Università degli Studi di Padova

Dipartimento di Ingegneria dell'Informazione

Scuola di Ingegneria
Corso di Laurea Magistrale in Ingegneria dell'Automazione

TESI DI LAUREA

IDENTIFICATION AND CONTROL OF AN ILLUMINATION SYSTEM

Autore:
Enrico Franzon

Relatore:
Prof. Maria Elena Valcher
Co-Relatore:
Prof. Ignacio De La Nuez Pestana

Padova, 20 Aprile 2015

ANNO ACCADEMICO 2014/2015

Contents

1 Introduction **4**

2 The experimental device **5**
- 2.1 PRODEL . . . 6
- 2.2 Arduino UNO board . . . 8
- 2.3 Base and circuit . . . 10
- 2.4 Oscilloscope . . . 12
- 2.5 Waveforms generator . . . 13
- 2.6 PC and circuit connections . . . 14

3 A/D conversion **15**

4 System Identification **20**
- 4.1 Identification by using a first-order model . . . 21
- 4.2 Identification by using a second-order models . . . 31
- 4.3 Identification by using a second-order models and delay . . . 34

5 First PID controller **39**
- 5.1 Simulink model . . . 45

6 PID controller improvement **49**

7 Conclusions **54**

A Appendix **55**
- A.1 Arduino code . . . 55
- A.2 Processing code . . . 57
- A.3 Matlab code . . . 59
  - A.3.1 Identification by using a first-order model . . . 59
  - A.3.2 J 1 . . . 62
  - A.3.3 Identification by using a second-order models . . . 63
  - A.3.4 J 1 dos . . . 66
  - A.3.5 Identification by using a second-order models and delay 67
  - A.3.6 J 1 Delta 1 dos . . . 69
  - A.3.7 PID Simulation . . . 70
  - A.3.8 PID frequency study . . . 72

B Bibliography **75**

C Webliography **75**

1 Introduction

Modern world technology has done remarkable progresses in the last decade as far as the world of automation and of control systems of different nature are concerned. A big contribute came from the introduction of controllers in the automated systems as the PID controller that thanks to the combination of three actions (Proportional, Integral and Derivative) has solved a lot of control problems for systems of various kinds.

In addition to that, in the last decade others devices were created that are used as a part of a general control device. One of them is Arduino, an easy to use open-source electronic platform, based on flexible hardware and software, that is having a good success.
The Arduino board can interact with the surrounding environment by collecting information from a big variety of sensors, controlling lights, engines and others actuators.

This thesis regards the identification of the transfer function of an electronic system device that allows to measure the luminous intensity that sorrounds it; the system to identify is principally composed of a voltage generator and a light sensor that displays the light intensity received from the generator.
An Arduino board is used to save analog signals taken from the voltage generator and from the light sensor that are necessary to the identification process.

The main problem of the light sensor are the disturbances from the surrounding that offsetting the read values, so a controller must be implemented to ensure the correct reading of the light sensor. Let us start from the system description.

2 The experimental device

The experimental device consists of a set of few electronic components and a PC for the elaboration of the data and all calculations; in particular, the system is composed of:

1. PRODEL that is a set of 6 electronic devices that are: the voltage source, a connector that allows the voltage to pass from the generator to the PID controller, the PID controller of the system, the light sensor, a disturbance generator to possibly add a voltage noise, a power supply;
2. an Arduino UNO board for the input and output storage;
3. a base where the electronic circuit with resistances is realized to limit the voltage values taken from the generator that is required not to damage the Arduino board;
4. a function generator necessary for the system identification;
5. an oscilloscope to visualize the functions supplied by the waveforms generator and by the system output;
6. a PC to elaborate and to realize the programs, for data storage and for the system identification and calculation of the appropriate parameters for the PID controller.

2.1 PRODEL

PRODEL is composed by 6 electronic pieces which are linked together and it can be seen in the figure below:

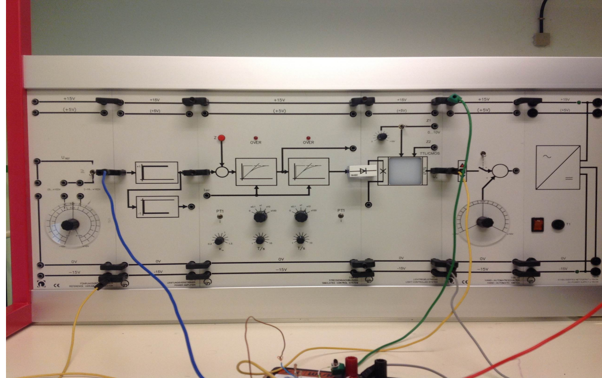

Figure 1: Electronic components.

In particular, starting from the left of this set of devices, the apparatus is composed of:

1. a voltage generator whose voltage value can be set by operating in two different ways: looking to the surface voltage generator, there is one disk that presents two rings; the bigger ring on the surface of the generator provides voltages in the range between -10 $[Volt]$ and 10 $[Volt]$, while the smaller ring provides voltages in the range between 0 $[Volt]$ and 10 $[Volt]$;

2. an electronic device that connects the voltage generator to the light sensor;

3. a PID controller acting on the error that exist between the input and output signal of the sensor light that will be used after having identified the system to compensate the aforementioned error;

4. a light sensor to measure the voltage supplied by the voltage generator. In fact this sensor contains a voltage transducer that arrives in input

and it is shown on the panel of the aforesaid device; in addition it also contains a voltage generator in the range between 0 [$Volt$] and 10 [$Volt$]. This generator can be used to add an electronic signal that simulates external noises to the light sensor.

In between the electronic component that connects the voltage generator to the light sensor it is necessary to insert a junction diode; this electronic component starts conducting when the applied voltage is bigger than the threshold voltage V_S; this value depends on the materials that compose the diode but for a silicon diode V_S is approximately 0,6 [$Volt$].
It is always necessary to limit the current that passes through the diode to guarantee that it does not exceed the maximum value settled for it, otherwise it could break the electronic device.
To ensure to close the circuit that is necessary to avoid damages to the circuit, a metal connector has been inserted in between the diode output and input.

During the data acquisition of the input and output signals of the sensor, the PID controller is not inserted in the circuit because the identification of the system transfer function must be realized without it; after having found a mathematical description for the sensor light, it will be necessary to add the PID in between the voltage generator and the light sensor. In this way the PID controller is able to correct the error that exists between the sensor input and output.

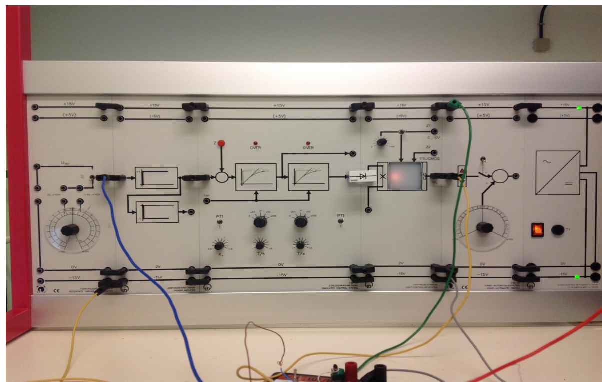

Figure 2: Electronic components with PID controller.

2.2 Arduino UNO board

The acquisition of the input and output signals from the sensor light is possible by using the Arduino UNO board, a stand-alone device that has wide application in the industrial sector; this is first due to its lower price compared to other control systems used nowadays in the technology field, and second due to the similarity of its programming language to Java and C, so diffused in the industries.

Arduino is an open-source electronic platform based on flexible hardware and software. The Arduino board is able to interact with the surroundings and receives informations from a variety of sensors controlling lights, engines and actuators.
One of the most important aspects of this device is that developed programs can be stand-alone which means they can run on this device without having to use an external computer for the calculations; furthermore this board can communicate with the software that runs on the computer, for instance Flash or Processing.

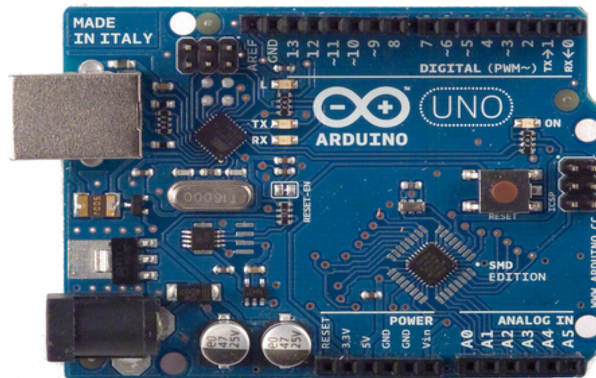

Figure 3: Arduino UNO board.

Arduino UNO has 14 input/output pins: 6 of them are used as PWM outputs, 6 as analog inputs, one is a USB connection, then there is a power jack and a reset button. It contains all that is needed for supporting the

microcontroller; it can be easly powered from the computer through a USB cable, or through an AC to DC adapter, or through a battery.
As far as the program language of the Arduino board is concerned, the developers, starting from the C and $C{+}{+}$ languages, have defined a similar but easier language.

To realize the system identification, the voltage generator and the light sensor values must be saved and those signals can be taken from a serial port of the PC; in fact, the analog data acquisition is performed by using the serial port of the PC, by using another program studied by the Arduino board developers called Processing.
Processing is a program language that allows to develop different applications such as games, animations and interactive contents. Processing has the same syntax, commands and program paradigms object-oriented of the Java language and in addition it offers a lot of high level functions to easly manage the graphic aspects and multimedia.
Since Arduino does not allow to save analog data on a file, it is necessary to use Processing to that purpose.

2.3 Base and circuit

The acquisition of analog data must be done carefully. Indeed, the Arduino board can acquire only voltage signals lower than +5 $[Volt]$: over this value the board could burn and the device would not work anymore.
The voltage generator supplies voltage at a maximum value of 10 $[Volt]$, so an electronic circuit must be realized using resistances to limit the acquired voltage at +5 $[Volt]$.
The electronic circuit is illustrated in Figure 4 below:

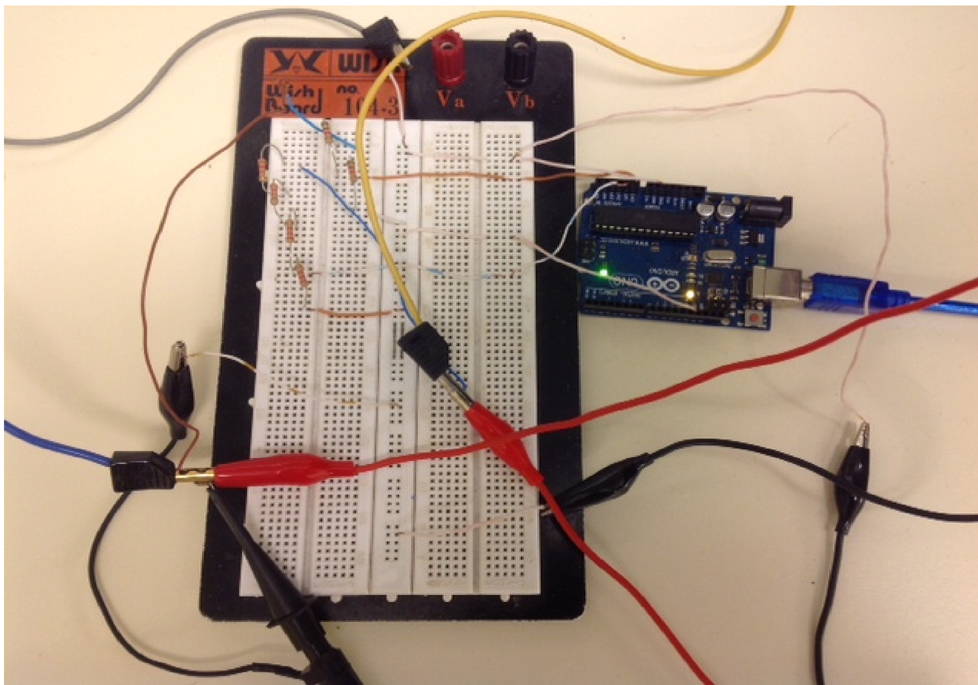

Figure 4: Electronic circuit for the analog signals acquisition.

