POLITECNICO DI TORINO

DIPARTIMENTO DI INGEGNERIA MECCANICA E AEROSPAZIALE

Corso di Laurea Magistrale in Ingegneria Aerospaziale

Politecnico di Torino

Development of the Prototype of the Oil Heating Control System for the Lubrification System of a Mechanical Transmission Components Test Rig.

Supervisor: Prof. Andrea MURA

Master Thesis by:
Alejandro MORENO-PÉREZ
Enrollment number: 274758

July Session 2021

Acknowledgments

I would like to thank prof. Andre Mura for the guidance, support, flexibility and understanding given to me during the realization of this thesis, specially in this pandemic year.

Quiero agradecer a mis padres la oportunidad que me han dado de venir a estudiar a Turín y el apoyo que me han dado toda la vida, han sido 7 años de duro trabajo y estudio, pero finalmente puedo decir que soy ingeniero aeronáutico.

Sommario

I sistemi di controllo sono usati tutto il tempo da tutto il mondo ovunque. Un sistema di controllo regola lo stato di una variabile di controllo di un sistema. I sistemi di controllo vanno da semplici sistemi per controllare la temperatura di una singola casa, a sistemi industriali più complessi che si usano per controllare processi o macchine.

L'obbiettivo di questo lavoro è quello di sviluppare il sistema di controllo di temperatura per l'impianto di lubrificazione di un banco prova per organi di trasmissione. Questo sistema è stato sviluppato con l'uso di Arduino, e nel suo disegno si includono la programmazione degli algoritmo di controllo in Arduino, il disegno del circuito di elettronica di potenza, e la implementazione e prova del sistema completo.

Questa tesi è divisa in tre capitoli: in primo luogo si da una breve introduzione ai sistemi di controllo e Arduino, dopo di che, si spiega la procedura usata per disegnare il prototipo, questo include l'implementazione dei diversi sottosistemi (il sensore di temperatura e l'attuatore) e il disegno dei controller. Tre strategie di controllo diverse sono state sviluppate, un ON OFF, una versione modificata dello stesso, e un controllo PID. Lo sviluppo di questo ultimo comprende il processo eseguito per il PID tuning.

Infine, i risultati delle tre configurazioni sono mostrati e comparati. Si può concludere che i tre controller sono stati sviluppati soddisfacentemente e funzionano . Inoltre, si fanno delle considerazioni per lo sviluppo del sistema reale dal prototipo.

Abstract

Control systems are used everyday by everyone, managing to run smoothly automated systems everywhere. A control system regulates the state of a controlled variable of a device or system. Control systems can range from a single home heating controller using a thermostat controlling a domestic boiler to large industrial control systems which are used for controlling processes or machines.

The aim of this thesis was to develop the prototype of the oil heating system for a new test rig for mechanical transmission components. This oil heating system was designed with the use of an Arduino board, and the design included everything from the Arduino programming, to the design of the power electronics circuit, to the implementation and testing of the whole system.

This work is divided into three chapters; firstly, a quick introduction to control systems and Arduino is given, after which, the procedure followed to design the prototype is shown; this includes the implementation of the different subsystems -the temperature sensor and the actuator-, the design of the controllers -three different control strategies were implemented to study and compare the possibilities for the real system, these are; an ON OFF controller, a modified version of the same, and a PID controller- and the testing of the different configurations, including the PID tuning.

Finally, the results of the three configurations are shown and compared, and it can be stated that the three controllers work successfully. Some considerations are also given for the transformation into a real system.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Automatic Control

One of the most frequently asked questions by newcomers to control systems is: what is a control system? To answer this question, it can be said that in the daily life, there are many "objectives" that need to be achieved. E.g., in buildings, it is necessary to regulate the temperature and humidity to be able to have a comfortable life. In transportation, it is necessary to guide cars and airplanes to go from one point to another accurately and safely. Industrial manufacturing processes contain many objectives for products to meet precision and cost-effectiveness requirements. These objectives are usually met with the use of a control system that has a control strategy included. Virtually all aspects of our daily activities are affected by some type of control system.

These objectives are not arbitrary, and as part of the control strategy it might be necessary to state a certain and predefined value of one of their variables, for which it will be necessary to have measurement systems that allow an understanding of the current state of the values of these variables. These measurements are obtained with the help of the instrumentation, which allows to know the current state of the variable. Furthermore, a basic control system consists of the following components, some of which have already been named, the objectives, these are the inputs or actuating signal, the results, which can also be called outputs or controlled variables, and the control system components, the aim of which is to control the output in a prescribed way defined by the value of the inputs. A simple scheme of this idea is shown in figure 1.1.

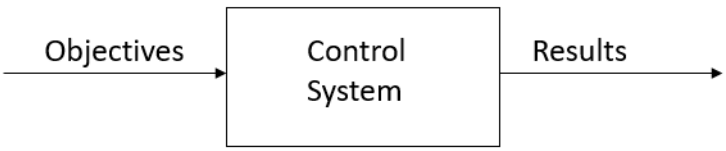


Figure 1.1: Basic components of a control system.

1.1.1 Open-Loop Control System

This type of control system is characterized by being a non-feedback system, this means that the output signal has no influence or effect on the control action of the input signal.

The elements of an open-loop control system can usually be divided into two parts; the controller and the plant. The response of the system is the combined response of these two elements. This is better exposed on figure 1.2. In this system, the real input from the user is the reference value, then the controller is in charge of turning that reference value input r into an input or actuating signal u, which is the one applied on the plant, the result of this input is the output, or controlled variable y.

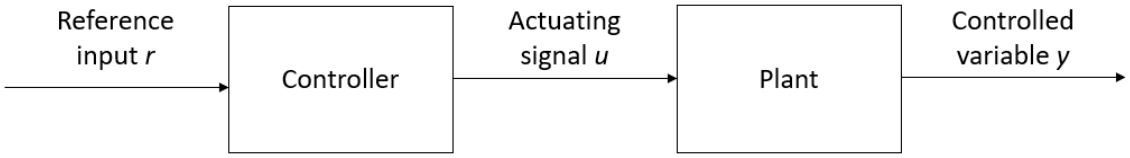


Figure 1.2: Components of an open-loop control system.

The problem behind the open-loop control system is that the input cannot be modified in order to reduce the difference between the output and the reference value. However, the simplicity and low cost of these systems makes them a cost efficient solution for many non essential applications, an example of this type of system could be a sprinkler system.

1.1.2 Closed-Loop Control System

This type of control system is characterized for being a feedback control system, this means that it has all that the open-loop control system is missing, this is; more accurate and adaptive control. This is due to the link between the output u and the reference value r.

The main characteristics of closed-loop control are; the reduction of error thanks to its ability to regulate the input of the controller, the guarantee of stability, even in unstable systems, the enhancement of the robustness against external disturbances to the process, and the ability to produce a reliable and repeatable performance [1]. Furthermore, the feedback also has effects on the following characteristics, it is reactive or error-driven, it automatically compensates for disturbances and follows change in the reference value, also, it can improve undesirable properties of the plant or system. On the other hand, these characteristics have associated a bigger complexity, this is due to the feedback path, in addition, if the controller is too sensitive to changes in its signal, this can cause the system to become unstable, as the controller tries to over correct itself [1]. The components of this type of control system are shown in figure 1.3.

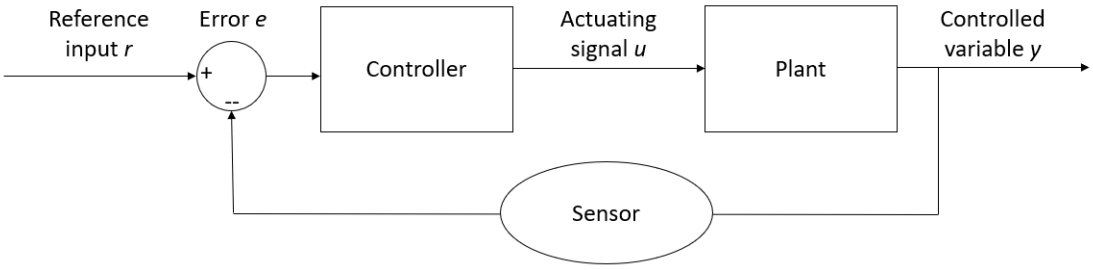


Figure 1.3: Components of a closed-loop control system.

1.1.3 Controller Configuration

In this section, the possible controller configurations are studied, some of these possible configurations are shown in figure 1.4. Since most of the control systems rely on the fixed-configuration design method, in which the designer chooses beforehand the basic configuration of the overall design. This means choosing where the controller is positioned with respect to the controlled process.

Each of the controller configurations showed in figure 1.4 represent the most commonly used configurations by designers and the difference between them is the place in which the controller is located with respect to the controlled process, as stated before. In this work, the chosen controller configuration is case (a) from figure 1.4. This configuration is called the series (cascade) compensation and it is the most commonly used system configuration. In this controller configuration, the controller is situated right before the controlled process or plant, in addition, the input of the controller is the error between the reference value of the controlled variable and the output.

1.1.4 ON OFF Controller Theory

This type of controller uses a very simple algorithm to decide whether the plant should be on or off states. This logic does not allow for a very precise control, but its advantage lies on the fact that it is very simple to design. However, it cannot be used in systems that required high accuracy. ON OFF controllers can be divided into two categories, with or without hysteresis.

1.1.4.1 ON OFF Controller without hysteresis

The control strategy is very simple, a set point is defined, for example this could be a certain temperature that wants to be reached. Figure 1.5 shows the response of a system controlled with an on off controller without hysteresis. This system has been implemented in Simulink, as observed in figure 1.6. In this Simulink model, the plant that wants to be controlled has been modeled as a first order system with a delay of 0.1 s. The ON OFF controller has been modeled as a relay in which the OFF value correspond to 0 and the ON value corresponds to 1.6. Finally, the input of the system corresponds to an unit step at time equal to zero.

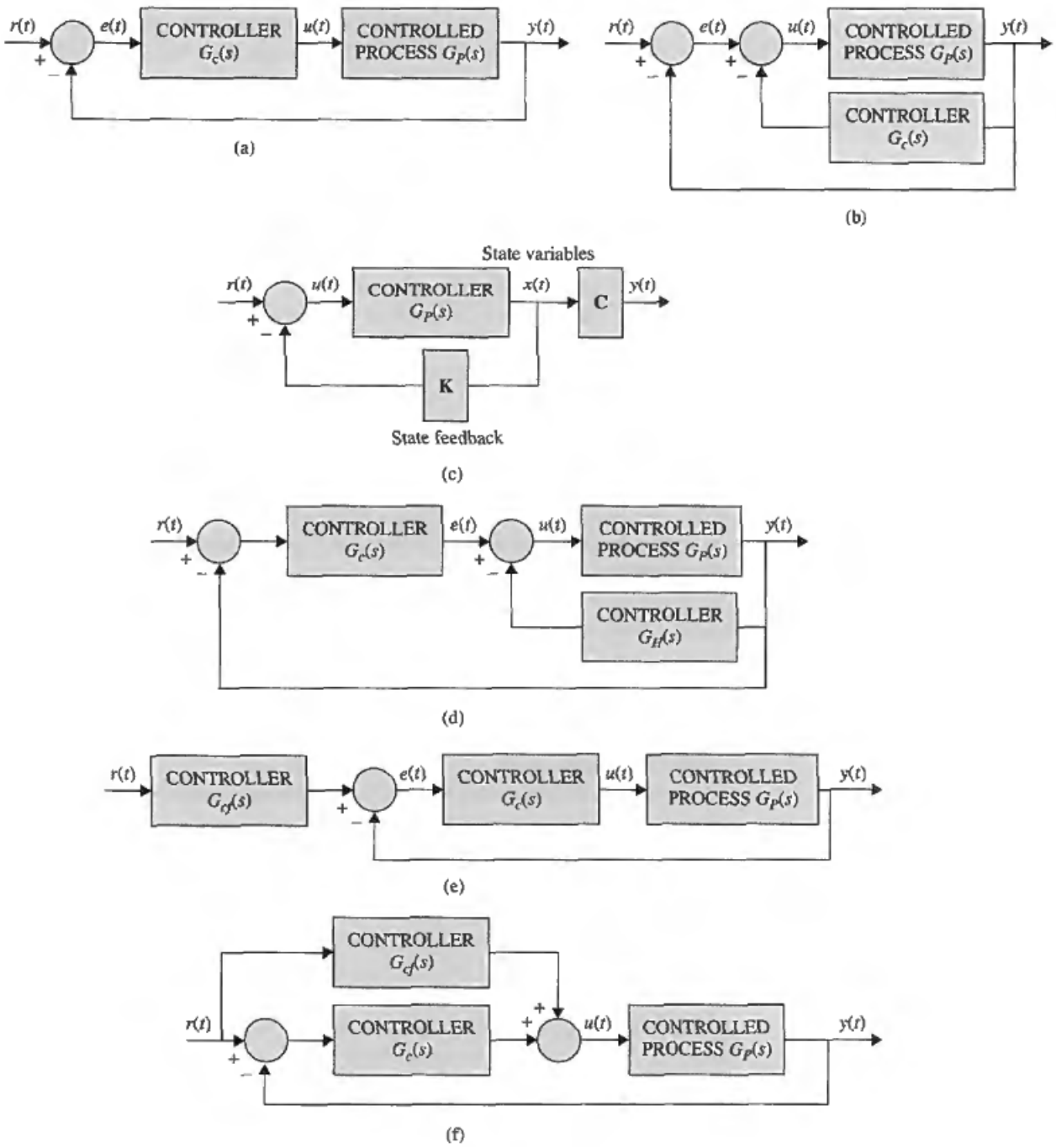


Figure 1.4: "Various controller configurations in control-system compensation. (a) Series or cascade compensation. (b) Feedback compensation. (c) State-feedback control. (d) Series-feedback compensation (two degrees of freedom). (e) Feedforward compensation with series compensation (two degrees of freedom). (f) Feedforward compensation (two degrees of freedom)." [2]

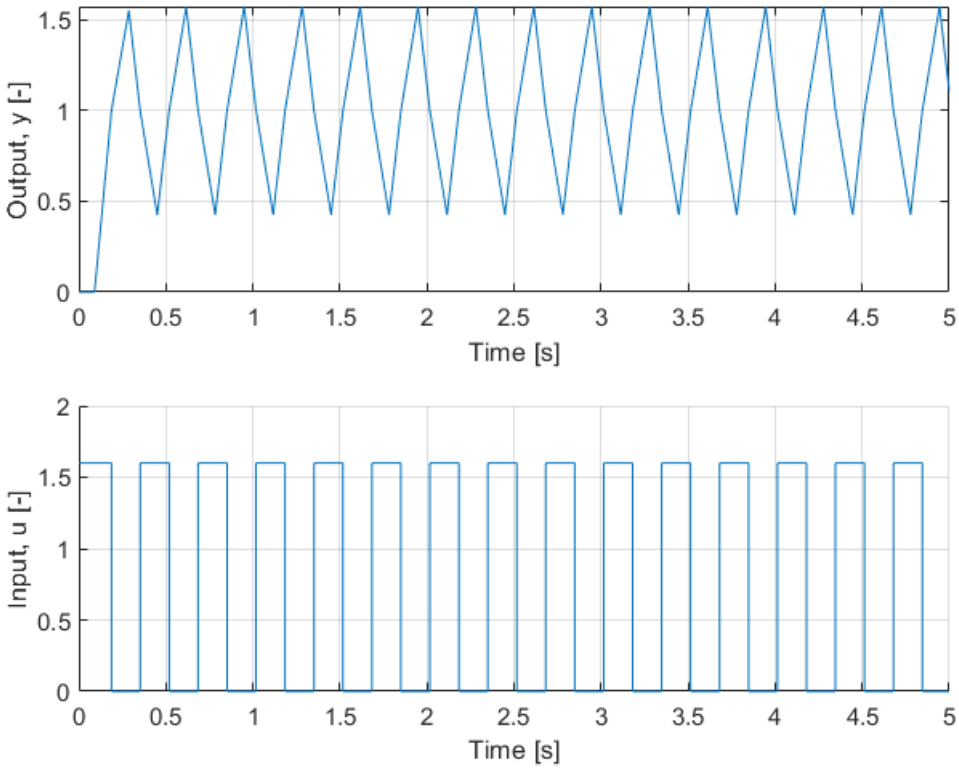


Figure 1.5: ON OFF controller without hysteresis response.

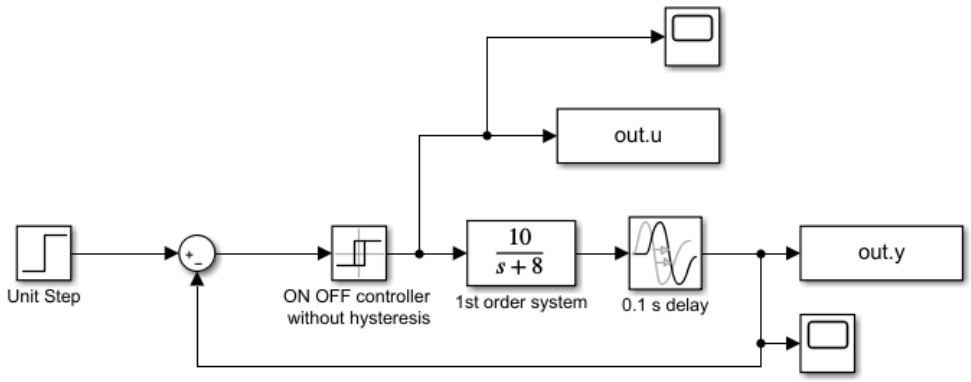


Figure 1.6: Controlled first order system with 0.1 s delay.

As it can be seen in figure 1.5, the controller stays on until the output of the system is bigger than one (which is the setpoint of the system), because of the delay the system tends to keep increasing until it reaches a maximum and then it decreases. Once the output is smaller than the setpoint, the controller activates the plant. As it happened before, because of the delay the output keeps on decreasing until it starts to increase, beginning the cycle once more.

1.1.4.2 ON OFF Controller with Hysteresis

While the previous controller might work well in a system like the one in which it was applied, it can be guessed that if the system did not have a delay and the output was decreasing right after the controller sent the OFF signal, it would mean that the plant would have to change between ON and OFF states with a very high frequency. This phenomena is known as chattering [4], and it is a problem because the actuator switches many times in a short period, which reduces the useful life of the instrument. This happens because the controller requires actuators with only two positions.

The hysteresis consists on applying an upper and lower margin to the setpoint, this is the hysteresis amplitude band, H. It can be guessed that if this amplitude band is big, then the oscillation period is big, whereas if it is small, the period decreases, which causes the controller to switch faster, the limit case would be to decrease the value of the amplitude band to zero, which is the ON OFF controller without hysteresis. This is better exposed in figure 1.7, in which the response of a first order system (without delay) to an unit step is shown. Here, the controller is again a relay, but the upper and lower limits of it have been defined as H/2 and -H/2, respectively.

In addition, the implemented Simulink model can be seen in figure 1.8.

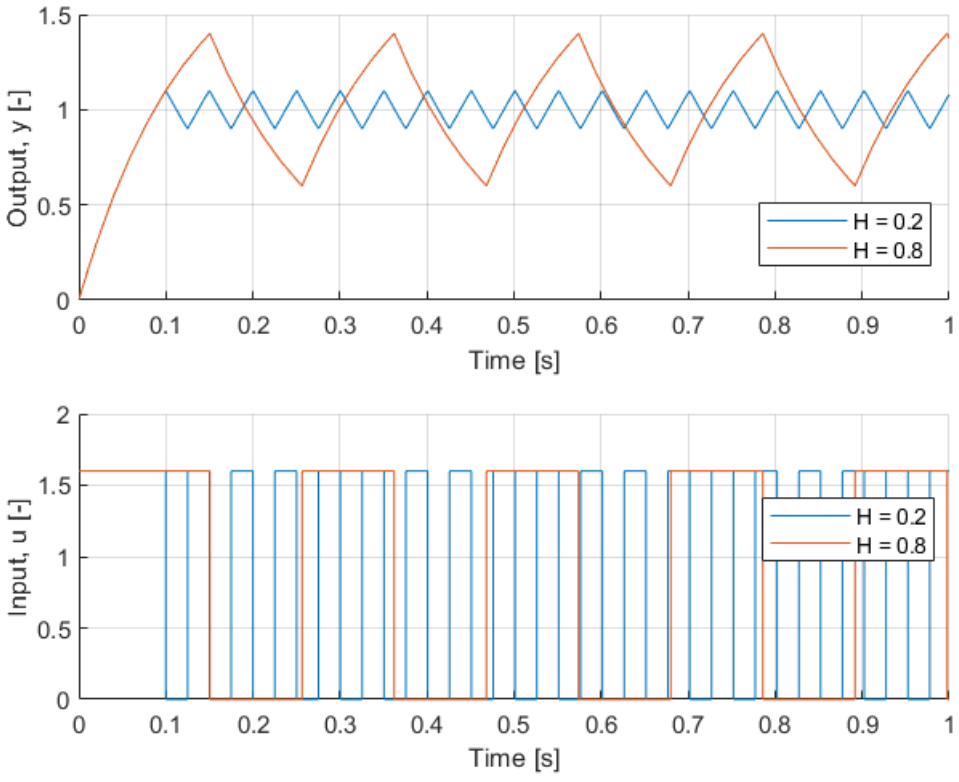


Figure 1.7: ON OFF controller with hysteresis response.

