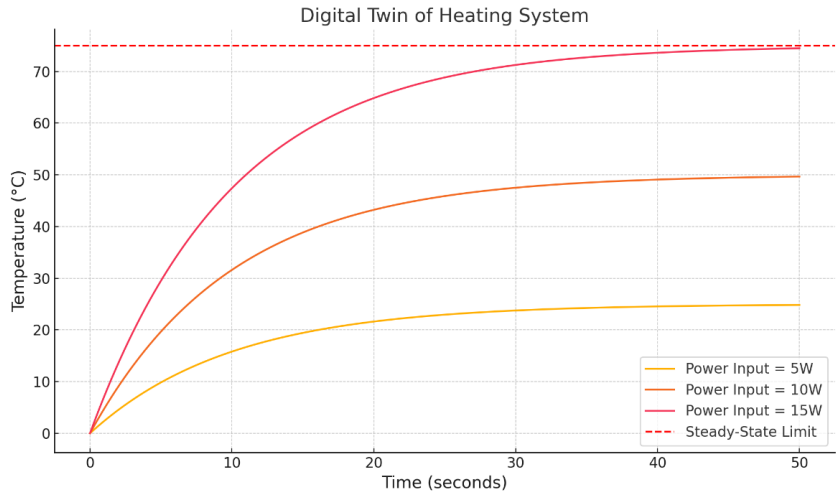
Output: The temperature of the extruder nozzle or heated bed over time.

Observations:

- For each power input, the temperature increases rapidly at first and then stabilizes at a steady-state value determined by K×Power Input.
- The time constant (τ) determines how quickly the system approaches steady-state.

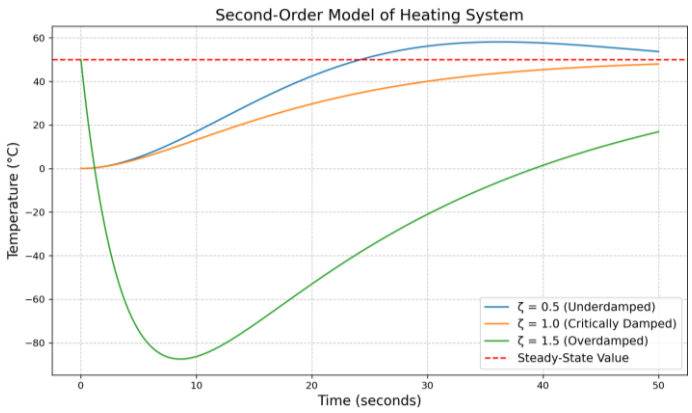
```
import numpy as np
import matplotlib.pyplot as plt

# Define the digital twin model for the heating system
def heating_system_model(time, K, tau, power_input):
    """
    Simulate the temperature response of the heating system.

    Parameters:
    time (array): Time vector in seconds.
    K (float): System gain (°C/W).
```

```
    tau (float): Time constant in seconds.
    power_input (float): Power input in watts.

    Returns:
    array: Temperature response over time.
    """
    return K * power_input * (1 - np.exp(-time / tau))  # First-order system equation

# Simulation parameters
time = np.linspace(0, 50, 500)  # Time vector (0 to 50 seconds, 500 points)
K = 5  # System gain (°C/W)
tau = 10  # Time constant (seconds)
power_inputs = [5, 10, 15]  # Different power levels (in watts)

# Simulate temperature responses for different power inputs
responses = [heating_system_model(time, K, tau, P) for P in power_inputs]

# Plot the results
plt.figure(figsize=(10, 6))
for power, response in zip(power_inputs, responses):
    plt.plot(time, response, label=f'Power Input = {power}W')

# Adding plot details
plt.title('Digital Twin of Heating System', fontsize=16)
plt.xlabel('Time (seconds)', fontsize=14)
plt.ylabel('Temperature (°C)', fontsize=14)
plt.axhline(y=K * max(power_inputs), color='r', linestyle='--', label='Steady-State Limit')
plt.legend(fontsize=12)
plt.grid(True, linestyle='--', alpha=0.7)
plt.tight_layout()
plt.show()
```



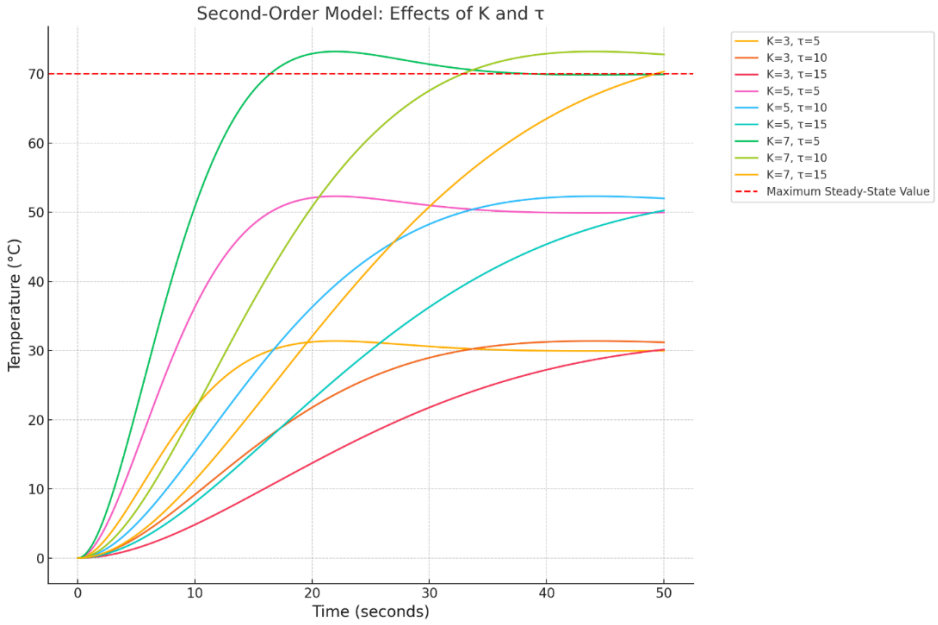

```
import numpy as np
import matplotlib.pyplot as plt

# Define the second-order system model
def second_order_model(time, K, tau, zeta, power_input):
    """
    Simulate the temperature response of a second-order heating system.

    Parameters:
    time (array): Time vector in seconds.
    K (float): System gain (°C/W).
    tau (float): Time constant (seconds).
    zeta (float): Damping ratio.
    power_input (float): Power input in watts.
```

```
    Returns:
    array: Temperature response over time.
    """
    omega_n = 1 / tau  # Natural frequency (rad/s)
    omega_d = omega_n * np.sqrt(1 - zeta**2) if zeta < 1 else 0  # Damped frequency
    response = np.zeros_like(time)

    for i, t in enumerate(time):
        if zeta < 1:  # Underdamped case
            response[i] = K * power_input * (1 - np.exp(-zeta * omega_n * t) * (
                np.cos(omega_d * t) + (zeta / np.sqrt(1 - zeta**2)) * np.sin(omega_d * t)
            ))
        elif zeta == 1:  # Critically damped case
            response[i] = K * power_input * (1 - (1 + omega_n * t) * np.exp(-omega_n * t))
        else:  # Overdamped case
            r1 = -omega_n * (zeta - np.sqrt(zeta**2 - 1))
            r2 = -omega_n * (zeta + np.sqrt(zeta**2 - 1))
            response[i] = K * power_input * (1 - (np.exp(r1 * t) - np.exp(r2 * t)) / (r1 - r2))
    return response

# Simulation parameters
time = np.linspace(0, 50, 500)  # Time vector (0 to 50 seconds)
K = 5  # System gain (°C/W)
tau = 10  # Time constant (seconds)
zeta_values = [0.5, 1.0, 1.5]  # Damping ratios for underdamped, critically damped, and overdamped cases
power_input = 10  # Power input (W)

# Simulate responses for different damping ratios
responses = [second_order_model(time, K, tau, zeta, power_input) for zeta in zeta_values]

# Plot the results
plt.figure(figsize=(10, 6))
for zeta, response in zip(zeta_values, responses):
    label = f"ζ = {zeta} (Underdamped)" if zeta < 1 else (
        f"ζ = {zeta} (Critically Damped)" if zeta == 1 else f"ζ = {zeta} (Overdamped)"
    )
    plt.plot(time, response, label=label)

# Add plot details
plt.title('Second-Order Model of Heating System', fontsize=16)
plt.xlabel('Time (seconds)', fontsize=14)
plt.ylabel('Temperature (°C)', fontsize=14)
plt.axhline(y=K * power_input, color='r', linestyle='--', label='Steady-State Value')
plt.grid(True, linestyle='--', alpha=0.7)
plt.legend(fontsize=12)
plt.tight_layout()
plt.show()
```



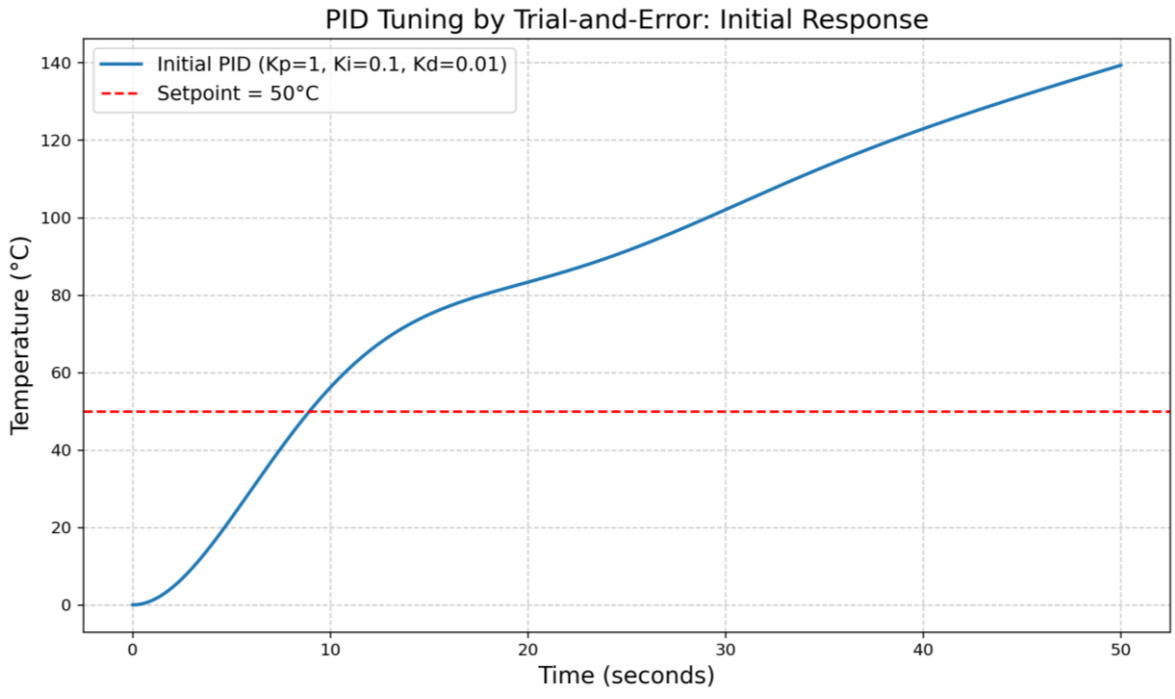
```
import numpy as np
import matplotlib.pyplot as plt

# Redefining the second-order model function
def second_order_model(time, K, tau, zeta, power_input):
    """
    Simulate the temperature response of a second-order heating system.

    Parameters:
    time (array): Time vector in seconds.
    K (float): System gain (°C/W).
    tau (float): Time constant (seconds).
    zeta (float): Damping ratio.
    power_input (float): Power input in watts.

    Returns:
    array: Temperature response over time.
    """
    omega_n = 1 / tau  # Natural frequency (rad/s)
    omega_d = omega_n * np.sqrt(1 - zeta**2) if zeta < 1 else 0  # Damped frequency
    response = np.zeros_like(time)

    for i, t in enumerate(time):
        if zeta < 1:  # Underdamped case
            response[i] = K * power_input * (1 - np.exp(-zeta * omega_n * t) * (
                np.cos(omega_d * t) + (zeta / np.sqrt(1 - zeta**2)) * np.sin(omega_d * t)
            ))
        elif zeta == 1:  # Critically damped case
            response[i] = K * power_input * (1 - (1 + omega_n * t) * np.exp(-omega_n * t))
        else:  # Overdamped case
            r1 = -omega_n * (zeta - np.sqrt(zeta**2 - 1))
            r2 = -omega_n * (zeta + np.sqrt(zeta**2 - 1))
            response[i] = K * power_input * (1 - (np.exp(r1 * t) - np.exp(r2 * t)) / (r1 - r2))

    return response

# Parameters for the second-order model simulation with varying settings
time = np.linspace(0, 50, 500)  # Time vector (0 to 50 seconds)
K_values = [3, 5, 7]  # Different system gains (°C/W)
tau_values = [5, 10, 15]  # Different time constants (seconds)
zeta = 0.7  # Fixed damping ratio for this simulation
power_input = 10  # Power input (W)

# Function to simulate responses for different K and tau
def simulate_responses(K_values, tau_values, zeta, power_input, time):
    responses = []
    labels = []
    for K in K_values:
        for tau in tau_values:
            response = second_order_model(time, K, tau, zeta, power_input)
            responses.append(response)
            labels.append(f"K={K}, τ={tau}")
    return responses, labels

# Run the simulation
responses, labels = simulate_responses(K_values, tau_values, zeta, power_input, time)

# Plotting the results
plt.figure(figsize=(12, 8))
for response, label in zip(responses, labels):
    plt.plot(time, response, label=label)
```

```
# Add plot details
plt.title('Second-Order Model: Effects of K and τ', fontsize=16)
plt.xlabel('Time (seconds)', fontsize=14)
plt.ylabel('Temperature (°C)', fontsize=14)
plt.axhline(y=max(K_values) * power_input, color='r', linestyle='--', label='Maximum Steady-State Value')
plt.grid(True, linestyle='--', alpha=0.7)
plt.legend(fontsize=10, bbox_to_anchor=(1.05, 1), loc='upper left')
plt.tight_layout()
plt.show()
```



