





Week 10: PID Control and Algorithm Analysis

Objectives:

- Understand the basics of PID control .
- Apply PID tuning to practical systems like line-following robots  and spring-damper systems .
- Analyze and improve algorithm performance .

What is PID Control?

PID stands for **Proportional-Integral-Derivative**. It is a control loop mechanism that continuously calculates an error value as the difference between a desired setpoint and a measured process variable.

Components:

- **P (Proportional)**: Reacts to the present error.
- **I (Integral)**: Reacts to accumulated past errors.
- **D (Derivative)**: Predicts future errors to mitigate oscillations.

Key Objective: Adjust system inputs to achieve a stable, desired output.

How Do PID Components Work?

Proportional Control (P)

- Adjusts output proportional to the current error.
- **Larger K_p** = Faster response, but can cause overshoot.

Integral Control (I)

- Addresses accumulated error over time.
- **Larger K_i** = Eliminates steady-state error, but can lead to "windup."

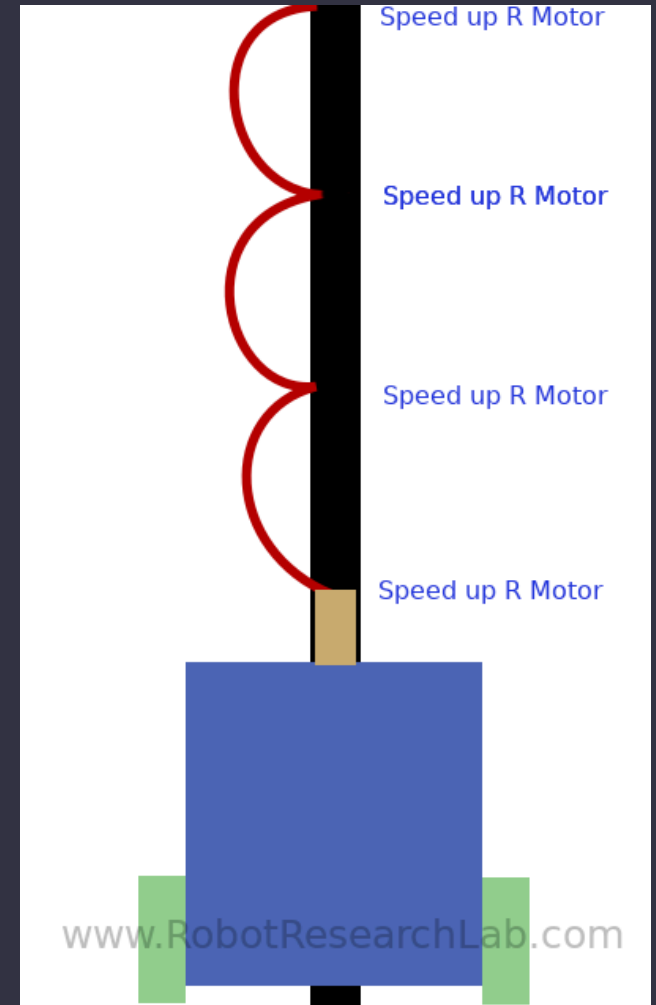
Derivative Control (D)

- Reacts to the rate of change of error.
- **Larger K_d** = Reduces overshoot and smooths response, but sensitive to noise.

