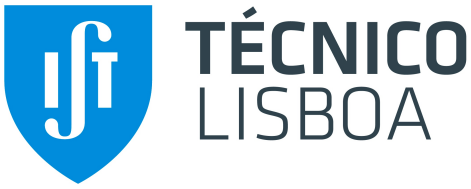


Instituto Superior Técnico

Distributed Real Time Control Systems

Master's degree in Electrical and Computer Engineering

Report 1

101784 Israel Sother

IST, april of 2022

Contents

1 Introduction

The high price in electricity combined with many after hours work on offices around the world has increased the research for smart illumination solutions. In this context this project aims to develop a smart system for lightning an office like scenario based on distributed real time control. Due to size and cost limitations, a scaled down version will be used for development.

In the first part of the project the goal is to create a stand alone smart luminary with a controller to track the illuminance reference. Later, in the second part, this luminary will be integrated in a network of luminaries with a distributed control algorithm.

2 Methodologies

The luminary system is composed of three key components: one LED (light emitting diode), one LDR (light dependent resistor), and one microcontroller. The LED is responsible for lightning the ambient, the LDR measures the current illuminance by changing its resistance accordingly to the light shed upon the top surface. And last the microcontroller read the measures and based on the reference drives the LED to reach the set point. The chosen models are:

- 10 mm Bullet Pure White LED *OSW5DKA201A*,
- PGM5**** series, Epoxy resin packaged LDR,
- Raspberry Pi Pico RP2040.

The components are mounted on a breadboard for connection, with some resistors are used in order to regulate the current flow, as the figure 1 shows:

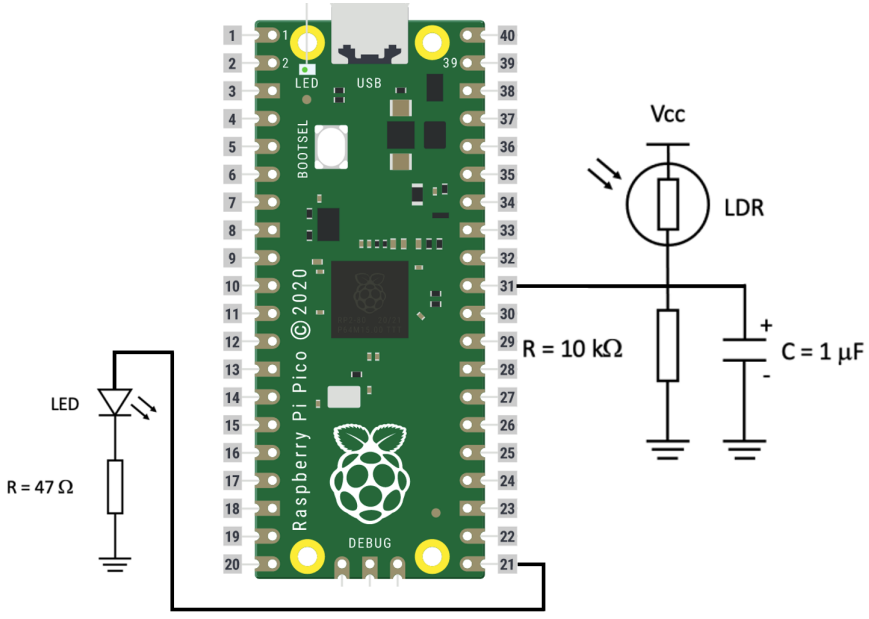


Figure 1: Luminary circuit schematic

The capacitor connected to the LDR is to filter the high frequency noise in the measurements. The power is supplied by the micro-USB port on the Raspberry Pi Pico, as the system do not consume much power.

Regarding the algorithm development and programming of the microcontroller, the Arduino-Pico package was chosen, due to its easy of use and similarity with the well know Arduino code. The Visual Studio Code was used as IDE to facilitate the work with several files and classes.

3 Experimental Setup

The physical setup it was pretty simple, after the circuit was mounted on the breadboard, the hole set was put inside a cardboard box to isolate the model an simulate a "office like" environment. This helps to minimize external disturbances on the system, and allows for better identification of the parameters.

The software setup is more intricate, so to ease the explanation it will be divided in a few topics.

3.1 Software

One of the advantages of using C++ over C is to be able to use classes, this approach helps the understanding and maintenance of the code. For this project 6 main classes were used. Their header dependency graph is as shown by figure 2.

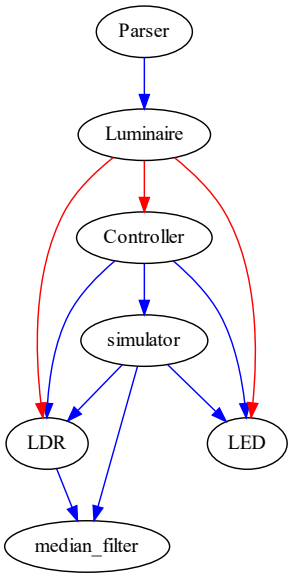


Figure 2: Header dependency graph

3.1.1 LED

The LED class stores the PWM pin used to drive the LED, and has a function to set a new duty cycle to the PWM, thus changing the overall brightness of the LED. There is also a function that blinks the LED for a few times. This function is used on the startup of the board to signal that its ready to serial connection.