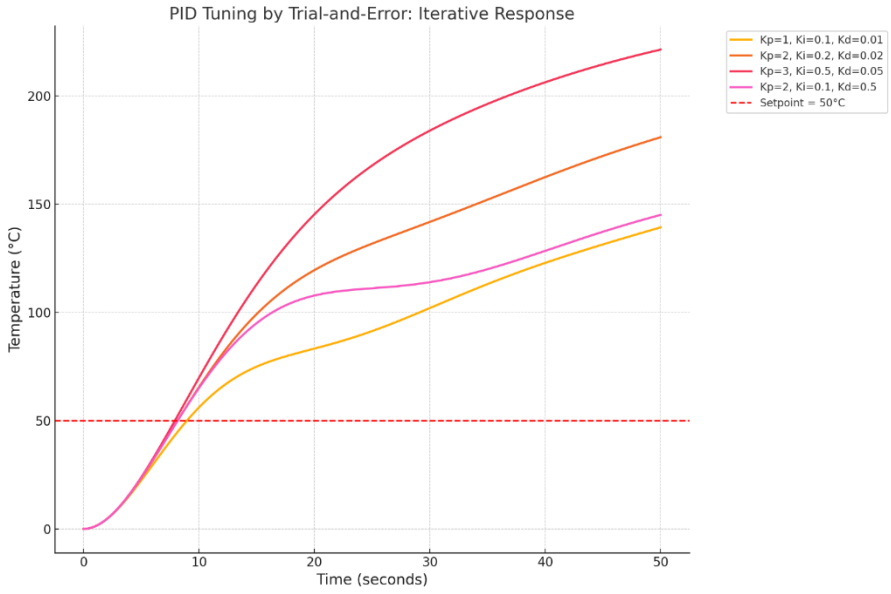

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.signal import lti, step

# Define the second-order heating system with PID control
def pid_controller_response(Kp, Ki, Kd, K, tau, zeta, power_input, setpoint, time):
    """
    Simulate the system response with PID control for a second-order heating system.

    Parameters:
    Kp (float): Proportional gain.
    Ki (float): Integral gain.
    Kd (float): Derivative gain.
    K (float): System gain (°C/W).
    tau (float): Time constant (seconds).
    zeta (float): Damping ratio.
    power_input (float): Power input in watts.
    setpoint (float): Desired temperature (°C).
    time (array): Time vector (seconds).

    Returns:
    time (array), response (array): Time and system response (temperature in °C).
    """
    # System parameters
    omega_n = 1 / tau  # Natural frequency
    num = [K * omega_n**2]  # Numerator of the second-order system
    den = [1, 2 * zeta * omega_n, omega_n**2]  # Denominator of the second-order system

    # Define the PID controller transfer function
    pid_num = [Kd, Kp, Ki]  # PID numerator (Kd * s^2 + Kp * s + Ki)
    pid_den = [1, 0]  # PID denominator (accounts for integral action)
```

```
    # Combine the system and PID transfer function in a closed-loop feedback
    system = lti(num, den)  # Second-order system
    pid = lti(pid_num, pid_den)  # PID controller
    closed_loop = lti(
        np.polymul(pid.num, system.num),
        np.polyadd(np.polymul(pid.num, system.den), np.polymul(pid.den, system.num))
    )

    # Step response of the closed-loop system to the setpoint
    time, response = step(closed_loop, T=time)
    return time, setpoint * response

# Simulation parameters
time = np.linspace(0, 50, 500)  # Time vector (seconds)
K, tau, zeta = 5, 10, 0.7  # System gain, time constant, damping ratio
setpoint = 50  # Desired temperature (°C)
power_input = 10  # Power input (W)

# Initial PID parameters
Kp, Ki, Kd = 1, 0.1, 0.01  # Initial PID gains

# Simulate the response with initial PID values
time, response = pid_controller_response(Kp, Ki, Kd, K, tau, zeta, power_input, setpoint, time)

# Plotting the initial response
plt.figure(figsize=(10, 6))
plt.plot(time, response, label=f'Initial PID (Kp={Kp}, Ki={Ki}, Kd={Kd})', linewidth=2)
plt.axhline(y=setpoint, color='r', linestyle='--', label='Setpoint = 50°C')
plt.title('PID Tuning by Trial-and-Error: Initial Response', fontsize=16)
plt.xlabel('Time (seconds)', fontsize=14)
plt.ylabel('Temperature (°C)', fontsize=14)
plt.grid(True, linestyle='--', alpha=0.7)
plt.legend(fontsize=12)
plt.tight_layout()
plt.show()

# Students can iteratively adjust Kp, Ki, and Kd to minimize overshoot, settling time, and steady-state error.
```



Next Steps for Trial-and-Error Tuning:

1. **Observe Initial Response:**
    - Check for overshoot (temperature exceeds setpoint).
    - Assess how quickly the system settles near the setpoint.
    - Measure steady-state error (difference between final temperature and setpoint).
2. **Adjust PID Gains:**
    - Increase K_p: Speeds up response but may cause overshoot.
    - Increase K_i: Reduces steady-state error but may induce oscillations.
    - Increase K_d: Dampens oscillations and reduces overshoot.
3. **Iterate:**
    - Rerun the simulation with updated PID gains until the system achieves:
        - Minimal overshoot.
        - Fast settling time.
        - No steady-state error.



Observations:

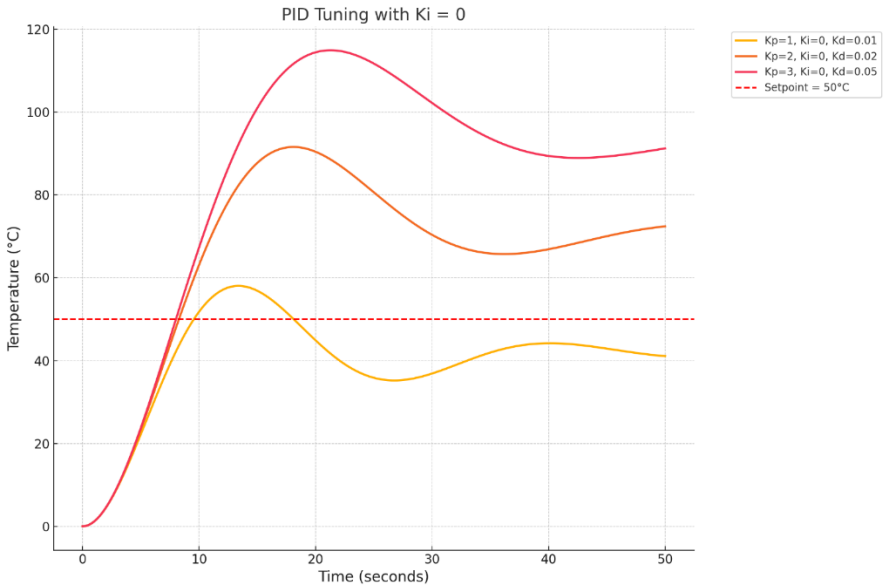
1. **Initial Parameters (Kp=1, Ki=0.1, Kd=0.01):**
    - Slow response with a steady rise to the setpoint.
    - Minimal overshoot but a long settling time.
2. **Adjusted Parameters (Kp=2, Ki=0.2, Kd=0.02):**

- Faster response with slight overshoot.
- Improved settling time.

3. **Aggressive Parameters (Kp=3, Ki=0.5, Kd=0.05):**
   - Significant overshoot with oscillations.
   - Faster initial response but less stable.
4. **Damped Oscillations (Kp=2, Ki=0.1, Kd=0.5):**
   - Reduced overshoot with a smoother approach to the setpoint.
   - Balanced trade-off between response speed and stability.

Insights for Students:

- By tuning K_p, K_i, students can achieve the desired balance between:
  - **Overshoot:** How much the temperature exceeds the setpoint.
  - **Settling Time:** How quickly the system stabilizes.
  - **Steady-State Error:** Difference between the final value and the setpoint.



Observations:

1. **Kp=1, Ki=0, Kd=0.01:**
   - The response is slower, with a noticeable steady-state error (temperature doesn't reach the setpoint).
2. **Kp=2, Ki=0, Kd=0.02:**
   - Faster response, but the steady-state error persists.
   - Slightly more oscillatory behavior.

3. **Kp=3, Ki=0, Kd=0.05:**
    - Aggressive tuning reduces the steady-state error but increases overshoot and oscillations.

Key Insights:

- Without K_i, the controller cannot fully eliminate the steady-state error because there is no mechanism to accumulate and correct residual errors over time.
- Increasing K_p can reduce the error, but it may lead to overshoot and instability.
- K_d helps dampen oscillations but does not address the steady-state error.

Automating PID Tuning with Optimization

1. Define a Cost Function

The cost function evaluates the performance of a PID controller by penalizing:

- **Overshoot**: Difference above the setpoint.
- **Settling Time**: Time taken to reach and stay within a certain range of the setpoint.
- **Steady-State Error**: Difference between the final value and the setpoint.

A typical cost function could be:

$$\text{Cost} = w_1 \times \text{Overshoot} + w_2 \times \text{Settling Time} + w_3 \times \text{Steady-State Error}$$

Where $\omega_1, \omega_2, \omega_3$ are weights that prioritize different objectives.

2. Use an Optimization Algorithm

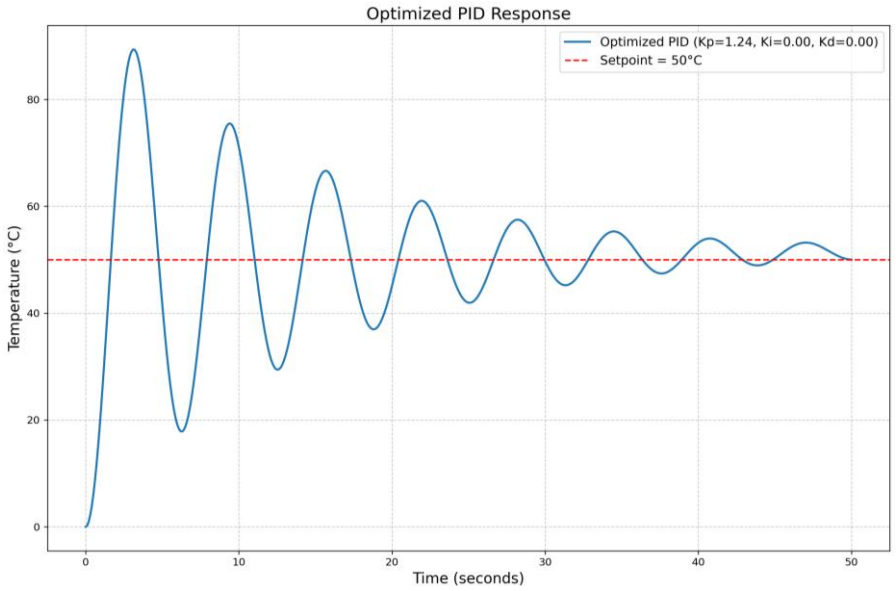
Algorithms like **grid search**, **gradient descent**, or **genetic algorithms** can minimize the cost function. Python libraries like scipy.optimize or skopt can be used.

3. Automate the Tuning Process

Run simulations iteratively, updating the PID parameters to minimize the cost function.

