POLITECNICO DI MILANO
School of Industrial and Information Engineering
Master of Science in Automation and Control Engineering

EMOTIONAL BIOINSPIRED CONTROL

AI & R Lab
Laboratory of Artificial Intelligence
and Robotics of Politecnico di Milano

Relatore: Prof. Andrea Bonarini

Tesi di Laurea Magistrale di:
Francisco José Aznar Bastías, matricola 876850
Santiago Meseguer Cervera, matricola 876935

Academic year 2017-2018

To our families

Contents

List of Figures

List of Tables

Abstract

The world of robotics has grown incredibly fast in the last years and now, a world where people and robots coexist in the daily life does not seem so far. This is why the field dedicated to the human-robot interaction is becoming more and more important every day. When we talk about human interaction we can't ignore human emotions, since they play an essential role in relationships between humans. The scope of this thesis is to build a tool to express human-like emotions in robot movements from the own control of the different servomechanisms. We have created a software library that allows controlling the servomechanisms of a robot in a way that it is able to show human-like emotions according to the external conditions.

Summario

Il mondo della robotica è cresciuto in maniera incredibilmente veloce negli ultimi anni, e ora un mondo in cui persone e robots coesistono non sembra così lontano. Questo è il motivo per cui il campo dedicato all'interazione uomo-robot sta diventando ogni giorno sempre più importante. Quando parliamo di interazione uomo-robot non possiamo ignorare le emozioni umane, in quanto svolgono un ruolo essenziale nelle relazioni tra umani. Lo scopo della tesi è esprimere emozioni analoghe a quelle umane nei movimenti dei robot attraverso il controllo dei diversi servomeccanismi. È stata creata una libreria software che consente di controllare i servomeccanismi di un robot in modo che sia possible esprimere emozioni simili a quelle umane in rapporto alle condizioni esterne.

Acknowledgements

We would first like to thank our thesis advisor Professor Andrea Bonarini for being always available to solve any question about our project.

We would also like to acknowledge our friends Jordi and Carolina for making the work in the laboratory more pleasant. We are specially grateful to Idil for her support and help along all this period.

Finally, we must express our very profound gratitude to our parents for providing us with unfailing support and continuous encouragement throughout our years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them.

Thank you.

Chapter 1

INTRODUCTION

The field of robotics has grown considerably in the last years and now, there is a wide range of tasks that can be performed by robots. For instance, cooking or imitating animal movements. Robots are being improved so fast that a world where humans and robots coexist in the daily life does not seem far. This is why human-robot interaction (HRI) has a leading role in recent research.

To improve this interaction, we build it on Maslow's hierarchy of needs [3]. This theory ranks the human needs in groups that must be satisfied in a specific order. This hierarchy is represented in a pyramid divided in five levels. See Figure 1.1. According to this idea, a person must satisfy lower level needs before progressing to meet higher level needs. At the base of the pyramid are the physiological needs, which include the physical requirements for a human to survive. The second level concerns the safety needs, which covers the security and freedom from fear. The third level is the social belonging and it represents the friendship, intimacy and being part of a group (family, friends, work). The fourth level is the esteem, which includes the needs of self-esteem and self-respect. Finally, the fifth level is the self-actualization and it covers the need of a person for being able to express its best potential.

Applying this theory, the first step to establish a proper HRI is to create a safe environment. Once the safety needs are fulfilled we can enrich the interaction with an element present in any human interaction: emotions.

First of all, we need to understand how humans express and perceive emotions which is a multidisciplinary study including fields of psychology, sociology, and human communication among others. Only then, we would be able to implement emotions in robots from the engineering point of view. It is at this point where this thesis takes place.

Robots created in the last decades express emotions through certain movements, i.e. the

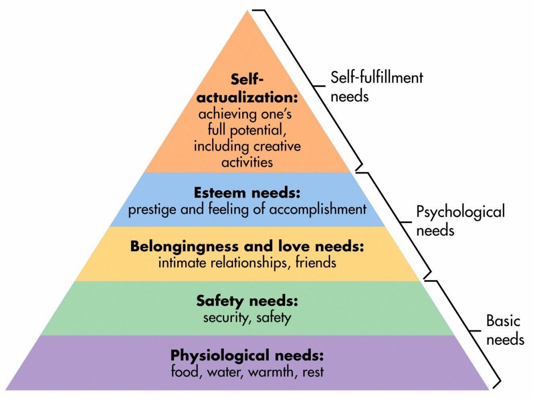

Self-actualization: achieving one's full potential, including creative activities
Esteem needs: prestige and feeling of accomplishment
Belongingness and love needs: intimate relationships, friends
Safety needs: security, safety
Physiological needs: food, water, warmth, rest
Self-fulfillment needs
Psychological needs
Basic needs

Figure 1.1: Maslow's hierarchy of needs

emotion is part of a secondary task of each robot. With this method, each robot requires different implementations for the same emotion. The aim of this thesis is to introduce emotional behavior to the movements using control theoretic tools and to implement a generic mechanism to switch between emotional modes by changing the control law.

The purpose is to create a general library in a worldwide known programming environment that implements the different controls for a set of selected emotions. To achieve this objective, we divide the work into four main parts: background analysis, simulation, implementation, and practical applications.

In 1990, psychologists that studied the relation between emotion of people and its physical response like Klaus Scherer (specialist in the psychology of emotion) and Harald G. Wallbot (psychologist in the University of Salzburg, Austria) argued that emotions are likely to be detectable through changes in the speed, rhythm and fluidity of movements[4]. This thesis is based on this principle. Instead of connecting each emotion with a different set of movements, the emotion is expressed by the quality of the motion. Therefore, the emotion of the robot can be appreciated just by watching the performance of the movements. However, an interaction is not complete if it can only express emotions. The robot should be able to change its emotions according to human actions, information that will perceive through its sensors. This behavior can make the robot to express important human values as empathy and solidarity, improving considerably the HRI.

After deciding the features of the movements corresponding to each emotion we designed the architecture of the control system and the tools to implement it on the servomechanism.

The next phase is dedicated to the simulation. We model the servomechanism including the motor and the internal controllers to develop an implicit speed controller. With a simulation software, we check the response of the system adjusting the parameters of the different PIDs until we get the desired behavior of the system. We assign a different controller the role to implement every emotion that the robot can express. The switch between emotions is simplified to an exchange of the active controller.

The third step is the implementation. We program the library that contains the methods that execute the control and the public methods that the programmer can use to make the robot express the default emotions. Each emotion is defined by the parameters of the controller that is active when the said emotion is selected. The main advantage of this solution is its flexibility. It can be adopted by all the robots that use the same variety of motors independently of their configurations.

Finally, we test the library on a robot. We define four different behaviors that combine emotions depending on the environmental conditions. The robot follows the position reference that is not determined by the emotion, allowing us to appreciate the differences in the quality of the movement, i.e., speed and acceleration when the robot changes its emotion.

The structure of the thesis is as follows:

The second chapter presents the state of the art. Topics include the safety perception of HRI and the brief history of the robots that express emotions.

In the third chapter, we introduce our approach to imitate emotional behavior in robots, its rationale and the principal advantages. In the fourth chapter, we describe the simulations and the implementation details. The fifth chapter presents the experiments. We comment on the four cases where the emotional behavior is implemented in a robot with the proposed method.

In the conclusion, the scope of the work and the evaluations are summarized and suggestions for future work are provided.

Chapter 2

STATE OF THE ART

The research area of this thesis is the study of robotics based on the human interaction, and more precisely, on the human emotions.

Human-robot interaction is a multidisciplinary field which covers areas like computer science, automation and control engineering, design and artificial intelligence. But it is not only related to engineering fields. If we want to define a human-robot interaction in a proper way we should take into account other essential areas as natural language understanding, psychology and social science. This makes that the adopted solutions become more ambiguous.

In this chapter, the first section describes the primary concern of human robot interaction, i.e. safety. And the second section explains how the focus has shifted towards the emotional perspective of the interaction.

2.1 Safe human-robot interaction

Along the 18th and 19th century the industrial revolution appeared. In this period, the factories started working with automated mechanisms and robots, which in the beginning were normally sharing the space with the workers. Here is were HRI became to gain importance.

In the first years, due to the fact that this field was new and the technology was not advanced enough, the human-robot interaction was considerably dangerous and the industrial accidents happened too often. This reason, besides the fear of losing job places, caused the workers to totally reject this new way of working.

In order to solve this problem and to calm the social situation, the first measures was to ensure the safety of the workers by not sharing the workspace with robots they are working.

Using this concept the problem was considerably reduced.

But with the improvement of the technology in the last decades, and especially the artificial intelligence, the robots are more and more flexible and dynamic making difficult to predict exactly the dimensions and geometry of their workspace limits. In that way, and assuming too difficult to avoid workers in the workspace of a robot while it is working, we should add other robust safety measures.

Nowadays, to ensure a safe human-robot interaction companies are working on the areas of motion and task planning, detection of objects and humans, intelligent behavior and force control.

Detection of objects and humans is based on a direct observation of the environment making use of position sensors. The objective of this measure is to detect near obstacles (persons or other equipment) and act according to it in order to avoid any unexpected collision.

Artificial intelligence can be also used as a safety measure and one example of this is the human movement prediction. In that feature a robot, based on its previous experience, can predict the movement that a person is going to do and avoid dangerous situations.

The principle of the force control concept is to limit the force that a robot can exert so that, if it comes in contact with a person, the force would be low enough to avoid any serious injury. There are two main strategies for force control interaction: impedance/admittance control and hybrid position and force control. The first one is based on describing a dynamic relation between interaction forces and position errors, simulating a generalized mass-spring-damper system. The second one is based on dividing the directions which are only position controlled and directions which are only force/torque controlled. In the directions constrained by the environment we establish a specific value for the force.

