TÉCNICO
LISBOA

Real-Time Cooperative Decentralized Control of a Smart Office Illumination System

Distributed Real-Time Control Systems Final Project

Daniel de Schiffart **81479**
João Gonçalves **81040**
Francisco Castro **78655**

Distributed Real-Time Control Systems
Master's Degree in Aerospace Engineering
Instituto Superior Técnico

2018/2019

Abstract

In this technical report we will present the design and development of a LED-powered office illumination system that implements distributed real-time control to maintain comfortable levels of illuminance for its users. We will discuss the approaches done to the modelling of each sensor and corresponding LED and its individual control system, the method of communication between all connected sensor-LED pairs (henceforth referred to as desks), the interaction between the lights emitted by each desk and their joint effort to reduce lighting costs and maximize user comfort, and the gathering of relevant data for diagnostics.

Keywords: Smart Office Illumination, Cooperative Distributed Control, Consensus Algorithm, Serial Communications, Arduino, Raspberry Pi, C++.

Contents

1 Introduction

In the last decade, the implementation of LED illumination systems has proven to be a low-energy consumption system in comparison to traditional approaches. As a consequence, this system is also low-cost, which has made possible to apply this kind of illumination devices on cars, offices and homes.

The adaptability of LED illumination systems has been used to improve energy optimization and comply with comfort levels. Distributed control systems have been presented as a possible solution for the problem, by keeping control of the occupation status of spaces and the illuminance values on them, taking into account external illumination levels.

In this project, a simple office-like scenario was considered. The considered approach to achieve the specifications stated above, rely on implementing luminance sensors (LDR) on each desk, as well as light emitting diodes (LED). Communication between each luminaire is also thought to be a key part on achieving a suitable real-time cooperative control system. Moreover, a distributed optimization algorithm (*Consensus* algorithm) is also a key tool on the way to achieve the lowest possible energy consumption.

In this project, the simulation of such a scenario was achieved by considering each *desk* to be composed by a LED, a LDR, a presence sensor (buttons) and communicational elements (Arduino). The entire office will be simulated by joining every *desk* together inside a cardboard box. For simplicity reasons, two desks were considered throughout the project. Energy minimization will be achieved by controlling the dimming level of each LED, so that the illuminance levels on each desk are higher than a certain value (for an occupied *desk*) or lower than another (unoccupied *desk*). These values were defined in the development phase (section 4).

Section 4.4 is devoted to the implementation of an independent control system for each *desk*, whose goal is to set the illuminance level to a reference value, this value measured on its desk, with no cooperation.

In the second part (section 4.6), both desk cooperate, whose main goals are to ensure communication between themselves in order to achieve suitable illuminance optimization levels, and to allow the access of many different kinds of data by clients through a server, to be placed on a Raspberry PI module.

In the remaining sections of the report, the results obtained with the cooperative distributed control system are presented and discussed.

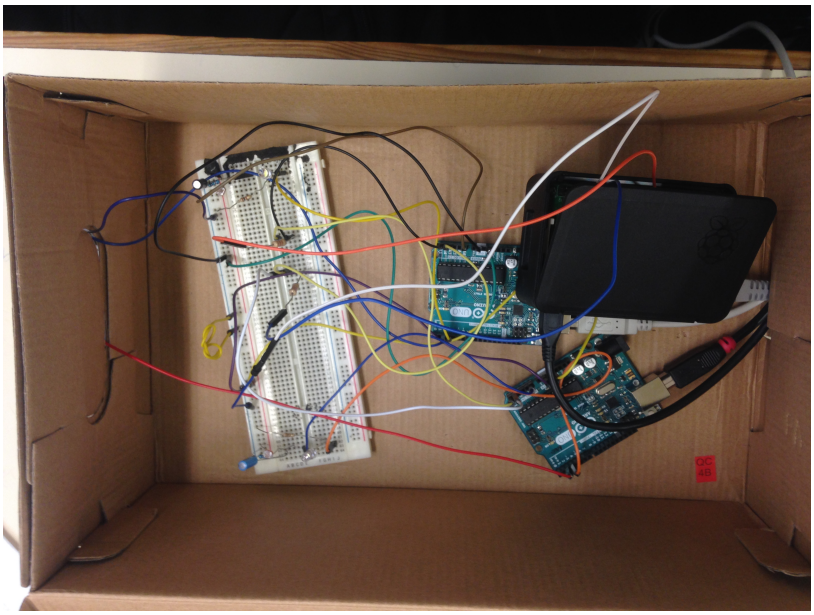


Figure 1: Hardware setup.

2 Background and Concepts

In this section, some basic concepts introduced in classes which were useful for the project implementation are briefly described.

2.1 Distributed Control

A distributed decentralized coordinated control system, with explicit communication is used in on the project. This means there may be multiple controllers, where *every controller makes a decision for it's own behaviour and the resulting system behaviour is the aggregate response* (decentralized). *Each node communicates with the others* by explicit communication, [Ber18a].

A possible application of this solution is the Consensus algorithm, [Ber18b].

2.2 Illuminance models

The illuminance metric used to characterize the dimming level on each *desk* is LUX. One LUX is equal to one $1\,\mathrm{lm\,m^{-2}}$ (lumen per meter square). One lm characterizes the *light produced by a light source emitting* 1 cd *(candela) over* 1 sr *(steradian).* Finally, a sr is the solid angle of a sphere section area, given by the division between the area of that surface and the sphere radius. A cd is equal to $\frac{1}{683}$ W *of green light per* sr *in a given direction*, [Ber18a].

The luminous power incident on a surface is measured in LUX. The luminous flux of light emitted by a source is expressed in lm. The luminous intensity is expressed in cd.

2.3 Hardware concepts

Each Arduino is responsible for processing and transmitting data at each node. The voltage for each LED is generated with a PWM (Pulse Width Modulation) signal. A PWM signal is a sequence of ones and zeros whose duration of each section is defined by a duty cycle (proportional to the analog voltage required). The value read by each LDR sensor is also received by the respective Arduino. Due to this, each Arduino also acts as a discrete microcontroller.

Discretization also adds quantization noise to the system. Moreover, the resolution (n-bit resolution) of a given quantization depends on the levels obtained with that process.

Serial communication between Arduinos (nodes) is ensured by the I2C communication protocol, where each Arduino may be a master or slave. *A master can send data or request data from slaves, and a slave can read data or send data to the master*.

The Raspberry PI module will read from the I2C bus. This bus collects data from the inter-node communications. *The RPI will connect to the I2C bus in read-only mode and collects all messages transmitted in the bus for analysis. The RPI will also implement a server providing an interface for an application client. This application should be able to read the state of any luminaire and access system statistics.*

3 Related Work

This works focuses mainly on the achievement of a stationary state where low-energy consumption doesn't compromise comfort levels, by complying with the required illumination levels.

Similar projects also tried to achieve good levels of energy efficiency for luminaires with a centralized control system [Pan+15]. The brute-force algorithm is also another optimization approach as stated in [MSN17] for a street lighting system.

Inter-node communications and data handling are also very positive features of the system, because communication between nodes eliminates the need to define a central controller. During the simulation, it is also possible to access data, such as the current illuminance at a given *desk* or the current occupancy state.