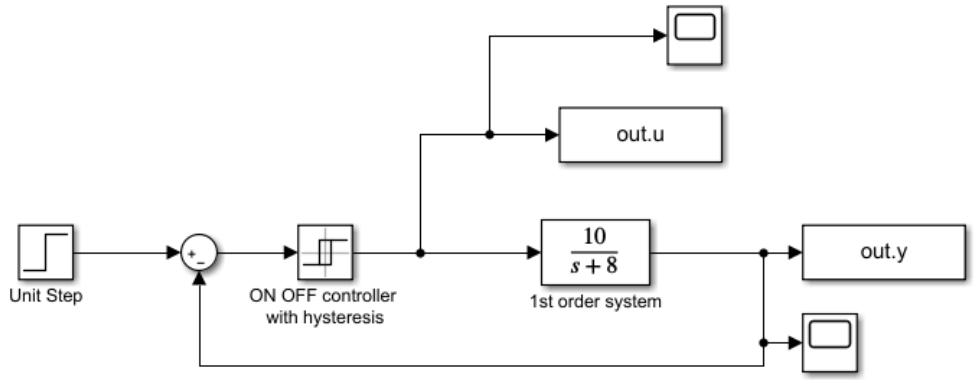
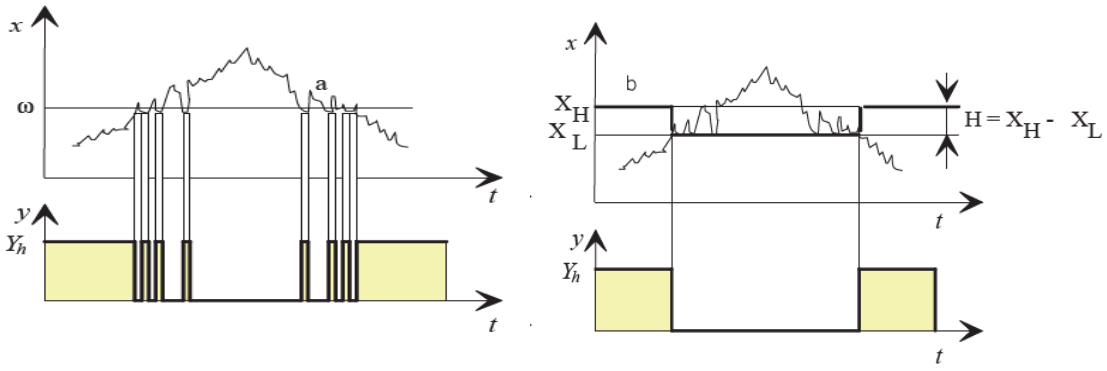


Figure 1.8: Controlled first order system.

In the presence of noise or small variations in the measured signal, adding the hysteresis to the ON OFF controller has another positive effect . Figure 1.9a, obtained from [4], shows how the switching frequency of the controller without hysteresis is very high, due to the effects of noise present, while figure 1.9b, also obtained from [4], shows how this problem is solved by adding the hysteresis to the controller, thereby preventing the actuator from being subjected to actions that could promote early aging or even damage. For example, the continuous opening and closing of a control valve causes deterioration of moving parts, and shortens their useful life.



(a) Without hysteresis

(b) With hysteresis

Figure 1.9: Noise effect on the ON OFF controller response, [4].

It can be advanced that in this project, the ON OFF controller developed has hysteresis, to avoid having high switching frequencies.

1.1.5 PID Controller

The PID (Proportional Integral Derivative) controller is a simultaneous control mechanism that uses feedback of the output signal. This type of controller is widely use by the industry in any field that requires automation. The input in the controller is the difference between the reference value and the output of the plant.

The PID algorithm consists of three different constants, the proportional or gain, the integral and the derivative. The proportional component takes into account the actual

error, the integral component takes into account the past error and the derivative component takes into account the future error. The three parameters are sum up to calculate the final output. The mission of the designer is to tune the controller, this is choose the right value of the constants, so that the answer of the system after a change in the reference behaves in the desired way.

Figure 1.10 shows a very simple scheme of a PID control, as it can be appreciated, the answer of the controller is the sum of the three terms. Furthermore, the plant has been divided into actuator and process, where the actuator is the component providing the energy to change the state of the overall system, and the process is the component to which the energy is been applied to. It is necessary to mention that in the real world, the plant is not perfect and even if it has been considered to be linear this does not always happen. In addition, the actuators have a limit on the maximum and minimum values that they can apply.

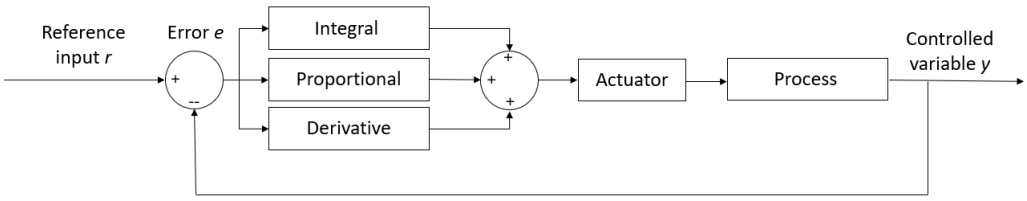


Figure 1.10: Components of a PID control system.

The PID controllers have the main advantage of being easy to tune, this provides the designer with a good control of the system dynamics. On the other hand, the main disadvantages are, the lack of adaptability and robustness due to changes in the dynamics, and the fact that they are very hard to design for MIMO (Multiple-Input Multiple-Output) control systems.

1.1.5.1 PID Historical background

PID controllers were firstly implemented in electronic devices, since it was easy to change the PID parameters to modify its behavior. Later, the PID controllers were introduced to automatically control ship steering, one of the oldest examples of these controllers is the one developed by Elmer Sperry in 1911, meanwhile, the first theoretical analysis of a PID controller was published by the Russian-American engineer Nicolar Minorsky in 1922. Minorsky was designing automatic ship steering systems and he realized that in order to do so he was taking into account the past, present and future error. In addition, he focused on giving stability to the system, rather than general control, this facilitated the problem a lot.

Meanwhile, the aviation industry did not stay out of this technology, seeing the first applications of PID controllers in the 1930's. What is more, PID controllers continue to fulfill 95% of the aerospace industrial applications, just like in the 1960's [1].

1.1.5.2 PID Theory

The components of a PID controller are:

- P - proportional term, it acts on the velocity of the answer, but it can't vary the steady state error.
- I - integral term, it eliminates the steady state error, but it worsens the transitory response.
- D - derivative term, decreases the overshoot, this improves the transitory term.

In the time domain, the equation that describes the output of the controller, *u(t)*, with respect to the error or input of the controller, *e(t)*, is:

$$u(t) = K_P e(t) + K_I \int_0^t e(t)dt + K_D \dot{e}(t) \tag{1.1}$$

where, K_P is the proportional gain, K_I is the integral gain, and K_D is the derivative gain. These two last gains can also be defined in terms of K_P, the integral time constant, T_I, and the derivative time constant, T_D as:

$$K_I = \frac{K_P}{T_I} \; ; \; K_D = K_P T_D \tag{1.2}$$

With these two new definitions, the equation of *u(t)* can be rewritten as:

$$u(t) = K_P(e(t) + \frac{1}{T_I} \int_0^t e(t)dt + T_D \dot{e}(t)) \tag{1.3}$$

As it can be seen in equation 1.3, the algorithm for the PID controller is very simple, this explains its wide use.

The difficulty of the PID controller eradicates in the adjustment of the PID parameters. The objective of the adjustments of the PID parameters is to achieve a correct and efficient control loop in the shortest time. If the parameters of the PID controller (the gain of the proportional, integral and derivative) are chosen incorrectly, the controlled process may become unstable, this means that its output varies, with or without oscillation, and is limited only by saturation or mechanical breakage.

1.1.5.3 PID Tuning

The PID parameters adjustment, or PID tuning, can be done in several ways. Firstly it is necessary to know if the system that wants to be controlled is well-behaved, a not well behaved system has one of the following characteristics: is highly non linear; is unstable; has a lot of delay in the system; or it is non-minimum phase [3]. If the system is well behaved there are two possibilities of tuning, if the design is being performed with a physical hardware, it is possible to apply a heuristic method, such as the Cohen Coon or Ziegler Nichols methods, which will be explained later. These heuristics methods give an initial guess with which it is possible to tweak the gains manually.

If a model of the system is available - even if it is not, it is possible to derive it from the physical system if the equations that governed the process are known, or by using MATLAB's System Identification Toolbox to obtain an estimated transfer function from an input sequence on the physical system-. Once the model is available

the possibilities for the tuning are: using the automatic tuning methods, such as the automatic tuning app from MATLAB, or using the manual tuning methods, pole placement, loop shaping, or heuristic methods. Again here, it is necessary to refine the response of the system by manually tweaking the response. Figure 1.11 shows a PID tuning guide in a more schematic way.

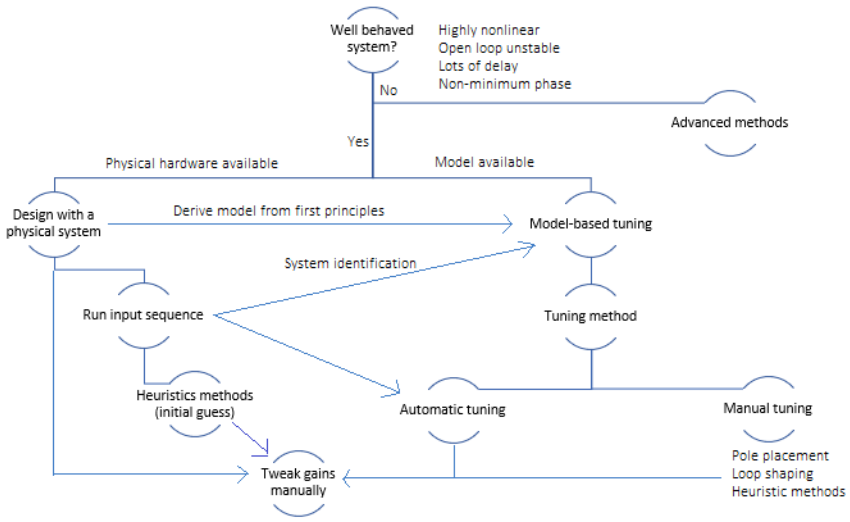


Figure 1.11: PID tuning guide.

From figure 1.11, it can be appreciated that all the strategies end up having to tweak the gains manually, usually the responses of the system have some requirements they need to fulfill. Table 1.1 shows the effects of increasing the gains in the main characteristics of a system response. It is important to know what happens when the gains are changed to be able to fulfill the requirements of the response. For example, if the overshoot of the response of the system is too big, by looking at table 1.1 it is able to understand that K_D needs to be increase.

However, it is important to highlight the necessity to have small gains, this is due to the fact that big K_P and K_I values can make the system unstable and increase the steady state error. On the other hand, big K_D values can result in an excessive overshoot and degrade stability [1].

Table 1.1: Effect of increasing the PID parameters in system response, according to [1].

	Rise time	**Overshoot**	**Settling time**	**Steady state error**	**Stability**
Increasing K_P	Decrease	Increase	Small Increase	Decrease	Degrade
Increasing K_I	Small decrease	Increase	Increase	Large decrease	Degrade
Increasing K_D	Small decrease	Decrease	Decrease	Minor change	Improve

The PID tuning strategy followed in this project was; firstly, since the physical hardware was available, an input sequence was run on the system to study its response to a step. With this response it was possible to derived a model of the system, as well as to obtain an initial guess with the use of heuristic methods, in this project the methods used were the Cohen-Coon and Ziegler-Nichols open-loop methods. The system was then implemented into Simulink, and with the initial guesses and the

transfer function of the system, it was possible to tweak the gains manually to obtain the desired response of the system when achieving a certain setpoint. Finally, the gains were introduced in the physical system to see if the response corresponded to the simulated one, and to finish adjusting the gains. Nevertheless, all of this process will be better explained in Chapter 2.

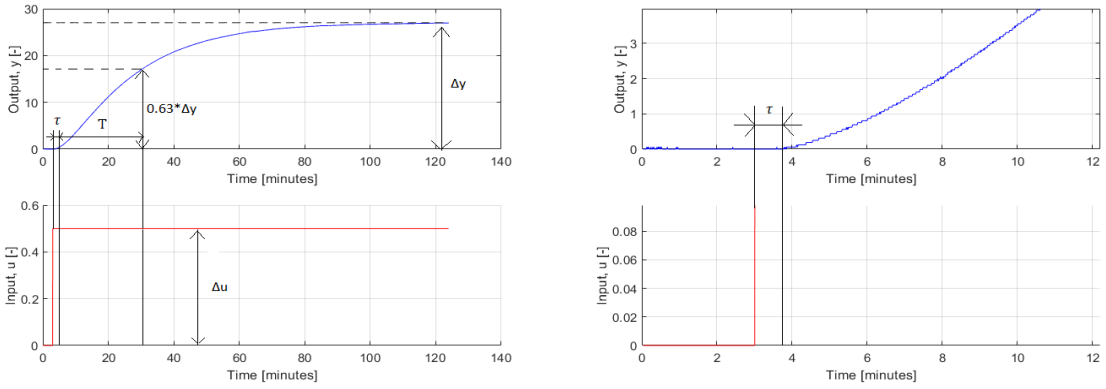
1.1.5.4 Cohen-Coon Open-Loop Method

As stated in the previous section, as part of the PID tuning, it is possible to use heuristic methods to have an initial guess of the gain values. One of the methods used in this project is the Cohen-Coon open-loop method, this method was firstly explained in [5].

This method's process is the following:

1. Wait for the process to reach steady state
2. Introduce a step in the input
3. Based on the output, obtain the following variables:
    - Delay τ
    - Amplitude T
    - Process static gain $\mu = \frac{\Delta y}{\Delta u}$

Figure 1.12a shows how this parameters are obtained based on the output, as it can be seen τ is the delay in the output - figure 1.12b zooms in figure 1.12 to better show the delay in the response -, T is the time that it takes the response to arrive to 63% of the value of Δy, with Δy being the increment in the controller variable caused by the step. Finally, μ is the process static gain, and it is calculated as $\mu = \frac{\Delta y}{\Delta u}$, as stated before.



(a) (b) Zoom in.

Figure 1.12: Open-loop tuning method parameter definition.

Table 1.2 shows how the PID gains are evaluated. It is necessary to note than depending on the type of controller that wants to be designed the estimated values change.

Table 1.2: Cohen-Coon open-loop method gains.

	$\mathbf{K}_P$	$\mathbf{T}_I$	$\mathbf{T}_D$
P controller	$\frac{3T+\tau}{3\mu\tau}$	-	-
PI controller	$\frac{10.8T+\tau}{12\mu\tau}$	$\tau\frac{30T+30\tau}{9T+20\tau}$	-
PID controller	$\frac{16T+3\tau}{12\mu\tau}$	$\tau\frac{32T+6\tau}{13T+8\tau}$	$\frac{4T\tau}{11T+2\tau}$

The advantages of this method include, the fact that it can be used for systems with time delay, and a quicker closed loop response time, [7].

1.1.5.5 Ziegler-Nichols Open-Loop Method

The Ziegler-Nichols open-loop method is the other heuristic method used in this project, with the aim of improving the performance of the PID controllers, the parameters of it need to be selected carefully. Ziegler and Nichols proposed some rules to determine the value of K_P, K_I, and K_D, [6].

This method's process is very similar to the Cohen Coon method, since its biggest difference eradicates on the way the methods estimate the gain values from the response parameters. The method follows the process below:

1. Wait for the process to reach steady state
2. Introduce a step in the input
3. Based on the output, obtain the following variables:
    - Delay τ
    - Amplitude T
    - Process static gain $\mu = \frac{\Delta y}{\Delta u}$

Figure 1.12 still applies in obtaining the delay, amplitude, and process static gain. Once these parameters have been obtained, the values of K_P, T_I, and T_D can be obtained with the formulas in table 1.3.

This method counts with some advantages, such as, the speed and the easiness to use it, its robustness, and its popularity. On the other hand, one of its disadvantages is the fact that it does not hold for I, D, and PD controllers, [7].

Table 1.3: Ziegler-Nichols open-loop method gains.

	$\mathbf{K}_P$	$\mathbf{T}_I$	$\mathbf{T}_D$
P controller	$\frac{T}{\mu\tau}$	-	-
PI controller	$\frac{0.9T}{\mu\tau}$	3τ	-
PID controller	$\frac{1.2T}{\mu\tau}$	2τ	0.5τ

1.2 Arduino

Arduino is a free software and hardware development company, as well as an international community that designs and manufactures hardware development boards to build digital devices and interactive devices that can detect and control real-world objects. Arduino focuses on bringing together and facilitating the use of electronics and embedded systems programming in multidisciplinary projects. The products that the company sells are distributed as free software and hardware. The boards can be assembled by hand or purchased preassembled; the open source IDE (Integrated Development Environment) can be downloaded for free from www.arduino.cc, [8].

Arduino is composed of two major parts: the Arduino board, which is the piece of hardware you work on when you build your objects; and the Arduino IDE (Integrated Development Environment), the piece of software you run on your computer. You use the IDE to create a sketch (a little computer program) that you upload to the Arduino board. The sketch tells the board what to do, [8].

1.2.1 Arduino Uno

Arduino counts with a series of hardware designs, which can be downloaded for free, as stated before. Since the Arduino board used for this project has been the Arduino Uno, it will be explained a little more in detail.

The Arduino Uno is a small microcontroller board, which is a small circuit (the board) that contains a whole computer on a small chip (the microcontroller). This chip with 28 "legs" is the ATmega328, and it can be considered as the heart of the Arduino Uno.

The main advantage of Arduino eradicates in the fact that it counts with all the components necessary to work and communicate with a computer. Figure 1.13 shows a sketch of an Arduino Uno.

As seen in figure 1.13, Arduino Uno counts with a serial of connectors, every one of them with a different function. The connectors of the Arduino Uno can be classified as:

- 14 digital IO pins, these are the pins 0-13. They can be defined as inputs or outputs, this needs to be specified in the IDE sketch.

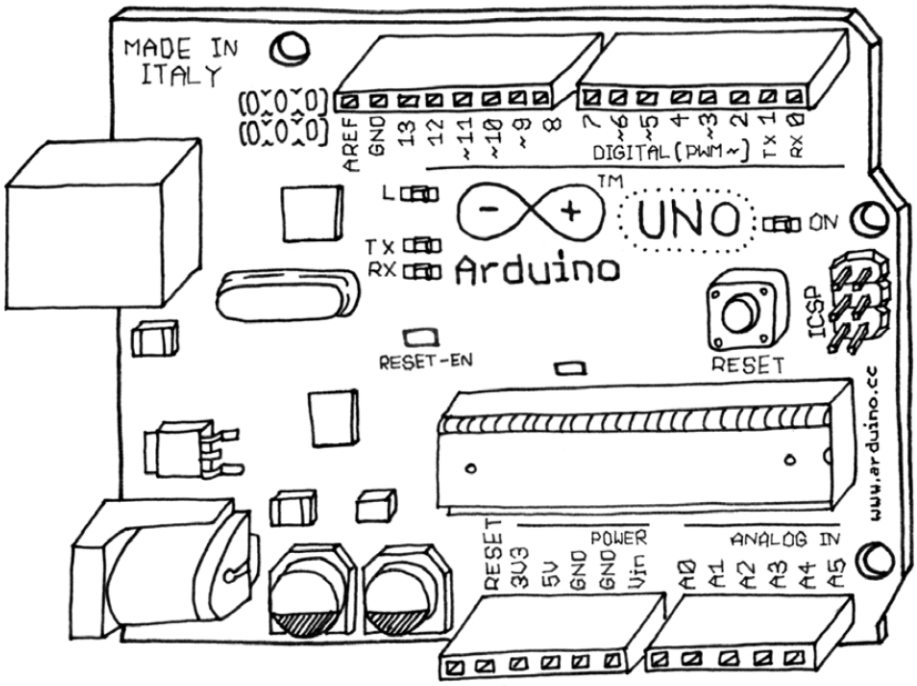


Figure 1.13: Arduino Uno sketch, [8].

- 6 analog In pins, these are the ones marked as A0-A5 in the board. They convert analogue values- i.e., a voltage reading from a sensor- into a number between 0 and 1024.
- 6 analog Out pins, these are the pins 3, 5, 6, 9, 10, and 11. In reality these are digital pins that can be reprogrammed for analogue output. This is done thanks to the Pulse Width Modulation (PWM), which will be explained further in detailed in section 1.2.3.

In addition, the board can be powered from:

- The computer's USB port.
- Most USB chargers.
- An AC adapter (it is recommended to use a 9 V source with a 2.1 mm barrel tip).

In the case of having both, a power supply connected to the power socket and a computer connected to the USB port, the power will come from the power supply. However, if the power supply is disconnected, then the power will be supplied through the USB port.

For the system to work properly, it is necessary to attach to the board some sensors and actuators. The microcontroller in the Arduino board is a very simple computer that allows it to process electric signals. For it to sense physical quantities, i.e., light, or temperature, it uses the help of sensors. These sensors transform the physical quantity into something the microcontroller can understand, an electric signal.

On the other hand, the microcontroller needs the help of the actuator to be able to react. Even if the decision-making process is performed by the microcontroller, its

reaction is caused by the actuators. The actuator could be a LED or an electric motor.

1.2.2 Arduino IDE

The IDE (Integrated Development Environment) is a special program that allows the user to write sketches for the Arduino boards in a simple language modeled after the Processing language [8, 10]. When the upload button of the IDE is clicked, the code written on the sketch is automatically translated into C language, and is passed to the avr-gcc compiler, to make it understandable for the microcontroller.