2.2 Emotive human-robot interaction

The first robots were confined in factories executing repetitive or dangerous tasks that humans preferred to avoid. In this case, it was not necessary for the robots to imitate emotional behaviors since the interaction with the workers was limited to programming or maintenance tasks to be performed in a short time, following pre-defined strategies. However, besides the industrial production tasks, new applications have emerged and now robots are present in social environments with entertainment, rehabilitation or care-taking objectives. This is why new functions and social behaviors must be incorporated into robots.

According to Kerstin Dautenhahn, Professor of Artificial Intelligence in the School of

Computer Science at University of Hertfordshire (England), we should define a set of social rules for robot behavior in order to make it acceptable by human [5]. These rules are known by the term ‘robotiquette’. Based on this idea, many modern interactive robots are designed and programmed to simulate human shape and behavior. We give some examples of the most advanced robots in this field, here below.

In April 2015 the ABB Company introduced the robot YuMi [6] for production applications. YuMi is a human-friendly, collaborative, dual arm, small parts assembly robot. Apart from all the safety measures as sensitive force control feedback and collision detection, it has features to improve the emotional interaction. Its structure has been specifically designed to be similar to the human body and it has been programmed to simulate human movements. It can predict the actions of the people around to act accordingly. This robot is designed especially for being part of a production line, so it is not necessary to have a more complex emotional interaction apart from the one required for a collaborative task. However, there are robots designed specifically for human interaction with the goal of entertaining so the robotiquette is essential for them.

In 2013 Prof Nadia Thalmann, director of the Institute for Media Innovation at Nanyang Technology University, Singapore, created the robot called Nadine which, according to the science editor Sarah Knapton, was the world’s most human-like robot [7]. It can meet and greet visitors, smile, make eye contact, shake hands, recognize people that it has already met and have a conversation remembering a previous one. Nadine has its own personality, mood and emotions. It can be happy or sad, depending on the topic of the conversation.

In 2014 Hiroshi Ishiguro, director of the Intelligence Robotics Laboratory at Osaka University, built a robot called Erica [8]. This humanoid has one of the most advanced artificial speech systems in the world and, in words of Dr. Ishiguro, an “independent consciousness” and “soul”. Erica is also able to develop compassion and make jokes. In April 2018 it is expected to make a national debut as a Japanese news presenter. Hiroshi Ishiguro Laboratories also developed Geminoid HI-2 (also existing in versions HI-1, HI-4 and HI-5), which has a similar appearance to Dr. Ishiguro and similar skills to Erica.

Another significant interactive robot is Pepper (SoftBank Robotics company, 2014 [9]), which was one of the first humanoid robots capable of recognizing some principal human emotions and adapt its behavior accordingly. It can recognize the emotions of joy, sadness, anger and surprise. It has been created to detect the emotions of people and deal with them in the best way.

In 2015 Hanson Robotics Ldt. built a new robot called Sophia [10], the first robot to receive a citizenship of any country (Saudi Arabia). It has been created to simulate a human conversation and has 62 facial expressions that change depending on the conversation. In January 2018, Sophia had an interview [11]. Apart from just answering the questions, it asked the interviewer “Can you imagine living your life without a cell phone?” Its creator, Dr. David Hanson said that “with Sophia we have found that the emotional connection really opens up like we’ve never seen before.”

In 2016 the Japan Science Museum exposed its new installation, Alter, a human-like robot created by robotics researchers of the Tokyo and Osaka Universities [12]. Its most interesting concept is that its movements are entirely governed by a neural network. It has a central pattern generator that allows the robot to create movement patterns of its own, influenced by sensors that detect proximity, temperature and humidity.

As we can appreciate, the method for implementing emotions in a robot is not unique. The most common method is to program directly all the movements of the robot and how it has to perform them according to its emotion, but, in this case, we are not defining a set of features that are common for each emotion; the emotion is expressed by realizing specific moves.

Other more advanced method for implementing emotions in a robot is based on artificial intelligence (AI). With this technique, instead of writing programs to simulate human behaviors, we implement an algorithm on the robot that allows it to learn how to simulate them. In that way when a robot is exposed to an HRI it can learn the different emotions from the person and how to express them.

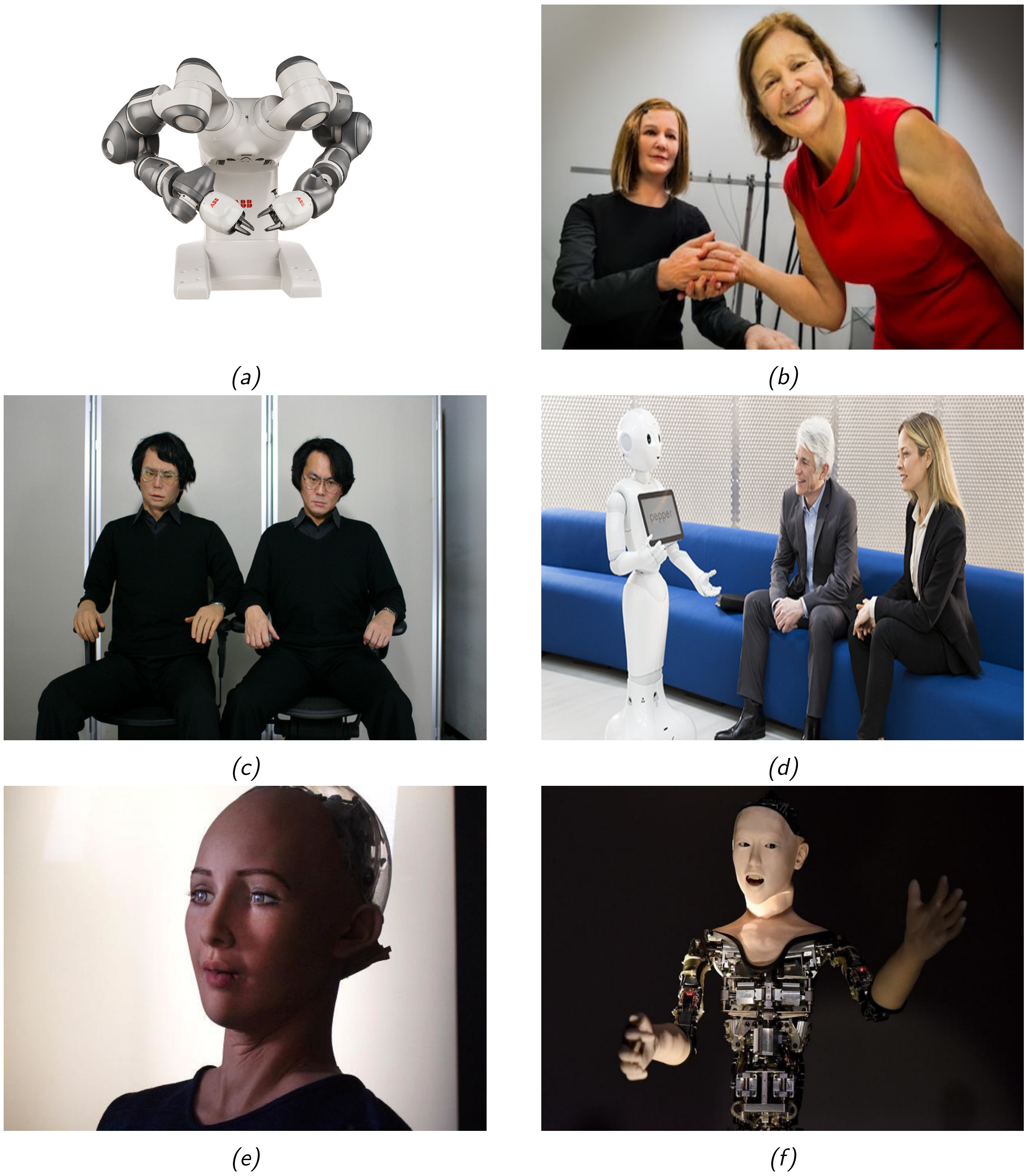


Figure 2.1: (a) YuMi, (b) Nadine and Nadia Thalmann, (c) Geminoid HI-1 and Hiroshi Ishiguro, (d) Pepper, (e) Sophia, (f) Alter

Chapter 3

PROPOSED METHOD

The problem with the current methods to implement emotional behavior in robots is that the implementation is dependent on the robot configuration, making it impossible to transfer the emotions between different robots. In addition, the emotions are not expressed by the quality of motions but by specific motions. In this thesis, we propose a method to express the emotions based on the control of the servomechanism of the robots. With our technique we do not change the movements that the robot does, they are the same for all emotions. What we change is the control system of the motor, getting different types of responses for the same reference movements.

The main advantage is that the emotions are independent on the hardware configuration of the robot because they lay over the own control of each actuator separately. It makes possible to transfer an emotion directly to any robot, without writing additional code, which is our objective.

Chapter 4

THEORETICAL INTRODUCTION

4.1 Servo behavior

A servo is a rotatory or linear actuator able to be precisely controlled due to a position sensor feedback coupled to it. It is a suitable motor for using in a closed-loop control system.

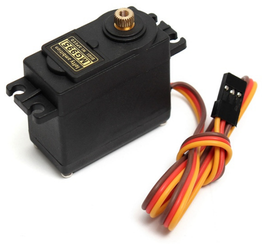


Figure 4.1: Servo motor MG995[1]

One of the most important parts of a servomotor is its internal position sensor that allows reading the position where it is in each moment. In this way it is just needed to specify the position that is has to reach and the internal position control assures it is reached.

Servomotors can rotate in both directions and the angle of rotation normally is of 180 degrees, but also there are servos with 360 degrees of rotation. They use a PWM signal (Pulse-Width Modulation) for controlling. Normally the frequency of the PWM is 50 Hz. This means that the signals that the motor receives are composed by pulses of a certain length separated from each other by 20 milliseconds. The length of these pulses determines the position it has to reach.