3.1.2 LDR

The main goal of the LDR class is, like the LED, to handle the LDR component. It stores the ADC pin used to read the LDR voltage and has a few handling functions to read the ADC, and convert the readings to resistance, lux, or volts. A few extra features were implemented as average or do a median with n measures of the ADC.

The conversion between ADC value and voltage is done with the expected Vcc (3.3v) and the ADC resolution (12 bits), while the conversion between voltage on the ADC and LDR resistance is done by knowing the R1 value (10kOhms). Lastly the conversion between resistance and lux happens with the use of the LDR characteristic curve, that on a log-log space is linear, thus allowing the conversion to be made using a linear and angular coefficient.

It is important to notice that as the components slightly varies from one another, some tweaking with the original coefficients were made to get a better linear representation of the data.

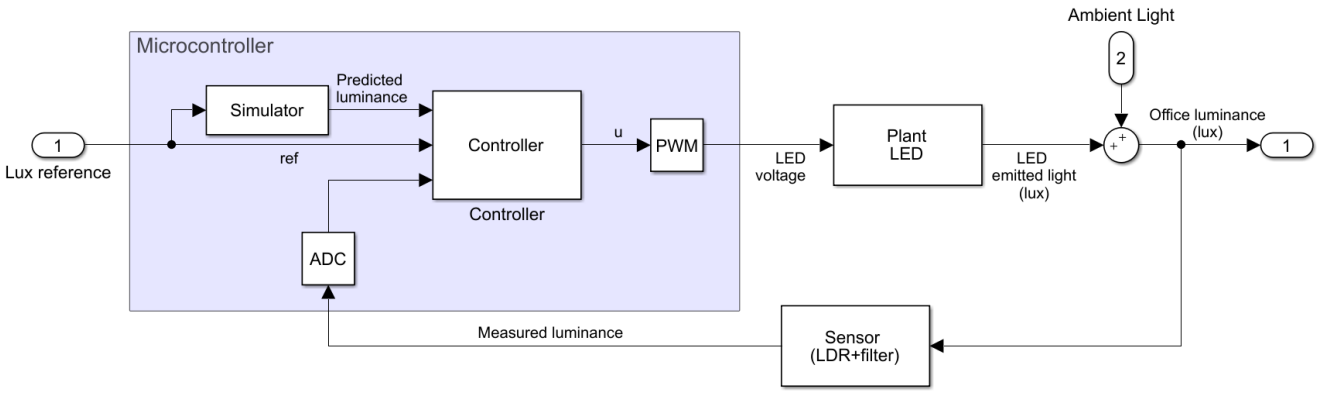


Figure 3: Block diagram of the designed system

3.1.3 Simulator

In order for the controller to work a simulator is needed (this need is explained on section 3.1.4). The idea behind the simulator is to based on previous knowledge of the plant and sensors, take the controller action and predict the measured illuminance. The block diagram of the system is on figure 3.

3.1.3.1 Parameter identification In the simulator class we have functions to identify the parameters of the system. The first one is to identify the steady state response of the Plant to the inputs. As this is verified to be a linear response, a function was to calibrate the gain with a fast and a full mode. In the fast mode, the code assumes that no external light is present on the office, sets the LED to maximum brightness and measure the resulting illuminance. The plant gain is given by:

$$G_{fast} = \frac{L_{max}}{DC_{max}} \tag{1}$$

where G_{fast} is the plant gain calculated on the fast mode, L_{max} is the illuminance measured with the maximum duty cycle, and DC_{max} is the maximum value of duty cycle.

The full mode first measure the base illuminance on the ambient, without the LED on (L0) and then perform a series of measures varying the duty cycle on its full range and stores the duty cycles (DC) and illuminances(L) in matrices. After the measures finished the microcontroler computes the pseudo-inverse of the DC matrix and multiply by the illuminance matrix, resulting in the plant gain (G_{full}) as showed in (2) and (3).

$$\mathbf{DC} \times G_{full} = (\mathbf{L} + L0) \tag{2}$$

$$G_{full} = (\mathbf{DC}^T \times \mathbf{DC})^{-1} \times (\mathbf{L} + L0). \tag{3}$$

There is also a function to test the G value and see if the predicted steady state values are close to the real ones. It uses the same method of stepping the DC and waiting for the voltage to settle, but this time it just prints the expected and measured voltages . Next is necessary to identify the transient dynamics of the sensor. The sensor has a capacitor that filter high frequency noises, but this introduces a transient response to an input that must be accounted for. This transient is modeled by the charge/discharge capacitor equation:

$$\dot{v}\tau(x) = -v + V_{cc}\frac{R_1}{R_1 + R_2(x)} \tag{4}$$

where τ is the time constant, $\dot{v}$ represents the measured sensor voltage, V_{cc} is the supply voltage, R_1 is as given in the electrical schematic (Figure 1) and R_2 is the LDR resistance. All these variables but τ are either know or can be measured in real time, so a function was created to calibrate the τ values.

