Distributed Real Time Control Systems Project Report

João Borrego
78990
joao.borrego@tecnico.ulisboa.pt

Abstract:
This report presents an autonomous distributed lighting system to ensure proper illumination in an open-space office environment. Our goal is to develop a method that ensures minimum illuminance requisites depending on whether each desk is currently occupied or not. We build a prototype for our system using common electronic devices such as light-emitting diodes (LEDs) and light-dependent resistors (LDRs) and prototyping platforms such as Arduino and Raspberry Pi boards. Each node runs a local real-time controller which is tasked with following a reference illuminance value. We leverage inter-node communication to reach the global optimum conditions with a distributed approach, thus solving a consensus problem. Furthermore we develop a server application that provides an interface to interact with our system, for monitoring and remote control. Our work compares the local and distributed controller approaches, obtaining a a better performance for the latter scenario as evidenced by energy consumption and comfort metrics.

Keywords: Distributed Systems, Real-Time Control, Consensus, Distributed Optimisation, Arduino, Raspberry Pi

1 Introduction

In 2003, illumination reportedly accounted for 40% of the electrical consumption in offices in the U.S.[1]. Later studies describe a decrease to little over 17%[2], which is still a substantial fraction. With the ever-increasing electricity prices it is desirable to reduce the cost of illumination. Light-emitting diodes, commonly known as LEDs are typically very efficient, as they operate at much lower voltage and dissipate less heat than incandescent light bulbs. Furthermore the can easily be integrated in a digital control system to provide increased flexibility and an overall better solution.

Recent research has developed autonomous systems for lighting control using LED lights. Lee and Kwon [1] proposed a distributed energy-saving lighting strategy for configuring a so called *lighting* network. This work shares some major similarities regarding the conditions of the problem at hand. The authors describe an indoor scenario in which a globally optimal configuration is reached using a distributed algorithm.

We intend to simulate an office work space, in which each desk has its own dedicated light. Depending on its occupancy, an automated system should regulate the light intensity while ensuring a given minimum value. To achieve this we built a prototype using a typical shoe-box coated with reflective white paper so as to maximise light reflection. Each node consists of an Arduino[3] board with an ATmega328P processor and a simple electronic circuit with a light sensor and an LED light. A picture of the developed system is depicted in Figure 1.

Whereas the current illuminance is measured by a light sensor, the occupancy is not detected using a presence sensor. Instead this is simulated using software, yet could easily be extended to support these sensors. This occupancy is thus modelled by a Boolean variable per virtual work desk. Therefore, two illuminance lower bound values are possible: `LOW` and `HIGH`, corresponding to $\frac{100}{3}$ lx and $\frac{200}{3}$ lx respectively. The standard SI measurement unit for illuminance is lux and denoted as lx. These values were chosen arbitrarily and are used throughout the report.

[1] 2003 CBECS Survey Data, as of January 10, 2018
[2] 2012 CBECS Survey Data, as of January 10, 2018
[3] `https://www.arduino.cc/`, as of January 10, 2018

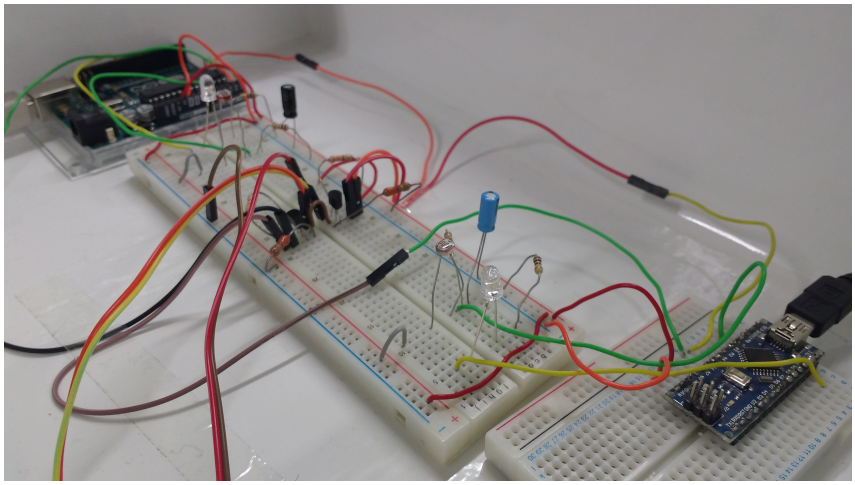
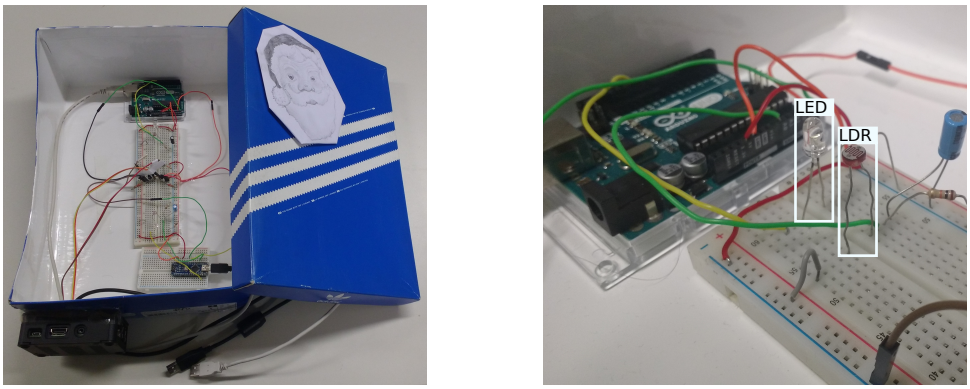


Figure 1: Close-up view of the developed prototype, inside an ordinary shoe-box. Each Arduino controls a single LED, LDR pair. An I^2C communication buffer connects the two nodes and can be seen between them.

The prototype box lid can be removed so as to introduce a disturbance in the controlled system. We provide more images of the system in Figure 2.

We further enrich the system with a TCP/IP server application running on a Raspberry Pi[4] Model 3B computer, which can be used to remotely extract system information, compute useful metrics and issue commands, such as changing occupancy values. It is lodged outside of the box so as to not interfere with reflection paths.



(a) Outside view of the prototype. The Raspberry Pi is visible in the bottom left corner.

(b) Close-up of a single system node. The LED light and LDR sensor are highlighted.

Figure 2: Different views of the developed prototype for a 2-node system

Report Outline

The remainder of the report is structured as follows. Section 2 consists of the system identification, for both the local and distributed control system scenarios. In Section 3 we describe the local real-time controller and its core features. Furthermore, we introduce a distributed optimisation algorithm to cope with system coupling and mutual interference for a multi-node system. Section 4 describes the communications between each implemented module, specifying the involved protocols and data packet structure. Section 5 presents the architecture of the Arduino program implementing the system node processing unit as well as that of the TCP/IP server. In Section 6 we validate our solution by presenting the results of several experiments and briefly discussing what they entail. Finally, Section 7 wraps up by stating our main accomplishments and limitations for possible future work.

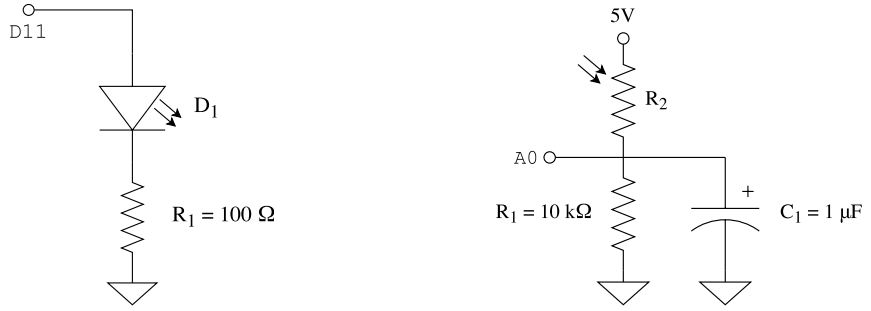
[4] https://www.raspberrypi.org/, as of January 10, 2018

2 System Identification

In this section we introduce the methods and tests performed in order to obtain a model of our system. First it is important to linearise the system, so we can control it a standard linear controller, such as PID (Proportional Integral Derivative). Once we have successfully attained a linear model for a single node we intend to study inter-node interaction, when in a multi-node system scenario.

2.1 Individual System identification

Each system node is consists of a processing unit, an LED and a light sensor. The corresponding electronic schematics are depicted in Figure 3.



(a) Led driving circuit (b) Illuminance reading circuit

Figure 3: Sensor and actuator schematics

The LED light intensity is simulated using PWM (Pulse-Width Modulation), which essentially consists of quickly alternating between on and off according to the desired duty cycle. This effect approximates a continuous light source with variable intensity. However, this transition is not instantaneous and will most likely lead to high frequency noise. In order to filter out this noise, we introduce a capacitor in the illuminance sensor circuit, which ultimately implements a low-pass filter.