Simple Line Follow

Single Sensor

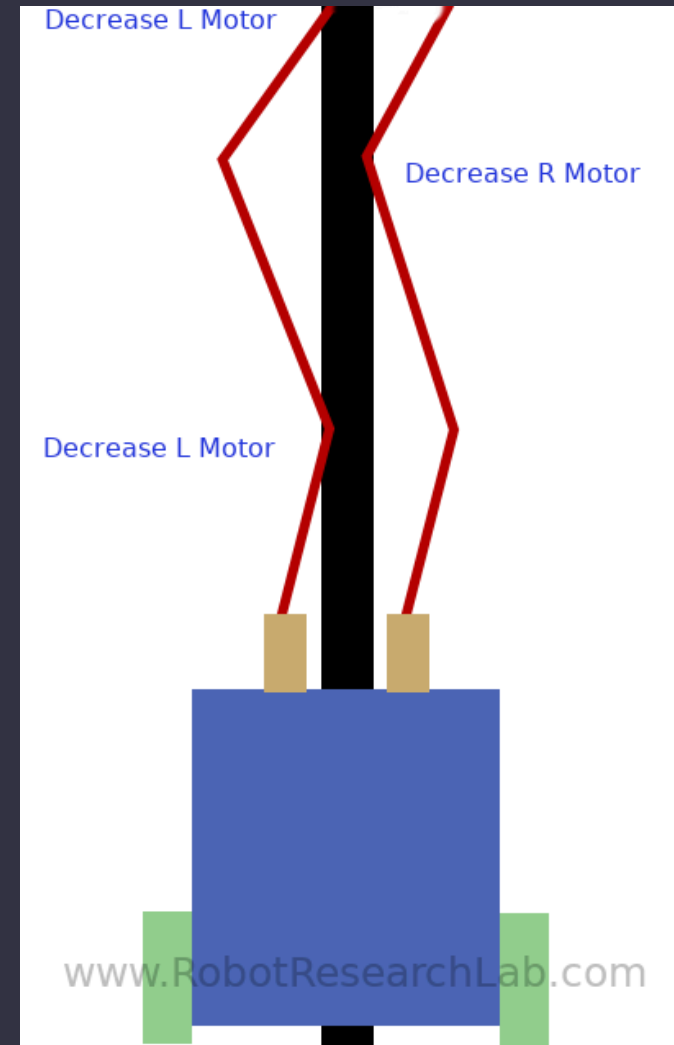
- Favor one side
- When you see the line, apply an opposing force for some time
- Very inefficient
- What happens if you overshoot?



Less Simple Line Follow

Two Sensors

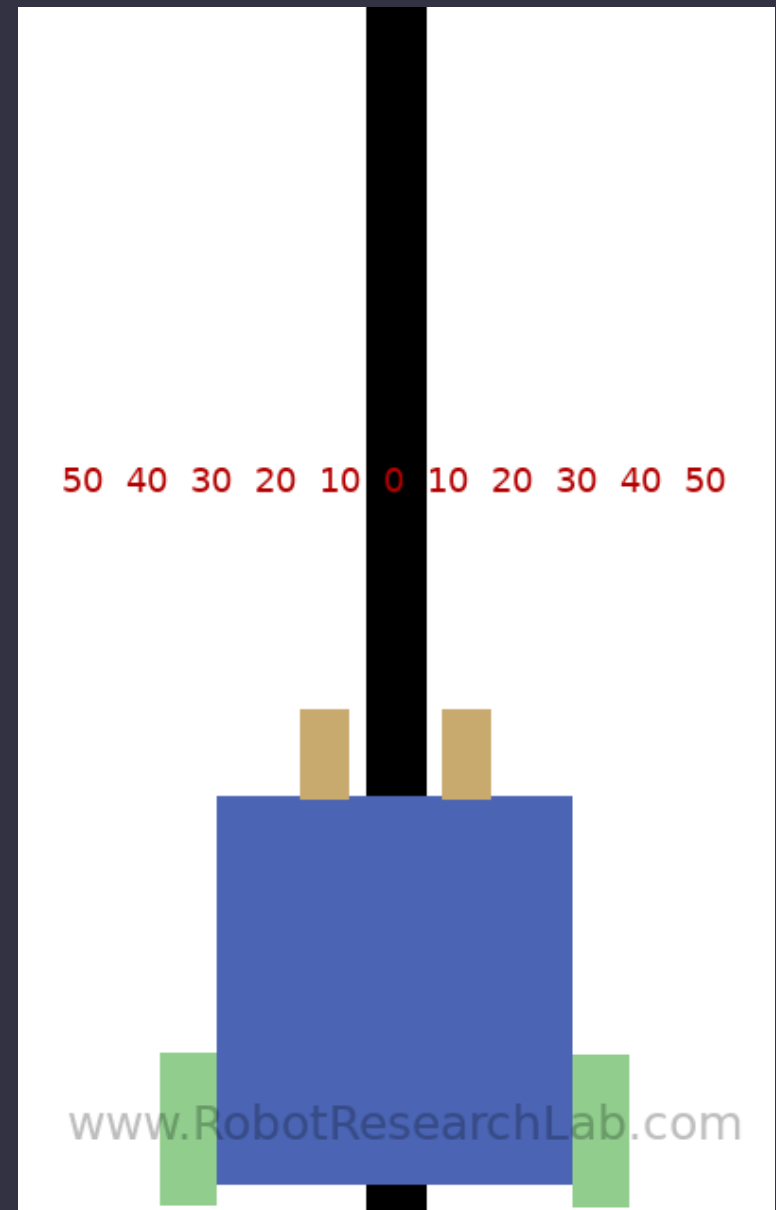
- When you see the line, apply power to the alternate motor
- Still inefficient
- What happens if you overshoot?



