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 Kp** = Faster response, but can cause overshoot.

Integral Control (I)

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

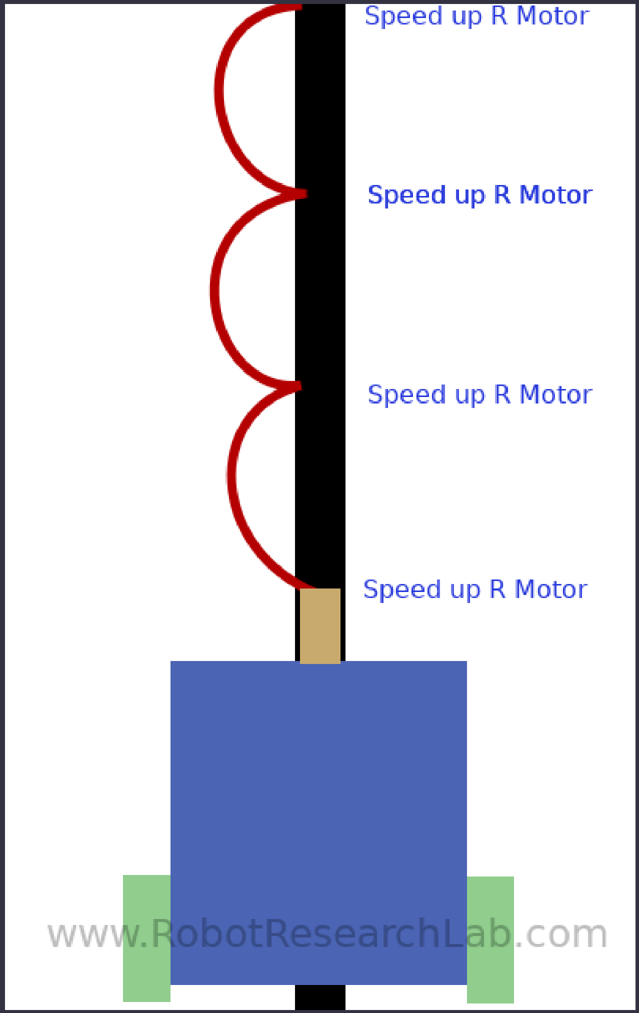
Derivative Control (D)

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

Simple Line Follow

Single Sensor

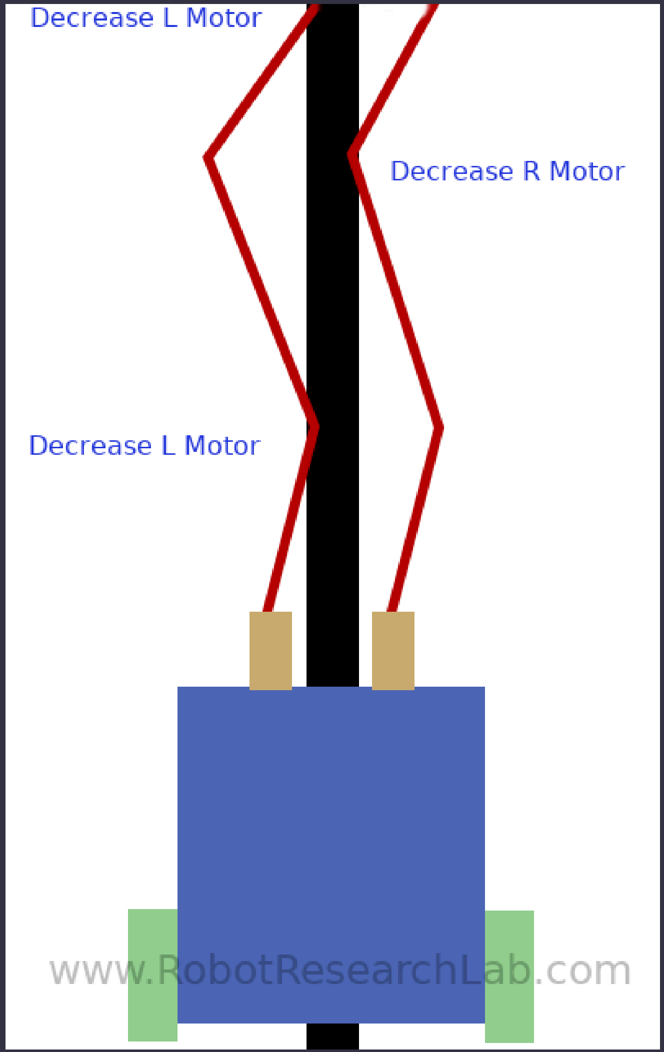
- Favor one side
- When you see the line, apply an opposing force for some time
- Very inefficient
- What happens if you overshoot?



