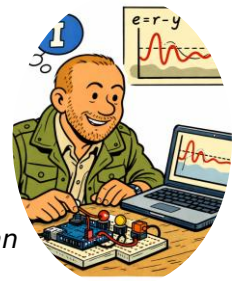
The Light Control Lab



Arduino & MATLAB

Adventure Professor Edition v1.0 (January 2026) , Prof. Edwin Zondervan

Exploring sensing, actuation, and feedback through light

1. Introduction

In this session, you will build and explore a **real feedback control loop** using:

- an **Arduino** as hardware interface,
- **MATLAB** as controller and data-acquisition environment,
- A **light-emitting diode (LED)** as an actuator,
- a **light-dependent resistor (LDR)** as a sensor.

The goal is **not** to write a controller from scratch, but to **understand how a controller behaves** when it interacts with the physical world.

You will proceed in small, logical steps:

1. Actuation only (turn an LED on/off)
2. Measurement only (sanity check of the light sensor)
3. Closed-loop control (PI control of light intensity)

2. Learning objectives

After this session, you should be able to:

1. Connect MATLAB to an Arduino and exchange signals with hardware.
2. Explain the difference between **actuation**, **measurement**, and **control**.
3. Interpret a feedback control loop in terms of:
    - setpoint
    - measurement
    - error
    - control action
4. Describe qualitatively what **proportional (P)** and **integral (I)** action do.
5. Observe how **environmental conditions** influence control performance.
6. Experimentally tune controller parameters to reduce oscillations.
7. Understand why **measurement noise** and **filtering** matter in control.

You are **not** expected to derive or program a PI controller yourself.
You are expected to **experiment, observe, and reason**.

3. System overview

You will work with the following system:

- **Actuator:** LED brightness (PWM output on Arduino pin D9)
- **Sensor:** Light-dependent resistor (LDR)
- **Measured variable:** Voltage at Arduino pin A0
- **Controller:** PI controller running in MATLAB
- **Plant:** LED → light → LDR → voltage divider

This is a **real physical control loop**, subject to:

- actuator limits,
- sensor noise,
- disturbances,
- nonlinearities.

You should think of the LED–environment–LDR combination as a small physical 'plant', similar to a process unit in chemical engineering.

4. Arduino basic concepts

What is Arduino Uno?

- A microcontroller board (not a computer)
- Reads inputs and writes outputs in real time
- Powered and programmed via USB

Main ports you will use:

- Digital pins (D0–D13): ON/OFF or PWM output (e.g. D9)
- Analog inputs (A0–A5): measure voltages (0–5 V)
- 5V and GND: power for sensors

Arduino IDE vs MATLAB:

- Arduino IDE: program runs autonomously on the board
- MATLAB: Arduino acts as I/O, control runs on your laptop

Figure 1 shows you what the Arduino IDE looks like. It can be downloaded from arduino.cc/en/software

What is Arduino IDE? Arduino IDE (Integrated Development Environment) is a software package that allows you to immediately communicate with Arduino. The IDE has its own programming syntax and the IDE software also allows you to upload your code to the Arduino board. This enables you to run programs on the board without a physical connection to a computer.

