

# DISTRIBUTED REAL TIME CONTROL SYSTEMS

## MID-TERM REPORT

---

### Real-Time Cooperative Control of a Distributed Illumination System

---

**L04:**

Manuel Passadouro (80840)

[manuel.passadouro@tecnico.ulisboa.pt](mailto:manuel.passadouro@tecnico.ulisboa.pt)

Prof. Alexandre Bernardino  
Prof. José Gaspar

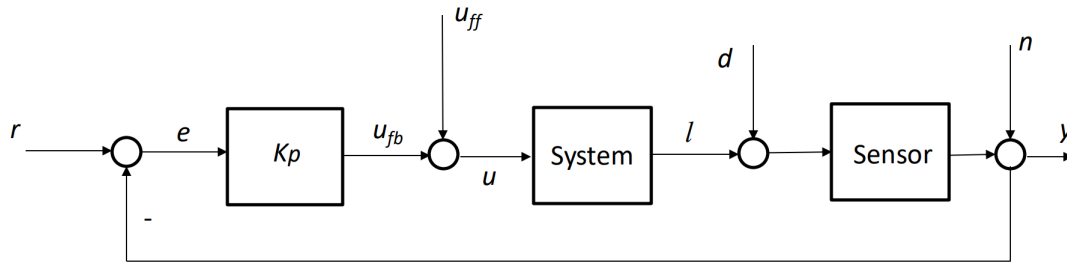
**MEEC – 2024/2025 – 2<sup>nd</sup> Semester, P3**

# Contents

<b>1</b>	<b>Introduction and Overview</b>	<b>2</b>
<b>2</b>	<b>LDR Characterization</b>	<b>5</b>
<b>3</b>	<b>Calibration and Steady State Characteristic</b>	<b>7</b>
<b>4</b>	<b>Controller Implementation</b>	<b>10</b>
<b>5</b>	<b>Can BUS Implementation</b>	<b>16</b>
<b>6</b>	<b>Final Remarks</b>	<b>17</b>
	<b>References</b>	<b>18</b>

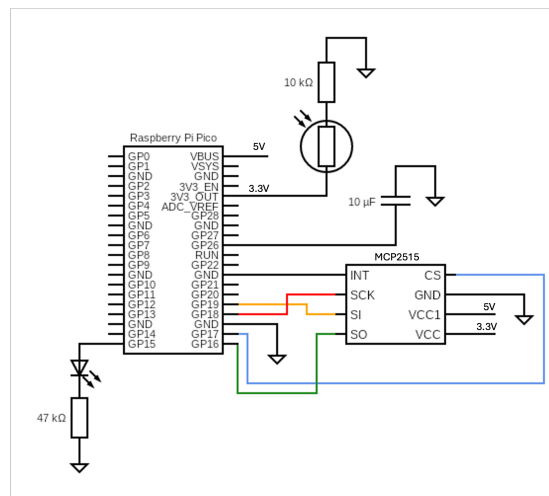
# 1 Introduction and Overview

The first stage of the project consists of creating a system that can sense and emit light in a stable and predictable way. It can be described by the block diagram in Figure 1:



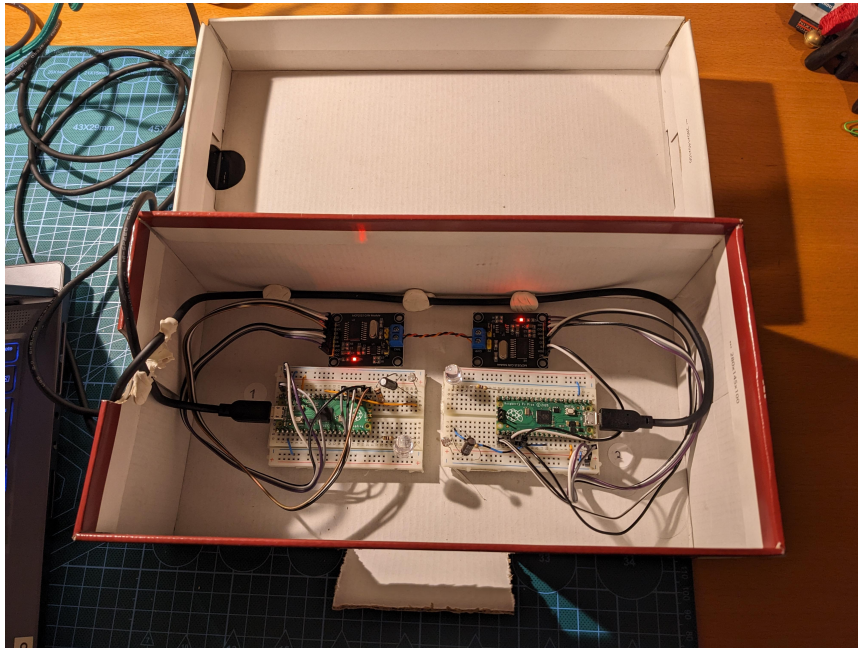
**Figure 1:** Block Diagram of the Luminary System.