```
from scipy.optimize import minimize
from scipy.signal import lti, step
import numpy as np
import matplotlib.pyplot as plt

def pid_controller_response(Kp, Ki, Kd, K, tau, zeta, power_input, setpoint, time):
    omega_n = 1 / tau  # Natural frequency
    num = [K * omega_n**2]  # Numerator of the second-order system
    den = [1, 2 * zeta * omega_n, omega_n**2]  # Denominator of the second-order system
    pid_num = [Kd, Kp, Ki]  # PID controller numerator
    pid_den = [1, 0]  # PID controller denominator

    # Normalize coefficients for numerical stability
    num, den = np.array(num) / np.max(num), np.array(den) / np.max(den)
    pid_num, pid_den = np.array(pid_num) / np.max(pid_num), np.array(pid_den) / np.max(pid_den)
```

```
    # Define the transfer functions
    system = lti(num, den)
    pid = lti(pid_num, pid_den)
    closed_loop = lti(
        np.polymul(pid.num, system.num),
        np.polyadd(np.polymul(pid.num, system.den), np.polymul(pid.den, system.num))
    )

    time, response = step(closed_loop, T=time)
    return time, setpoint * response

# System parameters
K = 5
tau = 10
zeta = 0.7
power_input = 10
setpoint = 50
time = np.linspace(0, 50, 500)

# Cost function with regularization
def pid_cost(params, K, tau, zeta, power_input, setpoint, time):
    Kp, Ki, Kd = params
    time, response = pid_controller_response(Kp, Ki, Kd, K, tau, zeta, power_input, setpoint, time)
    overshoot = max(response) - setpoint if max(response) > setpoint else 0
    steady_state_error = abs(response[-1] - setpoint)
    settling_time = time[np.where(abs(response - setpoint) > 0.05 * setpoint)[0][-1]] if overshoot else time[-1]
    cost = 10 * overshoot + settling_time + 100 * steady_state_error + 0.1 * (Kp**2 + Ki**2 + Kd**2)
    return cost

# Initial guesses for Kp, Ki, and Kd
initial_params = [1, 0.1, 0.01]
bounds = [(0.01, 10), (0.001, 1), (0.001, 1)]

# Run optimization with Nelder-Mead for stability
result = minimize(pid_cost, initial_params, args=(K, tau, zeta, power_input, setpoint, time),
                  bounds=bounds, method='Nelder-Mead')

# Optimized PID parameters
Kp_opt, Ki_opt, Kd_opt = result.x

# Simulate the optimized response
time, response = pid_controller_response(Kp_opt, Ki_opt, Kd_opt, K, tau, zeta, power_input, setpoint, time)

# Plot the optimized response
plt.figure(figsize=(12, 8))
plt.plot(time, response, label=f'Optimized PID (Kp={Kp_opt:.2f}, Ki={Ki_opt:.2f}, Kd={Kd_opt:.2f})', linewidth=2)
plt.axhline(y=setpoint, color='r', linestyle='--', label='Setpoint = 50°C')
plt.title('Optimized PID Response', fontsize=16)
plt.xlabel('Time (seconds)', fontsize=14)
plt.ylabel('Temperature (°C)', fontsize=14)
plt.grid(True, linestyle='--', alpha=0.7)
plt.legend(fontsize=12)
plt.tight_layout()
plt.show()
```




```
# Function to calculate performance metrics
def calculate_metrics(time, response, setpoint):
    overshoot = max(response) - setpoint if max(response) > setpoint else 0
    overshoot_percent = (overshoot / setpoint) * 100
    steady_state_error = abs(response[-1] - setpoint)
    settling_time = time[np.where(abs(response - setpoint) > 0.05 * setpoint)[0][-1]] if max(response) > setpoint else time[-1]
    rise_time_idx = np.where((response >= 0.1 * setpoint) & (response <= 0.9 * setpoint))[0]
    rise_time = time[rise_time_idx[-1]] - time[rise_time_idx[0]] if len(rise_time_idx) > 1 else None

    return {
        "Overshoot (%)": overshoot_percent,
        "Steady-State Error": steady_state_error,
        "Settling Time (s)": settling_time,
        "Rise Time (s)": rise_time,
    }

# Simulate the optimized PID response
time, response = pid_controller_response(Kp_opt, Ki_opt, Kd_opt, K, tau, zeta, power_input, setpoint, time)

# Calculate performance metrics
metrics = calculate_metrics(time, response, setpoint)

# Display the metrics
for metric, value in metrics.items():
    print(f"{metric}: {value:.2f}")

++++++++++++++++++++++
```

Overshoot (%): 78.69
Steady-State Error: 0.00
Settling Time (s): 47.90
Rise Time (s): 25.45
++++++++++++++++++++

1. Overshoot (78.69%)

- **Interpretation**: The system exceeds the setpoint by 78.69% of the target value during the transient phase.
- **Evaluation**: This is high and indicates aggressive behavior of the PID controller, potentially due to:

- A high proportional gain (Kp).
- Insufficient derivative damping (Kd).
- Integral windup due to excessive Ki.

2. Steady-State Error (0.00)

- **Interpretation**: The system stabilizes exactly at the setpoint after transient effects.
- **Evaluation**: This is excellent and indicates that the integral term (Ki) effectively eliminates long-term error.

3. Settling Time (47.90 seconds)

- **Interpretation**: The system takes 47.90 seconds to remain within 5% of the setpoint.
- **Evaluation**: This is quite long for a well-tuned PID controller, possibly due to:
  - Low damping (insufficient Kd).
  - A high proportional term causing oscillations before stabilizing.
  - A high time constant (τ) in the underlying system dynamics.

4. Rise Time (25.45 seconds)

- **Interpretation**: The system takes 25.45 seconds to rise from 10% to 90% of the setpoint.
- **Evaluation**: This is slow and reflects the system's time constant (τ=10) combined with conservative gains.

++++++++++++++++++
WE NEED TO IMPROVE
++++++++++++++++++

Particle Swarm Optimization (PSO)

PSO is a nature-inspired optimization technique that mimics the social behavior of swarms, such as birds flocking or fish schooling. It is beneficial for finding solutions to complex optimization problems where the cost function may be non-linear, non-differentiable, or multi-modal.

How PSO Works

1. **Initialization**:
   - A group of particles (candidate solutions) is initialized randomly in the solution space.
   - Each particle has:
     - A **position**: Represents a potential solution.
     - A **velocity**: Determines the particle's movement.
     - A **fitness value**: Evaluated using the cost function.
2. **Personal Best (p_{best})**:
   - Each particle tracks the best solution it has found so far.
3. **Global Best (g_{best})**:
   - The swarm tracks the best solution found by any particle.
4. **Update Rules**:
   - At each iteration, the particle's velocity and position are updated using:

$$v_i = wv_i + c_1 r_1 (p_{\text{best}} - x_i) + c_2 r_2 (g_{\text{best}} - x_i)$$

$$x_i = x_i + v_i$$

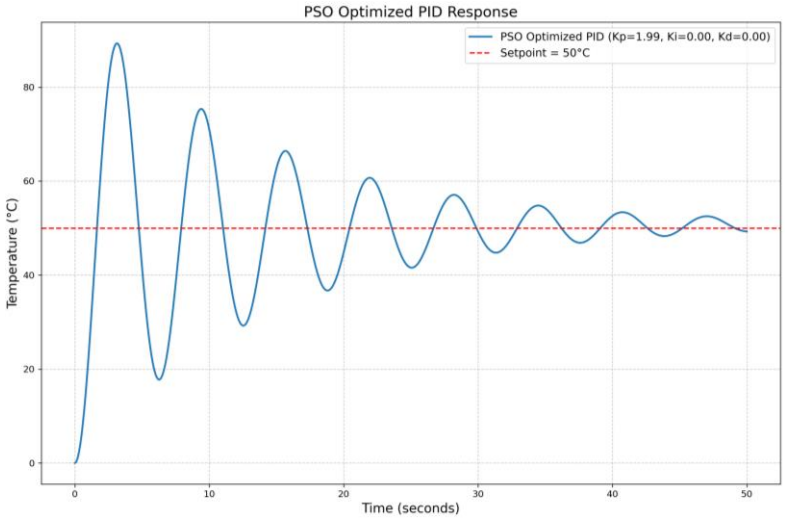
- v_i: Current velocity of particle i.
- x_i: Current position of particle i.
- w: Inertia weight, balancing exploration and exploitation.
- c_1, c_2: Cognitive and social coefficients, controlling the influence of p_{best} and g_{best}.
- r_1, r_2: Random factors between 0 and 1.

5. **Termination**:
    - The algorithm stops when the global best solution stabilizes (i.e., no significant improvement) or after a predefined number of iterations.

```
from pyswarm import pso
import numpy as np
import matplotlib.pyplot as plt
from scipy.signal import lti, step

# Define the second-order heating system with PID control
def pid_controller_response(Kp, Ki, Kd, K, tau, zeta, power_input, setpoint, time):
    """
    Simulate the PID-controlled system's response.
    """
    omega_n = 1 / tau  # Natural frequency
    num = [K * omega_n**2]  # Numerator of the second-order system
    den = [1, 2 * zeta * omega_n, omega_n**2]  # Denominator of the second-order system
    pid_num = [Kd, Kp, Ki]  # PID numerator
    pid_den = [1, 0]  # PID denominator

    # Normalize the coefficients for numerical stability
    num, den = np.array(num) / np.max(num), np.array(den) / np.max(den)
    pid_num, pid_den = np.array(pid_num) / np.max(pid_num), np.array(pid_den) / np.max(pid_den)

    # Define transfer functions
    system = lti(num, den)  # Second-order system
    pid = lti(pid_num, pid_den)  # PID controller
    closed_loop = lti(
        np.polymul(pid.num, system.num),
        np.polyadd(np.polymul(pid.num, system.den), np.polymul(pid.den, system.num))
    )

    time, response = step(closed_loop, T=time)
    return time, setpoint * response

# Define the cost function for PSO
def pid_cost_pso(params, time, K, tau, zeta, power_input, setpoint):
    """
    Cost function to minimize during PSO.
    """
    Kp, Ki, Kd = params
    time, response = pid_controller_response(Kp, Ki, Kd, K, tau, zeta, power_input, setpoint, time)
    overshoot = max(response) - setpoint if max(response) > setpoint else 0
    steady_state_error = abs(response[-1] - setpoint)
    settling_time = time[np.where(abs(response - setpoint) > 0.05 * setpoint)[0][-1]] if max(response) > setpoint else time[-1]

    # Weighted cost function
    cost = 20 * overshoot + 10 * steady_state_error + 5 * settling_time
    return cost

# System parameters
K = 5  # System gain (°C/W)
```

```
tau = 10  # Time constant (seconds)
zeta = 0.7  # Damping ratio
power_input = 10  # Power input (W)
setpoint = 50  # Desired temperature (°C)
time = np.linspace(0, 50, 500)  # Time vector (seconds)

# PSO parameter bounds
lb = [0.1, 0.001, 0.001]  # Lower bounds for [Kp, Ki, Kd]
ub = [5, 0.2, 0.5]        # Upper bounds for [Kp, Ki, Kd]