Two resistances of the same value must be taken and connected in series for the input acquisition. In this way, the voltage drop on the first resistance is the same as on the second one; therefore this halves the maximum voltage value from +10 $[Volt]$ to +5 $[Volt]$, and hence it does not damage the Arduino board. Then, during the data storage, it must be remembered that the

acquired values from the voltage generator ad from the light sensor are the half of the real generated ones.
The voltage generator value has to be taken from the middle of the two resistances with a bridge that links it, with a copper cable, to the A0 analog input pin of the Arduino board.

With regards to the acquisition of the values read in the light sensor, it is possible to verify that the maximum value that the light sensor reveals is higher than $12[Volt]$ and this damages the Arduino board; this comes from the fact that the light sensor is subject to external disturbances of different nature such as the surrounding lights. The solution is to use 4 resistances of the same value connected in series and then acquire the signal from the last resistance. In this way the acquired values are a quarter of the real ones.

2.4 Oscilloscope

Using the oscilloscope, an electronic measurement device, it is possible to display the time trend of electronic signals such as voltages and currents. The $CH1$ channel is linked with a probe to the waveforms generator, while the $CH2$ channel is linked to the sensor light output.

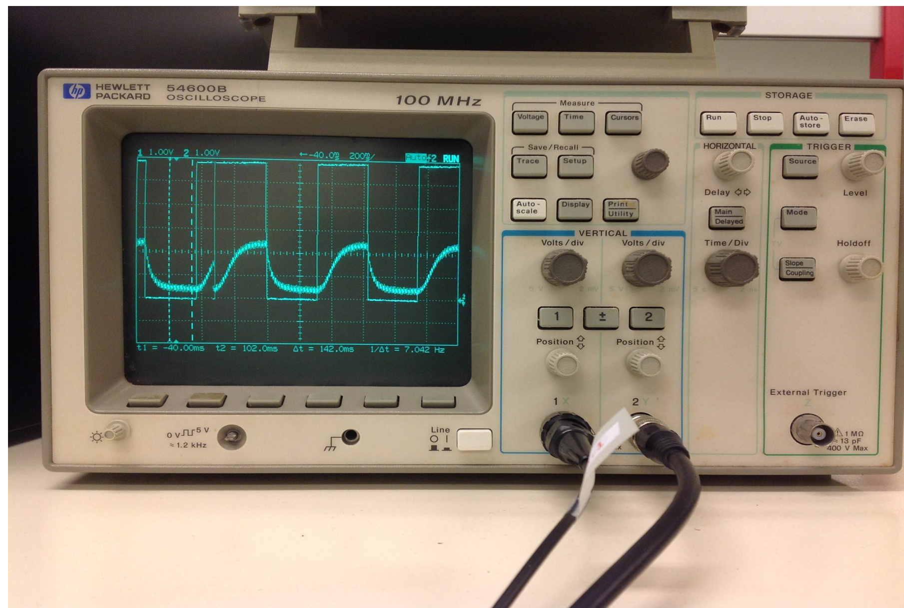

Figure 5: Oscilloscope.

2.5 Waveforms generator

A waveforms generator or functions generator is an electronic device that produces different signals of different shapes.
It is possible to change the parameters of the waveform such as the frequency, the amplitude and the possible duty cycle of the signal.
For the identification purposes, a square waveform is used, with a frequency equal to the minimum value produced by the generator, because it is necessary to obtain a lot of signal samples, as the Arduino board takes data at most at 115200 bps.

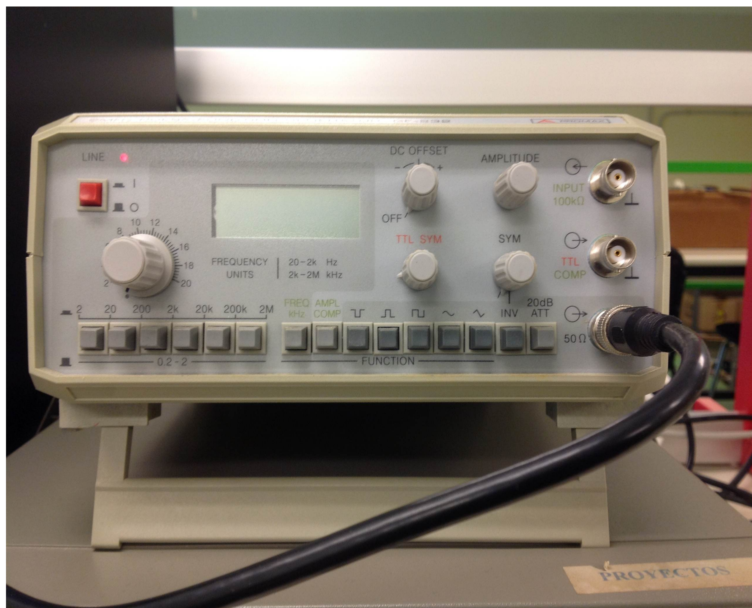

Figure 6: Waveform generator.

2.6 PC and circuit connections

The use of the computer is necessary to do all the calculations and to allow signal exchanges with the Arduino board. The system identification and the controller realization are obtained through Matlab, an elaboration software that provides a lot of important functions for the system management.
After having connected all the resistances as explained above, it is possible to connect the waveforms generator to the system input and to the A0 pin of the Arduino board, because the waveform must be stored for the subsequent elaboration; the A1 pin is used to save the information given by the light sensor.
Using the two oscilloscope channels, the input and output signals are shown on the oscilloscope screen. Finally, the Arduino board is connected to the PC through the USB port.

3 A/D conversion

The first step for the system identification is the signal storage; data have been taken by using the Arduino board that realizes an A/D conversion. Arduino in fact can take data by selecting the acquisition rate as previous explained; thus signals are saved by storing their samples and this process allows the conversion from analog to the corresponding digital signal.
The signals acquisition is possible thanks to two specific programs; first of all data are shown on the screen through the serial port using the Arduino software, while data storage is realized using another program called Processing.

As previously mentioned, the acquisition rate has been set to the highest possible bitrate compatible with the Arduino board, in order to obtain the maximum number of samples, and hence to perform a good identification.

```
/* Global variables definition */
String tiem;
String entr;
String sal;
String video;

/* Program's part for the signals print on the screen */
void setup()
{
        /* Bitrate per second speed is set at the biggest
           value */
        Serial.begin(115200);
```

Once the acquisition rate is chosen, data acquisition can start by creating a loop where data are taken by using the Arduino pins; the code inside the loop has been repeated 1500 times; this value was chosen sufficiently large to ensure the storage of a sufficiently high number of wave front edges of the input waveform provided by the waveform generator. In this case, the waveform generator generates a square waveform with a period set to $0.5[Hz]$; the half period of the waveform is $1[s]$.
After having realized some tests, it turns out that 300 samples represent half

of a period of the waveform. Starting from the previous results it follows that the sample time is: $T_{sample} = 1/300 = 0.0033[s]$.

```
for(int i=0; i < 1500; i++)
{
        /* Variables setting */
    tiem=String("\t");
    entr=String("\t");
    sal=String("\r\n");

    /* Sample's number */
    tiem=i+tiem;

    /* Signal input's sample */
    entr=analogRead(A0)+entr;

    /* Signal output sample */
    sal=analogRead(A1)*2+sal;

    /* String concatenation */
    video=String(tiem+entr+sal);

    /* String print */
    Serial.print(video);
}
```

The three String variables *tiem*, *entr* and *sal* must be initialized; using the Arduino program in fact it is not possible to realize the usual string concatenation as in Java/C. First it is necessary to initialize the three string variables and then to concatenate them with the corresponding acquired value; finally one single string called *video* was created to concatenate all the acquired values.
In this way it is possible to use only one *Serial.print* comand, so that the storage process is faster and this allows to save more data.

In this data saving program the acquisition loop is written in the "*Setup*" part of the program because it is enough to save only one complete wave front edge of the input waveform to identify the system transfer function, but it is not known if data acquisition starts after the starting of a wave front edge

that is not useful because to realize a good identification it is necessary to take a complete wave front edge of the signal, so some waves front edges have been saved and then only the first complete one will be selected; the "*loop*" part of the program that it must be inserted in every Arduino program to ensure the correct functioning of it but in this case it is not used to acquire data and it stays empty, without code. To perform the system identification the sample number is saved together with the input sample taken from the A0 pin and the output sample taken from A1 pin; finally the acquired output samples must be multiplied by 2 to have them in the same scale as the input samples.

Finally Processing is used to save the data on a file called *Data*.

```
/* It is necessary to import the processing.serial library */
import processing.serial.*;

/* Variables's initialization */
Serial mySerial;

 /* This comand serves to print the acquired data on the
    file */
PrintWriter output;
```

In the first part of the program the variables are initialized; the Serial library reads and writes data to and from external devices one byte at a time. It allows two computers to send and receive data. This library has the flexibility to communicate with custom microcontroller devices and to use them as inputs or outputs to Processing programs.
In addition, a PrinterWriter object is created with the purpose of printing to a text-output stream on a specified file. Then, the two above mentioned variables are initialized inside the "*Setup*" part of the program; in this part of the program usually all general variables that can be used in others parts of the program are initialized. "*Setup*" can be seen below:

```
/* Setup is necessary to run the program */
void setup()
{
  /* Selection of serial port and the acquisition speed  */
  mySerial = new Serial(this, Serial.list()[1], 115200);

  /* Creation of the file on the same folder of the Processing
     program */
  output = createWriter("Data.m");
}
```

In the *Setup* part of the program written above, a "*Serial*" object is allocated; this object is used to save data taken from the Serial port of the PC with the specified bitrate between the brackets; the instruction "*output = createWriter("Data.m");*" creates the file "*Data.m*" where the input data are stored.
To be sure to acquire data when the serial port is available and to avoid getting null values, it is necessary to impose some controls on the read data values as shown below.

```
/* This block of the program is realized to control that data
   are available and if they are to save them by taking from
   the serial port */
void draw()
{
  /* If the serial port is ready takes the data */
  if (mySerial.available() > 0)
  {
    /* Reads data to the serial port until new line */
    String value = mySerial.readStringUntil('\n');

    /* If the read value is not null then print it on the Data
       file */
    if( value!= null)
    {
      output.print( value );
    }
  }
}
```

As far as the "*Draw*" part of the program is concerned, when the serial port is available the incoming values are taken until the end of the read line using " *String value = mySerial.readStringUntil(' n');*"; another control is imposed to ensure that the acquired value is not null and in this case the value is written in the "*Data*" file.

```
/* Exit to the program when any button is pressed */
void keyPressed()
{
  /* Writes the remaining data on the file */
  output.flush();

  /* Close the file */
  output.close();

  /* Exit to the program */
  exit();
}
```

The "*keyPressed*" part of the program allows to exit from the storage phase when any button is pushed; for the file to be created correctly two methods call are required: the first one by writing "*output.flush()*" that ensures that everything in the stream of the serial port can be sent to the file, a sort of "cleansing " of the stream, and the second one by writing "*close()*" that closes the file.
When the program runs, the "*Data.m*" file is created and all the sample values are stored in it.

4 System Identification

When the "*Data*" file is ready, it is possible to perform the system identification. The identification is achieved using Matlab; the first step consists in creating the arrays for the acquisition samples time and input/output of the system as well. Only one rising edge is enough to realize the identification because it describes the behaviour of the sensor light to an input signal to it. The file created to do that is called "*Identification by using a first-order model*"; the name chosen refers to the Identification by using a first-order model and this means one zero in the denominator of the system transfer function. This is the first attempt to find out the mathematical description of the system using a single pole transfer function, because it is the simplest description of the system. This function is a rational transfer one with a degree equal to 1.
Furthermore there is the possibility of introducing a delay expressed using an exponential term; this can be used to improve the identification because there is a delay between the light sensor output and input. If the identification achieved in this way is not satisfactory, it is necessary to assume for the system transfer function a two pole function in order to produce better results.