The hardware used to implement the system consists of one LED (actuator) , one LDR (sensor), one Raspberry Pi Pico (micro controller) and one MCP2515 (CAN transceiver) assembled according to figure 2:



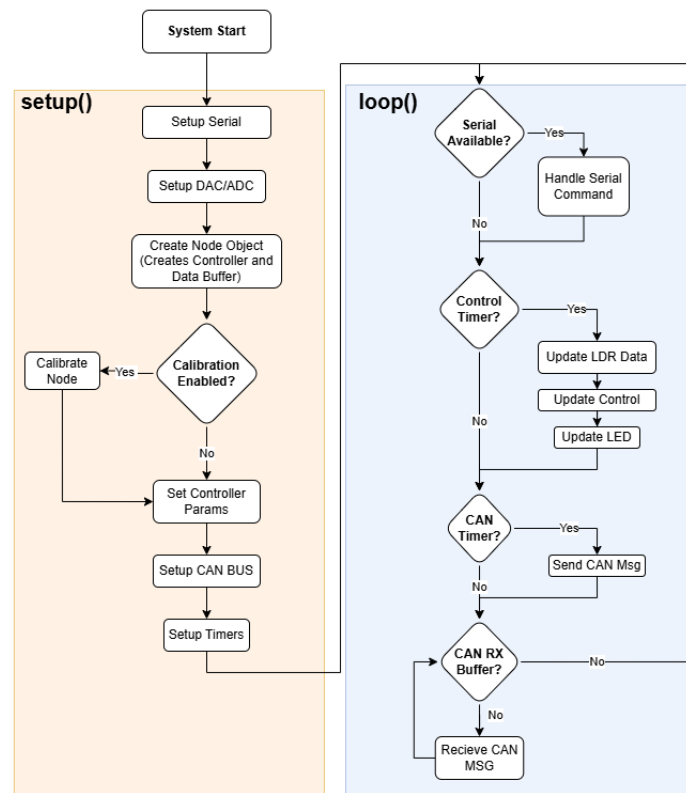
**Figure 2:** Hardware Schematic of the Luminary System.

Two luminaries are placed in a cardboard box with a white interior (good reflection of light). They are interconnected via a CAN bus and each can connect to a PC via a serial connection. Any holes in the box are covered with dark tape to keep external light out and the components inside are held down to keep testing conditions consistent. A door is cut out on the side of the box so that external light can be let in a controlled fashion:



**Figure 3:** Cardboard Box Where the Luminaries Are Housed.

The Software, developed in C++, makes use of the Arduino Pico and Pico SDK libraries. The flowchart in figure 4 provides an overview of the software architecture.



**Figure 4:** Luminary System Software Architecture.

A custom class *Node* encapsulates all the data acquired and computed as well as the functions required for the luminary's operation including controller functions defined in the *pid* class. Additionally, there is a class *DataBuffer* that stores and operates on a last minute buffer of node data.

Non-blocking code is implemented where possible, replacing active delays or polling with repeating timer interrupts and a flag system.

A command line interface allows users to view and adjust the parameters of the system. This interface and the data buffer are crucial tools that make debugging of the system much easier.

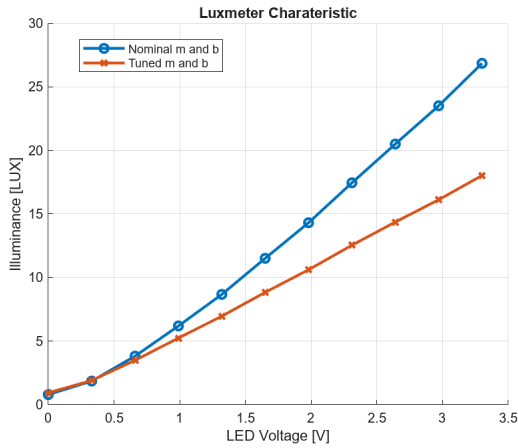
## 2 LDR Characterization

In order to use the LDR to implement a luxmeter, it is first necessary to obtain the values  $m$  and  $b$  from the formula that linearly relates the LDR resistance with the incident luminance (in LUX):

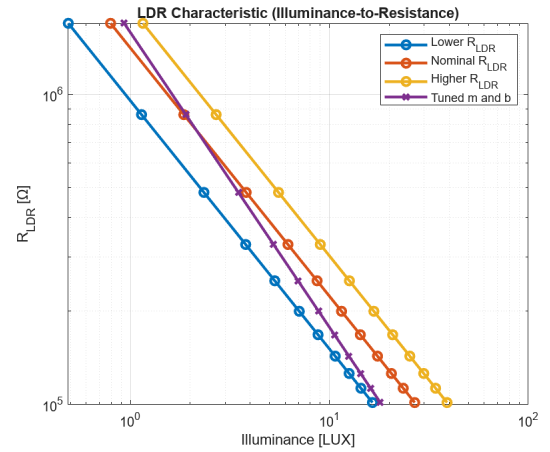
$$\log_{10}(R_{LDR}) = m \log_{10}(LUX) + b \quad (1)$$

From the LDR datasheet, it is possible to estimate an  $R_{LDR}$  of 225 k $\Omega$  for 10 LUX and a minimum slope ( $\lambda_{min}$ ) of 0.8. So,  $b = 6.15$  and  $m = -0.8$  can be used as default values for this model of LDR.

However, because of manufacturing tolerances and general unknown condition of the LDRs used, it is desirable to fine tune  $m$  and  $b$  to ensure a linear relationship between the data acquired from the ADC and then compute the value of illuminance. By collecting several samples across the range of actuation of the LED,  $m$  and  $b$  are iteratively altered in order to shape the best response:



**Figure 5:** Luxmeter Characteristic (Default Vs Tuned  $m$  and  $b$ ).



**Figure 6:** LDR Characteristic (Illuminance-to-Resistance, Log-Log).

Although the default values are acceptable in this particular case, it is possible to create a more linear response (figure 5) while staying within the boundaries of the LDR Characteristic (figure 6).