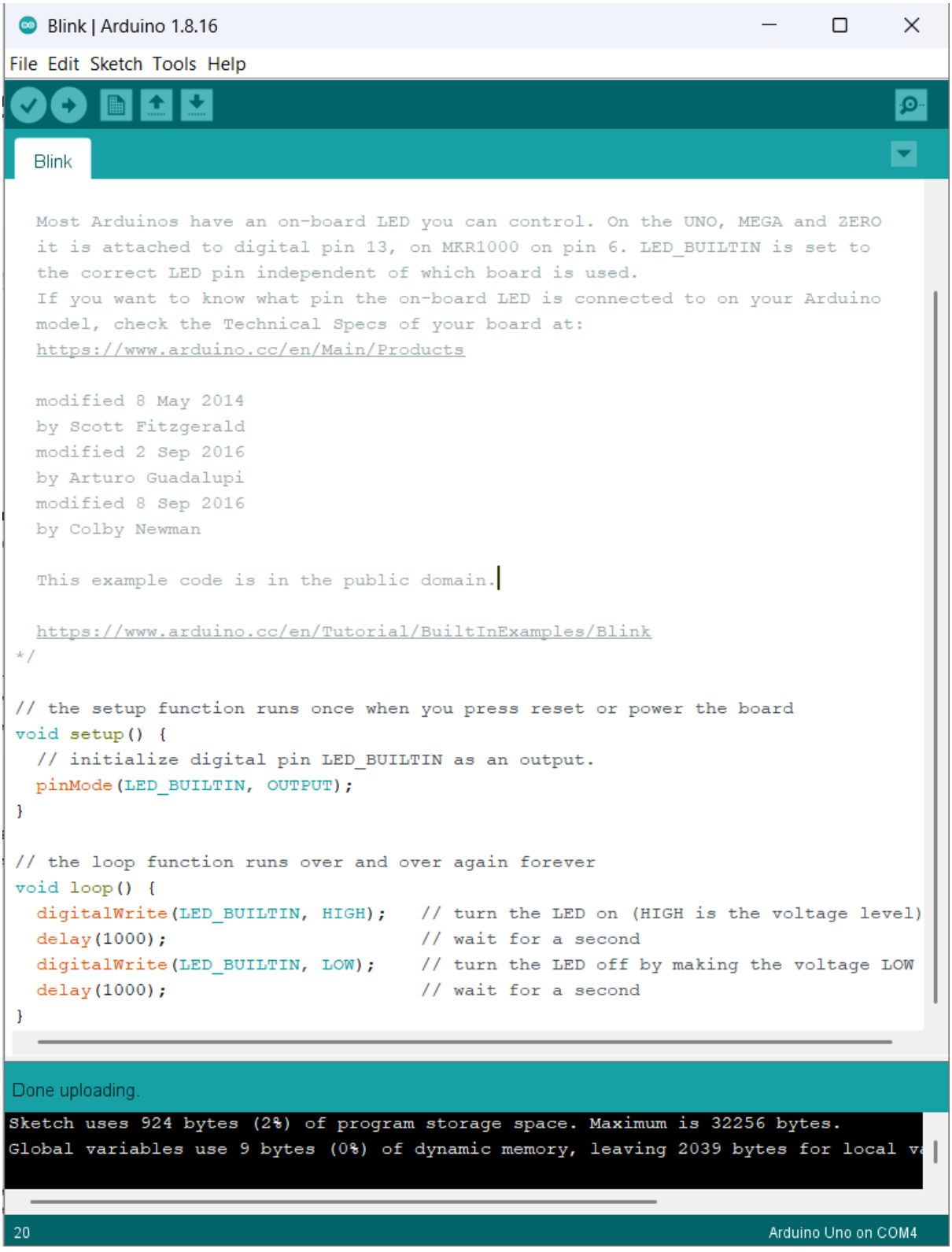


Figure 1: A Screenshot of the Arduino IDE, with code for blinking an LED

The Arduino IDE allows you to upload programs that run autonomously on the board. In this lab, however, we do not use the Arduino IDE: MATLAB acts as the controller, and Arduino is used purely as an input/output device.

5. Installing the Arduino support package for MATLAB

MATLAB can interact directly with Arduino via a support package. Check out: MATLAB Support Package for Arduino Hardware - File Exchange - MATLAB Central

- What you need:
  - MATLAB installed
  - Arduino board (e.g., Uno)
  - USB cable
- Install the support package (once):
  - Open MATLAB
  - Home → Add-Ons → Get Add-Ons
  - Search for "Arduino."
  - Install "MATLAB Support Package for Arduino Hardware."
- Test the connection:
  - `a = Arduino();`
  - If no error appears → ready to go

6. Hardware overview

The adventure professor has prepared a toolbox that you can use to successfully execute this tutorial.

It comprises the following:

- BT122900 Digital multimeter, which you can use to measure voltage, current, resistance, and transistor properties;
- An UNO R3 project kit, with an UNO R3 board, USB cable, breadboard, and several electronic components, including LEDs, resistors, LDR buttons, switch, jumper wires, IC, and several other components.
- Cross-head screw driver for electronic projects
- Point pliers.

All to get you started in becoming a nerd, to work out this tutorial and to invent many more ideas yourself! The costs for the above list are around €30,- (In case you consider buying your own professor's toolbox!).

In Figure 2, you can see what the different items look like.

