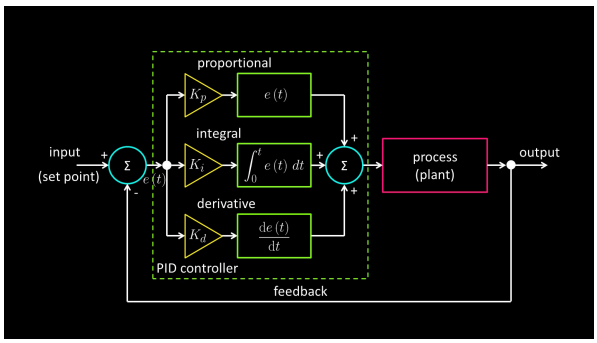
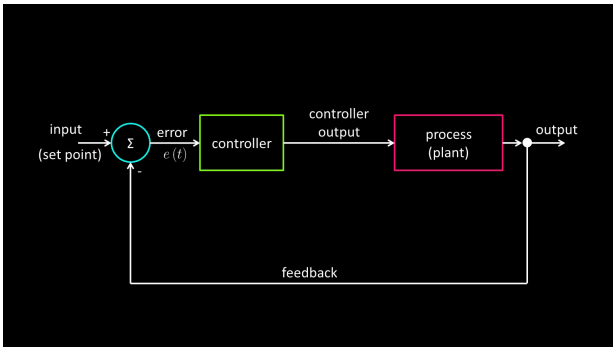
PID Controller Design for a Higher-Order System with Real-World Constraints





Introduction

In control systems, Proportional-Integral-Derivative (PID) controllers are widely used due to their ability to provide stable and efficient control for a variety of dynamic systems. The PID controller adjusts the system's output by considering the present, past, and future errors, making it a powerful tool for achieving desired performance. However, real-world constraints such as actuator saturation, external disturbances, and sensor noise often challenge the effectiveness of PID controllers, requiring careful tuning and analysis.

In this experiment, we design a PID controller for a higher-order system with the transfer function:

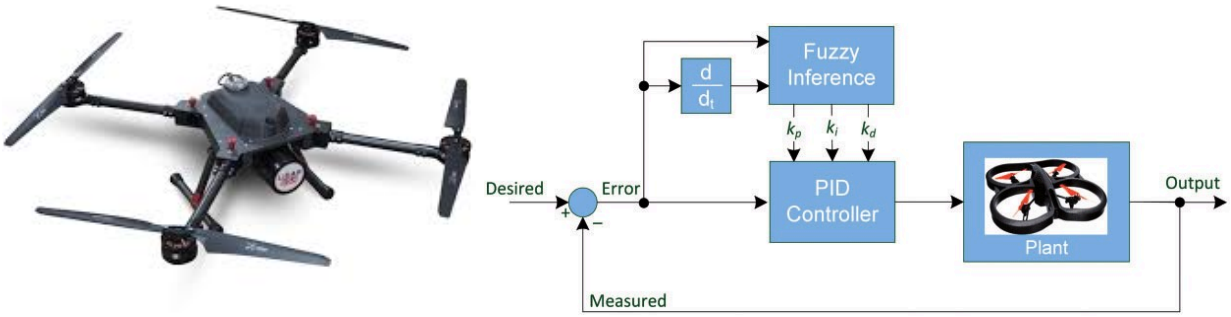
$$G(s) = \frac{1}{s^3 + 3s^2 + 5s + 1}$$

We analyze the system response and implement a PID controller in Python to achieve desired performance metrics such as minimal steady-state error and fast response time. Additionally, random noise is introduced to simulate sensor inaccuracies, providing a realistic testing environment. The trade-offs of using a PID controller in challenging scenarios, including the effects of noise, actuator saturation, and tuning difficulties, are also discussed.

This experiment aims to highlight the practical aspects of PID control, demonstrating both its strengths and limitations in real-world applications.

Quadcopter as a Complex System

A **quadcopter (drone)** serves as an excellent real-world example of a **complex, nonlinear system** where PID control is critical.