These tuned parameters can be stored and then loaded by the respective node, which is identified by its unique chip ID (figure 7).

```

// Configure node based on chip ID
if (strcmp(chipID, "E66118604B1F4021") == 0) {
    node_data.node_id = 1;
    node_data.ldr_m.value = -0.95f;
    node_data.ldr_b.value = 6.2f;
    node_data.G.value = 8.0f; // Default G if calibration is not run
}
else if (strcmp(chipID, "E660C0D1C71DA034") == 0) {
    node_data.node_id = 2;
    node_data.ldr_m.value = -0.93f;
    node_data.ldr_b.value = 6.15f;
    node_data.G.value = 6.5f; // Default G if calibration is not run
}
else {
    Serial.println("Unknown board ID. Using default calibration values.");
}

```

**Figure 7:** Default m, b and G Parameters for Each Node.

The computation of luminance values by the system is done as show in figure 8:

```

// Read from ADC
node_data.raw_adc = analogRead(A0);
node_data.filtered_adc = getFilteredADC(node_data.raw_adc);

// Compute the voltage at the capacitor
float V = (node_data.filtered_adc / 4095.0) * VCC;
node_data.c_voltage.value = V;

// Compute the resistance of the LDR
float R_LDR = R * ((VCC - V) / V);
node_data.ldr_resistance.value = R_LDR;

// Compute the total and external illuminance
node_data.ldr_lux.value = pow(10, (log10(R_LDR) - node_data.ldr_b.value) / node_data.ldr_m.value);

node_data.ldr_lux_extern.value = node_data.ldr_lux.value -
    (node_data.G.value * (float)((node_data.duty_cycle/ 4096.0)*VCC));

```

**Figure 8:** Illuminance Computation.

### 3 Calibration and Steady State Characteristic

At boot time, the system performs self-calibration to determine its static gain ( $G$ ). This is done by computing a linear regression across samples of the LED voltage (set by the PWM duty cycle) and the voltage across the capacitor on the LDR side ( $V_{Cap}$ ):

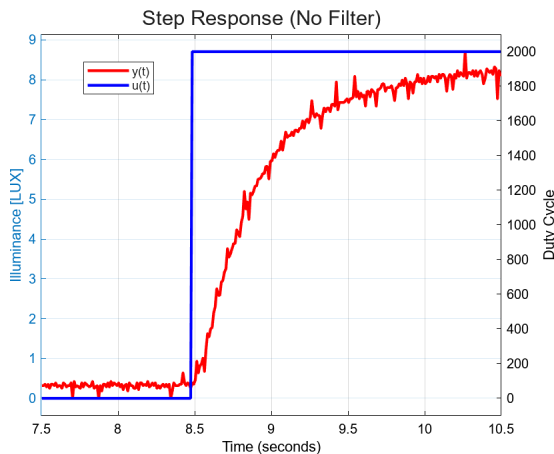
```
Duty Cycle: 0 - Duty Cycle Voltage: 0.00V - Filtered LUX: 0.32 - R_LDR: 4085000.25 Ohm
Duty Cycle: 409 - Duty Cycle Voltage: 0.33V - Filtered LUX: 1.62 - R_LDR: 900000.06 Ohm
Duty Cycle: 819 - Duty Cycle Voltage: 0.66V - Filtered LUX: 3.63 - R_LDR: 425638.31 Ohm
Duty Cycle: 1228 - Duty Cycle Voltage: 0.99V - Filtered LUX: 5.78 - R_LDR: 276363.63 Ohm
Duty Cycle: 1638 - Duty Cycle Voltage: 1.32V - Filtered LUX: 7.95 - R_LDR: 205526.31 Ohm
Duty Cycle: 2048 - Duty Cycle Voltage: 1.65V - Filtered LUX: 10.11 - R_LDR: 164255.31 Ohm
Duty Cycle: 2457 - Duty Cycle Voltage: 1.98V - Filtered LUX: 12.26 - R_LDR: 137302.16 Ohm
Duty Cycle: 2867 - Duty Cycle Voltage: 2.31V - Filtered LUX: 14.49 - R_LDR: 117570.09 Ohm
Duty Cycle: 3276 - Duty Cycle Voltage: 2.64V - Filtered LUX: 16.63 - R_LDR: 103434.90 Ohm
Duty Cycle: 3686 - Duty Cycle Voltage: 2.97V - Filtered LUX: 18.89 - R_LDR: 91865.67 Ohm
Duty Cycle: 4096 - Duty Cycle Voltage: 3.30V - Filtered LUX: 21.05 - R_LDR: 83068.18 Ohm
Calibration complete.

Hello, this is node 2
Chip ID: E6:60:C0:D1:C7:1D:A0:34
LDR Slope (m): -0.93000
LDR Intercept (b): 6.15000
Slope (G): 6.42838
Desk Occupancy: Unoccupied
```

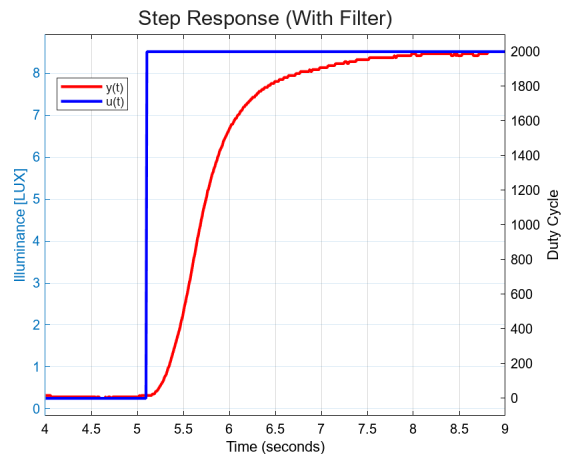
**Figure 9:** Serial Monitor Output After Static Gain Calibration.