Normally, the pulse duration is in between 1000 and 2000 microseconds, being 1500 the middle position (see Figure 4.2), but these values change depending on the model of the servo. For instance, the servomotors used in the practical application of this work have as limit the values 544 and 2400 microseconds, corresponding to 0 and 180 degrees respectively. In order to express the position in degrees having the pulse width (μs) we perform linear interpolation.

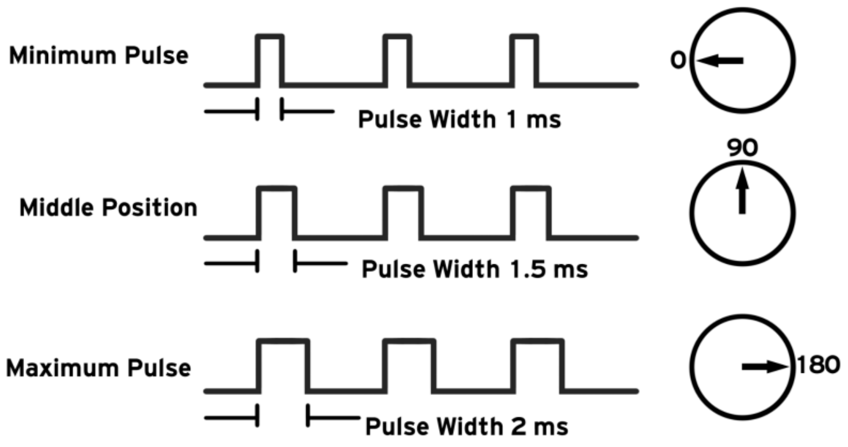


Figure 4.2: Behavior of a servomotor according to the incoming PWM signal [2]

In this way, by controlling the pulse length, the position can be controlled. We can vary the velocity of a servomotor also modifying the length of the pulses. When the servomotor reads the length of the pulse, it goes to the corresponding position at maximum speed, thanks to the internal controller. Then, we set the speed indirectly sending intermediate positions, that it reaches at maximum speed.

For example, if in a given time the position of the servomotor expressed in pulse width is 1000 μs and we want to send it to 2000 μs at maximum speed, the only thing we should do is to send a signal with pulse width of 2000 μs. In the case we want the servo to move slower, we can adopt the following solution: The width of each pulse will be constantly increasing or decreasing until it reaches the desired width. As the pulses are received every 20 ms, if we want to assign a duration time of one second to the previous movement each pulse will be 20 μs longer than the previous one. Then, after 50 pulses, that correspond to one second, the pulse width changes from 1000 μs to 2000 μs.

The maximum velocity that a servomotor can reach depends on the type of servomotors. Common commercial servomotors reach around 80 rpm of maximum velocity.

4.2 PID controller

A PID (proportional integral derivative) controller is a closed-loop control system. The input of the controller is an error calculated between a desired value, normally called reference, and a real value obtained by a measuring mechanism. The output of the controller is the control variable, that is send to the actuators with the purpose to make the real value to follow the reference value, trying to reduce the error over time. The PID is composed of:

- Proportional term. We obtain it multiplying a proportional parameter (Kp) times the current error. So the bigger this error is, the bigger the proportional term is.
- Integrative term. The goal of that term is to eliminate the residual error and it depends on the historic cumulative value of the error. It is obtained multiplying an integrative parameter (ki) times the cumulative error. This term improves the stationary region, eliminating the steady state error. However, it may produce oscillations in the transitory region.
- Derivative term. This term tries to estimate the future based on the previous evolution of the error. It multiplies a derivative parameter (kd) times the derivative of the error. This is, the control action is proportional to the error slope. It increases the stability of the system and reduces the overshooting and oscillation, improving the transitory behavior.

When we use a derivative term, typically a filter coefficient is added. The main objective of this filter is to soften the control action during the large jumps of the error. To achieve this, the filter limits the next error slope according to the previous slope of the error curve. The quantity of limitation is proportional to the filter coefficient .

In the Equation 4.1, we see the structure of a discrete PID using Backward Euler method, which is composed by three terms in this order: proportional term, integral term and derivative term. In this formula P is the proportional parameter, I is the integral parameter, D is the derivative parameter, Ts is the sample time of the PID, z is the forward shift operator and N is the filter coefficient.

A control action can include one or a combination of these parameters. Below we can see examples of different PIDs acting on a first order system.

$$P + I \cdot Ts \cdot \frac{z}{z-1} + D \cdot \frac{N}{1 + N \cdot Ts \cdot \frac{z}{z-1}} \tag{4.1}$$

- Proportional controller (P). When we use only the proportional controller, we can observe a response similar to Figure 4.3. We see that high error results in a high P control action which consequently causes a high slope. Thus the error reduces. The control effort decreases with decreasing error. The higher the proportional parameter is, the smaller steady state error and faster dynamics are. P control can reduce the rise time but it does not eliminate the steady state error. However, it can approximate it with a greater proportional parameter.

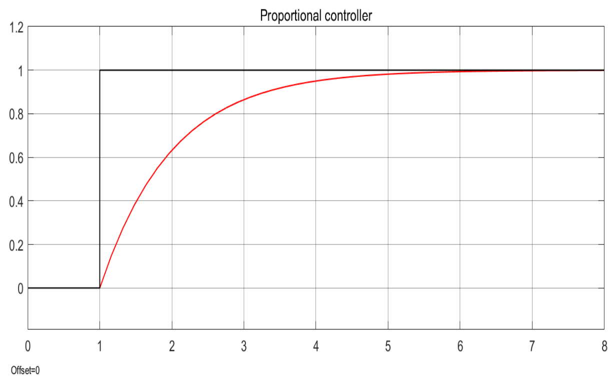


Figure 4.3: Example of a typical only proportional controller (P)

- Proportional-Integral controller (PI). A PI controller makes the steady state error zero but it does not increase the velocity of response and affects negatively the overall stability of the system. It is main used when the response speed is not an issue. It has problems when the controlled object is slightly nonlinear and uncertain. The advantage is that it is cheaper than a PID.

  In the Figure 4.4 there is an example of a response of a PI controller where we can observe the effect of the proportional (depending on the error of the last loop) and the integral (depending on all the cumulative error) terms. In the beginning the proportional part is decreasing at the same time that the error decreases too. But since the cumulative error (although the current error is decreasing) is increasing, the integral part increases too. So the proportional part is compensated with the integral part.

- Proportional-Integral-Derivative controller (PID). The principal advantage is that with the derivative part the controller is able to predict the evolution of the error in the near future, what reduces the time reaction and the oscillations (better dynamics). This controller is the most complete one and the most widely used. The proportional part helps to eliminate oscillations. The integral part makes the steady state error zero. The derivative part increases the reaction velocity and improves the stability of the system, reducing overshooting and improving the transitory region.

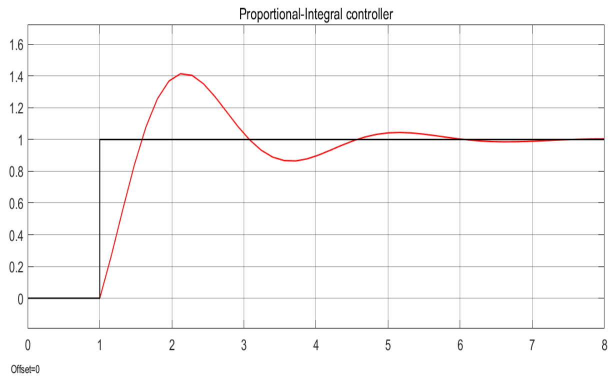


Figure 4.4: Example of a typical Proportional-Integral controller (PI)

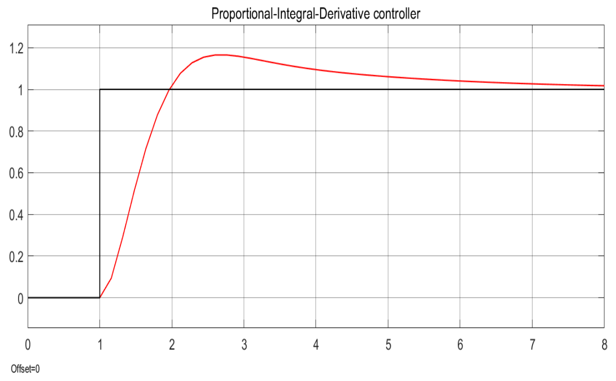


Figure 4.5: Example of a typical Proportional-Integral-Derivative controller (PID)

4.3 Arduino language and software library

Arduino uses an integrated development environment (IDE), very similar to the C and C++ languages. The C++ is an object-oriented programming language designed by Bjarne Stroustrup in the 1980s with the goal of adding to the language C the possibility of manipulation objects. In order to understand the base of the programming with C++ first it is needed to understand some important concepts as: object, class, attribute...

A class basically is a template for the creation of objects with a defined data structure. These data are organized in fields called attributes. A class is also able to perform a set of methods, which have the ability to operate also on other classes or objects.

For example a class could be "Drone" and the objects could be the different types of drones that are in the market. The attributes for that class could be: weight, maximum velocity, flight time... Moreover, its methods could be: move forward, move left, take a picture...

Another important concept is the accessibility, which can be associated to an attribute or to a method. When it is private the access is only possible for the methods that are part of the same class. However, if it is protected all the class and sub-classes have access. And finally, if it is public, any function can have access to it.

Then, using all these concepts we can program a sequence of code in order to obtain any desired behavior. But what if we want to save this behavior in order to not have to write all the code every time? Is in that moment where the use of a library acquires importance.

A software library is an implementation of behavior that can be invoked from an external program. These libraries allow obtaining some specific behavior just calling some parts of the libraries instead of implementing the entire code needed for that behavior. In other words, a library is a sequence of code structured in a way that it can be reused by external programs just knowing its interface and not the internal code. If someone wants to get the behavior that is implemented in a library it is enough to invoke the library from an independent program and the behavior will be transferred to this.