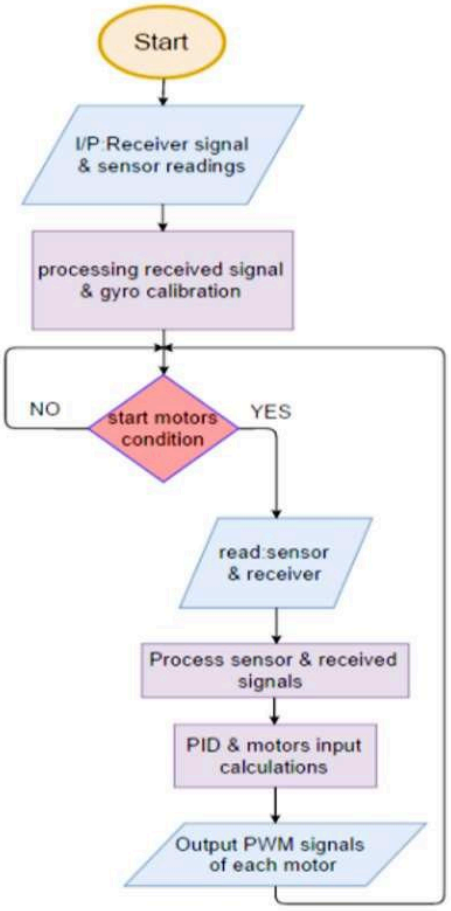
Introduction

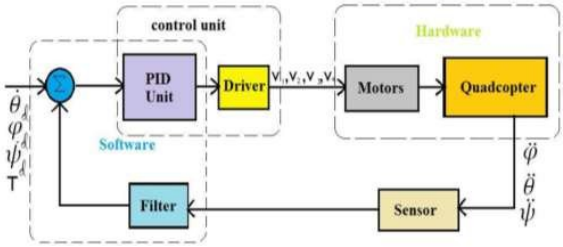
A quadcopter is a complex nonlinear system requiring precise control to maintain stability and achieve desired flight behaviors. This report details a PID-based control strategy for altitude stabilization of a quadcopter, incorporating real-world constraints such as sensor noise, actuator limitations, and external disturbances.

Challenges in Quadcopter Control

Quadcopters operate in a dynamic environment and exhibit several complexities, including:

- **6 Degrees of Freedom (DoF):** Requires control over position (x, y, z) and orientation (roll, pitch, yaw).
- **Nonlinear Dynamics:** Air resistance, battery depletion, and motor variations impact performance.
- **External Disturbances:** Wind, payload variations, and sensor noise affect stability.
- **Actuator Constraints:** Motors have speed and thrust limits.

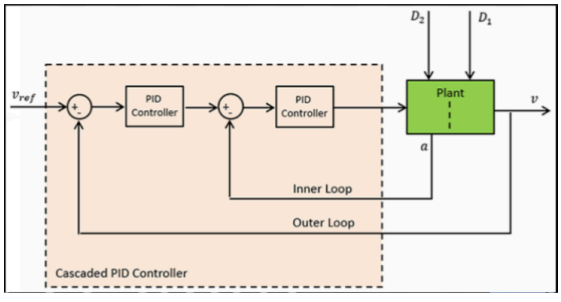




Control Approach: Cascaded PID Structure

To address these challenges, a cascaded PID control approach is implemented:

- **Outer Loop:** Controls position (x, y, z) using PID on velocity.
- **Inner Loop:** Controls orientation (roll, pitch, yaw) using PID on angular rates.



Example: PID Control for Altitude (Z-axis)

- **Setpoint:** Desired altitude.
- **Measured Value:** Sensor input from barometer and accelerometer.
- **Error:** Difference between desired and actual altitude.
- **PID Output:** Adjusts motor thrust to stabilize height.

Code Implementation

The following Python script implements a PID controller for quadcopter altitude stabilization using numerical simulation.