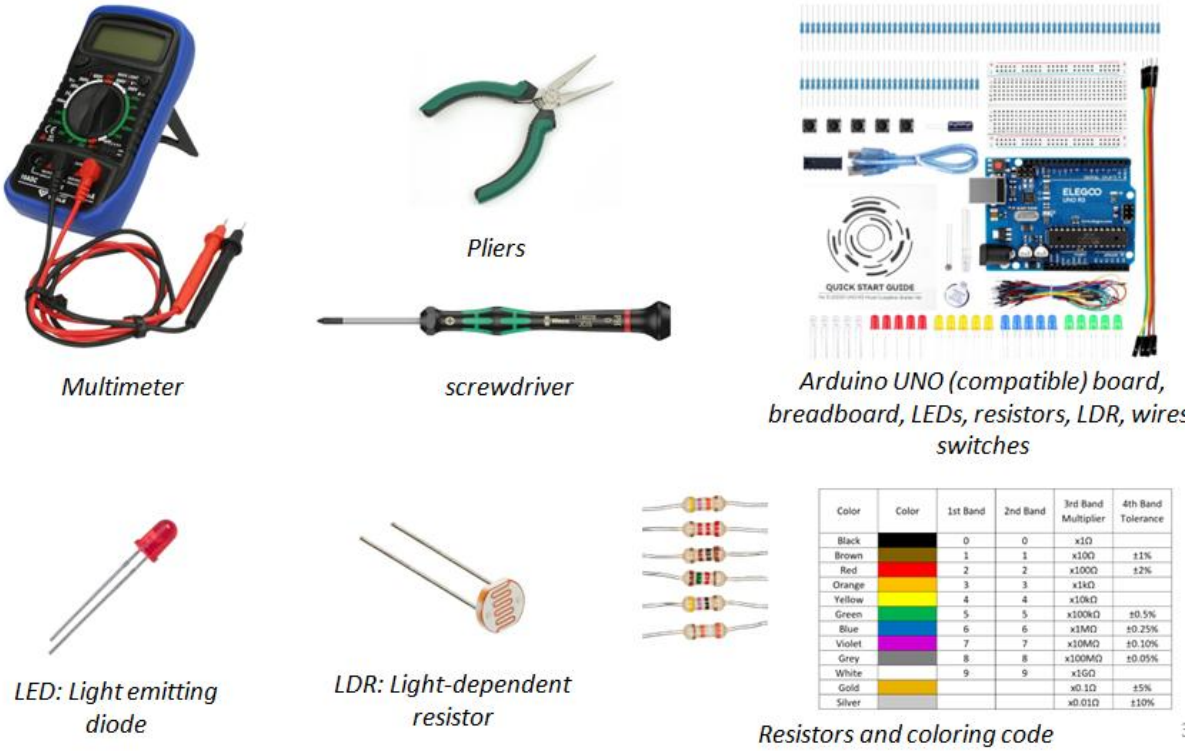


Figure 2: The content of the professor's toolbox

Consider the following :

Please read the instructions on how to properly use a multimeter (the cables have to be plugged into different ports, according to what you would like to measure).

Resistors have color coding that tells you what the actual resistance is (of course, you can measure the resistance of a resistor with a multimeter).

An LED is a diode, which only allows current to go in one direction. That is the reason that LEDs have a long leg and a short leg. So take care when connecting an LED.

7. Actuation only: Turning an LED on/off

Goal: The goal is to turn an LED on/off and to verify that MATLAB can **control hardware outputs** via Arduino.

You need the following hardware

- LED + 220–330 Ω resistor
- LED anode → **D9**
- LED cathode → resistor → **GND**

In Figure 3, you can see how to connect the components.

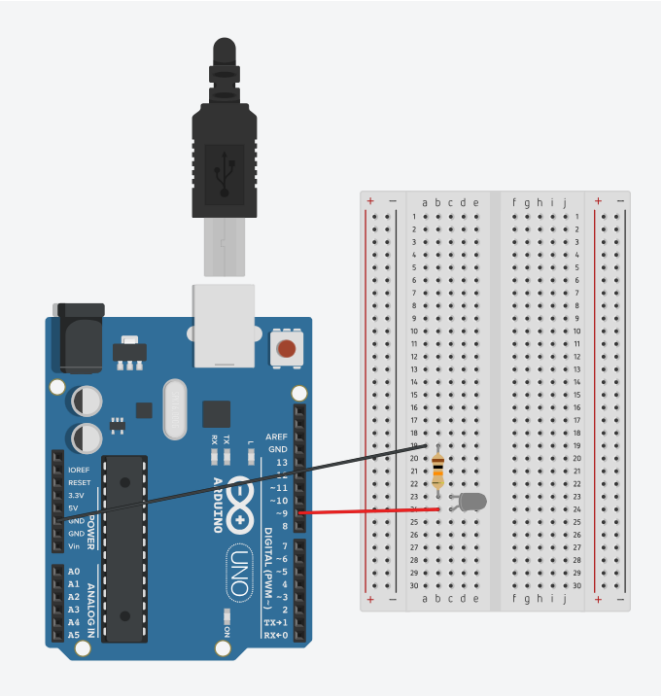


Figure 3: The position of the LED, resistor and jumper wires on the bread board and their connection to the UNO.

