Control Systems: A Journey Through Feedback and Compensation

Harsh Vardhan

1 Beginning: Open Loop to Closed Loop

We started with **open-loop systems**, which lack feedback mechanisms, leading to instability and errors as they cannot self-correct when disturbances occur. Without knowing the actual output, these systems operate blindly.

Recognizing these limitations, we moved to **closed-loop systems** that use feedback to correct errors. The feedback loop compares actual output with desired reference: $E(s) = R(s) - H(s)Y(s)$, where $E(s)$ is error, $R(s)$ is reference, $Y(s)$ is output, and $H(s)$ is feedback transfer function. This enabled systems to reach desired states earlier and more precisely, automatically compensating for disturbances.

2 Understanding System Response: Transfer Functions and S-Plane

Moving ahead, we learned **response analysis** and **transfer functions**: $G(s) = Y(s)/U(s) = \frac{b_m s^m + \cdots + b_0}{a_n s^n + \cdots + a_0}$. We learned converting differential equations to transfer functions using Laplace transform with zero initial conditions.

The **correlation between response time plots and s-plane location** revealed how pole-zero positions determine behavior:

- Left half plane poles: stable ($\mathrm{Re}(s) < 0$)
- Right half plane poles: unstable ($\mathrm{Re}(s) > 0$)
- Imaginary axis: marginally stable ($\mathrm{Re}(s) = 0$)
- Distance from origin affects response speed

3 System Classification and Tracking Ability

We learned **system types** (0, 1, 2) based on integrators: $G(s) = K(s + z_1)\cdots/(s^N(s + p_1)\cdots)$ where N is the type.

Using **Final Value Theorem** for steady-state error: $e_{ss} = \lim_{s\to 0} sE(s)$

For unity feedback: $E(s) = R(s)/(1 + G(s))$

Tracking ability for inputs:

- Step input $R(s) = A/s$: Type 0 has error $A/(1 + K_p)$; Type 1,2 track perfectly
- Ramp input $R(s) = A/s^2$: Type 0 fails; Type 1 has error A/K_v; Type 2 perfect
- Parabolic $R(s) = A/s^3$: Type 0,1 fail; Type 2 has error A/K_a

where $K_p = \lim_{s\to 0} G(s)$, $K_v = \lim_{s\to 0} sG(s)$, $K_a = \lim_{s\to 0} s^2 G(s)$

4 First and Second Order Analysis

We analyzed **1st order**: $G(s) = K/(\tau s + 1)$ and **2nd order**: $G(s) = \omega_n^2/(s^2 + 2\zeta\omega_n s + \omega_n^2)$

Found formulas for response properties:

First Order:

- Rise time: $t_r \approx 2.2\tau$
- Settling time: $t_s = 4\tau$ (2%) or 3τ (5%)

Second Order (underdamped, $0 < \zeta < 1$):

Poles: $s_{1,2} = -\zeta\omega_n \pm j\omega_n\sqrt{1 - \zeta^2}$

- Rise time: $t_r \approx 1.8/\omega_n$ or $t_r = (\pi - \cos^{-1}\zeta)/\omega_d$
- Peak time: $t_p = \pi/\omega_d = \pi/(\omega_n\sqrt{1 - \zeta^2})$
- Settling time: $t_s = 4/(\zeta\omega_n)$ (2%) or $3/(\zeta\omega_n)$ (5%)

- Overshoot: $\%OS = e^{-\pi\zeta/\sqrt{1-\zeta^2}} \times 100\%$

where $\omega_d = \omega_n\sqrt{1-\zeta^2}$ is damped natural frequency.

5 Introducing PID Control

Having understood system behavior, we started with **PID control** to actively shape the response. The PID controller combines three actions:

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

5.1 Integral (I) Component: The Steady-State Problem

I (Integral) reduces steady-state error by accumulating: $u_I(t) = K_i \int_0^t e(\tau)d\tau$

However, it adds oscillations. **Windup issue**: At actuator saturation, integrator keeps accumulating, causing severe overshoot upon desaturation.

