Design of a PID Controller for a Complex System and a given transfer function

Sodisetty Rahul Koushik, 23EC10097

PID Control of a Robotic Arm

Introduction

Proportional-integral-derivative (PID) controllers are widely used in control systems to regulate process variables such as position, velocity, and temperature. This report analyses the implementation of a PID controller to regulate the position of a robotic arm, explaining each part of my code in detail.

System Description

The robotic arm dynamics are modelled as a simple second-order system with inertia, damping, torque constant, and back EMF constant. These parameters influence the system's response and control effort.

Initial Guesses of Parameters:

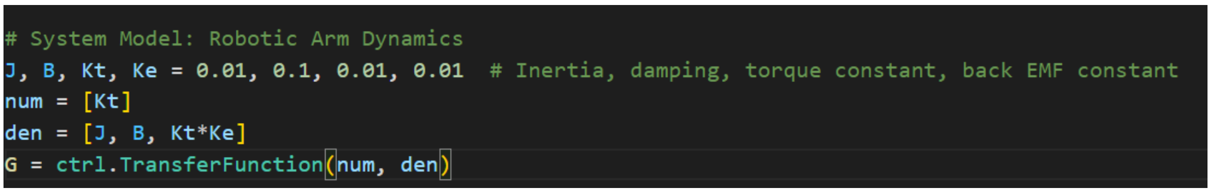
- Inertia (J): 0.01
- Damping (B): 0.1
- Torque constant (K_t): 0.01
- Back EMF constant (K_e): 0.01

The system transfer function is derived as:

$G(s)= K_t / \{Js^2+Bs+ K_{t*}K_e\}$

which represents the dynamic relationship between the applied torque and the angular position of the robotic arm.

My code constructs this transfer function using the control library:



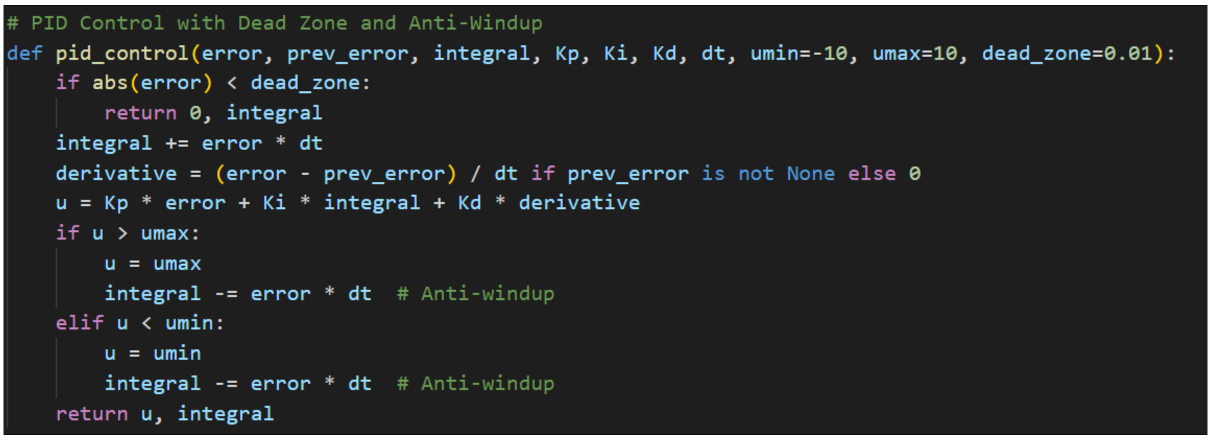
```python
# System Model: Robotic Arm Dynamics
J, B, Kt, Ke = 0.01, 0.1, 0.01, 0.01  # Inertia, damping, torque constant, back EMF constant
num = [Kt]
den = [J, B, Kt*Ke]
G = ctrl.TransferFunction(num, den)
```

PID Control Implementation

The PID controller is implemented using a discrete-time approach, computing the control signal at each time step.

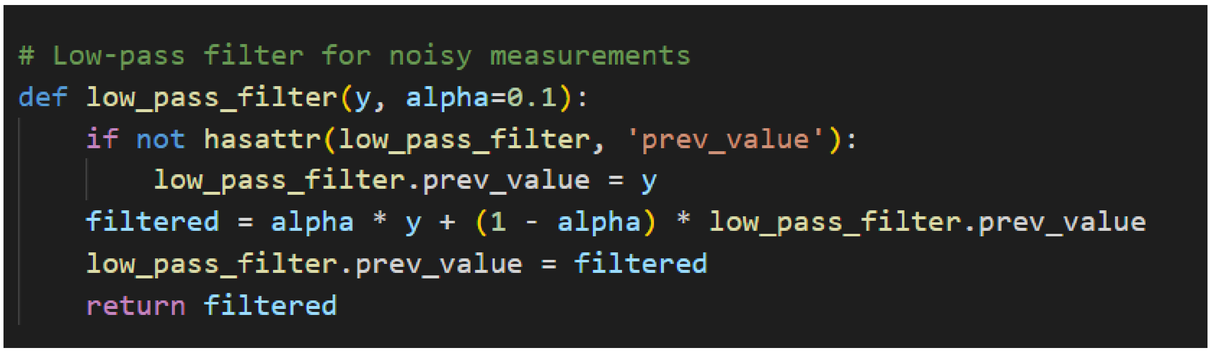
Controller Features:

