Distributed Real-Time Control Systems
Mid-term report

Murillo Hierocles A. S. Teixeira (ist1105038)

24 de março de 2023

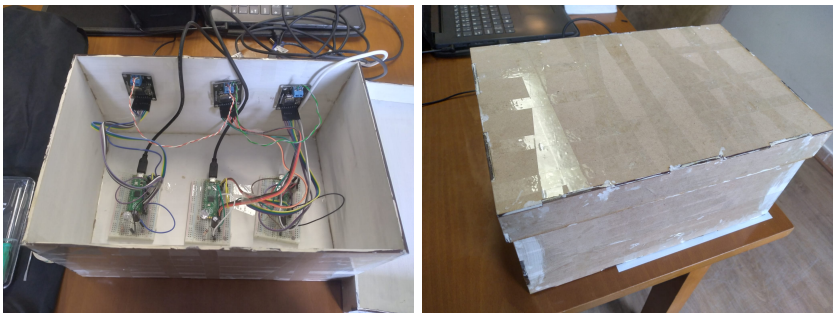
1 Motivation

The objective of this project is to build a prototype of an intelligent lighting system, consisting of 3 luminaries, that, using real-time distributed control concepts, manages the brightness of workspaces according to demand.

2 Prototype construction

For the construction of the prototype, a box in MDF with dimensions of 30cm×20cm×15cm was assembled to simulate an office-like scenario, which can be seen in Fig. 1.

In Fig. 2 it is possible to see in more detail the electronic components that make up the 3 luminaries. Among them, the main ones are the microcontroller (Raspberry Pi Pico) used to define the program's operating logic and interface with the PC, the CAN Bus modules, responsible for communication between microcontrollers, the LEDs, which work as actuators in this control system, and the photoresistors, the light sensors that allow measuring the ambient light.



(a) Prototype with open lid (b) Prototype with lid closed

Fig. 1

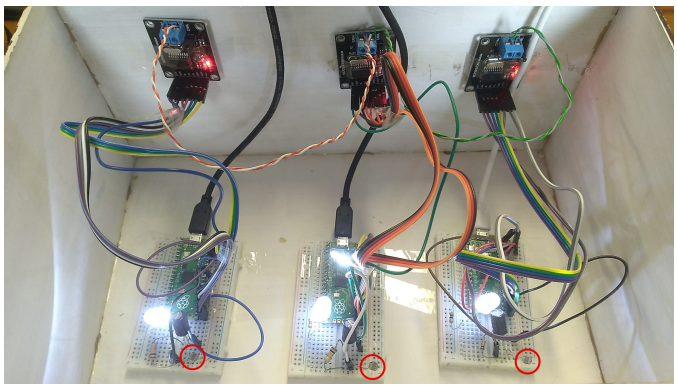


Fig. 2: Circuit with Raspberry Pi Pico microcontrollers, CAN Bus Modules, Leds, photoresistors (highlighted in red) and other accessories.

3 Signal acquisition

For the control project it is important to have the ambient illuminance levels. To obtain this data, the voltage across the photoresistor was sampled and the ADC value was converted to the corresponding illumination value, using the Code 1.

```
float sensor_value_to_lux(int sensor_value)
{
    V_adc = 3.3 * sensorValue / DAC_RANGE;
    R_ldr = 10000 * (3.3 - V_adc) / V_adc;
    lux = pow(pow(10, 6.15) / R_ldr, 1.25);
    return lux;
}
```

Code 1: Code to convert the ADC value to illuminance (lux)

In order to set the sampling period to 100Hz in a non-blocking way and to minimize jitter, one of the Raspberry Pi Pico Timers was used.

In Fig. 3 it is possible to see that this approach resulted in a sampling period without jitter (given that the points are exactly above the multiples of 10ms). Calculations with a large number of sample points indicate that there is no jitter present in the sampling.