To expedite the debugging process, calibration is turned off and a default value specific to the luminary is used (Figure 7). Only when there are significant changes to the environment is the calibration performed again.

Using the interface, the duty cycle  $u(t)$  can be set manually in order to evaluate the response of the luminance signal  $y(t)$ :



**Figure 10:** Step Response of the System (No Filter).



**Figure 11:** Step Response of the System (50 Sample Filter).

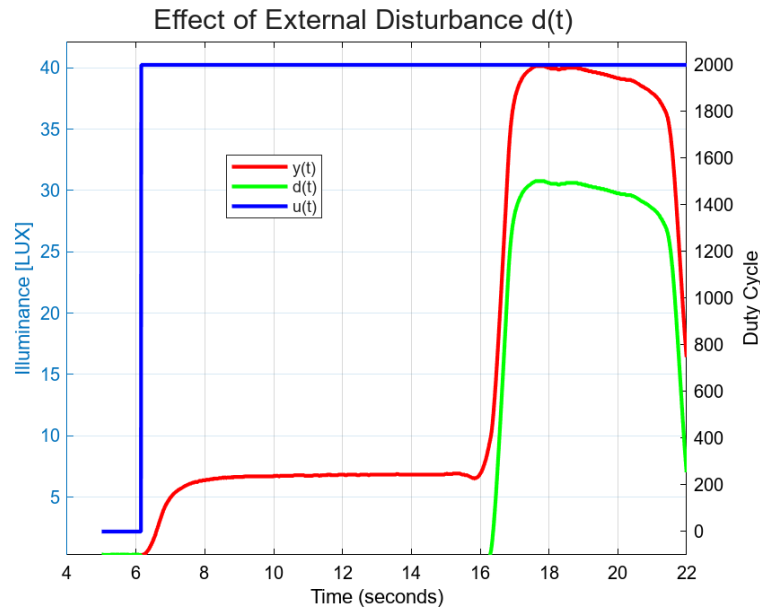
Figures 10 and 11 demonstrate the importance of adding a digital low-pass filter to the ADC output. The greater the number of samples used by the filter, the slower the response but lesser the noise coming from the sensor readings.



The signal  $d(t)$  represents the light coming from external sources (ambient light or other luminaries). It can be computed if an estimation of  $G$  is known:

$$y(t) = Gu(t) + d(t) \quad (2)$$

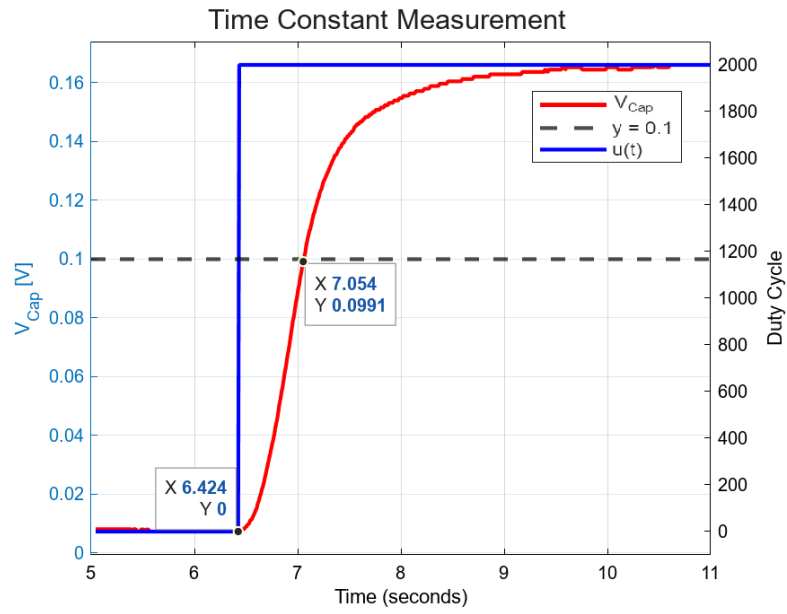
$$d(t) = \frac{y(t)}{Gu(t)} \quad (3)$$



**Figure 12:** Effect of External Disturbances on the Response of the System.

In Figure 12 it is possible to see how  $d(t)$  rises with  $u(t)$  when the duty cycle is steady, indicating the presence of external illuminance.

The time constant of the system  $\tau_c$  can be estimated by determining how long it takes for  $V_{Cap}$  to reach 63% of its final value in a step response:



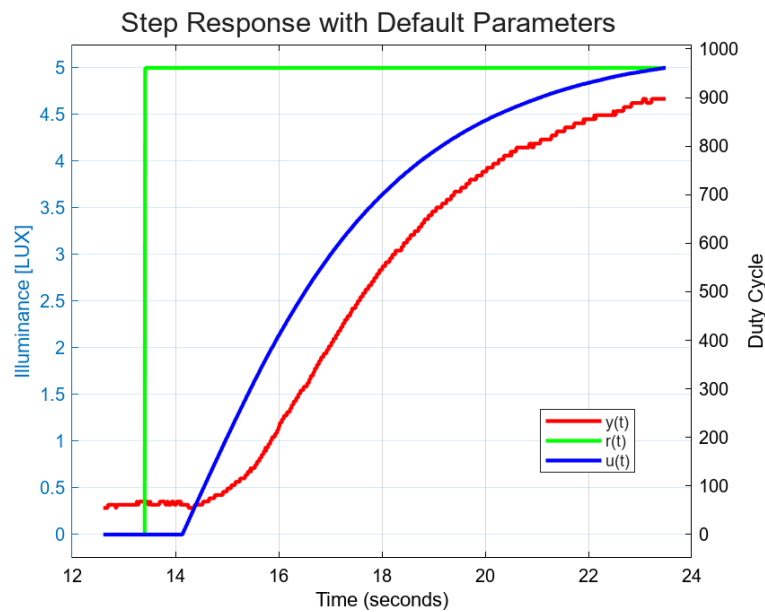
**Figure 13:**  $V_{Cap}$  Step Response and  $\tau_C$  Measurement.