2.1.1 Luxmeter Calibration

First and foremost, we need to convert our sensor input voltage into the SI unit for illuminance, known as lux (lx). As the name implies, the LDR's resistance alters when it is exposed to light. We can compute the value of its resistance R_2 by performing the voltage divider between R_1 and R_2 (cf. Figure 3b), obtaining

$$v_i = V_S \frac{R_1}{R_1 + R_2} \Leftrightarrow R_2 = R_1 \left(\frac{V_S}{v_i} - 1 \right), \tag{1}$$

where V_S is the positive supply voltage (5 V) and v_i is the measured voltage at the input analog pin `A0`. The measured illuminance depends on the calculated resistance value according to

$$\log_{10}(R_2) = a \log_{10}(L) + b. \tag{2}$$

We determined the values of a and b experimentally by obtaining two points and performing the linear regression. This linear regression between two points effectively corresponds to solving the equation system given by

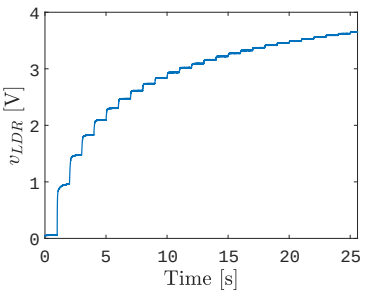
$$\begin{bmatrix} \log_{10} R_{2,1} \\ \log_{10} R_{2,2} \end{bmatrix} = \begin{bmatrix} \log_{10} L_1 & 1 \\ \log_{10} L_2 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix}. \tag{3}$$

By obtaining the LDR resistance corresponding to two separate lux values measured with an external lux meter, we obtained the data points ($R_{2,1} = 7517\ \Omega$, $L_1 = 35$ lx) and ($R_{2,2} = 3880\ \Omega$, $L_2 = 100$ lx). Then we computed the value of the LDR characteristic parameters by the solving the linear system, obtaining

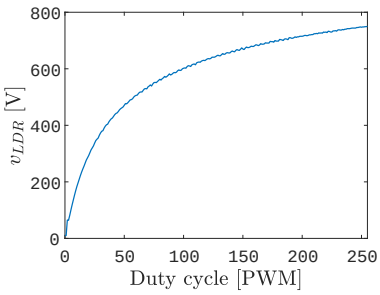
$$a = -0.62995 \quad b = 4.85870.$$

2.1.2 Equivalent First Order System

It is fairly simple to demonstrate the non-linearity of the light sensor voltage as a function of the LED intensity. We can achieve this by performing a simple stair response test, by varying the duty cycle from 0 to 255, with a fixed step of 10, as shown in Figure 4a. The steady state response to an input sweep (stair with step of 1) is presented in Figure 4b.



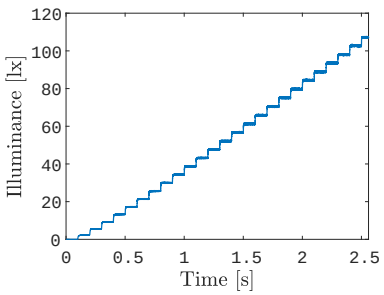
(a) $v_{LDR}(t)$ stair response, averaged 10 runs



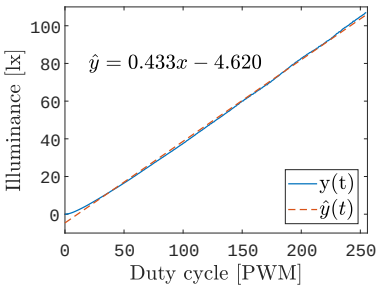
(b) $v_{LDR}(d)$ steady-state sweep response

Figure 4: System steady state characterisation, input in Volt

However, by performing the conversion of the input to lux units using Equation 3, we can obtain a linear characteristic for our system. Figure 5 presents the system response to the input signals shown in Figure 4, in terms of illuminance.



(a) $lux(t)$ stair response, averaged 10 runs



(b) $lux(t)$ steady-state sweep response, averaged 10 runs

Figure 5: System steady state characterisation, input in lux

By performing the linear regression of the curve in Figure 5b we obtain $r = 99.95\%$ for

$$\hat{y} = mx + b = 0.4330x - 4.6204 \quad \text{[lx]}. \tag{4}$$

Our objective is to model this system as a first order system, i.e. described by a continuous transfer function $G(s) = K_0/(1 + s\tau)$. Intuitively, this will not be the case. Even though the circuit in Figure 3b resembles a classic RC circuit, R_2 depends on the light intensity. Thus, so will the filter's properties, namely gain and time constant. We can however estimate the system parameters by analysing the step response for a fixed value (50%), as depicted in Figure 6.

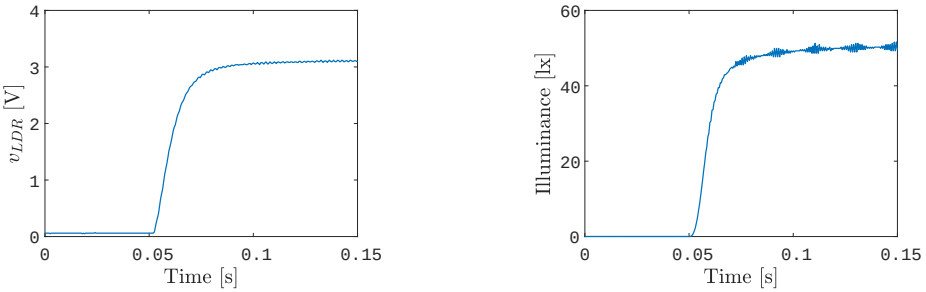
Notice that the sensor noise is considerably amplified when we perform the conversion to lux units, which compels us to use the voltage input directly. The output of the latter test allows us to compute the time constant and the static gain, obtaining

$$\tau = 11.2 \text{ ms} \quad G(0) = 0.02442 \text{ [V/PWM]}.$$

This time constant value corresponds to the time the system takes to stabilise, and should be a lower bound on the discrete controller sampling rate. For this reason, we picked 30 ms for the controller sampling rate f_S.

2.2 Coupled System Identification

Ultimately our aim is to build a system with many control nodes, each following a given setpoint. However, this multiple node scenario is expected to be more challenging, as we introduce inter-



(a) $v_{LDR}(t)$ step response in Volt

(b) $lux(t)$ step response in lux

Figure 6: System step response ($A = 128$ [PWM])

dependencies between each LED, LDR pairs. In short, we want to somehow model the influence each node has on the rest of the system and then leverage this information in order to optimise our control, in a distributed approach.

2.3 Calibration

We wish to model this coupling phenomenon using a calibration routine, which can run automatically on system startup. Assuming a linear illuminance response to an input PWM duty cycle, our multi-node system can be described by

$$\mathbf{y} = \mathbf{K}\mathbf{d}, \tag{5}$$

where $\mathbf{y}$ is the output vector, $\mathbf{K}$ is the matrix of coupling factors and $\mathbf{d}$ is the duty cycle vector. Specifically for a 2-node scenario, the linear system becomes

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} k_{11} & k_{12} \\ k_{21} & k_{22} \end{bmatrix} \begin{bmatrix} d_1 \\ d_2 \end{bmatrix}. \tag{6}$$

The elements on the matrix diagonal express the dependency of a system illuminance on its own output, whereas the remaining entries correspond to the coupling interference from other nodes. In order to calculate each coupling parameter we need to acquire (d_i,y_i) pairs. For instance, k_{11} can be estimated by setting $d_2 = 0$ and computing y_1/d_1. In order to obtain a robust estimation we acquire a pre-defined amount of samples per system node. Specifically, we start by choosing a value for the duty cycle of a given node's LED from an array of duty cycle values. Furthermore we ensure that all other nodes' output is set to zero. We measure the illuminance in each of nodes. Then we repeat this procedure for each duty cycle value in the aforementioned array, and for each node in the system.

Finally, we perform linear regression using least-squares method with each of the N samples per system node. Each coupling parameter is estimated according to

$$k_i = \frac{\sum_{i=1}^{N} \left(d_i - \bar{d}\right)\left(y_i - \bar{y}\right)}{\sum_{i=1}^{N} \left(d_i - \bar{d}\right)^2}, \tag{7}$$

in which d_i is the i-th value from the pre-defined N-dimensional array of duty cycle outputs, y_i the corresponding measured output and $\bar{d}$ and $\bar{y}$ are the averaged duty cycle and output values respectively. In the end, each node i is able to estimate the i-th row of the full $\mathbf{K}$ matrix. Our calibration routine scales the 8-bit value of the duty cycle to a percentage, which is equivalent to multiplying each entry of $\mathbf{K}$ by $255/100$.