Less Simple Line Follow

Two Sensors

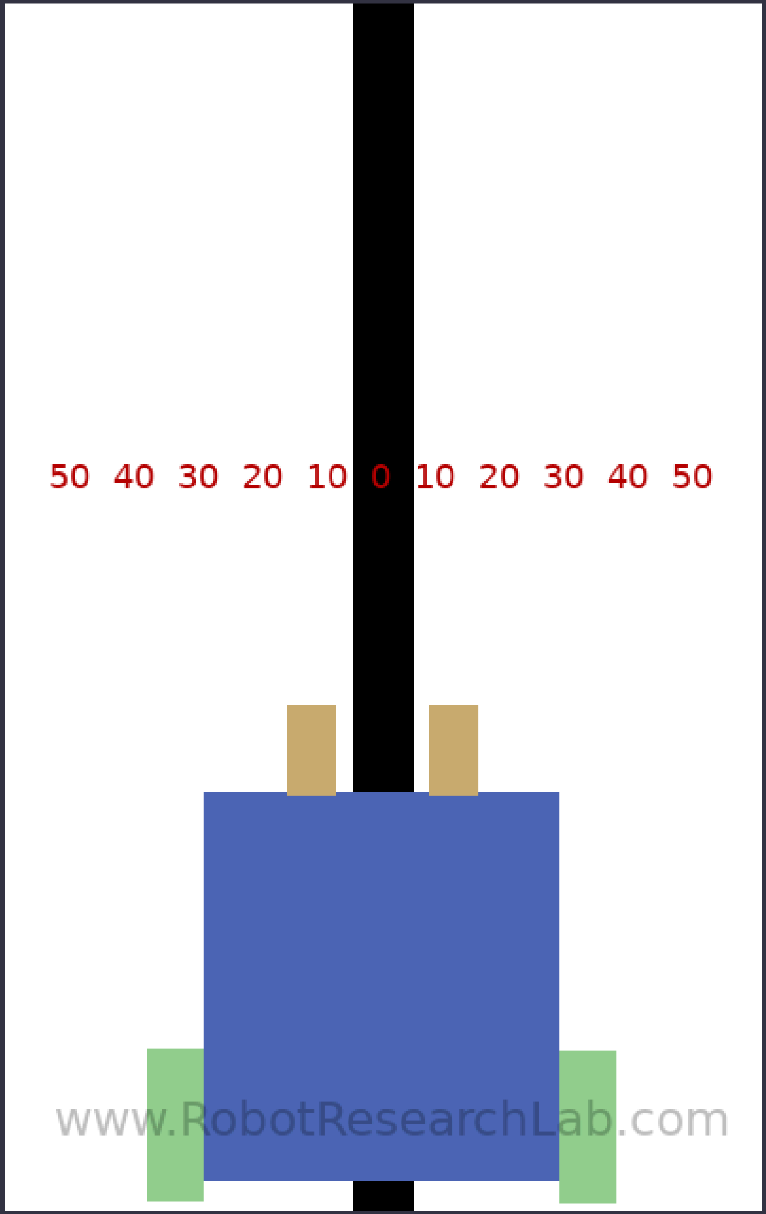
- When you see the line, apply power to the alternate motor
- Still inefficient
- What happens if you overshoot?