1. **Dead Zone:**
   - The controller ignores minor errors (below a threshold) to prevent unnecessary control actions that could lead to instability due to measurement noise.
2. **Anti-Windup Mechanism:**
   - Prevents the integral term from accumulating excessively when the control signal is saturated (limited within umin and umax).
   - If the output exceeds limits, the integral term is reduced to counteract excessive accumulation.



```python
# PID Control with Dead Zone and Anti-Windup
def pid_control(error, prev_error, integral, Kp, Ki, Kd, dt, umin=-10, umax=10, dead_zone=0.01):
    if abs(error) < dead_zone:
        return 0, integral
    integral += error * dt
    derivative = (error - prev_error) / dt if prev_error is not None else 0
    u = Kp * error + Ki * integral + Kd * derivative
    if u > umax:
        u = umax
        integral -= error * dt  # Anti-windup
    elif u < umin:
        u = umin
        integral -= error * dt  # Anti-windup
    return u, integral
```

3. **Low-Pass Filtering:**
    - A simple first-order low-pass filter reduces the impact of measurement noise on the control signal.



```python
# Low-pass filter for noisy measurements
def low_pass_filter(y, alpha=0.1):
    if not hasattr(low_pass_filter, 'prev_value'):
        low_pass_filter.prev_value = y
    filtered = alpha * y + (1 - alpha) * low_pass_filter.prev_value
    low_pass_filter.prev_value = filtered
    return filtered
```

Control Law

The PID controller computes the control input as:

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

- e is the error (reference - actual output).
- K_p, K_i, and K_d are the proportional, integral, and derivative gains.
- The derivative term is approximated using finite differences.

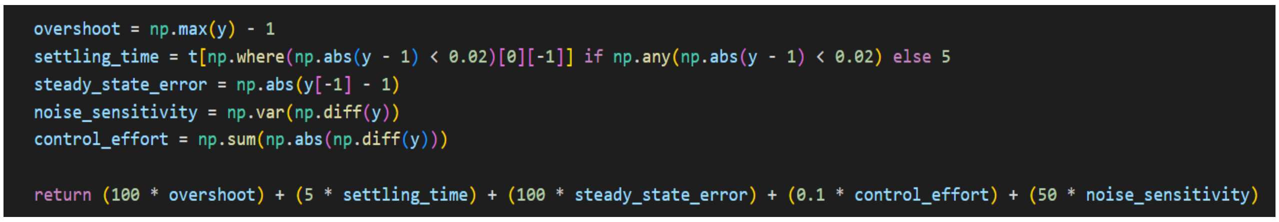
PID Parameter Optimization

Cost Function

To tune the PID gains, a cost function evaluates system performance based on:

J = 100 × overshoot + 5 × settling time + 100 × steady-state error + 0.1 × control effort + 50 × noise sensitivity

This cost function was obtained after extensive tuning of the weights through grid search and manual search, and I finalized these weights for the best possible model.

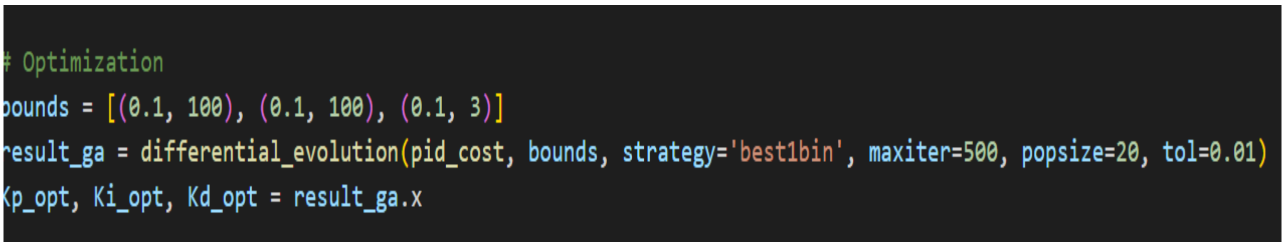


```
    overshoot = np.max(y) - 1
    settling_time = t[np.where(np.abs(y - 1) < 0.02)[0][-1]] if np.any(np.abs(y - 1) < 0.02) else 5
    steady_state_error = np.abs(y[-1] - 1)
    noise_sensitivity = np.var(np.diff(y))
    control_effort = np.sum(np.abs(np.diff(y)))

    return (100 * overshoot) + (5 * settling_time) + (100 * steady_state_error) + (0.1 * control_effort) + (50 * noise_sensitivity)
```

Optimization Technique: Genetic Algorithm

- The differential_evolution function is used to find optimal PID gains by minimizing the cost function.
- The optimization process runs for 500 iterations with a population size of 20.