However, this method requires explicit inter-node communication to ensure synchronisation, or race condition issues may arise. For instance, it is necessary to ascertain that each node has acquired and registered the illuminance value before allowing the LED to change. Moreover, we should guarantee that the light sensor has stabilised, which can be achieved by forcing a small delay between changing the LED output and reading the illuminance input.

It will also be useful to acquire the value of external illuminance at each node, i.e. the output value measured when every node's LED is turned off. These values are denoted by the vector $\mathbf{o}$. Each node only needs to compute its own o_i parameter, as well as corresponding row of $\mathbf{K}$, designated by $\mathbf{k}_i$.

3 Control

3.1 Individual Controller

Each system node is tasked with following a given illuminance reference. In order to achieve this we implemented a simple feed-forward controller combined with a Proportional Integral (PI) controller. A block diagram of the implemented controller is shown in Figure 7.

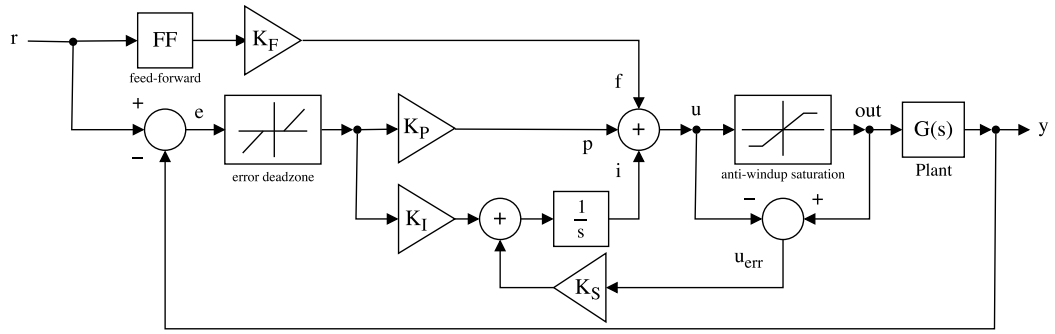


Figure 7: Individual Controller Block Diagram

We present a brief overview of each component.

3.1.1 Feed-forward Controller

The feed-forward controller assumes a linear approximation of the system response to a given input, and merely applies the corresponding output for the provided reference. Thus, its contribution is given by

$$f = K_F \, \frac{r - b}{m} \tag{8}$$

with K_F the feed-forward gain, r the setpoint reference, and m and b the parameters of the approximated linear characteristic of the system illuminance with respect to the output duty cycle, computed in Equation 4. In the case of the distributed controller we can instead use k_{ii} and o_i obtained from calibration as the parameters for the linear approximation ($\hat{y}_i = k_{ii}\, x + o_i$).

The feed-forward term does not incorporate the output in the system in any way, and is therefore prone to external disturbances. To address this issue we introduce a feedback PI controller.

3.1.2 PI Feedback Controller

The feedback controller is consisted of a proportional and an integral term, given by

$$p(t) = K_P \cdot e(t) \quad i(t) = K_I \cdot i(t-1) \cdot \frac{e(t) + e(t-1)}{T/2} \tag{9}$$

with $e(t) = r(t) - y(t)$ the measured error. We chose to use a parallel architecture in order to decouple the proportional and integral gains, K_P and K_I respectively, which should simplify their estimation. Even though the computation of these terms follows the standard approach, we improved the basic controller with additional features in order to mitigate possible issues.

3.1.3 Controller Features

We will now provide a succinct description of potential problems and respective proposed solution. Concrete results comparing the impact of adding each of these features are provided in Section 6.

Feed-forward Controller

Combining the feedback and feed-forward controllers can be done simply by summing their output values. This should allow the resulting controller to achieve faster transitions to abrupt setpoint changes.

The feed-forward term contribution to the controller output is adjusted with a feed-forward gain K_F. Whenever it is set to a high value, transitions should be swift, yet the reference tracking error may

be large. Moreover, this is likely to increase the controller's overshoot. If K_F is instead set to a low value, transitions should be slower, but the PI controller should be able to follow the reference with an improved level of accuracy.

Anti-windup

Integral windup is a phenomenon that occurs in a PI controller when an abrupt setpoint change or external disturbance causes the integrator term to accumulate a significant error (windup). This results in a considerable response delay, as the accumulated error is unwound.

A simple example of such a situation in the context of our project consists of setting the reference to a low illuminance value and opening the box in an illuminated environment. The controller will likely measure a very high illuminance in the sensor and turn off the LED completely. However, the integrator term will accumulate the error over time. If the box is closed once again, the controller will take a substantial amount of time before resuming a normal behaviour.

There are several solutions to this issue and we have implemented two of them, namely

1. Saturating the integrator term;
2. Saturating the controller output and adding a feedback loop to the integrator term to incorporate the difference between the unsaturated and saturated input, as shown in Figure 7.

Although the first approach produces decent results, the second proved to be superior, as it converges faster to the reference as soon as it becomes possible to reach it. Be that as it may, the first approach is simpler and does not require fine-tuning any parameter, as the saturation limits can simply be the minimum and maximum output values. The feedback loop solution involves fine-tuning a K_S gain.

Error Deadzone

Since we are working with a digital control system it is likely that issues regarding quantisation may arise.

Firstly, the measured input has a resolution of 1024 bits in a range of 5 Volt. This essentially translates to a precision of roughly 5 mV. Furthermore, there is also a source of quantisation error in the system output, as the PWM signal only has 8-bit resolution.

These errors are likely to add up and cause a variation in the system input, even for a fixed output. This essentially results in flickering, as the controller continuously tries to make up for the small error it detects at the input.

A simple solution for this problem is the usage of an error deadzone, which provides a band of values for which the error is considered to be zero. This mitigates the constant controller update and the undesired flickering.

There is an obvious trade-off between reduced quantisation noise impact and static error, for low and high deadband width values respectively.

3.2 Parameter Tuning

In order to determine the parameters for the PI controller gains, namely K_P and K_I we resorted to MATLAB Simulink® PID Tuner application. We used the response of the LDR signal to a PWM step of value 128 (middle of the available range). Given the respective parameters, the tool can calculate the approximate first order system. In order to do so, we establish that the system has a single pole. We determine an approximate linear model for our system using PID Tuner, which solves an optimisation problem using a line search method such as Gauss-Newton obtaining

$$H(s) = \frac{K}{T_1 s + 1} = \frac{0.02442}{0.01266\, s + 1}.$$

We then export the identified plant, thus acquiring the respective continuous transfer function. We discretise the system using Tustin method and compute the discrete transfer function given by

$$H(z) = \frac{0.01324\, z + 0.01324}{z + 0.08465}.$$

Finally we ran PID Tuner once more with specified parameters for the desired controller. We impose a response time of 0.6 seconds and set the program robustness parameter to 0.85, which results in

$$K_P = 2.0510 \quad K_I = 136.7362.$$

The algorithm is part of a proprietary toolbox from MathWorks® and although its implementation is not public, a brief explanation of is available in the official documentation[5]. We usually refrain from using this black-box approach, and merely decided to use these parameter values since they provided satisfactory experimental results and due to being similar to the preliminary parameters we determined by trial and error (mainly K_P). We provide detailed instructions for replicating the procedure [6].

We refer once more to Section 6, in which we present experimental results of our fine-tuning procedures. For now, we provide the final values for each of the relevant parameters

$$K_F = 0.5, \quad K_S = 0.5, \quad sat_{\min} = 0, \quad sat_{\max} = 255, \quad err_{\min} = -3, \quad err_{\max} = 3,$$

where $sat_{\min}$, $sat_{\max}$ are the anti-windup saturation negative and positive limits and $err_{\min}$, $err_{\max}$ are the threshold values for the error deadzone, respectively.

3.3 Distributed Controller

The projected controller was tested and fine-tuned using a single node setup. However, as soon as we transition into a multi-node system we should observe performance degradation. This is mostly because of the introduced system coupling, which led to the creation of the aforementioned calibration routine.

Our objective is to leverage this knowledge regarding system coupling and the possibility of inter-node communication in order to optimise the overall system behaviour while constrained on each node's illuminance requirements. In order to achieve it we introduce a distributed optimisation algorithm.

3.3.1 Consensus Algorithm

We wish to calculate the global optimum solution for the output values of each node, provided that each of them fulfils some restrictions. These are related to physical limits for the output, and lower bounds on illuminance value so as to ensure a certain degree of comfort. Moreover we intend to produce this result by distributing computation across system nodes. This is what is known as a consensus problem as it requires the agreement of every agent in this scenario regarding a single data value. In this case, the latter consists on the vector $\mathbf{d}$ which contains the optimal output value d_i for a set of given requirements.