The programming cycle in Arduino is very simple, once the board is connected to a USB port of the computer, it is possible to upload the written sketch into the board. After a couple of seconds the sketch is uploaded into the board and being executed.

Since the aim of this thesis is not to write an Arduino reference book, only the basic structure of Arduino sketches will be presented. The basic structure of Arduino programming language is very simple and it is divided in at least two parts, these are:

- setup(), the setup() function is called once when the program starts, the pin modes or the begin serial code lines have to be written here. The setup function should follow the declaration of any variables at the very beginning of the program [9].
- loop(), the loop() function should be called after calling the setup() function. As suggested by its name, the function loops consequently, allowing the program to respond change, respond and control the Arduino board [9].

This two parts are so essential, that when a new sketch is created in the IDE, the following code is already written:

```
1   void setup() {
2       // put your setup code here, to run once:
3
4   }
5
6   void loop() {
7       // put your main code here, to run repeatedly:
8
9   }
```

One of the main advantages of using Arduino is the wide community of users. Thanks to this community, thousands of libraries are available. Libraries are pieces of code made by third parties that can be used in a sketch. This makes programming in Arduino easier, as it is not necessary to lose time and resources in programming something that has been programmed before.

To use a library that has just been installed, it is only necessary to read the documentation for that library, if it is available, and then read and test the examples available in the library [19].

1.2.3 PWM

The PWM (Pulse Width Modulation) allows the Arduino boards to send analogical outputs with just digital means. Digital control is used to create a square wave, this is a signal that switches between on and off states, in the Arduino Uno, the on state corresponds to 5 V and the off to 0 V. A PWM frequency is defined, this allows us to know the PWM period, this means that the PWM will turn on and off the signal repeatedly with the same frequency. It is possible to vary the mean value of the output by changing the amount of time that the signal is on, this on time is called the pulse width, versus the time in which the signal stays off. Changing the value of the output is equal to vary the analog value, so by changing the pulse width an analog control is possible.

PWM (Pulse Width Modulation) of a signal or power source is a technique in which the duty cycle of a periodic square signal is modified, either to transmit information through a communications channel or to control the amount of energy that is sent to a load. In the case of Arduino, the PWM allows it to send analogical outputs with just digital means. Digital control is used to create a square wave, this is a signal that switches between on and off, in the Arduino Uno, the on value corresponds to 5 V and the off value to 0 V.

The PWM frequency of Arduino is equal to 500 Hz, this means that the PWM period is equal to 2 ms. The amount of time that the signal is on in every period is called the pulse width. It is possible to vary the mean value of the output by changing the pulse width. Changing the value of the output is equal to vary the analog value, so by changing the pulse width an analog control is possible.

The way in which the Arduino board is told how big the pulse width should be is through the duty cycle. The duty cycle of a periodic signal is the relative width of its positive part relative to the period. Mathematically expressed as:

$$D = \frac{\tau}{T} \tag{1.4}$$

where, D is the duty cycle, τ is the pulse width, and T is the PWM period.

In Arduino, the analogwrite() function is used to tell Arduino the duty cycle. The input of the analogWrite() function should be a number on a scale of 0 - 255, such that analogWrite(255) requests a 100% duty cycle (always on), and analogWrite(127) is a 50% duty cycle (on half the time) for example, [11]. Figure 1.14 shows this is a more graphic way, as it can be appreciated, the PWM period equals to 2 ms and depending on the duty cycle the amount of time that the signal is on changes.

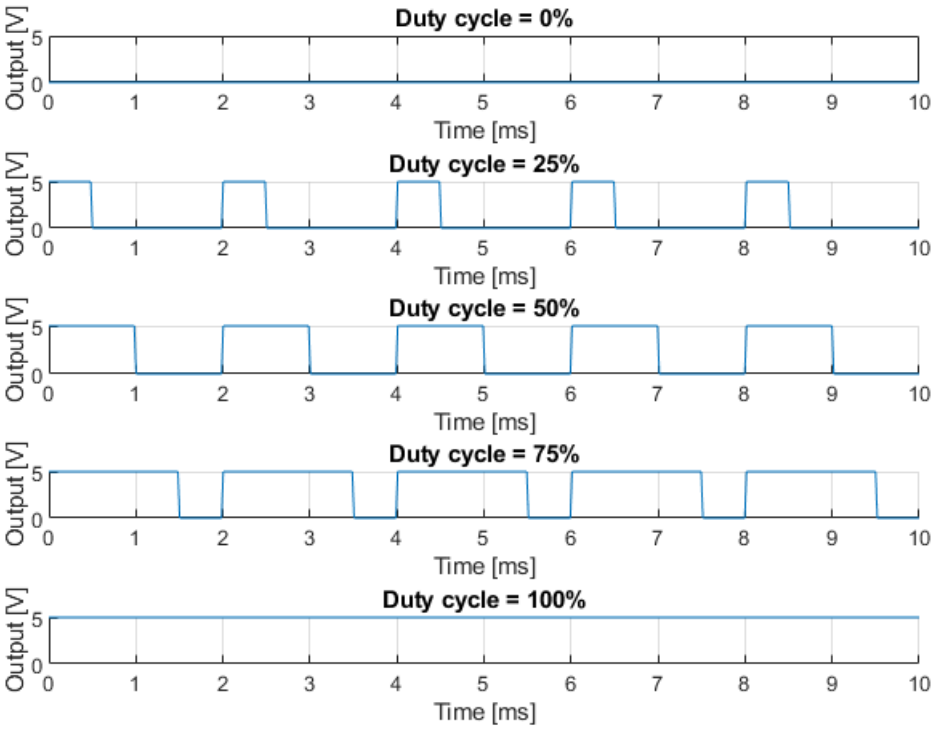


Figure 1.14: Arduino PWM functioning.

Chapter 2

Development

2.1 System description

In this chapter the steps followed for the development of the control system will be explained in detail, i.e., the power electronics circuit, the controller strategies, the process, and the implementation of the physical system with the controller.

Figure 2.1 shows the general scheme of what wants to be achieved. In this case, the controller is the Arduino and the task here is to program the control strategy. The actuator is an electric resistor and the sensor is a thermistor. Since this is just a prototype the process is just made up of a 400 ml beaker filled with 150 ml of water. To fully understand the development of the whole system, it is necessary to take a look at each subsystem.

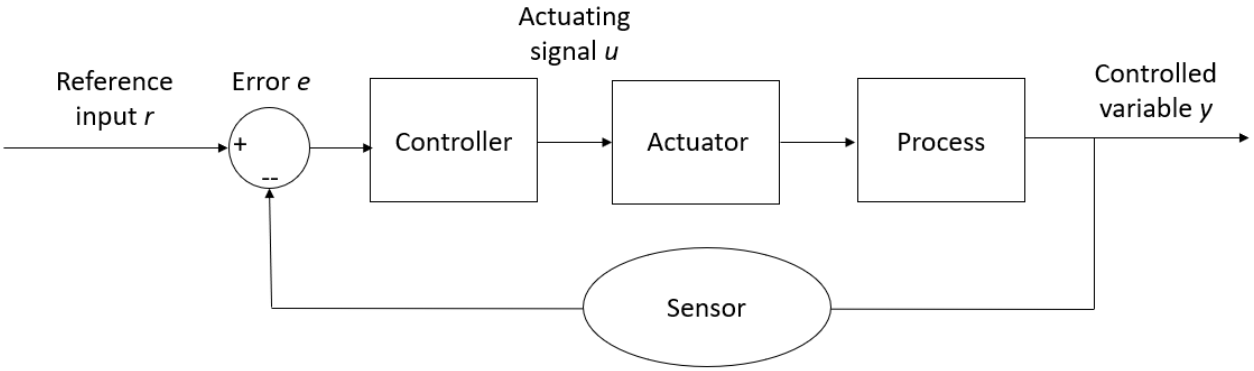


Figure 2.1: Functional scheme of the system.

2.2 Thermistor Subsystem

As stated before, the thermistor subsystem is the sensor of the system and is in charge of measuring the temperature of the water. A thermistor is a type of resistor which resistance is strongly dependent on temperature, more so than in standard

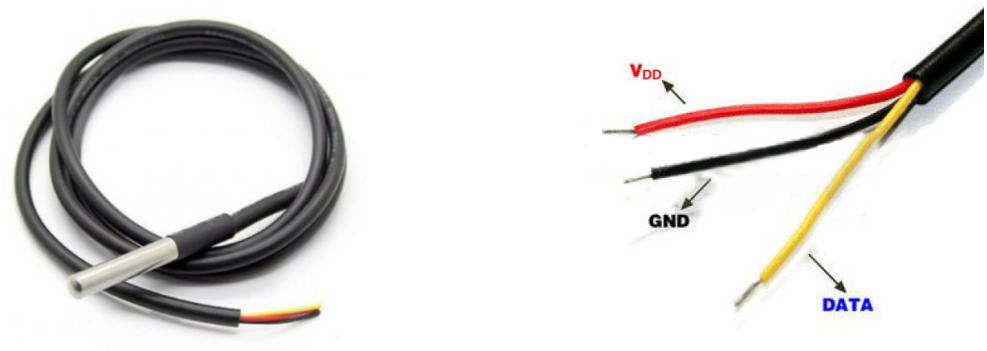
resistors. The word is a combination of thermal and resistor. The relation between temperature and resistance is so big that it is possible to draw a R-T curve.

The thermistor used in this thesis is the DS18B20 in the shape of a probe, so that it can be submerged under liquids. The DS18B20 is a Programmable Resolution 1-Wire Digital Thermometer and is widely used.

The main characteristic of this probe are, [16]:

- Measures Temperatures from -55°C to +125°C.
- Only needs one pin for communication with the controller.
- ±0.5°C Accuracy from -10°C to +85°C.
- Supply voltage from -3 V to 5 V.
- Stainless steel probe head makes it suitable for any wet or harsh environment.

Figure 2.2a shows an image of the actual sensor. As it can be seen in figure 2.2b, the sensor counts with the three wires, the black wire has to be connected to the ground, the red one to the power supply, and the yellow one to the microcontroller pin.



(a) DS18B20 image. (b) Wires.

Figure 2.2: Dallas DS18B20 temperature sensor.

2.2.1 Thermistor Circuit

Figure 2.3, obtained with [13], represents the way to connect the sensor to the Arduino board. As it has been stated before the input voltage is from 3 V to 5.5 V, this is an advantage when using Arduino since the 5 V pin from Arduino can be used as power supply. As it can be seen in the scheme, the ground wire (black) is directly connected to the Arduino ground pin, and the V_{DD} wire (red) and the data wire (yellow) are directly connected to the 5 V pin and one of Arduino's digital pins, respectively. In addition, for the data signal to be processed correctly, a 4.7 kΩ resistor mas be placed in between the V_{DD} and the data wires.

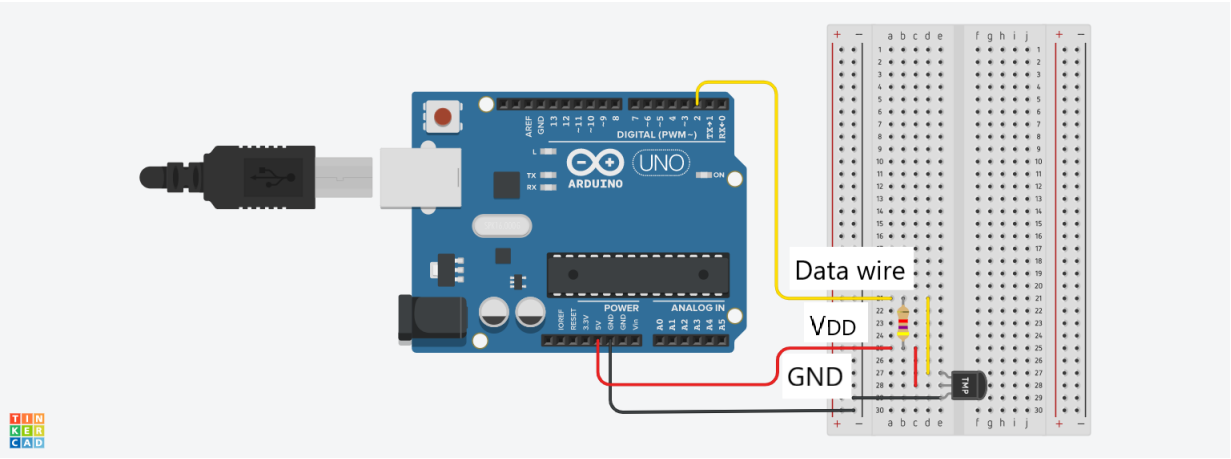


Figure 2.3: Scheme of the temperature sensor subsystem.

It is necessary to point out that in figure 2.3, the temperature sensor is represented as the three pin component with TMP written on it. On the other hand, figure 2.4 shows an image of the real subsystem.

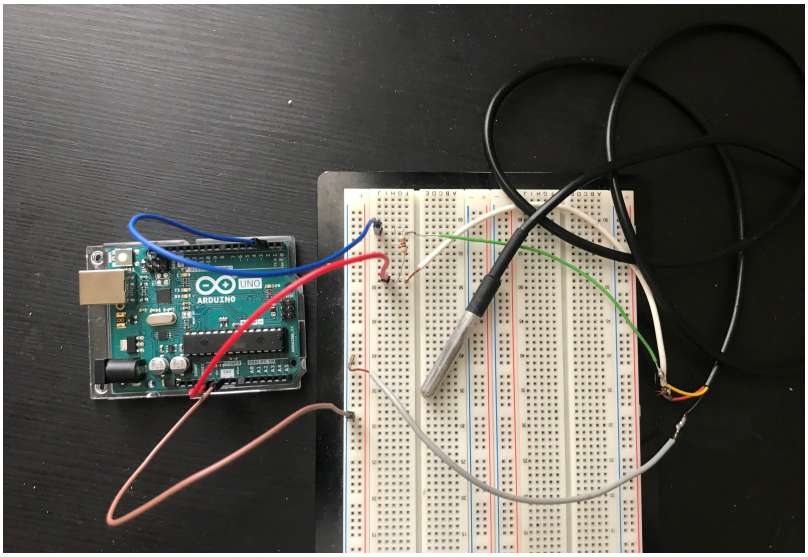


Figure 2.4: Image of the temperature sensor subsystem.

2.2.2 Thermistor Code

Once the connections between the temperature sensor and the Arduino board have been made, it is necessary to program the Arduino so that it can translate the voltage signals of the thermistor into a temperature value.

Before it is explained how it has been done in this case, it is important to understand how a thermistor works. As stated before, the thermistor is characterized by a strong relation between resistance and temperature, since the change in the resistance value cannot be measured directly, what is actually measured is the voltage of the data pin.

The T-R relation of the thermistor can be modeled with the Steinhart-Hart Equation, the equation is shown below:

$$\frac{1}{T} = A + Bln(R_t) + C(ln(R_t))^3 \tag{2.1}$$

Where T is the temperature, R_t is the resistance of the thermistor, and A, B and C are thermistor coefficients. This coefficients are usually given by the manufacturer, but they can also be calculated with the use of an external thermometer - once three points of the R-T curve are known, obtaining the coefficients from equation 2.1 is very simple-.

Since the circuit of the thermistor is known, it is possible to transform this voltage value into a resistance value. Then, the resistance value is introduced into the Steinhart-Hart equation and the temperature value is found.

Thankfully, all this process was not necessary since the thermistor is widely used and Arduino counts with libraries that can be included in the code in other to avoid having to program all this process.

More specifically it is necessary to install and include two libraries, these are the DallasTemperature.h and the OneWire.h libraries. The DallasTemperature.h contains commands for getting temperature measurements from the sensor. On the other hand, the OneWire library is used for communication with any one-wire device, [17]. Both of these libraries can be obtained from the Arduino Library Manager. The following code must be included in order to obtain the temperature readings from the sensor. The *sensors.requestTemperatures()* function can be placed where needed in order to obtain the temperature reading, here it has been included in the setup as an example.

```
    #include <DallasTemperature.h>
    #include <OneWire.h>
    // Data wire is plugged into digital pin 2 on the Arduino
    const int ONE_WIRE_BUS = 2;
    // Setup a OneWire instance to communicate with any OneWire device
    OneWire oneWire(ONE_WIRE_BUS);
    // Pass OneWire reference to DallasTemperature library
    DallasTemperature sensors(&oneWire);

    void setup() {
    // put your setup code here, to run once:
    sensors.begin()
    sensors.requestTemperatures();

}

void loop() {
    // put your main code here, to run repeatedly:

}
```

2.3 Resistor Subsystem

The resistor subsystem is the actuator of the control system, it receives the outputs of the controller in other to change the state of the process, in this case, the temperature

of the water. The resistor used in this project is a 12 V/ 40 W resistor. This resistor is shown in figure 2.5.

Figure 2.5: Image of resistor.

As it can be guessed, the Arduino Uno cannot be used to power the resistor since its maximum output voltage is equal to 5 V. Therefore, an external power supply of 12 V, capable of giving up to 40 W, is needed. In addition, it is necessary to design a circuit in which the output from Arduino which is only 5 V is converted into the 12 V of the resistor, this circuit can be made with the use of transistors, more specifically, in this project the design circuit counts with two BD237 BJT (Bipolar Junction Transistor) and one IRF520n MOSFET (Metal–Oxide–Semiconductor Field-Effect Transistor).

A BJT is a solid-state electronic device consisting of two PN junctions very close to each other, a bipolar transistor allows a small current injected at one of its terminals to control a much larger current flowing between the two other terminals, making the device capable of amplification or switching. The BD237 BJT counts with three terminals; emitter, collector, and base. Figure 2.6 shows the terminals of the BD237. There are two types of BJT, these are; NPN and PNP. The difference eradicates in way they are built and as a result, in the NPN transistor the output current flows from the collector to the emitter. Whereas in the PNP transistor the output current flows from the emitter to the collector. The BD237 is a NPN type BJT.

(a) (b)

Figure 2.6: BD237 BJT terminals, [14].

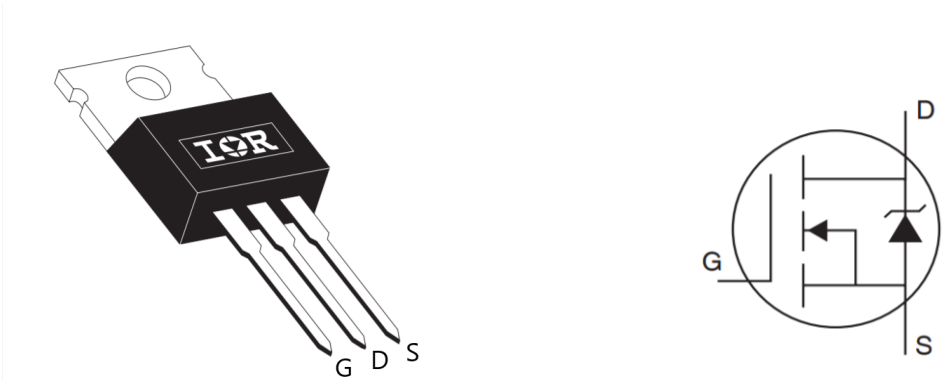
The BD237 BJT has the following characteristics, [14]:

- Collector-Emitter saturation voltage, $V_{CE(sat)}$, equal to 0.6 V - with I_C = 1.0 A, I_B

= 0.1 A-.

- Base-Emitter on voltage, $V_{BE(on)}$, equal to 1.3 V - with I_C = 1.0 A, V_{CE} = 2.0 V-.
- DC current gain equal to 40 - at T = 25°C, I_C = 0.15 A, V_{CE} = 2.0 V-.
- Maximum Collector-Emitter voltage, V_{CEO}, equal to 80 V.

A MOSFET is a type of insulated-gate field-effect transistor that is fabricated by the controlled oxidation of a semiconductor, typically silicon. The voltage of the covered gate determines the electrical conductivity of the device; this ability to change conductivity with the amount of applied voltage can be used for amplifying or switching electronic signals. MOSFET can be either Enhancement-Type or Depletion-Type and N-Channel or P-Channel. Roughly speaking, there are four different kinds, The IRF520n is a n-channel enhance type MOSFET. Just like the BD237, the IRF520n also counts with three terminals; gate, drain, and source, which are shown in figure 2.7.



(a) (b)

Figure 2.7: IRF520n MOSFET terminals, [15].

The IRF520 MOSFET has the following characteristics, [15].

- Maximum continuous drain current, I_D, equal to 9.7 A - at T_C = 25°, V_{GS} = 10 V.
- Maximum-minimum gate-to-source voltage, V_{GS}, equal to $\pm$20 V.
- Maximum power dissipation, P_D, equal to 48 W at T_C = 25°C.
- Maximum drain-to-source voltage, V_{DSS}, equal to 100 V.
- Drain-to-source resistance (on), $R_{DS_{(on)}}$, equal to 0.20 Ω.

Therefore, the BD237 is controlled with the current that enters through the base and the IRF520 is controlled with the voltage applied on the gate.

To understand how the resistor subsystem works, it is better to start by studying a circuit with just one BJT and one MOSFET. Figure 2.8, obtained with [18], shows this circuit, when the Arduino pin is HIGH (5 V output) the BJT is ON and the MOSFET is OFF. The opposite happens when the Arduino pin is LOW, in this case the BJT is OFF and the MOSFET is ON. This configuration would be enough to control the resistor system. However, this configuration presents two problems:

- The logic of the controller is inverted, because when it wants to increase the temperature (turn on the resistor), the digital output from the Arduino board is LOW.
- In case of a failure of the Arduino board, the resistor would stay on without a controller, heating up the water fluid without any control, which could be dangerous in the real system.