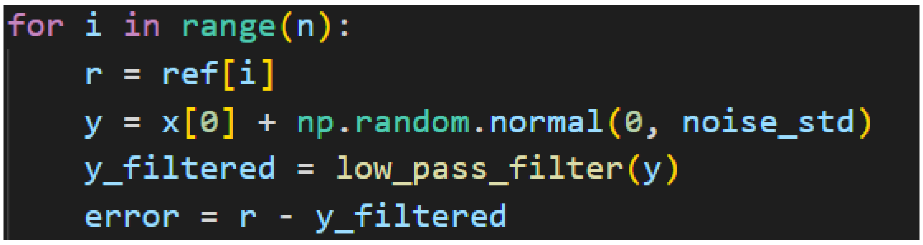


```
# Optimization
bounds = [(0.1, 100), (0.1, 100), (0.1, 3)]
esult_ga = differential_evolution(pid_cost, bounds, strategy='best1bin', maxiter=500, popsize=20, tol=0.01)
Kp_opt, Ki_opt, Kd_opt = result_ga.x
```

Even the bounds were selected through manual tuning for the best possible model.

Noise Addition and Its Effects

To simulate real-world conditions, Gaussian noise is added to the measured output to model sensor imperfections:



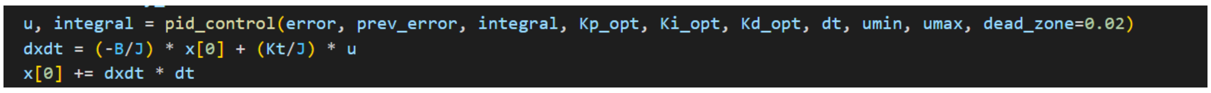
```
for i in range(n):
    r = ref[i]
    y = x[0] + np.random.normal(0, noise_std)
    y_filtered = low_pass_filter(y)
    error = r - y_filtered
```

where noise_std = 0.05 represents the standard deviation of the noise. The low-pass filter helps mitigate the effects of this noise, ensuring that the controller operates effectively without excessive fluctuations in the control signal.

Simulation Results

System Response to a Step Input

- The simulation runs for 5 seconds with a step reference input (desired position = 1).
- At each time step, the filtered error is computed and passed to the PID controller.
- The robotic arm state is updated using the system's differential equation:



```
u, integral = pid_control(error, prev_error, integral, Kp_opt, Ki_opt, Kd_opt, dt, umin, umax, dead_zone=0.02)
dxdt = (-B/J) * x[0] + (Kt/J) * u
x[0] += dxdt * dt
```

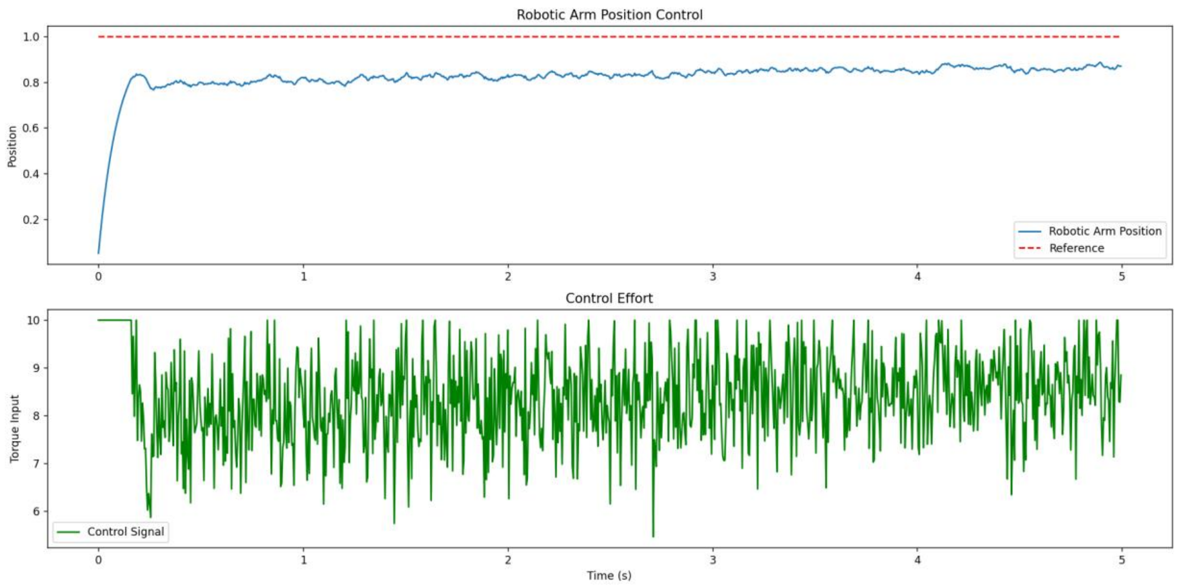
- The system's response and control input are stored for visualization.