Why do we place a resistor? An LED by itself does not limit the current it draws, so if it were connected directly to a 5 V pin, it could draw too much current and be damaged. The resistor limits the current to a safe value, protecting both the LED and the Arduino pin. In the blink-LED experiment, the resistor ensures the LED turns on reliably without overheating or burning out.

MATLAB code: Blink_LED

With the command `a = arduino()`, we connect MATLAB to the Arduino board. To turn on a LED, we can use the command `writeDigitalPin(a,"D9",1)`. **D9** refers to the pin that we want to turn on (5V). the **1** means that we turn it on. If we write a **0** as in `writeDigitalPin(a,"D9",0)`, we can turn the LED off (0V). In the program below `Blink_LED.m` we use a for-loop to make the LED blink 10 times with 0.5-second pauses.

```
% LED blink test using Arduino and MATLAB

clear; clc;

a = arduino("COM4","Uno");    % change COM port if needed
```

```
for k = 1:10
    writeDigitalPin(a,"D9",1);   % LED ON
    pause(0.5);
    writeDigitalPin(a,"D9",0);   % LED OFF
    pause(0.5);
end
clear a
```

Observe

- The LED should blink on and off.
- If it does not:
    - check wiring,
    - check COM port,
    - check LED polarity.

Think about

- Where is the "decision" made to turn the LED on or off?
- Is this already feedback control?

Instead of making the LED blink, we could also make the LED fade in and out. Instead of `writeDigitalPin`, we use the function `writePWMDutyCycle`. `writePWMDutyCycle` tells the Arduino to output a **PWM (Pulse Width Modulation) signal** on a digital pin, where the pin rapidly switches between 0 V and 5 V. The value you give (between 0 and 1) sets the **duty cycle**, i.e., the fraction of time the signal is HIGH in each cycle. Changing the duty cycle changes the **average power** delivered to the LED, and therefore its brightness.

What is a PWM (Pulse Width Modulation)? A PWM is a way to control power using a digital signal by rapidly switching a pin **ON (5 V)** and **OFF (0 V)**. Instead of changing the voltage level, PWM changes the **fraction of time the signal is ON** (the duty cycle) within each fixed-length cycle. By varying this duty cycle, the average power delivered to a device (like an LED or motor) is smoothly controlled.

Note that writeDigitalPin produces a purely ON/OFF signal, while writePWMDutyCycle produces a PWM signal that allows continuous control of brightness.

MATLAB code: Fade_LED

With the same hardware setup as shown in Figure 3, we can now fade the intensity of the LED with the MATLAB code of `Fade_LED.m` given below. We connect the Arduino and then start a loop where we increase the value of a scalar u from 0 to 1. We increase it value with steps of 0.02, and each time we pause for 0.05 seconds before we increase. We then start a loop where we fade out the LED by decreasing the value of u from 1 to 0.

```
% LED Fade test using Arduino and MATLAB
a = arduino();
clear; clc;
```

```
for u = 0:0.02:1
    writePWMDutyCycle(a,"D9",u);
    pause(0.05)
end
for u = 1:-0.02:0
    writePWMDutyCycle(a,"D9",u);
    pause(0.05)
end
clear a
```

8. Measurement only: LDR sanity check

Goal: Understand what the **sensor measures**, before trying to control anything.

You need the following hardware

Wire the LDR as a voltage divider:

```
5V ── LDR ──┬── A0
            |
          10 kΩ
            |
           GND
```

- Any LDR is fine
- Use a **10 kΩ** resistor as a starting point

In Figure 4, you can see how to connect the components.