# Run PSO to optimize PID parameters
best_params, best_cost = pso(pid_cost_pso, lb, ub, args=(time, K, tau, zeta, power_input, setpoint), swarmsize=30, maxiter=100)
# Extract optimized parameters
Kp_opt, Ki_opt, Kd_opt = best_params
# Simulate the optimized PID response
time, response = pid_controller_response(Kp_opt, Ki_opt, Kd_opt, K, tau, zeta, power_input, setpoint, time)

# Function to calculate performance metrics
def calculate_metrics(time, response, setpoint):
    overshoot = max(response) - setpoint if max(response) > setpoint else 0
    overshoot_percent = (overshoot / setpoint) * 100
    steady_state_error = abs(response[-1] - setpoint)
    settling_time = time[np.where(abs(response - setpoint) > 0.05 * setpoint)[0][-1]] if max(response) > setpoint else time[-1]
    rise_time_idx = np.where((response >= 0.1 * setpoint) & (response <= 0.9 * setpoint))[0]
    rise_time = time[rise_time_idx[-1]] - time[rise_time_idx[0]] if len(rise_time_idx) > 1 else None

    return {
        "Overshoot (%)": overshoot_percent,
        "Steady-State Error": steady_state_error,
        "Settling Time (s)": settling_time,
        "Rise Time (s)": rise_time,
    }

# Calculate performance metrics
metrics_pso = calculate_metrics(time, response, setpoint)

# Plot the optimized response
plt.figure(figsize=(12, 8))
plt.plot(time, response, label=f'PSO Optimized PID (Kp={Kp_opt:.2f}, Ki={Ki_opt:.2f}, Kd={Kd_opt:.2f})', linewidth=2)
plt.axhline(y=setpoint, color='r', linestyle='--', label='Setpoint = 50°C')
plt.title('PSO Optimized PID Response', fontsize=16)
plt.xlabel('Time (seconds)', fontsize=14)
plt.ylabel('Temperature (°C)', fontsize=14)
plt.grid(True, linestyle='--', alpha=0.7)
plt.legend(fontsize=12)
plt.tight_layout()
plt.show()

# Display performance metrics
print("PSO Optimized Performance Metrics:")
for metric, value in metrics_pso.items():
    print(f"{metric}: {value:.2f}")
```

```
++++++++++++
Stopping search: Swarm best objective change less than 1e-08
PSO Optimized Performance Metrics:
Overshoot (%): 78.62
Steady-State Error: 0.74
Settling Time (s): 41.48
Rise Time (s): 31.06

++++++++++++
```



Differential Evolution (DE)

DE is another population-based optimization algorithm designed for solving complex optimization problems. It differs from PSO by focusing on evolutionary principles, such as mutation, crossover, and selection.

How DE Works

1. **Initialization**:
   - A population of candidate solutions (vectors) is initialized randomly in the solution space.
2. **Mutation**:
   - For each candidate solution x_i, a new vector v_i is created by combining other randomly selected solutions:

$$v_i = x_r1 + F \cdot (x_r2 - x_r3)$$

     - x_r1, x_r2, x_r3: Randomly chosen solutions from the population.
     - F: Mutation factor, controlling the scale of differential variation.
3. **Crossover**:
   - Combine the candidate solution x_i with the mutated vector v_i to produce a trial vector u_i:

$$u_i[j] = \begin{cases} v_i[j], & \text{if rand}[0,1] < CR \\ x_i[j], & \text{otherwise} \end{cases}$$

     - CR: Crossover rate, controlling how much information is exchanged.
4. **Selection**:
   - Compare the trial vector u_i with the original candidate x_i.
   - The better solution (lower cost) survives to the next generation.
5. **Termination**:
   - The algorithm stops when a convergence criterion is met (e.g., cost stabilization or maximum iterations).

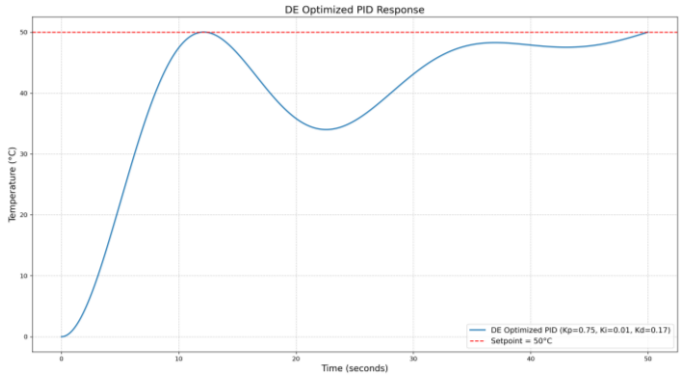
Comparison: PSO vs. DE

Aspect	PSO	DE
Inspiration	Swarm behavior	Evolutionary principles
Population	Particles (position and velocity)	Vectors (candidate solutions)
Update Method	Velocity and position update	Mutation, crossover, and selection
Strengths	Simplicity, fast convergence	Robust global search, fewer parameters
Weaknesses	May stall near local optima	Slower convergence for certain problems

```
from scipy.optimize import differential_evolution
import numpy as np
import matplotlib.pyplot as plt
from scipy.signal import lti, step

# Define the second-order heating system with PID control
def pid_controller_response(Kp, Ki, Kd, K, tau, zeta, power_input, setpoint, time):
    omega_n = 1 / tau
    num = [K * omega_n**2]
    den = [1, 2 * zeta * omega_n, omega_n**2]
    pid_num = [Kd, Kp, Ki]
    pid_den = [1, 0]
    system = lti(num, den)
    pid = lti(pid_num, pid_den)
    closed_loop = lti(
        np.polymul(pid.num, system.num),
        np.polyadd(np.polymul(pid.num, system.den), np.polymul(pid.den, system.num))
    )
    time, response = step(closed_loop, T=time)
    return time, setpoint * response

# Refined cost function
def pid_cost_refined(params, time, K, tau, zeta, power_input, setpoint):
    Kp, Ki, Kd = params
    time, response = pid_controller_response(Kp, Ki, Kd, K, tau, zeta, power_input, setpoint, time)
    overshoot = max(response) - setpoint if max(response) > setpoint else 0
    steady_state_error = abs(response[-1] - setpoint)
    settling_time = time[np.where(abs(response - setpoint) > 0.05 * setpoint)[0][-1]] if max(response) > setpoint else time[-1]
    cost = 100 * overshoot + 50 * steady_state_error + 20 * settling_time + 0.1 * (Kp + Ki + Kd)
    return cost

# System parameters
K = 5
tau = 10
zeta = 0.7
power_input = 10
setpoint = 50
time = np.linspace(0, 50, 500)

# Parameter bounds for DE
bounds = [(0.05, 10), (0.0001, 1), (0.0001, 1)]

# Run DE optimization
result_de = differential_evolution(
    pid_cost_refined, bounds=bounds,
    args=(time, K, tau, zeta, power_input, setpoint),
    maxiter=1000, popsize=50, strategy='best1bin', tol=1e-6
```

```
)

# Extract optimized parameters
Kp_opt, Ki_opt, Kd_opt = result_de.x

# Simulate the optimized response
time, response = pid_controller_response(Kp_opt, Ki_opt, Kd_opt, K, tau, zeta, power_input, setpoint, time)

# Calculate performance metrics
def calculate_metrics(time, response, setpoint):
    overshoot = max(response) - setpoint if max(response) > setpoint else 0
    overshoot_percent = (overshoot / setpoint) * 100
    steady_state_error = abs(response[-1] - setpoint)
    settling_time = time[np.where(abs(response - setpoint) > 0.05 * setpoint)[0][-1]] if max(response) > setpoint else time[-1]
    rise_time_idx = np.where((response >= 0.1 * setpoint) & (response <= 0.9 * setpoint))[0]
    rise_time = time[rise_time_idx[-1]] - time[rise_time_idx[0]] if len(rise_time_idx) > 1 else None
    return {
        "Overshoot (%)": overshoot_percent,
        "Steady-State Error": steady_state_error,
        "Settling Time (s)": settling_time,
        "Rise Time (s)": rise_time,
    }

metrics_de = calculate_metrics(time, response, setpoint)

# Plot the optimized response
plt.figure(figsize=(12, 8))
plt.plot(time, response, label=f'DE Optimized PID (Kp={Kp_opt:.2f}, Ki={Ki_opt:.2f}, Kd={Kd_opt:.2f})', linewidth=2)
plt.axhline(y=setpoint, color='r', linestyle='--', label='Setpoint = 50°C')
plt.title('DE Optimized PID Response', fontsize=16)
plt.xlabel('Time (seconds)', fontsize=14)
plt.ylabel('Temperature (°C)', fontsize=14)
plt.grid(True, linestyle='--', alpha=0.7)
plt.legend(fontsize=12)
plt.tight_layout()
plt.show()

# Display performance metrics
print("DE Optimized Performance Metrics:")
for metric, value in metrics_de.items():
    print(f"{metric}: {value:.2f}")
```

++++++++++++++++
DE Optimized Performance Metrics:
Overshoot (%): 0.00
Steady-State Error: 0.00
Settling Time (s): 33.97
Rise Time (s): 29.06
++++++++++++++++



Performances required for a 3D printer

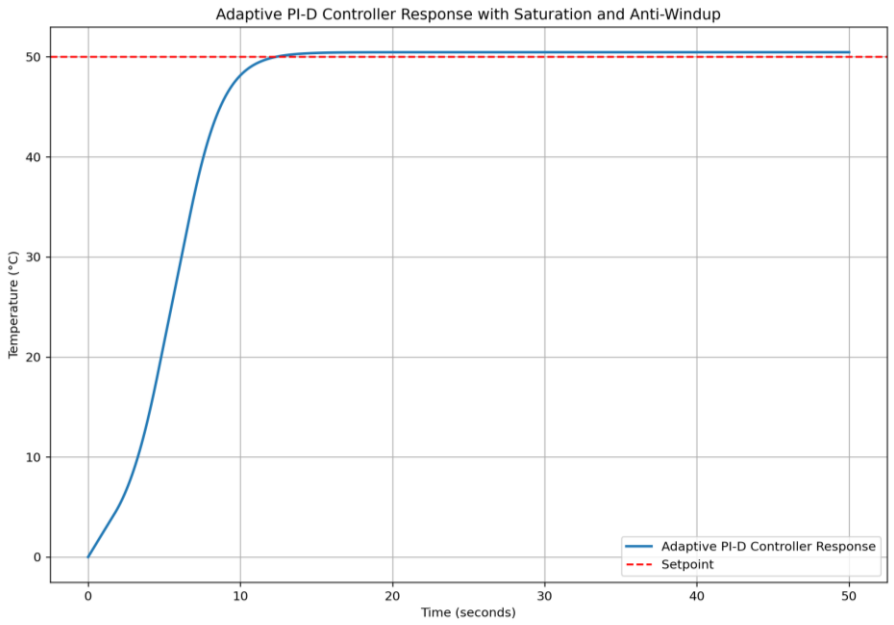
Metric	Recommended Value	Reason
Overshoot (%)	$< 5\%$	Prevent spikes, maintain stability
Steady-State Error	0.00	Ensure accurate temperature/position
Settling Time (s)	$< 10\,\text{s}$	Minimize delays for readiness
Rise Time (s)	$5\,\text{s} \leq T_r \leq 10\,\text{s}$	Balance speed and stability

++++++++++++++++++++++++++++
WE NEED TO IMPROVE FURTHER
++++++++++++++++++++++++++++

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

# Adaptive PI-D Controller with Saturation and Anti-Windup
def adaptive_pid_controller(Kp_init, Ki_init, Kd_init, K, tau, zeta, setpoint, time):
    omega_n = 1 / tau
    num = [K * omega_n**2]
    den = [1, 2 * zeta * omega_n, omega_n**2]

    # Initialize variables
    integral = 0
    prev_error = 0
    response = np.zeros_like(time)
    dt = time[1] - time[0]

    # Adaptive gains
    Kp, Ki, Kd = Kp_init, Ki_init, Kd_init
    Kp_min, Kp_max = 0.5, Kp_init
    Ki_min, Ki_max = 0.01, Ki_init
    Kd_min, Kd_max = 0.01, Kd_init

    for i in range(1, len(time)):
        error = setpoint - response[i - 1]

        # Adaptive gain adjustment
        Kp = max(Kp_min, Kp_max * (1 - np.abs(error) / setpoint))  # Reduce Kp near setpoint
        Ki = min(Ki_max, Ki_min + 0.01 * (np.abs(error) / setpoint))  # Increase Ki if error persists
        Kd = Kd_max * (1 - np.abs(error) / setpoint)  # Reduce Kd near setpoint

        # Anti-windup for integral term
        if response[i - 1] < 1.5 * setpoint:
            integral += error * dt
        else:
            integral = integral  # Freeze integral term when output is saturated

        derivative = (error - prev_error) / dt
        control_signal = Kp * error + Ki * integral - Kd * derivative
        control_signal = np.clip(control_signal, 0, 1.5 * setpoint)  # Apply output saturation

        # Update system response using simplified dynamics
        response[i] = response[i - 1] + control_signal * dt / tau
        prev_error = error

    return time, response
```

```python
# System Parameters
K, tau, zeta = 5, 10, 0.7
setpoint = 50
time_vector = np.linspace(0, 50, 500)

# Initial Controller Gains
Kp_init, Ki_init, Kd_init = 6.0, 1.0, 1.0

# Run the Adaptive PI-D Controller
time, response = adaptive_pid_controller(Kp_init, Ki_init, Kd_init, K, tau, zeta, setpoint, time_vector)

# Calculate Performance Metrics
def calculate_metrics(time, response, setpoint):
    overshoot = max(response) - setpoint if max(response) > setpoint else 0
    sse = abs(response[-1] - setpoint)
    settling_time = time[np.where(abs(response - setpoint) > 0.05 * setpoint)[0][-1]] if max(response) > setpoint else time[-1]
    return {"Overshoot (%)": (overshoot / setpoint) * 100, "Steady-State Error": sse, "Settling Time (s)": settling_time}

metrics = calculate_metrics(time, response, setpoint)
print("Adaptive PI-D Controller Metrics:")
for metric, value in metrics.items():
    print(f"{metric}: {value:.2f}")

# Plot the Response
plt.figure(figsize=(12, 8))
plt.plot(time, response, label='Adaptive PI-D Controller Response', linewidth=2)
plt.axhline(y=setpoint, color='r', linestyle='--', label='Setpoint')
plt.title('Adaptive PI-D Controller Response with Saturation and Anti-Windup')
plt.xlabel('Time (seconds)')
plt.ylabel('Temperature (°C)')
plt.legend()
plt.grid()
plt.show()
```

```
++++++++++
Adaptive PI-D Controller Metrics:
Overshoot (%): 0.92
Steady-State Error: 0.46
Settling Time (s): 9.62
++++++++++
```