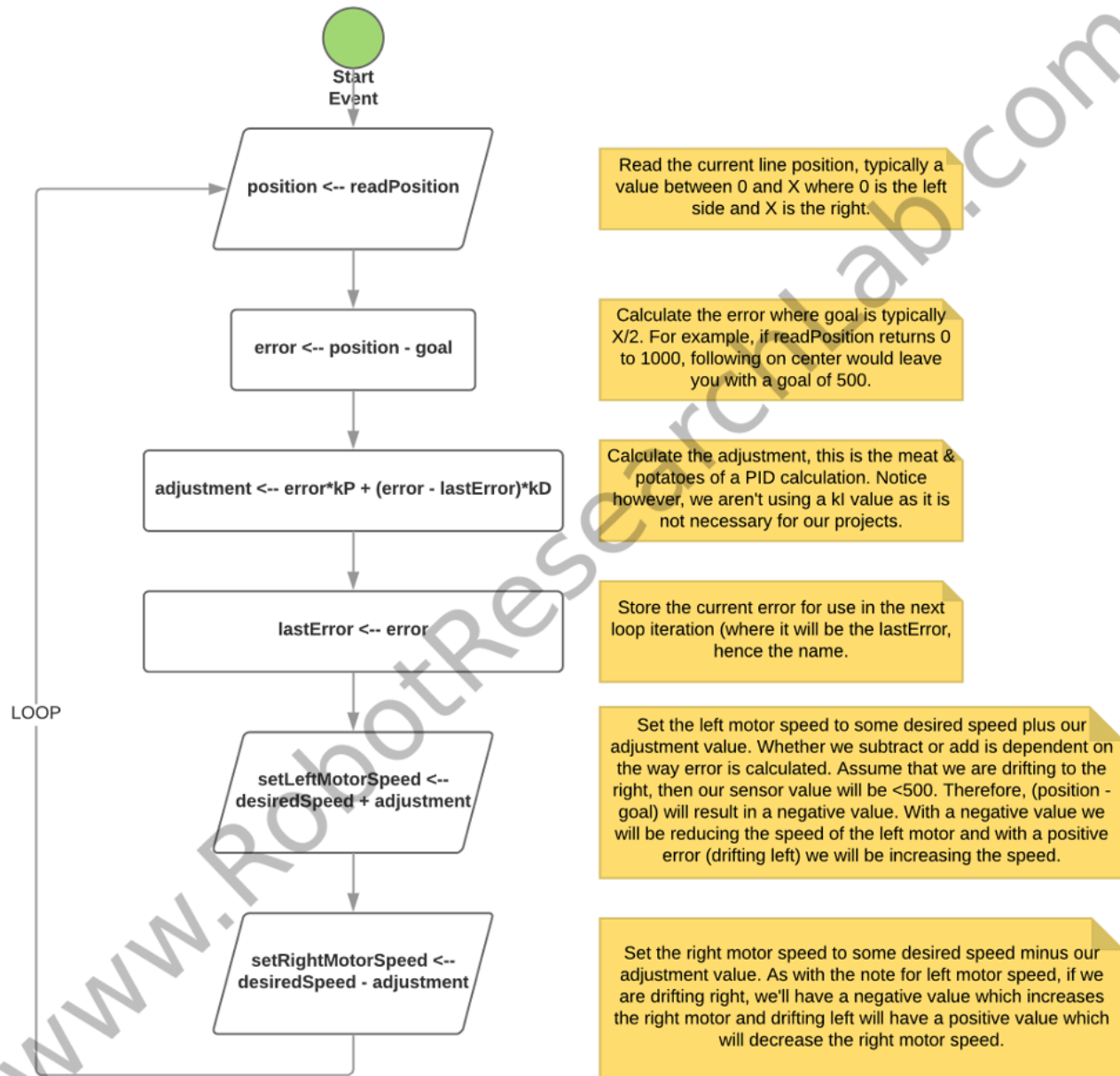
Better Line Follow

Many Sensors

- Gives the ability to apply analog adjustments
- More Efficient
- Better adjustment based on distance
- What happens if you overshoot?
- Without PID though, still inefficient