Normally software systems have their own libraries that provide most of the software possibilities, but also other programmers can create new libraries and share them with the community giving more possibilities for the system. Arduino is an open-source, which means that all the hardware and software designs are freely available under copyright licenses.

A library is mainly composed by a set of classes with its corresponding methods, also called functions. These functions are the ones that have to be called from the program that is being used in order to obtain a specific behavior attached to that function in the library.

Chapter 5

SIMULATION

Once we have studied the internal behavior of a servomotor and how we can control it, the next step is to test different controllers and to check how they respond. The objective of this part is to define a control system for each emotion.

In this case, instead of testing the controllers directly on the servomotor, we create a model in Simulink, a simulation environment inside the well-know software called Matlab.

First, we create a model that approaches the behavior of a real servomotor. Secondly, we design different control systems for the model and we simulate them.

5.1 Control system

5.1.1 Servomotor model

We write the model of the servomechanism as a first order transfer function that receives as input the signal that is sent by the microcontroller, i.e. the speed represented as a number from 0 to 255 and outputs the actual position of the motor.

As it was explained in the previous chapter, each position of the servomotor corresponds to a pulse width of a 50 Hz signal. The maximum speed is restricted mechanically, so we can calculate the maximum variation of the width of two consecutive pulses.

$$v_{max}[rpm] \cdot \frac{2 \cdot (max\_width\_pulse - min\_width\_pulse)[\mu s]}{1[rev]} \cdot \frac{1[min]}{60[s]} = v_{max}[\mu s/s] \qquad (5.1)$$

We work with timer ticks instead of microseconds. Knowing that the servo receives one pulse every 20 ms and that the timer works at 8 MHz we obtain:

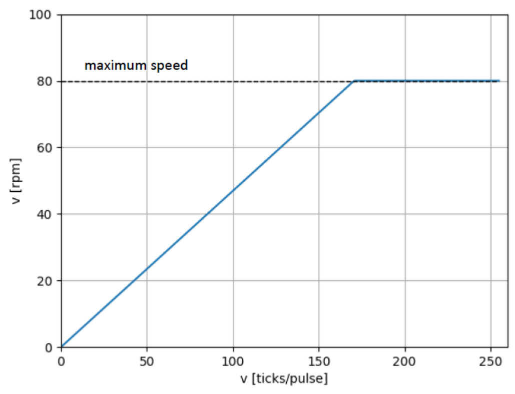


Figure 5.1: Line for mapping the velocity in variable of the servomotor to rpm

$$v_{max}[\mu s/s] \cdot \frac{2[ticks]}{1[\mu s]} \cdot \frac{1[s]}{10^3[ms]} \cdot \frac{20[ms]}{1[pulse]} = v_{max}[ticks/pulse] \tag{5.2}$$

With a maximum speed of 80 rpm and 1600 μs between maximum a minimum pulse width (typical values of commercial servos) we get a maximum difference of 170 ticks per pulse. Above this value we do not observe changes in the speed. See Figure 5.1.

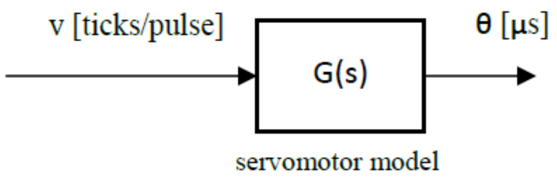


Figure 5.2: Sub-block for the change of variables

Then, the transfer function of the servomotor that inputs the ticks variation per pulse and outputs the position of the motor (see Figure 5.2) is:

$$G[s] = \frac{K}{s} \tag{5.3}$$

where:

$$\frac{v[ticks/pulse]}{v[\mu s/s]} = K = \frac{1[pulse]}{20[ms]}] \cdot \frac{10^3[ms]}{1[s]} \cdot \frac{1[\mu s]}{2[ticks]} = 25 \tag{5.4}$$

5.1.2 PID controller design

We add an implicit position PID controller that acts on the set-points of the internal controller of the servo. We assume that the internal controller of the servomotor is ideal and it follows the reference perfectly. Given the dimensions and the dynamics of the servomotors we consider, this assumption is realistic. Then, the PID should have as an input the position error and return the speed expressed as a number from 0 to 255. To express the different emotions we tune the parameters of the PID until getting the desired response. For some emotions, the derivative or even the integral action is not required so a P or PI controller it is adequate. However, in order to get a different behavior than the one we obtain with a PID, we add two extra parameters: a constant and a disturbance.

The constant is used when we want the speed of the movement to be independent from the position error. In this case, the PID does not act and the motor follows the reference without varying the speed.

The disturbance is used when we want to add a shaking on the motor to express nervousness or fear. In this case, a white noise is multiplied by a gain and added to the output of the PID.

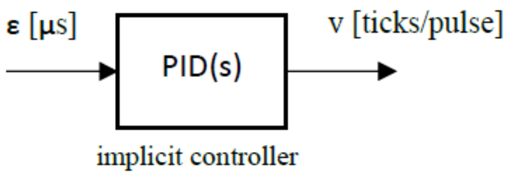


Figure 5.3: pid

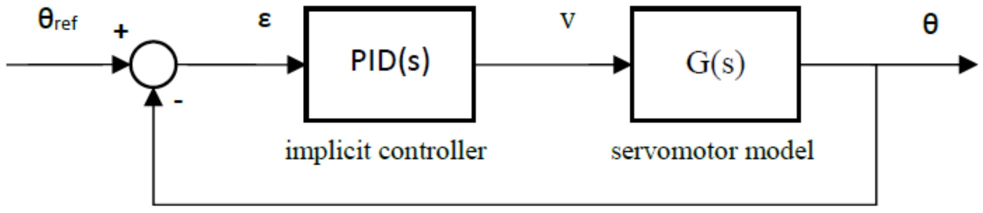


Figure 5.4: Model of the control system for a servomotor

5.2 Simulations

To tune the PID we build the closed loop on Simulink (see Figure 5.5) to get the parameters corresponding to each emotion. In the Table 5.8 we present the values obtained where P, I, D and N are the proportional, integral, derivative and filter parameters of the PID respectively, C is the value of the constant in case the PID is not acting and WN is the factor proportional to a default white noise.

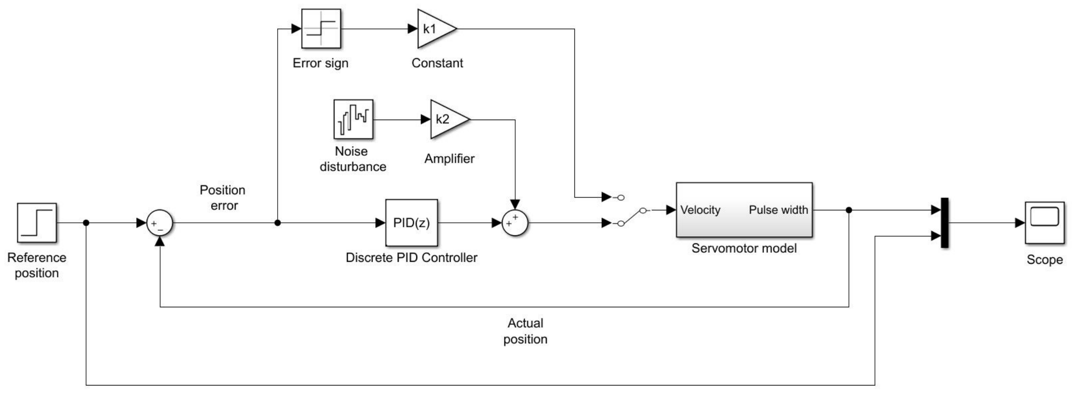


Figure 5.5: Model of the control system for a servomotor

We have to take into account that all these emotional expressions and the way to obtain them with different controllers are quite subjective. These emotions are just a set of human emotions that from a subjective point of view we have thought that are enough representative to be recognized. Obviously, more emotions could be added and also the existent ones could be modified if it is thought that they do not simulate properly an emotion.

In the next subsections we explain how all the emotions have been designed, exposing the basic behavior that they simulate and how we have implemented them in controller terms.

5.2.1 Sleepy

The sleepy emotion is characterized by very slow movements and late reactions. It is common that when we are sleepy we react slowly to an order. Then, due to the slow reaction the motions are characterized by an overpassing of the reference.

Sleepy desired response	**Implementation**
Very low accelerations and movements	Low proportional part
Overshooting	Integral part
Slow reaction and not oscillations	Derivative part
Smooth transitions	

Table 5.1: Design of sleepy

5.2.2 Bored

We design the bored emotion using constant velocity movements with the target of not expressing feelings. When a person is bored, it is not motivated to follow the reference and it starts and ends with the same velocity.

Bored desired response	**Implementation**
Constant velocity	Velocity independent of the error
Low velocity	The velocity is defined with a small constant
Not overshooting	

Table 5.2: Design of bored

5.2.3 Sad

When we are sad we perform passive movements. If we receive an order we realized it slowly and taking care of not overpassing the reference.

Sad desired response	**Implementation**
Slow movements	Low proportional part
Not overshooting	No integral part

Table 5.3: Design of sad

5.2.4 Normal

As the normal emotion we simulate a behavior in which we react fast under an order. Due to the high velocity reached we overpass the reference but we rapidly notice it and we try to stabilize in the reference the sooner as possible.

Normal desired response	Implementation
High velocity	High proportional part
Short rise time	Medium integrative part
Avoid oscillations	Medium derivative part

Table 5.4: Design of normal

5.2.5 Happy

When we are happy we are energetic and we use to pay attention about what we are doing. In that way we do the movement considerably fast but without crossing the reference.

Happy desired response	Implementation
High velocity	Medium-high proportional part
Not overshooting	No integral part

Table 5.5: Design of happy

5.2.6 Afraid

If something scares us we may become a bit paralyzed, resulting in a considerable slowing down of our movements. On the other side we could also loss the proper control of our body and it could start shaking. We do not overpass the reference because we are concerned of all our movements.