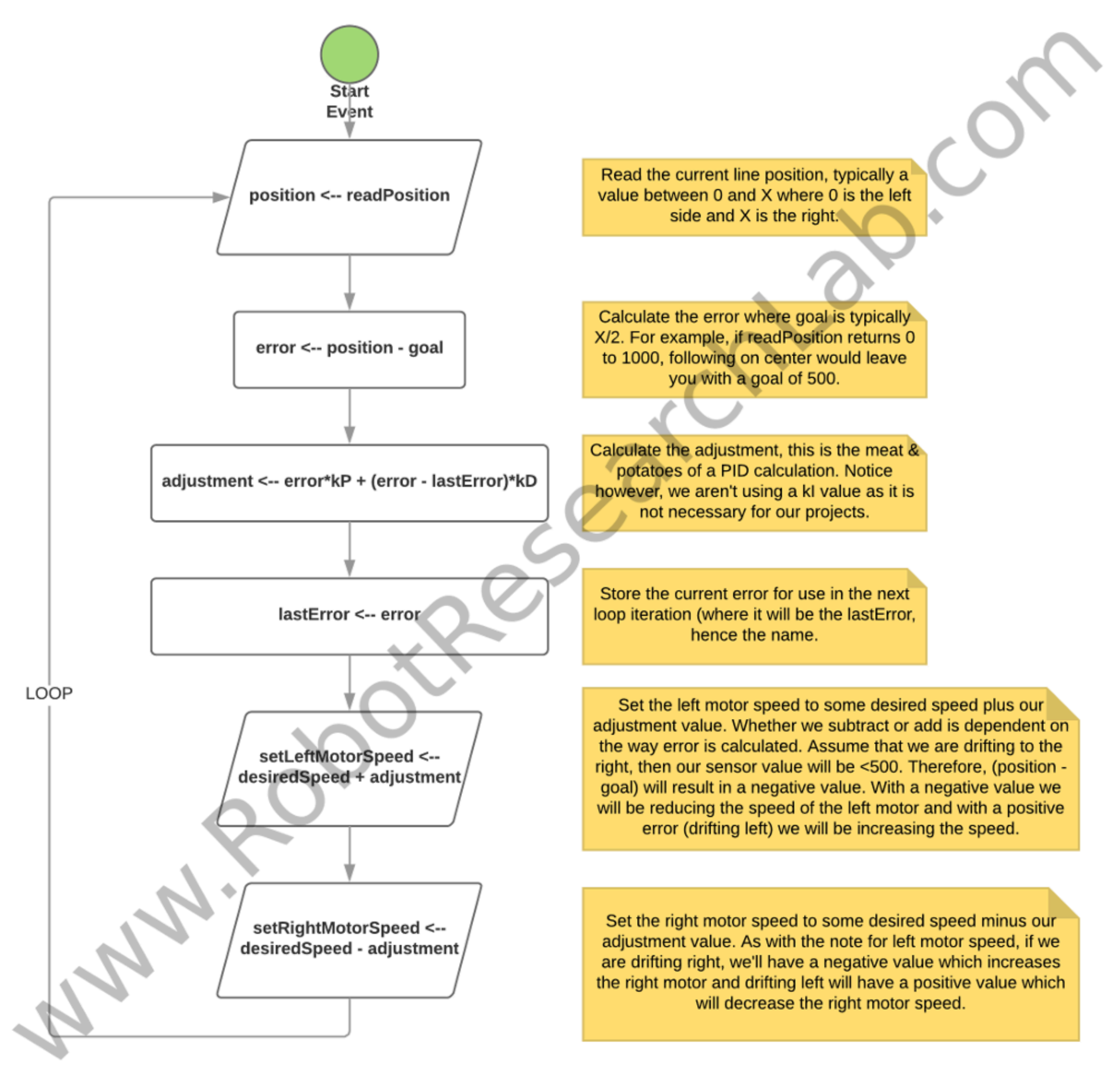
Better Line Follow

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

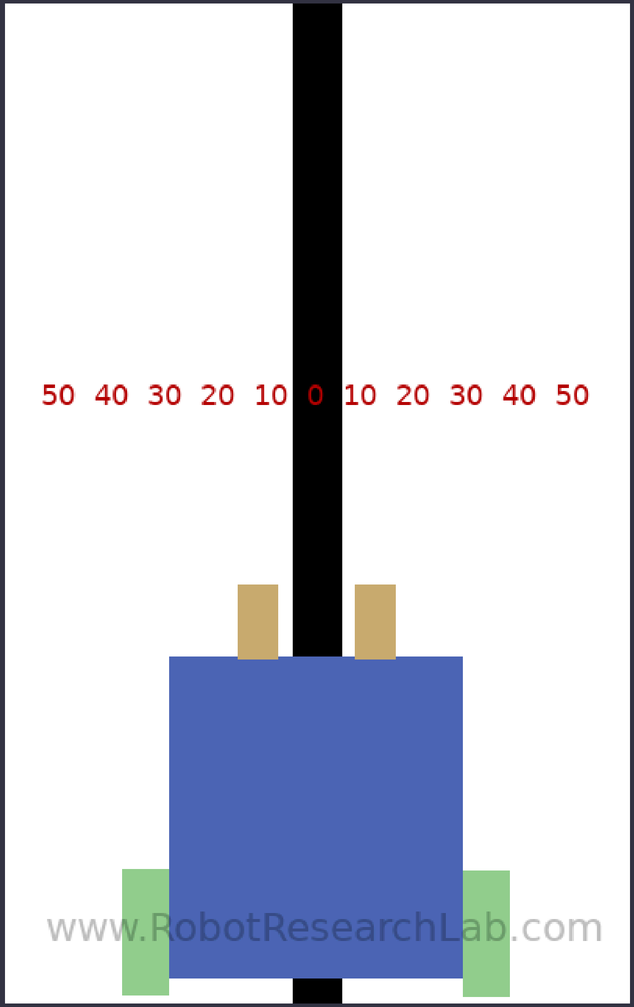


PID Line Follow


Start Event
position <-- readPosition
Read the current line position, typically a value between 0 and X where 0 is the left side and X is the right.
error <-- position - goal
Calculate the error where goal is typically X/2. For example, if readPosition returns 0 to 1000, following on center would leave you with a goal of 500.
adjustment <-- error*kP + (error - lastError)*kD
Calculate the adjustment, this is the meat & potatoes of a PID calculation. Notice however, we aren't using a kI value as it is not necessary for our projects.
lastError <-- error
Store the current error for use in the next loop iteration (where it will be the lastError, hence the name.
LOOP
setLeftMotorSpeed <-- desiredSpeed + adjustment
Set the left motor speed to some desired speed plus our adjustment value. Whether we subtract or add is dependent on the way error is calculated. Assume that we are drifting to the right, then our sensor value will be <500. Therefore, (position - goal) will result in a negative value. With a negative value we will be reducing the speed of the left motor and with a positive error (drifting left) we will be increasing the speed.
setRightMotorSpeed <-- desiredSpeed - adjustment
Set the right motor speed to some desired speed minus our adjustment value. As with the note for left motor speed, if we are drifting right, we'll have a negative value which increases the right motor and drifting left will have a positive value which will decrease the right motor speed.
www.RobotResearchLab.com

