Lab 7: Autonomy

1.104 Team
Department of Civil and Environmental Engineering
Massachusetts Institute of Technology

Introduction

In this lab, you will learn about Proportional-Integral-Derivative (PID) control, one of the most widely used control techniques in industry. PID controllers are found in a variety of applications, ranging from simple temperature control to complex motion systems in robotics. By the end of this lab, you will understand how each component of the PID controller affects system behavior, how to properly tune the parameters for optimal performance, and how the control system is embedded into the real-world robotic system. The lab consists of the following parts:

1. PID control of a double integrator system
2. Real-world robot car PID control
3. PID control of a rocket model
4. PID control of a nonlinear pendulum system

1 Theoretical Background

1.1 PID Control Fundamentals

A PID controller continuously calculates an error value $e(t)$ as the difference between a desired setpoint (SP) and a measured process variable (PV), and applies a correction based on proportional, integral, and derivative terms. The mathematical form of a PID controller is:

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau)d\tau + K_d \frac{de(t)}{dt} \tag{1}$$

Where: $u(t)$ is the control signal, $e(t)$ is the error (setpoint - measured value), K_p is the proportional gain, K_i is the integral gain, K_d is the derivative gain

1.2 The Role of Each Component

1.2.1 Proportional Term (K_p)

The proportional term produces an output proportional to the current error value:

$$P_{out} = K_p e(t) \tag{2}$$

Effect:

- Increases system responsiveness

- Reduces rise time (time to reach setpoint)
- Too high: causes overshoot and oscillation
- Too low: makes the system sluggish and unable to reach the setpoint

1.2.2 Integral Term (K_i)

The integral term accumulates error over time and helps eliminate steady-state error:

$$I_{out} = K_i \int_0^t e(\tau) d\tau \tag{3}$$

Effect:

- Eliminates steady-state error
- Helps overcome system biases
- Too high: causes overshoot and oscillation
- Too low: steady-state error may persist

1.2.3 Derivative Term (K_d)

The derivative term predicts system behavior by calculating the rate of change of error:

$$D_{out} = K_d \frac{de(t)}{dt} \tag{4}$$

Effect:

- Reduces overshoot and settling time
- Improves stability
- Too high: amplifies noise and causes instability
- Too low: system may exhibit excessive overshoot

1.3 Double Integrator System

A double integrator system represents a basic model for many physical systems, like a mass moving in a frictionless environment. Its mathematical representation is:

$$\frac{d^2x}{dt^2} = u \tag{5}$$

Where x is the position, and u is the control input (force). In state-space form:

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} u \tag{6}$$

Where x_1 is the position and x_2 is the velocity.

1.4 Rocket Model (Double Integrator with Gravity)

The rocket model extends the double integrator by adding the effect of gravity, which creates a constant external force. This model represents systems like a vertical rocket or an elevator, where gravity consistently affects the dynamics. Its mathematical representation is:

$$\frac{d^2x}{dt^2} = u - g \tag{7}$$

Where x is the position, u is the control input (force), and g is the gravitational acceleration. In state-space form:

$$\begin{bmatrix}\dot{x}_1\\ \dot{x}_2\end{bmatrix} = \begin{bmatrix}0 & 1\\ 0 & 0\end{bmatrix}\begin{bmatrix}x_1\\ x_2\end{bmatrix} + \begin{bmatrix}0\\ 1\end{bmatrix}u + \begin{bmatrix}0\\ -g\end{bmatrix} \tag{8}$$

The key difference from the simple double integrator is that the constant gravitational force creates a bias that must be compensated for by the controller. This makes the integral term (K_i) particularly important, as it accumulates error over time and provides the necessary constant control input to counteract gravity.