The values  $\tau_C$  and the static gain  $G$  provide a starting point for the tuning of controller parameters, more specifically  $K = \frac{1}{G}$  and  $T_i = \tau_C$ .

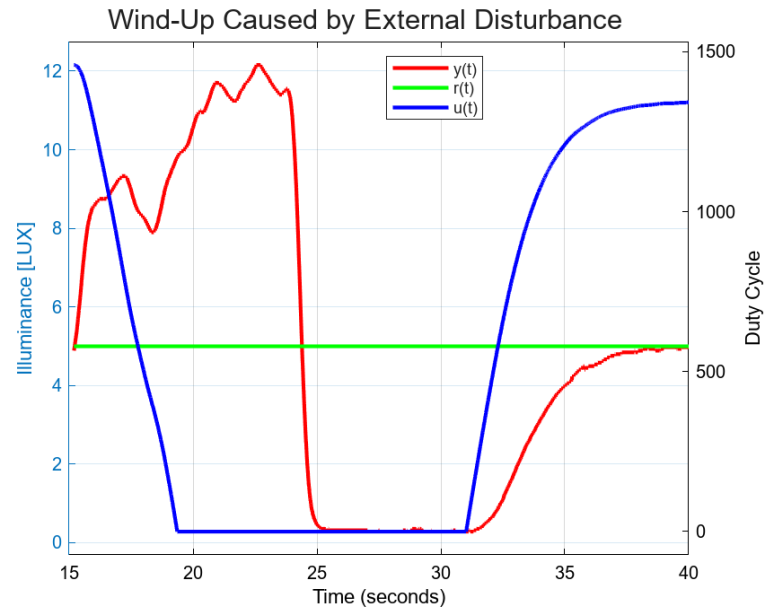
## 4 Controller Implementation

Focusing on luminary 2, the initial controller implementation is based on the code provided in the lectures with the following default parameters:  $K = 0.15$ ,  $h = 0.01$ ,  $b = 1$  and  $T_i = 0.6$ .

Upon testing, two issues are immediately apparent: First, the chosen parameters make the system far too slow, effectively taking several seconds for the LED to provide the proper illuminance (figure 14). Also, due to the lack of an anti-windup feature, the integration error is compounded, which leads to situations where a strong enough or long enough disturbance to the illuminance signal results in massive delays in the response of the system, even if parameters are changed to make it faster (Figure 15).

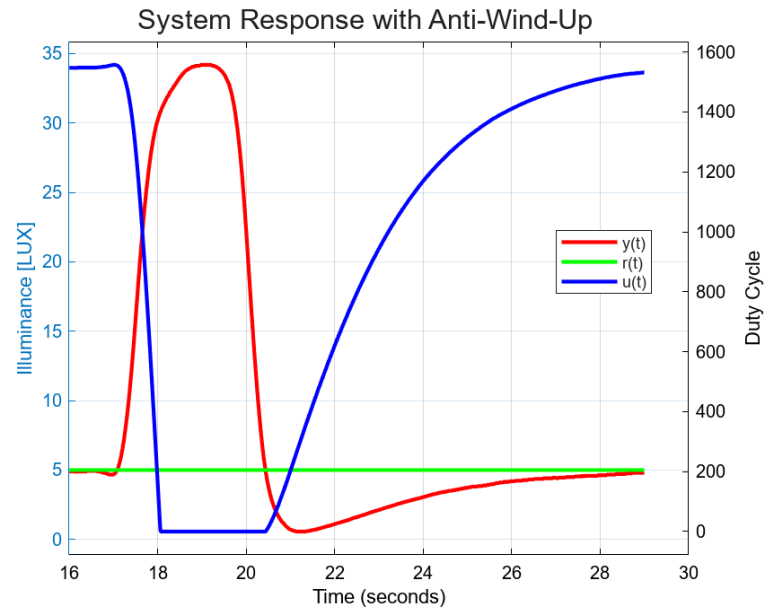


**Figure 14:** Step Response of the System With Default Controller Parameters.



**Figure 15:** Demonstration of Wind-Up Caused by External Disturbance.

By implementing anti-wind-up into the controller, the effect of error accumulation due to integrator action is nullified (Figure 16).



**Figure 16:** System Response After Anti-Wind-Up implementation.

Further changes are made to the original *pid* class, including the ability to enable or disable the feedforward and feedback mechanisms. The controller parameters are adjusted to improve performance. The figures below illustrate the final version of the controller, with added comments.