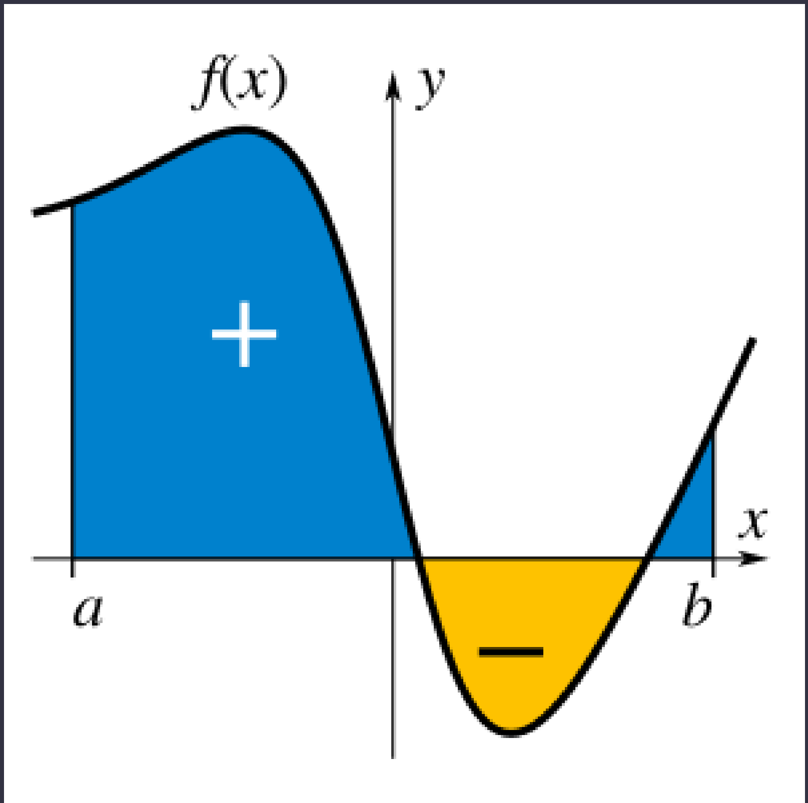
Proportional

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


50 40 30 20 10 0 10 20 30 40 50
www.RobotResearchLab.com

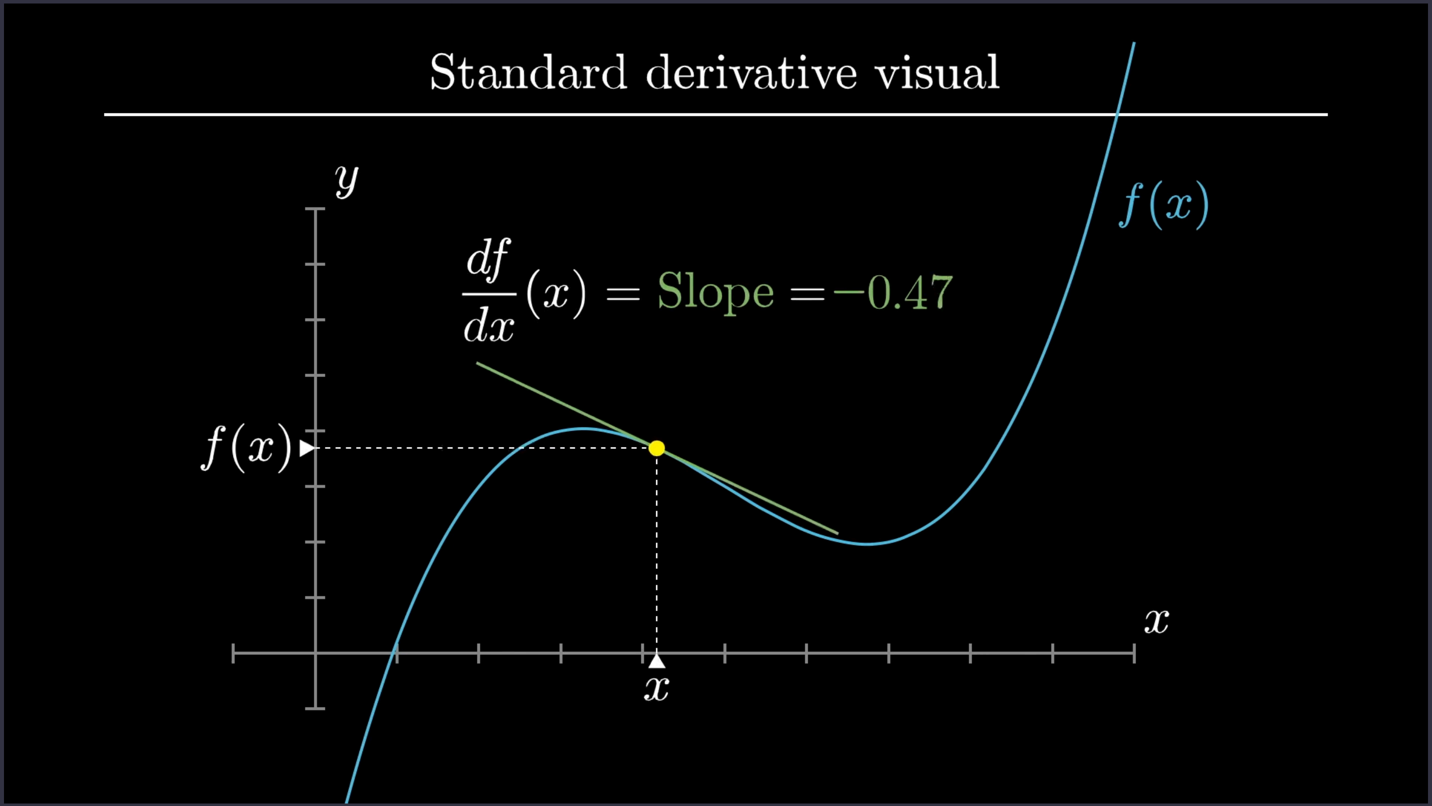
Integral

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

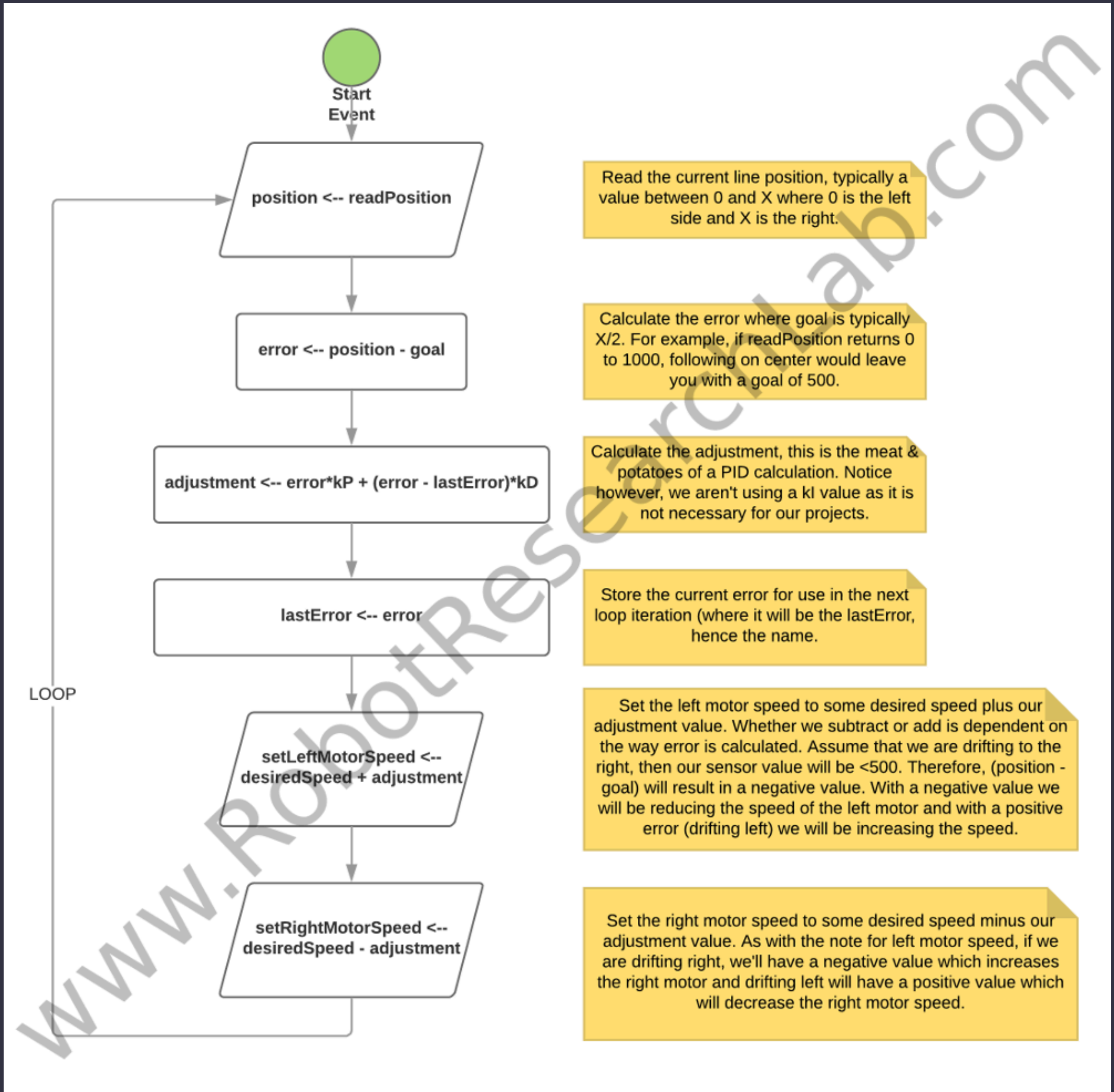


Derivative

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



Simply Put



Start Event
position <-- readPosition
error <-- position - goal
adjustment <-- error*kP + (error - lastError)*kD
lastError <-- error
setLeftMotorSpeed <-- desiredSpeed + adjustment
setRightMotorSpeed <-- desiredSpeed - adjustment
LOOP
Read the current line position, typically a value between 0 and X where 0 is the left side and X is the right.
Calculate the error where goal is typically X/2. For example, if readPosition returns 0 to 1000, following on center would leave you with a goal of 500.
Calculate the adjustment, this is the meat & potatoes of a PID calculation. Notice however, we aren't using a kI value as it is not necessary for our projects.
Store the current error for use in the next loop iteration (where it will be the lastError, hence the name.
Set the left motor speed to some desired speed plus our adjustment value. Whether we subtract or add is dependent on the way error is calculated. Assume that we are drifting to the right, then our sensor value will be <500. Therefore, (position - goal) will result in a negative value. With a negative value we will be reducing the speed of the left motor and with a positive error (drifting left) we will be increasing the speed.
Set the right motor speed to some desired speed minus our adjustment value. As with the note for left motor speed, if we are drifting right, we'll have a negative value which increases the right motor and drifting left will have a positive value which will decrease the right motor speed.
www.RobotResearchLab.com