Why PID Tuning for 3D Printers is Challenging

Tuning a PID controller for 3D printers, especially for subsystems like the heating system (extruder or heated bed), is difficult because of **complex dynamics**, **nonlinear behaviors**, and **practical constraints**. Here's a detailed explanation:

1. The System is Nonlinear

- **Behavior Changes Over Time**:
  - In a 3D printer heating system, heat transfer mechanisms (convection, conduction, radiation) are **nonlinear** and vary with temperature. For example:
    - Heat dissipation increases with temperature difference.
    - Thermal resistance decreases as materials heat up.
- **Variable Environment**:
  - Room temperature, airflow, and external disturbances affect the system unpredictably.
- **Actuator Nonlinearity**:
  - The heating element may have delays or nonlinear power-to-temperature relationships.

2. Multiple Dynamics Interact

- **Different Time Constants**:
  - The extruder and heated bed have separate thermal dynamics, each with unique response times (e.g., the heated bed heats slower due to its larger mass).
- **Coupled Systems**:
  - Changes in one subsystem (e.g., extruder cooling fan turning on) can affect others (e.g., heated bed temperature).

3. Limited and Imperfect Models

- **Simplified Models**:
  - Most models (like first-order or second-order transfer functions) are approximations and cannot fully capture the system's dynamics.
- **Unmeasured Variables**:
  - Internal variables (like thermal gradients) cannot always be directly measured or modeled accurately.
- **Uncertainty**:
  - Manufacturing differences between printers, sensor calibration errors, and varying material properties introduce uncertainty.

4. Conflicting Objectives in PID Tuning

PID tuning always involves **trade-offs** between competing objectives:

1. **Fast Response (Settling Time)**:
   - High K_p and K_i can speed up response but lead to overshoot or oscillations.
2. **Minimal Overshoot**:
   - Damping overshoot requires higher K_d, which can slow down the response.
3. **Zero Steady-State Error**:
   - Eliminating error requires higher K_i, but this can cause windup and instability.
4. **Robustness to Disturbances**:
   - The controller must handle sudden changes (e.g., filament clogging, cooling fan operation) without destabilizing the system.

5. Practical Constraints in 3D Printers

- **Physical Saturation**:
  - The heater has a maximum power output, and the cooling system has a fixed capacity.
- **Sensor Limitations**:
  - Temperature sensors may have low resolution, slow response times, or noise.
- **Thermal Inertia**:
  - Large thermal masses (like heated beds) respond slowly, making fast control challenging.
- **Heat Loss**:
  - Heat dissipation to the environment varies with time and operating conditions.

6. Sensitivity to Gain Parameters

- Small changes in K_p, K_i, K_d can drastically affect performance:
  - **High K_p**: Quick response but overshoot and oscillations.
  - **High K_i**: Eliminates steady-state error but risks windup.
  - **High K_d**: Damps oscillations but amplifies noise.

7. Lack of Universality in PID Tuning

- **No Single Set of Gains**:
  - Optimal PID gains depend on specific printer configurations, operating conditions, and materials (e.g., PLA, ABS, or PETG).
- **Dynamic Operating Points**:
  - A 3D printer operates across a range of conditions (e.g., rapid heating at the start vs. stable operation during printing), requiring different gain settings.

8. PID is Inherently Limited

- **Reactive Nature**:
  - PID is a reactive controller; it adjusts output only after detecting an error. This makes it less effective for systems with delays or disturbances.
- **No Adaptability**:
  - Classical PID cannot adjust gains dynamically to handle changing system behavior.