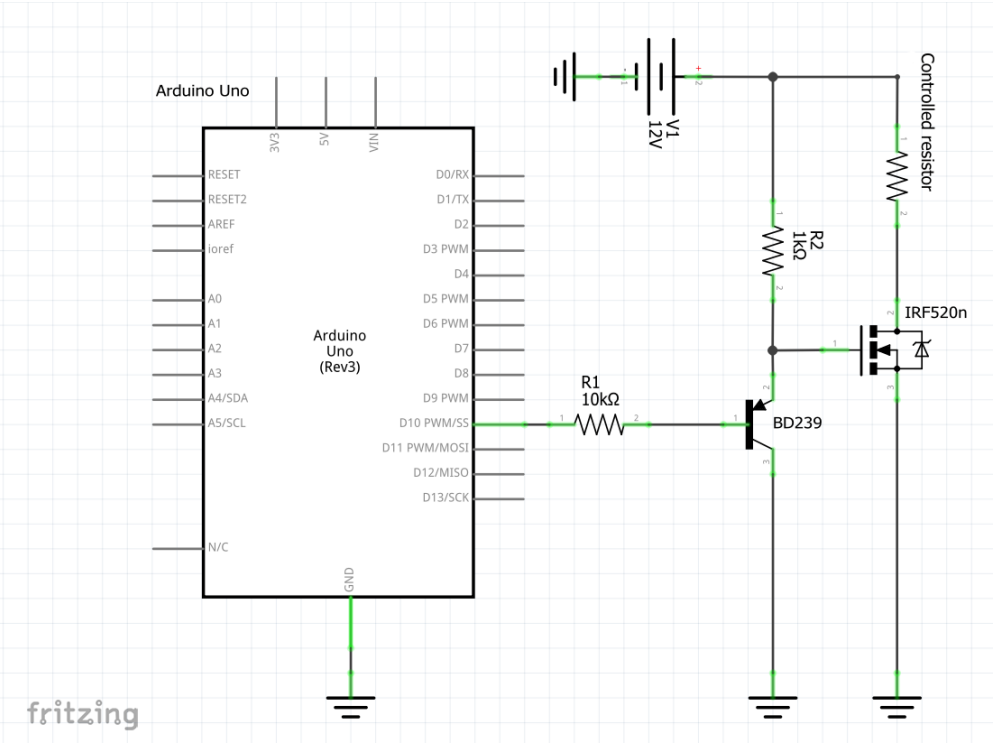


Figure 2.8: Scheme of resistor subsystem with one BD237 and one IRF520n.

These problems can be solved by adding another BJT to the system, even thought this increases the complexity of the circuit, this added complexity is small enough for the adoption of this other configuration. As stated before, the configuration that was finally adopted in this project counts with three transistors, two BD237 and one IRF520n. Figure 2.9 shows a scheme of this circuit. In this configuration when the Arduino digital output is LOW, the first BJT (by first BJT the one connected directly to the Arduino pin is intended) and the MOSFET are OFF - and no current runs through the controlled resistor-, and the second BJT is ON. On the other hand, when the digital signal is HIGH, the first BJT and the MOSFET are ON - which means that the resistor is turned on-, and no current runs through the second BJT. In addition to the transistor, in this circuit two LEDs were included, the function of these LEDs was to indicate if the current was flowing through the BJTs. This is very handy when it comes to understand if the circuit is behaving as expected during the testing, however, they could be removed from the circuit without a problem.

It is necessary to highlight the fact that with this configuration the Arduino PWM can be used. This allows for a control on the power that the resistor is applying, performing an accurate control of the temperature.

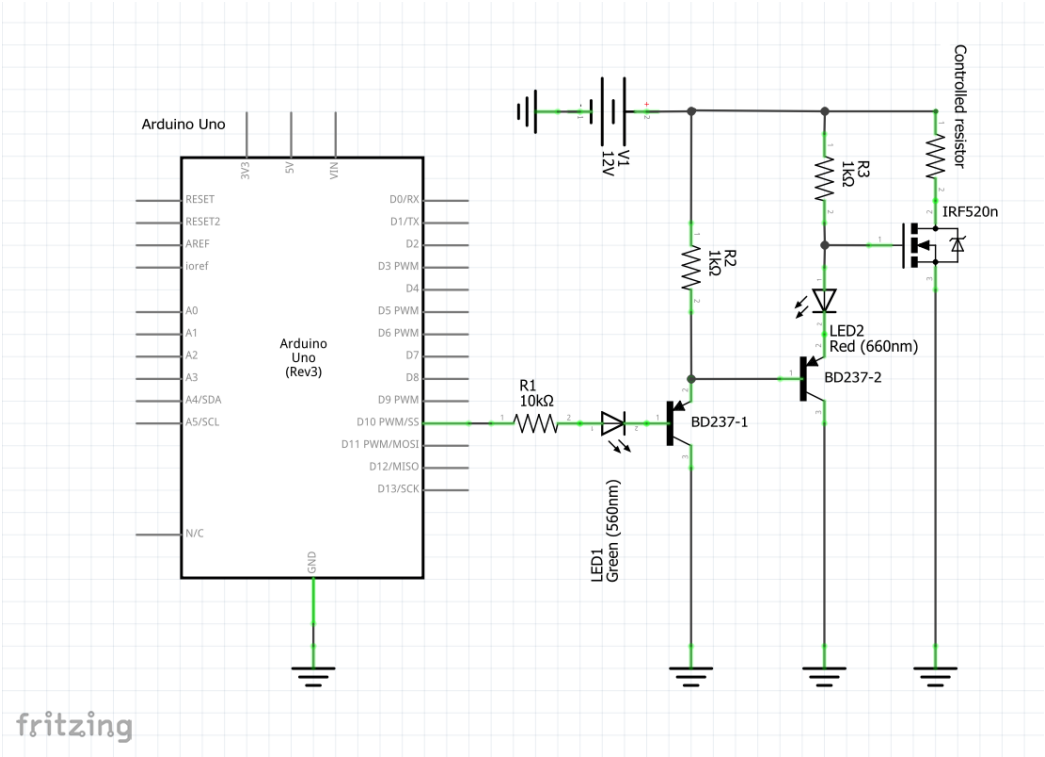


Figure 2.9: Scheme of resistor subsystem with one BD237 and one IRF520n.

For a better comprehension of the circuit, a more graphic scheme of the same is shown in figure 2.10, obtained with [13]. In this case, the power supply is represented with the battery, the BJT are represented with the components with a N written on them, and the MOSFET with the component with the NMOS written on it. The black wires are the ones that go to ground, the red ones are the ones than enter in the transistor - collector for the BJTs and drain for the MOSFET-, whereas the green wires are the connections that go into the base and gate of the BJTs and MOSFET, respectively.

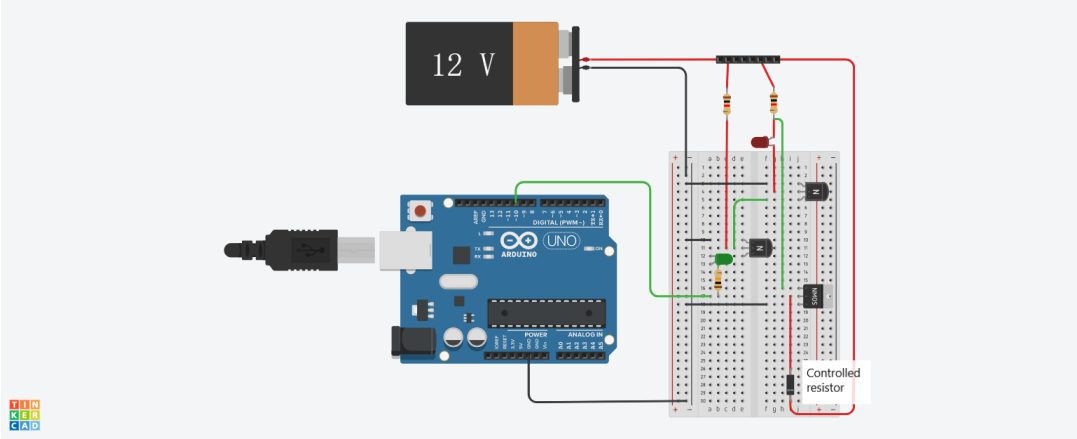


Figure 2.10: Scheme of the resistor subsystem.

Finally, figure 2.11 shows the real resistor subsystem, it is important to notice that

the MOSFET required the use of a heat sink, since the power dissipated is too high and the MOSFET would get burnt without it.

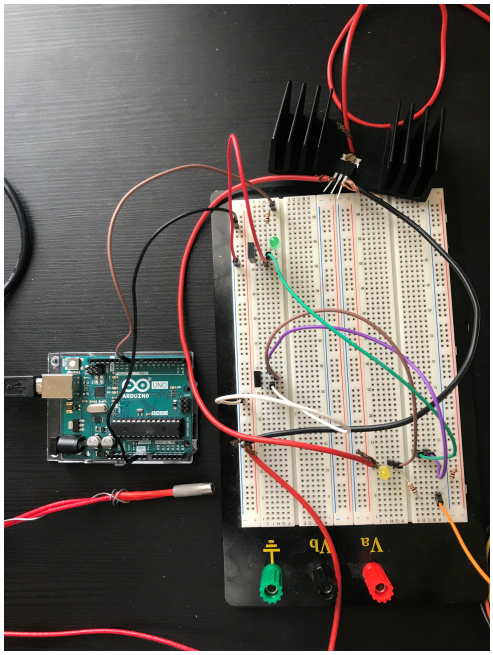


Figure 2.11: Image of the resistor subsystem.

2.4 System Implementation

The implementation of the two subsystems was very simple, since the breadboard counted with enough connections and so did the Arduino board.

As stated before, this is a prototype and the aim of it was to heat up 150 ml of water inside a 400 ml beaker. Both, the resistor and the thermistor were located inside of it in the same position to secure that every test was performed in the same way.

Figure 2.12 shows a graphic scheme of the whole system. As stated before, in the real system both the thermistor and the resistor would be placed inside the beaker. It can be anticipated that the pins in which the two subsystems are connected two are the ones that have been used in the code and during the realization of this project, these are; pin 10 for the resistor subsystem and pin 2 for the thermistor subsystem.

Figure 2.13 shows an image of the real system implemented and ready for its utilization, the power supply used can be seen in this image, as well as the beaker.

2.5 ON OFF Controller

The ON OFF controller developed in this project follows the classical ON OFF strategy. In this section, the strategy followed to program it will be explained, to assure a better understanding of the ON OFF controller code found in Appendix A.1. The only two variables that would need to be changed in this code would be the setpoint and the margin.

From the code in Appendix A.1, it is necessary to look at these lines of code:

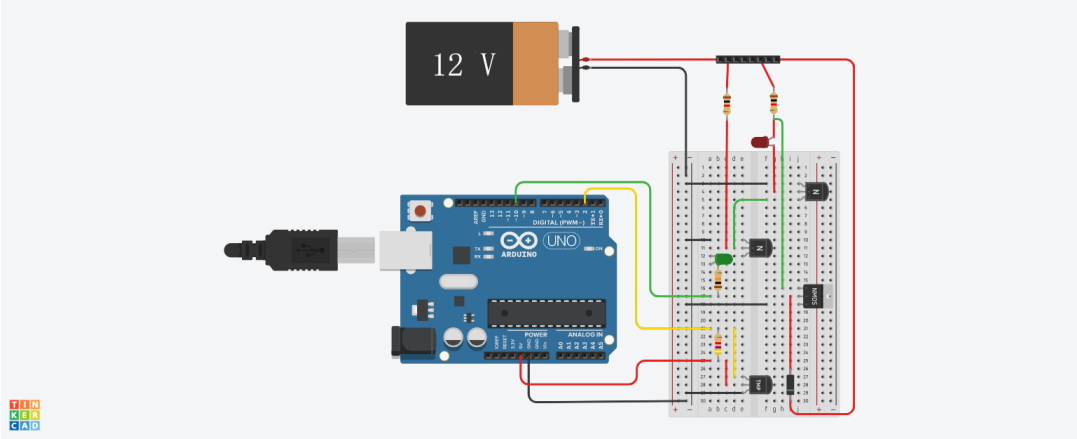


Figure 2.12: Scheme of the system.

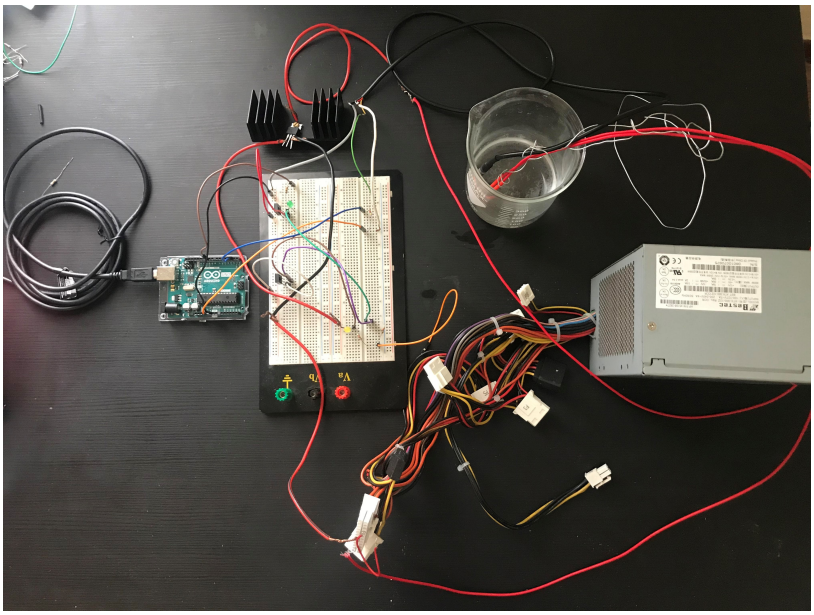


Figure 2.13: Image of the system.

```
    if ((Temperature < Setpoint + margin) && (flag == 1)) {
        digitalWrite(TRANSISTOR,HIGH);
    }
    else if (Temperature > Setpoint + margin){
        digitalWrite(TRANSISTOR,LOW);
        flag = -1;
    }

    else if ((Temperature < Setpoint - margin) && (flag == -1)) {
        digitalWrite(TRANSISTOR,HIGH);
        flag = 1;
    }
```

In order to program the ON OFF controller it is necessary to control two variables; the temperature -this is the most important since it is the variable that needs to be controlled-, and an auxiliary variable called flag, this variable can either be equal to

1 or -1, and it indicates if the temperature is increasing or decreasing. This variable is necessary because the temperature wants to be kept inside a bandwidth. If this variable was not defined, the controller would turn on the actuator until it reached the lower limit, at this point it would get turn off, and the value of the output would fluctuate around the lower limit value.

To better understand what has just been exposed, a very simple ON OFF controller has been designed, in the imagined system, if the actuator is OFF, the controlled variable, y, will decreased by 0.1 per second. On the other hand, when the actuator is on, the controlled variable will increase at a rate of 0.1 per second. The setpoint has been set to 5, and the margin has been set to 1. Figure 2.14 shows the case in which the variable flag is not defined. As it can be appreciated, the output keeps on fluctuating around the value of 4. Basically, without the flag, the controller is an ON OFF controller without hysteresis, which is not the aim of this project. In addition, the high frequency switching should be avoided in order to avoid chattering.

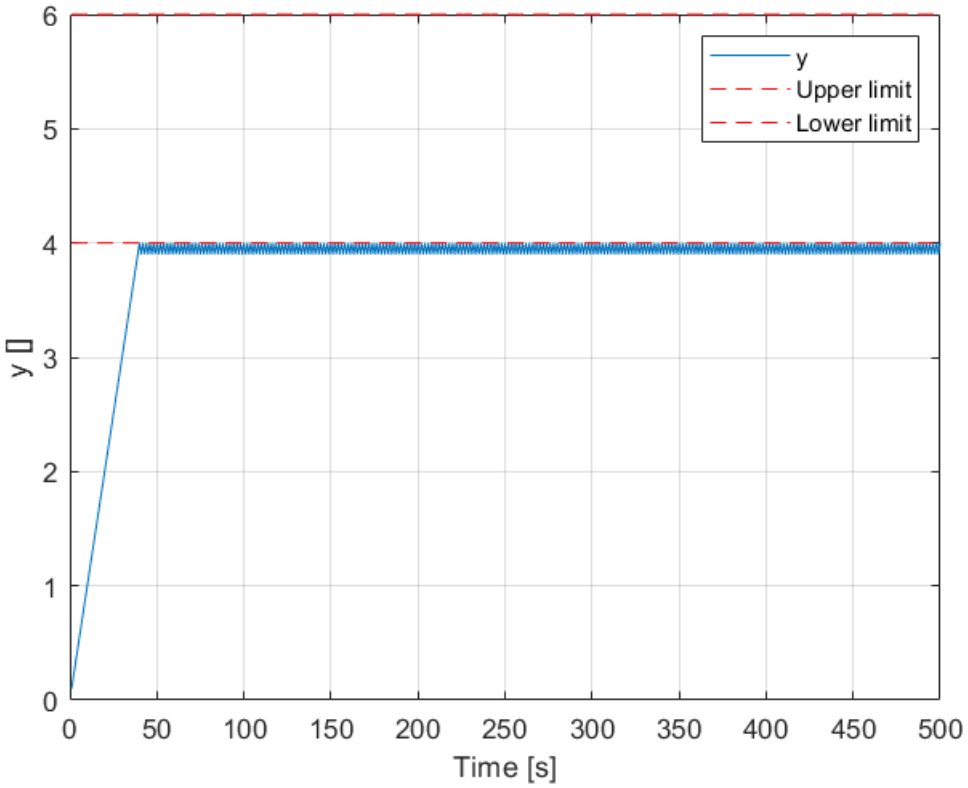


Figure 2.14: Example of ON OFF controller without the variable flag.

On the other hand, if the flag value is correctly defined -meaning that the controller increases the controlled variable until it reaches upper limit, and once reached this point decreases until it reaches the lower limit-, the controller will work as expected. Figure 2.15 shows the ON OFF strategy implemented into the simple system.

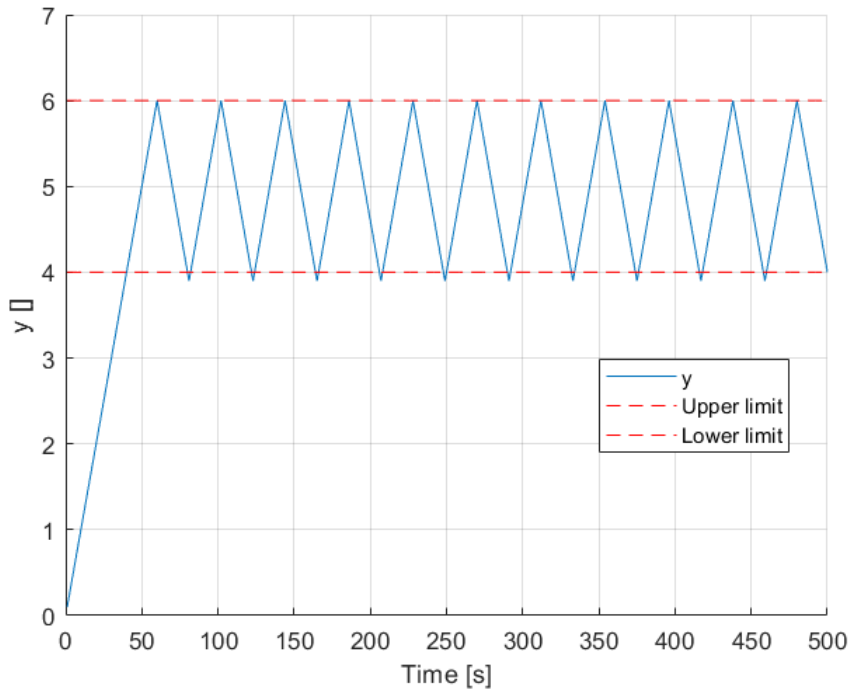


Figure 2.15: Example of ON OFF controller.

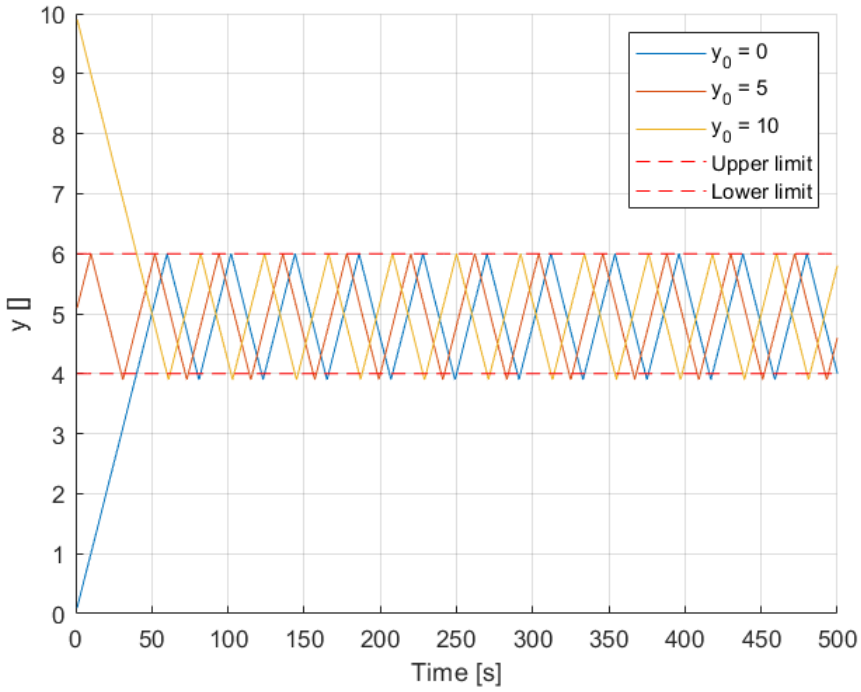


Figure 2.16: Example of ON OFF controller for different initial values.