Afraid desired response	Implementation
Slow velocity	Low proportional part
Considerable shake of the body	Medium-high white noise
Not overshooting the reference	No integral part

Table 5.6: Design of afraid

5.2.7 Bit nervous

When we are a bit nervous we perform fast and accelerated movements combined with the shake of our body due to internal human reactions that prepare our muscles in front of any possible dangerous situation. The adrenaline generated also makes us to loss the total control of our movements resulting in an often overpass of the reference and impossible total stabilization on the reference.

Bit nervous desired response	**Implementation**
Fast and accelerated movements	High proportional part
Overshooting	Integral part
Shake and never stabilization	White noise

Table 5.7: Design of bit nervous

5.2.8 Very nervous

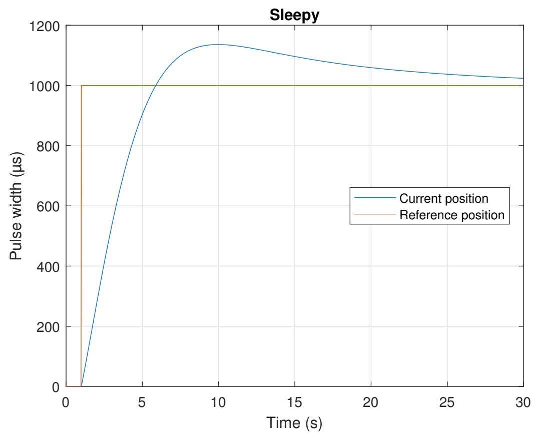
The behavior of very nervous is the same than bit nervous but with a stronger white noise.

5.2.9 Comparative of emotions

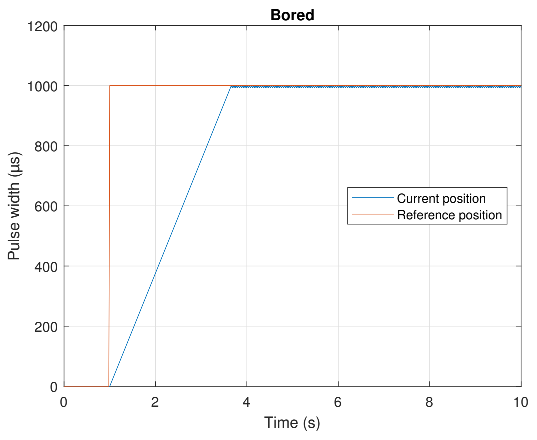
In the Table 5.8 we can see the values chosen for all the control parameters of every emotion designed. The Figure 5.6 and 5.7 are the responses of all the controllers with a step reference of 1000 μs.

Emotion	P	I	D	N	C	WN
Sleepy	0.0162112	0.0012247	-0.0088213	0.6748536	0	0
Normal	0.1467690	0.1071312	-0.0143992	9.1886803	0	0
Bit nervous	0.1467690	0.1071312	-0.0143992	9.1886803	0	0.5
Very nervous	0.1467690	0.1071312	-0.0143992	9.1886803	0	1.5
Happy	0.1089826	0	0	1	0	0
Sad	0.0377924	0	0	1	0	0
Afraid	0.0377924	0	0	1	0	1.5
Bored	0	0	0	1	15	0

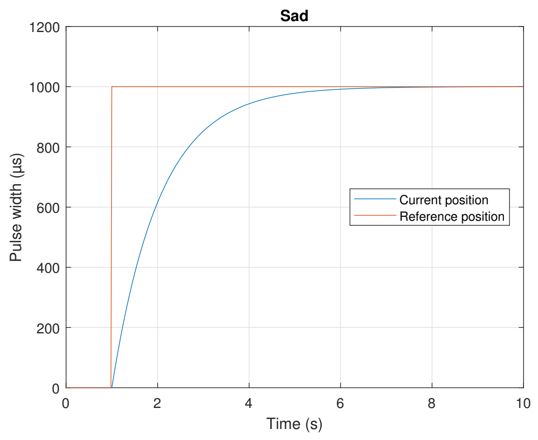
Table 5.8: Controller parameters of each emotion



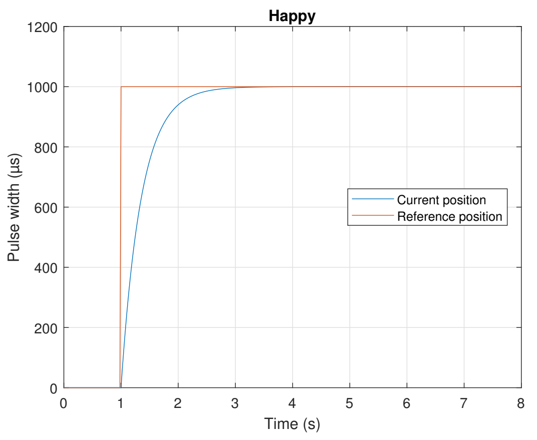
(a)



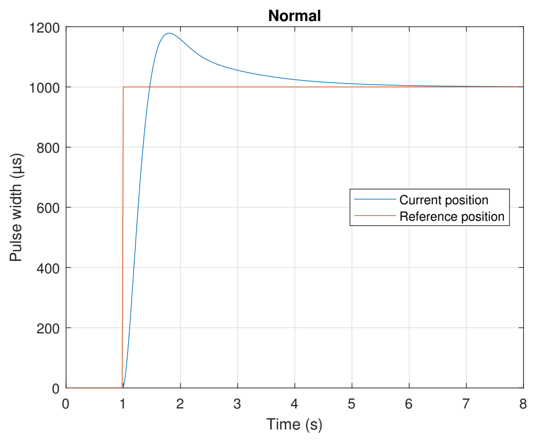
(b)



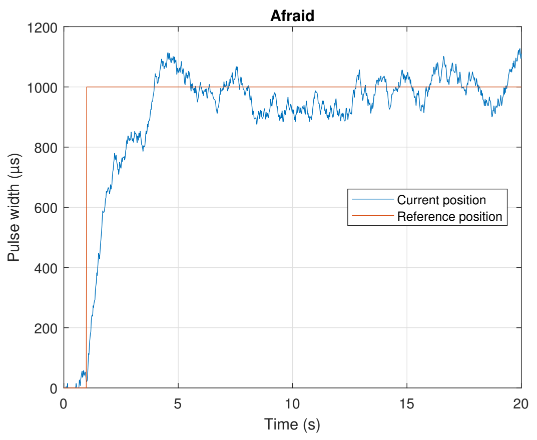
(c)



(d)

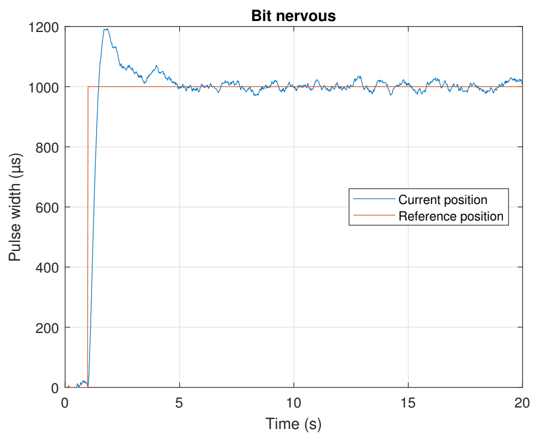


(e)

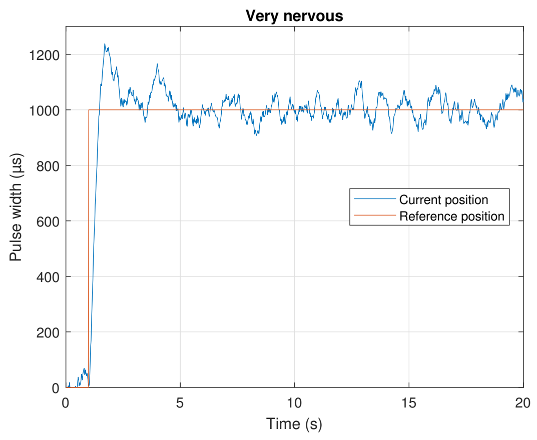


(f)

Figure 5.6: Emotion's responses of (a) sleepy , (b) bored, (c) sad, (d) happy, (e) normal and (f) afraid



(a)



(b)

Figure 5.7: Emotion's responses of (a) bit nervous and (b) very nervous

Chapter 6

IMPLEMENTATION

After the design of the different control systems in Simulink, the next step is to implement them in a hardware and software platform.

The goal of this part is to create a library which can be used for controlling servomotors in a way that any robot is able to express emotions by the quality of its movements.

First, we choose a proper software environment and then we implement all the desired behaviors and different controllers inside a library that we call *EmotionalServo*.

In this thesis we create two software libraries in Arduino environment.

- EmotionalServo: this library is the main objective of the thesis and allows to control servomotors expressing emotions thanks to a closed control loop of the position combined with a switching control system. This library is based on other already existing library called VarSpeedServo.
- EmotionalWheels: this is a secondary library created for moving DC motors with different velocities depending on the emotions, but we do not include a closed control loop.

First of all, we explain the library VarSpeedServo in order to understand its behavior. Then we explain the library EmotionalServo, and finally we talk about EmotionalWheels.

6.1 VarSpeedServo library

Before explaining how this library builds the signal that is sent to the servomotor, first of all we should mention some aspects of the timers. Arduino uses 16 bits timers so the maximum value of the timer is $2^{16} - 1 = 65535$. Working at 16 MHz it reaches this number in around 4 ms. This time is so small that if we want to build a 50 Hz signal (period of 20 ms) we need to use a prescale. The prescale slows the timer down setting the number of ticks of the clock that correspond to one tick of the timer. With a prescale of 8 the timer works at $16/8 = 2$ MHz so it resets every 32.7 ms approximately, enough for our aim.