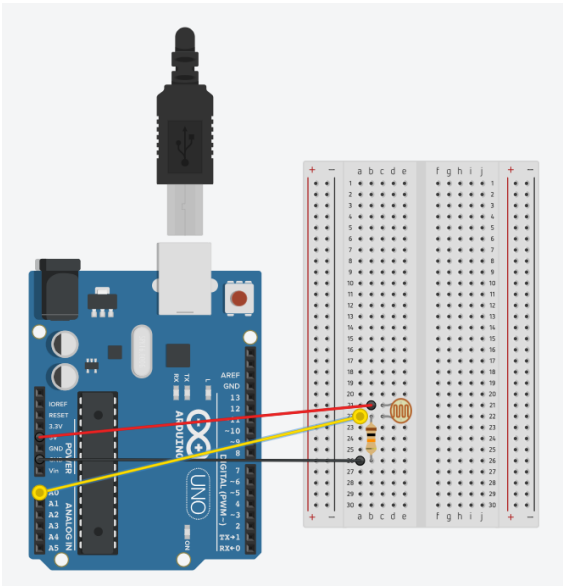


Figure 4: The position of the LDR, resistor, and jumper wires on the breadboard and their connection to the UNO.

What is a voltage divider? A voltage divider is a simple circuit made of two resistors in series that converts a fixed supply voltage into a **fraction of that voltage**, depending on their resistance ratio. We need it for the LDR because the Arduino can only **measure voltage**, not resistance: the LDR's resistance changes with light, and the voltage divider turns that changing resistance into a measurable voltage between 0 and 5 V.

MATLAB code: LDR sanity check

With the command `readVoltage(a,"A0")`, we can read a signal, in this case from port **A0**. In the MATLAB code below we perform a sanity check. We read the voltage that is measured by port **A0** for 10 seconds and print every 0.5 seconds the result to the screen.

```
% LDR sanity check (no control)
clear; clc;
a = arduino("COM4","Uno");
for k = 1:20
    v = readVoltage(a,"A0");
    fprintf("A0 = %.2f V\n", v);
    pause(0.5);
end
clear a
```

What to do while it runs

- Cover the LDR with your hand
- Shine your phone flashlight on it
- Change ambient lighting

Observe

- The voltage at A0 should change clearly.
- Typical values: ~0.5–4.5 V

Important insight

At this stage:

- You are **measuring**, not controlling.
- The system does not "react" to measurements.

9. Closing the loop: PI control of light

Goal: Observe **real feedback control** in action.

In figure 5 you can see the block diagram of this control setup.