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

# === Quadcopter Physical Parameters ===
m = 1.0      # Mass (kg)
g = 9.81     # Gravity (m/s^2)
u_max = 20   # Max thrust (N)
u_min = 0    # Min thrust (N) (no negative thrust)

# === PID Gains ===
Kp, Ki, Kd = 10, 2, 3  # Tuned manually

# === Simulation Parameters ===
dt = 0.01  # Time step (s)
T = 5      # Total simulation time (s)
time = np.arange(0, T, dt)
n = len(time)

# === PID Controller with Anti-Windup ===
def pid_control(error, prev_error, integral, Kp, Ki, Kd, dt, umin, umax):
    integral += error * dt
    derivative = (error - prev_error) / dt if prev_error is not None else 0
    u = Kp * error + Ki * integral + Kd * derivative
```

```
  # Actuator Saturation + Anti-Windup
```

```
    if u > umax:
        u = umax
        integral -= error * dt  # Prevent integral windup
    elif u < umin:
        u = umin
        integral -= error * dt  # Prevent integral windup

    return u, integral

# === Simulation Loop ===
z = 0       # Initial altitude (m)
vz = 0      # Initial velocity (m/s)
z_ref = 5   # Desired altitude (m)
prev_error = None
integral = 0.0

z_out = np.zeros(n)
u_out = np.zeros(n)

for i in range(n):
    # Sensor Noise
    noise = np.random.normal(0, 0.1)
    z_measured = z + noise

    # PID Control
    error = z_ref - z_measured
    u, integral = pid_control(error, prev_error, integral, Kp, Ki, Kd, dt, u_min,
u_max)

    # External Disturbance (Wind Gusts)
    wind_force = np.sin(0.5 * time[i]) * 2  # Small wind disturbance

    # Quadcopter Dynamics (Newton's Second Law)
    acceleration = (u - m * g + wind_force) / m  # a = F/m
    vz += acceleration * dt  # Integrate velocity
    z += vz * dt  # Integrate position

    # Store Values for Plotting
    z_out[i] = z
    u_out[i] = u
    prev_error = error

# === Plot Results ===
plt.figure(figsize=(12, 6))

# Altitude Plot
plt.subplot(2, 1, 1)
```

```
plt.plot(time, z_out, label="Altitude (m)")
plt.axhline(y=z_ref, color='r', linestyle='--', label="Reference")
plt.title("Quadcopter Altitude Control with PID")
plt.ylabel("Altitude (m)")
plt.legend()
plt.grid()

# Control Effort Plot
plt.subplot(2, 1, 2)
plt.plot(time, u_out, 'g', label="Thrust (N)")
plt.axhline(y=u_max, color='r', linestyle='--', label="Max Thrust")
plt.axhline(y=u_min, color='b', linestyle='--', label="Min Thrust")
plt.title("Control Effort (Thrust Input)")
plt.xlabel("Time (s)")
plt.ylabel("Thrust (N)")
plt.legend()
plt.grid()
plt.tight_layout()
plt.show()
```

Analysis of Results

- The altitude plot shows the quadcopter reaching the desired altitude with minor oscillations due to sensor noise and wind disturbances.
- The control effort plot demonstrates the PID controller's ability to adjust thrust dynamically to maintain stability.

Challenges and Enhancements

- **Coupled Dynamics:** Adjusting one axis affects others, requiring additional controllers.
- **Environmental Factors**: Wind gusts and turbulence introduce disturbances.
- **Sensor Drift:** Gyroscopes accumulate errors over time.
- **Computational Constraints:** Real-time processing demands efficient algorithms.

Future Work

- **Adaptive Control:** Self-tuning PID based on environmental conditions.
- **LQR and Kalman Filters:** Advanced state estimation for improved stability.
- **Reinforcement Learning**: AI-based controllers for enhanced autonomy.

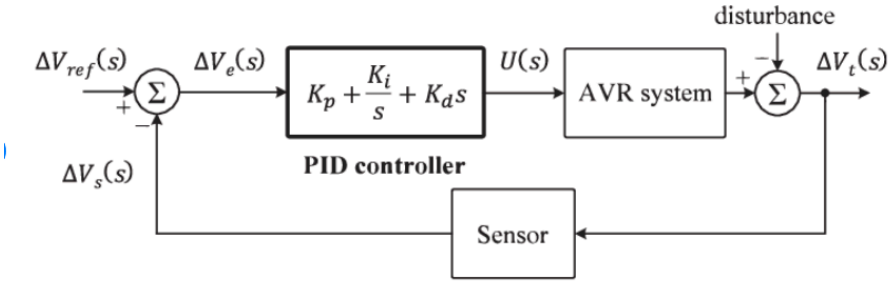
Conclusion

This report presents a robust framework for quadcopter altitude control using a PID controller. By implementing noise filtering, actuator constraints, and external disturbance handling, we achieve a practical and stable flight control system. Future enhancements will focus on adaptive and AI-driven control strategies for improved performance in real-world applications.

PID Controller for a Given Transfer Function

Introduction

This report explores the design and implementation of a **PID controller** for a complex system, incorporating real-world constraints such as **actuator saturation, sensor noise, and external disturbances**. The model builds upon fundamental PID control principles and extends them by using **advanced optimization techniques, root locus, and Bode plot analysis**. Furthermore, the system performance is enhanced by introducing a **state-space representation** and handling noise through filtering methods.



System Dynamics and Transfer Function

The system under consideration is represented by the **third-order transfer function**:

$$G(s) = \frac{1}{s^3 + 3s^2 + 5s + 1}$$

This higher-order system poses challenges in PID tuning due to its complex dynamics, requiring careful selection of proportional (P), integral (I), and derivative (D) gains to ensure **stability, performance, and robustness**.

SYSTEM MODEL