In this final implementation (figure 17), the controller operates as a PI controller. Since the system uses a fast actuator (LED) and a relatively slow sensor (LDR), the derivative term is unnecessary and potentially undesirable. The LDR's delayed response leads to outdated measurements, making the derivative term prone to amplifying noise without providing meaningful predictive control.

```

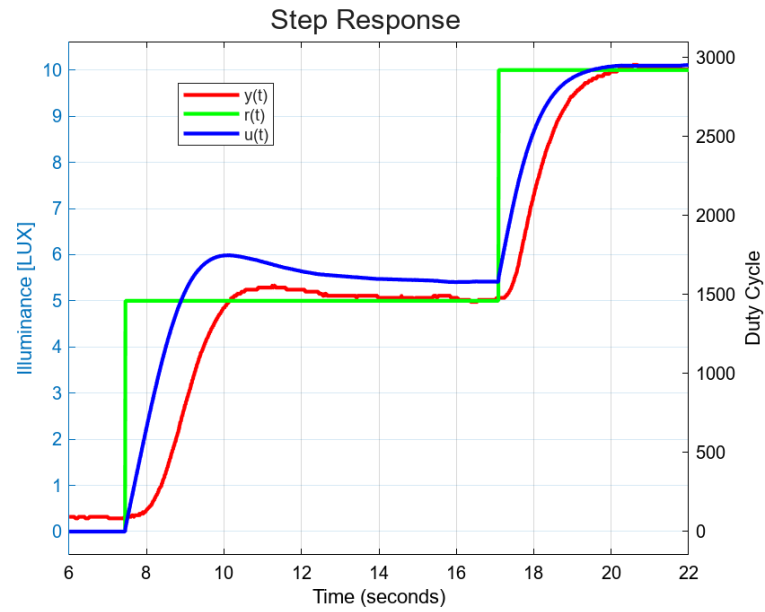
10 float pid::compute_control(float r, float y, float G){
11     float P = K * (b * r - y); // Proportional term w/ setpoint weighting (b)
12     //float ad = Td / (Td + N * h);
13     //float bd = Td * K * N / (Td + N * h);
14     //D = ad * D - bd * (y - y_old); // Differential term (not used)
15     float feedforward = uff * (r/((G/4095) * VCC)); // Get Duty Cycle directly from r
16     // (set uff to 0 to disable)
17
18     //float feedforward = uff * r;
19     float u = (fb_flag) ? (P + I + feedforward) : (feedforward); // FB + FF or just FF
20
21     if (u < 0) // Output Saturation
22         u = 0;
23     if (u > 4095)
24         u = 4095;
25     return u;
26 }
27
28 inline void pid::housekeep(float r, float y, float u){
29     float e = r - y; // Compute Error
30     float bi = K * h / Ti; // Integral gain
31     float ao = h / Ti; // Anti-windup back calculation gain
32
33     I += K_old * (b_old * r - y) - K * (b * r - y); // Bumpless transfer correction
34
35     I += bi * e + ao * (u - (I + K * (b * r - y))); // Update integral with anti-windup
36
37     //I += K * h / Ti * e; // Integral term (h is the sampling time)
38     //y_old = y; // Update y_old Por Differential
39     // term computation (not used)
40
41     K_old = K; // Store previous parameters for next cycle
42     b_old = b;
43 }

```

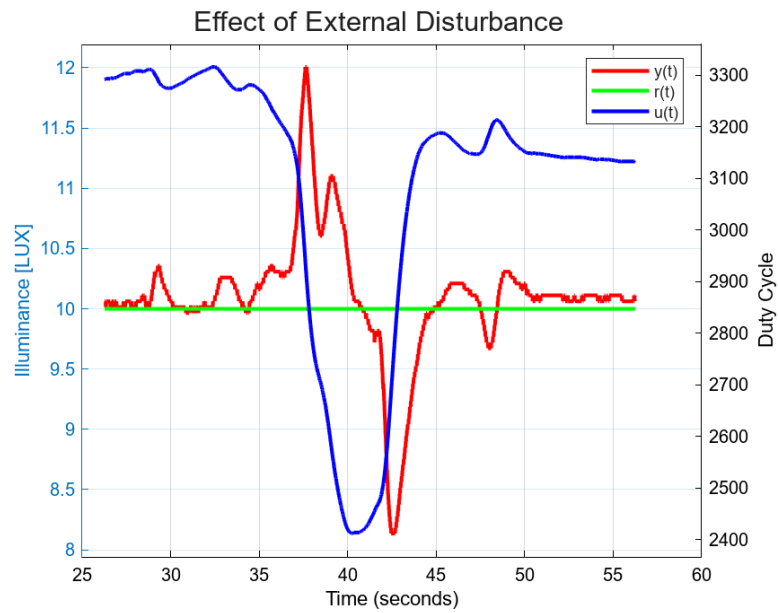
**Figure 17:** Final Controller Implementation.

By fine tuning the parameters, it is possible to improve the responsiveness of the system. This is achieved by increasing  $K$ ,  $b$  and the feedforward component  $u_{ff}$ , or decreasing  $T_i$ . This must be done with caution to avoid overshooting and flicker.

Figures 18 and 22 demonstrate the response of the system with the final parameters.



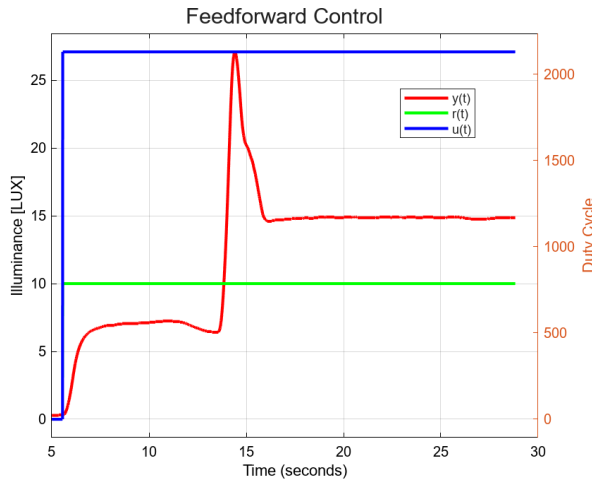
**Figure 18:** Step Response of the System With Final Controller Parameters.