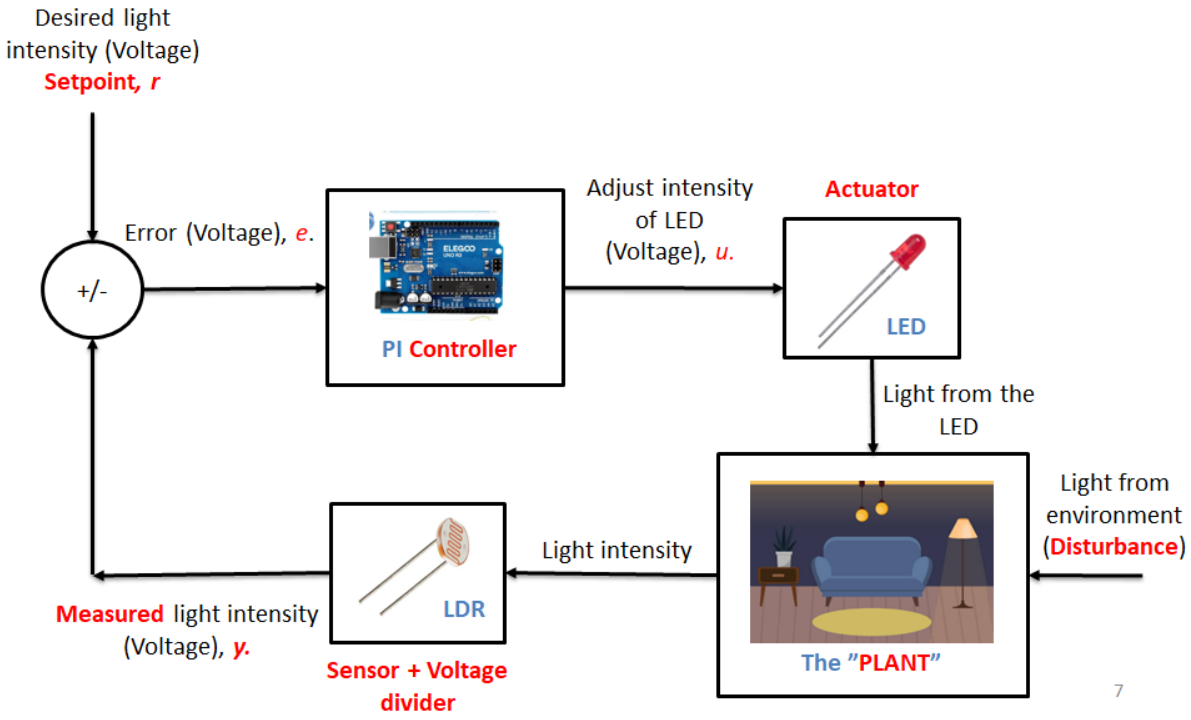


Figure 5: Block diagram of the experiment

You are now given a **complete PI controller**.
Do **not** change its structure , only the parameters.

PI control code (provided)

The MATLAB code below can be found in the `file PI_LED_control_with_filter.m`, it implements a PI-controller. In the section `YOU MAY CHANGE THESE PARAMETERS` you can modify the setpoint, r, the controller gains Kp and Ki and the measurement filter (to deal with noise). Maybe also good to know: the sample time Ts determines how often the controller updates; changing Ts affects stability and noise sensitivity.

```
% LED-LDR PI control with measurement filtering and anti-windup
clear; clc;
%% Connection
a = arduino("COM4","Uno");
pwmPin = "D9";
yPin   = "A0";
%% Timing
Ts   = 0.05;       % sample time [s]
Tend = 20;         % total runtime [s]
N    = round(Tend/Ts);
t    = (0:N-1)*Ts;
```

```
%% --- YOU MAY CHANGE THESE PARAMETERS ---
r     = 3.5;      % setpoint (sensor voltage) [V]
Kp    = 0.10;     % proportional gain
Ki    = 0.30;     % integral gain [1/s]
alpha = 0.2;      % measurement filter (1 = no filter)
%% ---------------------------------------
uMin = 0; uMax = 1;
y_raw = zeros(1,N);
y_f   = zeros(1,N);
u     = zeros(1,N);
I = 0;
y_f(1) = readVoltage(a,yPin);
for k = 1:N
    y_raw(k) = readVoltage(a,yPin);
    if k == 1
        y_f(k) = y_raw(k);
    else
        y_f(k) = (1-alpha)*y_f(k-1) + alpha*y_raw(k);
    end
    e = r - y_f(k);
    I = I + e*Ts;
    uUnsat = Kp*e + Ki*I;
    u(k) = min(max(uUnsat,uMin),uMax);
    if u(k) ~= uUnsat
        I = I - e*Ts;   % anti-windup
    end
    writePWMDutyCycle(a,pwmPin,u(k));
    pause(Ts);
end
writePWMDutyCycle(a,pwmPin,0);
clear a
figure;
plot(t,y_raw,t,y_f,t,r*ones(size(t)));
legend("raw","filtered","setpoint");
```

```
xlabel("Time [s]");
ylabel("Sensor voltage [V]");
grid on;
```

10: Experiments to perform

The main idea is that you play with this code according to the list of ideas provided below.

1 Setpoint sensitivity

- Change r
- Observe how LED brightness changes
- Notice how ambient light influences behavior

2 Disturbance rejection

- Cover the LDR
- Shine external light
- Observe recovery to setpoint

3 Proportional action (Kp)

- Increase Kp → faster but oscillatory
- Decrease Kp → slower but stable

4 Integral action (Ki)

- Set Ki = 0 → observe steady-state error
- Increase Ki → offset disappears, but oscillations may appear

5 Noise filtering (alpha)

- Set alpha = 1 → no filtering
- Set alpha = 0.1–0.3 → smoother signal
- Observe trade-off between smoothness and responsiveness

What you should realize

- Control performance is **not only about the controller**
- Actuators saturate
- Sensors are noisy
- The environment matters
- Perfect control is impossible — **robust control is the goal**

Short reflection question

What limited the performance of your control system most, and why?

Consider:

- controller tuning (P/I gains),
- actuator limits (LED saturation),
- sensor behavior (noise, nonlinearity),
- environmental effects (ambient light, disturbances).

Answer briefly (5–10 lines).
There is no single correct answer — your reasoning is what matters.