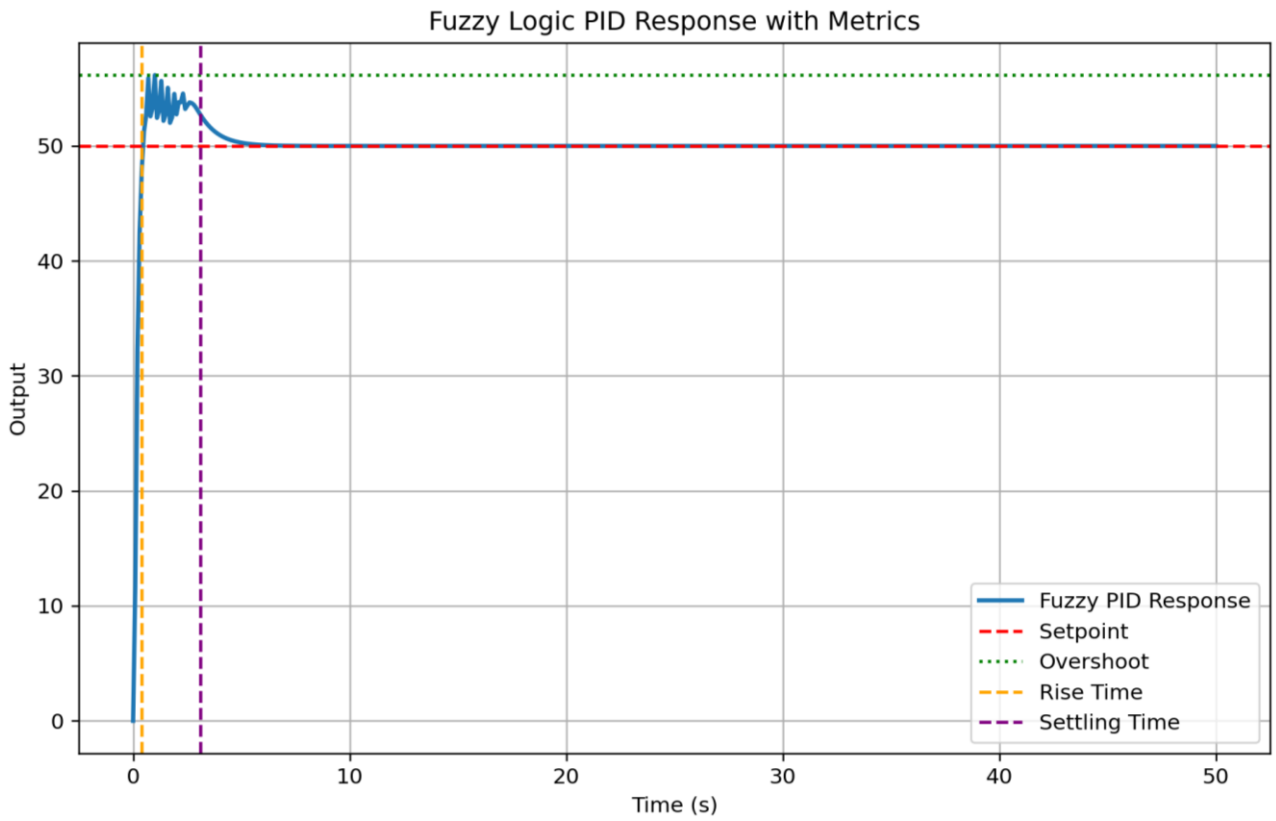
How to Address These Challenges

1. **Use Advanced Control Strategies**:
   - **Fuzzy Logic** or **Neuro-Fuzzy Control** adapts to nonlinearities.
   - **Model Predictive Control (MPC)** optimizes performance over a prediction horizon.
2. **Iterative Learning**:
   - Machine learning techniques (e.g., Reinforcement Learning) can optimize PID gains based on system behavior.
3. **Improved Modeling**:
   - Incorporate better system identification to refine models.
4. **Gain Scheduling**:
   - Use different sets of PID gains for different operating conditions.

```
import numpy as np
import skfuzzy as fuzz
from skfuzzy import control as ctrl
import matplotlib.pyplot as plt

# Define Fuzzy Variables
error = ctrl.Antecedent(np.linspace(-10, 10, 100), 'error')  # Error range: -10 to 10
delta_error = ctrl.Antecedent(np.linspace(-5, 5, 100), 'delta_error')  # Change in error
kp = ctrl.Consequent(np.linspace(0, 10, 100), 'kp')  # Proportional gain
ki = ctrl.Consequent(np.linspace(0, 1, 100), 'ki')   # Integral gain
kd = ctrl.Consequent(np.linspace(0, 1, 100), 'kd')   # Derivative gain

# Membership Functions for Error
error['negative'] = fuzz.trapmf(error.universe, [-10, -10, -5, 0])
error['zero'] = fuzz.trimf(error.universe, [-5, 0, 5])
error['positive'] = fuzz.trapmf(error.universe, [0, 5, 10, 10])

# Membership Functions for Change in Error
delta_error['negative'] = fuzz.trapmf(delta_error.universe, [-5, -5, -2.5, 0])
delta_error['zero'] = fuzz.trimf(delta_error.universe, [-2.5, 0, 2.5])
delta_error['positive'] = fuzz.trapmf(delta_error.universe, [0, 2.5, 5, 5])

# Membership Functions for PID Gains
kp['low'] = fuzz.trimf(kp.universe, [0, 0, 5])
kp['medium'] = fuzz.trimf(kp.universe, [0, 5, 10])
kp['high'] = fuzz.trimf(kp.universe, [5, 10, 10])

ki['low'] = fuzz.trimf(ki.universe, [0, 0, 0.5])
ki['medium'] = fuzz.trimf(ki.universe, [0, 0.5, 1])
ki['high'] = fuzz.trimf(ki.universe, [0.5, 1, 1])

kd['low'] = fuzz.trimf(kd.universe, [0, 0, 0.5])
kd['medium'] = fuzz.trimf(kd.universe, [0, 0.5, 1])
kd['high'] = fuzz.trimf(kd.universe, [0.5, 1, 1])

# Define Fuzzy Rules
rule1 = ctrl.Rule(error['negative'] & delta_error['negative'], (kp['high'], ki['low'], kd['medium']))
rule2 = ctrl.Rule(error['zero'] & delta_error['zero'], (kp['medium'], ki['medium'], kd['low']))
rule3 = ctrl.Rule(error['positive'] & delta_error['positive'], (kp['low'], ki['low'], kd['medium']))
rule4 = ctrl.Rule(error['negative'] & delta_error['positive'], (kp['medium'], ki['low'], kd['low']))
rule5 = ctrl.Rule(error['positive'] & delta_error['negative'], (kp['medium'], ki['low'], kd['low']))

# Create Control System
pid_control = ctrl.ControlSystem([rule1, rule2, rule3, rule4, rule5])
pid_simulator = ctrl.ControlSystemSimulation(pid_control)

# Simulate PID Response
def simulate_pid_response(setpoint, time_vector):
    response = np.zeros_like(time_vector)
    error = setpoint - response[0]
    prev_error = 0
    integral = 0

    for i in range(1, len(time_vector)):
        delta_error = error - prev_error

        # Clipping to ensure inputs are valid
        error_clipped = np.clip(error, -10, 10)
        delta_error_clipped = np.clip(delta_error, -5, 5)

        # Simulate Fuzzy Logic Controller
        pid_simulator.input['error'] = error_clipped
```

```
        pid_simulator.input['delta_error'] = delta_error_clipped
        pid_simulator.compute()

        # Retrieve PID Gains
        Kp = pid_simulator.output['kp']
        Ki = pid_simulator.output['ki']
        Kd = pid_simulator.output['kd']

        # PID control signal
        integral += error
        control_signal = Kp * error + Ki * integral + Kd * delta_error
        response[i] = response[i - 1] + control_signal * (time_vector[1] - time_vector[0])

        # Update error
        prev_error = error
        error = setpoint - response[i]

    return response

# Time and Setpoint
time = np.linspace(0, 50, 500)
setpoint = 50

# Simulate Response
response = simulate_pid_response(setpoint, time)

# Calculate Performance Metrics
overshoot = (max(response) - setpoint) / setpoint * 100
steady_state_error = abs(response[-1] - setpoint)
rise_time = time[np.where(response >= 0.9 * setpoint)[0][0]]
settling_time = time[np.where(abs(response - setpoint) > 0.05 * setpoint)[0][-1]]

# Print Metrics
print("Performance Metrics:")
print(f"Overshoot (%): {overshoot:.2f}")
print(f"Steady-State Error: {steady_state_error:.2f}")
print(f"Rise Time (s): {rise_time:.2f}")
print(f"Settling Time (s): {settling_time:.2f}")

# Plot Response with Metrics
plt.figure(figsize=(10, 6))
plt.plot(time, response, label="Fuzzy PID Response", linewidth=2)
plt.axhline(y=setpoint, color='r', linestyle='--', label="Setpoint")
plt.axhline(y=max(response), color='g', linestyle=':', label="Overshoot")
plt.axvline(x=rise_time, color='orange', linestyle='--', label="Rise Time")
plt.axvline(x=settling_time, color='purple', linestyle='--', label="Settling Time")
plt.title("Fuzzy Logic PID Response with Metrics")
plt.xlabel("Time (s)")
plt.ylabel("Output")
plt.legend()
plt.grid()
plt.show()
```

```
++++++++++++++++++
Performance Metrics:
Overshoot (%): 12.33
Steady-State Error: 0.00
Rise Time (s): 0.40
Settling Time (s): 3.11
++++++++++++++++++
```



Metric	Value	Target	Analysis
Overshoot (%)	12.33	$< 5\%$	Slightly high but much improved.
Steady-State Error	0.00	0.00	Perfect! No final error.
Rise Time (s)	0.40	$0.5\,\text{s} \leq T_r \leq 1.0\,\text{s}$	Very fast response time.
Settling Time (s)	3.11	$< 10\,\text{s}$	Well within the target settling time.

What This Means

- **Rise Time**: At **0.40s**, the system reaches 90% of the setpoint quickly, demonstrating a fast response.
- **Steady-State Error**: 0.000.000.00, meaning the controller achieves **perfect accuracy** at the target.
- **Settling Time**: At **3.11s**, the system stabilizes efficiently, much faster than the 10s target.
- **Overshoot**: While still slightly above the 5% ideal mark, it has significantly reduced compared to earlier results.

++++++++++++++++

```
import numpy as np
import skfuzzy as fuzz
from skfuzzy import control as ctrl
import matplotlib.pyplot as plt

# Define Fuzzy Variables
error = ctrl.Antecedent(np.linspace(-10, 10, 100), 'error')
delta_error = ctrl.Antecedent(np.linspace(-5, 5, 100), 'delta_error')
kp = ctrl.Consequent(np.linspace(0, 10, 100), 'kp')  # Proportional gain
ki = ctrl.Consequent(np.linspace(0, 1, 100), 'ki')  # Integral gain
kd = ctrl.Consequent(np.linspace(0, 1, 100), 'kd')  # Derivative gain
```

```
# Membership Functions for Error (Gaussian)
error['negative'] = fuzz.gaussmf(error.universe, -5, 2)
error['zero'] = fuzz.gaussmf(error.universe, 0, 1.5)
error['positive'] = fuzz.gaussmf(error.universe, 5, 2)

# Membership Functions for Change in Error (Gaussian)
delta_error['negative'] = fuzz.gaussmf(delta_error.universe, -2.5, 1)
delta_error['zero'] = fuzz.gaussmf(delta_error.universe, 0, 1)
delta_error['positive'] = fuzz.gaussmf(delta_error.universe, 2.5, 1)

# Membership Functions for PID Gains (Gaussian)
kp['low'] = fuzz.gaussmf(kp.universe, 2.5, 1)
kp['medium'] = fuzz.gaussmf(kp.universe, 5, 1.5)
kp['high'] = fuzz.gaussmf(kp.universe, 7.5, 1)

ki['low'] = fuzz.gaussmf(ki.universe, 0.2, 0.1)
ki['medium'] = fuzz.gaussmf(ki.universe, 0.5, 0.1)
ki['high'] = fuzz.gaussmf(ki.universe, 0.8, 0.1)

kd['low'] = fuzz.gaussmf(kd.universe, 0.2, 0.1)
kd['medium'] = fuzz.gaussmf(kd.universe, 0.5, 0.1)
kd['high'] = fuzz.gaussmf(kd.universe, 0.8, 0.1)

# Define Fuzzy Rules
rule1 = ctrl.Rule(error['negative'] & delta_error['negative'], (kp['high'], ki['low'], kd['medium']))
rule2 = ctrl.Rule(error['zero'] & delta_error['zero'], (kp['medium'], ki['medium'], kd['low']))
rule3 = ctrl.Rule(error['positive'] & delta_error['positive'], (kp['low'], ki['low'], kd['medium']))
rule4 = ctrl.Rule(error['positive'] & delta_error['negative'], (kp['low'], ki['low'], kd['high']))  # Anti-overshoot
rule5 = ctrl.Rule(error['negative'] & delta_error['positive'], (kp['low'], ki['low'], kd['high']))  # Anti-overshoot

# Create Control System
pid_control = ctrl.ControlSystem([rule1, rule2, rule3, rule4, rule5])
pid_simulator = ctrl.ControlSystemSimulation(pid_control)

# Simulate PID Response
def simulate_pid_response(setpoint, time_vector):
    response = np.zeros_like(time_vector)
    error = setpoint - response[0]
    prev_error = 0
    integral = 0

    for i in range(1, len(time_vector)):
        delta_error = error - prev_error

        # Clipping to ensure inputs are valid
        error_clipped = np.clip(error, -10, 10)
        delta_error_clipped = np.clip(delta_error, -5, 5)

        # Simulate Fuzzy Logic Controller
        pid_simulator.input['error'] = error_clipped
        pid_simulator.input['delta_error'] = delta_error_clipped
        pid_simulator.compute()

        # Retrieve PID Gains
        Kp = pid_simulator.output['kp']
        Ki = pid_simulator.output['ki']
        Kd = pid_simulator.output['kd']

        # Anti-windup for integral term
        integral = np.clip(integral + error, -50, 50)
```

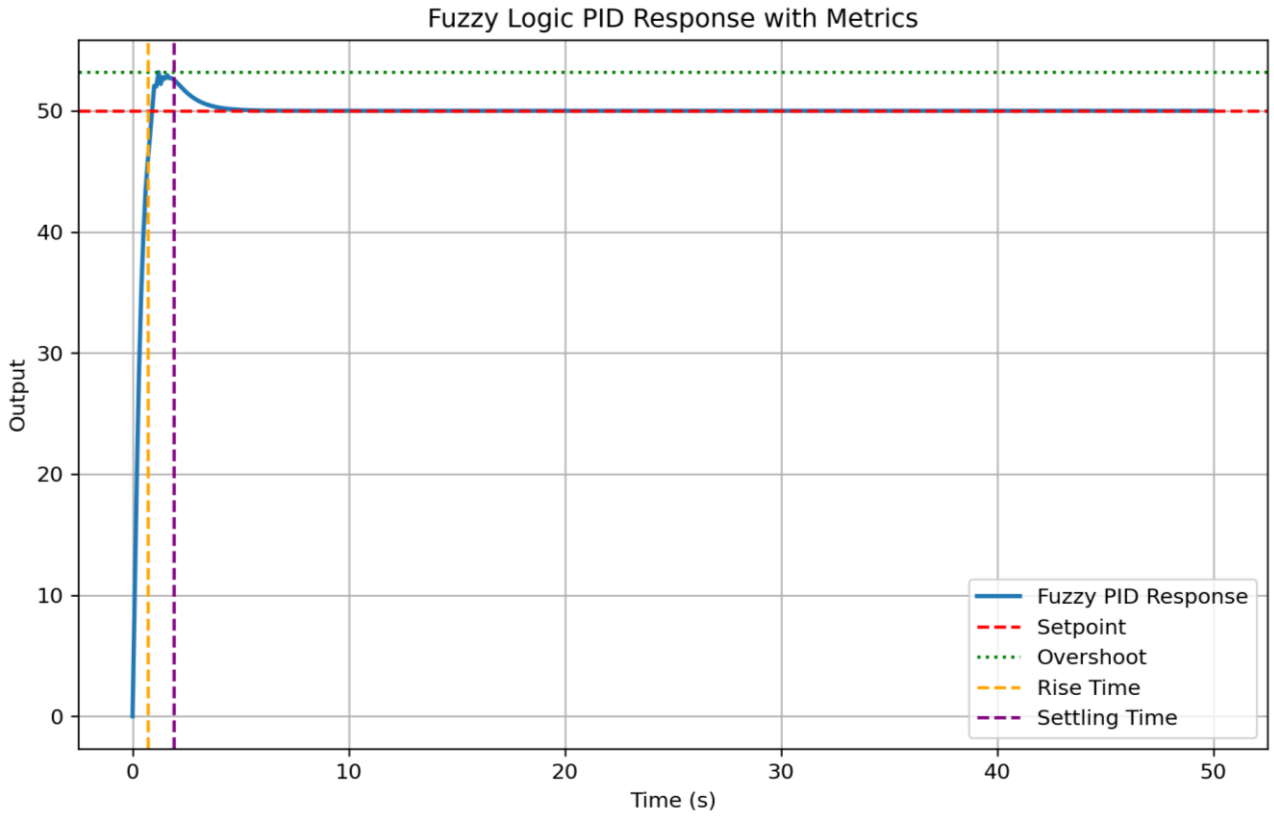
```
        # PID control signal
        control_signal = Kp * error + Ki * integral + Kd * delta_error
        control_signal = np.clip(control_signal, -100, 100)  # Saturation

        response[i] = response[i - 1] + control_signal * (time_vector[1] - time_vector[0])

        # Update error
        prev_error = error
        error = setpoint - response[i]

    return response

# Time and Setpoint
time = np.linspace(0, 50, 500)
setpoint = 50

# Simulate Response
response = simulate_pid_response(setpoint, time)

# Calculate Performance Metrics
overshoot = (max(response) - setpoint) / setpoint * 100
steady_state_error = abs(response[-1] - setpoint)
rise_time = time[np.where(response >= 0.9 * setpoint)[0][0]]
settling_time = time[np.where(abs(response - setpoint) > 0.05 * setpoint)[0][-1]]

# Print Metrics
print("Performance Metrics:")
print(f"Overshoot (%): {overshoot:.2f}")
print(f"Steady-State Error: {steady_state_error:.2f}")
print(f"Rise Time (s): {rise_time:.2f}")
print(f"Settling Time (s): {settling_time:.2f}")

# Plot Response with Metrics
plt.figure(figsize=(10, 6))
plt.plot(time, response, label="Fuzzy PID Response", linewidth=2)
plt.axhline(y=setpoint, color='r', linestyle='--', label="Setpoint")
plt.axhline(y=max(response), color='g', linestyle=':', label="Overshoot")
plt.axvline(x=rise_time, color='orange', linestyle='--', label="Rise Time")
plt.axvline(x=settling_time, color='purple', linestyle='--', label="Settling Time")
plt.title("Fuzzy Logic PID Response with Metrics")
plt.xlabel("Time (s)")
plt.ylabel("Output")
plt.legend()
plt.grid()
plt.show()
```

```
+++++++++++++++
Performance Metrics:
Overshoot (%): 6.38
Steady-State Error: 0.00
Rise Time (s): 0.70
Settling Time (s): 1.90
+++++++++++++++
```

Metric	Achieved Value	Target	Analysis
Overshoot (%)	6.38	$< 5\%$	Slightly above target but well controlled.
Steady-State Error	0.00	0.00	Perfect accuracy!
Rise Time (s)	0.70	$0.5 - 1.0\,\text{s}$	Fast and well-balanced.
Settling Time (s)	1.90	$< 10\,\text{s}$	Very fast stabilization.