The *calibrate_tau* function does a series of step inputs on the LED duty cycle and measure the corresponding time constant using the rule that one time constant is the duration taken for a capacitor to increase its voltage by 63% of the difference between its start and steady state voltage. These steps are done starting from 0 and from 100 duty cycle, as in the tests was verified that the time constant is very different for increasing and decreasing voltage. At each step the voltages are stored at a frequency of 10kHz and later a sliding median filter is applied

before the algorithm looks for the time where the voltage reached the 63%. These values of τ are then stored in vectors for later use.

The last calibration needed is for the delay in the microprocessor instruction to the PWM change. This delay (θ) is measured by logging the voltages at the fastest frequency supported by the microcontroller (around 140kHz) without using DMA, and at a given time doing a step on the duty cycle (DC). Then the function waits for a change in the voltage measured at the LDR and measures the time taken between the DC change and the LDR voltage change, again, using a median sliding filter.

3.1.3.2 Simulating For the simulating part was chosen to use a discrete version of (4) so a function was created that receives the last simulated voltage, the current DC, and the time step of simulation and calculate the next predicted voltage in the LDR. This function is handled by the main timer as explained later.

Regarding the τ used to simulate there is a function that based on the current and steady state voltage decides between the τ calculated for an up step or a down step. After this decision the function performs a simple linear interpolation between the corresponding closest τ for the current DC and return the result.

To account for the θ delay a circular buffer was created to store the predicted voltages, and a function retrieves the prediction for the current time from this buffer.

3.1.4 Controller

The controller can be divided in two branches (figure 4), there is a feedfoward branch and a feedback branch. This double control approach was chosen in order to get the fast response with no overshoot from the feedfoward but keeping the disturbance rejection and robustness of the feedback controller. The feedfoward controller is fairly

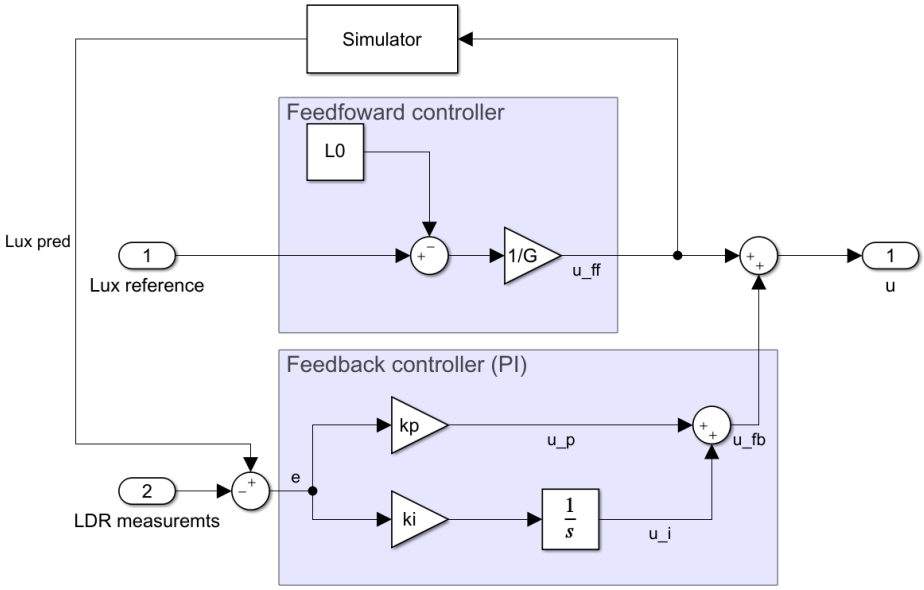


Figure 4: Controller block diagram

easy to develop, it basically is the inverse of the plant, so if you have a perfect model and no disturbances the system basically turns into an unitary gain. In some cases this is note feasible as the plant can be not inversible, or the inverse of the plant requires knowledge of the future states. In our case the plant is just a gain, so it's easy to invert it. To take it one step further the base illuminance on the ambient was take into consideration, resulting in the following equation for the feedfoward:

$$u_{ff} = \frac{ref - L_0}{G} \tag{5}$$

where u_{ff} is the feedfoward control action, and ref is the lux reference (target) of the controller.

The feedback is a little bit trickier. The idea is to use a PI (proportional and integral) controller, but in order to do that is necessary to define the error that will be fed to the PI. Usually this error (e) is defined as the difference between the reference and the sensor output, but in our case this would lead to a big overshoot and maybe instability, depending on the gains of the PI. This is due to the fact that the sensor has a delay caused by

the low pass filter. If the feedback controller took the sensor as ideal, it would see a slow step response from the plant and try to compensate it by increasing the actuation. The problem with that is that in reality the plant change is instantaneous, its the sensor that takes a moment to reach the right value. This is where the simulator comes in, it can simulate a fairly accurate sensor response so it can be used as a reference for the PI controller. Now its possible to compare the simulator output (expected output) with the real one given by the LDR and the low pass, and use the PI controller to eliminate this error, rejecting disturbances.

After this calculation, a simple sum is done between the branches to get the final control action(u).