PID Line Follow



Proportional

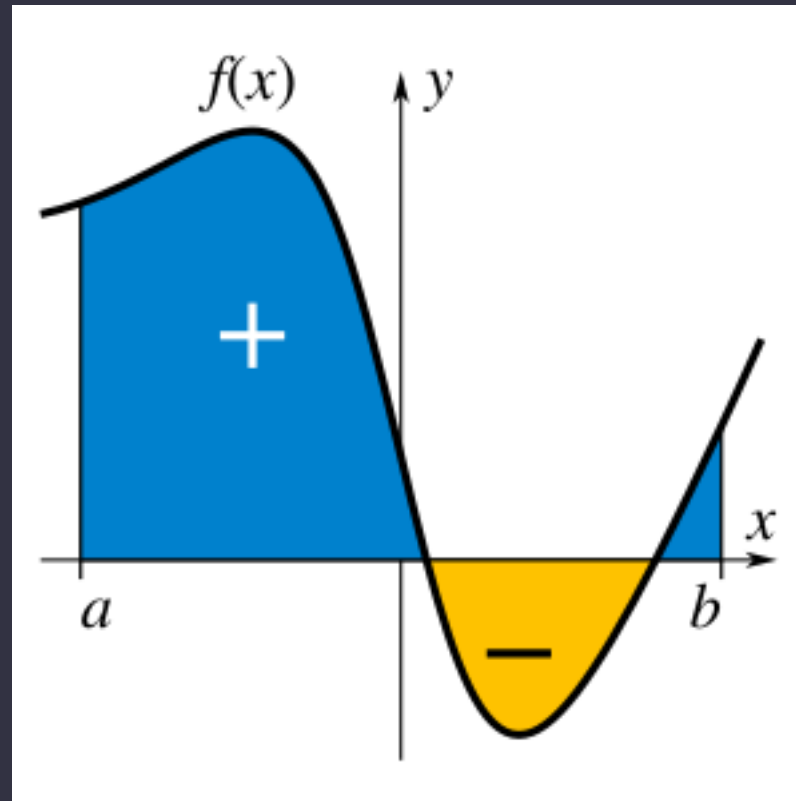
$$u(t) = K_p e(t) + K_i \int_0^t e(\tau) d\tau + K_d \frac{de(t)}{dt}$$