4 Development

4.1 Illuminance measurement

The luminaire node uses a light-dependant resistor (LDR) to measure illuminance, which is a non-linear component. Its resistance is approximately linear with illuminance in logarithmic units, and manufacturers provide a range of resistance for each illuminance value, as shown in figure 2. The relationship between resistance, R, and illuminance, L, is given by:

$$\log_{10} R = m \log_{10} L + b, \tag{1}$$

with m and b being particular to each LDR, estimated with linear regression. See attached file `calibration.R`.

The resistance in turn is measured on a resistance divisor, as seen in figure 3, and illuminance is computed as:

$$\begin{aligned} R &= \frac{V_{cc} - v}{v / R_{aux}} \\ L &= 10^{\frac{\log_{10} R - b}{m}}, \end{aligned} \tag{2}$$

where V_{cc} is 5 V, R_{aux} is 10 kΩ and v is the measured voltage, which is necessarily in the range [0 – 5]V. The capacitor helps to mitigate noise in the circuit.

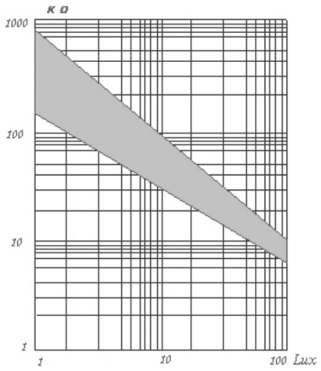


Figure 2: LDR illuminance-resistance characteristic.

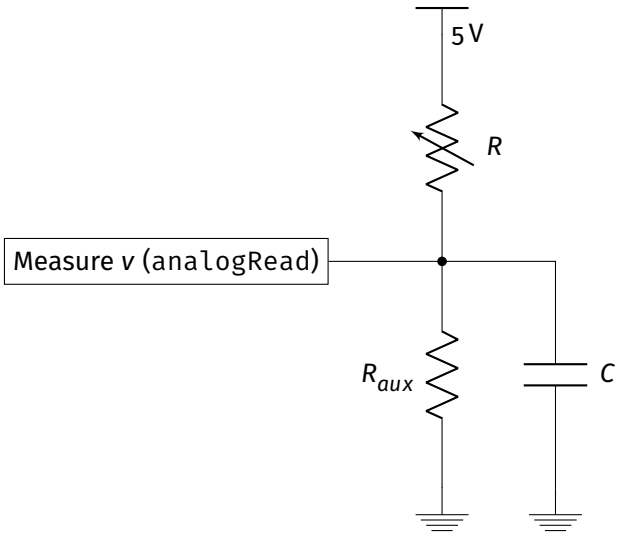


Figure 3: LDR measuring circuit

4.2 LED actuation

The LED is driven by a digital pin with a pulse width modulation (PWM) signal, to emulate an analog output. The frequency of the PWM signal is increased to 980 Hz to avoid the switching to be picked-up by the illuminance sensor, and cause unnecessary noise [eTe17].[1]

4.3 Dynamical system modelling

As discussed in section 2.2, illuminance at any point changes instantly when a light source changes its radiated power. However, in measured illuminance, there is a dynamic response, which is a property of the electric circuit that measures it. In particular, there is a delay, τ, that should be estimated.

For a single luminaire node, there is a light source, the LED, and a sensor, the LDR, with a certain characteristic reflection path. The dynamic model of the measured illuminance to the LED power is assumed to be 1st order linear, as:

$$\frac{L(s)}{P(s)} = \frac{K_0(x)}{1 + s\tau(x)}, \tag{3}$$

where $L(s)$ and $P(s)$ are the Laplace transforms of the measured illuminance and LED power respectively. K_0 and τ are a gain and time constant, a function of x to symbolize that they depend on the environment, through a specif reflection path setup.

To be noted that the model in equation 12 is BIBO-stable.

Experimentally, the LED is driven by step signals and the illuminance is measured, in order to estimate τ and K_0 for a common operation condition (box closed, see figure 4). The value of K_0 was used in earlier stages to drive the feed-forward controller, although a better methodology to estimate this gain for a multi-node configuration is described in section 4.7. The value of τ is measured to be 20 ms, and will influence the choice of sampling time. It is also verified that $\tau \approx RC$ for this circuit. See attached file `model_estimation.R`.

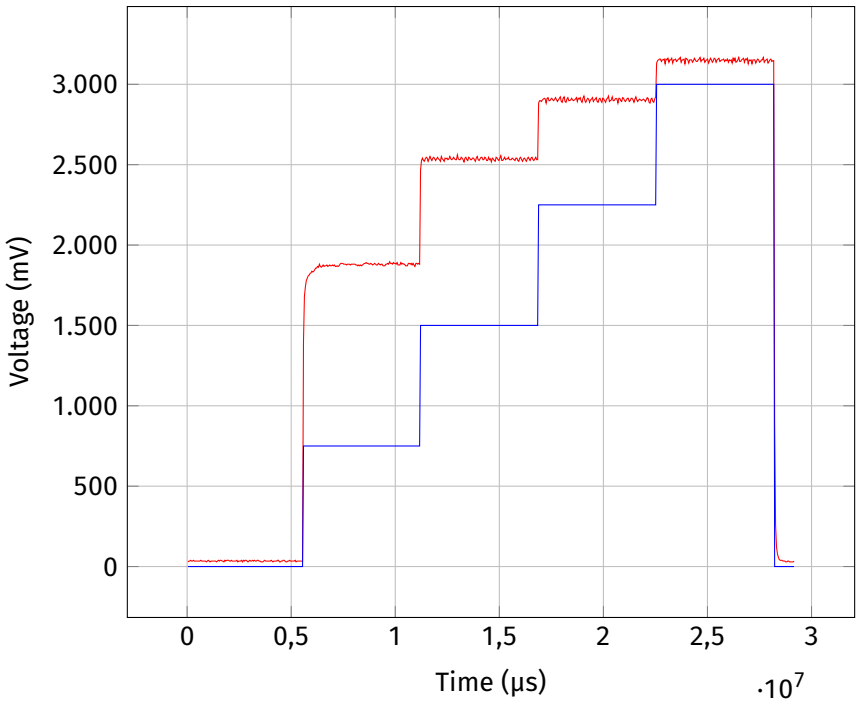


Figure 4: Measured illuminance (as voltage v) for step actuation in the LED, box closed.

[1] This was not implemented at the time of the first demo.

4.4 Individual luminaire controller

The purpose of the controller is to actuate on the system in order to have the illuminance L to follow a reference value L_{ref}. In this section, L represents the physical illuminance, i.e. the true value, while $\tilde{L}$ represents measured illuminance. The control signal is the LED duty cycle, represented here as u.

The controller is implemented with a feedback component and a feed-forward component, described separately.

4.4.1 Feed-forward control

The feed-forward control aims to set a constant actuation by predicting the necessary value so that the output converges to the reference value. It has the advantage of not requiring sensors and being fast, and in this problem it is used to speed-up the response of the feedback-only controlled system. For usage, it requires that the system is open-loop stable, and also requires precise modelling, and its main disadvantage is that it cannot reject disturbances, or completely remove steady-state error.

With the available model in equation 12, it is clear that a feed-forward controller can be used, by having simply:

$$C_{ff}(s) \equiv \frac{u(s)}{L_{ref}} = \frac{1}{K_0}, \tag{4}$$

so that the system response is:

$$\tilde{L}(s) = \frac{L_{ref}}{1 + s\tau}, \tag{5}$$

which settles to the steady-state value L_{ref}.