Our Solution: Anti-windup caps integration at saturation:

$$\int e(t)dt = \begin{cases} \text{Saturation limit} & \text{if actuator saturated} \\ \int e(t)dt & \text{otherwise} \end{cases}$$

But oscillations persisted and overshoot remained high.

5.2 Proportional (P) Component: The Speed-Stability Trade-off

P (Proportional): $u_P(t) = K_p e(t)$ pushes system harder, reducing response time.

Problems: High K_p leads to instability. Results in non-zero steady-state error (cannot eliminate error completely).

5.3 Derivative (D) Component: The Noise Challenge

D (Derivative): $u_D(t) = K_d de(t)/dt$ provides predictive damping, controlling overshoot and rise time based on rate of change.

Problem: Noise in measurements gets amplified when differentiated, causing huge control signal spikes.

Solution: High-pass filter: $D(s) = K_d s/(1+\tau_f s)$ where τ_f is filter time constant.

6 Two Fundamental Problems Identified

Through our PID tuning experiences, two fundamental problems emerged:

Problem 1: Trade-off Dilemma — Improving one metric worsens another; simultaneous optimization was an issue. Want faster response? Increase P, but get overshoot. Want zero steady-state error? Add I, but get oscillations. Want to reduce overshoot? Add D, but amplify noise. Every improvement came at a cost.

Problem 2: Reactive Nature — PID acts only after an error appears. PID acted late, causing deviations. The system must first deviate from the setpoint before PID can respond, resulting in unavoidable tracking errors during transients and disturbances.

7 Advanced Solutions: Beyond Basic PID

7.1 Feed-Forward Control

For 2nd order or known reference changes, **feed-forward** responds proactively: $U(s) = C(s)E(s) + F(s)R(s)$ where ideal $F(s) = 1/G(s)$. Compensates before errors appear.

7.2 Lead-Lag Compensators

For 1st order, **lead-lag compensators** separate PID objectives:

Lead: $G_{lead}(s) = K_c(s+z)/(s+p)$ where $p > z$

- Improves speed and stability
- Adds phase lead: $\phi_{max} = \sin^{-1}[(p-z)/(p+z)]$

Lag: $G_{lag}(s) = K_c(s+z)/(s+p)$ where $z > p$

- Improves steady-state accuracy
- Increases low-frequency gain

Pole-zero placement for desired response based on specifications:

- $\sigma = 4/t_s$ for settling time
- $\zeta = -\ln(\%OS/100)/\sqrt{\pi^2 + \ln^2(\%OS/100)}$ for overshoot

8 MIMO Systems

MIMO (Multi-Input Multi-Output) systems: Matrix form $\mathbf{Y}(s) = \mathbf{G}(s)\mathbf{U}(s)$ where:

$$\mathbf{G}(s) = \begin{bmatrix} G_{11}(s) & G_{12}(s) \\ G_{21}(s) & G_{22}(s) \end{bmatrix}$$

Requires multivariable control for coupled channels.