1.5 Pendulum System

The pendulum is described by the nonlinear differential equation:

$$ml^2\ddot{\theta} = mgl\sin\theta - b\dot{\theta} + u \tag{9}$$

Where: θ is the angle from the upright position (0 is upright, π is downward), m is the mass at the end of the pendulum, l is the length of the pendulum, g is the gravitational acceleration, b is the damping coefficient, u is the control torque

This system is nonlinear due to the $\sin\theta$ term, making control more challenging. Note that in this formulation, the gravity term has a positive sign, as it acts to increase the angle away from the upright position.

1.6 Linear Quadratic Regulator (LQR)

The Linear Quadratic Regulator (LQR) is an optimal control technique that provides a systematic way to design feedback controllers for linear systems. Unlike PID control, which requires manual tuning, LQR determines control gains mathematically to minimize a specified performance index.

Each experiment includes a ground truth solution using LQR. This provides a benchmark for comparison with your PID controller. LQR is an advanced control method that provides optimal performance. Use it as a reference when tuning your PID controller.

2 Performance Metrics for Control Systems

Before beginning the experiments, it's important to understand the key metrics used to evaluate control system performance. Throughout all experiments in this lab, you'll need to record and analyze these metrics:

- **Rise Time**: The time it takes for the system output to rise from 10% to 90% of the steady-state value. A shorter rise time indicates a more responsive system but may lead to overshoot.
- **Overshoot**: The amount by which the system output exceeds the setpoint, expressed as a percentage of the setpoint value. For example, if the setpoint is 1.0 and the maximum value reached is 1.2, the overshoot is 20%.
- **Settling Time**: The time required for the system response to reach and stay within a specified range (typically ±2% or ±5%) of the final value. This indicates how quickly the system stabilizes.
- **Steady-State Error**: The difference between the desired setpoint and the final value of the system output after transients have died out. Ideally, this should be zero or very close to zero.

Each script **automatically computes and displays** these four metrics directly on the response plot, along with visual markers (rise-time point, peak overshoot, settling-time line, and a ±2% settling band). A summary text box in the corner of the plot shows the numerical values, so you can read and record them without manual calculation.

You should record these metrics under different controller configurations and analyze how changes to the PID parameters affect them. These observations will form the core of your lab report. Note that it's not necessary to record these metrics for **each run**, but you should record the metrics for several runs to get a sense of the performance.

3 Laboratory Procedure

3.1 Setup

Before beginning the lab, ensure you have the necessary software installed. You need **Python 3.8 or newer** (check with `python --version`).

1. On the lab laptop, open `VS Code`, and press `Ctrl+‘` to open the terminal. Select "Open Folder" in VS Code, and open the folder `MIT-1104-Autonomy-Lab` on the desktop (select "I trust the authors" if prompted).
2. Install the required Python packages from the provided `requirements.txt`:

   ```
   pip install -r requirements.txt
   ```

   This installs `numpy`, `scipy`, `matplotlib`, `opencv-python`, `pyzbar`, and `pyserial`.
3. Verify the installation by running one of the scripts:

   ```
   python simulation/double_integrator.py
   ```

   Two windows should appear: a line plot and an animation. Close them to return to the terminal.

4. Read the `README.md` file for additional setup details and troubleshooting tips.

Note on running scripts. Throughout this lab you will run simulation and real-world scripts. The commands are:

```
python simulation/double_integrator.py
python simulation/rocket_model.py
python simulation/pendulum.py
python real_world/run_experiment.py
```

3.2 General Tuning Protocol

For each of the three simulation systems below, you are required to test and record the performance of **5 different parameter sets**. Open the corresponding Python script in VS Code and change the `Kp`, `Ki`, and `Kd` variables at the top of the file (inside the clearly marked block), save the file (`Ctrl+S`), then run the script in the terminal. The performance metrics will be displayed automatically on the plot—record them in your lab report table.