4.1 Identification by using a first-order model

To realize the system identification using a one pole model, the thing to do is to take the storage data from "*Data*" file and organize them to be easly treated.

```
% Data reading from "Data.m" file %
[samples referencias sensorValues]=textread('Data.m','%d%d%d');
```

From the code above, the keyword "*textread*" is used to read data from the specified file, written in brackets, and to convert the strings in the file in double-type values:

```
textread('Data.m','%d%d%d')}.
```

This way three arrays "*samples*", "*referencias*" and "*sensorValues*" have been created: the first one contains the number of the samples, the second one contains the input samples and the last one contains the light sensor output samples.

Then it is necessary to take only a subset of values because the realization of the system identification takes only a rising edge; the data are taken from the first nonzero reference sample.

```
% Array declaration for data treatment %

help_array_tiempo=[];
help_array_refer=[];
help_array_sensor=[];

% Sample time %
```

```
T_sample = 1/300;

% Time array %
tiempo_1=[];

% Input array %
entrada_1=[];

% Output array %
salida_1=[];

 % Array that saves the minimum function %
 % cost with differents delta values %
J_1_delay_vect=[];

% Counter used as controller of the data %
count_1=0;
```

The three arrays specified at the beginning of this part and called "*help array tiempo*", "*help array refer*" and "*help array sensor*" are used to easly manage the data. These arrays permit to take all the rising edge of the waveforms; one sample before the begin of the steep has been taken with the purpose to show better the time course of the light sensor. The zones where reference and sensor values take null values are discarded; other three arrays are created to take, as explained above, the first complete rising edge of the waveform because the identification can be achieved by using only one of those.
The "*J1 delay vect*" is used to save all J_1 optimum values; J_1 is a cost function that represents the error value between the light sensor output and the model output corresponding to the same input reference and it is used to calculate the *Correct Standard Deviation* between those two outputs. It is clear that a smaller Standard Deviation and J_1 value as well ensures a better identification.
J_1 results:

$$J_1 = \sigma = \sqrt{\frac{\sum_{k=1}^{N}(x_{mod} - \bar{x}_{ref})^2}{N-1}}$$

The J_1 function is created apart to be called when it is requested.
The reason for saving the J_1 value comes from the need to compare the different J_1 functions realized with different delay terms, so it is possible to plot all those values in the same graphic and choose the delay value that provides the smallest J_1 value and hence the best identification.

All 1500 data are analized and treated by using a "*for*" loop to retain only the useful samples and to discard the not useful ones:

```
j=1;

% Loop used to treat data stored %
for i=1:1500

    % The firsts data aren't used because the reference %
    % value is zero %
    if((referencias(i)==0)&&(count_1==0))

        count_1=count_1+1;

    end

    % When the reference isn't equal zero data are taken %
    % then is taken one previous sample to have a better %
    % visualization on the screen %
    if((referencias(i)~=0)&&(count_1==1))
        help_array_tiempo(j)=samples(i-1);
        help_array_refer(j)=referencias(i-1);
        help_array_sensor(j)=sensorValues(i-1);
        j=j+1;
    end
```

With the introduction of the *count 1* counter it is possible to control when the first reference sample value is different from 0 for the first time; after having performed the control, this variable value is set to 1 to record that the data acquisition has already started.
After that, another control is imposed to be sure to save in the arrays only samples taken from the rising edge.

In the Figure 7 it is possible to see the input reference taken from the waveforms generator and the system output corresponding to it:

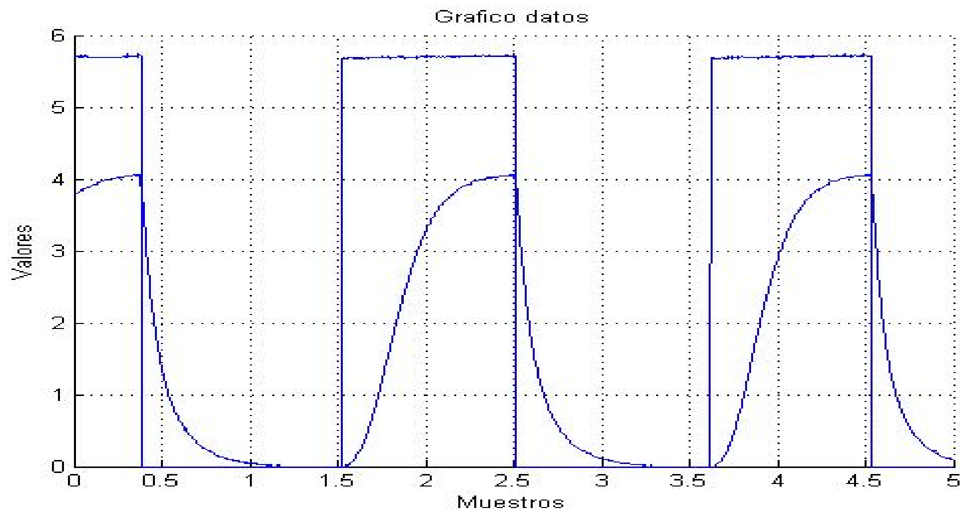

Figure 7: Waveforms.

Only the first subset of the useful data is taken from the start of the rising edge until the drop of it:

```
% This variable counts the number of the first useful %
% rising edge sample %
help_1=help_array_tiempo(1)-1;

% 300 samples are the biggest number that is possible %
% take from the signals with the highest Arduino %
% bitrate %
for i=1:size(help_array_tiempo')

    if(count_1<301)
        count_1=count_1+1;

        % Data are saved in the arrays that will be used %
        % for the identification %
```

```
        tiempo_1(i)=help_1;
        help_1=help_1+1;
        entrada_1(i)=help_array_refer(i);
        salida_1(i)=help_array_sensor(i);
    end
end
```

The variable "*help1*" is used to save the number of the samples of the first complete rising edge, so the sample number is saved in the "*tiempo 1*" array and then incremented by 1 at every loop iteration [1].
The input and output samples are saved respectively in the "*entrada 1*" array and in the "*salida 1*" array to have a synchronized acquisition of the data.
To easly manage the arrays it is better to transposed them, then all the acquired samples are converted to the right scale because Arduino maps the data by using 10 bits that means $2^8 = 1024$ values in the range $[0, 1023]$; as it is shown below, the "*tiempo 1*" array is multiplied by T_{sample} to have the true acquisition sample time; then the input and output samples are multiplied by 10 because the samples conversion does not map them from 0 to 10 but from 0 to 1:

```
% The arrays are transposed to manage them easly %
tiempo_1=tiempo_1';
tiempo_1=tiempo_1*T_sample;
entrada_1=entrada_1';
entrada_1 = ((entrada_1)/1023)*10;
salida_1=salida_1';
salida_1 = ((salida_1)/1023)*10;

% Saving data %
save prueba_1.mat tiempo_1 entrada_1 salida_1
```

The three arrays are saved into a file called "*prueba 1*" to be used later for the identification.

[1]It is necessary specify that the "*tiempo 1*" array has integer entries.

After having done that, the identification process can start.
The "*fminserach*" function realizes a multidimensional unconstrained nonlinear minimization Known as Nelder-Mead; fminsearch starts from the values specified in between the brackets and it tries to find out a local minimizer for the function J_1:

```
\textit{parametros_ident_prueba_1 = fminsearch('J_1',[1 1]);}
```

The array called "*parametros ident prueba 1*" is created and it contains the optimum parameter values that are obtained using the minimum J_1 value. The optimum J_1 value is saved in the "*J 1 delay vect*" array:

```
% Print stored data and identification  model output %
figure
parametros_ident_prueba_1 = fminsearch('J_1',[1 1]);
J_1_delay_vect(1) = J_1(parametros_ident_prueba_1);
hold on
plot(tiempo_1,entrada_1,'*r')
title('Grafico␣comparacion␣modelo␣y␣datos␣prueba␣1')
xlabel('Muestros')
ylabel('Valores')
axis([1.5 2.55 0 6])
grid on
```

The input reference and the system and model outputs are plotted in the same graphic to prove the quality of the identified model of the system. Then, the parameter values and the optimum J_1 value are displayed in the Command Window adding the code below:

```
% Print of the identified parameters %
fprintf('Valores:\n␣K_1␣=␣%04.3f␣\n␣tau_1␣=␣%04.3f\n', ...
parametros_ident_prueba_1(1), parametros_ident_prueba_1(2));
```

The identified light sensor transfer function has "*Km*" value in the numerator and the "*Tm*" value in the denominator and in this case represents the only pole :

```
% First position (1) for the constant Km, second position %
% (2) for time constant Tm in the denominator of the %
% transfer function %

num_1=parametros_ident_prueba_1(1);
den_1=[parametros_ident_prueba_1(2) 1];
fprintf('\nFuncion de trasferencia G_1 identificada:\n');
G_1=tf(num_1,den_1)
```

The cost function J_1 that calculates the parameters starts by constructing the light sensor model by using the initial parameters values written through the function called [2]:

```
% Function J_1 to calculate the transfer function parameters %
function J_1=prueba(x)

    % Load data saved into prueba_1 %
    load prueba_1;

    % Transfer function definition %
    num=x(1);
    den=[x(2) 1];
    g1=tf(num,den);
```

The saved data are loaded from the file "*prueba 1*" and the transfer function "*g1*" is created by using the input values taken from the x array; the first element of the array is used as function numerator and the second

[2]The parameters are provided to the J_1 function in the main program

element of the array is used as the time constant in the function denominator. The model is built by using the Matlab function "*lsim*":

```
% Construction of the model %
ymodelo_1=lsim(g1,entrada_1,tiempo_1);
```

The comand "*lsim*" simulates the time response of a dynamic system, the one written as the first element in the brackets, to an arbitrary input. By writing the expression above, the time response of the dynamic system g_1 to the input signal saved into "*entrada 1*" is printed and by using "*tiempo 1* to be synchronized with the output samples stored in the "*salida 1*" array.
To calculate the Standard Deviation that exists between the system and the model output it is necessary to calculate the error between them:

```
% Error calculation between system output and model output %
err=(salida_1-ymodelo_1);
```

After having calculated the error, the Standard Deviation can be obtained:

```
% J_1 is equal to the standard deviation value between the %
% system output and the model output to the same input %
serr=sum(err.*err);
disp('Valor␣funcion␣costo:')
J_1=sqrt(serr)/max(size(tiempo_1)-1)
```

Only one graphic is realized to compare the input and output signals of the system and the identified model output:

```
% All the signals are compared in the same graphic %
plot(tiempo_1,salida_1,'g',tiempo_1,ymodelo_1,'b')
```

The identification results can be obtained by running "*Identification by using a first-order model*"; the identified parameters and the system transfer function are printed in the Command Window, so the results are:

```
Valor funcion costo:

J_1 =

    0.0314

Valores:
 K_1 = 0.903
 tau_1 = 0.562

Funcion de trasferencia G_1 identificada:

G_1 =

     0.9026
  ------------
  0.5624 s + 1
```

As it is possible see in the graphic below, the system identification is not sufficiently good because the model output follows the behavior of the system output but it is not sufficiently close to it; if a delay is added the identification does not provide better results.
To improve the light sensor identification, one pole more will be added to the transfer function of the model.