When the first servo is attached to the system, the corresponding timer is initialized and a signal with a default pulse width is generated. This signal is generated interrupting the program every time that the value changes from LOW to HIGH and from HIGH to LOW. In the case that more than one servo is attached, in the same interruption that sets to LOW the output of each servo, the output of the next one is set to HIGH. (see Figure 6.1). This method has a little inconvenience: the time between two consecutive pulses is not exactly 20 ms when the previous servo changes its position. It is advanced or delayed with an amount δ, which is the pulse width variation of the previous servo.

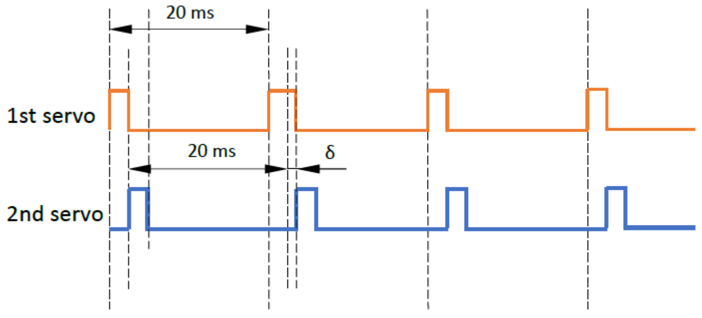


Figure 6.1: VarSpeedServo signals

The main advantage of this library is that allows to specify the speed of the movement, as long as this speed is not higher than the mechanically restricted one.

6.1.1 VarSpeedServo Class

The library declares a class, called *VarSpeedServo*, that includes methods to operate with a maximum number of servomotors depending on the timers of the board. Every time a new object of the *VarSpeedServo* class is declared, an index is assigned to it and a servo counter is increased to check whether or not the limit of servos has been exceeded. We can divide the public functions of this class in three groups: attach functions, write functions and read functions. Some of these methods are used in our library, possibly with modifications.

6.1.2 Attach Functions

This group includes the following functions:

- uint8_t attach (int pin, int min, int max):

  This function is used to attach an object to the pin number where the servo is connected. The parameters *min* and *max* indicate the maximum and minimum width pulse expressed in microseconds. These values can be found in the datasheet of the servo. It also initializes the corresponding timer if it has not been done before.

- uint8_t attach (int pin):

  This function calls the previous one with the default values for *min* and *max* (544 and 2400).

- void detach():

  This function deactivates the pin that the servo object is attached to. If the timer is not used by any servo, the interruption functions related to that timer are ended.

- bool attached():

  It returns TRUE if the object is attached.

6.1.3 Write Functions

In this group we include the functions that we use to set the position and the speed of the servo.

- void writeMicroseconds(int value):

  This function receives the pulse width in microseconds. It constrains this number between the limits of the servo and does the corresponding operations using the timer frequency to calculate the number of ticks that the signal should be active.

- void write(int value):

  This functions calls the previous one after checking the value received is greater than the minimum width. If the value is less than the minimum, it is perceived as an angle. Then, it is constrained between 0 and 180 and converted to microseconds to be used as a parameter in *writeMicroseconds*.

- void write(int value, uint8_t speed):

  This function calls the previous one when the second parameter is 0. When the speed is different from 0, it does the conversion of the parameter *value* as is done in the previous function and gets the corresponding number of ticks. But in this case, the pulse width is not directly set. This number is kept as a target position. The number of ticks that the signal is HIGH increases or decreases (depending on the current and the target position of the servo) in each pulse a constant amount specified by the parameter *speed* until the target is reached.

- void write(int value, uint8_t speed, bool wait):

  This function is used instead of the previous one when we want the program to wait until the servo reaches the desired position.

6.1.4 Read Functions

- int readMicroseconds():

  This function returns the last servo pulse width in microseconds.

- int read():

  This function returns the position servo in degrees. It calls the previous one and maps the returned value using the minimum and maximum pulse width of the servo.

6.2 EmotionalServo library

For the creation of *EmotionalServo* library we use *VarSpeedServo* library as a layout.

The first change we perform is the building of the output signals. As it is explained in the Section 6.2. the pulses are not separated in the time exactly 20 ms when the pulse width of the previous servo changes. To avoid this situation, we reserve one period of time for the pulse of each servo. This period must be bigger than the maximum pulse width of the servos. We divide the 20 ms in periods of 2500 μs so each timer can handle 8 servos at the same time (See Figure 6.2).

The second change corresponds to the initialization of some internal variables needed for control functions when an object of the class is declared.

We also define an enumerated type *emotions*. A variable of this type can be assigned any of the designed emotions as a value.

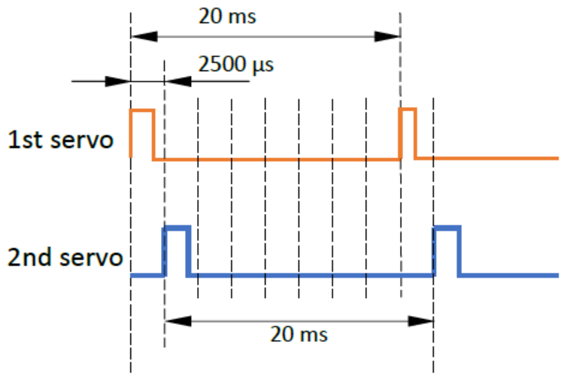


Figure 6.2: EmotionalServo signals

6.2.1 *EmotionalServo Class*

We can divide the new functions of two groups. In the first group we include the functions that compose the control system. In the second group we include a set of user interface functions to simplify the use of the library.

6.2.2 Control Functions

In this group we include two functions: *control* and *moveWithEmotionTo*. The first one computes the PID controller depending of the emotion, and the second one sets the speed and position of the servo. Below we explain in detail these two functions.

- int control (int err, emotions emo)

  The function *control* is the main part of the closed control loop. It receives as parameters the error and the emotion and returns the computed value of the control variable. As it is explained in Chapter 5, each emotion is represented by a set of six parameters (P, I, D, N, C and WN) so every time the emotion changes these parameters are actualized.

  The control variable v is calculated as the sum of the three terms of the PID (proportional, integral and derivative) plus a constant and a disturbance:

$$v(k) = up(k) + ui(k) + ud(k) + cte(k) + dist(k) \tag{6.1}$$

  The proportional term of the PID is obtained multiplying as follows:

$$up(k) = P \cdot err(k) \tag{6.2}$$

where *err(k)* is the error received as a parameter.

The integral and derivative terms are obtained using discrete Backward Euler formulation:

$$ui(k) = I \cdot err(k) \cdot sample\_time + ui(k-1) \tag{6.3}$$

$$ud(k) = D \cdot \frac{err(k) - err(k-1)}{sample\_time} + ud(k-1) \cdot \frac{1}{sample\_time \cdot N \cdot (1 + \frac{1}{sample\_time \cdot N})} \tag{6.4}$$

where *sample_time* is constant and equal to 20 ms.

The term *cte* does not depend on the magnitude but on the sign of the error:

$$\begin{aligned} cte &= C \cdot sign(err(k)) \\ sign(err(k)) &\in \{-1, 0, 1\} \end{aligned} \tag{6.5}$$

This term is different from 0 only if the PID is not acting so the parameters of every emotion should satisfy:

$$C \neq 0 \rightarrow P, I, D = 0 \tag{6.6}$$

In this case, the speed is constant and directly set by the parameter *C*.

Finally, we include the noise disturbance using a function that returns a random value. We choose a default range of -30 to 30 but this amplitude will be modified adjusting the parameter *WN*:

$$dist = random(-30, 30) \cdot WN \tag{6.7}$$

The current values of the terms *err(k)*, *ui(k)* and *ud(k)* are saved to be used as *err(k-1)*, *ui(k-1)* and *ud(k-1)* respectively in the next time step.

- int moveWithEmotionTo (int pos_ref, emotions emo)

  The function *moveWithEmotionTo* calculates the position error with the position reference that is received as a parameter and the current position of the servo that gets by calling the function *readMicroseconds*. This value is passed as a parameter together with the emotion to the function *control* that returns the speed *vel*. Then, depending on the speed sign it takes one of three different actions:(t

  – If $vel < 0$, then *write(0,abs(vel))*

  – If $vel > 0$, then *write(180,abs(vel))*

  – If $vel = 0$, then it stops the servo by passing as a parameter the current position to *writeMicroseconds*.

6.2.3 User Interface Functions

The user interface functions are created to simplify the use of the library. These functions receive as a parameter the position reference that the servo should follow, the condition to stop the movement and the emotion that sets the features of the motion by setting the controller parameters. We can classify the functions depending on the nature of these three parameters:

- Reference:

  The reference that the servo should follow can be received as:

  - a constant position *pos_ref*,
  - a variable value mapped from an analog input of the pin *pin_ref*, or
  - a variable value returned by a function *fref*.

- Stop condition:

  Depending on the stop condition the servo terminates the motion when:

  - the position stabilizes around the reference for a prefixed time,
  - the analog input of the pin *pin_stop* is greater than a constant value *value*,
  - the analog input of the pin *pin_stop* is less than a constant value *value*,
  - the analog input of the pin *pin_stop* is equal to a constant value *value*, or
  - the function *fstop* received as a parameter returns TRUE.

- Emotion

  The emotion that selects the controller must be an enumerator of *emotions* and can be received as:

  - a constant value *emo*.
  - a variable value returned by a function *femo* received as a parameter.