4.4.2 Feedback control

A feedback controller is required to eliminate steady-state error and reject disturbances. The usage, however, introduces delay and subjects the system to noise from the sensor. It is also important to verify that stability is maintained.

For this problem, the used controller is a proportional-integrative controller (PI).

The proportional term penalizes error, e, the difference between the measured value and the reference, getting the actuation closer to the required value.

The integral term integrates the error in order to remove it in steady-state, however it must always be combined with proportional control as it slows the system and deteriorates stability.

A derivative term is skipped, as it is amplifies high-frequency noise, especially after dicretization. Its purpose is to penalize oscillations and stabilize the response faster, but the plant is not prone to oscillations itself, and the feed-forward controller helps speed-up the response.

Additionally, the PI algorithm implemented is a *velocity algorithm*, as described in [Ben94], since the actuation in equilibrium is not 0 for this system. It consists in calculating the velocity of change in control, $\dot{u}$ with a PID controller, in this case just PI, such as:

$$C_{fb}(s) \equiv \frac{U(s)}{E(s)} = \frac{1}{s} K_P \left(1 + \frac{K_I}{s}\right), \tag{6}$$

with $E(s) \equiv L_{des}(s) - \tilde{L}(s)$, and K_I, K_P gains to be chosen. L_{des} is the "desired illuminance", covered next.

The choice of gains is done experimentally, based on the Ziegler Nichols frequency response method. In short, pure proportional gain is applied until the system starts to oscillate, which gives us the point on the Nyquist plot where the phase lag is π. Then, the gains take some values based on the applied gain and observed frequency of oscillations.

As discussed in [AM10], this method lacks robustness since too little information is used about the system. Therefore, the gains are manually tuned around the values given by this method.

4.4.3 Feed-forward and feedback control

The controllers are joined as shown in figure 5. Normally, the feedback controller receives a reference value, like L_{ref}, however, with the usage of a feed-forward term, we already expect a certain response, and from the feedback controller it is only needed that the system loosely follows that first order response to the reference value. Therefore, a *predictor* is used to provide the feedback controller with a desired illuminance at each time, L_{des}.

The value of L_{des} can be predicted by converting equation 5 to the time domain, considering that the reference value in this problem always varies in steps. Let L_{ref} be the current reference and L_{prev} the previous (constant) reference, and t_0 the instant when the reference changes. Then:

$$L_{des}(t) = L_{ref} - \left(L_{ref} - L_{prev}\right) \exp\left(-\frac{t - t_0}{RC}\right). \tag{7}$$

Obviously, $L_{des} \to L_{ref}$, as is expected in steady state.

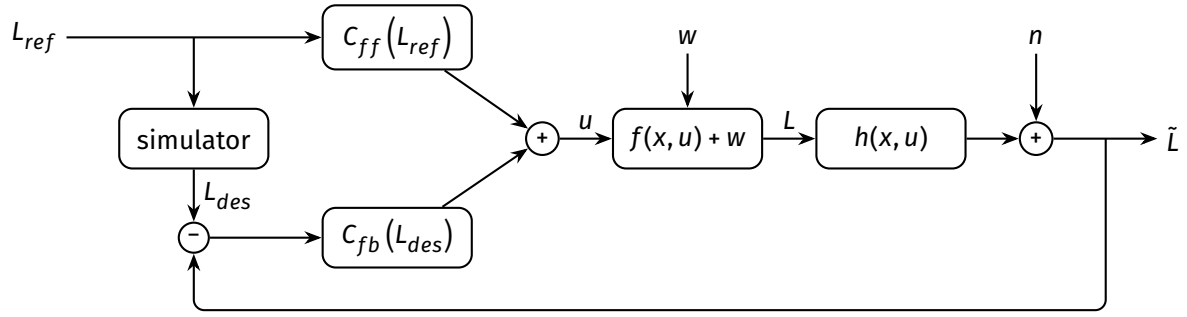


Figure 5: Feed-forward + feedback architecture. w is disturbances, n is sensor noise.

4.4.4 Discretization

The controller must be discretized in order to be implemented in the Arduino board. A fixed sampling time of $h = 5\,\text{ms}$ is chosen, as it proved difficult to execute the calculations with smaller periods, although it should be noted that it is still large when compared with the characteristic time constant.

The integral controller is discretized using the Tustin transformation, which maps stable continuous-time poles to the inside of the unit circle in discrete-time, preserving stability. Additionally, it places a zero at $z = -1$ which attenuates noise, and suffers no delay. The result is:

$$i[k] = i[k-1] + \frac{h}{2}\left(e[k] + e[k-1]\right). \tag{8}$$

The other integral which converts $\dot{u}$ to u is implemented with a backward difference:

$$u[k] = u[k-1] + h\left(K_P(1 + K_I i[k])\right). \tag{9}$$

4.4.5 Implementation

For implementation in the luminairs, some additions are made:

1. A 3-sample median filter is added to the sensor output, taking 3 measurements every control loop. This reduces sensor noise even further at a low computational cost.

2. An error deadzone is added, to prevent flicker when the illuminance is still close to the reference value. The digital sensor readings only allows certain values of illuminance to be possible, so a small error is unavoidable. The deadzone behaves as:

$$e = \begin{cases} e - \epsilon, & \text{if } e > \epsilon \\ 0, & \text{if } -\epsilon < e < \epsilon \\ e + \epsilon & \text{if } e < -\epsilon, \end{cases} \tag{10}$$

   where ϵ is set to a value in LUX corresponding to the resolution interval of this signal.

3. An anti-windup mechanism is added, by cancelling integration. This prevents windup when the actuator saturates, and greatly improves performance, for example when the box is opened and closed.

4. Branching is kept to a minimum, and only important operations are done before the for the current actuation is set. In particular, writing to serial monitor and anti-windup are done after `analogWrite`. This minimizes the computation delay and also jitter.

5. Sampling time is measured with an interrupt on `Timer 1`, which raises a flag to enable control on the main problem loop. This allows for very precise scheduling of the control procedure, while not ignoring other possible interrupts.

The final controller is present in annexed file `controller.h` as a class method. Algorithm 1 describes the controller. In this, d is a *reference dimming*, obtained from the distributed algorithm as described in section 4.6.

Algorithm 1 PI + feed-forward control algorithm, called by an interrupt with period 5 ms.

Require: L_{ref}, d

$s \leftarrow$ MEDIAN3(*3 measurements from sensor*)
$L \leftarrow$ COMPUTE_LUX(s)
$ff \leftarrow d$
$L_{des} \leftarrow$ SIMULATOR
$e \leftarrow$ DEADZONE($L_{des} - L$)
$i \leftarrow i + \frac{h}{2}(e + e_{prev})$
$u \leftarrow u + h \cdot K_P(1 + K_I i)$
WRITE($u + ff$) ▯ Actuation
if $u + ff$ is saturated **then** ▯ Anti-windup
    $i \leftarrow i - \frac{h}{2}(e + e_{prev})$
    wrap $u + ff$ to the actuation bounds
end if
$e_{prev} \leftarrow e$

4.5 Inter-node communications

Arduino nodes communicate via I2C bus. The communication protocol consists of a fixed message size, with a control byte for defining the contents of the data. From `comms.h`:

```
19 enum MsgCode : uint8_t{
20
21     calibration_request,
22     data,
23     cont,
24     acknowledge,
25     consensus_data,
26     consensus_stop,
27     consensus_new,
28     sampling_time_data,
29     none,
30 };
```

```
35 typedef struct msg{
36     MsgCode code;    //!< Message code
37     uint8_t address; //!< Sender's I2C address
38     uint8_t aux1;    //!< field for data
```

```
39      uint8_t aux2;    //!< field for data
40      float value[3];  //!< field for data
41
42  }message_t;
```

The message is fixed to 16 bytes which is the maximum size usable by the PigPio library on the Raspberry Pi. All messages are sent as broadcasts, some are for logging purposes while others are used by the distributed control algorithm and calibration procedures. The purpose of the message is identified by its code, and the nodes are permanently listening for incoming messages. The code also identifies what variables are sent on the data fields.