One problem that arises from the PI controller is the actuator (in our case the LED) saturation. There is a limit on how much light the LED can provide and it cannot absorb ambient light to reduce the measured illuminance. The issue with that is that if the LED reach its limits and the feedback error is not zero, the integrator will continue to increase/decrease the control action trying to zero the error, but the LED cant keep up with the requested action (DC can only be between 0 and 100%) so it will grow indefinitely. When later the system changes its references to a more feasible value or the external disturbances are reduced to a point where the set point can be achieved, the integral term will be built up, resulting in a really slow response while the integral is decreased to a normal value. This combined with a slow plant/sensor can result in a unstable system. To avoid the wind up of the integral the saturation of the integral method was chosen. This checks the control action and if it is outside the LED limits changes the integral term to the closest value that is within LED limits and recompute the sum of branches.

3.1.5 Luminary

The luminary class is basically what brings the other classes together, it is the main object of the system, containing the LED, LDR, controller, and simulator objects. It has basic functions to get values of the objects and functions to calculate some performance metrics.

It calculates three metrics, as defined in the guidelines, the power consumption, the visibility error, and the flicker. As these metrics are defined as an average for a time period rather than a single value, and as the request for their values will be for all the duration of the system functioning, it was chosen to store the sum of their values and the number of steps, so uppon request this sum is divided by the number of steps and the user gets the end result.

3.1.6 Parser

The parser class is responsible for all the serial communication. It implements the protocol for sending commands and for requesting data. It defines the microprocessor as an slave, and whatever is on the other side of the serial port as the master, meaning that after the full initialization it will not send messages unless requested. The possible commands are very similar with the ones defined on the guidelines, with some minor adjustments for better code developing and consistency on argument positioning. They are given by the tables 1, 2a, and 2b.

Table 1: Set commands. <i>refers to the Luminary ID

Command	Client Request	Observation
Duty cycle	"d <i><val>"	<val>is a float expressing duty cycle (0-100%)
Illuminance reference	"r <i><ref><occ>"	<occ>is a bool, 1 for occupied, 0 for unoccupied state <ref>is the reference value
Occupancy status	"o <i><occ>"	<occ>is a bool, 1 for occupied and 0 for unoccupied
Anti windup state	"a <i><aw>"	<aw>is a bool, 1 for anti windup on and 0 for off
Feedfoward state	"w <i><ff>"	<ff>is a bool, 1 for feedfoward on and 0 for off
Feedback state	"b <i><fb>"	<ff>is a bool, 1 for feedback on and 0 for off
Feedback controller gains	"c_g <i><kp><ki>"	<kp>is the proportional gain (float) <ki>is the integral gain (float)
Toggle stream of variable <var>	"s <i><var>"	Can stream multiple variables

Each command has a function that is called by the parser with the respective arguments and set a variable or get a value. The set commands are repplied with an "ack" or the respective error. The history bufffer is a circular buffer for each variable and is used only by the parser. The stream function only sets the variable respective bit on a *int16_t* control variable, and later the main timer prints the variables that have a high bit at each cycle. Regarding the variable code on table 2b, those are the available variables for the stream and history commands.

Table 2: Get and stream commands

(a) Get commands. <i>refers to the Luminary ID

Command	Client Request	Server Response
Duty cycle	"g_d <i>"	"d<i><val>"
Current Illuminance reference	"g_r <i>"	"g_r<i><val>"
Measured Illuminance	"g_l <i>"	"g_l<i><val>"
Occupancy status	"g_o <i>"	"g_o<i><val>"
Anti windup state	"g_a <i>"	"g_a<i><val>"
Feedfoward state	"g_w <i>"	"g_w<i><val>"
Feedback state	"g_b <i>"	"g_b<i><val>"
External Illuminance	"g_x <i>"	"g_x<i><val>"
Power consumption	"g_p <i>"	"g_p<i><val>"
Time since startup	"g_t <i>"	"g_t<i><val>"
Accumulated energy consumption	"g_e <i>"	"g_e<i><val>"
Accumulated Visibility error	"g_v <i>"	"g_v<i><val>"
Accumulated Flicker	"g_f <i>"	"g_f<i><val>"
Last minute buffer off variable <var>	"h <i><var>"	"h <var><i> <val1>, <val2>, ..., <val_n>"

(b) Variable code for history and stream commands

Command	Variable code
Duty cycle	dc
Current Illuminance reference	r
Measured Illuminance	lms
Predicted Illuminance	lp
Feedback error	e
Proportional control action	p
Integral control action	int
Feedfoward control action	uff
Feedback control action	ufb
Control action	u
External Illuminance	xi
Power consumption	pw
Flicker	fl

There is a function in the parser file that is responsible for converting the variable code char array into a *unsigned long* that can be compared by the switch case statement on the stream and history variables.

The command parser is called on the main loop, while the controller works on interrupts, granting the priority of the controller over the communication.