The 5 sets must follow this progression:

1. **Set 1 (P-Only):** Set $K_i = 0$ and $K_d = 0$. Tune K_p to achieve a reasonable rise time.
2. **Set 2 (Add I or D):** Keep your K_p from Set 1. Add *either* $K_i > 0$ or $K_d > 0$ to observe how it alters steady-state error or transient overshoot.
3. **Set 3 (Full PID):** Use non-zero values for K_p, K_i, and K_d.
4. **Set 4, 5 (Free Tuning):** Experiment freely to improve your metrics relative to Set 3.

3.3 Part 1: Double Integrator System

Open `simulation/double_integrator.py` in VS Code. In the terminal, run:

```
python simulation/double_integrator.py
```

Two windows will appear: a **response plot** (with annotated performance metrics) and an **animation**. Close both windows to return to the terminal, then change the PID gains and re-run. Follow the general tuning protocol to complete your 5 sets.

Question 1. *Can merely increasing K_p alone simultaneously minimise both rise time and overshoot? Why?*

3.4 Part 1b: Real-World Robot Car Test

In this section you will tune a PID controller on a **real robot car**. Although the PID control law is the same ($u = K_p e + K_i \int e\,dt + K_d \dot{e}$), the car's plant dynamics are fundamentally different from the double-integrator simulation.

Why a Single Integrator?

In the double-integrator simulation, the control input u acts as a *force* (acceleration): $\ddot{x} = u$. On the real car, the PID output u is mapped to a motor **PWM duty cycle**, which roughly commands a *velocity*—the car speed is approximately proportional to the PWM signal. This means the plant is closer to a **single integrator**:

$$\dot{x} = u \tag{10}$$

A single integrator is theoretically easier to stabilise than a double integrator. However, as you will discover, real-world effects make the control problem surprisingly challenging. **Do not** reuse the gains you found for the double-integrator simulation; they will not work here because the plant model is different. You will tune the PID gains from scratch.

Physical Setup

The robot car is built around an **ESP32** microcontroller and carries two DC motors driven by an **L298N** H-bridge board. An **HC-SR04** ultrasonic sensor is mounted on the front of the car, facing the wall, to provide a second distance measurement.

A webcam is mounted above the test area, pointing downward at the floor. Two QR-code markers are used:

- A marker labelled `CAR` is attached to the top of the robot car.
- A marker labelled `WALL` is attached to the wall (the target).

The laptop runs a Python script (`real_world/run_experiment.py`) that sends PID parameters to the car, starts the control loop, continuously detects both QR codes, computes the real-world distance between them, and streams the measurement to the robot car over Bluetooth.

The car fuses two distance sources via a simple **sensor fusion** strategy:

- If both the webcam and ultrasonic readings are available and agree (within 100 mm), the car uses their **average**.
- If the two readings disagree significantly, the car trusts the one closer to the previous measurement.
- If only one sensor is available, the car uses that reading alone.
- If neither sensor is usable, the car holds the last known distance.

This approach makes the system robust to momentary dropouts or outliers from either sensor.

Question 1b-1. *What is the advantage of sensor fusion over using only one sensor?*

Real-world examples

This experiment can be viewed as a simple automated parking case where the car needs to keep an appropriate distance from the wall.

Procedure

1. The TA has already uploaded the firmware to the ESP32 on your robot car. You do **not** need to re-flash the board—all PID parameters are sent from the laptop.

2. Ensure your laptop is paired with the robot car's Bluetooth (the TA should have done this; the device name and serial port are on the car's label).

3. Open `real_world/run_experiment.py` in VS Code. At the top of the file you will see the PID gain variables. **Start with P-only control** (e.g. $K_p = 1.0$, $K_i = 0$, $K_d = 0$) and tune incrementally, just as you did in the simulation:

   ```
   Kp = 1.0     # start small, increase gradually
   Ki = 0.0
   Kd = 0.0
   ```

   Also check the `SETPOINT_MM` matches the 100 mm target distance and the `SERIAL_PORT` matches the Bluetooth serial port written on the car's label.