Practical Tips for PID Tuning 🛠️

General Guidelines:

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

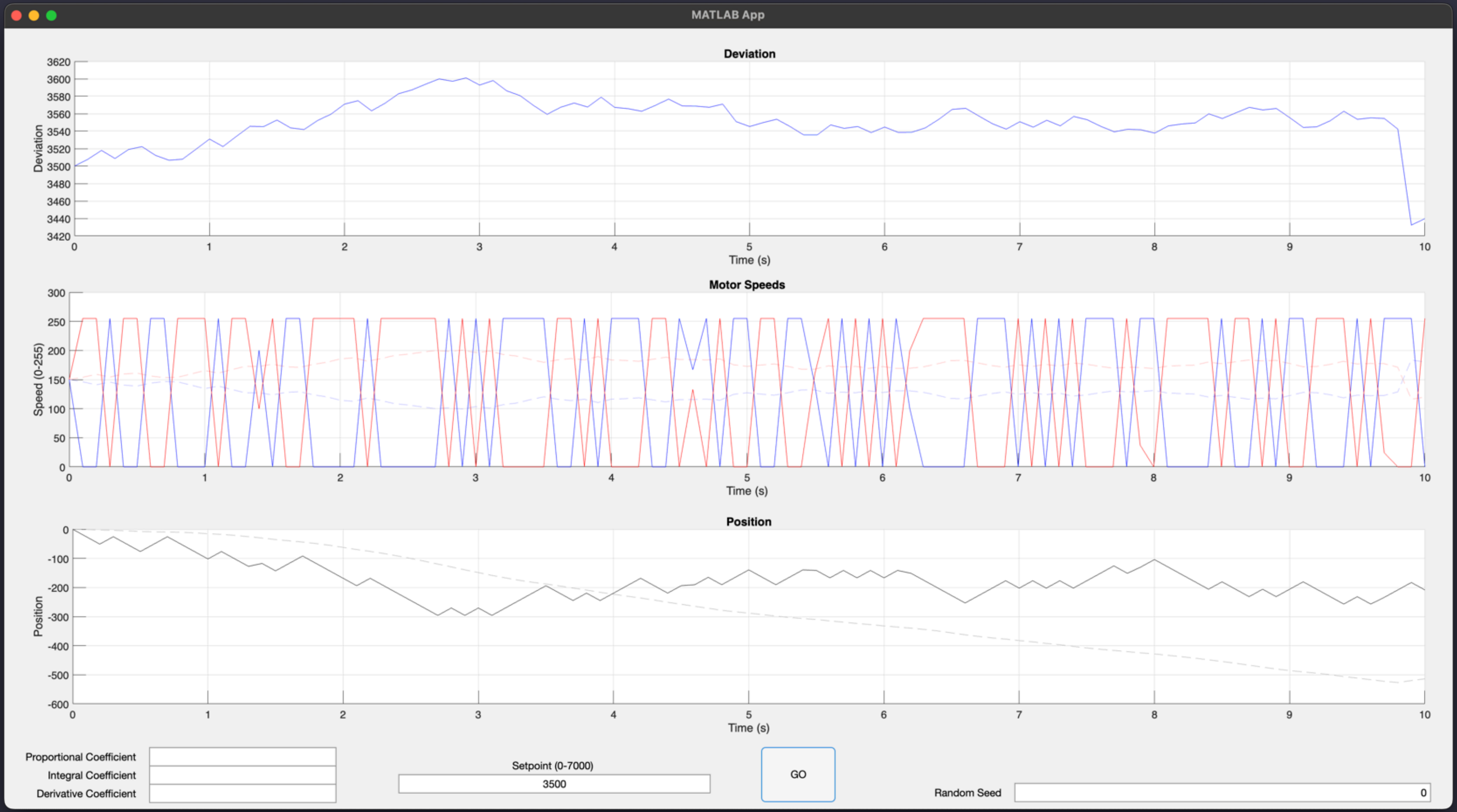
💡 **Tip**: Fine-tuning each parameter requires careful observation of the system's behavior.