First and foremost we must formally define our case as an optimisation problem. Our system should minimise energy consumption, which is linearly dependent on each node's LED output duty cycle. Thus, our cost function should incorporate a linear term $\mathbf{c}$ which penalises the energy consumption at each node. Furthermore, we wish to avoid nodes with very high output values, as it severely decreases their longevity, especially if they are unable to properly dissipate heat. For this reason we incorporate a quadratic term in the cost function. Therefore the cost function can be written as

$$f(\mathbf{d}) = \mathbf{c}^{\mathsf{T}}\mathbf{d} + \mathbf{d}^{\mathsf{T}}Q\mathbf{d}, \tag{10}$$

where $\mathbf{Q}$ is a diagonal matrix of the quadratic costs associated with each node's output. Notice that this function is *separable* in $\mathbf{d}$, i.e. can be partitioned in a sum of functions $f_i(d_i)$.

We want to impose certain constraints, which define the feasible region for the problem solutions. First, each node has a lower and upper bound on the output value, respectively 0 and 100%. Hence, we introduce $2n$ linear inequality constraints, with n the number of nodes in the system. Additionally, we want to impose a minimum illuminance L_i per desk depending on their occupancy status and external illuminance o_i. This adds yet another linear inequality constraint per node. Summarising, our constraints are given by

$$\mathcal{C} : \left\{ 0 \leq d_i \leq 100 \quad \wedge \quad \sum_{j=1}^{N} k_{ij} d_j \geq L_i - o_i, \quad \forall i \right\}. \tag{11}$$

[5] https://www.mathworks.com/help/slcontrol/ug/introduction-to-automatic-pid-tuning.html, as of January 10, 2018

[6] : PID Tuner Instructions, as of January 10, 2018

Our optimisation problem can be written as a quadratic programming problem

$$\begin{aligned} \underset{\mathbf{d}}{\text{minimise}} \quad & \mathbf{c}^\mathsf{T}\mathbf{d} + \mathbf{d}^\mathsf{T} Q \mathbf{d} \\ \text{subject to} \quad & -\mathbf{K}\mathbf{d} \leq \mathbf{o} - \mathbf{L}, \\ & -\mathbf{d} \leq [0], \\ & \mathbf{d} \leq [100]. \end{aligned} \tag{12}$$

In order to apply a distributed approach, we must first partition the global cost function and constraints with respect to $\mathbf{d}$. The objective function can be decomposed by manipulating the entries of the local copies of $\mathbf{c}$ and $\mathbf{Q}$, respectively $\mathbf{c}_i$ and $\mathbf{Q}_i$, obtaining

$$\mathbf{c}_i = [0 \cdots c_i \cdots 0]^\mathsf{T} \quad \mathbf{Q_i} = \text{diag}(0, \ldots, q_i, \ldots, 0). \tag{13}$$

Decomposing the constraints effectively corresponding to each node testing exclusively the respective output limits and its illuminance lower bound, which eliminates the need for each node to hold the entire $\mathbf{K}$ matrix and $\mathbf{o}$ vector. Thus, we have

$$\mathcal{C}_i : \left\{ 0 \leq d_i \leq 100 \quad \wedge \quad \mathbf{k}_i^\mathsf{T}\mathbf{d}_i \geq L_i - o_i, \quad \forall i \right\}, \quad \mathbf{k}_i = [k_{i1} \cdots k_{iN}]^\mathsf{T}. \tag{14}$$

The optimisation itself is performed using ADMM (Alternating Direction Method of Multipliers) [2] which is extensively described in Boyd et al. [3]. It is an iterative method according to which each node calculates a possible solution $\mathbf{d}_i$. We ensure that at each iteration every node has an identical vector of local copies $\mathbf{d}_i$ of the global variable $\mathbf{d}$. ADMM attempts to keep desirable properties of two other algorithms, namely the decomposability of dual ascent and fast convergence of the method of multipliers. It computes the augmented Lagrangian as in the method of multipliers, but separates the minimisation over the input variables. In each iteration at time t, the i-th system node performs

$$\begin{cases} \mathbf{d}_i(t+1) = \underset{\mathbf{d_i} \in \mathcal{C}_\mathbf{i}}{\arg\min} \left\{ \frac{1}{2} + \mathbf{c}_i^\mathsf{T}\mathbf{d}_i(t) + \mathbf{y}_i^\mathsf{T}(t)\left(\mathbf{d}_i(t) - \bar{\mathbf{d}}_i(t)\right) + \frac{\rho}{2}\left\|\mathbf{d}_i(t) - \bar{\mathbf{d}}_i(t)\right\|_2^2 \right\} \\ \bar{\mathbf{d}}_i(t+1) = \sum_{j=1}^{N} \mathbf{d}_j(t) \\ \mathbf{y_i}(t+1) = \mathbf{y}_i(t) + \rho\left(\mathbf{d}_i(t+1) - \bar{\mathbf{d}}_i(t+1)\right) \end{cases} \tag{15}$$

where $\mathbf{y}_i$ are the Lagrange multipliers, ρ is the penalty parameter and $\bar{\mathbf{d}}_i$ is the average solution. Our local cost function becomes

$$f_i(\mathbf{d}_i) = \frac{1}{2}\mathbf{d}_i^\mathsf{T} Q_i \mathbf{d}_i + \mathbf{c}_i^\mathsf{T}\mathbf{d}_i + \mathbf{d}_i^\mathsf{T}\mathbf{y}_i - \bar{\mathbf{d}}_i{}^\mathsf{T}\mathbf{y_i} + \frac{\rho}{2}\left(\mathbf{d}_i - \bar{\mathbf{d}}_i\right)^\mathsf{T}\left(\mathbf{d}_i - \bar{\mathbf{d}}_i\right). \tag{16}$$

In our case, the optimal solution to the global optimisation problem should lie along one of the linear constraints defining a convex feasible region in its interior. ADMM makes few assumptions for convergence, namely that the objective functions f_i are convex, closed and proper. However, it assumes that the unaugmented Lagrangian L_0 has a saddle point, which ensures there *exists* indeed a (not necessarily unique) global minimum $\mathbf{d}_i^*$.

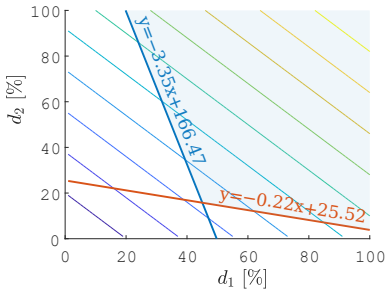
Figure 8 provides the output of the algorithm for example scenarios in the implemented two-node system using real calibration data, concretely [7]

$$\mathbf{L} = [66.(6) \quad 33.(3)]^\mathsf{T} \quad \mathbf{K} = \begin{bmatrix} 1.33537 & 0.39906 \\ 0.28057 & 1.29531 \end{bmatrix} \quad \mathbf{o} = [0.23447 \quad 0.27403]^\mathsf{T}.$$

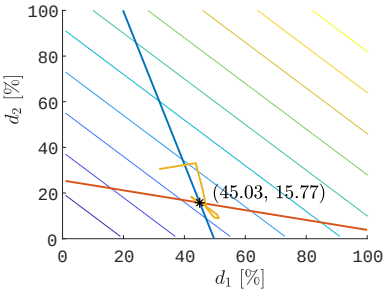
In some scenarios, the optimal solution is not given by an intersection of two constraints, but is coincident with the line segment of a linear constraint contained in the feasible region, highlighted in blue in Figure 8b. This is the case in Figure 8c where a quadratic cost function is employed. Figure 8d shows the scenario where the cost function contour lines are parallel to a linear constraint, leading to infinite optimal solutions. We have verified empirically that indeed similar situations ,i.e. cost function with contour lines nearly parallel to one of the linear constraints, may require a large number of iterations before converging to the global optimum (upwards of 1000 in some cases). Notice that by changing the cost function parameters we can penalise using a certain LED in particular, for instance.

In Figure 9 we provide the evolution of the local solutions (9a, 9b) and of the local cost function (9c) for the scenario of Figure 8b per node.

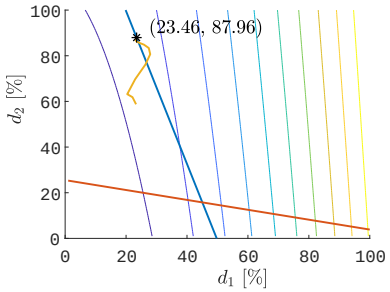
[7] These values corresponds to a valid illuminance configuration and the calibration output when performing the experiment reported in Figures 18a and 18b.