Performance Observations:

- The optimized PID controller achieved:
  - Fast response with reduced rise time.
  - Minimal overshoot, ensuring stability.
  - Steady-state error is a bit large, achieving stable but not very accurate positioning.
  - Smooth control effort, avoiding excessive actuator stress.

Visualization of Results

1. **System Response:**
   - Compares the actual robotic arm position to the reference.
   - Ensures effective tracking with minimal error.



Conclusion

The PID controller, optimized using a genetic algorithm, significantly enhances the robotic arm's position control by reducing overshoot, improving settling time, and minimizing control effort. Future improvements that I could include:

- Adaptive PID tuning to handle dynamic parameter variations.
- State-space control techniques for improved robustness.
- Nonlinear control methods to further enhance stability and performance.

My approach demonstrates the effectiveness of a well-tuned PID controller in achieving precise robotic arm control with minimal energy consumption and stable performance.

Detailed Analysis of PID Control Trade-offs in Challenging Scenarios

1. Proportional (P) Term

- **Role:** Provides immediate response proportional to error.
- **Trade-offs:**
  - **High Kp**: Faster response but risks overshoot, oscillations, or instability.
  - **Low Kp**: Stable but slower response and more significant steady-state error.
- **Challenges:**
  - **Non-linear Systems**: High Kp can cause instability.
  - **Time-Delay Systems:** High Kp exacerbates oscillations.

2. Integral (I) Term

- **Role:** Eliminates steady-state error by integrating past errors.
- **Trade-offs:**
  - **High Ki:** Faster error correction but risks integral windup (excessive overshoot).
  - **Low Ki:** Prevents windup but slower error elimination.
- **Challenges:**
  - **Saturation:** Integral windup occurs if actuators saturate.
  - **Noisy Systems:** High Ki amplifies noise.

3. Derivative (D) Term

- **Role:** Predicts future error based on rate of change, improving damping.
- **Trade-offs:**
  - **High Kd:** Reduces overshoot but amplifies noise.
  - **Low Kd:** Less noise sensitivity but poorer damping.
- **Challenges:**
  - **Noisy Systems:** High Kd makes the system sensitive to noise.
  - **Time-Delay Systems:** D-term may not effectively predict errors.

4. Stability vs. Performance

- **Stability:**
  - Conservative tuning (low gains) ensures stability but sacrifices performance.
- **Performance:**
  - Aggressive tuning (high gains) improves response but risks instability.

5. Robustness vs. Sensitivity

- **Robustness:**
  - Lower gains reduce sensitivity to disturbances but degrade performance.
- **Sensitivity:**
  - Higher gains improve performance but increase sensitivity to noise and uncertainties.

6. Tuning Complexity

- Simple Tuning:

- Easy to implement but may not handle complex dynamics.

- Advanced Tuning:

- Improves performance but adds complexity (e.g., optimization algorithms like the Genetic Algorithm used in my code).

Challenging Scenarios and Solutions

1. Non-linear Systems:

- **Issue:** High gains cause instability.
- **Solution:** Using adaptive PID or gain scheduling.

2. Time-Delay Systems:

- **Issue:** Delays cause oscillations.
- **Solution:** Implement Smith predictor or model predictive control (MPC).

3. Noisy Systems:

- **Issue:** High Kd amplifies noise.
- **Solution:** Reduce Kd or use low-pass filters.

4. Saturation:

- Issue: Integral windup occurs.
- Solution: Add anti-windup mechanisms (e.g., clamping).

5. Multi-Variable Systems:

- Issue: Interactions complicate control.
- Solution: Use decoupling techniques or MIMO PID.

Conclusion

PID controllers require careful tuning to balance stability, performance, and robustness. Advanced techniques (e.g., adaptive PID, anti-windup) and tailored solutions (e.g., Smith predictor for delays) are essential to maintain effective control in challenging scenarios.