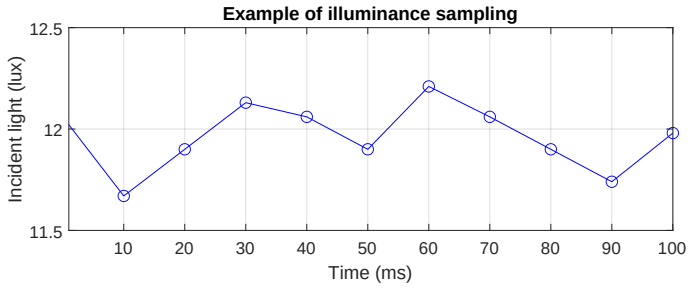


Fig. 3: Example of illuminance sampling using the Raspberry Pi Pico Timer

4 Illumination control

In this first phase of the project, each luminaire controls the lighting independently, using a PID controller with Bumpless transfer, Anti-Windup with back calculation and feedforward. Controller parameters for each luminair have been tuned using trial and error.

4.1 Steady state

In the steady state, it is possible to see, in Figs. 4 and 5, which all luminaries have the expected behavior: illuminance following the reference and a fixed duty cycle. The different duty cycle values can be explained by the fact that, if a luminaire responds in a faster, the next one to turn on needs less light, so it will reach a lower duty cycle.

4.2 Step response

To understand the dynamic behavior of the system, the luminaires were tested for responses to reference steps and external disturbances.

It is possible to see in Fig. 6 that the step response is fast and without overshoots, as requested in the specifications. It is also possible to visualize that the systems respond well to external disturbances (configured in the situation by increasing the reference in adjacent luminaires), quickly and without oscillations.

In Fig. 7 it is possible to understand how the duty cycles of each luminaire were adjusted to carry out the control. Here, it is possible to notice that in the control carried out there is no flickering, given the absence of duty cycle oscillations.

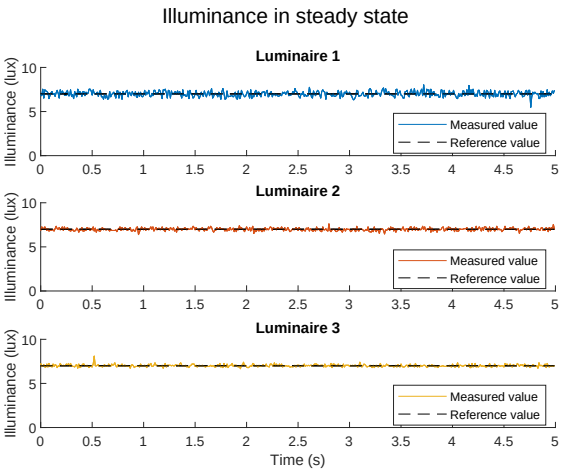


Fig. 4: Illuminance in steady state

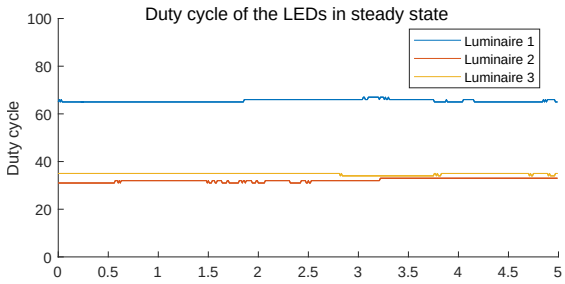


Fig. 5: Duty cycle of the LEDs in steady state

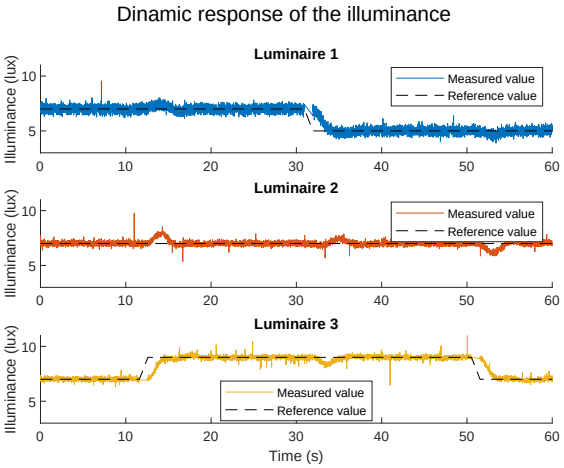


Fig. 6: Dinamic response of the illuminance

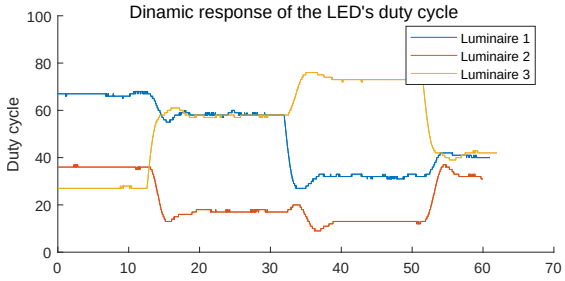


Fig. 7: Dinamic response of the duty cycle

To understand the anti-windup effect, two other simulations were performed, shown in Fig. 8 and Fig. 9, where the box was opened, so that a large uncontrollable error was entered. It is noticeable there that windup is a big problem in these situations, since the return to the normal level took considerably longer in the case without anti-windup.