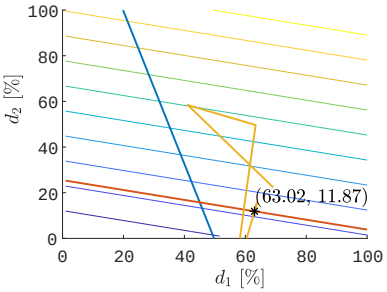
(a) Constraints and feasible region



(b) $c = [1.0\ 1.0],\ q = [0\ 0]$

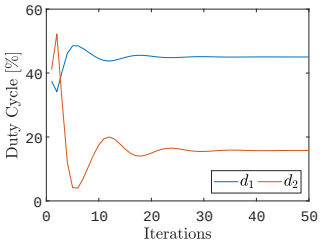


(c) $c = [1.0\ 1.0],\ q = [0.1\ 0.0]$

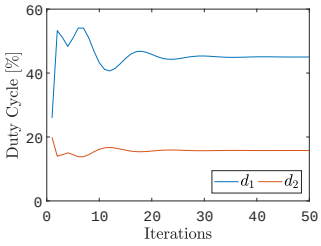


(d) Infinite optimal solutions

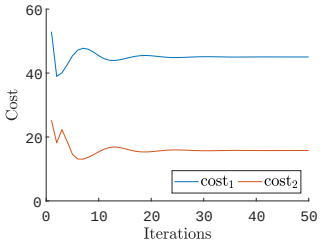
Figure 8: Impact of changing the cost function parameters



(a) Node 0 proposal over time



(b) Node 1 proposal over time



(c) Cost function in each node

Figure 9: Consensus progress over time, per node

Our implementation of the consensus algorithm is generic, as the problem is formulated using matrices with dimension relative to the number of nodes N. In order to achieve this we had to implement a small C library to work with this data. Furthermore, the algorithm requires explicit communication at the end of each iteration, so nodes can compute the new average solution. The communication procedure is further detailed in section 4.

There are however a few possible improvements that are worth being mentioned. First off, we use an arbitrary stopping criterion of a maximum of 50 iterations. Figure 9 shows that the algorithm has already converged roughly after the 30^{th} iteration. Finding a better stopping criterion should diminish the total run time an thus provide a faster real-time transition when modifying occupancy values. We could also optimise to the algorithm itself, implementing for instance a variable penalty parameter so as to boost convergence.

We should finally mention that the output of the algorithm is the set of optimal duty cycle values $\mathbf{d}^*$ and we intend to follow an illuminance reference in lux. Recall that although the duty cycle is calculated as a percentage, the matrix $\mathbf{K}$ estimated during the calibration routine already accounts for the transformation to 8-bit values. The new value of the reference to be followed can be obtained by computing [8]

$$L_i^* = \mathbf{k}_i^\mathsf{T} \mathbf{d}^* \quad [\text{lx}]. \tag{17}$$

[8] For instance, in the case of Figure 8b we obtain $L_1 = 66.43$ lx and $L_2 = 33.06$ lx.

4 Communication

We have seen that the distributed control system requires explicit communication for a calibration routine and running the consensus optimisation algorithm. Furthermore, we implement a TCP/IP server that can handle client requests asynchronously. The latter include displaying system information and computing performance metrics as well as changing the occupancy of a virtual desk. Thus, the server must itself maintain a two-way connection with the system. In this section we elaborate on how each type of communication is achieved.

4.1 Inter-Node Communication

All system nodes are connected via I^2C (Inter-Integrated Circuit) bus, with a multi-master paradigm. Instead of employing a master request, slave responds approach, each node is allowed to communicate directly with all others.

We used the original Wire TWI/I2C library[9] for the Arduino in order to implement these communications. The single major drawback from doing so is the inability to send broadcast messages, i.e. addressed to id 0x00, which are received by all other nodes. For this reason, if we wish to share a value with N nodes on the network, we must send N separate messages.

We enhance the robustness of the protocol by implementing a blocking send function, that only proceeds when an acknowledgement packet is received in return. If a single broadcast message was sent, it is more likely to occur a conflict in the bus between ACK responses from the remaining nodes. This situation might be further aggravated by the conflict resolution issues known to exist in the Wire implementation, when working with a multi-master paradigm[10].

The blocking send function is always called from within a loop, ensuring that the node keeps sending a message until it obtains an acknowledgement packet. This is useful when some synchronisation issues are present and the receiver node fails to obtain the incoming packet. A minimum delay is specified between attempts so as to not saturate the I^2C bus.

4.1.1 I^2C Packet Types

We have defined three main types of data packets, namely

1. Header-only packets, used for simple commands and acknowledgements;
2. Consensus packet, used in the distributed optimisation algorithm;
3. Information packet, containing relevant system variables.

A graphic representation of the contents of each packet for a 2-node system can be seen in Figure 10. The data in a consensus packet holds different meaning depending on whether the algorithm

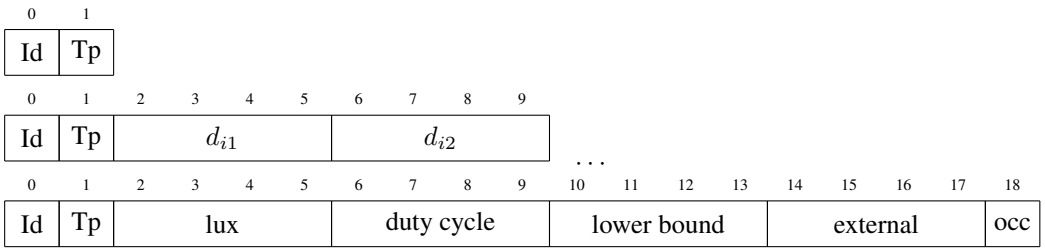


Figure 10: I^2C packet structure for each of the three possible types: header-only, consensus and information. The numbers represent the byte indices. Id and Tp refer to the sender's device identifier and the type of message respectively.

has already started or not. A consensus packet received in the control state signals the start of the consensus algorithm. Its data includes the respective lower bounds for each of the nodes. A consensus packet is also used during the algorithm run itself for each neighbour to communicate its local solution to its neighbours every iteration.

[9] https://github.com/arduino/Arduino/tree/master/hardware/arduino/avr/libraries/Wire/src, as of January 10, 2018

[10] https://github.com/arduino/Arduino/issues/1313, as January 10, 2018

Be that as it may, notice that even though these are said to be both consensus packets, they are branded with a different type flag, respectively ICO (Initate COnsensus) and CON (CONsensus).

An information packet holds the floating point variables for currently measured illuminance, duty cycle (%), the illuminance lower bound, the external illuminance, and a byte representing the occupancy as a Boolean variable.

4.1.2 Latency

The Wire library sets a default I^2C bus frequency of 100 kHz, which we left unchanged.

We have estimated the latency by calculating the round-trip time (RTT). This corresponds to measuring the difference between sending the packet and obtaining the acknowledgement from the receiver. We measured an average round-trip time of 20.1 ms over 20 samples, 10 per each node. It is fair to assume that the latency is approximately half of this delay, roughly 10 ms.

There is a clear trade-off between communication speed and robustness. Most communication procedures involve a node broadcasting a message and then waiting for each node to acknowledge. However, if we were to alter the underlying communication library we would have to handle the communication conflicts that could occur when two nodes respond with an ACK message at the same instant.

In any case, our implementation is merely sufficient for the project context. For instance, we assume that for each sent packet a single acknowledgement is needed. This limits our communication to a single packet at a time. We could for instance improve the communication by simply adding a sequence number to each packet, which would allow us to acknowledge several packets at once. However, improving the communication much further could easily end up in an implementation of a simplified version of TCP, which is clearly excessive and beyond the scope of this project.

4.2 Node-Server Communication

The server is required to monitor all of the nodes in the system yet only needs to interact directly with a single node. The latter is referenced as the master and is responsible for propagating requests to the remaining nodes in the system, namely reset commands or setpoint changes.

Our solutions uses separate channels for each of the required interactions. To gather system information, the server device listens to the I^2C bus connecting each node. In order to issue commands, the server uses a Serial connection with the so-called master node.

4.2.1 I^2C Sniffer

The server is connected to the previously introduced I^2C bus. It is our objective to obtain a real-time stream of relevant information using this read-only channel. This data can then be sent to a client or used to compute performance metrics and evaluate the overall performance of the system. However, the server runs on a Raspberry Pi device, which runs on 3.3 V unlike the Arduino boards, which operate on 5 V. The safest way to connect them both to an I^2C bus is to implement a bi-directional level shifter[11], as shown in Figure 11.

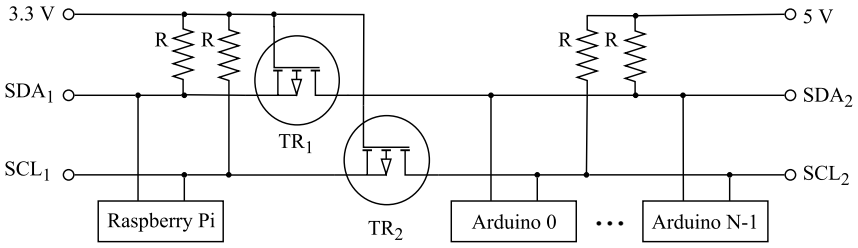


Figure 11: I^2C bus with level shifter circuit. R=10kΩ

The Raspberry Pi can have a dedicated identifier and behave as a regular node on the network. Alternatively, we can monitor the bus using an I^2C sniffer application. In doing so we achieve a

[11] https://playground.arduino.cc/Main/I2CBi-directionalLevelShifter, as of January 10, 2018

more modular system, and can reuse existing code. We modified the I^2C sniffer example script in the `pigpio` library[12] in order to fit our needs.