Physically, the bus is a line between the Arduinos' SDA and SCL pins, which operates at logical levels 0 V - *LOW* and 5 V - *HIGH*. The Raspberry, however, has logical levels 0 V - *LOW* and 3.3 V - *HIGH*, therefore the connection is made with resistors to provide appropriate voltage drop, as shown in figure 6.

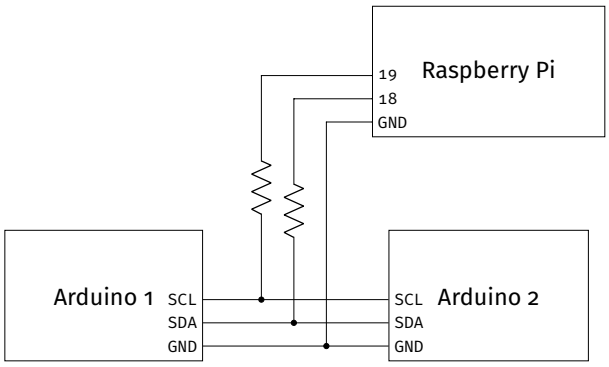


Figure 6: Arduino and Raspberry Pi setup.

4.6 Distributed control

The distributed control system was based on the Consensus Algorithm as stated in [Ber18b].

This optimization approach can be applied for n nodes. However, in this project, only two nodes are used, with indexes given by $i = 1, 2$. This distribution control system relies on two main parts: inter-node communication for exchange of information and optimization of a cost-function.

First of all, each node saves the local illuminance lower bound, L_i, the local external illuminance influence, o_i, the local cost coefficient, c_i, the local actuator bounds (to be 0 and 100 in this case), the coupling gains from the other luminaire to itself, k_{ji} and a global optimization parameter, ρ.

Then, iterations start in order to optimize the current state of the system so that the sum of the cost functions (equation 11) from both nodes is on its lowest possible value.

$$f_i(\mathbf{d}) = \begin{cases} \mathbf{c}_i^T \mathbf{d} & \text{if} 0 \leq d_i \leq 100 \text{ and } \sum_{j=1}^{N} k_{ij} d_j \geq L_i - o_i \\ +\infty & \text{otherwise} \end{cases} \tag{11}$$

with

$$\mathbf{d} = [d_1 \ \dots \ d_N]^T$$
$$\mathbf{c}_i = [0 \ \dots \ c_i \ \dots \ 0]^T$$

The consensus algorithm distributes the computation over the nodes. Each node holds a copy, $\mathbf{d}_i$ of the global variable $\mathbf{d}$ at every iteration.

The performed iterations focus on updating the dimming for each node and communicating them to every node. The average dimming of all nodes, $\mathbf{d}_{av}$, and a local variable of Lagrange multipliers, $\mathbf{y}_i$ are also updated, according to the equations stated in [Ber18b]. At each iteration, a feasibility check of the obtained dimmings is also performed. The dimming to be taken at each iteration is given, at a first stage by the solution in the interior. If it is not feasible, the solutions in the border are evaluated and the minimum feasible solution is taken.

4.6.1 Implementation details

Implementation was made for a 2-node case only. For every restart, $\mathbf{d}_i$, $\mathbf{d}_{av}$ and $\mathbf{y}$ are set to $\mathbf{0}$. A node computes its proposal for $\mathbf{d}$, then sends that via I2C with the message code `consensus_new`. When it is not the first iteration, the message code is `consensus_data`.

After sending a copy of $\mathbf{d}$, the node does not perform another iteration until it receives the peer's copy. With this, the message protocol is asynchronous, not request-response driven, but the alternation in messages is guaranteed for the algorithm to succeed.

When the averaging of of $\mathbf{d}$ does not change the previous values by more than 0.5 %, the algorithm is said to have reached its solution, optimal by definition. Then, the iterations are stopped, with the message code `consensus_stop`.

Only after this happens are the references updated. While the consensus algorithm is being executed, the older references are still in place, and a new session of the consensus is started every time a lower bound is changed (the desk occupancy state changed).

On the Arduino code, the consensus is included within `Class Controller`, and the communication functions in the module `comms.h`.

The outputs of the consensus algorithm are dimmings, $\mathbf{d}_{sol}$, but the controller accepts a reference in LUX. This reference is used computing $\mathbf{l}$ with equation 12, discussed in the next section. The value of $\mathbf{d}_{sol}$ are still used as feed-forward gains.

The result given by this implementation was tested against that which was given by the Matlab script provided by the course staff, by hard-coding all conditions the as same.

4.7 Model calibration

The illuminance at each *desk* follows the model:

$$L = K\mathbf{d} + \mathbf{o}$$
$$\begin{bmatrix} l_1 \\ l_2 \end{bmatrix} = \begin{bmatrix} k_{11} & k_{12} \\ k_{21} & k_{22} \end{bmatrix} \begin{bmatrix} d_1 \\ d_2 \end{bmatrix} + \begin{bmatrix} o_1 \\ o_2 \end{bmatrix} \tag{12}$$

where l_i, d_i and o_i are the illuminance, LED power (in percentage) and background illuminance at desk i, respectively. The purpose of the calibration is to find the values for k_{ij} and o_i for a given office layout.

The calibration is performed by turning the luminairs synchronously and taking measurements. One node takes initiative and sends a message to trigger the process. First, both LEDs are switched off and o_i is measured. Then, node 2 turns on its LED, measures the resulting illuminance and computes k_{22}, and sends the illuminance to node 1. Receiving this value, node 1 can also measure its illuminance and compute k_{21}. The operation inverts and k_{12} and k_{11} are determined. In the end, nodes share the results.

4.7.1 Implementation details

The previous procedure is implemented as its own module, with a special communication protocol. The flow chart in figure 11 illustrates messages sent and summarizes the finite-state machine. Between changing the LED state and taking measurements, the nodes must sleep, to guarantee that the measurement is taken in steady state. For this, they were set to sleep for a length of time 10 times the value of the system time constant, 10 × 20 ms = 0.2 s.

It also should be noted that internally, the nodes are not distinguishable as 1 and 2. Each considers itself node 1 and the peer as 2, so the representation of the coupling matrix K is transposed by both diagonals, and identically o_1 and o_2 are swapped.

4.8 Data server

The objective of the data server is to receive data from each desk and store it however it sees fit. It is also meant to provide this and other relevant data to other client programs that connect to it.

The data server was implemented in the provided Raspberry Pi computer and communicates with the Arduinos in much the same way as the Arduinos communicate among themselves, via I2C. Each Arduino sends a message to the implemented data server each sampling period with the relevant information about its current state, which the data server processes and stores accordingly.