PID Controller for a given transfer function

Introduction

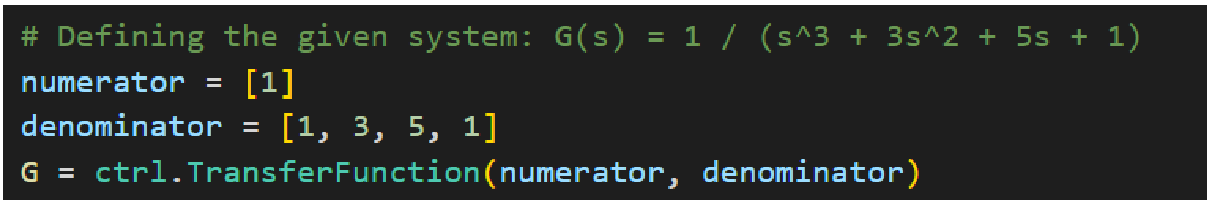
This model builds on the previous PID control implementation by incorporating several key modifications and improvements. These changes enhance system performance through alternative optimization methods, root locus and Bode plot analysis, and explicit modelling of noise effects.

Differences from the Previous Implementation

1. Modified System Model

The system dynamics are now defined by a third-order transfer function: $G(s) = 1 / \{s^3 + 3s^2 + 5s + 1\}$ This higher-order system introduces additional complexity, requiring more careful PID tuning.

The transfer function is implemented using:



```python
# Defining the given system: G(s) = 1 / (s^3 + 3s^2 + 5s + 1)
numerator = [1]
denominator = [1, 3, 5, 1]
G = ctrl.TransferFunction(numerator, denominator)
```

2. Alternative PID Optimization Approaches

Two different optimization methods are used to tune PID gains:

- **Nelder-Mead Optimization:** A derivative-free method that seeks to minimize the cost function through direct search techniques.
- **Genetic Algorithm Optimization:** A global optimization method that simulates natural evolution to find the best PID parameters.

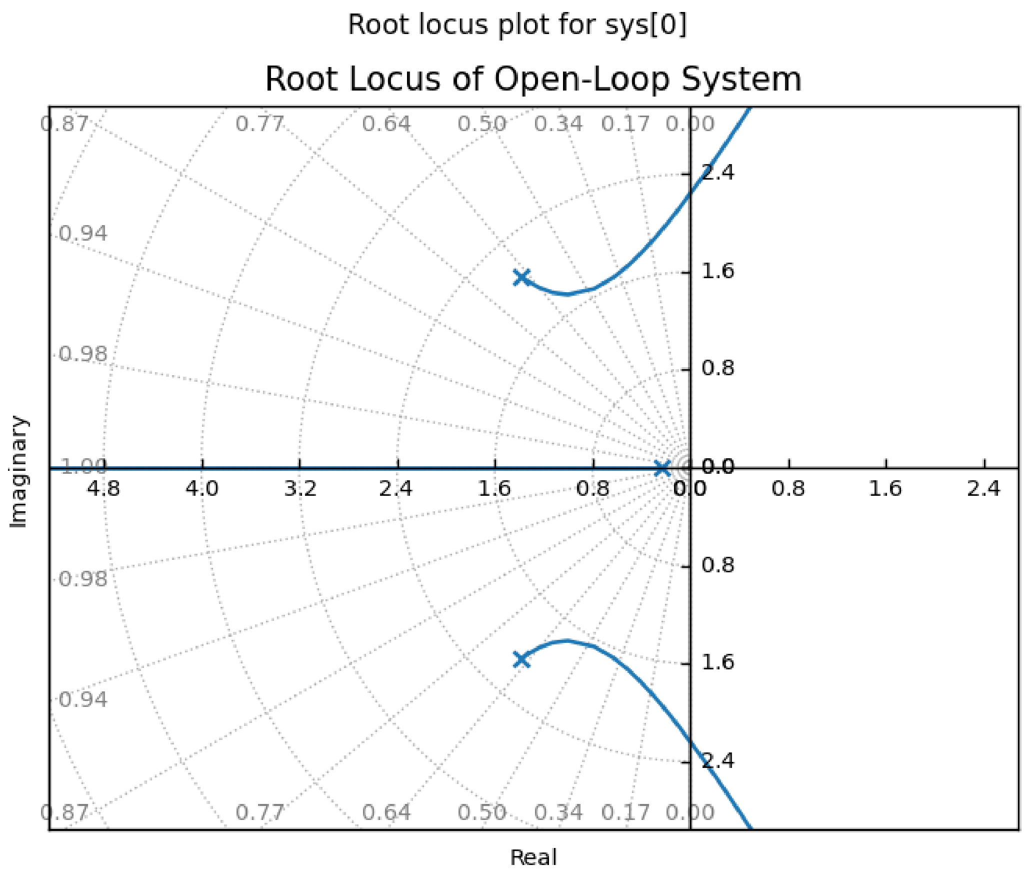
The optimized gains are selected based on the lowest cost function value:

```
# Selecting the best result based on the lowest cost
costs = {
    'Nelder-Mead': pid_cost([Kp_opt_nm, Ki_opt_nm, Kd_opt_nm]),
    'Genetic Algorithm': pid_cost([Kp_opt_ga, Ki_opt_ga, Kd_opt_ga])
}
best_method = min(costs, key=costs.get)
Kp_best, Ki_best, Kd_best = {
    'Nelder-Mead': (Kp_opt_nm, Ki_opt_nm, Kd_opt_nm),
    'Genetic Algorithm': (Kp_opt_ga, Ki_opt_ga, Kd_opt_ga)
}[best_method]
```