4.3 Controller class

To implement the control, the `pid` class was created, responsible for instantiating the PID controller.

In the Code 2 it is possible to see the file `pid.h`, with the signatures of the controller functions, as well as the function, defined inline, `pid::housekeep`, responsible for the update of the integrator. The functions `pid::get_anti_windup_status`, `pid::set_anti_windup_status`, `pid::get_feedforward_status` and `pid::set_feedforward_status` are responsible for real-time user interaction with the controller, which can change the presence of the controller's feedforward and anti windup mechanisms.

The function `pid::compute_control` is where, in fact, the calculation of the control takes place, as shown in Code 4. In general, the control takes place from 3 terms: proportional, derivative and integral. The integral is calculated in the `pid::housekeep` function, but the rest are calculated here. In this implementation, the `u` output can admit values between 0 and 255, since the controller acts on a PWM signal.

Another relevant observation is that the system is prepared for changing parameters during execution, with the implementation of bumpless transfer, whose logic is present in Code **??** and in lines 10 to 14 of Code 4.

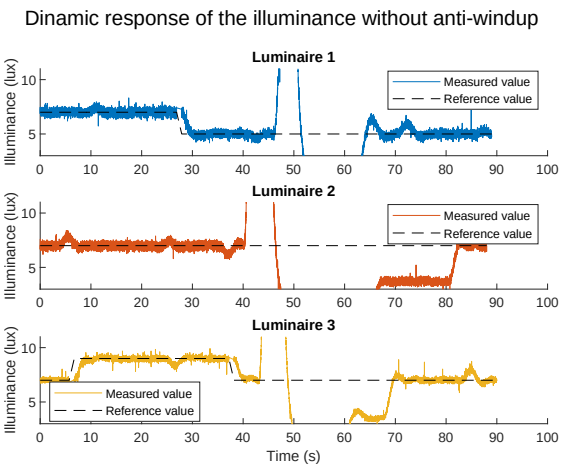


Fig. 8: Dinamic response of the illuminance without anti-windup

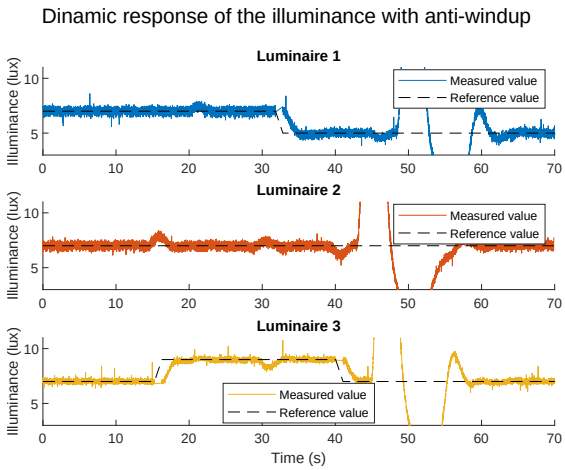


Fig. 9: Dinamic response of the illuminance with anti-windup

```
#ifndef PID_H
#define PID_H

class pid
{
public:
  float I, D, K, Ti, Td, b, h, y_old, N, Tt, es, K_old, b_old;
  bool in_transition, anti_windup_status, feedforward_status;
  explicit pid(float h_, float K_, float b_, float Ti_, float Td_, float N_, float Tt_);
  ~pid(){};
  float compute_control(float r, float y);
  void change_parameters(float K_new, float b_new);
  void set_anti_windup_status(bool b);
  bool get_anti_windup_status();
  void set_feedforward_status(bool b);
  bool get_feedforward_status();
  void housekeep(float r, float y);
};

inline void pid::housekeep(float r, float y)
{
  float e = r - y;
  // Updating the integrator with
  // anti-windup (back calculation)
  I += K * h / Ti * e + anti_windup_status * h / Tt * es;
  y_old = y;
}

#endif // PID_H
```