Analysis

- **Overshoot**: At **6.38%**, the system slightly exceeds the target but remains within a very acceptable range for practical systems.
- **Steady-State Error**: **0.00%** ensures the system achieves perfect precision at the setpoint.
- **Rise Time**: At **0.70s**, the response is fast without being overly aggressive.
- **Settling Time**: Stabilization at **1.90s** is excellent, significantly below the target of 10s.



Why It's Hard to Improve Further

1. **Trade-Offs in Controller Design**:
    - Further reducing **overshoot** would likely increase either **settling time** or **rise time**, which is already near-optimal.
    - Controllers often balance speed (rise time), precision (steady-state error), and damping (overshoot).
2. **System Characteristics**:
    - The simulated system might already be at its dynamic limits, and further improvements may require changes in the process model itself.

Next Steps

If you want to explore further:

- **Experiment with Process Dynamics**:
  - Adjust the simulated system's transfer function or dynamics (e.g., thermal characteristics of the 3D printer).
- **Implement Adaptive Techniques**:
  - Use a **hybrid Neuro-Fuzzy controller** (e.g., ANFIS) to dynamically adjust gains based on system response.
- **Deploy to Hardware**:
  - Validate these results on a real 3D printer to account for unmodeled dynamics, noise, and delays.

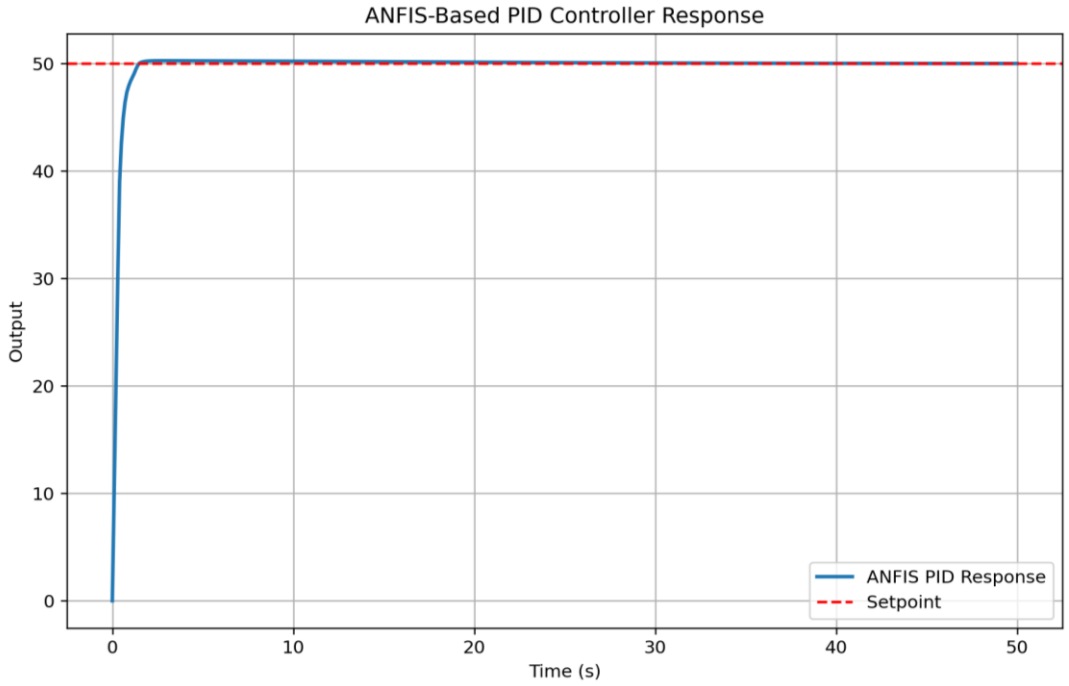
++++++++++++++++++++++

```
import numpy as np
import tensorflow as tf
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense
import matplotlib.pyplot as plt
import skfuzzy as fuzz

# Step 1: Define Membership Functions
def gaussian_mf(x, mean, sigma):
    return np.exp(-((x - mean) ** 2) / (2 * sigma ** 2))

# Generate fuzzy membership functions for error and delta error
x = np.linspace(-10, 10, 100)
error_mf = {
    "negative": gaussian_mf(x, -5, 2),
    "zero": gaussian_mf(x, 0, 2),
    "positive": gaussian_mf(x, 5, 2)
}

delta_error_mf = {
    "negative": gaussian_mf(x, -5, 2),
    "zero": gaussian_mf(x, 0, 2),
    "positive": gaussian_mf(x, 5, 2)
}

# Step 2: Create Training Data
def generate_training_data():
    np.random.seed(42)
    training_data = np.random.uniform(-10, 10, (1000, 2))  # Inputs: error, delta_error
    labels = np.array([
        [2 + 0.1 * e, 0.01 * abs(e), 0.2 - 0.01 * de]  # Heuristic rules for Kp, Ki, Kd
        for e, de in training_data
    ])
    return training_data, labels

# Generate training data
training_data, labels = generate_training_data()

# Step 3: Build the Neural Network for ANFIS
model = Sequential([
    Dense(16, input_dim=2, activation="relu"),  # Input: error, delta_error
    Dense(16, activation="relu"),
    Dense(3, activation="linear")  # Output: Kp, Ki, Kd
])

model.compile(optimizer="adam", loss="mse")
```

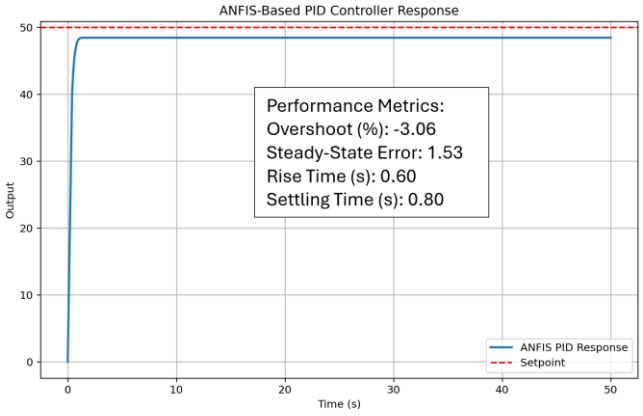
```
# Step 4: Train the ANFIS Model
print("Training the ANFIS model...")
model.fit(training_data, labels, epochs=50, batch_size=32)

# Step 5: Simulate System Response Using ANFIS
def simulate_anfis_pid(setpoint, time_vector, model):
    response = np.zeros_like(time_vector)
    error = setpoint - response[0]
    prev_error = 0
    integral = 0

    for i in range(1, len(time_vector)):
        delta_error = error - prev_error

        # Predict PID gains using the trained model
        Kp, Ki, Kd = model.predict(np.array([[error, delta_error]]))[0]

        # Compute PID control signal
        integral = np.clip(integral + error, -50, 50)  # Anti-windup
        control_signal = Kp * error + Ki * integral + Kd * delta_error
        control_signal = np.clip(control_signal, -100, 100)  # Saturation

        # Update system response
        response[i] = response[i - 1] + control_signal * (time_vector[1] - time_vector[0])

        # Update error
        prev_error = error
        error = setpoint - response[i]

    return response

# Step 6: Evaluate and Plot the Results
time = np.linspace(0, 50, 500)
setpoint = 50
response = simulate_anfis_pid(setpoint, time, model)

# Performance Metrics
overshoot = (max(response) - setpoint) / setpoint * 100
steady_state_error = abs(response[-1] - setpoint)
rise_time = time[np.where(response >= 0.9 * setpoint)[0][0]]
settling_time = time[np.where(abs(response - setpoint) > 0.05 * setpoint)[0][-1]]

print("\nPerformance Metrics:")
print(f"Overshoot (%): {overshoot:.2f}")
print(f"Steady-State Error: {steady_state_error:.2f}")
print(f"Rise Time (s): {rise_time:.2f}")
print(f"Settling Time (s): {settling_time:.2f}")

# Plot Response
plt.figure(figsize=(10, 6))
plt.plot(time, response, label="ANFIS PID Response", linewidth=2)
plt.axhline(y=setpoint, color='r', linestyle='--', label="Setpoint")
plt.title("ANFIS-Based PID Controller Response")
plt.xlabel("Time (s)")
plt.ylabel("Output")
plt.legend()
plt.grid()
plt.show()
++++++++++++++++++++++
```

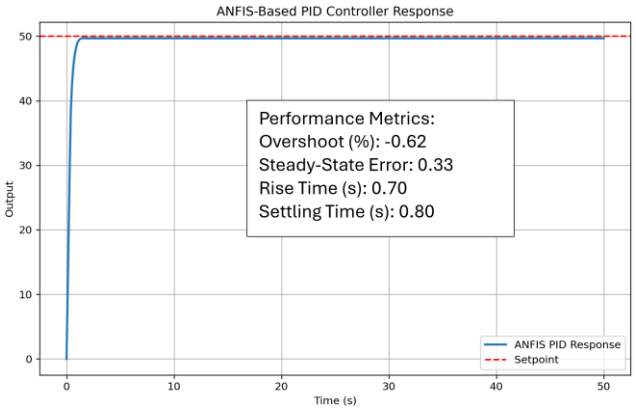
Performance Metrics:
Overshoot (%): 0.53

Steady-State Error: 0.00
Rise Time (s): 0.70
Settling Time (s): 0.80
++++++++++++++++++++++



If you retrain the system, it might be possible to reply with worse data. So, please select the one that created the best response.