According to this, we define the following functions:

- void moveToPos (EmotionalServo &servo, int pos_ref, emotions emo)
- void moveToPin (EmotionalServo &servo, int pin_ref, emotions emo)
- void moveToPosUntil (Emotional &servo, int pos_ref, int pin_stop, inequationsymbol symbol, int value, emotions emo)

Function	Reference	Stop	Fixed emotion
moveToPos	Fixed position	Stable	Fixed emotion
moveToPin	Analog pin	Stable	Fixed emotion
moveToPosUntil	Fixed position	Analog pin	Fixed emotion
moveToPinUntil	Analog pin	Analog pin	Fixed emotion
moveToPosUntilFunc	Fixed position	Stop function	Fixed emotion
moveToPinUntilFunc	Analog pin	Stop function	Fixed emotion
moveToRefUntilFunc	Reference function	Stop function	Fixed emotion
emotionalmove	Reference function	Stop function	Emotion function

Table 6.1: Functions of EmotionalServo

- void moveToPinUntil (Emotional &servo, int pin_ref, int pin_stop, inequationsymbol symbol, int value, emotions emo)
- void moveToPosUntilFunc (EmotionalServo &servo, int pos_ref, int (*f)(),emotions emo)
- void moveToPinUntilFunc (EmotionalServo &servo, int pin_ref, int (*f)(),emotions emo)
- void moveToRefUntilFunc (EmotionalServo &servo,int (*fref)(), int (*fstop)(),emotions emo)
- void emotionalmove (EmotionalServo &servo,int (*fref)(), int (*fstop)(),emotions (*femo)())

In the Table 6.1 we can see the nature of the three parameters in each function.

All these functions have also their equivalent for controlling two servos at the same time.

6.3 EmotionalWheels library

To complement *EmotionalServo*, we create some functions that we include in a new library *EmotionalWheels* to operate with the wheels of the robot setting the speed also depending on the emotion.

Chapter 7

PRACTIAL APPLICATIONS

The goal of this thesis is to implement human emotions in a way that can be transferred to robots without the need of writing additional code for each different configuration. Until now we have explained how we approach the problem and our final proposed solution. The next step is to check if this solution fulfills the expected goals and to do a critical evaluation of the results obtained.

To check if the goals are fulfilled we realize experimental cases using the robot Puppy (see Figure 7.1). In this chapter we explain the different practical applications realized, exposing the logical behind them and a final evaluation of the results. Puppy is a robot composed by an

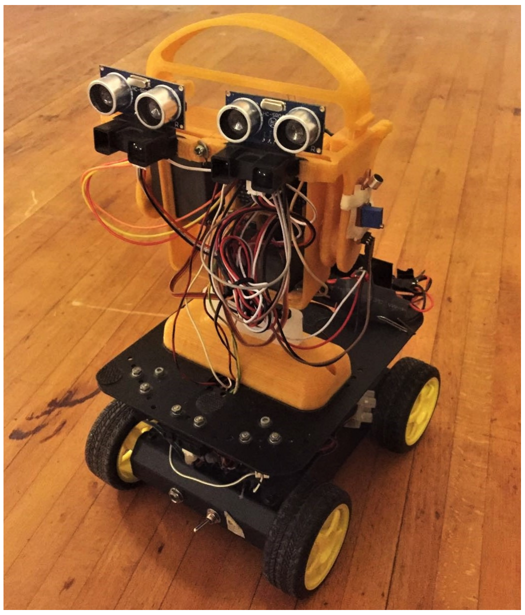


Figure 7.1: Photo of robot Puppy

ArduQuad and a robotic head implemented by designers at Phy. Co. Lab. The Arduquad is based on a Rovera 4WD Arduino Robot kit and has four wheels controlled by DC motors. The head is able to turn 180 degrees in its horizontal and vertical planes thanks to two servomotors located on it. Puppy also has a set of position sensors (infrared and ultrasonic) around the body to detect objects and allow it to interact with them.

For the practical cases realized in this thesis, which we explain in the next subsections, we have added two more sensors: a light sensor and a sound sensor.

If the reader is interested in more information about Puppy it can address to the official website of Artificial Intelligence and Robotics Laboratory (AIRLab) of Politecnico di Milano [13].

7.1 Basic behavior of the head

To be able to appreciate the change of emotion in the robot, the movements that Puppy realizes are set previously, and they do not depend of the current emotion. Then, the reference of the different servomotors of Puppy is followed continuously, but the performed motion is different.

In order to simulate a realistic motion of a head we set the reference of the motors as follows:

The servomotor that moves the head up and down has a constant reference in a middle position. The reference of the motor that turn the head in the horizontal plane changes every second to a one of five different positions: -45, -22.5, 0, 22.5 and 45 degrees, being 0 degrees the central position. To specify which position it is chosen as a reference we create a probabilistic sequence. As the central position is the most common we assign to it a probability of 50%. Turning the head to left or right has probability of 25% for each side. Now, we only have to specify the probability for moving to the two positions of each side. Since little turnings of the face are more common in the human movements we select a 75% of likelihood for the position at ± 22.5 degrees and 25% for ± 45. In the Figure 7.2 we can see a scheme of this behavior with the positions and probabilities represented.

This is the basic behavior of the head that we use in the experimental part to show the emotions.

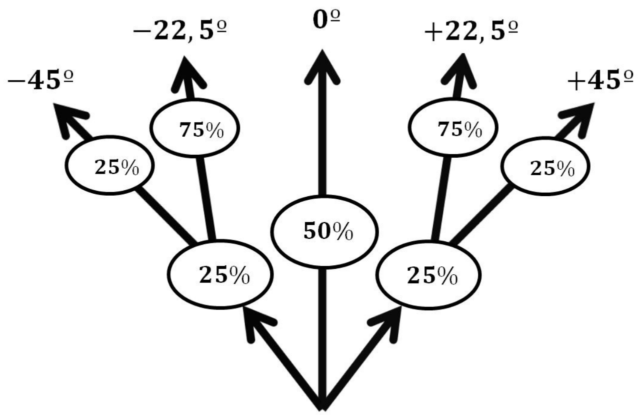


Figure 7.2: Scheme of the normal behavior of the head

7.2 Experiment 1: Sound Sensor

This practical application simulates how the emotion could be affected by unknown sudden noises. In order to be able to respond to the noise of the environment we add a sound sensor to Puppy and then we implement a program making use of the library EmotionalServo. In this case we make use of three emotions: sleepy, normal and afraid.

Puppy starts in a sleepy state. While it is not hearing any noise its movements are slow, smooth and low accelerated. When it hears a noise it changes its emotion to normal. It becomes focused and starts moving its head faster with a high acceleration rate trying to find the origin of the noise. If it does not hear any other noise within five seconds it comes back to the sleepy state. In this way, after every time that it hears a noise it is focused for five seconds.

However, every time it hears three noises in a period of five seconds it becomes afraid and starts moving slow and shaking for five seconds. After that time it behaves in the same way as it would have detected just one noise, coming back to the focused state for five seconds.

Additionally we have programmed that if Puppy does not hear any noise within ten seconds in the sleepy state, it starts singing a song thanks to a buzzer installed on it.

The results of this experimental robot are positive. It detects the noises very clearly changing of emotion every time it has to. We can easily distinguish each different emotion observing the quality of its movements.

We can see how, when it is sleepy, its movements are very slow and sometimes it does not reach completely the reference. If it changes to normal, it starts moving with high accelerations reaching quickly the reference. When it becomes afraid we can clearly notice how it starts moving slow and shaking around its head. The transitions between these three emotions are appreciable and we cannot see any instabilities.

7.3 Experiment 2: Light Sensor

In this experiment we simulate an emotional behavior that is conditioned by the light intensity. For this specific application, we add to Puppy a light sensor and we use three emotions: sleepy, normal and happy.

The emotion is selected depending of the value returned by the light sensor, in such a way that as the light intensity increases, Puppy changes its emotion mode from sleepy to normal and finally to happy.

The performing of the implemented behavior is correct. It detects the amount of light and it acts changing properly the emotions. We can appreciate in every moment which emotion it is expressing without knowing the value returned by the sensor, just paying attention in the motion.

7.4 Experiment 3: Obstacle Avoidance

In this practical case we simulate a shy personality that avoids any physical contact. For this application we introduce the use of the wheels and the emotions selected are: sad, bit nervous and very nervous.

Puppy moves randomly on the floor combining the basic behavior of the head with movements of the wheels at the same time. When it detects an obstacle nearer than a certain distance it becomes a bit nervous. When the obstacle is nearer than a limit distance, Puppy moves backward. After that, and if it does not detect any object nearer than a specific distance, Puppy continue advancing again. But if an obstacle appears suddenly very close to it, it becomes very nervous and if the obstacle gets closer it nods and turns the head and moves backward.

In this case, we can clearly appreciate the different emotions. In addition, when Puppy turns and nods the head we check that we obtain good results when both motors move at the same time.

7.5 Experiment 4: Tracking Objects

In this experiment, the robot follows either a person or an object. While Puppy does not detect an object near enough it is bored and performs the basic behavior of the head. But, when it detects an object in a certain distance it changes the behavior to happy and it starts following the object with its head. If Puppy misses the object it returns to the bored state.

For the wheels we implement an algorithm that enforce them to follow the head. In this way, Puppy not only follows the object directly with its head, but also indirectly with its wheels (body).

In this case, when it is following an object, due to the small and fast changes in the reference we have no margin to clearly appreciate the features of the emotion. We perceive a constant speed that depends on the emotion but no the rest of features (acceleration, overshooting...).

Chapter 8

SWITCHING CONTROL

This last chapter addresses the stability problem of switching control systems.

A mode switching controller consists of a finite number of modes $m_1, m_2, \dots, m_n$ which are deterministically applied in specific regions of the state space. That is, the state space is divided into regions specified by the mode boundaries. Control systems in real applications are often endowed with several actuators. It is then desirable to design switching control strategies guaranteeing that, at each instant of time, only one control is activated.

A switching control system is attached to a regulated switching law that defines the switching sequence. The switching sequence can be fixed or arbitrary. Our focus is the arbitrary switching laws where the switching is done based on external conditions.