50 40 30 20 10 0 10 20 30 40 50

www.RobotResearchLab.com

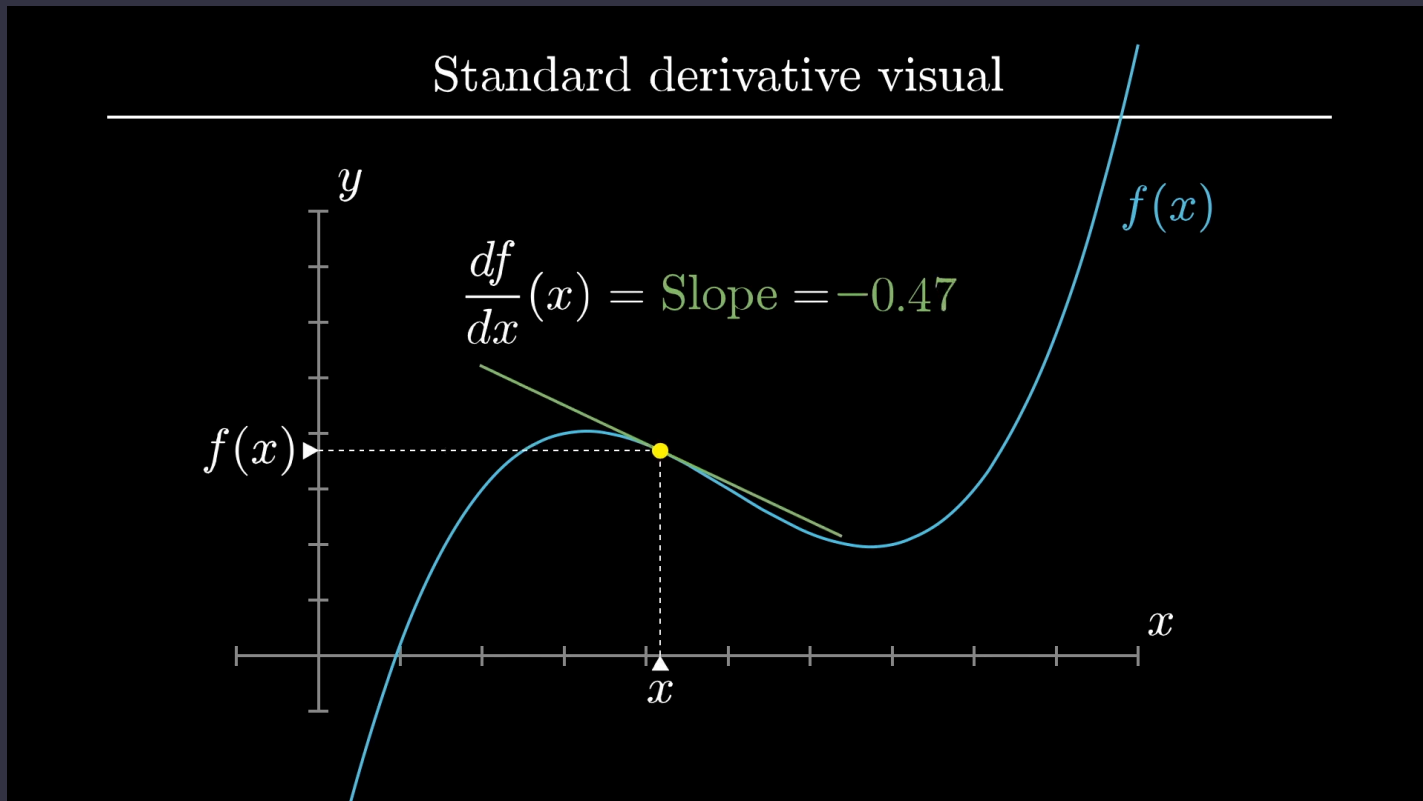
Integral

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

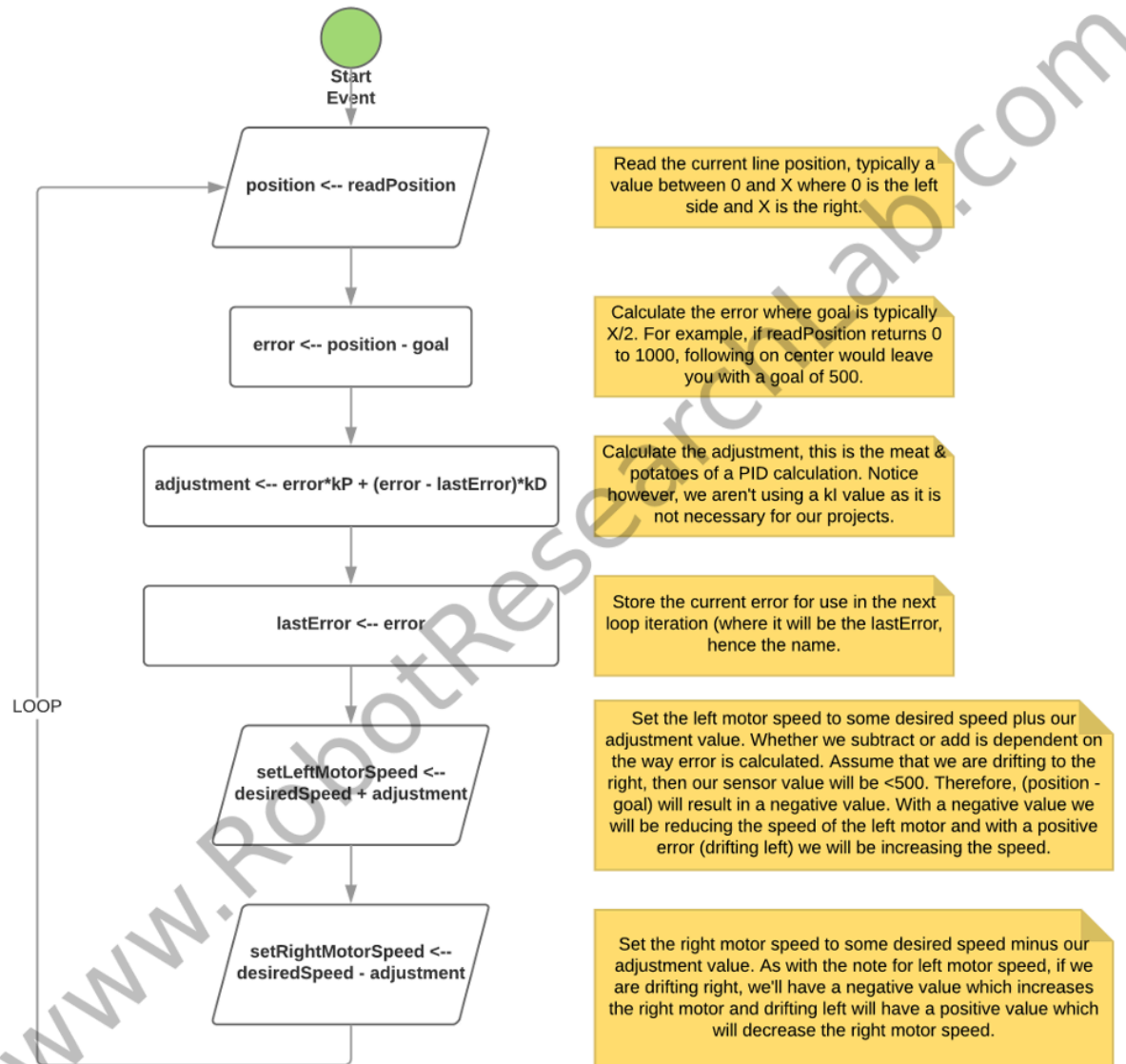


Derivative

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




Simply Put



Practical Tips for PID Tuning

General Guidelines:

- Start by setting **K_i** and **K_d** to 0.
- Gradually increase **K_p** until the system oscillates, then back it off.
- Increase **K_i** to eliminate any residual steady-state error.
- Adjust **K_d** to reduce overshooting and smooth out oscillations.