Code 2: File `pid.h`, that implements the PID controller used in the luminaries.

```
float pid::compute_control(float r, float y)
{
  // Proportional term
  float P = K * ((feedforward_status * (b - 1) + 1) * r - y);
  float ad = Td / (Td + N * h);
  float bd = Td * K * N / (Td + N * h);

  // Procedure to update the integrator
  // to create bumpless transfer
  if (in_transition)
  {
    I = I + K_old * (b_old * r - y) - K * (b * r - y);
    in_transition = false;
  }
  // Derivative term
  D = ad * D - bd * (y - y_old);
  float v = P + I + D;
  float u = v;
  if (v < 0) u = 0;
  if (v > 255) u = 255;
  es = u - v;
  return u;
}
```

Code 3: Implementation of the method `compute_control`, that calculates the control output

```
void pid::change_parameters(float K_new, float b_new)
{
  K_old = K;
  b_old = b;
  K = K_new;
  b = b_new;
  in_transition = true;
}
```

Code 4: Implementation of the method `compute_control`, that deals with changes in parameters in a bumpless way during control

5 Code structure

With the control solution already defined, it is now relevant to discuss how the implementation of the functionalities in the microcontrollers took place.

Both Raspberry Pi Pico Cores were used. `Core0` was responsible for dealing with the serial communication with the user (see Section 6), controlling the microcontroller's state machine and controlling the lighting. The state machine, in this first phase of the report, is implemented in a simple way, with just two states: `AUTO`, in which the controller is on and the user has no control over the duty cycle value of the LEDs, and `MANUAL`, where the user can manually set the duty cycle value. The alternation between these two states can be performed using the serial command `k 0` (to activate the `MANUAL` state) and `k 1` (to activate the `AUTO` state).

`Core1` was responsible only for CAN communication. In the current implementation, the user can send messages by command `m [val]`, which sends the message `[val]` on the CAN line. The other microcontrollers connected to the CAN line are listening at all times and, as soon as they receive a new message, they print the information about it on the serial.

5.1 File structure

In order to organize the workflow, the code is divided into some packages and files, described in Tab. 1.

File	Description
`Project.ino`	Arduino file with declaration of variables, the main setups and loops of each core.
`pid.h` and `pid.cpp`	Header and implementation of the custom PID controller described in Section 4.3
`buffered_data_struct.h`	Custom data struct to store the buffered data.
`circular_buffer.h`	Class created to store the last minute buffer. It stores `buffered_data_struct` structs.
`helpers_can.ino`	File with custom methods for dealing and printing CAN errors.
`helpers_interface.ino`	File with methods used to create the user interface
`can.h`	Header of the can module
`mcp2515.h` and `mcp2515.cpp`	MCP2515 library (module used to control the CAN Bus)

Tab. 1: Descriptions of the files used in the project

6 User interface

The user interface was implemented using simple serial commands, as suggested in the design. All the commands listed for the first step in the Appendix of the Lab Guide were implemented - sometimes with some changes, when instructed by the professor in charge (as is the example of the calculation of energy metrics, visibility error and flickering error, which are being calculated from the last minute buffer).

However, new commands were created to facilitate the interaction and debug of the luminaire. These new commands are found in Tab. 2.

Command	Description
`m <val>`	Send the message `<val>` to the CAN Bus
`s a`	Start the streaming of the following: time, duty cycle, reference, illuminance
`S a`	Stops the streaming started by `s a`

Tab. 2: New commands of the user interface

7 Processing times

In order to evaluate the performance of the code, some average processing times were surveyed and can be seen in Tab. 3. In general, the code's routine methods show great performance, with operations always in the microsecond range. As, except for small exceptions, the code is always open to interruptions, with few situations in which `noInterrupt()` is used, an extremely efficient code does not become such a big differential, since periodic tasks can interrupt longer processes. However, as the second part of the project involves synchronization operations between cores and between microcontrollers, this could be a differential in the future.

Process	Time (μs)
Calculate control	35
Signal acquisition	63
CAN Comunication	2
Listening serial	5
Simple serial command (change reference)	984383
Run state machine	1
Core0 initialization	277
Core1 initialization	14744

Tab. 3: Processing times for different operations