Our program monitors I^2C signals and reconstructs a data packet one byte at a time. Each packet is written to a named pipe (FIFO) as soon as it is complete. The server can monitor the named pipe asynchronously using the Boost.Asio[13] library. Thus, our solution provides an asynchronous method to handle incoming I^2C packets received in an otherwise synchronous polling fashion.

It should also be mentioned as an implementation note that even though the Raspberry Pi device does not have to be registered and have its address for I^2C communications, it is still required that each packet is acknowledged. We observed that if this is not the case, the sniffer application would be unable to recognise the contents of the full packet. In order to ensure that each device acknowledges a single packet so as to avoid congestion, a given node i sends a messages to $(i+1)\%N$, with N the number of nodes in the system.

4.2.2 Serial Communication

An Arduino board is directly connected via USB to the server device, and is tasked with broadcasting incoming messages to the rest of the system. This channel operates a Serial protocol with a baud rate of 115200 bps. Issued commands are transmitted as encoded strings and interpreted in the master node. The latter then produces the corresponding I^2C packet with the request information and broadcasts it to the bus.

The server has a dedicated I/O handler for asynchronous Serial write operations. The main motivation behind this design choice is that introducing a blocking function call in the request handler would likely stall pending requests, if the Serial communication channel happened to be busy.

4.3 Client-Server Communication

The developed server application employs TCP/IP (Transmission Control Protocol / Internet Protocol). TCP ensures a reliable and ordered transmission of packets. Our implementation resorts to the Boost.Asio library for providing asynchronous request handling, thus supporting multiple clients while using a single I/O service object.

A TCP server has a listener socket associated with a known port (17000 specifically). Clients first connect to this endpoint and are then redirected to a handler socket once the connection has been established. Boost.Asio provides a higher level of abstraction and although the concepts holds, the terminology differs slightly. The TCP Server object has a connection acceptor which creates a TCP session whenever a new client successfully establishes a connection. This TCP Session object and its attributes are specific to the respective client. Moreover, we can run several TCP sessions concurrently associated with just one I/O service instance.

Our server provides an API (Application Programming Interface) with a full specification of allowed requests and expected response format, and adhere to the project requirements. These are fully specified in the project documentation.[14]

The server can be requested to provide the current value of a given system variable or compute a certain performance metric. These requests can be answered immediately, thus completing the interaction and allowing the server to move on to other requests. However, the server also has to be able to stream data to the client periodically. For this reason there are two major routines performed per each TCP session. The first is a simple asynchronous read write loop which handles the requests which do not require a continuous interaction. Our implementation follows the recommended structure of Boost.Asio asynchronous applications, further dividing the read and write operations in a setup and handler functions. The second important routine involves using a timer which is triggered every 300 ms. The client's current preference regarding which data it expects in the data stream is stored in flag variables. Each time the timer is triggered, these flags are evaluated and a message is sent with the desired information. Notice however that this does not correspond to a real-time data feed, as the server is expected to receive system information at a higher rate than the stream update frequency. This is however a design choice in order to simplify interactions between the server components, which were separated in order to achieve a modular system. Furthermore it helps decoupling the server communication and request handling with system data logging and command issuing (cf. Section 5).

[12] `http://abyz.me.uk/rpi/pigpio/examples.html`, as of January 10, 2018

[13] `http://www.boost.org/doc/libs/master/doc/html/boost_asio.html`, as of January 10, 2018

[14] : Server API documentation, as of January 10, 2018

5 Architecture Description

This section provides a detailed description of the structure of the controller system that is run in each system node as well as the server application. It gives a brief high-level overview of the main modules, and then describes relevant implementation details.

5.1 Single Node Architecture

Each device has a simple state machine which determines which should be the behaviour of the given node. The latter includes controlling the physical system, performing calibration or running the consensus algorithm. Each of these routines is separately implemented by its own module. Furthermore, we developed utilities for I^2C outgoing communication, performing computations involving matrices and unit conversion between voltage and lux for obtaining the illuminance value.

A diagram of the module interaction and access is depicted in Figure 12.

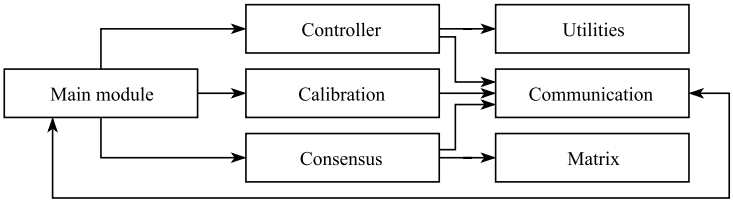


Figure 12: Node architecture divided in relevant modules. Arrows represent a module dependency.

There exist three possible states, specifically

1. `CONTROL` in which the PI controller ensures the reference illuminance is followed;
2. `CALIBRATION` in which the nodes recompute the values of $\mathbf{K}_i$ and o_i;
3. `CONSENSUS` in which the nodes attempt to agree on new reference illuminance values.

The main loop essentially performs two tasks, updating its current state and output the values of the system variables, namely the current illuminance, LED duty cycle and the desk occupancy. Additionally, the master node has to read commands from the Serial interface, which is done synchronously, yet one character at a time.

The main module holds the system variables, namely the measured illuminance input, the duty cycle output, the control targets i.e. setpoint and illuminance lower bound and state machine flags. Furthermore, the main program has an instance of a PI controller object, which is used to perform the calculations for the real-time controller. A custom interrupt routine is set to activate every 30 ms, which in turn triggers the controller update function. The refresh of the LED output is performed between the controller computations and the update of its internal variables. This is only possible by passing a pointer to an external update function to the controller class. This is a tiny implementation detail, but ensures that the system updates the output as soon as possible. The controller is only allowed to operate if the device is in `CONTROL` state. Otherwise the interrupt routine simply returns.

Apart from the controller update, only receiving packets is an asynchronous operation. This typically occurs when the master node sends a command to the rest of the system via I^2C bus. Every other operation is synchronous, even sending the system information packets. Here, the periodic behaviour is ensured by using `millis` function and comparing the current time with the last registered timestamp. A message from the master node can cause a device to change it's state. The asynchronous receive function may modify volatile flag variables which trigger a state transition on the next call to the update state function.

The state machine is responsible for launching both the calibration and consensus procedures. These require explicit synchronisation and communication, yet this is handled by each module. It is possible to achieve this by changing the `Wire::onReceive` callback function during run time. Each module defines which action should be performed when a packet is received, yet they all share the communications module for sending messages. The latter defines the communication protocol seen in Section 4. This provides separation between modules and an abstraction layer which should be useful should we wish to modify their implementation, for instance by altering the communication protocol.

5.2 Server Structure

The server application is somewhat more complex, mostly due to having to interface with three different types of I/O services. These include UNIX named pipes, TCP sockets and Serial. Fortunately, the Boost.Asio library provides an abstraction layer allowing us to handle distinct interfaces uniformly.

Figure 13 represents the overall structure of the server, identifying which processes run on the host machine, and the relevant execution threads.

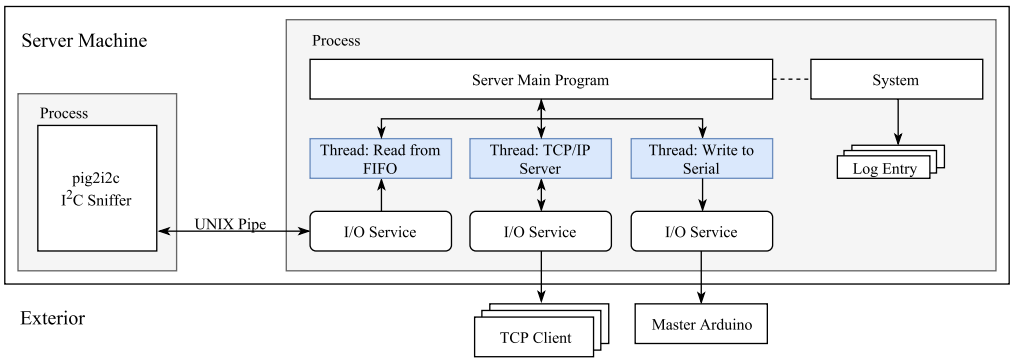


Figure 13: Server architecture divided in relevant modules. Arrows represent explicit access or communication. Grey and blue boxes represent processes and threads respectively.