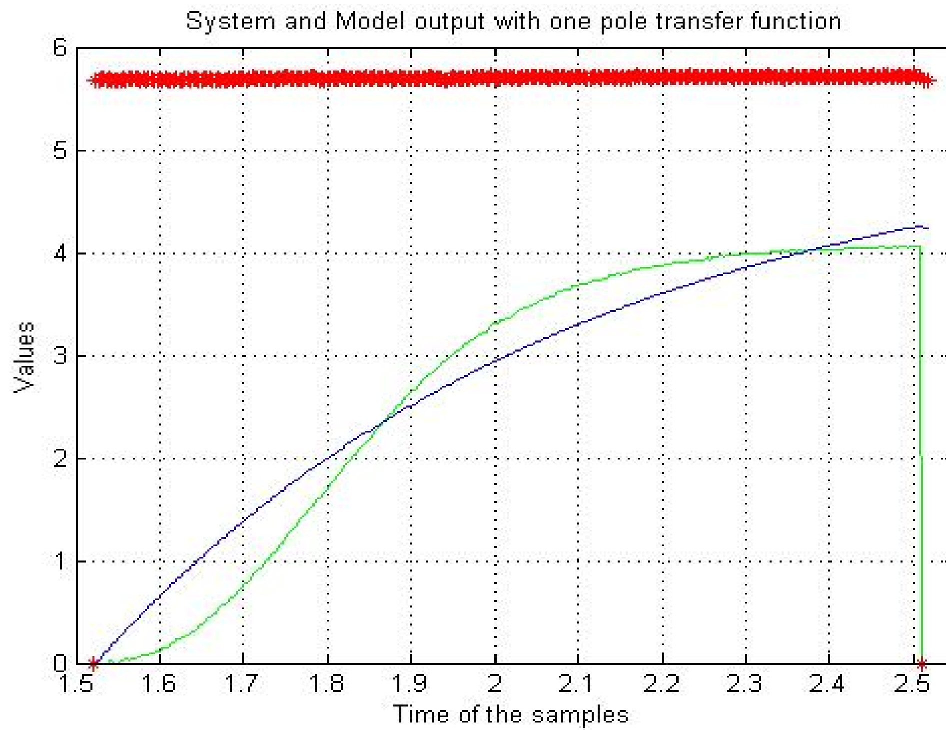

Figure 8: Identification by using a first-order model.

4.2 Identification by using a second-order models

A better identification can be realized by using a rational transfer function with two poles, which means to have two zeros in the denominator of the function. Starting from the previous identification, it is necessary change the J_1 function by adding one pole in the identification model; a new file called "*Identification by using a second-order models*" is created that is similar to the previous one:

```
% Print stored data and identification model answer %
% The first element inside fminsearch is the constant K %
% value of the system transfer function %

figure
parametros_ident_prueba_1 = fminsearch('J_1_dos',[1 1 1]);
J_1_delay_vect_dos(1) = J_1_dos(parametros_ident_prueba_1);
```

The difference between the two solutions is that by adding one pole we can ensure a higher precision in the identification process, and hence a lower error value and J_1 value are obtained; one parameter more is added in the "*fminsearch*" function to initialize the new identification process.

In this second process the new J_1 function presents only few differences; the "g1" transfer function has a different denominator that derives from the addition of the pole:

$$G(s) = \frac{K}{(\tau_1 s + 1)(\tau_2 s + 1)} \exp^{-\Delta s} = \frac{K}{(\tau_1 \tau_2 s^2 + (\tau_1 + \tau_2)s + 1)} \exp^{-\Delta s}$$

and it can be achieved by writing:

```
% Transfer function definition %
num=x(1);
den=[x(2)*x(3) (x(2)+x(3)) 1];
g1=tf(num,den);
```

Running the "*Identification by using a second-order models*" file, the identified parameters and system transfer function can be read from the Command Window:

```
Valores:
 J_1_dos = 0.02555

Valores:
 K_1 = 0.7350667811
 tau_1_1 = 0.1799512302
 tau_1_2 = 0.1800225230

Funcion de trasferencia F_1 identificada:

G_1 =

          0.7351
  ----------------------
  0.0324 s^2 + 0.36 s + 1
```

In this case the two time constants take the same values; the comparison between the system and model output is shown in Figure 9.

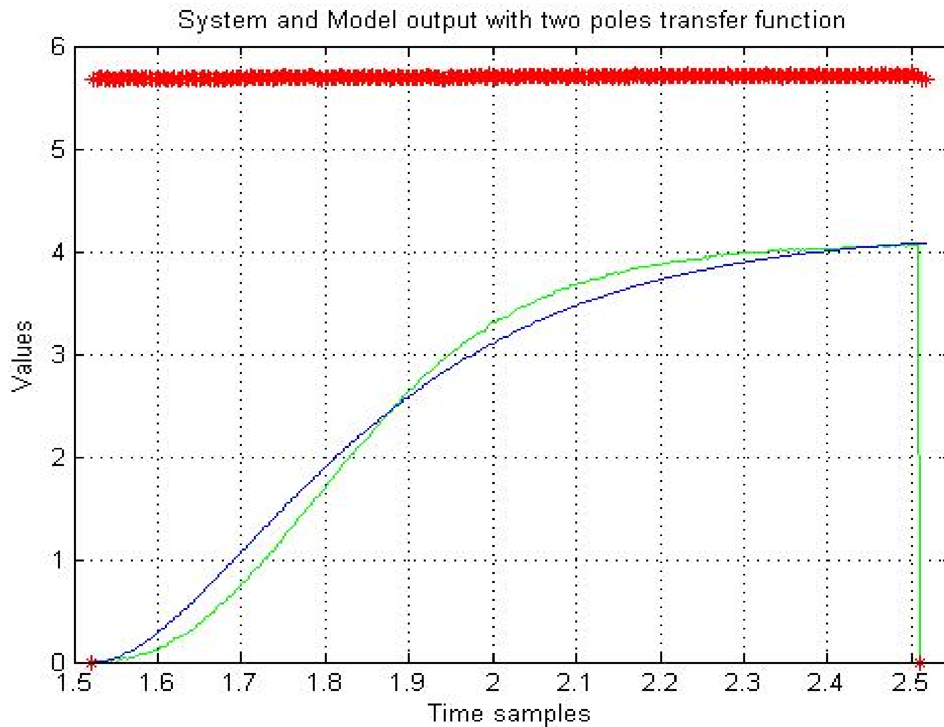

Figure 9: Identification by using a second-order models.

The graphic shows that the system Identification by using a second-order models provides considerable improvements, so it is better than the previous one; looking at the graphic it is clear that the model output has a delay compared to the system output and this gives a possible solution to improve the identification by adding an exponential term.
This exponential term represents the model delay; the delay is progressively increased by adding the sample time at every loop iteration until we find out the minimum and optimum J_1 value; the lowest J_1 value obtained provides the parameter value for the best identified model of the system and the optimum delay as well.

4.3 Identification by using a second-order models and delay

All the J_1 optimal values are saved in the array called "*J 1 delay vect*" and they are plotted in the same graphic; looking at this graphic it is possible to choose the best delay value that gives the J_1 lowest value; this is done by using the "*Identification by using a second-order models delta*" file:

```
t=0;

for delta=1:30
     t=t+1;
     delta=delta*T_sample;
     save delta

     % The first element inside fminsearch is the constant K %
     % value of the system transfer function %
     figure
     parametros_ident_prueba_1 = fminsearch('J_1_delta_1_dos'...
     ...,[1 1 1]);
     J_1_delay_vect(t+1) = J_1_delta_1_dos...
     ...(parametros_ident_prueba_1);
```

In the code above there is a main loop created to calculate and save different J_1 values by introducing different Δ values that represent the different proposed delay values to compare them later; the identification process is the same as in the "$\Delta = 0$" case. The delay must be added to the transfer function:

```
% Delay %
s=tf('s');
delay_1=tf(exp(-s*delta));
G_1=tf(num_1,den_1);
F_1=G_1*delay_1
```

First the "*delay 1*" transfer function is created by using the delay value starting from $delay = T_{sample} = 1/300$ to arrive in the last lap to the value $delay = T_{sample} * 30 = 0.1[s]$; the exponential delay term is then multiplied by the "*G 1*" transfer function to have the requested one.
Introducing a new loop, the J_1 optimum values with different delays can be compared to find out the optimum delay:

```
D = 0:1:30;
D = D*T_sample;
% J_1 comparison with different delay %
figure
plot(D,J_1_delay_vect,'*r')
title('Grafico␣comparacion␣J_1_{dos}␣con␣diferentes␣retardos')
xlabel('Delta')
ylabel('J_1␣delta_1␣dos')
axis([0 30*T_sample 0.023 0.042])
grid on
D = D/T_sample;
```

Looking at the resulting graphic below, it is clear that there is a minimum J_1 value:

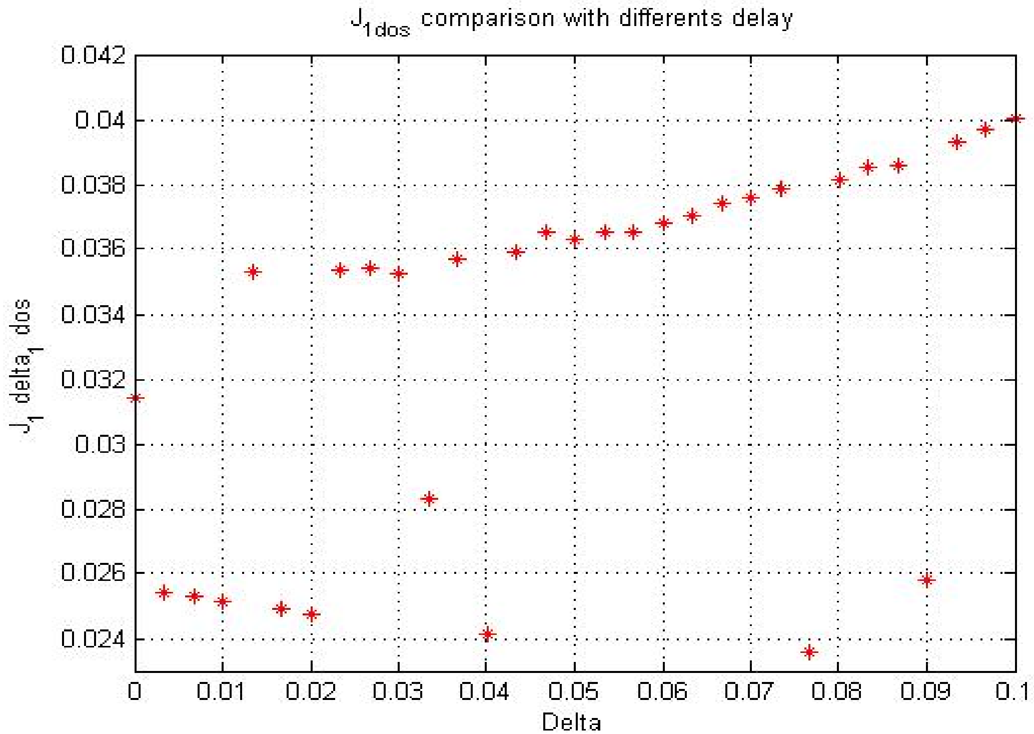

Figure 10: Identification by using a second-order models with differents Δ.

The minimum and optimum J_1 value is close to $\Delta = 0.08[ms]$; not all the J_1 values in the curve above takes value of the same order of the smallest one because the identification process stops before at a higher value, so those J_1 contain a not useful delay value.
After having analyzed the J_1 values printed in the Command Window, the optimum one is obtained for $\Delta = 0.0767[ms]$:

```
Valores:
 J_1_delta_1_dos = 0.02360
 delta = 0.0767

Valores:
 K_1 = 0.6995959914
```

```
 tau_1_1 = 0.1289396326
 tau_1_2 = 0.1288624035

Funcion de trasferencia G_1 identificada:

F_1 =

                                 0.6996
  exp(-0.0767*s) * -------------------------
                   0.01662 s^2 + 0.2578 s + 1
```

The identification process has calculated the optimum F_1 system transfer function:

$$F(s) = \frac{K}{\tau_1\tau_2 s^2 + (\tau_1 + \tau_2)s + 1} e^{-\Delta s} = \frac{0.6996}{0.01662s^2 + 0.2578s + 1} e^{-0.0767s}$$

As far as the graphic below is concerned, it is clear that the identification by using two poles and a delay gives better results than the one by using only one pole, in fact the identified model output is closer to the system output and the J_1 optimum value is lower.