4.8.1 Arduino communication

The data stored by the data server was balanced as to contain all the data that satisfies the project's client requirements but also to allow the message transmitted to be as short as possible. For reference, the client requirements for any desk i can be found in table 1.

Single-sample variables	**Cumulative variables (since last restart)**
Illuminance	Time
Duty cycle	Node energy spent
Occupancy state	Energy spent by all nodes
Lower bound	Node comfort error
Control reference	Comfort error for all nodes
Background illuminance	Node flicker
Power consumption	Node flicker for all nodes
Power consumption for all nodes	

Table 1: Data requested by any connected clients.

The server is meant to be able to retrieve this data at any moment from the Arduino and be able to provide it to the client upon request. It might be a bit too much to transmit from the Arduino to the Raspberry Pi server every sample; however, a lot of the data can be deduced from other values and, especially the cumulative variables, deduced from values stored in the server's own database.

We applied this reasoning by storing the initial value for each cumulative variable and adding to it for every received sample. By also adding the values for all desks stored and making the calculations for accumulation and the remaining variables within the server, we alleviated a lot of data load from the desk-server communication while still making sure all the request data is available to the client. We reduced the variables to be received by the server to the ones found in table 2.

Variable
Illuminance
Duty cycle
Occupancy state
Lower bound
Control reference
Instantaneous power consumption

Table 2: Final variables transmitted by the Arduino to the server at each sampling time.

As currently defined, each Arduino transmits messages using a format described in section 4.5. From that very section we can see `enum MsgCode` which describes the different types of messages

transmitted, and only a select group of these formats can be accepted by the data server. The main format, `sampling_time_data`, contains the data included in table 2, and is formatted according to table 3.

Variable	Definition	Size (bytes)
`code`	`sampling_time_data`	1
`address`	Arduino's I2C address	1
`aux1`	Dimming value (resolution of 1%)	1
`aux2`	Lower bound value	1
`value[0]`	Illuminance value	4
`value[1]`	Power consumption	4
`value[2]`	Target illuminance value	4

Table 3: Formatting of the Arduino `sampling_time_data` message.

The background illuminances are the most notable missing feature when comparing tables 2 and 3 with table 1. This particular variable was left out of the message by considering it constant after each calibration procedure, and therefore does not need to be resent for every sample. Again referring to the code from page 8, it makes use of the `data` message type transmitted by one of the Arduinos to set the value of the background illuminance for the program's runtime.

A final feature from the server's one-sided communication with the desk Arduinos is the auto-reset function, which detects a calibration procedure and resets the program. This resets the database and readies the program for the start of new time period of data acquisition. This detection is done using the `calibration_request` data format analogous to the ones used previously, which starts the reset procedure.

4.8.2 Data storage

The data received from the desk Arduinos is processed by the server upon reception and stored, divided in the program's data segment and heap. The stored data format can be seen in the `desk_t` struct of the `data_structures.hpp` file:

```
4  typedef struct desk{
5      double illuminance; // Current illuminance in LUX
6      double duty_cycle; // Current LED applied duty_cycle, from 0 to 1
7      int occupancy; // ocupancy state, 0-empty 1-occupied
8      double illuminance_lb; // Current illuminance lower bound
9      double illuminance_bg; // Background measured at the last calibration
10     double illuminance_ref; // Illuminance control reference
11     double power; // Instantaneous power consumption
12     uint32_t time_acc; // Timestamp of the last restart
13     double energy_acc; // Accumulated energy
14     double comfort_error_acc; // Accumulated comfort error
15     double comfort_flicker_acc; // Accumulated flicker error
16
17     uint32_t comf_err_samples; // Samples where there was confort error
18     uint32_t samples;
19     double l_prev; // Illuminance in the i-1 previous sample
20     double l_pprev; // Illuminance in the i-2 previous sample
21 } desk_t;
```

and in `main.cpp`:

```
41 const int desk_count = 2;
42 desk_t desks[desk_count]; //!< Data structure to store last sample desk info
```

```
43 std::pair< std::deque< std::pair<float, float > >,
44         std::deque<std::pair<float, float> > > last_minute_buffer;
45         /*!< Deque to hold last-minute buffer of values for dimming and LUX */
```

This data is stored in a desk array, where each entry represents a desk connected to the server. There is a `deque`, which is internally a doubly-linked list, used as a FIFO queue, to hold data for one minute's worth of illuminance and dimming values.

4.8.3 Client I/O

To integrate client applications, an I/O (input/output) interface was developed for an expected client application format. This interface was also developed so that it could handle multiple clients at the same time.

The initial approach to developing this I/O interface was to begin by considering a single client, and further on in the development stage expand this functionality to an arbitrary *n* number of clients. The method of communication used were TCP sockets, and all the client communication was done asynchronously. This means that we can send the request data at any time, which will correspond to the last received data for any desk. This approach allows the server to handle more client requests than its synchronous counterpart (as often mentioned in the C10k problem). This aspect of the server data processing does require careful memory management to avoid sending data while it is being received from the Arduinos. This topic will be covered in section 4.8.4.

The server handles each client connection request by starting a new `Session` class instance, and using it to listen and reply to that specific client's requests. Each of these type of objects uses the `boost asio` resources to create threads and perform most of the synchronization between threads.

4.8.4 Memory management

The consequence of making use of threads for the handling of multiple clients means the need for a clear definition of how each thread access the server's memory and the prevention of conflicts in this access (*i.e.* a client requesting data as it is being received from one of the Arduinos).

This memory management was done using shared memory and `mutex` (mutual exclusion) objects to prevent the aforementioned conflicts. This allows each thread to lock the segment of memory that it is using and unlock it after it is done, allowing the next thread in line to access that very same information.

These implementations were mainly done to prevent any client handling thread from accessing any desk's data if it is being written by the server with newly received data from a desk's Arduino. As clients only read data from the server, their access to the data is relatively unrestricted as reading data poses little threat to the validity of the stored data.

4.8.5 Server classes

The server program works using two classes: the `Server` class, which accepts client connections, and holds the references to the IO service, and the `Session` class, which is responsible for handling all of one client's requests.

`Server` The server class only stores the interface for the usage of the IO service provided by the boost library. It listens on a port for client connections and launches a handler.

`Session` The Session class is responsible for serving clients. Each instance serves one and only one client. All the requests and responses are asynchronous, and the the object is destroyed automatically when connection is lost or closed. For this purpose, this class inherits from `enable_shared_from_this`, introduced in C++11, so that it can create shared pointers to itself and this way manage its lifetime.

One limitation is the inability to only handle one client request at a time (there is no queue of requests in case many are sent in a short time), which is an issue that could be improved. Another

aspect that is not ideal, is the usage of UNIX timestamps as timekeeping all throughout the program, which could be done in a more straightforward way with the chrono library.

4.9 Data collection

Data collection is done via the Pigpio library, running on the only explicitly created thread. Every time a sample is received, the mutex is locked, the data is written, and the mutex is unlocked again, allowing clients client handlers to access the data. Also here, a broadcast is done on a condition variable to wake up threads waiting for data used in streaming.

5 Experiments

In this section, experiments are designed to evaluate the performance of the system operating in different conditions. The results are presented in section 6.

Experiment 1 – PWM analysis

This experiment consists of holding a constant duty cycle on a LED and measuring the output for different PWM frequencies. The variance of the measured noise is computed.