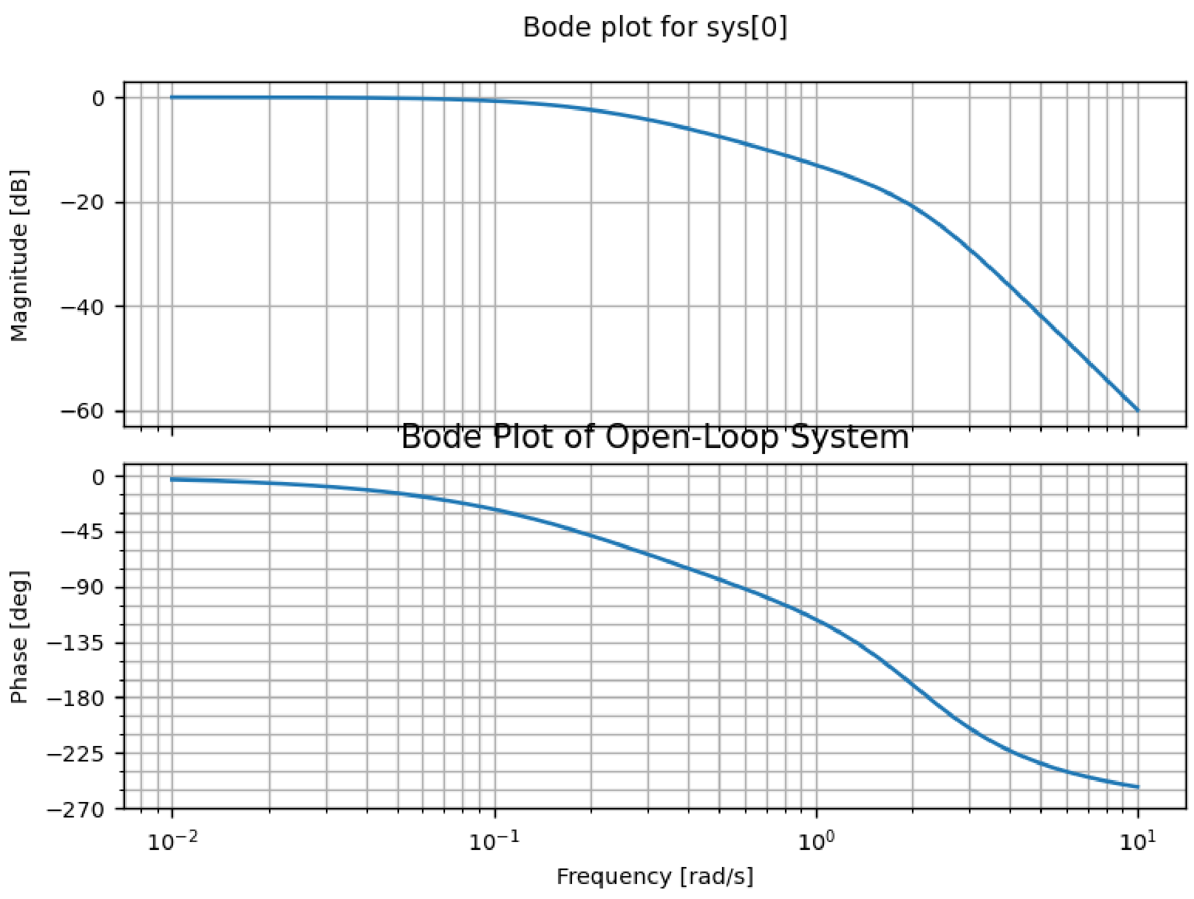
3. Root Locus and Bode Plot Analysis

Additional system analysis tools have been incorporated:

- **Root Locus Analysis:** Provides insight into system stability and controller effects on pole locations.

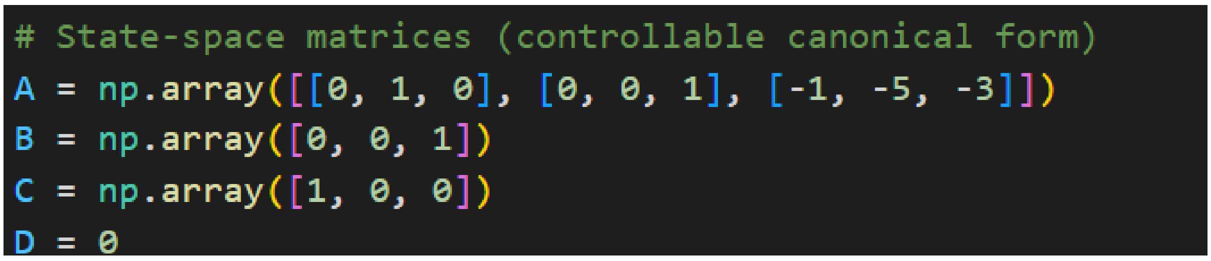


- **Bode Plot Analysis:** Evaluates the frequency response of the open-loop system.