4 Results

4.1 Calibration

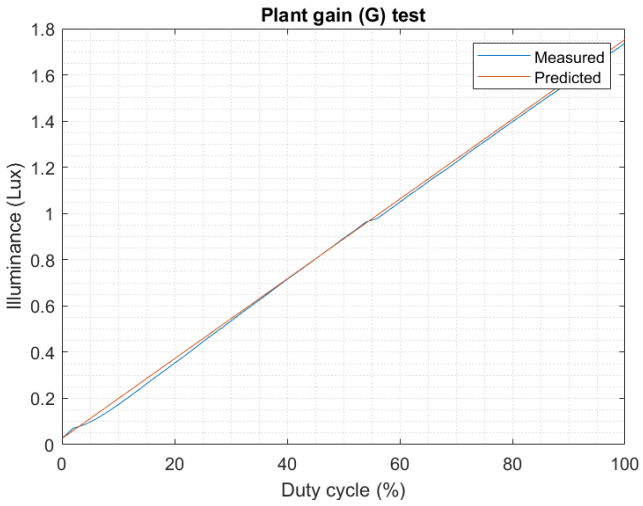
Regarding the plant identification, after the G gain was calibrated by the full mode, a test was performed to see if the expected steady state value was right. It was done by changing the DC of the LED and waiting a few milliseconds for the system to stabilize. After the system is stable a few measures were made and filtered by a median filter. The result is compared with the predicted value on figure 5a.

Its noticeable that the system is not perfectly linear, but the least squares approximation does a good job on find good coefficients. Its important to see also that using the fast approach would cause some errors because the fast approach does not use information about all points, just the last one, and assume the rest is linear coming from the origin.

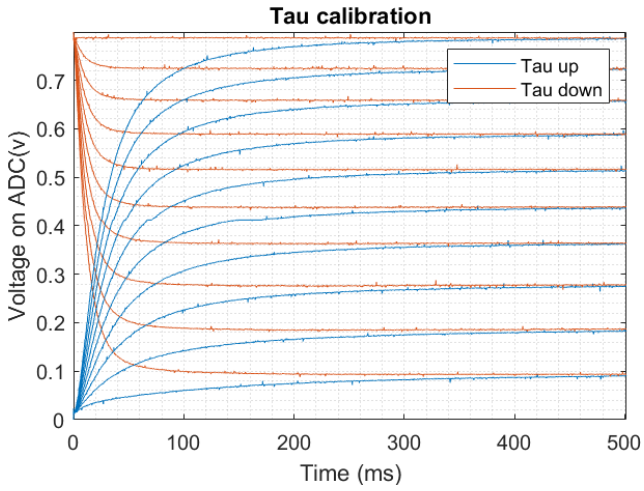
Going forward to the τ calibration, as stated on previous sections, several curves were made coming from 0 and 100 DC. You can see on figure 5b that they have very different time constants. While when the DC comes from 100 the time constant is always close to $13ms$, when the DC comes from 0 its noticeable a big difference, with τ values ranging from 40 to 80ms. Its important to check that whatever the initial condition, the steady state value is the same, as expected.

The last calibration step is the θ delay. One challenge when calculating this value is regarding the noise, as we need the biggest frequency of acquisition possible, its not possible to heavy filter, so the median filter has to work with a small window, otherwise it would add too much delay. For a balanced result, a sliding window of 5 points was chosen.

To ensure the result is with the smaller error possible, the test is configurable to run several times and output an average of results. In figure 6 one of these results is presented. The vertical line represents the moment of change in DC. The system starts to respond around 80µs afterthe change. Now is necessary to account for the filter delay. The acquisition frequency is around 140kHz, so the period of acquisition is close to 7µs, and considering that the



(a) Test of plant gain G



(b) Calibration curves of τ

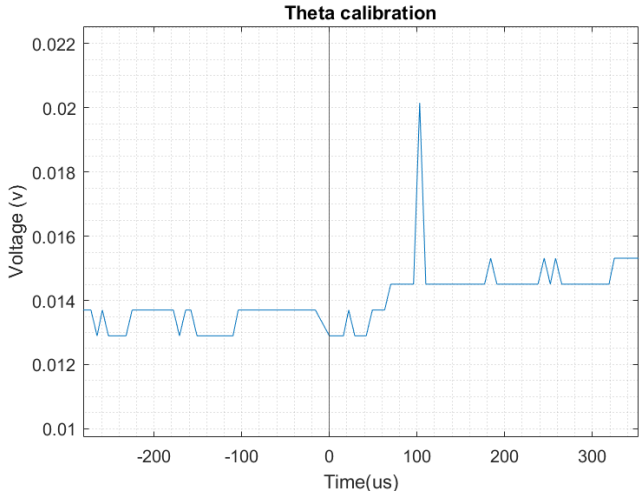


Figure 6: θ calibration

median filter is of 5 samples, assuming an perfect increasing signal, it would take 3 samples to notice any change. This results in a θ value of approximately 60µs.

The guidelines stated that our control loop should have a frequency of 100Hz, this implies a delay of 10ms or 10000µs that is three orders of magnitude bigger than the θ value. As the difference of magnitude is this big, although the functions implemented can account for the θ delay, it was chosen to not use it, for the sake of simplicity.