Each communication channel has an associated Boost.Asio I/O service instance. These are handled by separate threads, in order to ensure totally asynchronous behaviour. However, this may cause synchronisation issues, as the various threads attempt to read from or write to a shared resource.

The system class behaves as an interface between the server and the physical system. It holds the logged entries containing timestamp, illuminance, duty cycle and reference illuminance. These are kept in a simple list, as the host device has more than enough memory for normal execution. Nevertheless, swapping the data structure should be fairly straightforward. In the final stages of the project we decided to add the comfort error and variance values to the log entry, for testing purposes. The access to the logs is protected by a read-write lock (shared mutex) which grants concurrent access to any number of readers, yet only allows one writer at a time.

TCP client messages are redirected to an interpreter which parses the requests and executes the necessary commands. These include accessing the system for log entries, changing flags in the TCP session or writing messages to the Serial port.

With regard to error cases, we attempted to implement proper exception handling, at least to exit the program gracefully. Boost.Asio typically resorts to error flags instead, but their usage ends up being similar.

We used the example client provided in the official Boost.Asio website [15] in order to test the server correctness. In order to perform automated tests we also developed a simple test client[16] that follows a linear script with a list of commands, each followed by a specified delay.

We should mention that we provide a `makefile` to compile the server application. However the latter does not compile the I^{2}C sniffer program, as it is platform specific (although it can be compiled normally if the `pigpio` library is installed in the system).

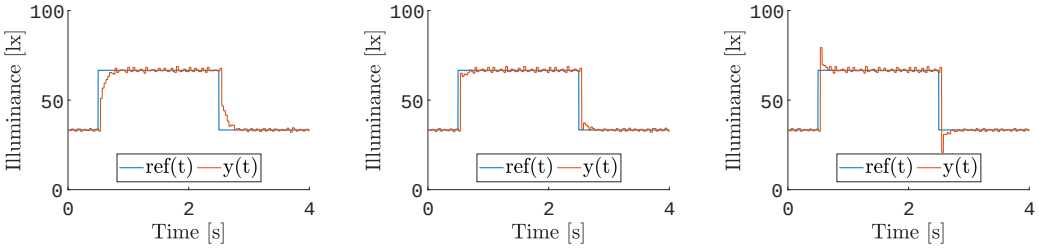
[15] `http://www.boost.org/doc/libs/1_45_0/doc/html/boost_asio/example/timeouts/async_tcp_client.cpp`, as of January 10, 2018.

[16] : Test client, as of January 10, 2018.

6 Experiments and Results

6.1 Local controller features

In this subsection we tested each of the local controller features. We started by testing the impact of adding the feed-forward controller and fine-tuning the value of the corresponding gain K_F, as seen in Figure 14.

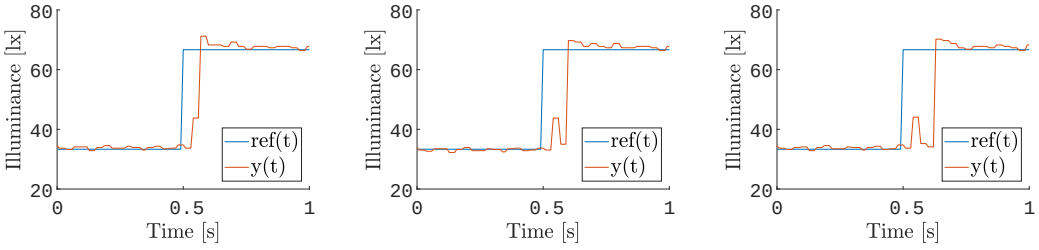


(a) No feed-forward ($K_F = 0$) (b) $K_F = 0.5$ (c) $K_F = 1.0$

Figure 14: Feed-forward gain fine-tuning

The purpose of adding a feed-forward term is to decrease the rise time and improve the controller response time. This causes the overshoot to increase as expected to 21.12% for $K_F = 1.0$. However, the response is much faster as we can observe comparing Figures 14a and 14b.

In order to maximise the efficiency of the feed-forward controller, we could delay the reference signal fed to the PI controller for a small number of samples. In doing so, the feed-forward controller is given a time window to respond to abrupt changes in the reference, before the PI controller error starts to increase. This should reduce the overshoot in the controller response to sudden reference shifts. We implemented and tested this feature, but found the results to be underwhelming. The results are shown in Figure 15, for a variable amount of delayed samples.



(a) 1-sample delay (b) 2-samples delay (c) 3-samples delay

Figure 15: Reference input n-samples delay

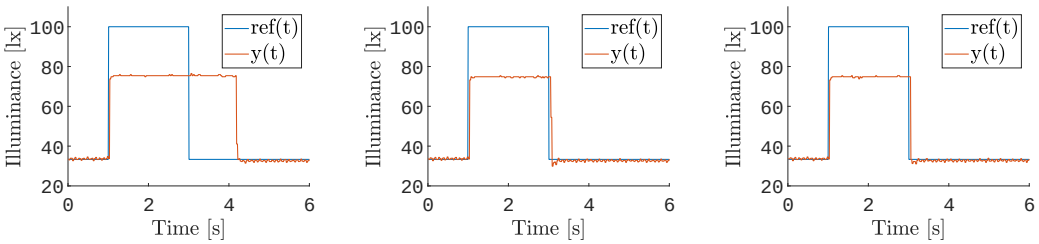
The overshoot is indeed reduced, but the rise time is worsened. Furthermore, if we delay more than a single sample, the PI controller will see the feed-forward action as a disturbance and react to it, which causes the particular behaviour seen in Figures 15b and 15c.

In order to test the anti-windup feature we can just set the reference to an unattainable value for some period of time and then revert it back to some value the controller can reach. The results of these tests are depicted in Figure 16.

Notice the substantial delay in Figure 16a as the controller takes a full second to respond to the reference change. We can observe the impact of the anti-windup mechanism is undeniable. Comparing the two anti-windup implementations we can see that indeed the feedback version is a few samples faster to respond (although it is an almost negligible improvement).

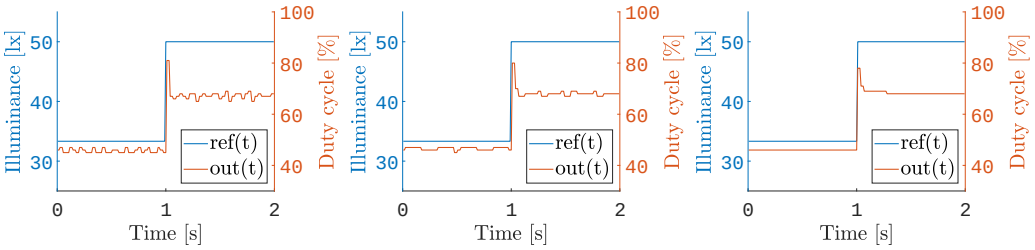
Finally, we wish to evaluate the error deadzone in order to stabilise the output and prevent a flickering behaviour. We test the controller's response to a step input with varying error deadband limits, as seen in Figure 17.

The error deadzone is quite effective as the output jitter is suppressed entirely for a deadband between $[-1.6, 1.6]$ lx.



(a) No anti-windup (b) Simple anti-windup (c) Feedback anti-windup

Figure 16: Anti-windup performance



(a) No error deadzone (b) $err_{\max} = -err_{\min} = 0.5$ (c) $err_{\max} = -err_{\min} = 1.6$

Figure 17: Error deadzone performance for varying error deadband

Even though our controller had satisfactory performance, we suggest improvements that were not implemented mainly due to time constraints. We could have implemented a software low-pass filter, in order to minimise the input error. However, the simpler error deadzone approach already provided sufficient robustness to the system. Be that as it may, the deadzone module could have been further improved by forcing a hysteretic behaviour. This helps preventing flickering behaviour at the limits of the error deadzone.

6.2 System evaluation

Our objective now is to quantitatively evaluate the performance of the projected distributed controller. Moreover we wish to compare it with the local controller, acting without any inter-node synchronisation. For this, we must first define objective metrics.

6.2.1 Evaluation metrics

In order to assess the performance of the developed system we use four different metrics

1. Instant power P,
2. Accumulated energy consumption E,
3. Comfort error C_{err},
4. Comfort variance C_{var}.

The instant power is assumed to be nominal, i.e, has the value 1 W for the maximum led intensity, thus coinciding with a node's duty cycle in percentage. The energy consumption is merely the integral of the instant power over time. Therefore, we can approximate it with Equation 18,

$$E = \sum_{i=2}^{S} d_{i-1} \left(t_{i-1} - t_i \right) \quad \text{[J]}. \tag{18}$$

where S is the total amount of control samples, t_i the sample time stamp and d_i is the normalised led duty cycle. Whereas the aforementioned criteria are concrete and simple to formulate, the comfort is no so straightforward to model.