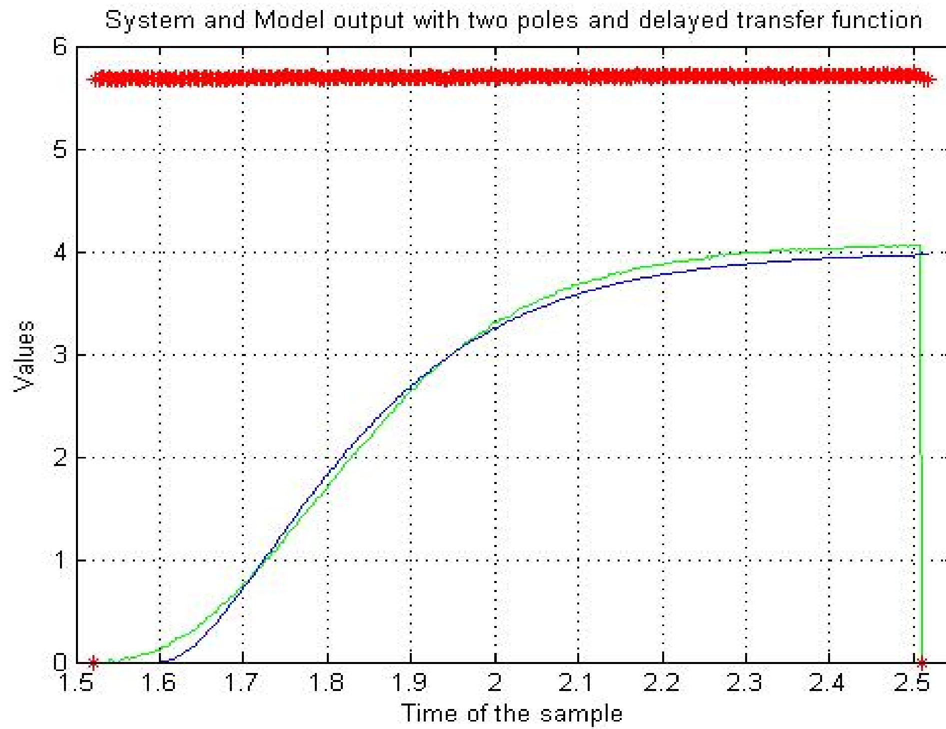

Figure 11: Identification by using a second-order models with $\Delta = 0.0767$.

The next step is to realize a PID controller for the system to compensate the error between the input reference and the system output taken from the light sensor.

5 First PID controller

At the beginning of the PID realization it is necessary to simulate the light sensor output by introducing the PID in between the voltage generator and the light sensor; if the PID costants are calculated and tested directly in the real circuit, this will be dangerous for the electronic components and the circuit could break. To determine the PID constant it is used a particular method, taken from a book called "HANDBOOK OF PI AND PID CONTROLLER TUNING RULES", written by Aidan O'Dwyer (Dublin Institute of Technology, Ireland), 3rd Edition, for a transfer function with two poles; the "*PID simulation*" file is used to find out the PID parameters.
This method is used for systems called SOSPD that means "*Second Order System Plus time Delay model*" with a transfer function of the same kind of the identified one:

$$\frac{K_m e^{-s\tau_m}}{T_{m1}^2 s^2 + 2\xi T_{m1} s + 1}$$

or

$$\frac{K_m e^{-s\tau_m}}{(1 + T_{m1}s)(1 + T_{m2}s)}$$

The first step is define the general variables and the found light sensor transfer function that will be used for the calculations:

```
% System transfer function parameters $
K_m=0.6996;
T_m1=0.1289;
T_m2=0.1289;
tau_m=23;
T_muestreo=1/300;
ter=(T_m1*T_m2);
sec=(T_m1+T_m2);
xi=sec/(2*T_m1);

% Transfer function definition $
P=tf([K_m],[ter sec  1]);
s=tf('s');
Delay=tf(exp(-s*tau_m*T_muestreo));
```

```
G=P*Delay
```

After this preparation, the data storaged must be charged to be ready to use them when requested:

```
% Load data %
load prueba_1_dos;

% Array declaration for data treatment %
datos_entrada = [];

% Saving data in the array %
for j = 0:296
    datos_entrada(j+1,1)=j;
    datos_entrada(j+1,2)=entrada_1(j+1);
end
```

As shown above the data has to be loaded by taking them from the "*Data*" file created before; an array is created and its name is "datos entradas" which is a matrix with 2 columns whose first column contains the sample number and whose second one contains the sample value.

```
% Real delay value definition %
real_delay=tau_m*T_muestreo;

% Time constants ratio %
time_rap = real_delay/T_m1;

average = mean(datos_entrada(:,2),1);
datos=[];
datos(:,1)=datos_entrada(:,1)*T_muestreo;
datos(:,2)=datos_entrada(:,2);
```

Then the delay must be defined as the time ratio between the values "*real delay*" and "*T m1*". The real acquisition sample time has been saved to emulate the real system behaviour and it is achieved by multiplying the stored sample number by the sample time.

For the PID parameters, starting values have been evaluated by making use of a procedure proposed in the book "*HANDBOOK OF PI AND PID CONTROLLER TUNING RULES*"; in this book it is possible to find out a lot of different techniques to realize a controller for many different systems as the identified one, namely:

$$G_m(s) = \frac{K_m e^{-s\tau_m}}{(T_{m1}s+1)(T_{m2}s+1)}$$

or

$$\frac{K_m e^{-s\tau_m}}{(T_{m1}^2 s^2 + 2\xi_m T_{m1}s + 1)}.$$

The PID controller transfer function can be expressed as:

$$G_P(s) = K_P(1 + \frac{1}{T_I s} + \frac{T_D}{1+sT_L}s)$$

This procedure calculates four PID constants by using the transfer function parameters. First of all it is necessary to calculate the three different values called "*x1*", "*x2*", "*x3*"; those values can be found by looking to the table proposed by the method used to this PID realization and they are related first to the ratio τ_m/T_{m1} that in this system is $real_{delay}/T_{m1} = 0.0767/0.1289 = 0.5948$, then to the coefficient ξ that appears in the transfer function denominator; in this case the two time constants in the denominator take the same values that means there are two distinct real poles, so $\xi = 1$.

Looking at the table above, it is necessary to calculate the weighted average between two values for every "x_i"; τ_m/T_{m1} value stays in between $[0.5, 1.0]$ so:

"x_1" results from the weighted average between "$x_1 = 4.0$" and "$x_1 = 1.85$"; "x_2" results from the weighted average between "$x_2 = 1.06$" and

Minimum IAE – Lopez (1968), *pp. 79–90.*	x_1/K_m			$x_2 T_{m1}$			$x_3 T_{m1}$		*Model: Method 1*
	Coefficients of K_c, T_i *and* T_d *deduced from graphs. These are representative results.*								
ξ_m	0.5			0.6			0.8		
	x_1	x_2	x_3	x_1	x_2	x_3	x_1	x_2	x_3
$\tau_m/T_{m1} = 0.1$	27	-	0.28	28	-	0.26	30	-	0.26
$\tau_m/T_{m1} = 0.2$	9.5	0.61	0.48	10.5	0.59	0.45	11.5	0.57	0.41
$\tau_m/T_{m1} = 0.5$	2.25	0.97	0.92	2.6	1.00	0.82	3.2	1.05	0.69
$\tau_m/T_{m1} = 1.0$	0.78	1.11	1.40	1.0	1.25	1.20	1.4	1.47	0.94
$\tau_m/T_{m1} = 2.0$	0.39	1.27	1.40	0.52	1.56	1.30	0.77	1.92	1.15
$\tau_m/T_{m1} = 5.0$	0.42	2.7	1.40	0.45	2.9	1.50	0.53	3.0	1.70
$\tau_m/T_{m1} = 10.0$	0.38	4.9	1.15	0.41	5.3	1.40	0.47	5.7	1.90

ξ_m	1.0			1.5			2.0			4.0		
	x_1	x_2	x_3	x_1	x_2	x_3	x_1	x_2	x_3	x_1	x_2	x_3
$\tau_m/T_{m1} = 0.1$	31	-	0.24	35	-	0.22	40	-	-	68	-	-
$\tau_m/T_{m1} = 0.2$	13.0	0.59	0.43	16.0	0.56	0.30	19.5	0.56	0.25	47	0.53	-
$\tau_m/T_{m1} = 0.5$	4.0	1.06	0.59	6.0	1.11	0.46	7.8	1.14	0.39	16.5	1.14	0.29
$\tau_m/T_{m1} = 1.0$	1.85	1.56	0.82	3.0	1.75	0.68	4.3	1.85	0.60	9.0	1.92	0.52
$\tau_m/T_{m1} = 2.0$	1.05	2.3	1.10	1.65	2.6	1.05	2.3	2.9	1.00	4.8	3.3	0.98
$\tau_m/T_{m1} = 5.0$	0.62	3.6	1.90	0.85	4.3	2.10	1.1	4.8	2.15	2.15	6.5	2.3
$\tau_m/T_{m1} = 10.0$	0.52	6.1	2.35	0.60	6.9	2.9	0.70	7.4	3.45	1.2	10.0	4.0

Figure 12: SOSPD tuning method.

"$x_2 = 1.56$"; "x_3" results from the weighted average between "$x_3 = 0.59$" and "$x_3 = 0.82$";

Thus the resulting values are:

$$x_1 = 3.57 \qquad x_2 = 1.16 \qquad x_3 = 0.636$$

Then the three PID constants are calculated by using the equations proposed: the first constant to be calculated has to be K_P and the other two depends on its value through the time constants T_I and T_D:

$$K_P = \frac{x_1}{K_m} = \frac{3.57}{0.6996} = 5.1029$$
$$T_I = x_2 * T_{m1} = 1.16 * 0.1289 = 0.1495$$
$$T_D = x_3 * T_{m1} = 0.636 * 0.1289 = 0.082$$

In the derivative term a pole at high frequency has been added by introducting T_L on the denominator of the derivative component and this solution does not amplify the error at the high frequencies; this pole results from the equation:

$$T_L = \frac{T_D}{\beta} = \frac{0.082}{15} = 0.055$$

The β constant is an integer chosen in the range $[3, 15]$; the β value is chosen to ensure that the pole $1/T_L$ is outside of the band of control; this time constant introduces a limit at the high frequencies. If T_L is such that $1/T_L$ is bigger enough than the crossing frequency ωc, its negative contribution to the phase margin can be neglected; therefore $1/T_L$ does not decrease the phase margin. On the other hand, a too small T_L amplifies the error in the high frequencies. β is chosen close to 10 or bigger and it can be set till 15 without any significant change, but not bigger than that value because a higher β brings a worse phase margin Φ_{PM}.
These constants have been obtained by writing the code below:

```
% Proportional PID constant%
K_p = x_1/K_m;

T_i = x_2*T_m1;
T_d = x_3*T_m1;
T_l = T_d/beta;
```

Finally, the other two PID constants can be calculated from the K_P value as follows:

$$K_I = \frac{K_P}{T_I} = \frac{5.1029}{0.1495} = 16.7392$$
$$K_D = K_P * T_D = 5.1029 * 0.082 = 0.4183$$

5.1 Simulink model

The next step is to build the system model, by adding the PID controller corresponding to the previously derived constants parameters:

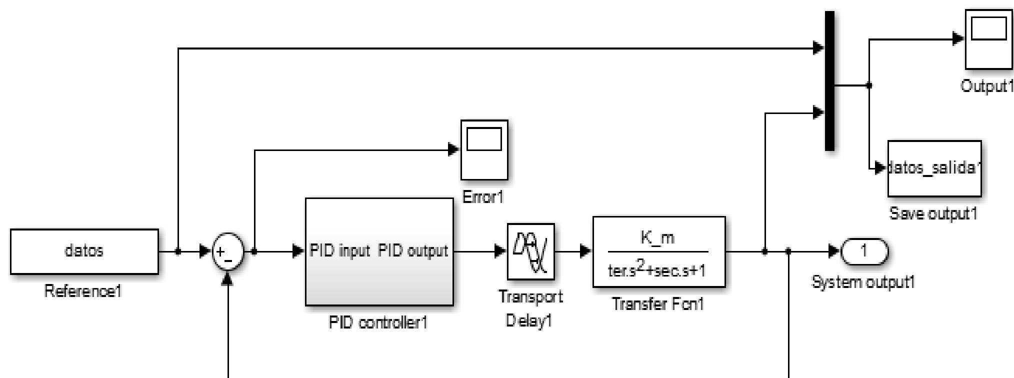

Figure 13: Simulink model.