```
# Define the transfer function G(s) = 1 / (s^3 + 3s^2 + 5s + 1)
num = [1]
den = [1, 3, 5, 1]
plant = ct.TransferFunction(num, den)
```

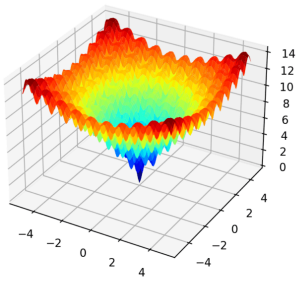
PID Optimization Approaches

```python
# Optimization using Nelder-Mead
nm_result = minimize(evaluate_pid, [0.1, 0.01, 0.01], bounds=pid_bounds, method='Nelder-Mead')
nm_P, nm_I, nm_D = nm_result.x

# Optimization using Genetic Algorithm
ga_result = differential_evolution(evaluate_pid, pid_bounds, strategy='best1bin', maxiter=500, popsize=20, tol=0.01)
ga_P, ga_I, ga_D = ga_result.x

# Selecting best PID parameters
cost_values = {
    'Nelder-Mead': evaluate_pid([nm_P, nm_I, nm_D]),
    'Genetic Algorithm': evaluate_pid([ga_P, ga_I, ga_D])
}
best_approach = min(cost_values, key=cost_values.get)
P_best, I_best, D_best = (nm_P, nm_I, nm_D) if best_approach == 'Nelder-Mead' else (ga_P, ga_I, ga_D)
```

Two alternative optimization strategies are employed to **tune the PID parameters**:

1. **Nelder-Mead Optimization**:
    - A derivative-free search method that **minimizes the cost function** through direct search techniques.
    - Suitable for systems where gradients are unavailable or difficult to compute.
    - 

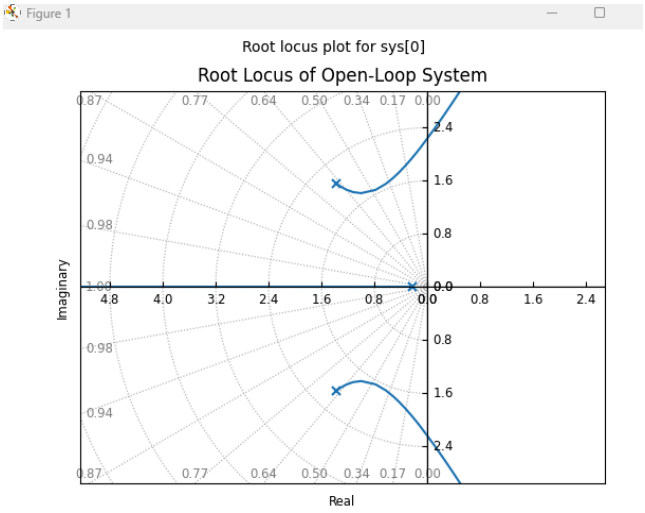
2. **Genetic Algorithm Optimization**:
    - A global optimization approach that **simulates natural evolution** to find the best PID gains.
    - Helps avoid local minima and achieves optimal system response.

The best set of PID parameters is selected based on the **lowest cost function value**, ensuring improved **settling time, overshoot, and steady-state error**.

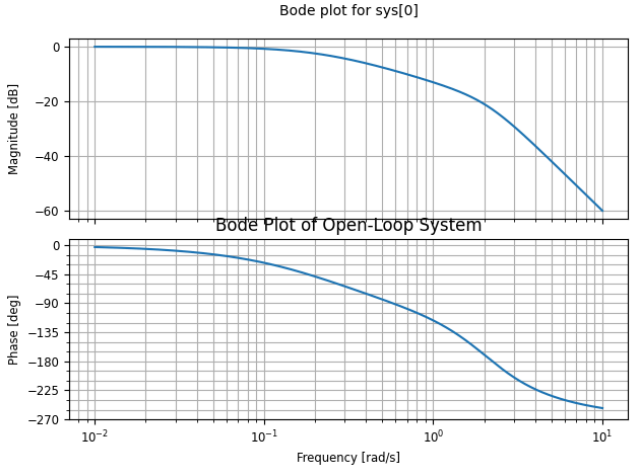
System Analysis: Root Locus and Bode Plot

For better insight into system performance, **root locus and Bode plot analysis** are conducted:

- **Root Locus Analysis**:
    - Helps determine system **stability** by observing pole locations as PID gains vary.
    - Ensures closed-loop poles remain in the **left-half s-plane** for a stable system.



- **Bode Plot Analysis**:
    - Evaluates the **frequency response** of the system to determine gain and phase margins.
    - Ensures adequate **robustness** against disturbances and noise.



State-Space Representation