4. Place the robot car at a starting position **15-20 cm** away from the wall, under the camera's field of view. Power on the car—it will stay still until it receives a START command.

   Question 1b-2. *Why shouldn't you put the car too far away from the wall?*

5. Save the file (`Ctrl+S`) and run it in the terminal:

   ```
   python real_world/run_experiment.py
   ```

   The script will connect to the car via Bluetooth, send your PID gains, issue a START command, and begin streaming webcam distance measurements. A live camera window will appear showing the detected QR codes, measured distance, and PID gains. **Press the `q` key on the keyboard to stop the car when it shows signs of being out of control** (for instance, turning in place and becoming parallel to the wall). **Always put one finger on the emergency stop!**

6. Observe the car as it drives toward the setpoint distance from the wall.

7. When the car has settled (or after a reasonable time), press `q` in the video window to stop. The script sends a STOP command to the car (motors stop) and displays a **performance plot** with the same annotated metrics as the simulation scripts. Record these values.

8. **Edit** the PID values at the top of `run_experiment.py`, save, and re-run. No re-flashing is needed. Follow the same tuning progression as the simulation (P-only → add I or D → full PID → free tuning) and record at least **3 parameter sets** with their metrics.

What to Observe

A single integrator ($\dot{x} = u$) is one of the simplest plants to control in theory—a proportional controller alone guarantees exponential convergence with no overshoot. In practice, however, you will encounter effects that make tuning surprisingly difficult:

- **Measurement noise:** the distance fluctuates even when the car is stationary, injecting high-frequency error into the derivative term.
- **Transport delay:** Bluetooth communication and sensor processing introduce latency between measurement and actuation.
- **Motor dead-zone and asymmetry:** DC motors require a minimum PWM to start moving, and the two motors may not respond identically, causing the car to veer and the ultrasonic sensor to return spurious readings.
- **Nonlinear friction:** static and dynamic friction mean the mapping from PWM to velocity is not the clean $\dot{x} = u$ assumed by a single-integrator model.

These effects are absent from the simulation and illustrate why real-world control is harder than theory suggests, even for a "simple" plant.

Question 1b-3. *Identify at least two real-world effects that you encountered and make this problem harder than the theory predicts, and explain how each one affected your tuning choices.*

3.5 Part 2: Rocket Model System

Open `simulation/rocket_model.py` in VS Code. In the terminal, run:

```
python simulation/rocket_model.py
```

Follow the general tuning protocol. Pay special attention to how the ground truth (LQR) solution handles the gravity disturbance. Note how the control signal must overcome gravity to maintain position. The annotated plot will show you how large the steady-state error is when $K_i = 0$—this is a key observation for the rocket system.

Question 2-1. *Why does the integral term reduce the steady-state error for this system but was less critical for the double integrator? (Hint: think about the role of the constant gravitational force g.)*

Question 2-2. *Compare your best results for the double integrator and the rocket model. Which system was harder to tune, and why?*

3.6 OPTIONAL: Part 3: Pendulum System

Open `simulation/pendulum.py` in VS Code. In the terminal, run:

```
python simulation/pendulum.py
```

Follow the general tuning protocol. Observe the system response and note whether the pendulum successfully reaches the upright position. Watch the animation that shows both the PID-controlled pendulum and the ground truth (LQR) solution.

Question 3-1. *The pendulum equation contains a* $\sin\theta$ *nonlinearity, whereas the double integrator and rocket are linear systems. However, now the LQR controller can successfully swing up the pendulum. Can you explain this? (Hint: think about Taylor expansion)*

4 References

1. Åström, K. J., & Hägglund, T. (2006). Advanced PID control. ISA-The Instrumentation, Systems, and Automation Society.

2. Ogata, K. (2010). Modern control engineering (5th ed.). Prentice Hall.

3. Dorf, R. C., & Bishop, R. H. (2011). Modern control systems (12th ed.). Pearson.