However, it is still necessary to check if the controller will work if the initial point is not below the setpoint. Other than below the lower limit, the initial value can either

be over the upper limit or inside the bandwidth of the controller. Figure 2.16 shows how the controller is able to control the variable no matter what the initial condition of the output is.

2.6 ON OFF Modified Controller

A modification to the ON OFF controller strategy was performed on this thesis, in order to improve the behavior of the controller. This modification consisted on changing the rate with which the controller increases the output after reaching the upper limit for the first time. This can be effectively done with the use of the analogWrite() function in Arduino. The code for this ON OFF modified strategy can be found in Appendix A.2. The aim of this modification is to increase the time that the temperature is within the established limits.

The difference with the code of the previous section is that a new auxiliary variable has been defined, a, this auxiliary variable indicates the value inside analogWrite() used to turn on the actuator, therefore stating the duty cycle of the actuator. The effect on the answer of this variable is that the rate with which the output increases when the actuator is on decreases. Figure 2.17 shows the implementation of this strategy in the simple model developed in the previous section, it can be seen that the slope of the curve when the output is increasing decreases once the lower the upper limit is reached for the first time.

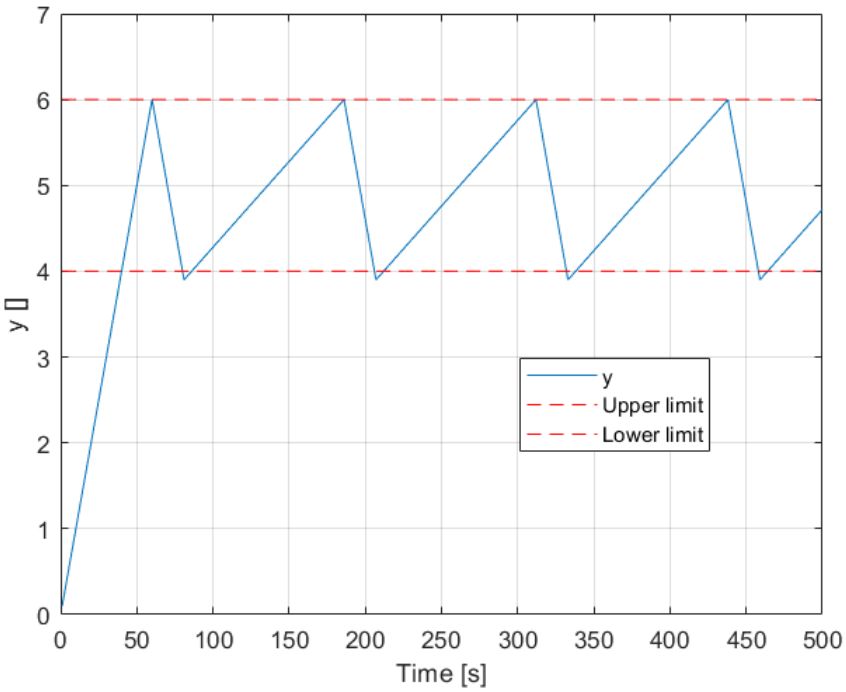


Figure 2.17: Example of ON OFF modified controller.

Since the code is very similar to the one in Appendix A.1, it will behave in the same way when the initial point is changed, assuring the safe operation of the controlled system.

2.7 PID Controller

In this section the procedure followed for the development and tuning of the PID will be exposed.

2.7.1 PID Code

Firstly, the PID code will be explained. This code can be found in Appendix A.3. The code works thanks to the use of the Arduino PID Library [20]. In order to use this library, it is necessary to include it at the beginning of the code. The PID Library allows the change of some parameters of the PID like the sample time or the output limits. The default values of these parameters are 0.1 s and [0, 255], respectively. In this project the output limit has been changed to [0, 1]. The use of this library is very simple, since to initialize the controller it is only necessary to define the input and output variables, the setpoint, the PID gains and the controller direction.

Once the PID has been initialized, the PID parameters can be changed in the setup loop. To run the function it is only necessary to write myPID.Compute(), where myPID would be the name given when the library is initialized.

In addition, a condition has been added before computing the new output from the PID to avoid computing mistaken input values. The condition imposed has been that the input - which is equivalent to the temperature from the sensor- must be greater than zero, this is because when there is a signal error, the sensor gives out a value equal to -127, which is clearly wrong and might lead to mistakes in the computation of the output.

Finally, the output is multiply by 255 to scale it to the analogWrite() function. This has been done like this because when the open loop methods have been apply a unit step and a step of value equal to 0.5 have been considered, as it will be seen in Section 2.7.2.1.

2.7.2 PID Tuning

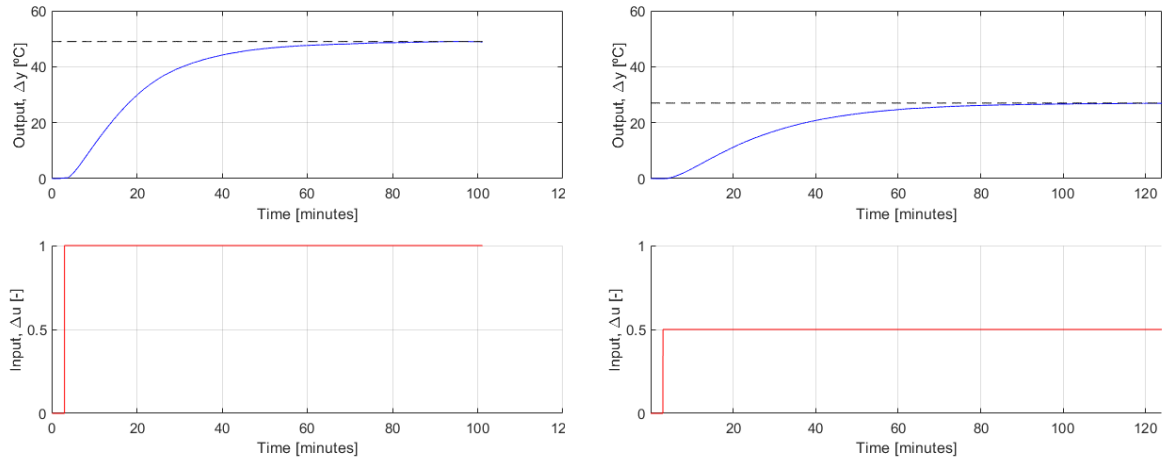
In this section, the procedure followed to perform the PID tuning will be explained. This process can be summarized in the following steps:

1. Run a step response on the real system.
2. Apply the open-loop tuning method to obtain an initial guess of the PID gains.
3. Obtain a model of the system with the use of the System Identification Toolbox.
4. Manually tune the PID controller with the help of the simulated model.
5. Finish tuning the controller with the real system.

2.7.2.1 Step Response

To study the step response of the real system it was not necessary to use the PID code, on the other hand a dedicated code was programmed, in which the only parameter to be changed are the step width and time, and the outputs are the temperature and the time. This code is shown in Appendix A.4.

With this code it was possible to analyze the step response to two step sizes; an unit step and a 0.5 step. In both cases the step time has been set equal to three minutes, and the response was run until the stationary value was reached. The responses to these steps are shown in figure 2.18.



(a) Step value equal to 1.

(b) Step value equal to 0.5.

Figure 2.18: Step responses.

2.7.2.2 Open Loop Tuning Methods

With the step responses obtained in the previous section, it is possible to obtain the values for the open-loop tuning, these are τ, T , and μ. Table 2.1 shows the values for both cases.

Table 2.1: Open-loop tuning parameter values.

Δu [-]	**Δy [°C]**	μ **[°C]**	τ **[s]**	**T** [s]
1	49	49	49.2	1,018.3
0.5	27	54	49.2	1,582.5

Introducing these values into the formulas of tables 1.2 and 1.3 it is possible to estimate the initial PID gains following both the Ziegler-Nichols and Cohen-Coon methods. The values obtained for both methods and both step widths are shown in table 2.2.

As it can be appreciated in the table 2.2 the values are very similar for both step values and both methods. According to [12], the Cohen-Coon method offers better results, therefore the values from the Cohen Coon method will be chosen as the initial guess of the PID gains. Moreover, the results with the step value equal to 0.5 will be the ones selected, this is due to the fact that the transfer function of obtained with the 0.5 step response better simulate the system, as it will be discussed in the next section.

Table 2.2: Open-loop tuning methods PID gains.

Ziegler-Nichols Method

	$\mathbf{K}_P$	$\mathbf{T}_I$	$\mathbf{T}_D$	$\mathbf{K}_I$	$\mathbf{K}_D$
Δu = 1	0.5068	98.4	24.6	0.0052	12.4684
Δu = 0.5	0.7148	98.4	24.6	0.0073	17.58

Cohen-Coon Method

	$\mathbf{K}_P$	$\mathbf{T}_I$	$\mathbf{T}_D$	$\mathbf{K}_I$	$\mathbf{K}_D$
Δu = 1	0.5683	118.6761	17.7351	0.0048	10.0782
Δu = 0.5	0.7988	119.5269	17.7903	0.0067	14.2117

2.7.2.3 System Identification

Once the open-loop response has been obtained, it is possible to obtain a transfer function of the system by using MATLAB's System Identification Toolbox, [23]. This tool estimates the transfer function that better adapts to the input and output of a system. However, it is necessary to indicate beforehand the type of transfer function that wants to be calculated. In this thesis, considering the type of system that is being heated up, two transfer functions were estimated; a first order with a delay, and a second order transfer functions.

Figure 2.19 shows the procedure to follow to import data and to calculate the transfer function, it also introduces two useful tools of the app to check the results. Once the transfer functions have been obtained these can be sent to the workspace by dragging them to the square with *To Workspace* written. More information on how to use the MATLAB System Identification Toolbox can be found in [24].

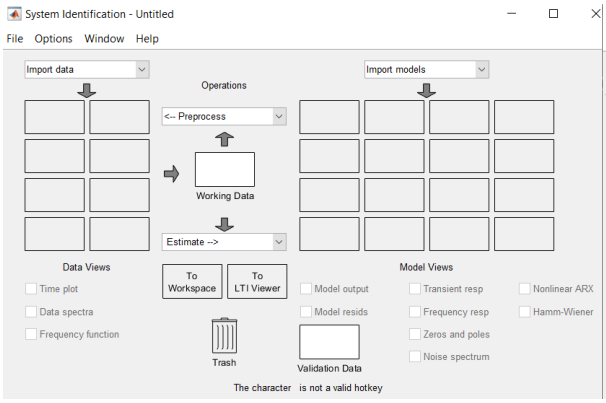
Once the data was imported, it was easy to calculate the transfer functions, for each set of data imported two transfer functions were calculated, a first order with delay and a second order transfer function. Figure 2.20 shows the result of this calculation.

It was then possible to simulate the model output by using the model output tool. In addition to showing the graph with the simulated output, the tool gives the % of how similar the two outputs are. Figure 2.21 shows the results of these simulations, where figures 2.21a and 2.21b are the results for Δu equal to 0.5 and 1, respectively.

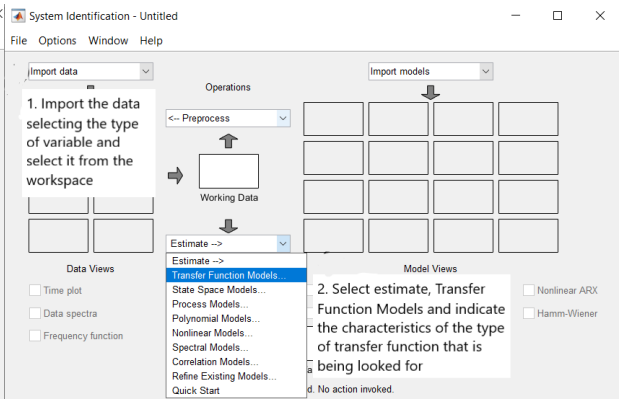
From figure 2.21 it can be concluded that a second order transfer function better describes the system. In addition, when the transfer function obtained with a Δu is used to simulate the other data, it can be seen how the best fit is the second order transfer function obtained with Δu equal to 0.5. Because of this, the transfer function used to model the system is the second order one obtained with Δu equal to 0.5. Table 2.3 shows the % of each transfer function for a better comparison.

Table 2.3: Transfer function model output best fits.

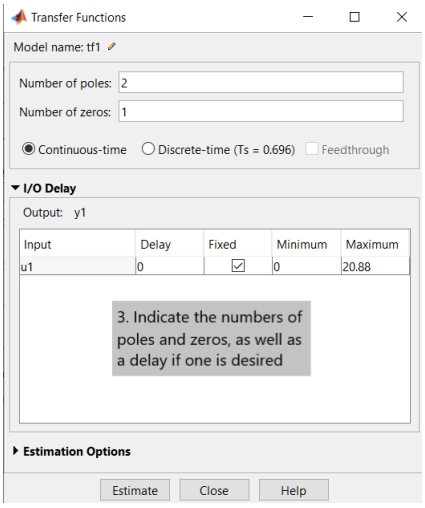
	Response to Δu = 0.5	**Response to Δu = 1**
1^{st} order with delay for Δu = 0.5 response	95.43%	95.69%
2^{nd} order for Δu = 0.5 response	99.61%	99.17%
1^{st} order with delay for Δu = 1 response	77.1%	78.81%
2^{nd} for Δu = 1 response	78.91%	80.83%



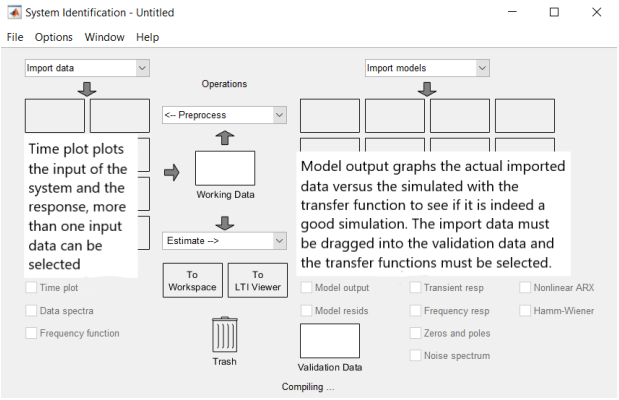
(a) System Identification Toolbox App.



(b) Step 1.



(c) Step 2.



(d) Useful tools.

Figure 2.19: Simulation Identification Toolboox tutorial.

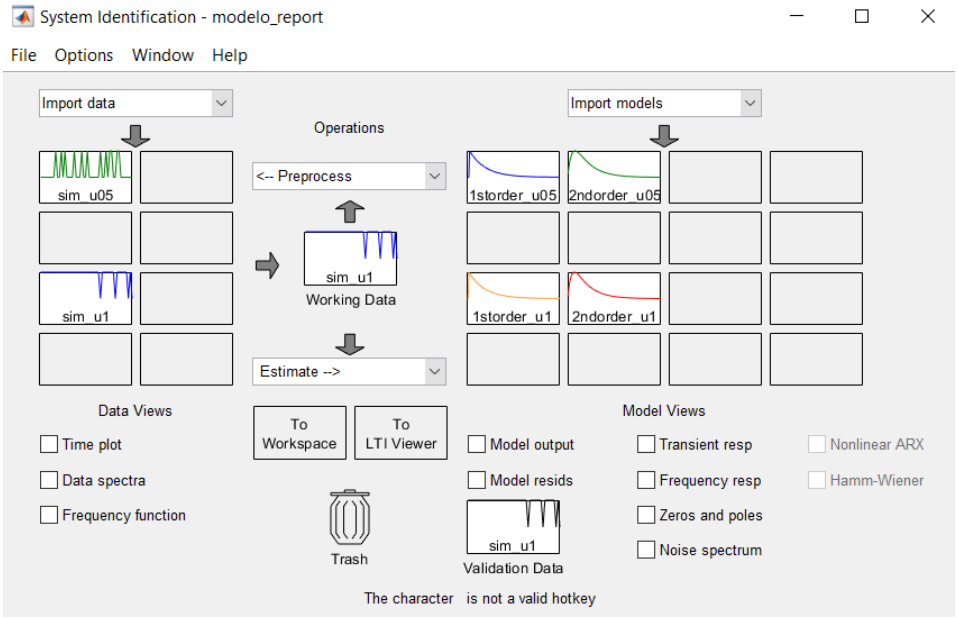
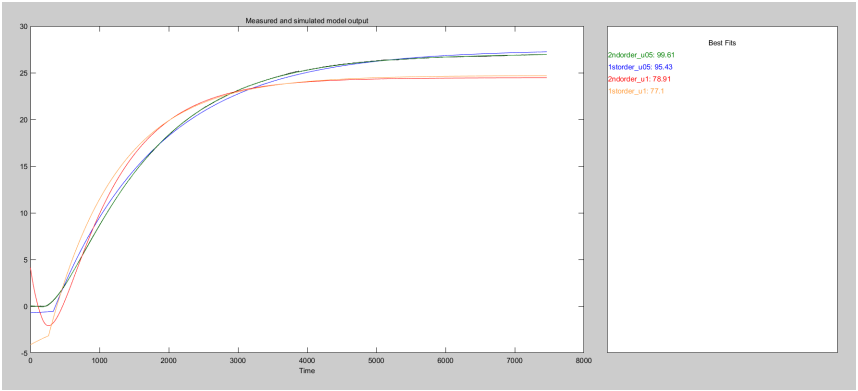
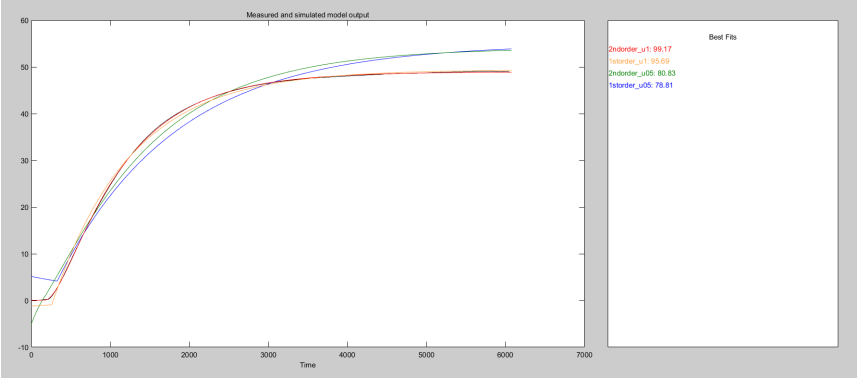


Figure 2.20: Simulation Identification Toolbox.



(a) Response to Δu = 0.5.



(b) Response to Δu = 1.

Figure 2.21: Transfer function output comparison.

Finally the transfer function used is:

$$\text{TF} = \frac{0.003312s + 0.0001205}{s^2 + 0.003586s + 2.228x10^{-06}}$$

2.7.2.4 PID Manual Tuning Based on the Simulation

A simple Simulink model was designed in order to implement the transfer function and being able to simulate the response of the plant, this model is shown in figure 2.22. The setpoint has been set to 40°C and the initial temperature, T_0 has been set equal to 21°C. The PID controller has the output limited in the range [0, 1]. The transfer function has been set equal to the one calculated in the previous section. Again, the PID output has been saturated because of the way in which the transfer function was calculated. In the System Identification Toolbox, the input that causes the system to change is suppose to be in the range [0, 1]. As stated before, the only inconvenient that this brings is that when the Arduino code is programmed, the output of the PID must be multiplied by 255 to scale it to the analogWrite() function.

Before starting the test it is necessary to define the characteristics of the response that wants to be obtained, for this project, these characteristics are; quick rise time, small overshoot, and small steady error. Afterwards, with the Simulink model, the transfer function and the initial PID gains, it was possible to initiate the manual PID tuning with the simulation. Firstly, the responses with Ziegler-Nichols and Cohen-Coon gains were simulated, this was done to see if they would provide similar re-

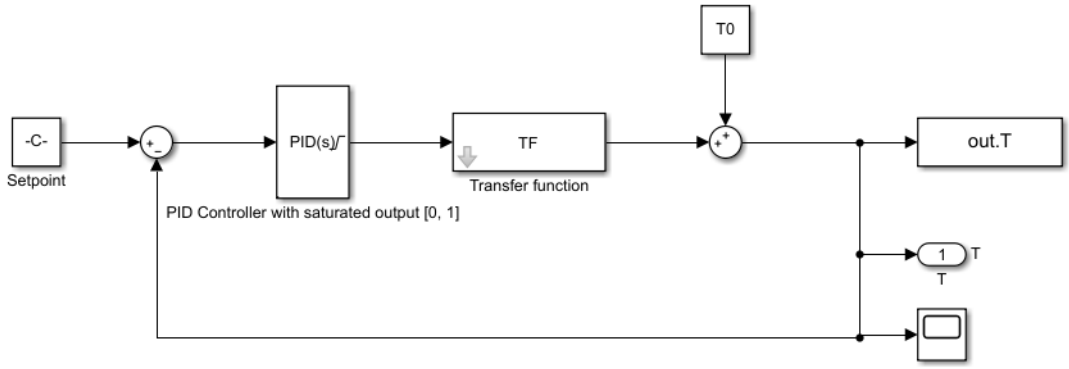


Figure 2.22: System Simulink model.

sponses, even though it had already been decided to chose the Cohen-Coon method gains, [12]. Figure 2.23 shows the result of this simulation.