4.2 Control Tuning

To tune the control first was defined a reference curve (figure 7). Although in an office scenario the reference is not expected to change so fast and to such different values, it is a good curve to evaluate the controller performance as it hast fast and slow changes, with low and high reference values.

After the reference curve was defined, a grid search was performed with the Kp and Ki gains, as the feedfoward branch is not tunable. The results of the grid search were evaluated by five metrics:

- Average Power consumption,
- Average Visibility error,
- Average Flicker,
- Feedback Mean squared error,

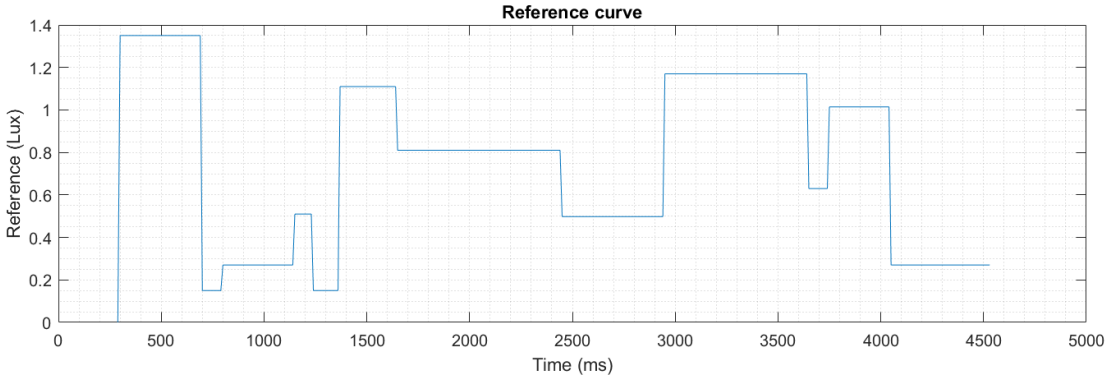


Figure 7: Tunning reference curve

- Linear combination of other metrics.

The fifth metric was defined by the author as a combination of the average power consumption, visibility error, and flicker. The combination aims to get a single value that represents all others. It was defined as:

$$CM = Vis_e + Pwr + Flck \tag{6}$$

where CM is the combined metric, Vis_e is the average visibility error,Pwr is the average power consumption, and $Flck$ is the average flicker. The mean squared error was not included as the other metrics already represents well the goal of the controller.

The grid search was performed with values ranging from 0 to 200 for Kp and from 0 to 900 for Ki. Three of the best results are displayed on figures 8, 9, and 10.

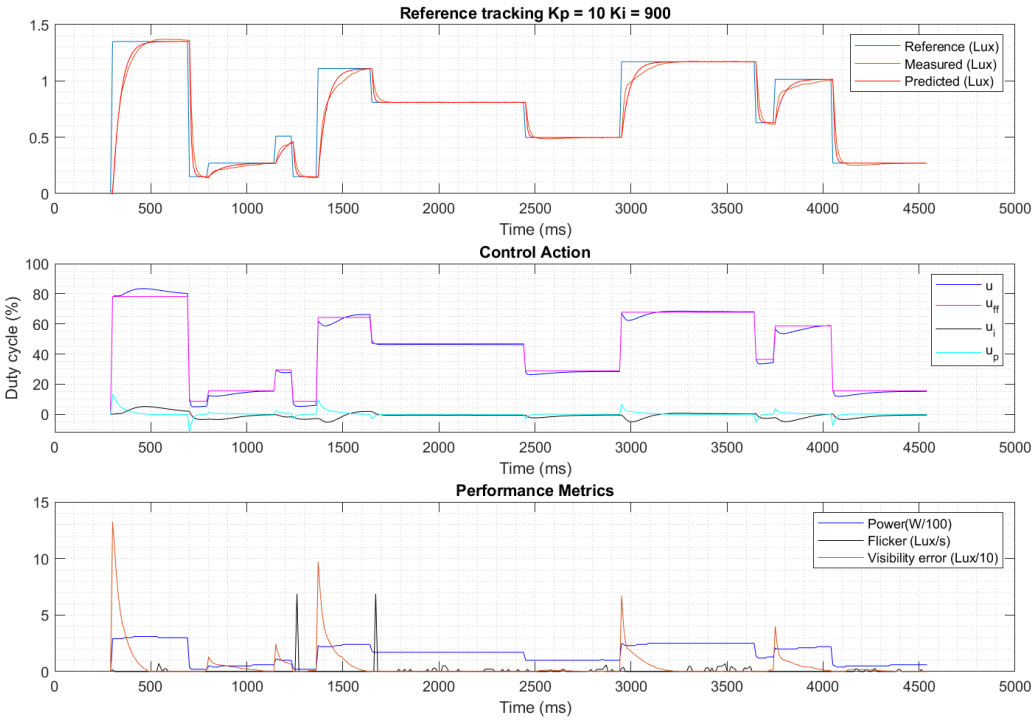


Figure 8: Tunning results with Kp = 10 and Ki = 900.

The performance metrics of these tests are presented on table 3.