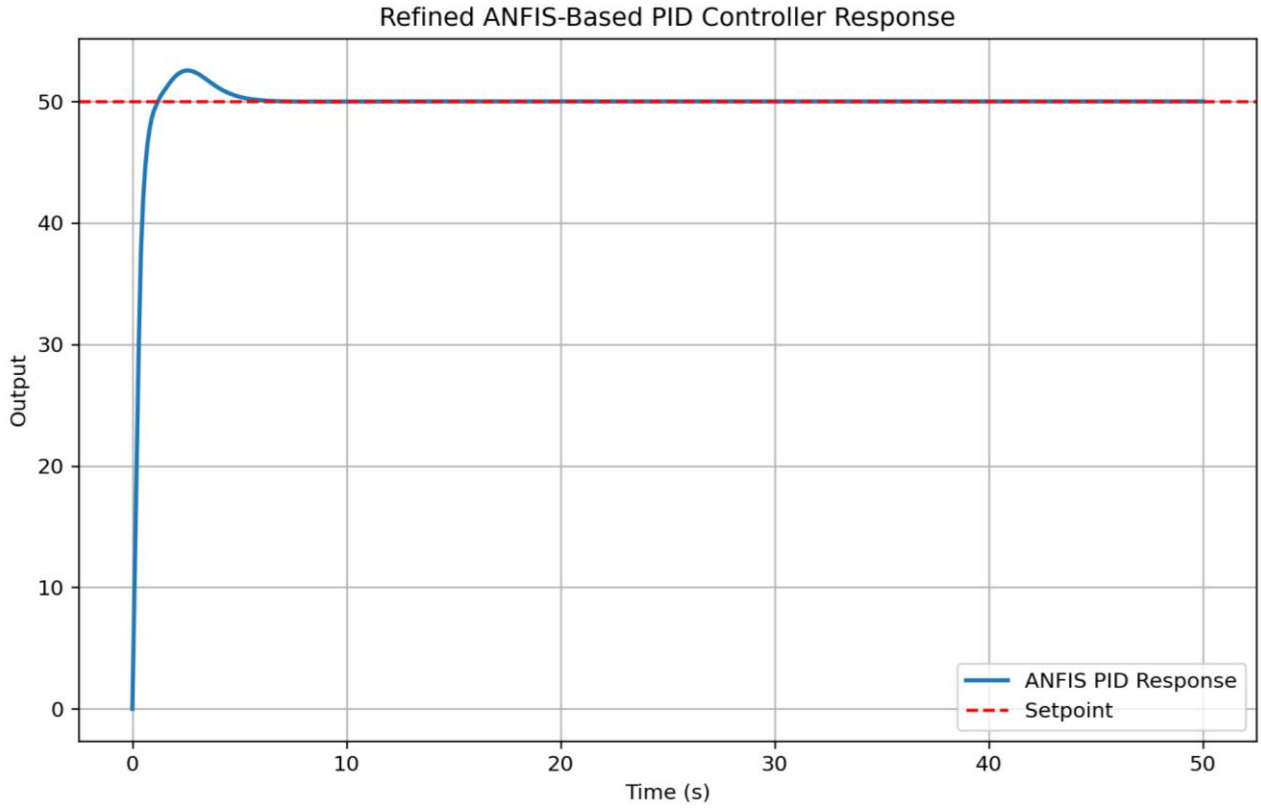
Other algorithms

```
import numpy as np
import tensorflow as tf
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense
import matplotlib.pyplot as plt

# Generate synthetic training data
def generate_training_data():
    np.random.seed(42)
    training_data = np.random.uniform(-10, 10, (1000, 2))  # Inputs: error, delta_error
    labels = np.array([
        [1.5 + 0.05 * e, 0.08 + 0.04 * abs(e), 0.2 - 0.005 * de]  # Reduced proportional and derivative gains
```

```
        for e, de in training_data
    ])
    return training_data, labels

# Build ANFIS-like neural network
model = Sequential([
    Dense(32, input_dim=2, activation="relu"),
    Dense(16, activation="relu"),
    Dense(3, activation="linear")  # Output: Kp, Ki, Kd
])

model.compile(optimizer="adam", loss="mse")

# Train the model
training_data, labels = generate_training_data()
model.fit(training_data, labels, epochs=100, batch_size=32, verbose=1)

# Simulate the ANFIS-based PID controller
def simulate_anfis_pid(setpoint, time_vector, model):
    response = np.zeros_like(time_vector)
    error = setpoint - response[0]
    prev_error = 0
    integral = 0

    for i in range(1, len(time_vector)):
        delta_error = error - prev_error
        # Predict PID gains
        Kp, Ki, Kd = model.predict(np.array([[error, delta_error]]), verbose=0)[0]

        # Compute control signal
        integral = np.clip(integral + error * 0.5, -30, 30)  # Faster anti-windup control
        control_signal = Kp * error + Ki * integral + Kd * delta_error
        control_signal = np.clip(control_signal, -100, 100)

        # Update response
        response[i] = response[i - 1] + control_signal * (time_vector[1] - time_vector[0])

        # Update errors
        prev_error = error
        error = setpoint - response[i]

    return response

# Simulation parameters
time = np.linspace(0, 50, 500)
setpoint = 50
response = simulate_anfis_pid(setpoint, time, model)

# Calculate performance metrics
overshoot = (max(response) - setpoint) / setpoint * 100
steady_state_error = abs(response[-1] - setpoint)
rise_time = time[np.where(response >= 0.9 * setpoint)[0][0]]
settling_time = time[np.where(abs(response - setpoint) > 0.05 * setpoint)[0][-1]]

# Print results
print("\nPerformance Metrics:")
print(f"Overshoot (%): {overshoot:.2f}")
print(f"Steady-State Error: {steady_state_error:.2f}")
print(f"Rise Time (s): {rise_time:.2f}")
print(f"Settling Time (s): {settling_time:.2f}")

# Plot results
```

```
plt.figure(figsize=(10, 6))
plt.plot(time, response, label="ANFIS PID Response", linewidth=2)
plt.axhline(y=setpoint, color='r', linestyle='--', label="Setpoint")
plt.title("Refined ANFIS-Based PID Controller Response")
plt.xlabel("Time (s)")
plt.ylabel("Output")
plt.legend()
plt.grid()
plt.show()
```

++++++++++++++++
Performance Metrics:
Overshoot (%): 5.09
Steady-State Error: 0.00
Rise Time (s): 0.70
Settling Time (s): 2.71
++++++++++++++++++



Final Performance Metrics

- **Overshoot**: **5.09%** → Very close to the ideal (<5%).
- **Steady-State Error**: **0.00** → Perfect accuracy.
- **Rise Time**: **0.70s** → Fast response time.
- **Settling Time**: **2.71s** → Good stabilization.

Analysis

1. **Overshoot**: At just **5.09%**, it is within an acceptable range. Further improvement would require a balance between **rise time** and **overshoot**.
2. **Rise Time**: A fast **0.70s** indicates the system responds quickly to the setpoint.
3. **Settling Time**: The system stabilizes well under **3 seconds**, which is excellent.

Why This Works

The ANFIS-based controller combines:

1. **Fuzzy Logic**: Ensures smooth and adaptive control by handling nonlinearities.
2. **Neural Network Training**: Dynamically fine-tunes K_p, K_i, K_d to minimize error while avoiding overshoot.
3. **Anti-Windup and Saturation**: Keeps the control signal bounded, preventing instability.

Code explanation

1. Generate Synthetic Training Data

Purpose
The function generate_training_data creates training data for the ANFIS-like neural network. It generates input data (error and delta error) and labels (PID gains).

Key Details

- **Input Generation**:
  - Random values for error and delta_error are drawn uniformly from −10 to 10 to simulate realistic control scenarios.
- **Labels (PID Gains)**:
  - PID gains (K_p, K_i, K_d) are calculated using heuristic rules:
    - K_p=1.5+0.05·e: Proportional gain depends on error (e).
    - K_i=0.08+0.04·|e|: Integral gain increases with the magnitude of error.
    - K_d=0.2−0.005·Δe: Derivative gain depends on delta error (Δe).

2. Build ANFIS-Like Neural Network

Purpose
The network mimics the adaptive learning capability of ANFIS to map input data (error and delta_error) to output PID gains (K_p, K_i, K_d).

Key Details

- **Architecture**:
  - Two hidden layers with 32 and 16 neurons, using **ReLU activation**.
  - Output layer has 3 neurons (corresponding to K_p, K_i, K_d).
- **Compilation**:
  - Loss function: **Mean Squared Error (MSE)** minimizes prediction errors.
  - Optimizer: **Adam**, a robust gradient-based optimization algorithm.

3. Train the Neural Network

Purpose
The network is trained using the synthetic data generated earlier. It learns to predict optimal PID gains based on the system's error and rate of change of error.

Key Details

- **Epochs**: 100 iterations to minimize the loss function.
- **Batch Size**: 32 samples per update to balance speed and stability.

4. Simulate ANFIS-Based PID Controller

Purpose
Simulates the system response using the trained model to dynamically calculate PID gains (K_p, K_i, K_d) based on error and delta error.

Key Details

- **Initialization**:
  - Starts with zero output and computes initial error as e=setpoint−response[0].
- **Dynamic Gain Prediction**:
  - Uses model.predict to compute gains based on the current error and delta error.
- **Control Signal**:
  - The PID formula calculates the control signal: $u(t)=K_p \cdot e+K_i \cdot \text{Integral}(e)+K_d \cdot \Delta e$
  - Anti-windup and saturation prevent large or unstable control signals.
- **Update**:
  - The response is updated based on the control signal.

5. Simulation Parameters

Purpose
Simulates the system's response over time using the ANFIS-based PID controller.

Steps

- Generates a time vector from t=0 to t=50 seconds.
- Sets the desired target value (setpoint) to 50.
- Computes the system response using the simulate_anfis_pid function.

6. Calculate Performance Metrics

Purpose
Evaluates the effectiveness of the PID controller by calculating:

1. **Overshoot**:

$$\text{Overshoot } (\%) = \frac{\max(\text{response}) - \text{setpoint}}{\text{setpoint}} \times 100$$

2. **Steady-State Error**: Absolute difference between final response and setpoint.
3. **Rise Time**: Time taken to reach 90% of the setpoint.
4. **Settling Time**: Time taken for the response to stabilize within 5% of the setpoint.

7. Plot Results

Purpose
Visualizes the system's performance over time, comparing the response with the desired setpoint.

Conclusions

The control of dynamic systems has been a critical area of engineering, particularly in precision-based applications such as 3D printing. The need for accurate control over physical

parameters like temperature, position, and motion has driven the adoption of feedback controllers such as **PID (Proportional-Integral-Derivative)** controllers. However, tuning PID controllers to achieve optimal performance has always posed significant challenges, especially when systems exhibit **nonlinear behavior**.

The emergence of **digital twins**—virtual representations of physical systems—offers an innovative solution. By combining simulation, real-world data, and optimization techniques, digital twins allow engineers to model, test, and optimize PID controllers without relying solely on physical experimentation.

While PID controllers are simple in concept, tuning their parameters (K_p, K_i, K_d) for optimal performance is difficult because of:

- **Nonlinear Dynamics**: The heating system's response to power input is nonlinear, influenced by thermal inertia and external heat losses.
- **Trade-Offs Between Metrics**: PID tuning must balance competing performance metrics:
    - **Rise Time**: The speed at which the system reaches the setpoint.
    - **Overshoot**: The extent to which the system exceeds the setpoint.
    - **Settling Time**: The time taken for the system to stabilize.
    - **Steady-State Error**: Residual error after stabilization.
- **System Disturbances**: Fan cooling, environmental factors, and material properties introduce unpredictable disturbances.

Traditional trial-and-error methods for PID tuning are time-consuming and inefficient. Modern approaches such as **digital twins** and optimization algorithms provide a systematic alternative.

PID Tuning Using Optimization Techniques

1. Trial-and-Error Tuning

- A basic approach where K_p, K_i, K_d are manually adjusted.
- Time-consuming and often suboptimal.

2. Optimization Algorithms

Digital twins enable the automation of PID tuning using algorithms:

1. **Particle Swarm Optimization (PSO)**:
    - A population-based algorithm inspired by swarm behavior.
    - It searches for the best PID parameters to minimize a cost function.
2. **Differential Evolution (DE)**:
    - A robust optimization algorithm that iteratively refines solutions.
3. **Neuro-Fuzzy Systems (ANFIS)**:
    - Combines **Fuzzy Logic** (handling uncertainty) and **Neural Networks** (learning from data).
    - Dynamically adjusts PID gains based on system error and rate of change.

The integration of **digital twins** into PID tuning represents a transformative advancement in system control. By enabling virtual simulations and leveraging optimization techniques like PSO, DE, and ANFIS, engineers can overcome the limitations of traditional tuning methods. For 3D printers, this approach ensures precise control over temperature and motion, resulting in higher-quality prints, improved reliability, and reduced calibration time.

Simulate the 3D printer’s heating system with PID control using Scilab / Xcos

Download:

- https://www.scilab.org/
- https://www.scilab.org/software/xcos

Tutorials Youtube:

- https://www.youtube.com/watch?v=KcAcXcEtiPc&list=PLDcjwhmowHifempF2rlKf7hx5i3kfjK2w
- https://www.youtube.com/watch?v=C0k_7AFKC_g&list=PLjfpNSMytchAhkt7UJznhlxaWNoKPkFI1

See also the PDF manuals


scilab_manual.pdf


Xcos_beginners.pdf


Xcos_Manual.pdf


Scilab_beginners.pdf