The comfort error is defined as the average error between the reference and measured illuminance for the periods of time during which the output of the system is **below** the reference, as follows

$$C_{err} = \frac{1}{S}\sum_{i=1}^{S} \max\left(l_{ref}(t_i) - l_{meas}(t_i), 0\right) \quad [\text{lx}]. \tag{19}$$

This metric could be updated in order to evaluate whether the system fulfils the required specification instead of the fitness of the controller by replacing the controller reference with the illuminance lower bound.

Finally, the system should minimise flickering, i.e. abrupt variation in the output for a constant reference. This criterion is measured with the comfort variance, which is defined as the absolute average variation of illuminance during periods of constant occupation. It can be calculated by approximating the second derivative of the measured illuminance, using second order finite differences

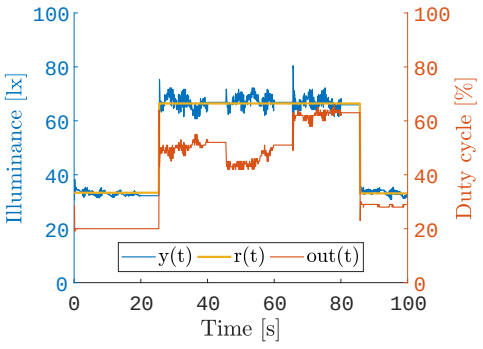
$$f''(x) \approx \frac{\Delta_h^2[f](x)}{h^2} = \frac{f(x+h) - 2f(x) + f(x-h)}{h^2}.$$

Thus, assuming a constant sample time of T_S, the comfort variance is computed as

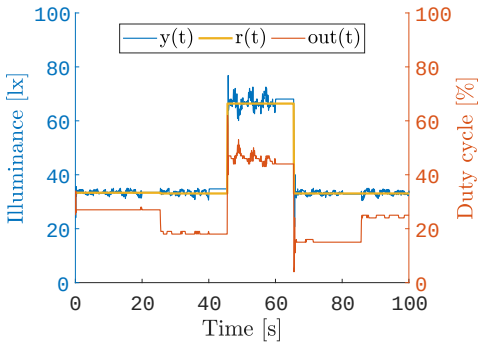
$$C_{var} = \frac{1}{S}\frac{\sum_{i=3}^{S} |l_{meas}(t_i) - 2l_{meas}(t_{i-1}) + l_{meas}(t_{i-2})|}{{T_S}^2} \quad [\text{lx}/s^2]. \tag{20}$$

6.2.2 Local and Distributed Controllers Comparison

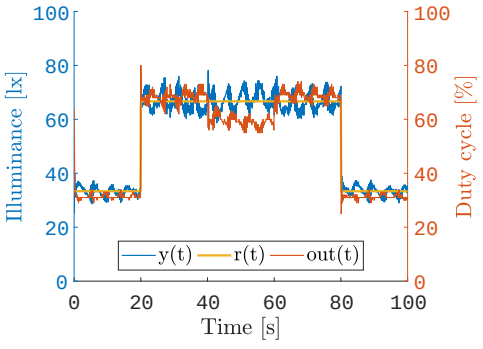
In order to compare the performance of the distributed and non-distributed controller, we designed a simple experiment. In both settings we issue commands to change the occupancy of each desk at given times. We denote the occupancy at node i as occ_i. Our experiment starts with $occ_0 = occ_1 = 0$. Then we issue the following command sequence $occ_0 = 1$, $occ_1 = 1$. $occ_1 = 0$ and $occ_0 = 0$, with a time interval of 20 seconds in-between, thus exhausting every possible combination in the two-node scenario. The experimental results of this comparison are presented in Figure 18.



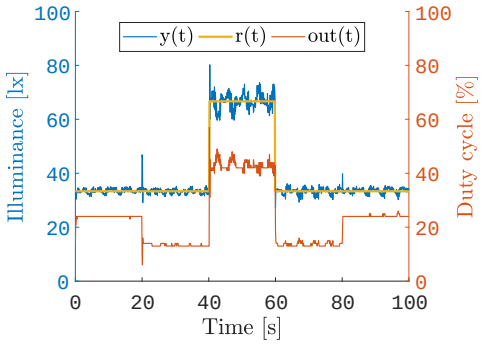
(a) Node 0 with distributed control



(b) Node 1 with distributed control



(c) Node 0 without distributed control

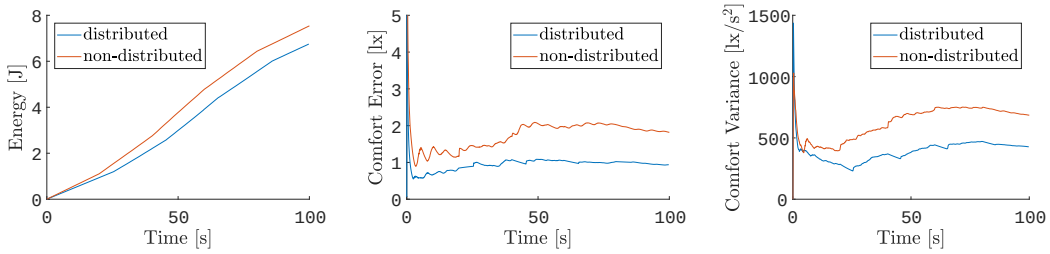


(d) Node 1 without distributed control

Figure 18: Distributed and non-distributed control comparison

At first glance it is possible to notice that the non-distributed controller struggles harder to follow the reference, mainly in node 0. Notice that the distributed controller takes longer to update the reference. This delay corresponds to the execution of the consensus algorithm, during which the controller is frozen. It is possible to verify that the obtained values for the reference corresponds to the optimal solution calculated in subsubsection 3.3.1.

We then apply the proposed performance metric to these results. The comparison of the values of each metric is shown in Figure 19.



(a) Energy consumption metric comparison (lower is better)

(b) Comfort error metric comparison (lower is better)

(c) Comfort variance metric comparison (lower is better)

Figure 19: Distributed and non-distributed control comparison using proposed metrics

It is possible to observe that the distributed controller outperforms the non-distributed for every metric, attaining simultaneously lower energy consumption and higher comfort. We can see that it has an average comfort error of 0.9522 lx which seems to be a fair result given the range of values the controller operates on, and the existing sensor noise.

However, our system still exhibits a high value for the comfort variance. In any case, we have observed that the input signal itself displays a high variance even for a constant LED intensity output. As stated before, this could probably be improved by adequately filtering the intput, or even replacing the electronic components. On the other hand, whenever the controller abruptly changes reference the controller reacts too rapidly, which results in overshoot. Note that the PI controller parameters were fine-tuned in the individual node scenario, and should have been adapted, so as to mitigate this kind of behaviour.

Essentially, even though we produced a calibration routine, the chosen characteristic for the LDR was arbitrarily chosen provided it met the manufacturer's specification. Perhaps an informed choice could improve the sensor measurements and even allow a quick and robust calibration of each sensor.

Overall we have projected a system that fulfils the requirements, and observed that a distributed approach is advantageous when comparing to a disconnected local controller.

7 Conclusion

We have succeeded in developing an efficient distributed real-time controller system in order to automate lightning in a small scale prototype of an indoors office environment. Our distributed controller outperforms the local approach, by leveraging the knowledge of the system obtained in the calibration routine and communicating in order to reach the global optimum. We have performed tests that confirm these claims, and evaluate the proposed energy consumption and comfort metrics, obtaining favourable results.

However, the system could be further improved. The two main obstacles consist of communication delays and calibration issues. The former result in a bottleneck to the amount of consensus iterations and number of nodes n in the system, as the absence of a broadcast message effectively means that communications increase with $\mathcal{O}(n^2)$. The system performance can certainly be improved by fine-tuning each parameter in the multiple node scenario. However, this is a very laborious process and our work is merely a proof of concept, resorting to a physical prototype, as opposed to a real-world office.

External Links

The project code is publicly available on GitHub, and the respective documentation is currently hosted in IST's Sigma cluster, but should be eventually migrated to GitHub as well. Furthermore we provide a (short) video demonstration of a trial which involves applying a disturbance to the system.

Project repository Documentation Live Demo

References

[1] S. H. Lee and J. K. Kwon. Distributed dimming control for led lighting. *Optics express*, 21 (106):A917–A932, 2013.

[2] R. Glowinski and A. Marroco. Sur l'approximation, par éléments finis d'ordre un, et la résolution, par pénalisation-dualité d'une classe de problèmes de dirichlet non linéaires. *Revue française d'automatique, informatique, recherche opérationnelle. Analyse numérique*, 9(R2): 41–76, 1975.

[3] S. Boyd, N. Parikh, E. Chu, B. Peleato, and J. Eckstein. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Foundations and Trends® in Machine Learning*, 3(1):13–24, 2011.