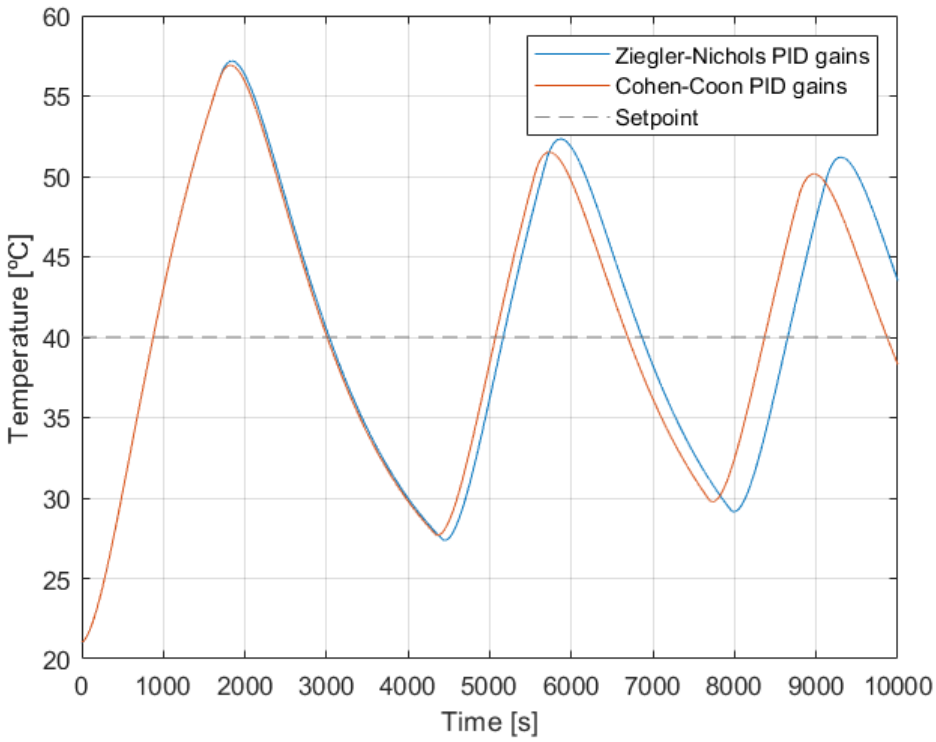


Figure 2.23: Simulation with Ziegler-Nichols and Cohen-Coon methods gains.

From figure 2.23 it was clear that the initial parameters needed some tuning, since the response was not desirable at all. To check if the simulation was really describing the system, the controller on the real system was set with the Cohen-Coon method parameters to study its response. Figure 2.24 shows the result of this test. As it can

be seen in the graph, the simulation overestimates the overshooting of the system, however it is accurate to predict the instability of the response. Since the PID gains gave out an unstable response, the simulated model was still considered valid.

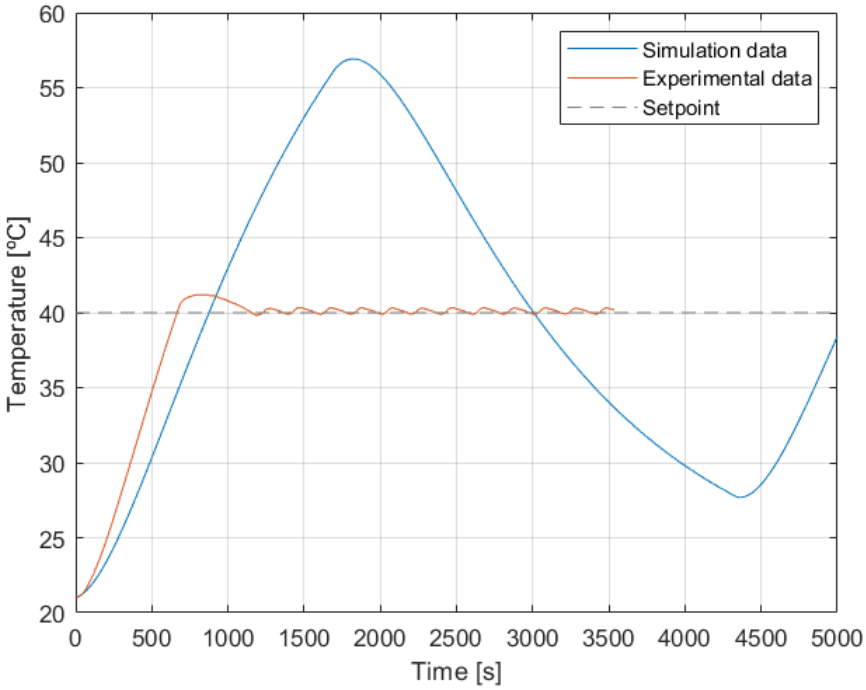


Figure 2.24: Simulated and real response with Cohen-Coon method gains.

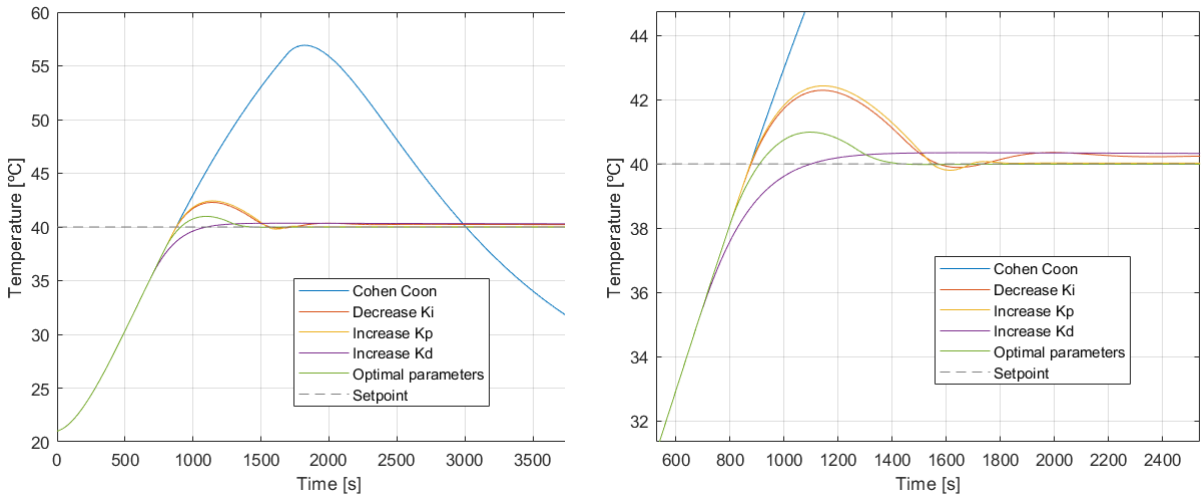
The instability of the response indicates that K_I is too big, therefore it was decreased. Afterwards, K_P and K_D were increased to decrease the steady state error,as well as the overshoot, respectively. These two changes were done separately as to check the effect on the new response. Finally a mix solution was adopted in order to meet all the requirements of the response. Figure 2.25 shows the result of this process.

Table 2.4 shows the exact gain values used for each simulation.

Table 2.4: Gain values in PID tuning process.

	Values compared to the Cohen-Coon method gains			**Values**		
	$\mathbf{K_P}$	$\mathbf{K_I}$	$\mathbf{K_D}$	$\mathbf{K_P}$	$\mathbf{K_I}$	$\mathbf{K_D}$
1. Cohen-Coon gains	$1\times K_{P_{CC}}$	$1\times K_{I_{CC}}$	$1\times K_{D_{CC}}$	0.7988	0.006683	14.21
2. Decrease $\mathbf{K_I}$	$1\times K_{P_{CC}}$	$K_{I_{CC}}/100$	$1\times K_{D_{CC}}$	0.7988	6.683×10^{-5}	14.21
3. Increase $\mathbf{K_P}$	$10\times K_{P_{CC}}$	$K_{I_{CC}}/100$	$1\times K_{D_{CC}}$	7.988	6.683×10^{-5}	14.21
4. Increase $\mathbf{K_D}$	$1\times K_{P_{CC}}$	$K_{I_{CC}}/100$	$10\times K_{D_{CC}}$	0.7988	6.683×10^{-5}	142.1
5. Optimal parameters	$5\times K_{P_{CC}}$	$K_{I_{CC}}/200$	$20\times K_{D_{CC}}$	3.994	3.342×10^{-5}	284.2

All this process describe above, is shown in the code in Appendix B.2. In addition, the last section of the code allows for a quick comparison between two set of gains in order to compare the responses and make the optimization in the response easier. Figure 2.26 shows an example of this comparison tool, to use it it is only necessary to change the gains in the last section of the code of Appendix B.2.



(a) (b) Zoom in.

Figure 2.25: PID tuning process.

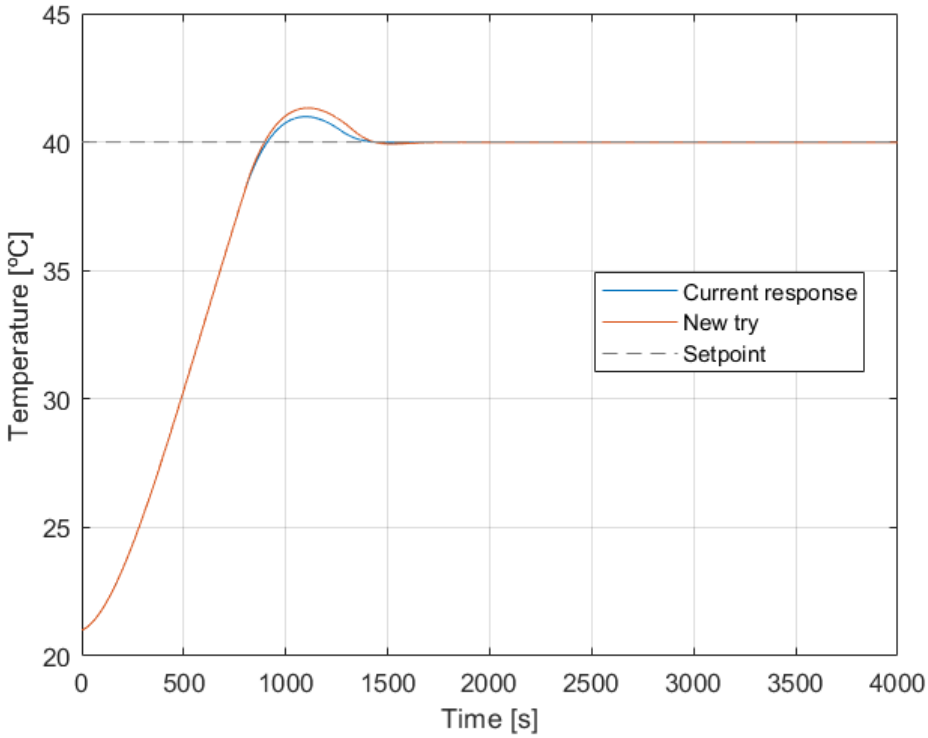


Figure 2.26: Simulated and real response with Cohen-Coon method gains.

2.7.2.5 Real System PID Tuning

In this section, it will be highlighted the fact that PID gains obtained in the simulation cannot be directly included into the Arduino code. This is due to the fact that the simulation time in Simulink is in seconds, whereas in Arduino it is in milliseconds. For this reason, it is necessary to multiply K_I by 1,000 and divide K_D by 1,000, when

converting the MATLAB values in Arduino.

2.7.3 Data Analysis

In this section, the programs used in order to save and process the data will be exposed in detail. Firstly, in order to load the data into a .csv file, once the code had been loaded into the Arduino board, the data streamer app from Excel was used, [21]. This app can be downloaded for free at [22], and it allows the Arduino to be connected to an Excel file that saves the Serial.print() commands from Arduino. For the Excel to work it is necessary to print all the variables that want to be saved, separating them with a coma and printing a final blank new line, as in the following example:

```
1     // Data processing
2     Serial.print(variable_1);
3     Serial.print(",");
4     Serial.print(variable_2);
5     Serial.print(",");
6     Serial.print(variable_3);
7     Serial.print(",");
8     Serial.print(variable_4);
9     Serial.print(",");
10    Serial.print(variable_5);
11    Serial.print(",");
12    Serial.println();
```

Figure 2.27 shows the option from the Data Streamer tab. In addition to the options described in figure 2.27, if the Record Data option is selected, the live data coming from the Arduino board will be saved into a .csv file, which is the objective of using this tool.

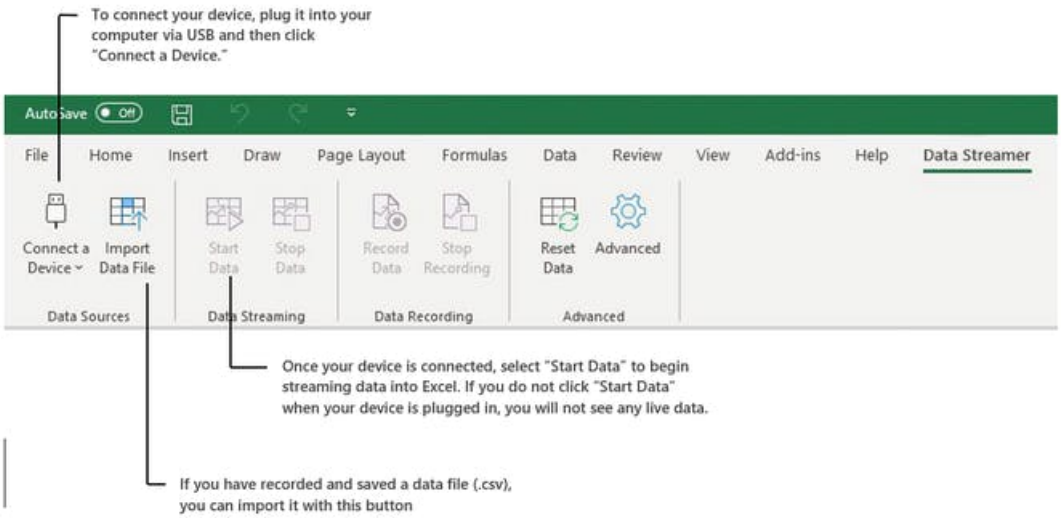


Figure 2.27: Excel Data Streamer Options, [21].

Even if it has not used in this project, this tool also allows the user to send commands to the Arduino board while it is operating, this would be useful to change if for example the setpoint needed to be changed.

Since the size of the .csv file was too big for Excel to handle it efficiently, it was decided to import the data into MATLAB using the uiimport() function, an example code for importing data from .csv files can be seen in Appendix B.1. Since the data from the sampling had points that were not correct it was necessary to do some filtering to correct the discontinuities and NaN values. This value consisted of ignoring the sample if the time or the value presented a discontinuity or the value was NaN. The filtering allowed for a smooth plot of the data, this filtering was possible due to the fact that the temperature values change smoothly, so a big change in their value necessarily has to be a mistake from the sensor. It was done as shown below, the first column corresponds to the temperature data and the second column corresponds to the time.

```
result = uiimport('data_file.csv'); % opens the import tool of matlab, ...
    select the file
% filter
for ii = 2:length(result.data(1:end,1))
    % filters the time discontinuities
    if abs(result.data(ii,1)-result.data(ii-1,1)) >  3000
        result.data(ii,2) = result.data(ii-1,2);
        result.data(ii,1) = result.data(ii-1,1);
    end
    % filters the temperature values discontinuities
    if abs(result.data(ii,2)-result.data(ii-1,2)) > 1
        result.data(ii,2) = result.data(ii-1,2);
        result.data(ii,1) = result.data(ii-1,1);
    end
    % filters the NaN values
    if isnan(result.data(ii,2))
        result.data(ii,2) = result.data(ii-1,2);
    end
end
```

Chapter 3

Results and conclusion

3.1 ON OFF Controller

Once the ON OFF controller was implemented in the Arduino board and the system was connected, it was possible to test it. Figure 3.1 shows the result of this test. In this test, the setpoint was set to 40°C and the margin was set to 1°C, which means that the upper limit is equal to 41°C and the lower limit is equal to 39°C.

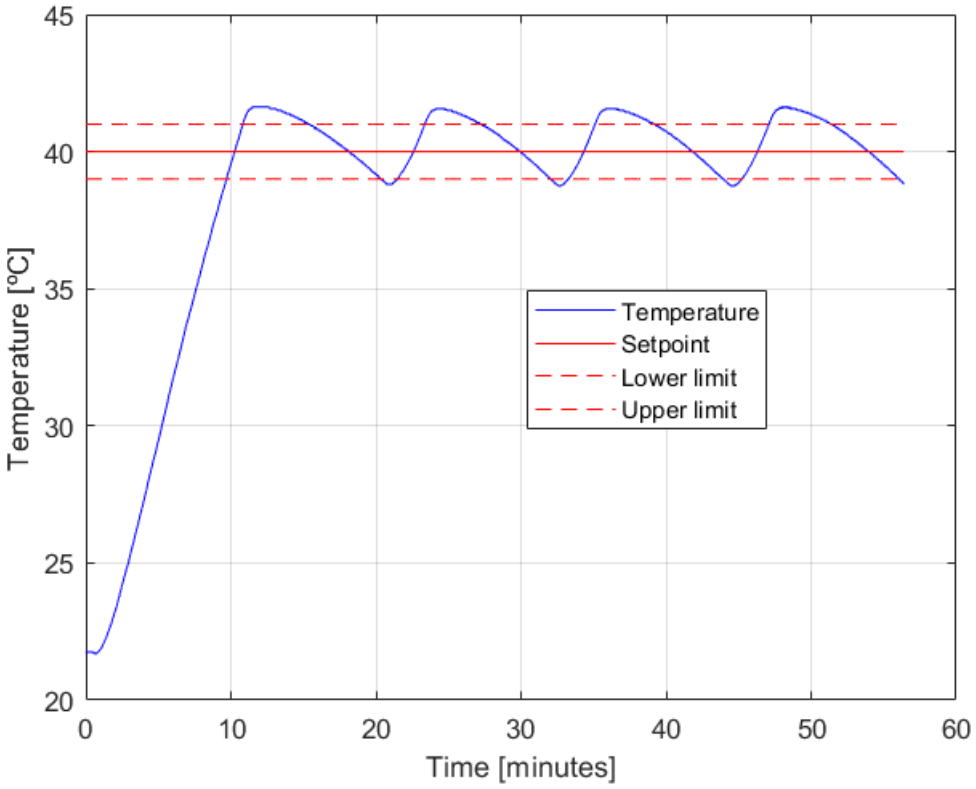


Figure 3.1: Temperature evolution with the ON OFF controller.

It must be highlighted that the reason why the temperature goes above the upper limit

is because the electric resistor has thermal inertia, this means that even if the resistor is turned off when the thermistor measures a temperature equal to 41°C, the resistor will still be hot and will continue to warm the system, until it reaches the thermal equilibrium. On the other hand, the system goes below the lower limit because even though the resistor is turned on when the sensor measures a temperature of 39°C, it takes it some time to get hot and start warming the water.

Moreover, figure 3.2 shows a graph with the transistor state, which indicates if the resistor is in ON or OFF state, and the temperature evolution. This graph allows to check that the results are correct, as well as to provide a better understanding of the ON OFF controller strategy.

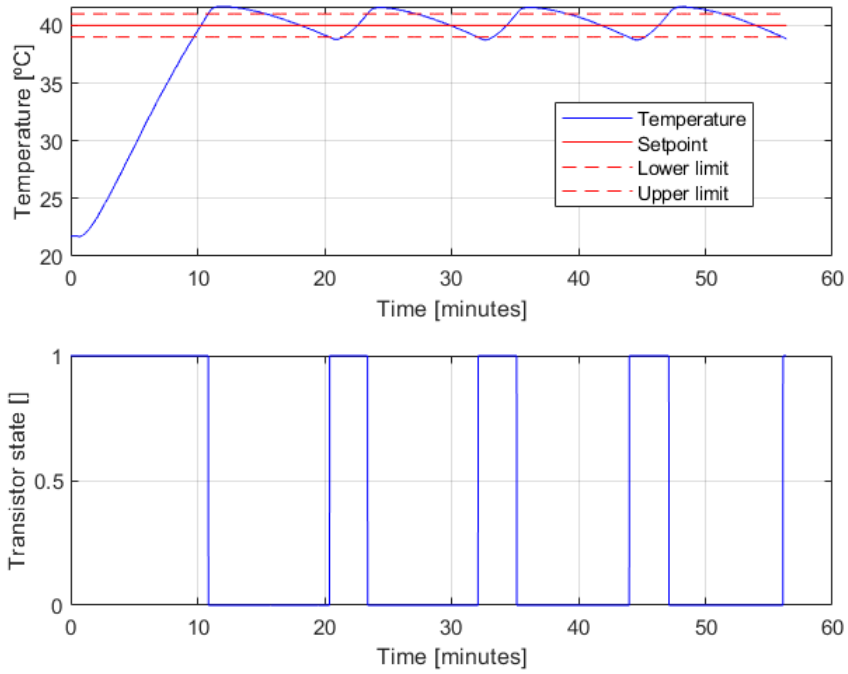


Figure 3.2: Transistor state and temperature evolution with the ON OFF controller.

Overall the controller works fine, the limit is passed in the upper limit, reaching a temperature equal to 41.63°C and below the lower limit, reaching a temperature equal to 38.81°C.