9 Tools and Techniques

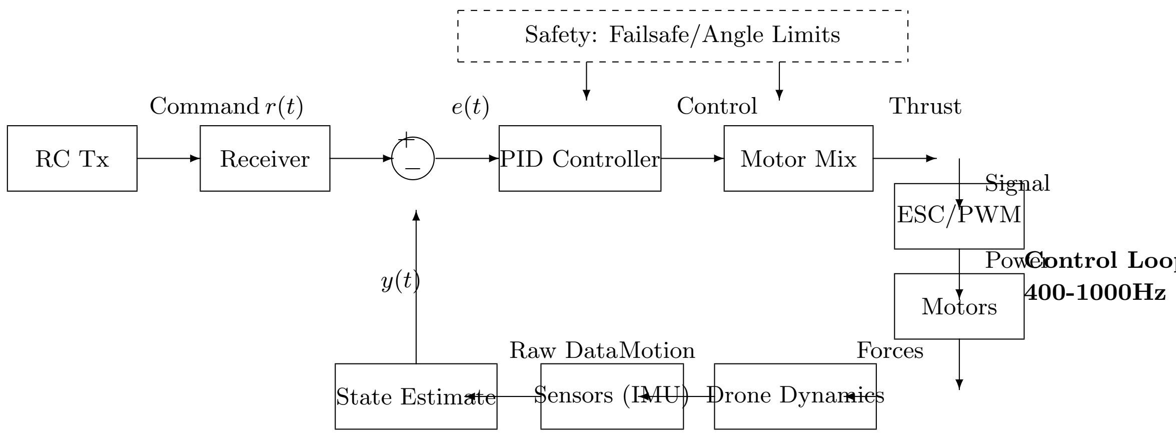
MATLAB/Simulink: Used for modeling and simulation with functions: `tf` (transfer function creation), `step` (time response), `stepinfo` (performance metrics), `feedback` (closed-loop formation), `pidTuner` (controller tuning). Tuning using **Bode plots** for different types of inputs and controllers enabled frequency-domain analysis and design.

Hardware Implementation: Studied electrical circuit components for PID individually. Introduction to **digital control implementation** covered ADC conversion (reading analog sensors in binary by microcontrollers) and DAC conversion (outputting control signals). Explored working of Arduino with PID, using encoders for wheel distance sensing in practical applications.

State Space Controls: Introduction to state-space representation $\dot{\mathbf{x}} = \mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{u}$, $\mathbf{y} = \mathbf{C}\mathbf{x} + \mathbf{D}\mathbf{u}$ provided an alternative framework for modern control techniques, handling MIMO systems and providing complete internal state information.

10 Real World Application - DRONE

10.1 System Block Diagram



Forward Path: RC → Receiver → Error → PID → Motor Mix → ESC → Motors → Drone

Feedback Path: Drone → Sensors → State Estimation → Error Calculation

Pilot Command (Setpoint): You move the joystick on your remote (RC Transmitter). This sends a desired angle or velocity (e.g., "Tilt forward 10°").

Receiver (Rx): The drone's radio receiver catches this signal and sends it to the Flight Controller (the brain).

Sensors (Feedback): Simultaneously, onboard sensors measure the actual status of the drone:

- **IMU (Gyro/Accel):** Measures current tilt and rotation speed
- **GPS/Barometer:** Measures current height and location

10.2 2. Processing Stage (The "Brain")

State Estimation: The microcontroller filters raw sensor data to figure out the drone's true orientation (removing noise/vibrations).

Error Calculation: The code compares *What you want* (Remote input) vs. *What is happening* (Sensor input).

- Example: You want 10° tilt, Drone is at 0°. Error = 10°

PID Controller: The Control System algorithm processes this error:

- **P:** Reacts to the current error
- **I:** Fixes long-term drift (like wind pushing it)
- **D:** Dampens the movement so it doesn't overshoot/wobble

Motor Mixing: The controller converts the PID result into specific motor speeds (e.g., "Spin front motors slower, rear motors faster").

10.3 3. Actuation Stage (The Muscle)

PWM Signal: The Flight Controller sends a digital pulse (PWM) to the ESCs (Electronic Speed Controllers).

Power Switching: The ESC takes battery power and switches it rapidly to drive the motors.

Thrust Generation: The motors spin the propellers, creating lift and torque to physically move the drone.

10.4 4. The Loop (Feedback Mechanism)

Physics Reaction: The drone physically tilts or lifts.

Re-Sensing: The IMU detects this new movement immediately.

Loop: This data goes back to Step 2. This cycle happens hundreds of times per second (e.g., 400Hz or 1kHz loop rate) to keep the drone stable.

10.5 5. Safety Layer (Background Check)

Failsafe: If the Remote signal is lost → Trigger "Return to Home" or "Land".

Angle Limit: Prevents the drone from flipping over (e.g., max tilt 45°).

Summary: Remote/Sensors (Input) → Microcontroller calculates Error (Process) → ESC/Motors adjust speed (Actuate) → Sensors read new position (Repeat).