Identifying Good vs. Bad PID Tuning 📊

Characteristics of Good PID Tuning:

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

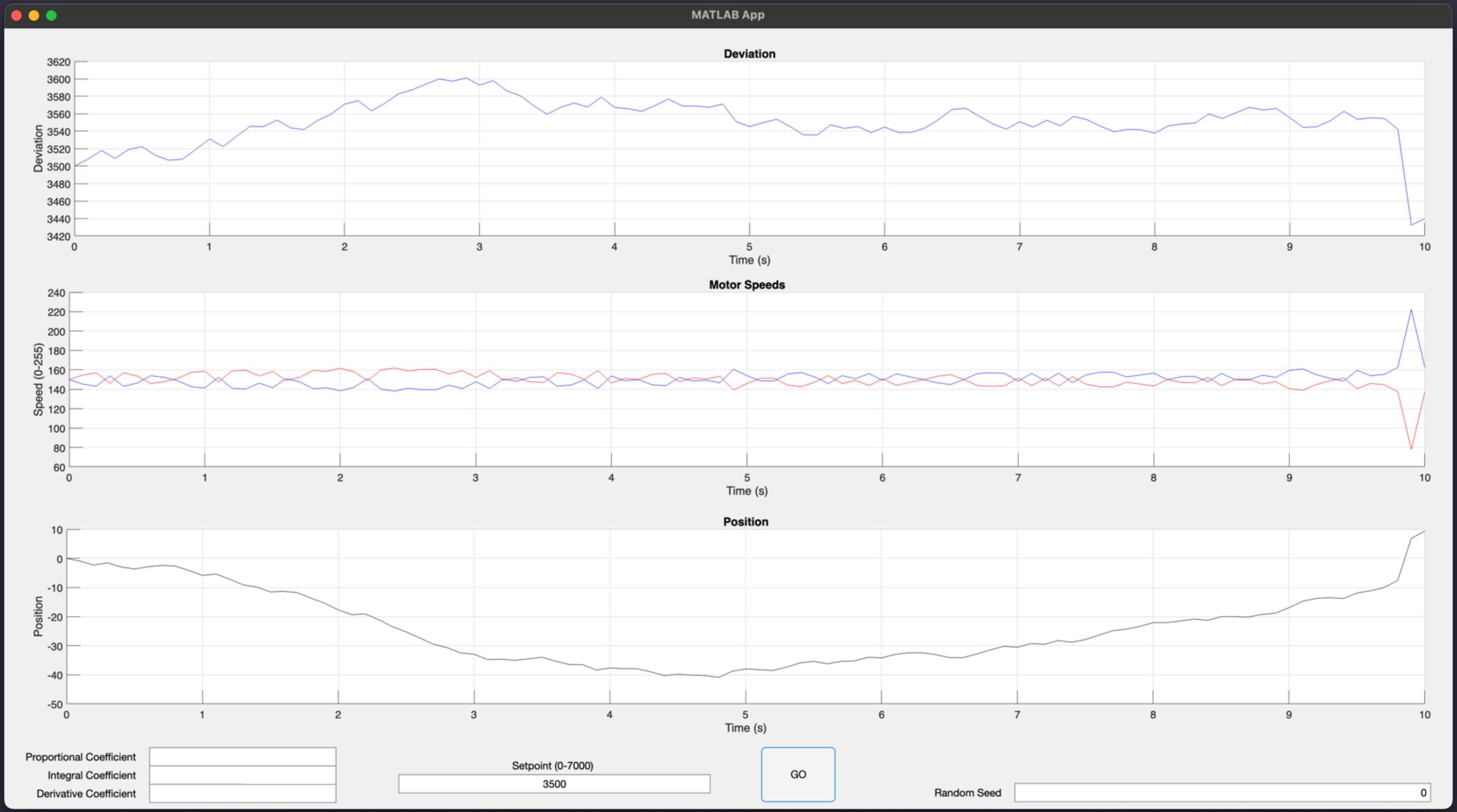
Bad PID Example:



Causes for bad PID

- **Overshoot & Oscillation**: High **Kp**, low **Kd**.
- **Slow Response**: Low **Kp**, low **Ki**.
- **Integral Windup**: High **Ki** 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 **Kp** to respond quickly to errors.
2. Tune **Ki** to reduce steady-state error.
3. Adjust **Kd** 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 Kp**: Can lead to overshoot and instability.
- **High Ki**: May cause integral windup.
- **Inadequate Kd**: Will not dampen oscillations effectively.
- **Random Deviations**: Ensure the system performs well even with unpredictable inputs.