11. The Graphical user interface

In addition to the PI LED control, I have created a GUI (Graphical User Interface) that can be used for the same purpose. You can open `led_ldr_pi_gui.m` and run it. Figure 6 gives a screenshot of the user interface, which was created with MATLAB GUI (See: MATLAB GUI - MATLAB & Simulink)

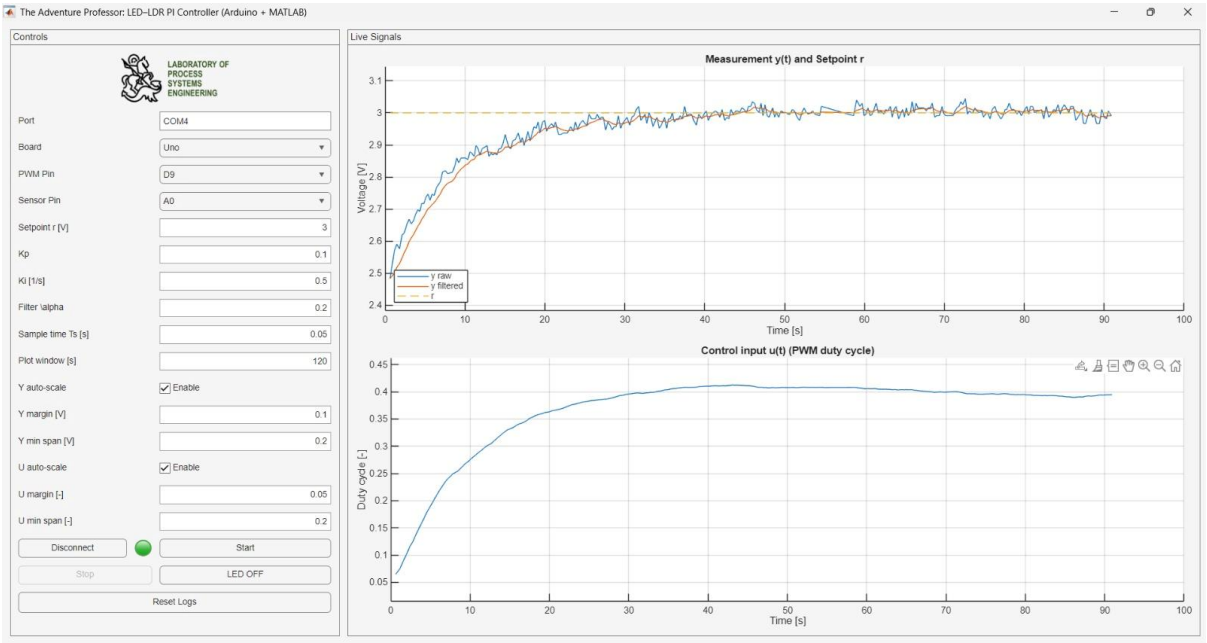


Figure 6: Screenshot of the PI LED Control user interface.

12. The Simulink version

I have also created a MATLAB SYSTEM object that can be read as a block in SIMULINK. This allows you to simulate similarly to what you have done elsewhere throughout the course. This file can be accessed through: `PI_led_ldr_control_simulink.mdl`. In Figures 7 and 8, you can observe how the Simulink model and simulation results can look. This Simulink version allows you to connect real hardware to the same block-diagram structures used in control theory.