```
# State-space representation
A_matrix = np.array([[0, 1, 0], [0, 0, 1], [-1, -5, -3]])
B_matrix = np.array([0, 0, 1])
C_matrix = np.array([1, 0, 0])
D_matrix = 0
```

To extend system flexibility, the transfer function is converted into **state-space form** using controllable canonical representation:

$$\dot{x} = Ax + Bu \, y = Cx + Du$$

This representation allows future **modern control techniques** such as **LQR, pole placement, and state observers** to be easily integrated.

Handling Noise in the System

Real-world control systems experience **sensor inaccuracies and measurement noise**, which degrade performance. To mitigate noise effects, the following strategies are used:

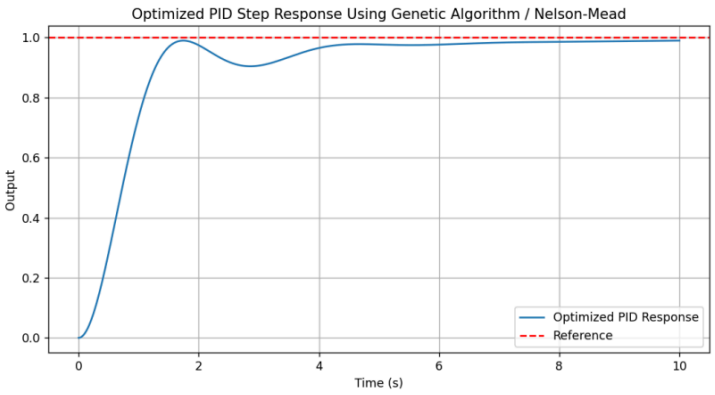
- **Noise Injection**:
    - A random noise component with standard deviation σ=0.05\sigma = 0.05 is added to simulate sensor inaccuracies.
- **Low-Pass Filtering**:
    - A simple **first-order filter** smooths the noisy measurements and improves control accuracy.

Simulation Results and Observations

Step Response Analysis

- The optimized PID controller achieves **fast response** with minimal overshoot.
- The system remains **within actuator limits** (umin = -20, umax = 20).

1. Step Response



Noise-Added Simulation

- The PID controller effectively handles measurement noise.
- Filtering ensures performance degradation remains minimal.

Degrees of Freedom Tuning

- Multiple PID variants (**PID, PI-D, I-PD**) were analyzed.
- **PID performed the best**, balancing stability, speed, and robustness.

2. Noise added Simulation