The system model is created by using the following blocks:

1. Two "*Data*" blocks, one to provide the input reference and one for saving the output data;
2. One "*Subsystem*" block that contains the PID controller;
3. One "*Transport Delay*" block to introduce the delay;
4. One "*Transfer function*" block to introduce the light sensor transfer function;
5. Two "*Scope*" blocks, the first one to display on the screen the error between the system input and output, the second one to display on the screen the system output.

The "*Data*" block called "*Reference*" provides the data that are taken from the "*datos*" array to give to the simulated system the reference samples taken through the Arduino board; the other "*Data*" block called "*Save output*" is used to save the data in the "*datos salida*" array.

The "*Subsystem*" block contains a parallel PID realization and it specifically contains:

1. three "*Gain*" blocks to realize the three PID constants;
2. two "*Transfer function*" blocks, the first one to introduce the null pole for the integral action, the second one to introduce the null zero and the last one for the T_L pole introduced by the derivative term;
3. three "*Scope*" blocks to display on the screen the effects of the three PID actions on the input signals;
4. one "*Sum*" block to add the three PID constants effects.

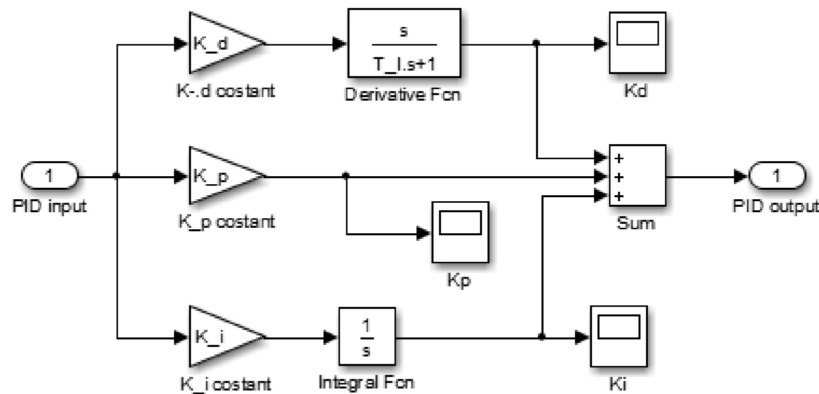

Figure 14: Simulated PID.

The first test has been run by using a classical parallel PID realization and by setting as controller parameters those obtained with the SOSPD method. The three obtained constants have been inserted respectively in the "K_d",

"K_p" and "K_i" "*Gain*" blocks and the T_L time constant has been inserted in the "*Derivative Fcn*" "*Transfer function*" block to realize the derivative PID term.

The "*Transport Delay*" block has been used to introduce the delay:

$\Delta = T_{sample} * \tau_m = 0.0033[s] * 23 = 0.0767[s]$.

In the "*Transfer function*" block, the identified transfer function must be defined; the transfer function numerator contains the K_m constant, the transfer function denominator contains the poles that have been found previously and they are expressed by using the equation $(ter)s^2 + (sec)s + 1$ that represents the product between $(T_{m1}s + 1)$ and $(T_{m2}s + 1)$, with $ter = (T_{m1} * T_{m2})$ and $sec = (T_{m1} + T_{m2})$.

Using the two "*Scope*" blocks, the error and the system output can be visualized.

Running the "*PID simulation*" file, the parameters have been saved in the Workspace to be ready for the simulation; the "*Output*" "*Scope*" block shows the comparison between the model output and the input reference:

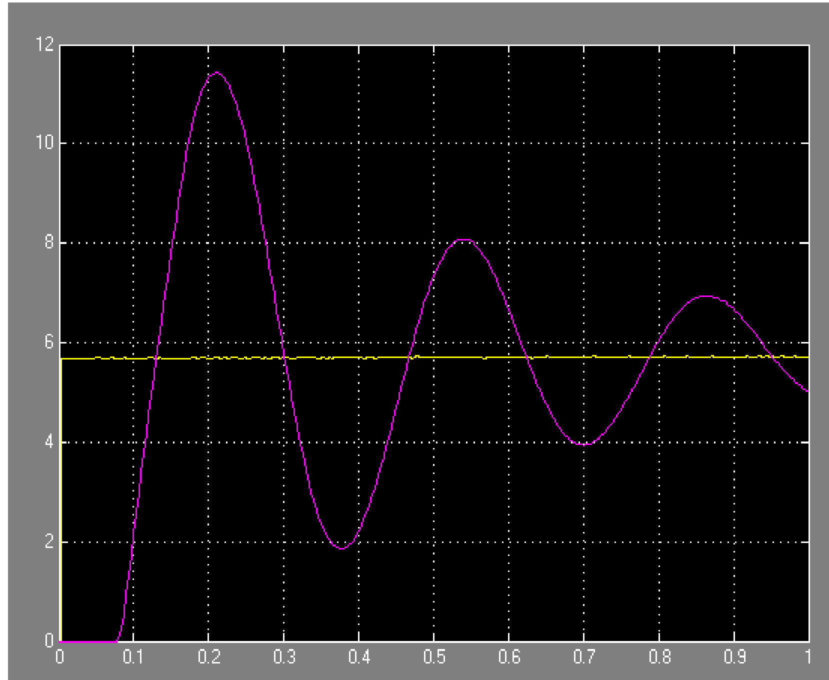

Figure 15: System output.

From the graphic above, it is possible to observe the first results: the two signals intersect many times, the first one close to $0.13[s]$; the simulated system output has many overshoots and the highest one is $11.42[Volt]$, so the maximum overshoot value in the step response is the double of the reference signal, and the system output does not come close to the reference before the end of the half period.
There are no limits imposed on the system response but to realize a good PID controller that ensures good performance it is better to introduce two specific: to the settling time of the system response on the input reference and to the overshoot maximum value. These specific allow to realize a faster controller to have a more precise light sensor response to the input reference on it.

6 PID controller improvement

The timing constraints chosen to impose on the light sensor output are:

1. Settling time T_{set} at the 5%;
2. Overshoot $S \leq 10\%$.

The settling time of an output device is the time elapsed from the application of an ideal instantaneous step input, in this case is the average of the input reference samples to the light sensor because voltage that is provides by the generator is subjected to electronic noises, to the time at which the light sensor output has entered and remained within a specified error band, in this case at 5%, usually symmetrical about the final value.
In signal processing or control theory, overshoot is when a signal or function exceeds its target; in this case the request is $S \leq 10\%$ that means that the light sensor output must be lower than the sum of input reference, in this case the average of the input samples value has been calculated for the same reasons mentioned above, plus the 10% of that value for all the time.
By introducing three "*Step*" blocks that represent the two specific mentioned above and that are linked to the same "*Multiplier*" component, the new Simulink model results:

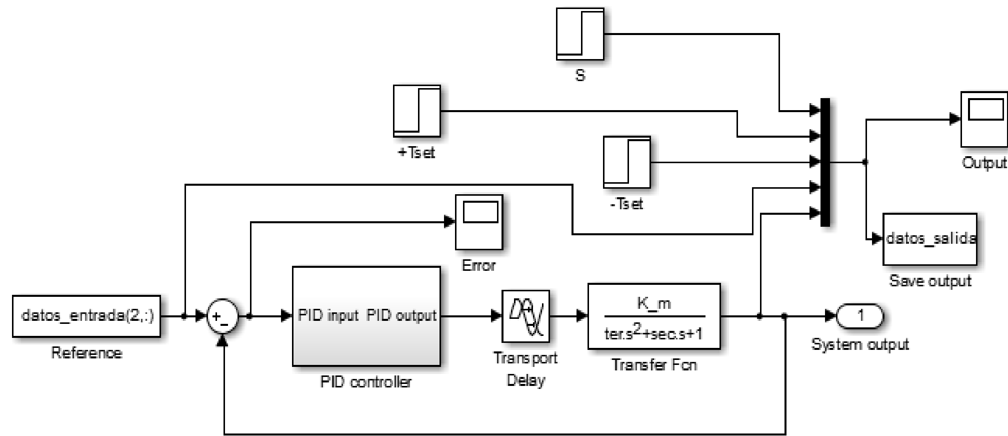

Figure 16: Simulated system with time limits.

So it is possible to display in the "*Output*" block the light sensor response to the input reference together with the specific; in fact, from the graphic below it is clear that the system output with this PID realization presents some overshoots S, the maximum one exceeds the specific given; furthermore the settling time T_{set} is not respected.

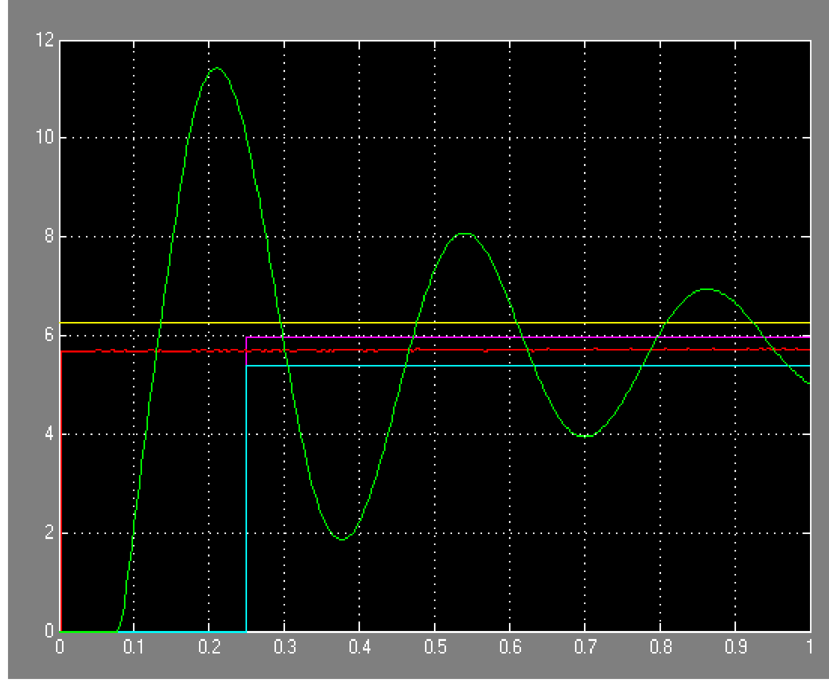

Figure 17: Simulated system output with timing constraints.

To improve the system performances, the three PID constants can be manually tuned till we find out the best PID constants value that ensure the system output to respect the constraints.
After having tried some attempts, a good choice of the PID parameters is:

$$K_P = \frac{x_1}{K_m} - 2.8 = \frac{3.57}{0.6996} - 2.8 = 2.3029$$

$$T_I = x_2 * T_{m1} = 1.16 * 0.1289 = 0.1495$$

$$T_D = x_3 * T_{m1} = 0.636 * 0.1289 = 0.082$$

$$T_L = \frac{T_D}{\beta} = \frac{0.082}{15} = 0.055$$

$$K_I = \frac{K_P}{T_I} - 5 = \frac{2.3029}{0.1495} - 5 = 10.4016$$

$$K_D = K_P * T_D = 2.3029 * 0.082 = 0.1888$$

Running the Simulink model it is possible to see in the "*Output1*" "*Scope*" block the PID improvement.