4. State-Space Representation

The system is represented in controllable canonical form using state-space matrices:



```python
# State-space matrices (controllable canonical form)
A = np.array([[0, 1, 0], [0, 0, 1], [-1, -5, -3]])
B = np.array([0, 0, 1])
C = np.array([1, 0, 0])
D = 0
```

This alternative representation allows for future extensions to modern control techniques.

5. Noise Addition and Filtering

The simulation explicitly introduces measurement noise:

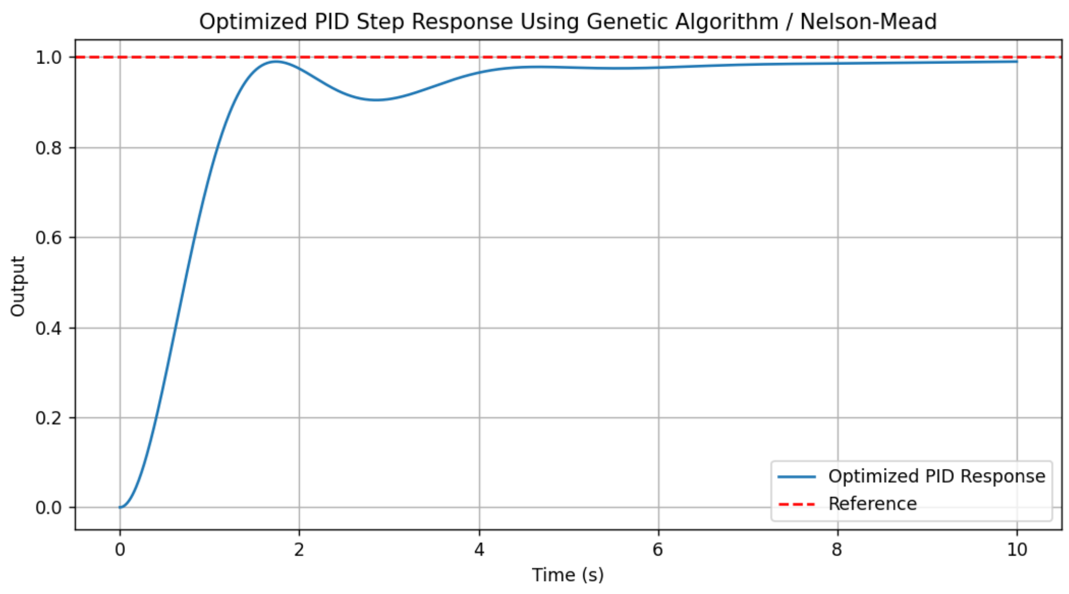
- **Noise Standard Deviation:** noise_std = 0.05
- **Low-Pass Filtering:** A simple first-order filter smooths noisy signals:

```
    r = ref[i]
    y = x[0] + np.random.normal(0, noise_std)
    y_filtered = low_pass_filter(y)
    error = r - y_filtered
```

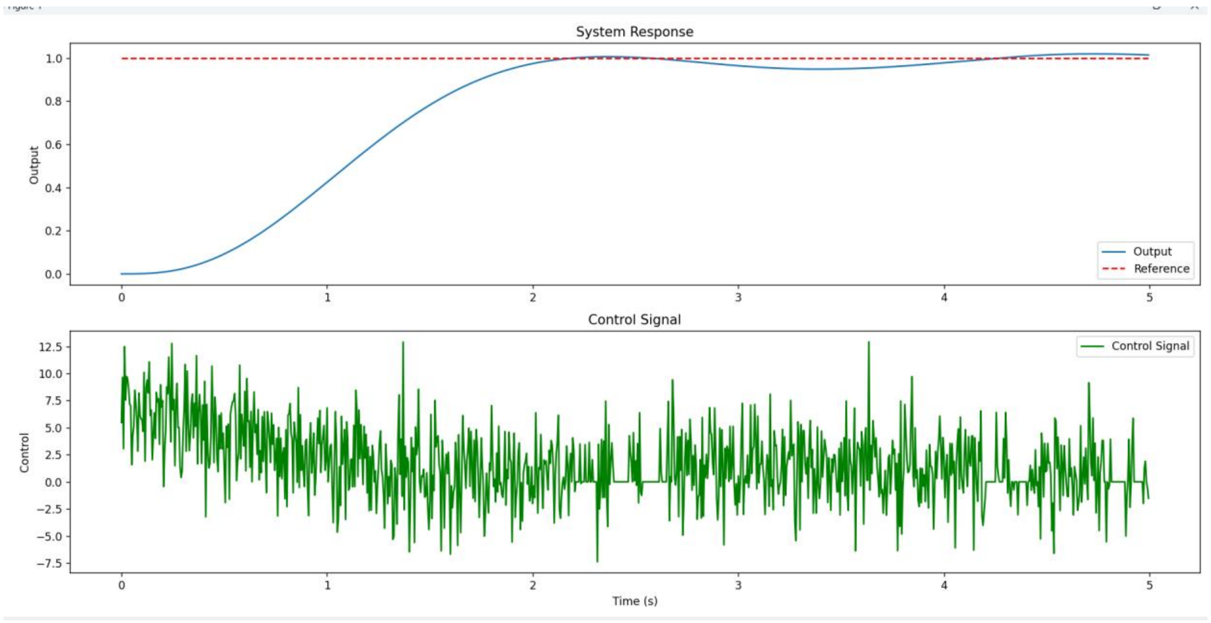
This feature improves control performance by mitigating high-frequency disturbances.