As expected, by increasing the gains the system response is faster, at the cost of power consumption increase, but resulting on an smaller visibility error. Also is worth noticing that the predominant factor for flickering is the

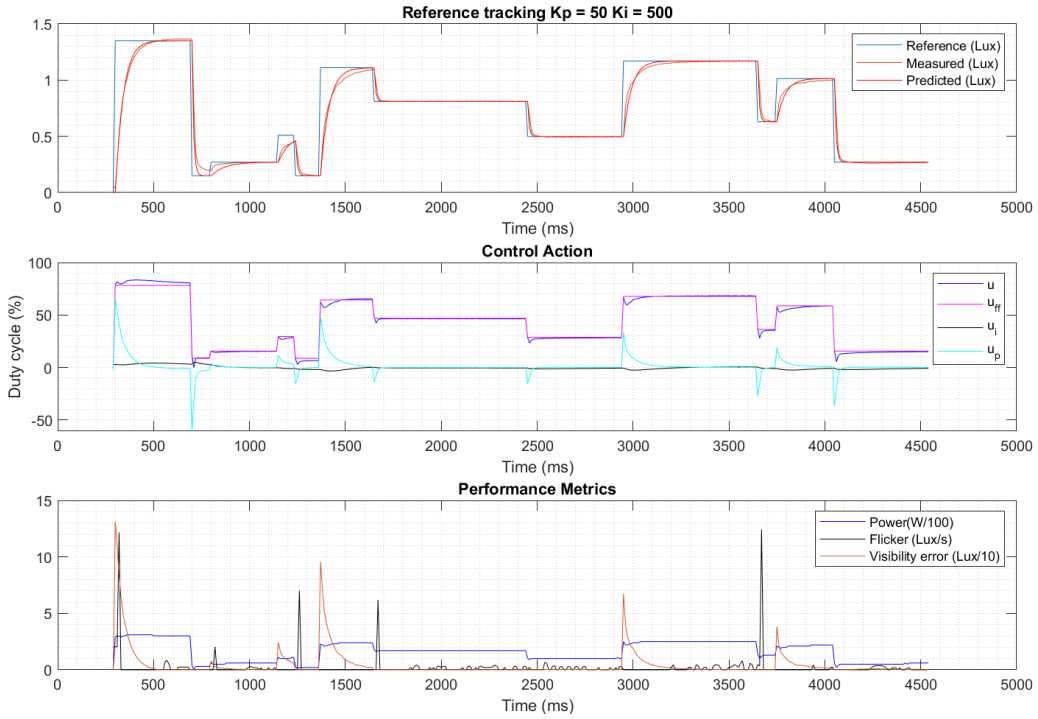


Figure 9: Tunning results with Kp = 50 and Ki = 500.

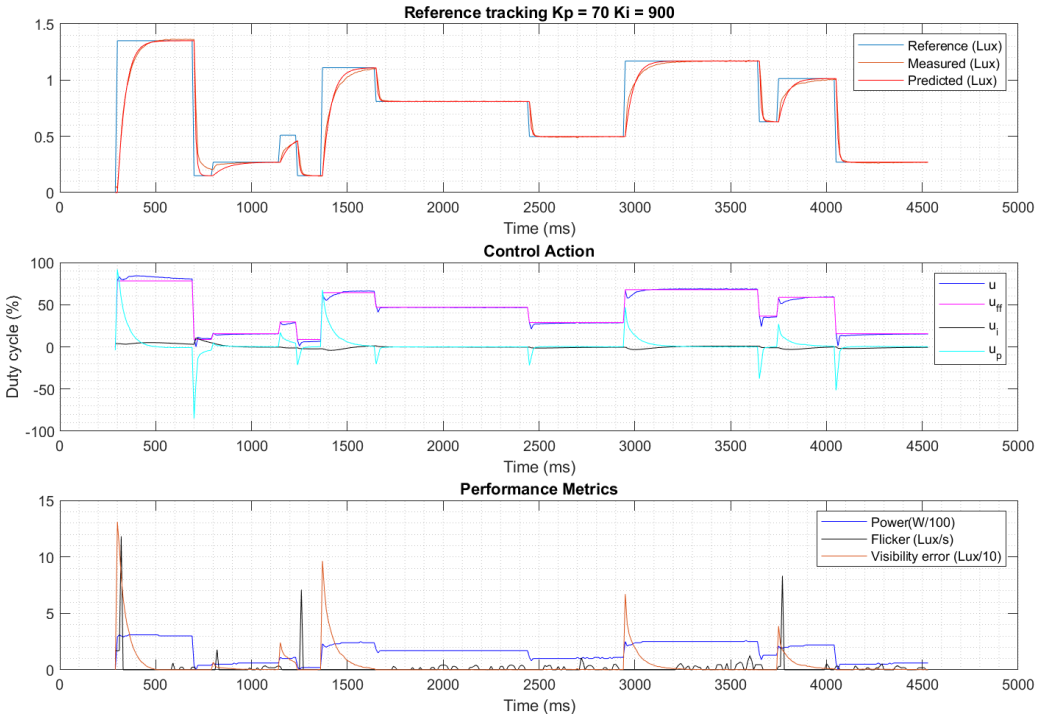


Figure 10: Tunning results with Kp = 70 and Ki = 900.

proportional gain, and not the integral. The values of Kp and Ki were chosen as 50 and 500, as it represents a good balance between the metrics.