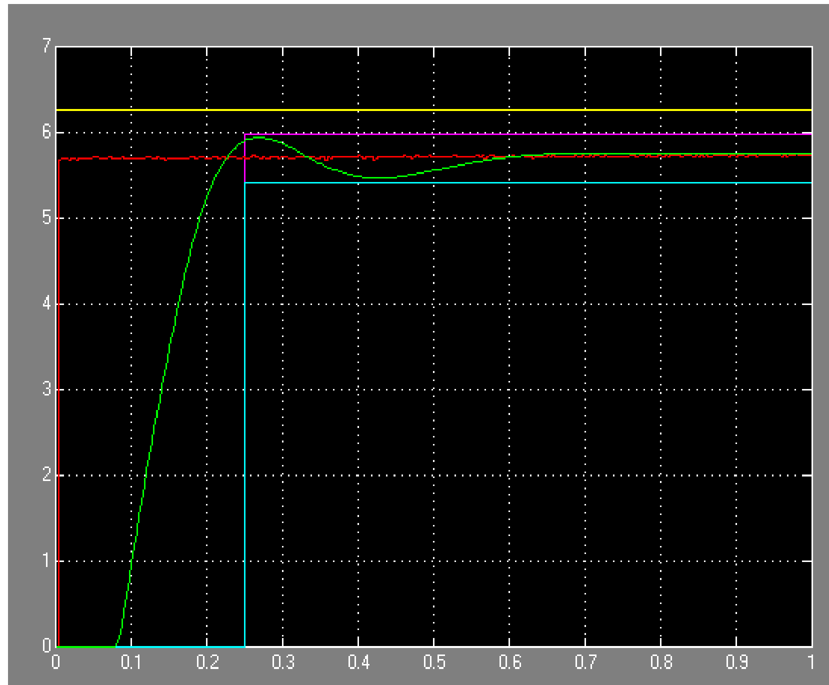

Figure 18: Simulated system output improvement.

Looking at the graphic that shown the light sensor output by using the new PID realization, we see that the constraints on the step response have been respected, and the resulting PID transfer function:

$$PID(s) = K_P * (1 + \frac{1}{T_I s} + \frac{T_D s}{(1 + T_L s)}) = K_P + \frac{K_I}{s} + K_d \frac{s}{1 + T_L s}$$

corresponding to the above parameter values becomes:

$$PID(s) = 2.3029 + \frac{10.4016}{s} + 0.1888 \frac{s}{1 + 0.055s}$$

Another method that can be implemented to find out the PID controller parameters is to design the controller in the frequency domain but it will not explained in this thesis because the resulting parameters do not provide better result; the obtained parameters are:

$$K_p = 2.8654 \qquad K_d = 0.3359 \qquad K_i = 2.4445$$

and by running the same Simulink model previously used to the first simulation, there are not better result as it is possible see by looking to the "*Output*" "*Scope*" block:

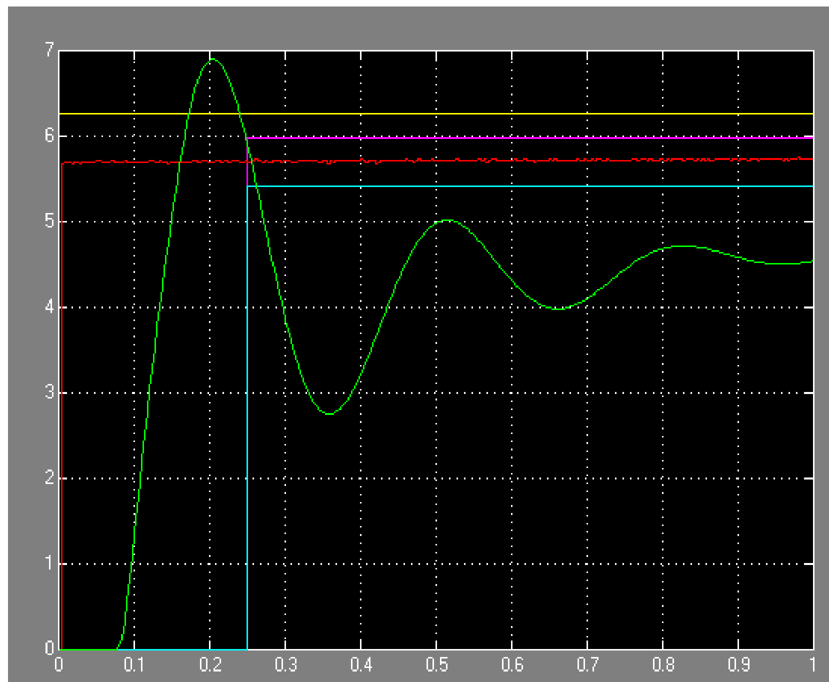

Figure 19: Simulated PID in frequency.

A manually tuning should be done but it is not realized because it would provide the same PID constants obtained in the first PID realization, so the Matlab code to realize this PID controller has been inserted at the end of the thesis only to consultation.

7 Conclusions

At the end of the identification process a light sensor transfer function of the second order was obtained with an exponential delay term; a transfer function with one pole would not has been enough to a correctly identification.

The best PID controller realization results by starting from a method developed for SOSPD system as the transfer function that it has been identified and then manually changing the PID constants values till find out the best PID constant while the frequency study does not provide better result.

A Appendix

A.1 Arduino code

```
/* This program saves analog signals with Arduino UNO */

  /* Global variables definition */
  String tiem;
  String entr;
  String sal;
  String video;

  /* Start of the main loop */
  void setup()
  {
    /* Rate definition at maximum value */
    Serial.begin(115200);

    for(int i=0; i < 1500; i++)
    {
        /* Variables initialization to print on the screen */
        tiem=String("\t");
        entr=String("\t");
        sal=String("\r\n");

        /* Sample number */
        tiem=i+tiem;

        /* Input sample */
        entr=analogRead(A0)+entr;

        /* Output sample */
        sal=analogRead(A1)*2+sal;

        /* Data concatenation */
        video=String(tiem+entr+sal);

        /* Data are printed on the screen */
        Serial.print(video);
    }
  }/* Loop end */

  /* This method must be written although it isn't used */
```

```
void loop()
{
}
```

A.2 Processing code

```
/* It is necessary import the processing.serial library */
import processing.serial.*;

/* Variables's initialization */
Serial mySerial;

 /* This command serves to print on the file acquired data */
PrintWriter output;

/* Setup method necessary to run the program */
void setup()
{
  /* Selection of serial port and the acquisition speed  */
  mySerial = new Serial(this, Serial.list()[1], 115200);

  /* Creation of the file on the same folder of the
  Processing program */
  output = createWriter("Data.m");
}

/* This method is done to storage all the rows printed on the
   serial port */
void draw()
{
  /* If the serial port is ready takes the data */
  if (mySerial.available() > 0)
  {
    /* Reads data to the serial port until new line */
    String value = mySerial.readStringUntil('\n');

    /* If the read value is not null then print it on the Data
       file */
    if( value!= null)
    {
      output.print( value );
    }
  }
}

/* Exit to the program when any button is pressed */
void keyPressed()
```

```
{
  /* Writes the remaining data on the file */
  output.flush();

  /* Close the file */
  output.close();

  /* Exit to the program */
  exit();
}
```

A.3 Matlab code

A.3.1 Identification by using a first-order model

```
%% System identification using Transfer function with one %%
%% pole %%
% Program initialization %
clc
close all

% Message errors disabled %
[msg, id] = lastwarn;
warning('off', id)

% Data reading from "Data.m" file %
[samples referencias sensorValues]=textread('Data.m','%d%d%d');

% Array declaration for data treatment %

help_array_tiempo=[];
help_array_refer=[];
help_array_sensor=[];

% Sample time %
T_sample = 1/300;

% Time array %
tiempo_1=[];

% Input array %
entrada_1=[];

% Output array %
salida_1=[];

 % Array that saves the minimum function %
 % cost with differents delta values %
J_1_delay_vect=[];

% Counter used as controller of the data %
count_1=0;

j=1;
```

```
% Loop used to treat data stored %
for i=1:1500

    % The firsts data aren't used because the reference %
    % value is zero %
    if((referencias(i)==0)&&(count_1==0))

        count_1=count_1+1;

    end

    % When the reference isn't equal zero data are taken %
    % then is taken one previous sample to have a better %
    % visualization on the screen %
    if((referencias(i)~=0)&&(count_1==1))
        help_array_tiempo(j)=samples(i-1);
        help_array_refer(j)=referencias(i-1);
        help_array_sensor(j)=sensorValues(i-1);
        j=j+1;
    end
end

% This variable counts the number of the first useful %
% rising edge sample %
help_1=help_array_tiempo(1)-1;

% 300 samples are the biggest number that is possible %
% take from the signals with the highest Arduino %
% bitrate %
for i=1:size(help_array_tiempo')

    if(count_1<301)
        count_1=count_1+1;

        % Data are saved in the arrays that will be used %
        % for the identification %
        tiempo_1(i)=help_1;
        help_1=help_1+1;
        entrada_1(i)=help_array_refer(i);
        salida_1(i)=help_array_sensor(i);
    end
end

% The arrays are transposed to manage them easily %
tiempo_1=tiempo_1';
```

```
tiempo_1=tiempo_1*T_sample;
entrada_1=entrada_1';
entrada_1 = ((entrada_1)/1023)*10;
salida_1=salida_1';
salida_1 = ((salida_1)/1023)*10;

% Saving data %
save prueba_1.mat tiempo_1 entrada_1 salida_1

% Print stored data and identification  model output %
figure
parametros_ident_prueba_1 = fminsearch('J_1',[1 1]);
J_1_delay_vect(1) = J_1(parametros_ident_prueba_1);
hold on
plot(tiempo_1,entrada_1,'*r')
title('Grafico␣comparacion␣modelo␣y␣datos␣prueba␣1')
xlabel('Muestros')
ylabel('Valores')
axis([1.5 2.55 0 6])
grid on

% Print of the identified parameters %
fprintf('Valores:\n␣K_1␣=␣%04.3f␣\n␣tau_1␣=␣%04.3f\n', ...
parametros_ident_prueba_1(1), parametros_ident_prueba_1(2));

% First position (1) for the constant Km, second position %
% (2) for time constant Tm in the denominator of the %
% transfer function %

num_1=parametros_ident_prueba_1(1);
den_1=[parametros_ident_prueba_1(2) 1];
fprintf('\nFuncion␣de␣trasferencia␣G_1␣identificada:\n');
G_1=tf(num_1,den_1)
```

A.3.2 J 1

```
% Function J_1 to calculate the transfer function parameters %
function J_1=prueba(x)

    % Load data saved into prueba_1 %
    load prueba_1;

    % Transfer function definition %
    num=x(1);
    den=[x(2) 1];
    g1=tf(num,den);


    % Construction of the model %
    ymodelo_1=lsim(g1,entrada_1,tiempo_1);

    % Error calculation between system output and model %
    % output %
    err=(salida_1-ymodelo_1);

    % J_1 is equal to the standard deviation value between %
    % the system output and the model output to the same %
    % input %
    serr=sum(err.*err);
    disp('Valor funcion costo:')
    J_1=sqrt(serr)/max(size(tiempo_1)-1)

    % All the signals are compared in the same graphic %
    plot(tiempo_1,salida_1,'g',tiempo_1,ymodelo_1,'b')
end
```