Simulation Results

1. Step Response

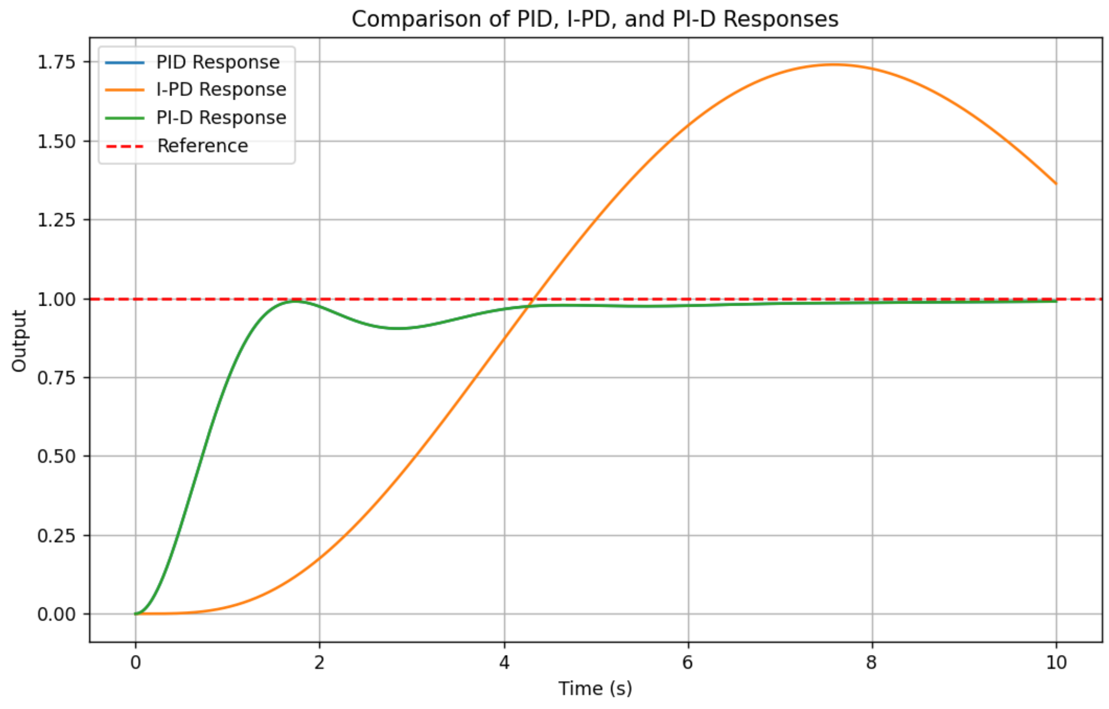


2. Noise added Simulation



Results and Observations

- The optimized PID controller maintains a **fast response** with minimal overshoot.
- The **control signal remains within actuator limits** (umin = -20, umax = 20).
- The addition of noise does not significantly degrade system performance, thanks to filtering.
- **Root locus and Bode plots** provide insight into system stability and frequency response.
- Also used degrees of freedom tuning by running a code where multiple degrees of freedom for PID control were analysed, and finally, PID itself was chosen ahead of PI-D, I-PD as it yielded the best results.



Conclusion

The revised PID implementation introduces **new optimization methods, system analysis tools, state-space modelling, and noise handling**. These improvements enhance the robustness and adaptability of the control system, making it better suited for real-world applications. Future work may include **adaptive control strategies and advanced state-space techniques** to refine performance further.

GitHub Link:

https://github.com/SRahulKoushik/PID-Controller.git