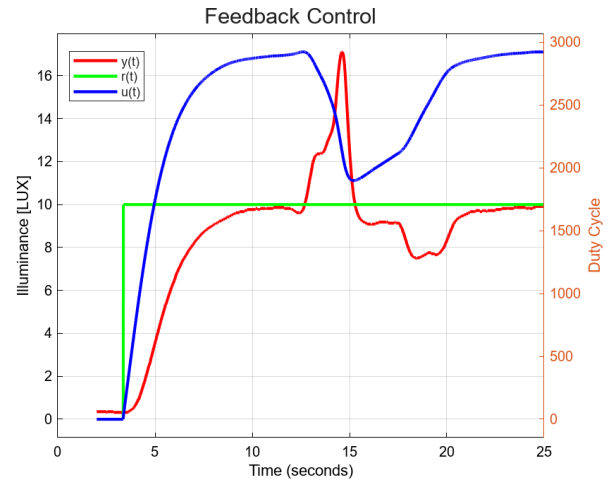
**Figure 19:** Effect of External Disturbance on the System With Final Controller Parameters

With these parameters, the system is faster, there is some small overshoot (less than 0.5 LUX) and the steady-state error is acceptable. The system reacts quickly to disturbances such as irregular ambient light, returning to the reference illuminance whenever the capabilities of the actuation allow it to.

By enabling and disabling the feedforward and feedback mechanisms, their effect on the system response can be studied independently:



**Figure 20:** System Response with Feedforward-Only Controller.



**Figure 21:** System Response with Feedback-Only Controller.

From Figure 20, it can be observed that the response to reference step-change is practically instantaneous, however the steady-state error is significant since there is no way for the system to evaluate how far off it is from the reference. Additionally, there is no rejection of disturbance, making feedforward-only a poor solution for the system, outside of ideal conditions.

Using only the feedback mechanism (Figure 21), this system now has noticeable lag in the actuator response, nevertheless the ability to follow the reference signal, even with strong disturbances make this mechanism adequate for the final implementation of the controller.

Indeed, the feedforward should be used only as a complement to the PI controller, to improve its speed, but not on its own.

The final parameters for the controller on luminary 2 are as follows:

```
void Node::set_controller_params() {
    switch (node_data.node_id) {
        case 1: // For node 1
            controller.set_K(1.0f);
            controller.set_h(0.01f);
            controller.set_b(1.0f);
            controller.set_Ti(0.05f);
            controller.set_uff(0.0f);
            break;

        case 2: // For node 2
            controller.set_K(20.0f);
            controller.set_h(0.01f);
            controller.set_b(0.3f);
            controller.set_Ti(0.08f);
            controller.set_uff(0.001f);
            break;

        default:
            //Keep default values
            break;
    }
}
```

**Figure 22:** Final Controller Parameters (Luminary 2).

The parameters on luminary 1 are kept as default in order to have a point of comparison during the tuning phase. In the second stage of the project, all luminaries will have their own set of parameters.

Taking advantage of data available in the last minute buffer, it is possible to compute some performance metrics.

The energy consumption of the system can be approximated to what is spent by keeping the LED on. Using the method discussed in the lab and assuming  $P_{MAX} \approx 10\mu A \times VCC$  (the current is measured with the LED at max duty cycle) it is possible to compute energy consumption since system start-up.

For a scenario similar to the one on Figure 18, the total energy consumption is 310 mJ.

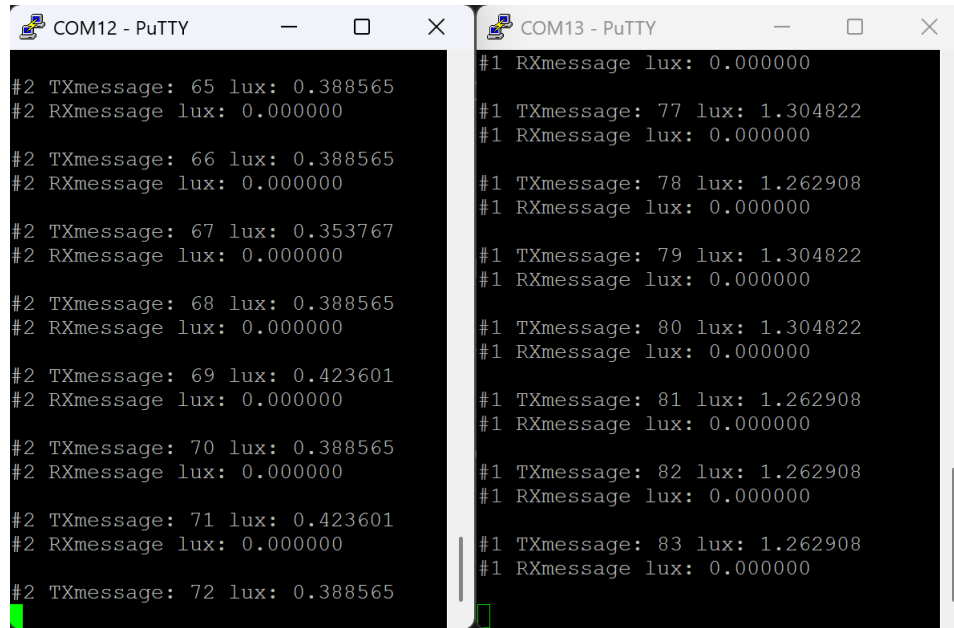
As for the visibility error and flicker, in steady-state after a step response, the system measures approximately 0.2 LUX and  $2 s^{-1}$  respectively. These values are higher in the presence of disturbance, as is expected. In steady-state, these results are considered satisfactory, with no significant discomfort suffered by the user.