Experiment 2 – Benchmarking

In this experiment, the `micros()` function is used to time critical sections of code – computation delay, communication times and jitter are characterized.

Experiment 3 – Local controller

The performance of the controller is accessed in this experiment. In particular, the step response of the system is measured, for the feedback only controller, the feed-forward only controller, and the joint controller. From the step response, the damping coefficient, rise time and overshoot are computed.

With the joint controller, the performance of the simulator is also measured, by comparing its output value with the actual measured illuminance.

Experiment 4 – Anti-windup

This is an experiment designed to test the closed-loop system in the presence of heavy disturbances, such that the actuation is likely to saturate.

The experiment starts by opening the box to intense external light, such that the background illuminance alone is higher than the reference value. This causes the computed actuation duty cycle to be lower than 0, but the illuminance doesn't decrease any further, which can cause the integral term to build up. The box is closed and the system must regain its original reference.

The experiment is repeated for the designed anti-windup scheme and for no anti-windup scheme implemented.

Experiment 5 – Distributed controller

On this experiment, some performance metrics are compared, for the the local and distributed controller modes. The most relevant one being total energy consumption, defined by:

$$E = \int \left(\sum_{i=1}^{N} \alpha c_i d_i \right) dt, \tag{13}$$

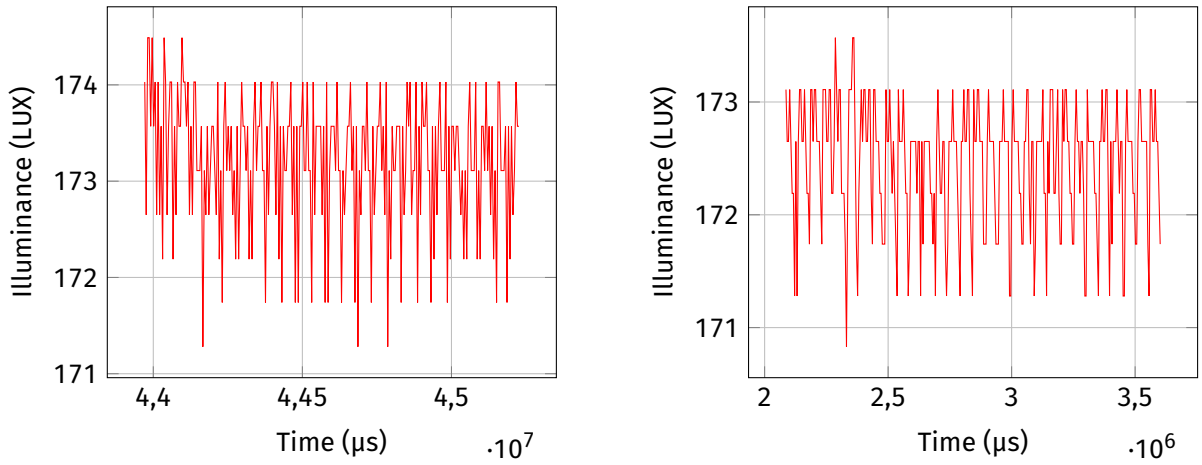
where c_i is the local cost for node i, and α is an arbitrary constant to convert $c_i d_i$ into units of power. Note that $d \in [0 - 1]$, so the metric is non-negative.

The experiment consists of setting the local costs for node 1 to be 5× that of node 2, and observing the illuminance at node 2 for some changes in the lower bound. The expected behaviour is for node 2 to increase its reference illuminance in order to *help* node 1, minimizing global energy consumption. On the same conditions, some metrics including global energy consumption are recorded, each after 10 s from a restart.

6 Discussion and results

Experiment 1

The signals are shown in figure 7.



(a) Signal for a PWM frequency of 490 Hz (the default.)

(b) Signal for a PWM frequency of 980 Hz (the used value.)

Figure 7: Results for experiment 1.

The noise variance is 0.4861 LUX^2 for the signal in 7a, and 0.2952 LUX^2 for the signal in 7b. No improvements were made by further increasing the PWM frequency, and the noise appears identical to that of a natural light source.

Experiment 2

The results are listed in table 4. With respect to jitter, the most important factors are the beginning of the control cycle and the computational time taken to compute a new actuation value. The former is evidently very precise, the variance is residual, as it is signaled by an interrupt. The computation time however is considerable, there is a delay between measurements are taken and actuation is ready. What is more important is for this delay to be approximately constant, so that the actuation is made in regular periods of time – no jitter. The evidence for that is the low variance of this delay, which was achieved by reducing branching.

Serial communications were only allowed for debugging purposes, so its effect is irrelevant. However, it is an accommodable cost for streaming a variable to the serial monitor, for example.

I2C communications are the biggest constraint to the node performance. Their speed is 100kbps, which for 18 bytes (16 bytes with address and acknowledge) takes theoretically 1.44 ms, a value close to the measured one. This has an important consequence – as the sampling time is 5 ms, and for every consensus iteration each node has to exchange data, then only one iteration can be carried out per sampling time.

Another problem encountered that could not be solved was bus locking. Occasionally, the Arduino programs freeze on a call to `Wire.endTransmission()`, and only a restart on both devices resumes operation.

	Mean	Variance
PI time until actuation	1393 µs	39.16 μs^2
I2C Communication time (16 bytes)	1761.79 µs	694.3145 μs^2
Serial communication time (5 bytes)	384 µs	24 μs^2
Sampling time	5.00 ms	0.3248 μs^2

Table 4: Results for measuring task time on the Arduino.

Experiment 3

Figure 8 shows the step response of different control schemes. From the time-series data of 8c, some relevant parameters are calculated, and shown in table 5.

The first conclusion is that the illuminance model is not entirely accurate, looking at figure 8b, the gain tracks well a reference of 80 LUX, but not the 150 LUX one. However, it is fast, and does not overshoot.

The feedback controller tracks the reference with minimal error. It must be stated that the sensor reading is digital, and only 256 different values are possible, therefore the resolution is not high enough to ensure a smaller steady-state error. Some overshoot is present, and the initial response is slow, evident by the rise time[2].

Coupling the two controllers grants the best performance. Overshoot is eliminated by providing the feedback controller with the expected transient response, so that it does not overcompensate on actuation.

Overshoot	Damping, ξ	Rise time	Settling time
0.68 %	0.8499	0.135 s	0.191 s

Table 5: Step response characteristics of the PI controller.

Experiment 4

The time plots for this experiment are present in figure 9.