3.2 ON OFF Modified Controller

With the aim of improving the behavior of the ON OFF controller designed, a change in the control strategy was done, as explained in Section 2.6. The aim of this change was to increase the time that the temperature was inside of the upper and lower limits. In this control strategy, other than the margin and the setpoint it is necessary to indicate the duty cycle that wants to be applied when the resistor is ON. The control system has been tested with the following duty cycle values; 0.8, 0.5 ,and 0.3. Figures 3.3, 3.4, and 3.5 show the results of these tests, respectively.

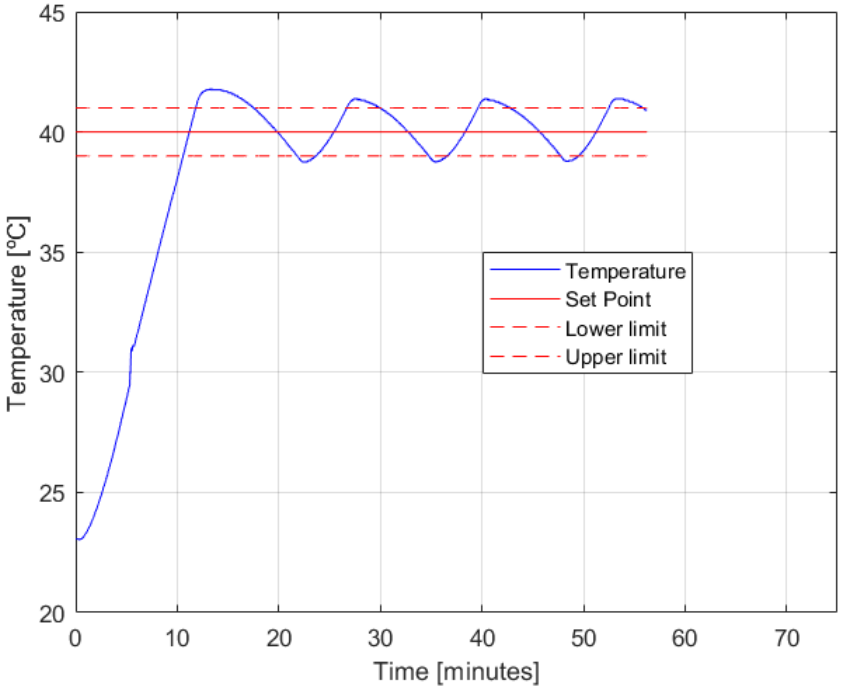


Figure 3.3: Temperature evolution with the ON OFF modified controller with duty cycle equal to 0.8.

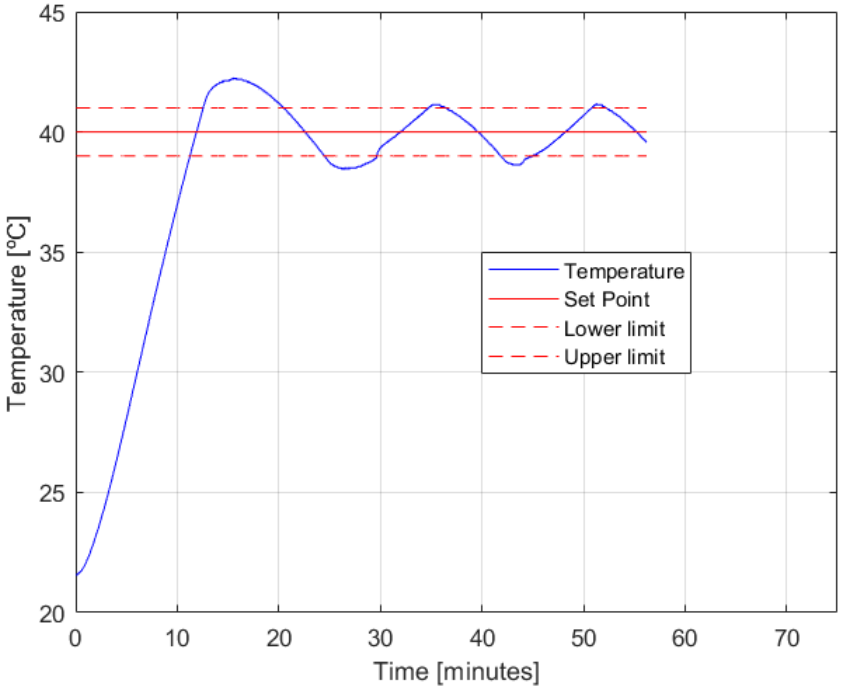


Figure 3.4: Temperature evolution with the ON OFF modified controller with duty cycle equal to 0.5.

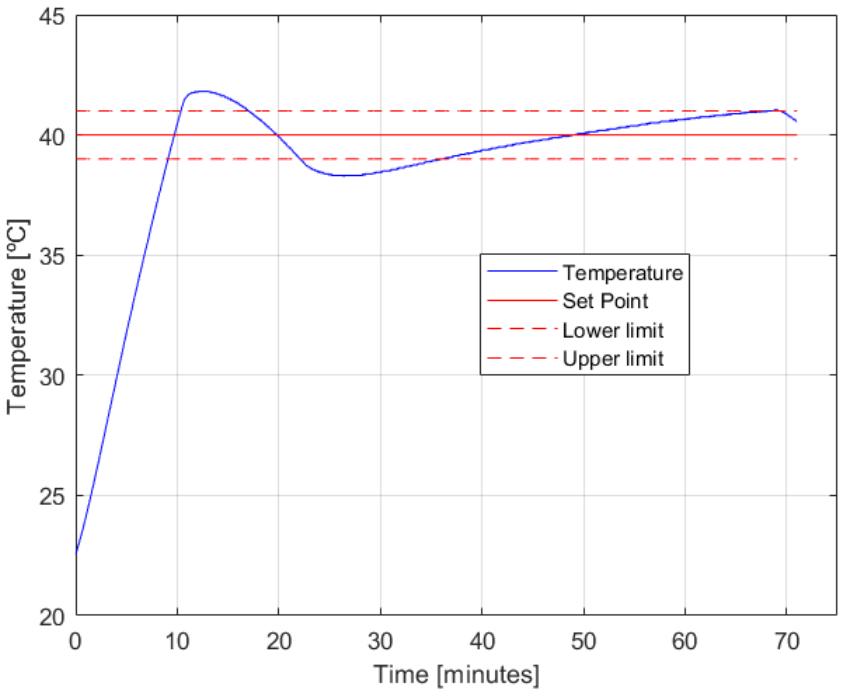


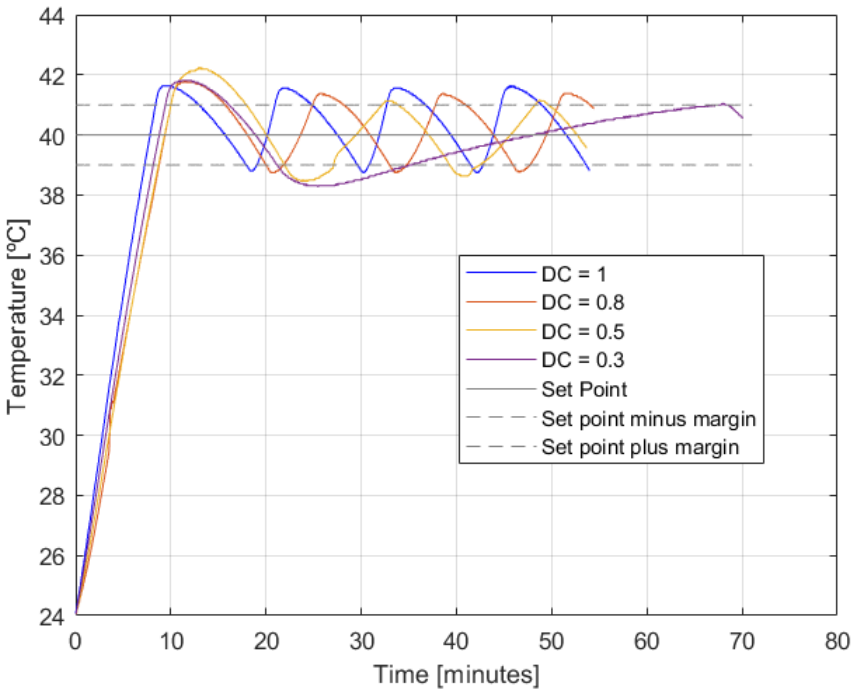
Figure 3.5: Temperature evolution with the ON OFF modified controller with duty cycle equal to 0.3.

By comparing figures 3.3, 3.4, and 3.5 the following conclusions can be derived:

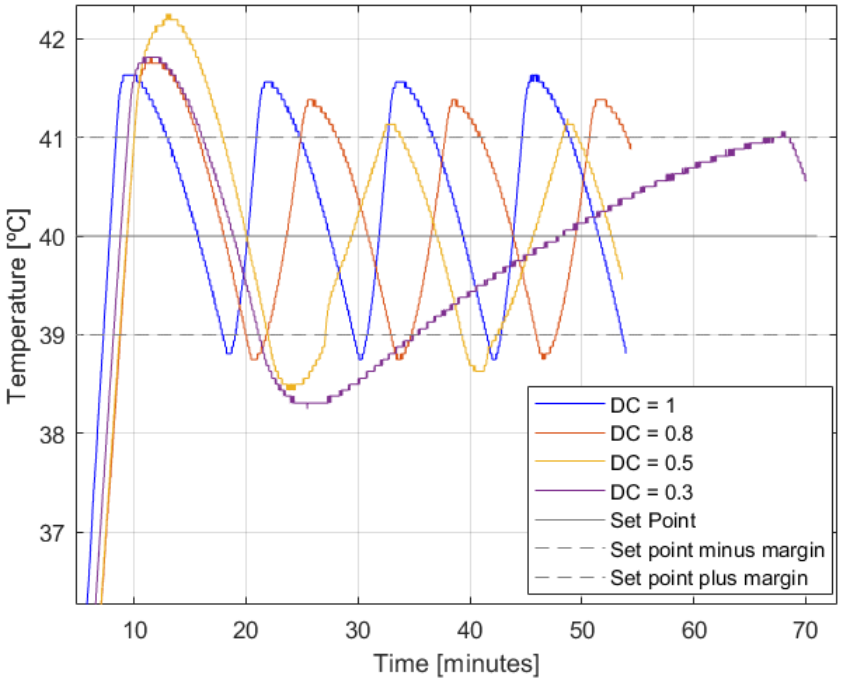
- As the duty cycle decreases, the time in which the temperature is over the upper limit decreases too, whereas the time in which the temperature is below the lower limit increases.
- The time that it takes the system to go from the lower limit from the upper limit increases when the duty cycle increases.
- The increment of the time that it takes to heat up from the implies that the ON OFF frequency decreases, which is good to avoid chattering.

These conclusions can be better observed in figure 3.6, where the temperature evolution for the different duty cycles is shown. The temperature evolution of the non modified ON OFF controller has also been included, and it corresponds to the duty cycle equal to 1. As it can be seen the difference between the responses comes after the upper limit is reached for the first time.

If this strategy was to be implemented the duty cycle chosen would be 0.5. This is due to the fact that the aim of this modification is to increase the time that the temperature is within the limits, and the temperature evolution of the duty cycle equal to 0.5 represents the compromise solution between the reduction in the time spent over the upper limit and the increment of the time spent below the lower limit. In addition, a smaller duty cycle means a very slow response, which is not desirable either, because the resistor can hardly change the state of the plant, as seen in figure 3.5.



(a)



(b) Zoom in.

Figure 3.6: Comparison of the temperature evolution with the ON OFF controller for different duty cycles.

3.3 PID Controller

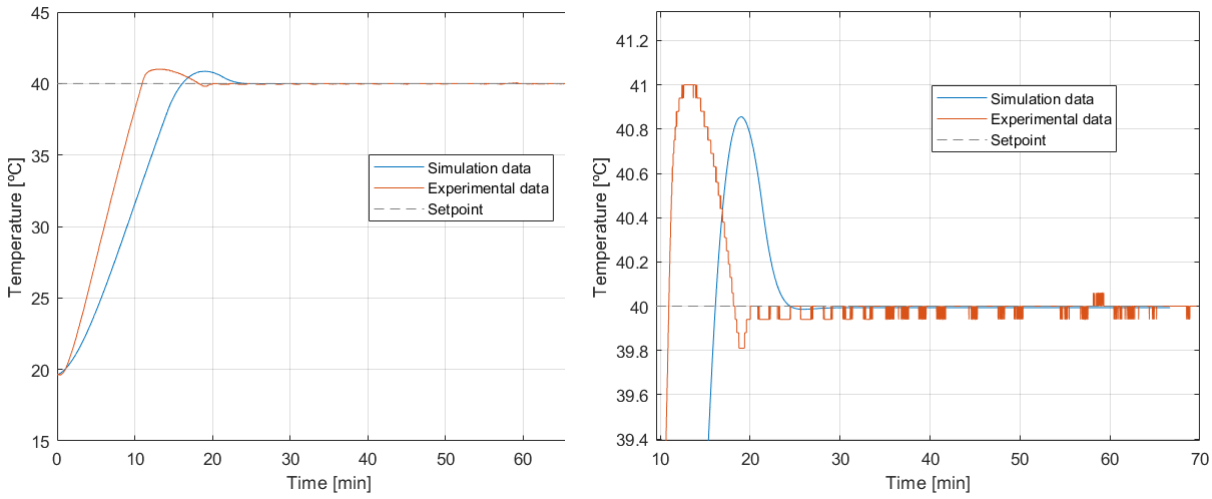
From the simulation it was possible to derive an optimal set of PID parameters for the controller. These gains are shown again in table 3.1 and they show the numerical values for both, MATLAB and Arduino.

Table 3.1: Gain values in PID tuning process.

	$\mathbf{K}_P$	$\mathbf{K}_I$	$\mathbf{K}_D$
Arduino	3.994	0.03342	0.2842
MATLAB	3.994	3.342×10^{-5}	284.2

When these values where included in the code shown in Appendix A.3. The PID algorithm was then used to heat up the water to 40°C, the result of this test is shown in figure 3.7. In addition, the simulated response of the system is also shown to prove that the system works correctly. The response is characterized by an overshoot of 1°C and a steady state error equal to almost zero, with the steady final value equal to 40±0.06°C. The difference in the simulated response and the real one can be explained by the fact that the conditions of the room in which the test was performed were not the same, this applies mainly to the temperature of the room, which influences the heat exchange rate with the water. Another factor is the initial temperature of the water which depends on the conditions of the room of that day.

If the parameters would not have obtained a solution meeting all the requirements it would have been necessary to tune the final solution manually. This can be done knowing the effect of increasing or decreasing each gain, as shown in table 1.1. However, since the solution provides a response that fills all the requirements no longer tuning was needed.

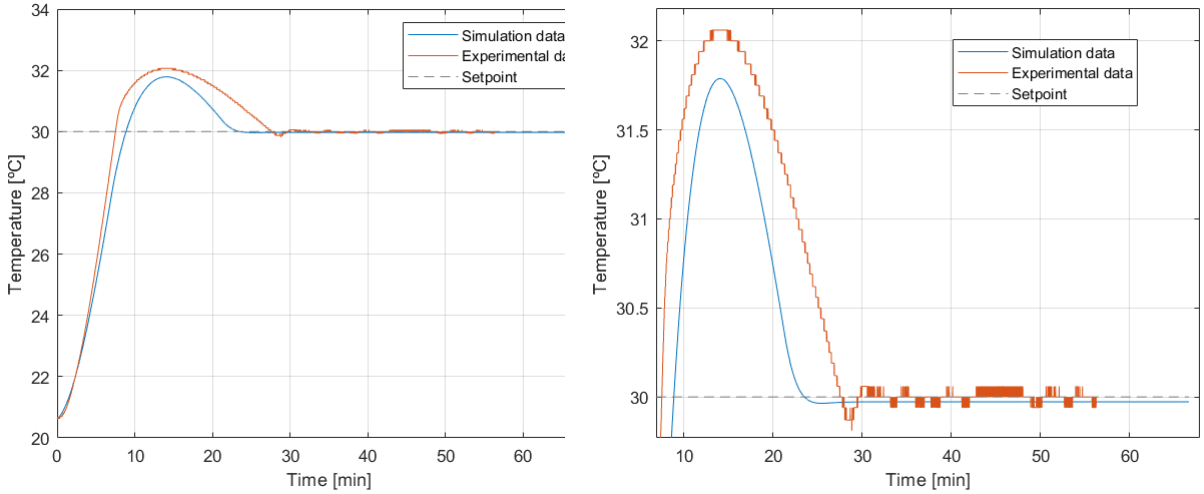


(a) (b) Zoom in.

Figure 3.7: Temperature evolution with the PID controller.

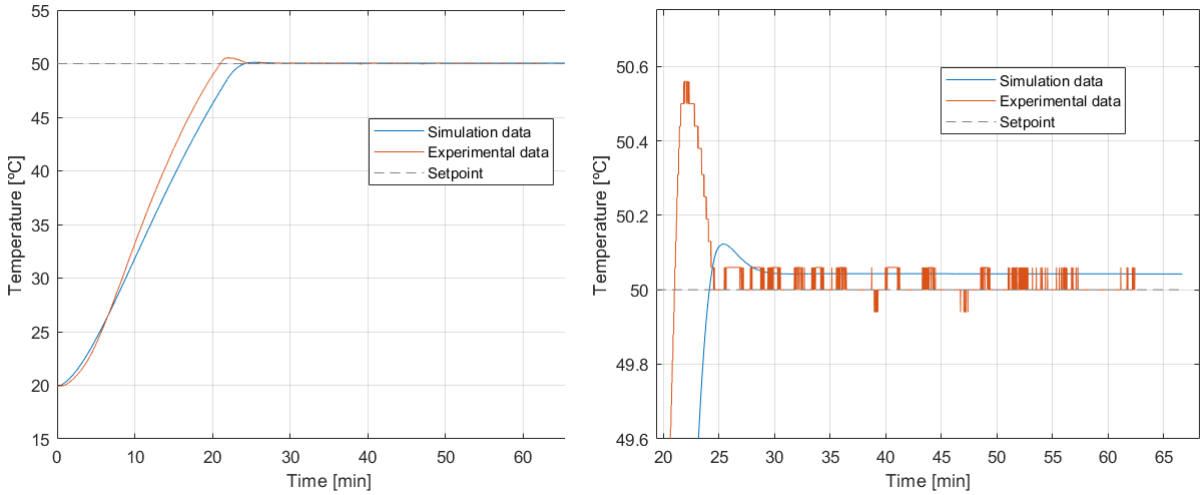
3.3.1 Robustness

A more in detail analysis of the obtained parameters was performed by studying the response when the setpoint was change. Starting from the room temperature the system was set to 30°C and 50°C. The results of this analysis is shown in figures 3.8 and 3.9. In both cases the simulated and real responses have been plotted together, to further prove that the simulated model works. Both responses reach the setpoint with a tolerance of ±0.06°C. Furthermore, the higher the setpoint the lower the overshoot, this is due to the fact that the higher the setpoint the closer it is to the maximum temperature that can be reached with the actuator.



(a) (b) Zoom in.

Figure 3.8: Temperature evolution with the PID controller with setpoint equal to 30°C.



(a) (b) Zoom in.

Figure 3.9: Temperature evolution with the PID controller with setpoint equal to 50°C.

3.4 Comparison

Since the aim of this project was to develop a series of controllers for the temperature control, the performance of these was compared. Figure 3.10 shows the result of this comparison. From this figure it is possible to say that until the desired temperature is reached for the first time, the the three responses are very similar (this is due to the fact that the resistor is just turned on to the maximum level), although in the PID controller it can be seen how the overshoot is minor, this is due to the fact that the ON OFF controllers keep on warming up the water until the upper limit has been reached. Once this point has been reached the PID response just decreases until it stabilizes around a temperature value equal to 40±0.06°C. On the other hand the ON OFF controllers at this point start the hysteresis cycle in which the resistor switches from ON and OFF depending on the temperature measured. The hysteresis period of the modified controller is longer, this can be seen on the slope of the temperature when it increases - it is smaller than the ON OFF one-. Again, the best controller when it comes to maintaining the temperature in the setpoint is the PID, however, as it has seen its development requires a more complex process.

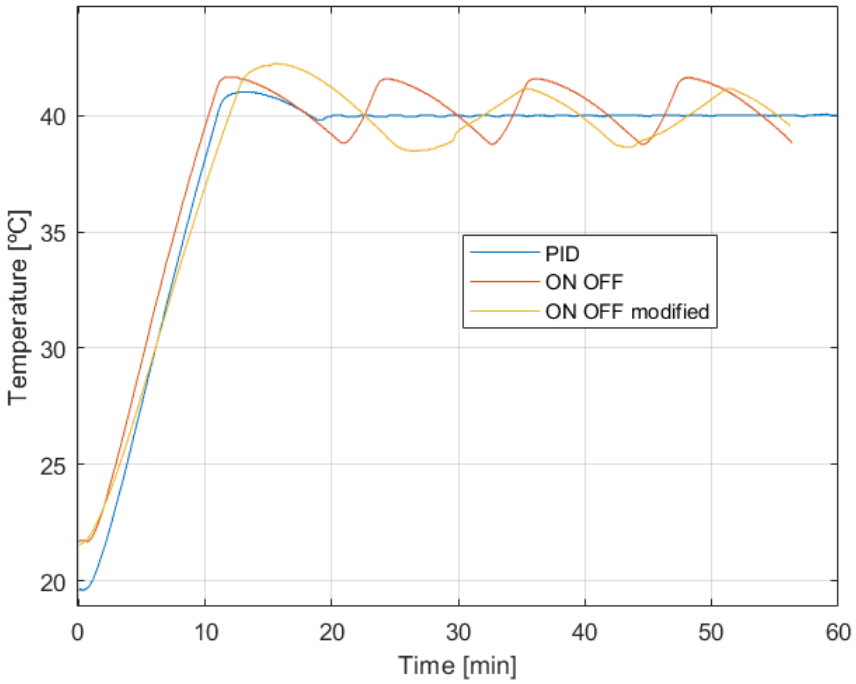


Figure 3.10: Temperature evolution comparison for the three controllers developed.