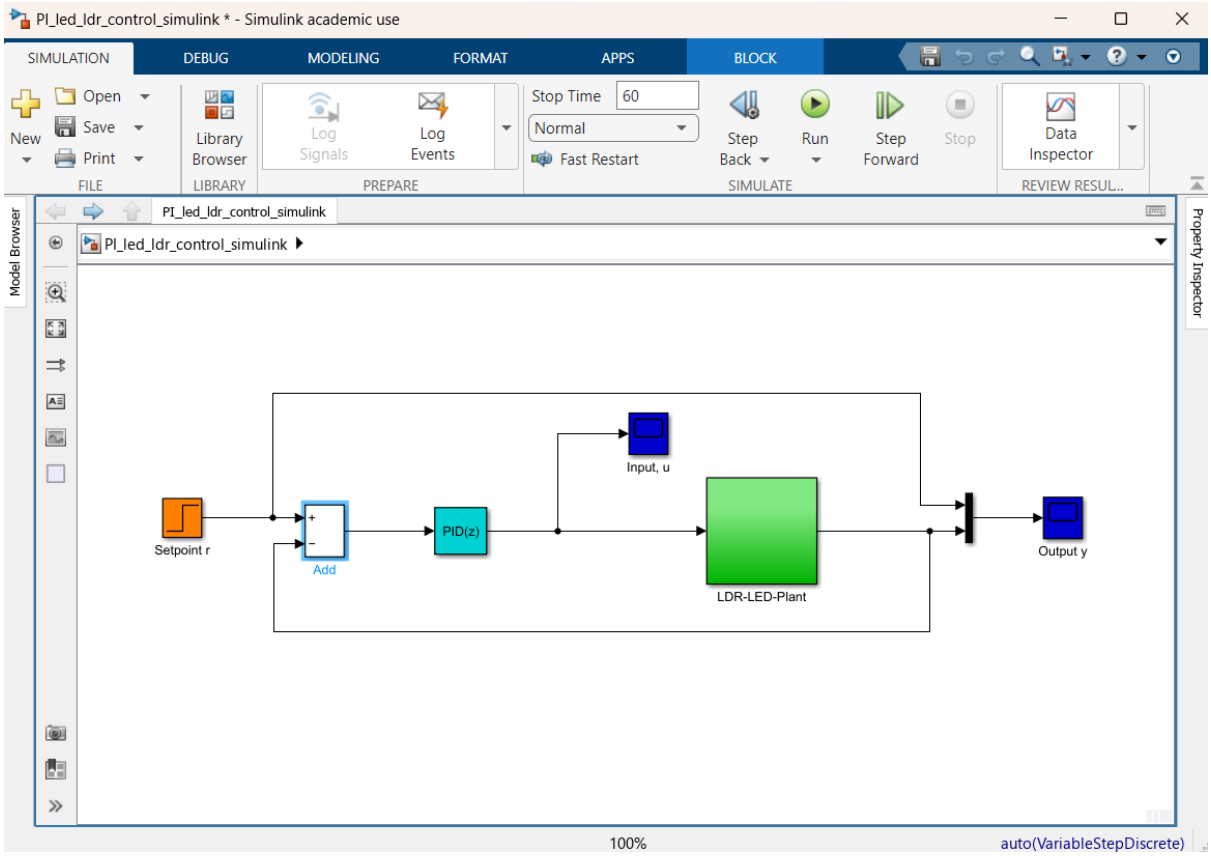


Figure 7: The SIMULINK model of the PI LED controller

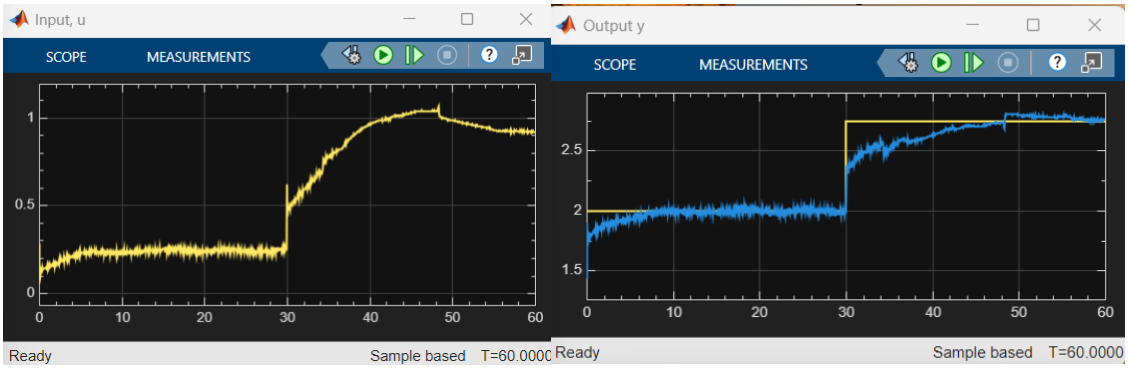


Figure 8: Screenshots of the simulation results, left the input u (duty cycle) and right the output, y (voltage measured over the LDR).

12. MATLAB files that go with this tutorial

`ArduinoLedLdrPlant.m`	File that is used in the MATLAB SYSTEM block, in Simulink
`Blink_led.m`	Code that makes an LED blink
`Fade_led.m`	Code that makes an LED Fade
`LDR_sanity_check.m`	Code that reads out the voltage over the LDR voltage divider
`Led_ldr_pi_gui.m`	Code that creates the MATLAB PI Control graphical user interface
`PI_led_control.m`	Code that creates the PI controller for the LED/LDR
`PI_led_control_with_filter.m`	Code that creates the PI Controller for the LED/LDR + filter
`PI_led_ldr_control_simulink.mdl`	Simulink implementation of the LED/LDR experiment