A.3.3 Identification by using a second-order models

```
%% System identification using Transfer function with two %%
%% poles %%
% Program initialization %
clc
close all

% Message errors disabled %
[msg, id] = lastwarn;
warning('off', id)

% Data reading from "Data.m" file %
[samples referencias sensorValues]=textread('Data.m','%d%d%d');

% Variables initialization %
% Array declaration for data treatment %
help_array_tiempo=[];
help_array_refer=[];
help_array_sensor=[];
T_sample = 1/300;

tiempo_1=[];                    % Time array %

entrada_1=[];                   % Input array %

salida_1=[];                    % Output array %

% Array that saves the minimum function cost with differents %
% delta values %
J_1_delay_vect_dos=[];

% Counter used as controller of the data %
count_1=0;
j=1;

% Loop used to treat data stored %
for i=1:1500

    % The firsts data aren't used because the reference value %
    % is zero %
    if((referencias(i)==0)&&(count_1==0))
```

```
        % Here the initial reference value is zero %
        count_1=count_1+1;
    end

    % When the reference isn't equal zero are taken data, then %
    % is taken one previous sample that makes a better %
    % visualization on the screen %
    if((referencias(i)~=0)&&(count_1==1))
        help_array_tiempo(j)=samples(i-1);
        help_array_refer(j)=referencias(i-1);
        help_array_sensor(j)=sensorValues(i-1);
        j=j+1;
    end
end

% This variable counts the number of the first useful %
% rising edge sample %
help_1=help_array_tiempo(1)-1;

% 300 samples are the biggest number that is possible %
% take from the signals with the highest Arduino bitrate %
for i=1:size(help_array_tiempo')

    if(count_1<301)
        count_1=count_1+1;

        % Data are saved in the arrays that %
        % will be used for the identification %
        tiempo_1(i)=help_1;
        help_1=help_1+1;
        entrada_1(i)=help_array_refer(i);
        salida_1(i)=help_array_sensor(i);
    end
end

% The arrays are transposed to manage them easily %
tiempo_1 = tiempo_1';
tiempo_1 = tiempo_1*T_sample;
entrada_1 = entrada_1';
entrada_1 = ((entrada_1)/1023)*10;
salida_1 = salida_1';
salida_1 = ((salida_1)/1023)*10;

% Saving data %
save prueba_1_dos.mat tiempo_1 entrada_1 salida_1
```

```
% Print stored data and identification's model answer %
figure
parametros_ident_prueba_1 = fminsearch('J_1_dos',[1 1 1]);
J_1_delay_vect_dos(1) = J_1_dos(parametros_ident_prueba_1);

fprintf('Valores:\n␣J_1_dos␣=␣%04.5f␣\n\n', ...
J_1_dos(parametros_ident_prueba_1));

hold on
plot(tiempo_1,entrada_1,'*r')
title('Grafico␣comparacion␣modelo␣y␣datos␣prueba␣1')
xlabel('Muestros')
ylabel('Valores')
axis([1.5 2.55 0 6])
grid on

% Print of the identified parameters %
fprintf('Valores:\n␣K_1␣=␣%04.10f␣\n␣tau_1_1␣=␣%04.10f␣...
\n␣tau_1_2␣=␣%04.10f\n',parametros_ident_prueba_1(1),...
parametros_ident_prueba_1(2), parametros_ident_prueba_1(3));

% First position (1) for the constant Km, second and third %
% position for time constants Tm1 and Tm2 in the denominator %
% of the transfer function %
num_1=parametros_ident_prueba_1(1);

den_1=[parametros_ident_prueba_1(2)*...
parametros_ident_prueba_1(3) (parametros_ident_prueba_1(2)...
+parametros_ident_prueba_1(3)) 1];

fprintf('\nFuncion␣de␣trasferencia␣F_1␣identificada:\n');
G_1=tf(num_1,den_1)

run Identificacion_two_pole_delta_2
```

A.3.4 J 1 dos

```
% Function J_1 to calculate the transfer function parameters %
function J_1_dos=prueba(x)

    % Load data saved into prueba_1 %
    load prueba_1_dos;

    % Transfer function definition %
    num=x(1);
    den=[x(2)*x(3) (x(2)+x(3)) 1];
    g1=tf(num,den);

    % Construction of the model %
    ymodelo_1=lsim(g1,entrada_1,tiempo_1);

    % Error calculation between system output and model %
    % output %
    err=(salida_1-ymodelo_1);

    % J_1 is equal to the standard deviation value between %
    % the system output and the model output to the same %
    % input %
    serr=sum(err.*err);
    %disp('Valor funcion costo:')
    J_1_dos=sqrt(serr)/max(size(tiempo_1)-1);

    % All the signals are compared in the same graphic %
    plot(tiempo_1,salida_1,'g',tiempo_1,ymodelo_1,'b')
end
```

A.3.5 Identification by using a second-order models and delay

```
%% System identification using Transfer function with two %%
%% poles and variable delay %%
t=0;

for delta=1:30
    t=t+1;
    delta=delta*T_sample;
    save delta

    % The first element inside fminsearch is the constant K %
    % value of the system transfer function %
    figure
    parametros_ident_prueba_1 = fminsearch('J_1_delta_1_dos'...
    ,[1 1 1]);
    J_1_delay_vect(t+1) = J_1_delta_1_dos...
    (parametros_ident_prueba_1);

    % Print results %
    fprintf('Valores:\n␣J_1_delta_1_dos␣=␣%04.5f␣\n␣delta...
␣␣␣␣=␣%01.4f\n\n', J_1_delta_1_dos(parametros_ident_prueba_1)...
    , delta);
    fprintf('Valores:\n␣K_1␣=␣%04.10f␣\n␣tau_1_1␣=␣%04.10f␣\n...
␣␣␣␣tau_1_2␣=␣%04.10f\n', parametros_ident_prueba_1(1),...
    parametros_ident_prueba_1(2), parametros_ident_prueba_1(3));

    % Transfer function definition %
    num_1=parametros_ident_prueba_1(1);

    den_1=[parametros_ident_prueba_1(2)*...
    parametros_ident_prueba_1(3) (parametros_ident_prueba_1(2)...
    +parametros_ident_prueba_1(3)) 1];

    fprintf('\nFuncion␣de␣trasferencia␣G_1␣identificada:\n');

    % Delay %
    s=tf('s');
    delay_1=tf(exp(-s*delta));
    G_1=tf(num_1,den_1);
    F_1=G_1*delay_1
    hold on
```

```
    % Print stored data and identification's model answer %
    plot(tiempo_1,entrada_1,'*r')
    title('Grafico␣comparacion␣modelo␣y␣datos␣prueba␣1')
    xlabel('Muestros')
    ylabel('Valores')
    grid on
end

D = 0:1:30;
D = D*T_sample;
% J_1 comparison with different delay %
figure
plot(D,J_1_delay_vect,'*r')
title('Grafico␣comparacion␣J_1_{dos}␣con␣diferentes␣retardos')
xlabel('Delta')
ylabel('J_1␣delta_1␣dos')
axis([0 30*T_sample 0.023 0.042])
grid on
D = D/T_sample;
```

A.3.6 J 1 Delta 1 dos

```
% Function J_1 to calculate the transfer function parameters %
function J_1_delta_1_dos=prueba(x)

    % Load saved data %
    load prueba_1_dos;
    load delta;

    % Delay definition %
    s=tf('s');
    delay_1=tf(exp(-s*delta));

    % Transfer function definition %
    num=x(1);
    den=[x(2)*x(3) (x(2)+x(3)) 1];
    g1=tf(num,den);
    f1=g1*delay_1;

    % Construction of the model %
    ymodelo_1=lsim(f1,entrada_1,tiempo_1);

    % Error calculation between system output and model %
    % output %
    err=(salida_1-ymodelo_1);

    % J_1 is equal to the standard deviation value between %
    % the system output and the model output to the same %
    % input %
    serr=sum(err.*err);

    %disp('Valor funcion costo:')
    J_1_delta_1_dos=sqrt(serr)/max(size(tiempo_1)-1);

    % All the signals are compared in the same graphic %
    plot(tiempo_1,salida_1,'g',tiempo_1,ymodelo_1,'b')
end
```

A.3.7 PID Simulation

```
%% PID simulation $$
% Program initialization %
clc
clear all
close all

% System transfer function parameters $
K_m=0.6996;
T_m1=0.1289;
T_m2=0.1289;
tau_m=23;
T_muestreo=1/300;
ter=(T_m1*T_m2);
sec=(T_m1+T_m2);
xi=sec/(2*T_m1);

% Transfer function definition $
P=tf([K_m],[ter sec  1]);
s=tf('s');
Delay=tf(exp(-s*tau_m*T_muestreo));

G=P*Delay

% Load data %
load prueba_1_dos;

% Array declaration for data treatment %
datos_entrada = [];

% Saving data in the array %
for j = 0:296
    datos_entrada(j+1,1)=j;
    datos_entrada(j+1,2)=entrada_1(j+1);
end

% Real delay value definition %
real_delay=tau_m*T_muestreo;

% Time constants rapport %
time_rap = real_delay/T_m1;
```

```
% Time limits %
set_perc = 5/100;
average = mean(datos_entrada(:,2),1);
range = set_perc*average;
datos=[];
datos(:,1)=datos_entrada(:,1)*T_muestreo;
datos(:,2)=datos_entrada(:,2);

% Settling time %
T_set = 0.25;

% Overshoot limit %
s = 10;
S = s/100*average;

% Selected constant for TL definition %
beta = 15;

% PID method for SOSPD model with two poles transfer function
%
x_1 = 3.57;
x_2 = 1.16;
x_3 = 0.636;

% Proportional PID constant%
K_p = x_1/K_m-2.8;

T_i = x_2*T_m1;
T_d = x_3*T_m1;
T_l = T_d/beta;

% Integrative PID constant%
K_i = K_p/T_i-5;

% Derivative PID constant%
K_d = K_p*T_d;
```

A.3.8 PID frequency study

```
%% PID frequency study %%
% Program initialization %
clc
clear all
close all

% System transfer function parameters $
K_m=0.6996;
T_m1=0.1289;
T_m2=0.1289;
tau_m=23;
T_muestreo=1/300;
ter=(T_m1*T_m2);
sec=(T_m1+T_m2);
xi=sec/(2*T_m1);

% Transfer function definition $
P=tf([K_m],[ter sec  1]);
s=tf('s');
Delay=tf(exp(-s*tau_m*T_muestreo));

G=P*Delay;

P=G

% Load data %
load prueba_1_dos;

% Array declaration for data treatment %
datos_entrada = [];

% Saving data in the array %
for j = 0:296
    datos_entrada(j+1,1)=j;
    datos_entrada(j+1,2)=entrada_1(j+1);
end

% Real delay value definition %
real_delay=tau_m*T_muestreo;

% Time costants rapport %
```

```
time_rap = real_delay/T_m1;

% Time limits %
set_perc = 5/100;
average = mean(datos_entrada(:,2),1);
range = set_perc*average;
datos=[];
datos(:,1)=datos_entrada(:,1)*T_muestreo;
datos(:,2)=datos_entrada(:,2);

% Settling time %
T_set = 0.25;

% Overshoot limit %
s = 10;
S = s/100*average;

% Csi %
csi = cos(atan(-pi/(log(s/100))));

% Crossing pulsation calculation in the close loop %
omega_n = log(average)/(csi*T_set);

% Costant value to relation TI and TD taken from 3 to 10 %
alfa = 10;

% Selected costant for TL definition %
beta = 15;

% It is calculated the P(j*omega_n) phase margin %
M_phase=rad2deg(2*csi);

% Absolute and phase process calculation %
[ModP,faseP]= bode(P,omega_n);

% Controller phase %
theta=M_phase-180-faseP;

polinomy = [1 -tand(theta)/omega_n -1/(alfa*omega_n^2)];
radici=roots(polinomy);
TD=radici(find(radici>0));
TI=alfa*TD;
T_l=TD/beta;

% PID parameters calculation %
```

```
K_p=1/(ModP*sqrt(1+(TD*omega_n-1/(TI*omega_n))^2));
K_i=K_p/TI;
K_d=K_p*TD;
```

B Bibliography

References

[1] Aidan O'Dwyer, *HANDBOOK OF PI AND PID CONTROLLER TUNING RULES*, 3rd edition, Dublin Institute of Technology, Ireland, Imperial College Press, 2009.

[2] Friedrich Heinz Effertz, Hans-Willi Huesch *Fundamentos de los sistemas automaticos de control II, Volumen 1,* ***Introduccion experimental al control de variables fisicas***, Universidad de Colonia, Leybold Didactic GmbH, Colonia, Alemana, 1992.

[3] Friedrich Heinz Effertz, Hans-Willi Huesch *Fundamentals of Automatic Control Technology II, Volume 2,* ***Experiment-based Fundamentals of Automation Systems***, University of Cologne, Leybold Didactic GmbH, Cologne, Germany, 1992.

C Webliography

References

[1] Arduino website, *Arduino*, in: http://arduino.cc/en/guide/introduction, https://www.processing.org/reference/PrintWriterflush.html

[2] Processing website, *Processing*, in https://processing.org/