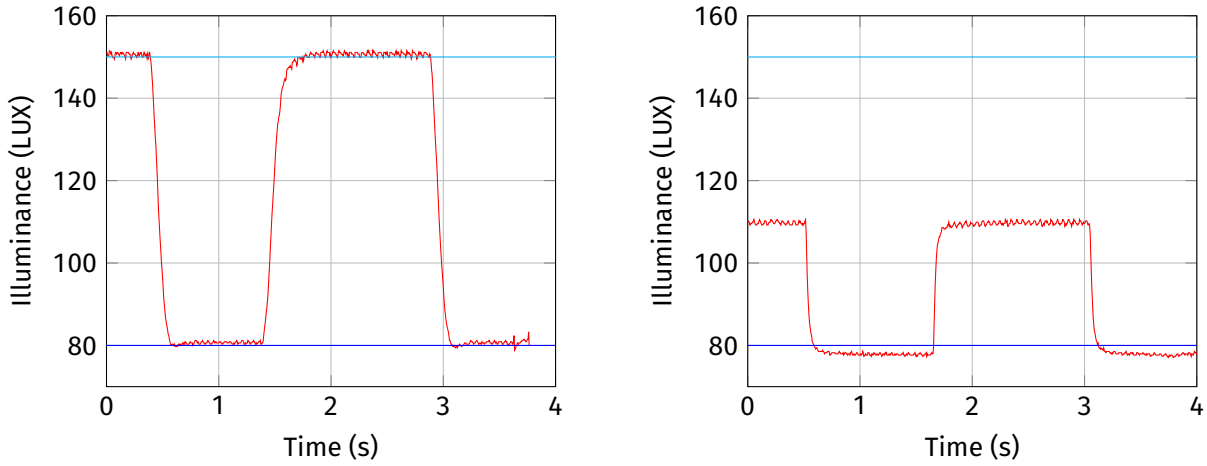
In figure 9a, when the box is closed, the illuminance drops close to 0, and stays that way for close to a second, before regaining its reference of 150 LUX. It is clear that when the external light was above the reference value, the error kept accumulating, so that after the bow was closed, the integrator must accumulate error in the inverse direction for a while, so that the actuation rises above 0.

In figure 9b, the illuminance still drops after the box is closed, but the actuation responds immediately, and the reference is regained faster. This shows that cancelling integration is an adequate mechanism for preventing windup due to actuator saturation.

Experiment 5

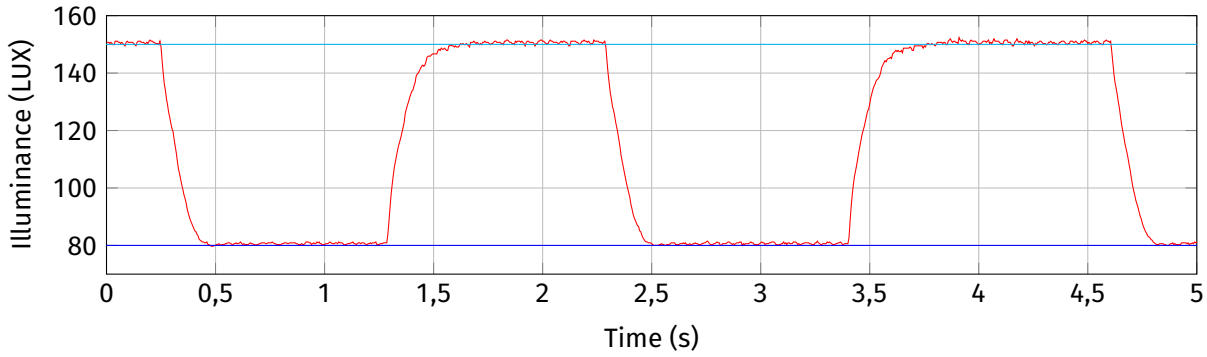
The result for this experiment is presented in figure 10 and table 6. The lower bound for this experiment is defined as 120 LUX for an empty desk, and 250 LUX for an occupied desk.

[2] Rise time is considered here to be the time taken for the signal to reach 90 % of its final value.



(a) Pure feedback controller.

(b) Pure feed-forward controller.



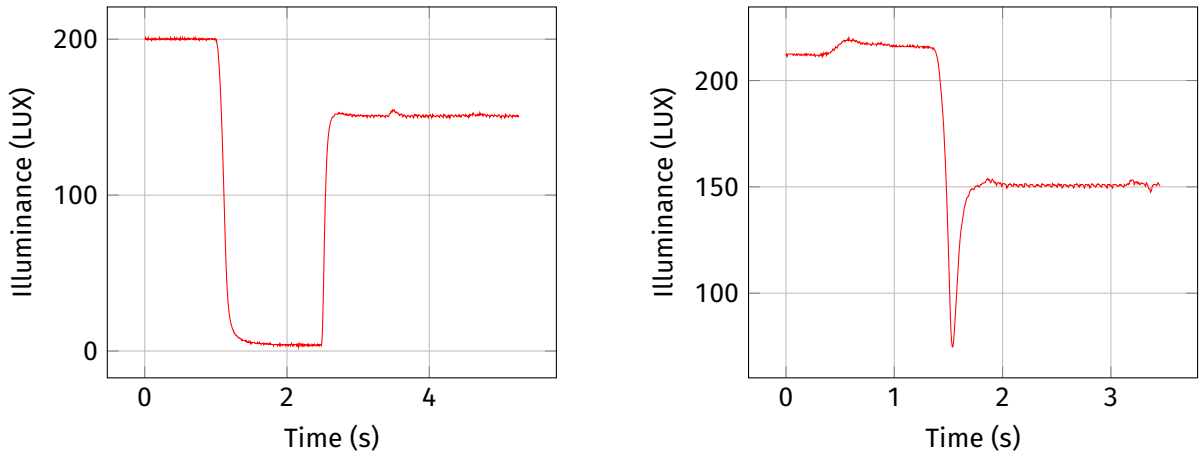
(c) Feed-forward + feedback controller.

Figure 8: System output with successive step inputs of 150 and 80 LUX.

As expected, for an independent control architecture, the node only tries to minimize its local power usage, therefore it is obvious that the reference values are simply the lower bounds, as evidenced by figure 10b. On a side note, the same is also the case if both nodes have the same cost coefficients.

For the cooperative control architecture, it is clear from figure 10a that the illuminance value is considerably above the lower bounds. This is the optimal result found by the distributed control algorithm, because increasing the dimming on this node, where power is *cheap*, and decreasing it on the other node, where power is *expensive*, results in overall lower energy consumption. It is also seen in this picture that the excess illuminance is higher while the desk is empty than while it is occupied, which is evidence that the solution is on the upper boundary for this node's dimming, for the occupied desk state. In other words, if it could increase its dimming more, it would.

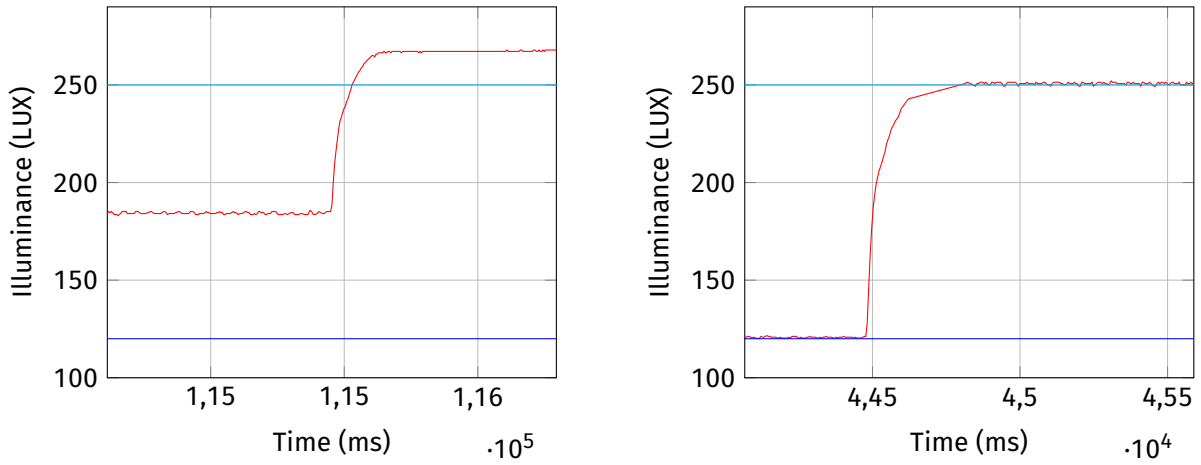
Some performance metrics are gathered in table 6. It should be admitted though that it was challenging to take accurate measurements for the same conditions, and the data can be inaccurate. As expected, total energy spent is lower for the cooperative control type. Only slightly, because the test was carried out with both desks occupied, where the solution is maximum dimming on the node with lower cost. The same is true for power. Total comfort error is also lower because the reference value is higher than the lower bound for one of the nodes, so noise never causes the illuminance to go below the lower bound. As for flicker, there is no apparent reason for it to be higher or lower on any case, and the difference seen is most likely due to inaccurate measurements.