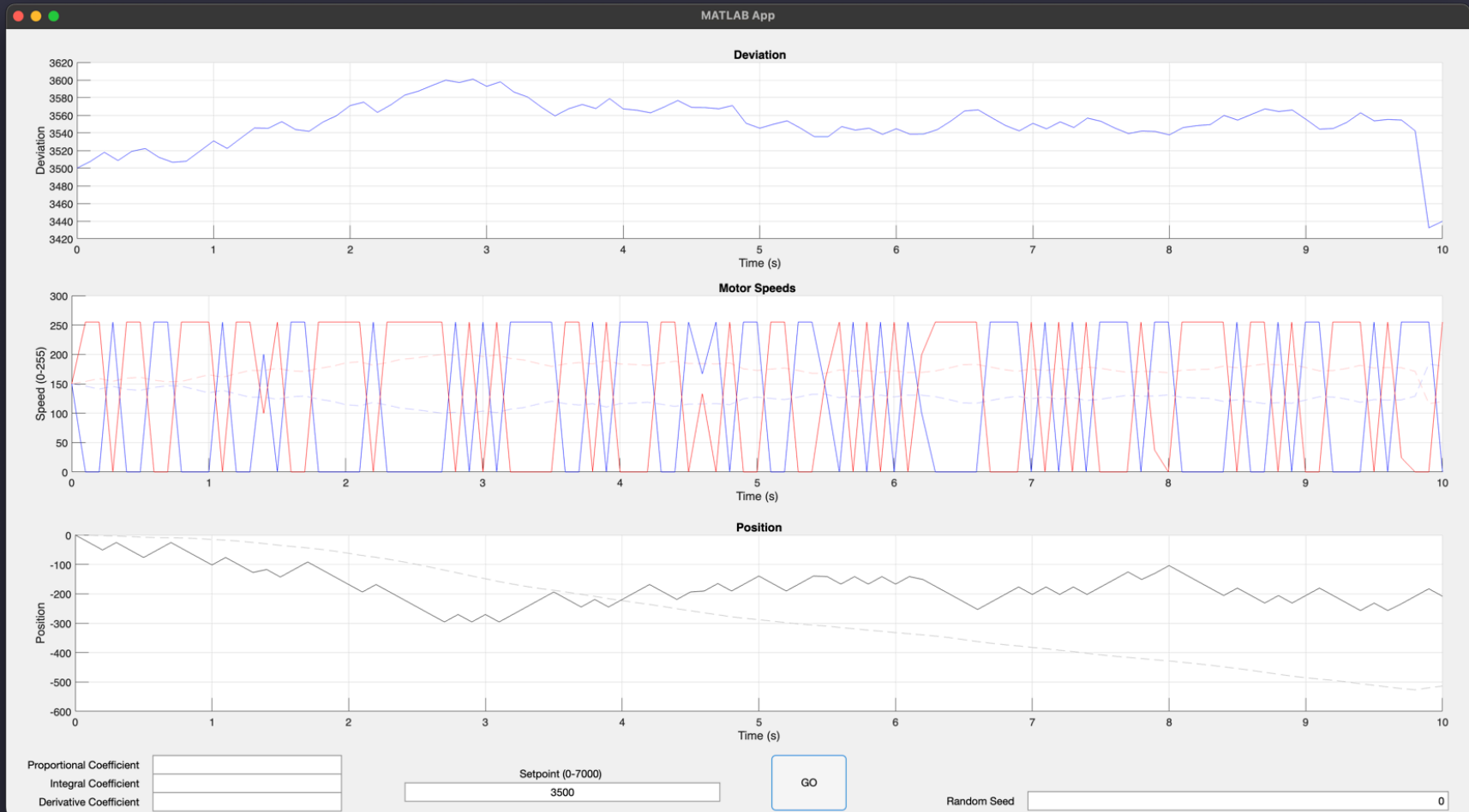
 **Tip:** Fine-tuning each parameter requires careful observation of the system's behavior.

Identifying Good vs. Bad PID Tuning

Characteristics of Good PID Tuning:

- Quick response to reach the setpoint.
- Minimal overshoot.
- Stable, steady behavior without oscillations.

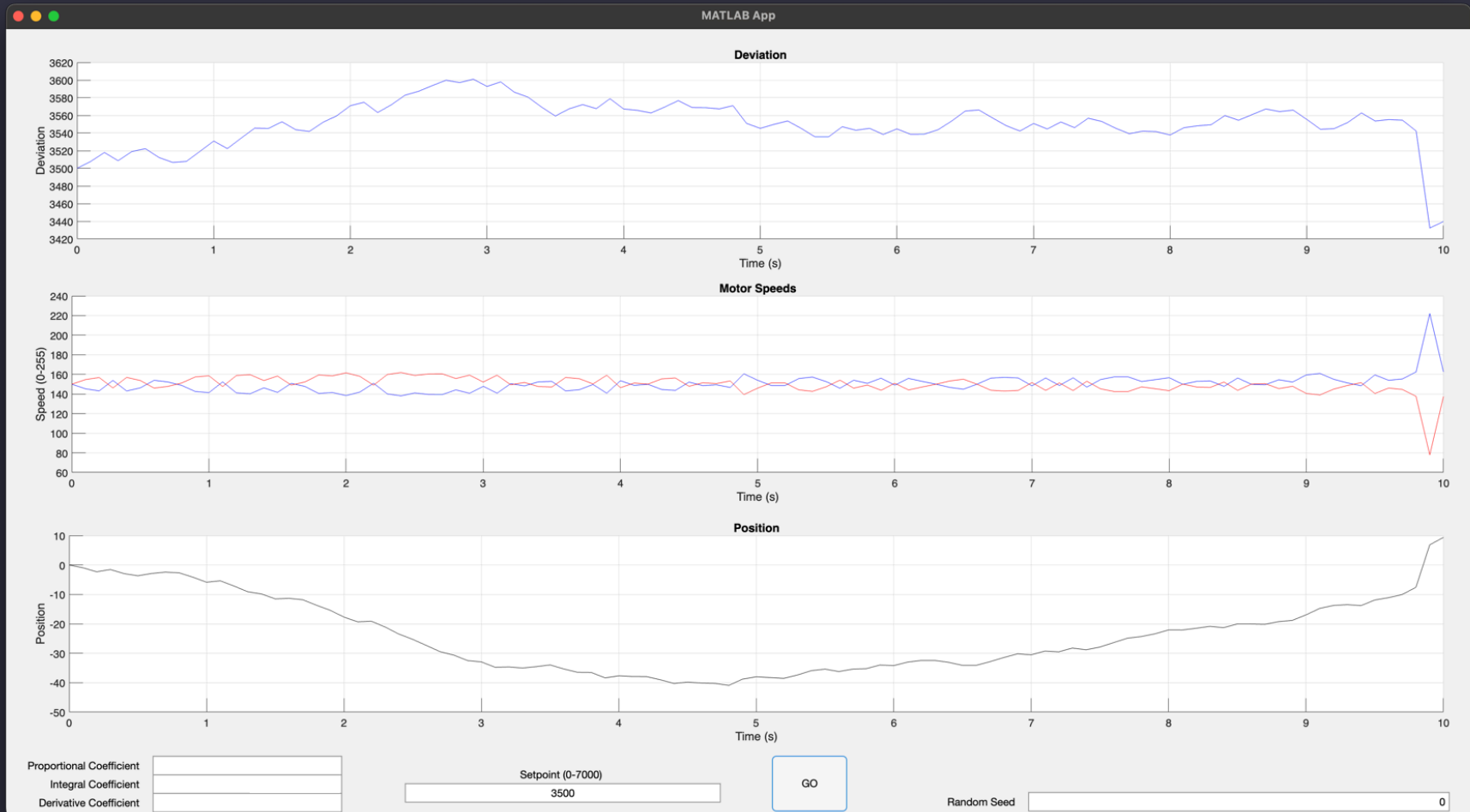
Bad PID Example:



Causes for bad PID

- **Overshoot & Oscillation:** High **K_p**, low **K_d**.
- **Slow Response:** Low **K_p**, low **K_i**.
- **Integral Windup:** High **K_i** causing prolonged error correction.

Good PID Example:



Good PID Characteristics

- Smooth, quick, and stable response.
- Staying relatively close to the setpoint/goal
- Keeping the position at 0

Line-Following Robot Example

Objective:

Use PID to adjust motor speeds to keep a robot centered on a line.

- **Input:** Sensor readings from Pololu line sensor array.
- **Setpoint:** 3500 when the line is centered.

Example Code:

```
[time, sensorReading, leftMotorSpeed, rightMotorSpeed, position] = ...  
simulateLineFollowerPololu(Kp, Ki, Kd, 3500, setpoint, randomSeed);
```

Adjust Kp, Ki, Kd to improve line-following performance.

- Compare plots of sensor readings and motor speeds to fine-tune.

Algorithm Analysis: Performance Evaluation

How to Analyze PID Performance:

- **Sensor Readings:** How accurately does the system follow the setpoint?
- **Motor Speeds:** Check for smooth, consistent adjustments.
- **Position Tracking:** Verify if the system remains stable over time.

Key Metrics:

- Overshoot
- Settling Time
- Steady-State Error
- Response Time

 **Tip:** Visualize these metrics with MATLAB plots.

Practical Example: Adjusting PID for Best Results

Approach:

1. Set initial **K_p** to respond quickly to errors.
2. Tune **K_i** to reduce steady-state error.
3. Adjust **K_d** to dampen oscillations.

Simulation Examples:

Compare your tuning results:

- Last run (lighter colors) vs. current run (regular colors).
- Visualize differences to understand the effect of each parameter.

Key Takeaways

- PID is fundamental for controlling dynamic systems.
- Tuning involves balancing **P**, **I**, and **D** parameters.
- Use MATLAB to simulate and refine control algorithms.
- Consistent testing and visualization are essential for effective tuning.

Additional Tips 💡

- Experiment with different random seeds to test robustness.
- Save simulation results for easy comparison.
- Regularly adjust and observe how each change affects performance.

Gotchas to Watch Out For ⚠

- **High K_p :** Can lead to overshoot and instability.
- **High K_i :** May cause integral windup.
- **Inadequate K_d :** Will not dampen oscillations effectively.
- **Random Deviations:** Ensure the system performs well even with unpredictable inputs.