Table 3.2 shows a comparison in the characteristics of the controllers. As it can be seen, the ON OFF controllers accuracy is lower, whereas the complexity in the design is also lower. On the other hand, the difference between the two ON OFF controllers has to do with the switching frequency of resistor, which is smaller for the modified version, this means that if the temperature was controlled for an hour, the modified ON OFF controller switches less time, this might make the useful life of the actuator longer.

Table 3.2: Controllers comparison.

Controller strategy	Accuracy	Complexity	Switching frequency
ON OFF	Low	Low	Medium
Modified ON OFF	Low	Medium	Low
PID	High	High	-

3.5 Conclusion

In this project, the development of a series of controllers is described, more in particular these controllers are; an ON OFF, a modified version of the same, and a PID controller. This development included the programming of the Arduino board, the design of the system, the testing and optimization.

The results showed that the PID controller managed to set the temperature to the desired value better than the ON OFF controller. However, this was the expected result. On the other hand, this bigger accuracy came with the disadvantage of being harder to implement. This difficulty is reflected in the development Chapter, whereas for the implementation of the ON OFF controller it was a matter of applying the algorithm, the PID presented the difficulty of tuning the gains.

This bigger complexity can only be justified when the controlled variable needs a certain precision. This complexity lies on the PID tuning, not in the controller itself, in fact, as mention before, the PID controller is the most used controller, due to the ratio between accuracy and complexity.

When the two ON OFF controllers are compared, it can be seen how the main difference between them lies on the time that it takes the controller to go from the lower limit to the upper one. This is directly related to the switching frequency of the actuator, overall, for the same amount of time working, the ON OFF modified controller switches less times, which in the long term might mean a longer life.

This project only describes the design of the prototype of the temperature control system that could be applied for the control of the oil temperature of the heating system for a new test rig for mechanical transmission components. In order to implement the prototype into the real system some considerations have to be taken. If it is assumed that the thermistor subsystem is not changed and the same probe is used to measure the temperature, then the implementation would be facilitated a lot, since the same Arduino code could be used. What is more, the same probe could also be used since the maximum operating temperature is equal to 125 °C and the desired oil temperature in the real system would be equal to 120 °C. However, some tests should be performed to assure that the thermal inertia does not make the oil temperature to rise above this maximum temperature threshold. As for the resistor subsystem, the actuator should be change in order to deliver the adequate heat to reach the desired temperature. However, this does not necessarily imply that a new power electronic circuit should be designed, firstly, it should be checked whether the IRF520n MOSFET can withstand the current and voltage required by the new resistor.

When talking about the different controllers, for the ON OFF controllers, the code could be implemented just as it is, since the logic in the algorithm is still the same, the only changes that should be made are the setpoint, the margin.In the modified

ON OFF controller the duty cycle should also be modified, this is the rate with which the temperature increases after reaching the upper limit for the first time. As for the PID controller, here again the setpoint should be changed, as well as the PID gains. In order to tune the PID controller, it is recommended to follow the procedure described in Chapter 2. To obtain the step response from the system it is possible to use the code in Appendix A.4.

In conclusion, it can be stated that the three controllers fulfill their requirements, and their differences lie on accuracy, switching frequency and complexity. Depending on the requirements of the controlled variable one controller should be chosen over the others.

Bibliography

[1] DABBENE, Fabrizio; Dynamics and Control of Aerospace Vehicles notes. Politecnico di Torino, 2019.

[2] KUO, Benjamin C.; GOLNARAGHI, Farid. Automatic control systems. Englewood cliffs, 2003.

[3] DOUGLAS, Bryan; Understanding PID Control series. MathWorks®. Videos and webinar series, Understanding PID Control

[4] HERNÁNDEZ, José Aldemar Muñoz; HERNÁNDEZ, Luis Alfonso Muñoz; BARRERO, Carlos Antonio Rivera. Control automático I: estrategias de control clásico. Universidad del Tolima, Facultad de Ingeniería Agronómica, 2014.

[5] COHEN, GHp; COON GA. Theoretical consideration of retarded control. Trans. Asme, 1953, vol. 75, p. 827-834.

[6] ZIEGLER, John G., et al. Optimum settings for automatic controllers. trans. ASME, 1942, vol. 64, no 11.

[7] WOOLF, Peter. PID Tuning via Classical Methods. Libretexts, 2021. Book: Chemical Process Dynamics and Controls (Woolf)

[8] BANZI, Massimo; SHILOH, Michael. Getting started with Arduino: the open source electronics prototyping platform. Maker Media, Inc., 2014.

[9] EVANS, Brian W. Arduino programming notebook. Brian Evans, 2007.

[10] Processing Foundation website

[11] Arduino Foundations PWM website

[12] JOSEPH, E. A.; OLAIYA, O. O. Cohen-coon PID Tuning Method; A Better Option to Ziegler Nichols-PID Tuning Method. ENginerring Research, 2017, vol. 2, no 11, p. 141-145.

[13] Autodesk Tinkercad website

[14] BD237 Datasheet, Motorola Bipolar Power Transistor Device Data. Link to pdf.

[15] IRF520N Mosfet Datasheet, International Rectifier HEXFET Power MOSFET Device Data. Link to pdf.

[16] Dallas DS18B20 Waterproof Temperature Sensor Cable, Quick Tech Datasheet Link to pdf.

[17] How to Use Temperature Sensor Tutorial. Link to website

[18] Fritzing website

[19] Arduino Library Guide

[20] Arduino PID Library website

[21] Stream Data from Arduino into Excel. Arduino Project Hub website

[22] Excel Data Streamer website

[23] System Identification Toolbox website, MathWorks®

[24] Introduction to System Identification Toolbox, MathWorks®

Appendix A

Arduino Codes

A.1 ON OFF Controller Code

```
#include <DallasTemperature.h>
#include <OneWire.h>

// Transistor wire is plugged into digital pin 10 on the Arduino
const int TRANSISTOR = 10;

// Data wire is plugged into digital pin 2 on the Arduino
const int ONE_WIRE_BUS = 2;

float Temperature;
int trans_state;
int flag;

/*********************************************************/
/*************** Parameters to be changed ****************/
/*********************************************************/
/* Setpoint is the temperature that wants to be achieved in degree Celsius */
double Setpoint = 40; // degree Celsius

/* Set the margin above and below that we want to achieve to turn on and off ...
    the control once the temperature has been reached*/
const int margin = 1; // degree Celsius
/*********************************************************/

// Setup a oneWire instance to communicate with any OneWire device
OneWire oneWire(ONE_WIRE_BUS);

// Pass oneWire reference to DallasTemperature library
DallasTemperature sensors(&oneWire);

void setup()
{
    Serial.begin(9600);
    sensors.begin();
    pinMode(TRANSISTOR,OUTPUT);
    /* flag is the variable that indicates whether the temperature needs to ...
        be increased or decreased, it is supposed that the temperature needs ...
        to be increased at the beginning, even if it was not the case the ...
        code is prepared to not activate the resistor if the temperature is ...
```

```
          already above the setpoint plus the margin */
36      flag = 1;
37  }
38
39  void loop()
40  {
41      unsigned long currentMillis = millis();
42      sensors.requestTemperatures();
43      Temperature = sensors.getTempCByIndex(0);
44
45      /* ON OFF strategy, with these three if conditions it is possible to ...
           keep the value of the temperature in between the setpoint +- margin ...
           values. Other than the temperature the value of the variable flag is ...
           looked at, it expresses whether the temperature should increase or ...
           decrease.*/
46
47      if ((Temperature < Setpoint + margin) && (flag == 1)) {
48          digitalWrite(TRANSISTOR,HIGH);
49      }
50      else if (Temperature > Setpoint + margin){
51          digitalWrite(TRANSISTOR,LOW);
52          flag = -1;
53      }
54
55      else if ((Temperature < Setpoint - margin) && (flag == -1)) {
56          digitalWrite(TRANSISTOR,HIGH);
57          flag = 1;
58      }
59
60      // Data processing
61      Serial.print(currentMillis);
62      Serial.print(",");
63      Serial.print(Setpoint);
64      Serial.print(",");
65      Serial.print(Temperature);
66      Serial.print(",");
67      Serial.print(trans_state);
68      Serial.print(",");
69      Serial.print(flag);
70      Serial.print(",");
71      Serial.println();
72  }
```

A.2 ON OFF Modified Controller Code

```
1       #include <DallasTemperature.h>
2       #include <OneWire.h>
3
4       // Transistor wire is plugged into digital pin 10 on the Arduino
5       const int TRANSISTOR = 10;
6
7       // Data wire is plugged into digital pin 2 on the Arduino
8       const int ONE_WIRE_BUS = 2;
9
10      float Temperature;
11      int trans_state;
12      int flag;
```

```
13      int value_trans;
14      /*************************************************************/
15      /************** Parameters to be changed *********************/
16      /*************************************************************/
17      /* Setpoint is the temperature that wants to be achieved in degree ...
           Celsius */
18      double Setpoint = 40; // degree Celsius
19
20      /* a is the duty cycle that wants to be obtained after the temperature ...
           overpasses  the Setpoint plus margin for the first time*/
21      float a = 0.5;// Equivalent to a 0.5 duty cycle
22
23      /* Set the margin above and below that we want to achieve to turn on and ...
           off the control once the temperature has been reached*/
24      const int margin = 1; // degree Celsius
25      /*************************************************************/
26
27      // Setup a oneWire instance to communicate with any OneWire device
28      OneWire oneWire(ONE_WIRE_BUS);
29
30      // Pass oneWire reference to DallasTemperature library
31      DallasTemperature sensors(&oneWire);
32
33      void setup()
34      {
35          Serial.begin(9600);
36          sensors.begin();
37          pinMode(TRANSISTOR,OUTPUT);
38          //initialize the variables we're linked to
39          /* variable that indicates whether the temperature needs to be ...
               increased or decreased, it is supposed that the temperature needs ...
               to be increased at the beginning, even if it was not the case the ...
               code is prepared to not activate the resistor if the temperature ...
               is already above the setpoint plus the margin */
40          flag = 1;
41
42          // initiate the value that is introduced into the analogWrite function
43          value_trans = 255;
44      }
45
46      void loop()
47      {
48          unsigned long currentMillis = millis(); // Measure the current time
49          sensors.requestTemperatures();
50          Temperature = sensors.getTempCByIndex(0); // Measured temperature
51
52          /* ON OFF strategy, with these three if conditions it is possible to ...
               keep the value of the temperature in between the setpoint +- ...
               margin values. Other than the temperature the value of the ...
               variable flag is looked at, it expresses whether the temperature ...
               should increase or decrease. Once the setpoint + margin value has ...
               been achieved, the value_trans is updated to the new duty cycle ...
               selected*/
53          if ((Temperature < Setpoint + margin) && (flag == 1)) {
54              analogWrite(TRANSISTOR, value_trans);
55          }
56          else if (Temperature > Setpoint + margin){
57              analogWrite(TRANSISTOR, 0);
58              flag = -1;
59              value_trans = a*255; // change the value introduced into the ...
                   analogWrite()
```

```
60          }
61
62          else if ((Temperature < Setpoint - margin) && (flag == -1)) {
63              analogWrite(TRANSISTOR, value_trans);
64              flag = 1;
65          }
66
67          // Data processing
68          Serial.print(currentMillis);
69          Serial.print(",");
70          Serial.print(Setpoint);
71          Serial.print(",");
72          Serial.print(Temperature);
73          Serial.print(",");
74          Serial.print(trans_state);
75          Serial.print(",");
76          Serial.print(flag);
77          Serial.print(",");
78          Serial.println();
79
80      }
```

A.3 PID Controller Code

```
1       #include <DallasTemperature.h>
2       #include <OneWire.h>
3       #include <PID_v1.h>
4
5       // Transistor wire is plugged into digital pin 10 on the Arduino
6       const int TRANSISTOR = 10;
7
8       // Data wire is plugged into digital pin 2 on the Arduino
9       const int ONE_WIRE_BUS = 2;
10
11      //Define PID variables
12      double Input, Output;
13
14      // Auxiliary variable used to scale the output for the analogWrite() ...
            function
15      float y;
16
17
18      /********************************************************/
19      /*************** Parameters to be changed ***************/
20      /********************************************************/
21      /* PID gains */
22      float Kp=0.7988 , Ki=6.7, Kd=0.0142117;
23      /* Setpoint is the temperature that wants to be achieved in degree ...
            Celsius */
24      double Setpoint = 40; // degree Celsius
25      /********************************************************/
26
27      // Setup a oneWire instance to communicate with any OneWire device
28      OneWire oneWire(ONE_WIRE_BUS);
29
30      // Pass oneWire reference to DallasTemperature library
31      DallasTemperature sensors(&oneWire);
```

```
32
33
34      //Specify the links and initial tuning parameters
35      PID myPID(&Input, &Output, &Setpoint, Kp, Ki, Kd, DIRECT);
36
37      void setup()
38      {
39          Serial.begin(9600);
40          sensors.begin();
41          pinMode(TRANSISTOR,OUTPUT);
42          //initialize the variables we're linked to
43          Input = 0;
44          Output = 0;
45
46          // set the PID parameters
47          myPID.SetMode(AUTOMATIC);
48          myPID.SetOutputLimits(0, 1);
49      }
50
51      void loop()
52      {
53          unsigned long currentMillis = millis(); // Measure the current time
54          sensors.requestTemperatures();
55          Input = sensors.getTempCByIndex(0); // Measured temperature
56          if (Input>0) { // to avoid introducing sample errors (temperature ...
                sensor gives a value equal to -127 when there is a mistake
57              myPID.Compute();
58              y =Output*255; /* Since the PID output is measured between 0 and 1,
59              it is multiply by 255 to scale it into the minimum and maximum ...
                    of the analogWrite() function*/
60          }
61          // Data processing
62          Serial.print(currentMillis);
63          Serial.print(",");
64          Serial.print(Setpoint);
65          Serial.print(",");
66          Serial.print(Input);
67          Serial.print(",");
68          Serial.print(Output);
69          Serial.print(",");
70          Serial.print(y);
71          Serial.print(",");
72          Serial.println();
73
74          analogWrite(TRANSISTOR, y); // Send the command to the transistor
75      }
```

A.4 Step simulator

```
1       #include <DallasTemperature.h>
2       #include <OneWire.h>
3
4       // Transistor wire is plugged into digital pin 10 on the Arduino
5       const int TRANSISTOR = 10;
6
7       // Data wire is plugged into digital pin 2 on the Arduino
8       const int ONE_WIRE_BUS = 2;
```

```
9
10      float Temperature;
11
12      /*******************************************************/
13      /************** Parameters to be changed ***************/
14      /*******************************************************/
15      float step = 1; // size of the step, 1 equals to an unit step and ...
            maximum power
16      float time_step = 180000; // [ms], time at which the step is started
17      /*******************************************************/
18
19      // Setup a oneWire instance to communicate with any OneWire device
20      OneWire oneWire(ONE_WIRE_BUS);
21
22      // Pass oneWire reference to DallasTemperature library
23      DallasTemperature sensors(&oneWire);
24
25      void setup()
26      {
27          Serial.begin(9600);
28          sensors.begin();
29          step *= 255; // scale up step to analogWrite()
30          pinMode(TRANSISTOR,OUTPUT);
31          analogWrite(TRANSISTOR,0);
32      }
33
34      void loop()
35      {
36          unsigned long currentMillis = millis();
37          sensors.requestTemperatures();
38          Temperature = sensors.getTempCByIndex(0);
39          if (currentMillis > time_step){
40              analogWrite(TRANSISTOR,step);
41          }
42
43          // Data processing
44          Serial.print(currentMillis);
45          Serial.print(",");
46          Serial.print(Temperature);
47          Serial.println(",");
48      }
```

Appendix B

Matlab Codes

B.1 Data Import

```
% Script that imports data from a .cvs file
% opens the import tool of matlab, select the file
imported_file = uiimport('example_file.csv');

%% filter
for ii = 2:length(imported_file.data(1:end,1))
    if abs(imported_file.data(ii,1)-imported_file.data(ii-1,1)) > 3000
        imported_file.data(ii,3) = imported_file.data(ii-1,3);
        imported_file.data(ii,1) = imported_file.data(ii-1,1);
    end
    if abs(imported_file.data(ii,3)-imported_file.data(ii-1,3)) > 1
        imported_file.data(ii,3) = imported_file.data(ii-1,3);
        imported_file.data(ii,1) = imported_file.data(ii-1,1);
    end
    if isnan(imported_file.data(ii,3))
        imported_file.data(ii,3) = imported_file.data(ii-1,3);
    end
end

% the imported file is now ready to be used, for example it can be plotted:

%% Graphs
plot(imported_file.data(:,1)/1000, imported_file.data(:,3))
grid on
xlabel('Time [s]')
ylabel('Temperature [°C]')
legend(''Experimental data')
```

B.2 PID Simulink Simulation

```
% Script that runs the Simulink close loop model
% parameters needed to run a simulation, Kp, Ki, Kd, t_finish, Setpoint, ...
    T0 and TF, which are the parameters that need to be specified in the ...
    Simulink model
%% Initialization
% Ziegler Nichols parameters
kp_ZN = 0.7148;
ki_ZN = 0.0073;
kd_ZN = 17.58;

% Cohen Coon parameters
kp_CC = 0.7988;
ki_CC = 0.0067;
kd_CC = 14.21;

% The transfer function shown below was obtained with the System ...
    Identification Tool, TF is included in the Simulink model
TF = Second_order_withu05;

%% Simulation with the Cohen Coon and Ziegler Nichols PID parameters
% With Ziegler Nichols parameters
Kp = kp_ZN;
Ki = ki_ZN;
Kd = kd_ZN;
Setpoint = 40; % [°C]
T0 = 21; % [°C]
t_finish = 10000;
% Simulation
simOut = sim('PID_model.slx',t_finish);
figure(2)
plot(simOut.T)
grid

% With Cohen Coon parameters
Kp = kp_CC;
Ki = ki_CC;
Kd = kd_CC;
Setpoint = 40; % [°C]
T0 = 21; % [°C]
t_finish = 10000;
% Simulation
simOut = sim('PID_model.slx',t_finish);
figure(2)
hold on
plot(simOut.T,'r')
hold off
grid on
xlabel('Time [s]')
ylabel('Temperature [°C]')
legend('Ziegler-Nichols PID gains','Cohen-Coon PID gains')

%% PID Tuning
% As it can be seen in the graph below, the response is unstable, Ki ...
    should be decreased
Setpoint = 40; [°C]
Kd = kd_CC;
Ki = ki_CC;
```

```
54      Kp = kp_CC;
55      simOut = sim('PID_model.slx',t_finish);
56      figure(3)
57      plot(simOut.T)
58      grid on
59      xlabel('Time [s]')
60      ylabel('Temperature [°C]')
61
62      % Decrease Ki to make it stable
63      Kd = kd_CC;
64      Ki = ki_CC/100;
65      Kp = kp_CC;
66      simOut = sim('PID_model.slx',t_finish);
67      figure(3)
68      hold on
69      plot(simOut.T)
70      hold off
71      grid on
72      xlabel('Time [s]')
73      ylabel('Temperature [°C]')
74
75      % increase Kp to decrease steady error
76      Kd = kd_CC;
77      Ki = ki_CC/100;
78      Kp = kp_CC*10;
79      simOut = sim('PID_model.slx',t_finish);
80      figure(3)
81      hold on
82      plot(simOut.T)
83      hold off
84      grid on
85      xlabel('Time [s]')
86      ylabel('Temperature [°C]')
87
88      % increase Kd to decrease overshoot
89      Kd = kd_CC*10;
90      Ki = ki_CC/100;
91      Kp = kp_CC;
92      simOut = sim('PID_model.slx',t_finish);
93      figure(3)
94      hold on
95      plot(simOut.T)
96      hold off
97      grid on
98      xlabel('Time [s]')
99      ylabel('Temperature [°C]')
100
101     % mix solution to obtain optimal solution
102     Kd = kd_CC*10;
103     Ki = ki_CC/100;
104     Kp = kp_CC*5;
105     simOut = sim('PID_model.slx',t_finish);
106     figure(3)
107     hold on
108     plot(simOut.T)
109     hold off
110     grid on
111     xlabel('Time [s]')
112     ylabel('Temperature [C]')
113     legend('Cohen Coon','Decrease Ki','Increase Kp','Increase Kd','Mix ...
            solution')
```

```
114
115     %% PID tuning
116     Kd = kd_CC*10;
117     Ki = ki_CC/200;
118     Kp = kp_CC*5;
119     simOut = sim('PID_model.slx',t_finish);
120     figure(4)
121     hold on
122     plot(simOut.T)
123     hold off
124     grid on
125     xlabel('Time [s]')
126     ylabel('Temperature [°C]')
127     Kd = kd_CC*10;
128     Ki = ki_CC/200;
129     Kp = kp_CC*3.3;
130     simOut = sim('PID_model.slx',t_finish);
131     figure(4)
132     hold on
133     plot(simOut.T)
134     hold off
135     grid on
136     xlabel('Time [s]')
137     ylabel('Temperature [°C]')
```