(a) No anti-windup mechanism.

(b) Anti-windup prevention by cancelling integration.

Figure 9: Anti-windup performance.



(a) Node 2 illuminance, distributed control.

(b) Node 2 illuminance, independent control.

Figure 10: Time responses for experiment 5, lower bound is switched from 120 LUX to 250 LUX.

7 Conclusion

In the project, other distributed control approaches could be used, such as the centralized control architecture. The global results obtained with the non-centralized approach are satisfactory, but one doesn't know if other techniques could provide better outcomes.

In [Pan+15], a similar application with a central controller is implemented, which receives all the illuminance and occupancy information, and outputs the next dimming values of the luminaires. The results obtained with this approach have proven to be valid solutions, as they have shown negligible overshoot, some illuminance deviations and low energy savings. It also seems to comply with comfort requirements. In general, this approach seems to be as valid as the decentralized approach.

Despite that, non-centralized control has considerable advantages. It is less dependant on a single agent in order to work correctly, and thus is more fault-tolerant. Additionally, the computational overhead is evenly split among the available hardware, usually resulting in better performance. For the case of the consensus algorithm, each node only needs to search on its boundaries, yet the algorithm still guarantees optimality. With an increase in the number of nodes, the boundary regions grow exponentially if they are all considered at the same time in one centralized controller. We then conclude that distributed control has a huge potential on large networks of agents.

A PID controller with no feed-forward may also be a possible solution to implement. From its definition, a feed-forward segment allows a control system to act in terms of the inputs and not from

Metric	Independant control	Cooperative control
Total accumulated energy – g e T	8523.97 J	8490.16 J
Total instantaneous power consumption – g p T	864.45 W	851.65 W
Total confort error – g c T	4.5378 LUX	2.6512 LUX
Total flicker error – g v T	181.848 LUX s^{-1}	138.587 LUX s^{-1}

Table 6: Performance metrics, taken 10 s after a restart.

the responses obtained. Thus, it is the major reason why little to none overshoot is seen on the results. However, it could be even smaller if the performed calibration was more exact. A control design with no feed-forward could be implemented, but the settling time would be higher.

On the server front, an effort was made to include modern conventions and programming paradigms. The boost library is extensive and allows the construction reliable IO agents for usage in real time control, as it maintains the low level requirements for computer control.

8 References

[Ben94] Stuart Bennett. *Real-time computer control: an introduction*. Vol. 88. Prentice Hall Englewood Cliffs, 1994.

[AM10] Karl Johan Aström and Richard M Murray. *Feedback systems: an introduction for scientists and engineers*. Princeton university press, 2010.

[Pan+15] Ashish Pandharipande et al. "Centralized lighting control with luminaire-based occupancy and light sensing". In: *Industrial Informatics (INDIN), 2015 IEEE 13th International Conference on*. IEEE. 2015, pp. 31–36.

[eTe17] eTechnophiles. *How To Change Frequency On PWM Pins Of Arduino UNO*. 2017. URL: https://etechnophiles.com/change-frequency-pwm-pins-arduino-uno/ (visited on 01/11/2019).

[MSN17] Mohsen Mahoor, Farzad Rajaei Salmasi, and Tooraj Abbasian Najafabadi. "A hierarchical smart street lighting system with brute-force energy optimization". In: *IEEE Sens. J* 17.9 (2017), pp. 2871–2879.

[Ber18a] Alexandre Bernardino. "Real-Time Distributed Control Systems course slides". 2018.

[Ber18b] Alexandre Bernardino. "Solution of Distributed Optimization Problems: The consensus algorithm". Real-Time Distributed Control Systems 2017/18 course material. Nov. 14, 2018.

Task	**Daniel**	**João**	**Francisco**
Modelling and LDR calibration		R	I,R
PID control		I,R	I
Distributed control		I	I,R
Calibration	I,R		I,R
I2C Communications	I,R		
Server: client services	R	I	
Server: data management and concurrency	R	I	
Microcontroller programming	I	I	
C++ programming		I,R	

Table 7: Member tasks completed. R-reporting, I-implementation.

A Auxiliary schematics

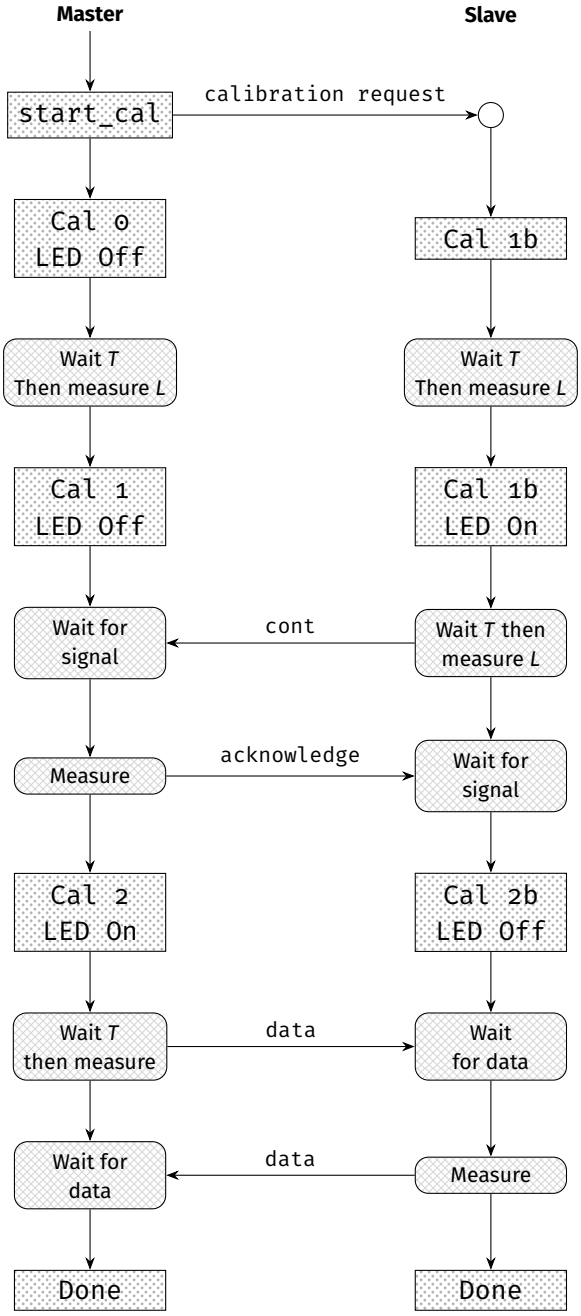


Figure 11: Calibration procedure.