The jitter of the system, for the set sampling rate of 100 Hz, is unnoticeable. This can be checked in the timestamps stored in the data buffer, which are consistently 1 ms apart from one to the next. It is possible to get more precision by using *micros()* instead of *millis()* to obtain the timestamps, but this implies more overhead, and it is clear that the system is not affected by jitter at this sampling rate.



## 5 Can BUS Implementation

In order to test the functionality of the CAN bus, a simple back and forth exchange between the two luminaries is programmed. In this case, one luminary is able to receive the illuminance measured by the other, this is vital for the implementation of distributed control in the second stage of the project.



The image displays two side-by-side serial monitor windows. The left window, titled 'COM12 - PuTTY', shows messages from 'luminary 2' (ID #2). It contains a series of TX and RX messages where TX messages report lux values (0.388565, 0.388565, 0.353767, 0.388565, 0.423601, 0.388565, 0.423601, 0.388565) and RX messages all report 0.000000. The right window, titled 'COM13 - PuTTY', shows messages from 'luminary 1' (ID #1). It contains RX messages all reporting 0.000000 and TX messages reporting lux values (1.304822, 1.262908, 1.304822, 1.304822, 1.262908, 1.262908, 1.262908, 1.262908).

```
COM12 - PuTTY
#2 TXmessage: 65 lux: 0.388565
#2 RXmessage lux: 0.000000

#2 TXmessage: 66 lux: 0.388565
#2 RXmessage lux: 0.000000

#2 TXmessage: 67 lux: 0.353767
#2 RXmessage lux: 0.000000

#2 TXmessage: 68 lux: 0.388565
#2 RXmessage lux: 0.000000

#2 TXmessage: 69 lux: 0.423601
#2 RXmessage lux: 0.000000

#2 TXmessage: 70 lux: 0.388565
#2 RXmessage lux: 0.000000

#2 TXmessage: 71 lux: 0.423601
#2 RXmessage lux: 0.000000

#2 TXmessage: 72 lux: 0.388565

COM13 - PuTTY
#1 RXmessage lux: 0.000000

#1 TXmessage: 77 lux: 1.304822
#1 RXmessage lux: 0.000000

#1 TXmessage: 78 lux: 1.262908
#1 RXmessage lux: 0.000000

#1 TXmessage: 79 lux: 1.304822
#1 RXmessage lux: 0.000000

#1 TXmessage: 80 lux: 1.304822
#1 RXmessage lux: 0.000000

#1 TXmessage: 81 lux: 1.262908
#1 RXmessage lux: 0.000000

#1 TXmessage: 82 lux: 1.262908
#1 RXmessage lux: 0.000000

#1 TXmessage: 83 lux: 1.262908
#1 RXmessage lux: 0.000000
```

**Figure 23:** Serial Monitor Output Showing Exchange of CAN Messages Between Luminaries.

## 6 Final Remarks

Using the *micros()* function, the processing time for the discrete tasks performed by the system is measured. Results are detailed in the table below:

Task	CPU Time [ $\mu$ s]
Serial Command Processing	300 ~ 350
LDR Data Update	70 ~ 80
Control Update	30 ~ 40
CAN Tx	70
CAN Rx	< 1

**Table 1:** Measured CPU Time for Different Tasks.

All tasks execute well below the sampling rate of the controller ( $10\mu$ s). However, the CAN bus tasks are handling very lightweight messages so an increase in CPU time is expected as more complexity is added. One important edge case is when the system prints the data buffer on the terminal, printing all of the data can take up to 1 s.

The System can be improved by implementing concurrency between tasks, this can be achieved by way of multi-core processing or using a real-time kernel like FreeRTOS. This way, is it possible to define higher priority tasks like the controller computations that will take precedence over less important task that might block CPU access. It also ensures that shared data is protected from simultaneous access thorough mechanisms of mutual exclusion.

Further improvements can be made, such as using the MCP2515 interrupt signal to trigger the CAN receive task, as opposed to actively polling the receive buffer. The CAN implementation can be refined by using masks and a graphical interface can be added for easier data viewing and storing.

## References

- [1] Prof. Alexandro Bernardino, SCDTR Course Slides, IST, 2025.
- [2] Arduino-Pico. “Arduino Core for Raspberry Pi RP2040,” <https://github.com/earlephilhower/arduino-pico>, 2025.
- [3] Raspberry Pi. “Raspberry Pi Pico SDK Documentation,” <https://www.raspberrypi.com/documentation/pico-sdk/>, 2025.
- [4] Rpi Pico Datasheet:<https://datasheets.raspberrypi.com/pico/pico-datasheet.pdf>.
- [5] LED Datasheet:<https://fenix.tecnico.ulisboa.pt/downloadFile/845043405623743/LED.pdf>.
- [6] LDR Datasheet:<https://fenix.tecnico.ulisboa.pt/downloadFile/845043405623742/PGM5659D.pdf>.
- [7] Project Repository: [https://github.com/manuel-passadouro/SCDTR\\_Project](https://github.com/manuel-passadouro/SCDTR_Project).