With the values defined its a good practice to test the system response with disturbances, so a constant reference was set, and some external disturbances were introduced on the system by another Luminary running the base reference curve control.

Table 3: Performance metrics for part of the grid search

Kp	Ki	Combined	Average Visibility error	Average Power Comsumption	Average Flicker	MSE
10	900	0.13973	0.047417	0.015948	0.076365	0.028988
50	500	0.24459	0.043484	0.016045	0.043484	0.027859
70	900	0.23144	0.042946	0.016134	0.17236	0.028072

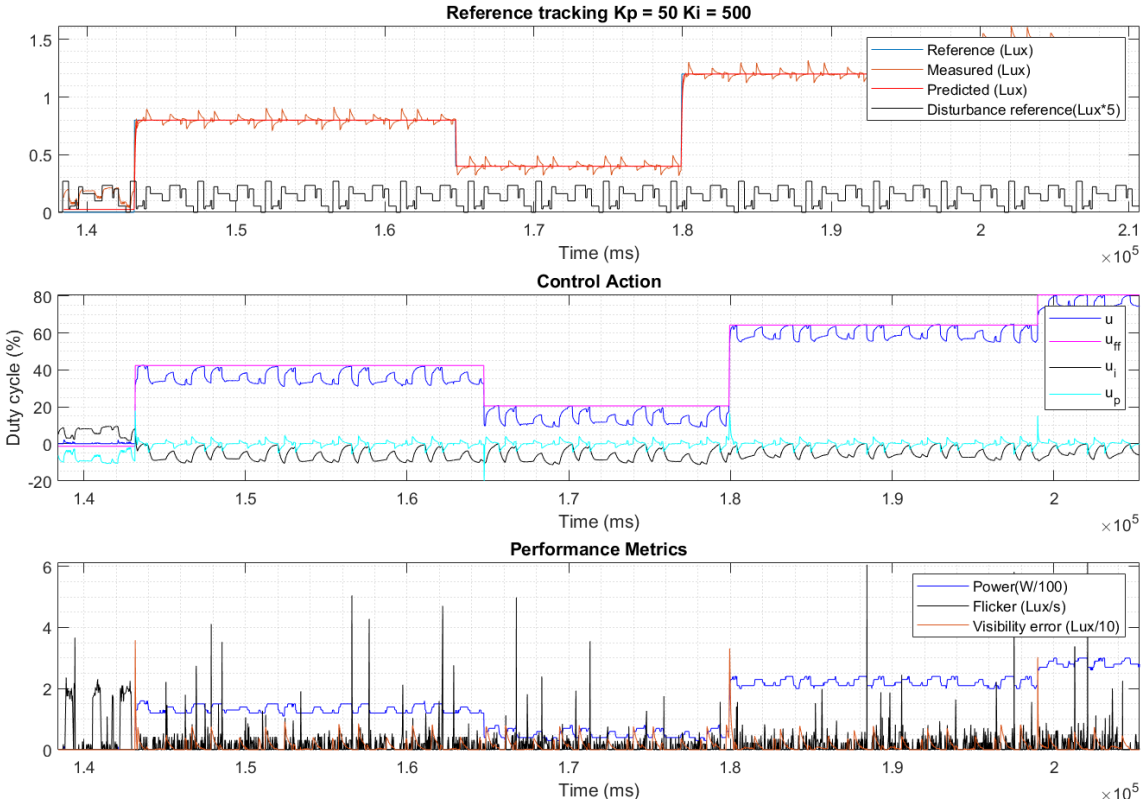


Figure 11: Disturbance test.

Figure 2 shows the result of the disturbance test. Its visible that the second luminary had an impact on the ambient illuminance, but the control managed to counter the disturbances. As expected the disturbances caused some flickering and some errors on the reference tracking when the external illuminance changed, but the controller achieved a good result in countering this.

5 Conclusions

This project of smart lightning has a great potential, although it is still on the phase of individual control, it can track well references, has a fast response, reject disturbances, and can receive instructions by serial.

Even though the code is functional, some performance improvements can be made, like using DMA, multi threading, reducing time spent on computations, and some other minor issues. Also the communication between luminaries need to have some fail safes, as of now it just throws an error and continues.

A good point of the approach used to plant identification is that is fully standalone and automated, so it calibrates itself to new environment at each boot time. Nonetheless this needs to have better handling, as if the ambient is the same as last use, there is no need to re calibrate, and sometimes is necessary to re calibrate without the reboot.

The controller presents a good result, but a major issue is convergence. The current implementation converges to the predicted value by the simulator, meaning its driven by the feedfoward, and the feedback just reject disturbances. The problem arises if the plant gain (G) is not perfect, causing the controller not to converge to the reference, but to a different value. Perhaps adding another feedback loop, but this time between the lux reference and measurements, can improve this but it has to have a switch to turn on only after the transient response is gone.