The controller of our system changes with the choice of the emotion, which is done by the user at arbitrary times. Therefore, in order to study the stability of our system, we need to include the switching behavior to the system definition.

One of the main problems for switching systems is that all the controllers composing the system can be asymptotically stable, while the overall behavior of the system may be unstable due to an inappropriate switch. In this part, the problem arises from guaranteeing the overall stability of an arbitrary switching sequence.

8.1 Stability of systems with impulse effects

To guarantee the overall stability of a switching system we have used the theory of systems with impulse effects [14], which is based on the reset of the state system every switching time.

Considering a finite set of matrices $\mathcal{A} := \{A_p : p \in \mathcal{P}\}$, a switched time-varying system can be defined by the equation:

$$\dot{x} = A_\sigma x \tag{8.1}$$

In this equation σ is a piecewise constant signal called switching signal such $\sigma : [0, \infty) \to \mathcal{P}$. The times in which σ is discontinuous are called switching times of σ. Assuming two consecutive switching times t_1 and t_2 of $\sigma \in \mathcal{G}$, then σ is constant on $[t_1, t_2)$. The set of all piecewise constant switching signals is represented by $\mathcal{G}$. During these periods in which σ is constant the controller is required to behave as a linear time-invariant system. We also define p such $p = \sigma(t), p \in \mathcal{P}$, for every constant $\sigma(t), t \geq 0$.

Now we can redefine the system as follows.

For the intervals in which the switching signal $\sigma \in \mathcal{G}$ remains constant:

$$\dot{x} = A_\sigma x \tag{8.2}$$

While for the moment immediately after switching time t of σ we can define the rule of Equation 8.3, being r the called *reset map*. Assuming linear relation between x_c and the reset map we arrive to the Equation 8.4, where R_c is called *reset matrices*.

$$x(t) = r(x(t^-); \sigma(t), \sigma(t^{-1})) \tag{8.3}$$

$$x(t) = R(\sigma(t), \sigma(t^-))x(t^-) \tag{8.4}$$

Now, we assume that there exist symmetric and positive definite matrices $Q_p \in \mathbb{R}^{\ltimes \cap \ltimes}$: $p \in \mathcal{P}$, such

$$Q_p A_p + A_p' Q_p < 0, \quad p \in \mathcal{P} \tag{8.5}$$

The Equation 8.5 guarantees that for any interval where σ is constant and equal to $p \in \mathcal{P}$, the positive definite Lyapunov function $V_p(z) := z' Q_p z$ decreases exponentially along solutions of Equation 8.2. But this does not guarantee the stability of the overall switching system. For this we need to define an additional condition such as:

$$R(p,q)' Q_p R(p,q) \leq Q_q, \quad p, q \in \mathcal{P} \tag{8.6}$$

where $p := \sigma(t)$ and $q := \sigma(t^-)$, i.e. the system passes form the state q to p.

This equation establishes that, in the same way that the Lyapunov function must decrease exponentially for any constant interval of σ, the Lyapunov function must also decrease, or remain constant, every time that a switch takes place.

However, we should consider two important situations in which this problem takes a particular solution: absence of reset and total reset.

In the case of absence of reset the reset matrix $R(p, q)$ is equal to the identity $\forall p, q \in \mathcal{P}$. If we add this condition in 8.6 we obtain that $Q_p \leq Q_q, \forall p, q \in \mathcal{P}$. Then, and due to the arbitrariness of the switching signal, it could happen that the systems also passes from the state p to q. In this case the condition would be $Q_q \leq Q_p, \forall p, q \in \mathcal{P}$. Hence, we can conclude that $Q_p = Q_q, \forall p, q \in \mathcal{P}$, i.e. the Lyapunov function has to be the same for every piecewise switching in which σ is constant.

On the other side, if we do a total reset of the state every time there is a switch the reset matrix can be defined as a matrix of zeros $R(p, q) = 0, \forall p, q \in \mathcal{P}$. In this specific case the condition of Equation 8.6 would become $0 \leq Q_q, \forall q \in \mathcal{P}$. Since Q_q is a symmetric, positive definite matrix this last condition will be always guaranteed. In that way we arrive to the conclusion that it is not necessary any relation between the different Lyapunov functions of each system and it is enough to prove that there exist a $Q_p, \forall p \in \mathcal{P}$ symmetric and positive definite such the condition 8.5 is fulfilled, i.e. we can guarantee the overall stability of the switching system just by proving its stability for every different controller separately.

8.2 Stability analysis of the model under consideration

We base the stability of our switching system in the total reset of the state every switching time.

In the Figure 8.1 we can see a scheme of our system in closed loop control, where r is the reference, e_r is the error, σ is the switching time and y the output of the system.

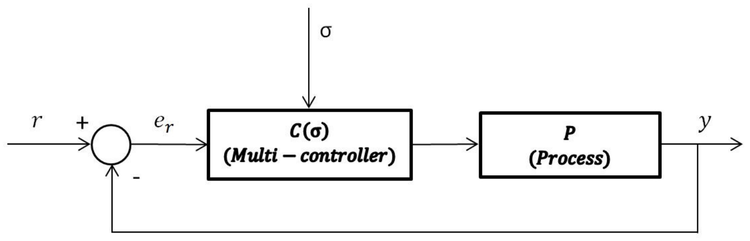


Figure 8.1: Closed control loop of our switching system

Every time that it is produced a switching time, the controller changes and the state is totally reset, ensuring the stability of the system.

Hence, the next step is to guarantee the stability of each one of the controllers separately.

From the scheme of the Figure 8.1 we can obtain the closed-loop (G_{CL}) and open-loop (L) transfer functions.

$$G_{CL} = \frac{G(\sigma)P}{1 + G(\sigma)P} \tag{8.7}$$

$$L = G(\sigma)P \tag{8.8}$$

Therefore, we can rewrite the closed-loop transfer function such as:

$$G_{CL} = \frac{L}{1 + L} \tag{8.9}$$

The process P of the system is defined by a gain and a discrete integrator to convert the velocity into position

$$P = \frac{25Ts}{z - 1} \tag{8.10}$$

where Ts is the sample time, which in our case is 20 milliseconds.

Then, if we consider all the possible configurations in which the multi-controller $C(\sigma)$ can be (P, PI, PID) and using the Backward Euler method for discrete time, we obtain from 8.8 the following open-loop transfer functions.

For only a Proportional controller (P):

$$L_P = P \cdot \frac{25Ts}{z - 1} \tag{8.11}$$

For a Proportional-Integral controller (PI):

$$L_{PI} = (P + I\frac{z}{z - 1}) \cdot \frac{25Ts}{z - 1} \tag{8.12}$$

And finally for a Proportional-Integral-Derivative controller (PID):

$$L_{PID} = (P + I\frac{z}{z - 1} + D\frac{N}{1 + NTs\frac{z}{z-1}}) \cdot \frac{25Ts}{z - 1} \tag{8.13}$$

The Nyquist stability criterion establishes that we can guarantee the stability of any system if the roots of the characteristic equation of the transfer function have a norm smaller than one.

For example, given a closed-loop system as the one in the Figure 8.2 in which the closed-loop transfer function is $\frac{G}{1+GH}$, the Nyquist criterion establishes that it is stable if the roots of the characteristic equation $(1 + GH) = 0$ are < 1 in absolute value.

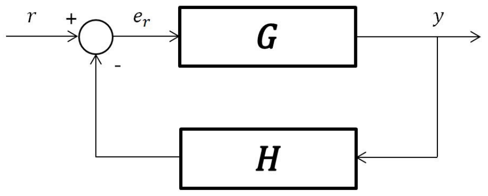


Figure 8.2: Example of closed-loop system

Applying this criterion to our model we can establish that each of our systems is stable if the roots of the characteristic equation $(1 + L) = 0$ are < 1, in absolute value.

In the particular case in which we add a noise in the output of the PID the system follows

$$y = \frac{G(\sigma)P}{1 + G(\sigma)P} r + \frac{G(\sigma)P}{1 + G(\sigma)P} dist \tag{8.14}$$

where r is the input reference and $dist$ is the noise.

We can easily notice than the characteristic equation of the closed-loop transfer function of the output y with respect r is the same than with respect $dist$ loop transfer function results as:

$$\frac{y}{r} = \frac{G(\sigma)P}{1 + G(\sigma)P} \qquad \frac{y}{dist} = \frac{G(\sigma)}{1 + G(\sigma)P} \tag{8.15}$$

Since the two characteristic equations are the same, we can conclude that ensuring the stability of the controller respect the reference, we also ensure it respect the noise.

To guarantee the overall stability of the switching system it has to be true for all the controllers. This is, the roots of the following characteristic functions have to be less than one in absolute value.

$$\begin{aligned} 1 + L_P &= 0 \\ 1 + L_{PI} &= 0 \\ 1 + L_{PID} &= 0 \end{aligned} \tag{8.16}$$

Hence, each time that we add a new emotion we have to prove that its corresponding characteristic equation 8.16 satisfies the condition to ensure that every controller is stable. Doing this and performing a total reset of the state at every switching time we guarantee that the overall switching system is stable. In the Figures 8.3 and 8.4 we see the poles and zeros of the chosen emotional controllers. We can appreciate that all the poles are inside the circle with unitary radius and center in the origin.

For example, assume that the controller selected is a Proportional controller (P). The closed (G_{CL_P}) and open (L_P) loop transfer functions are the followings.

$$\begin{aligned} G_{CL\_P} &= \frac{P\frac{25Ts}{z-1}}{1+P\frac{25Ts}{z-1}} \\ L_P &= P\frac{25Ts}{z-1} \end{aligned} \tag{8.17}$$

Then according to Nyquist criterion

$$\begin{aligned} 1+L_P &= 0 \\ 1+P\frac{25Ts}{z-1} &= 0 \end{aligned} \tag{8.18}$$

with $Ts = 20$ milliseconds

$$\begin{aligned} 1+P\frac{0.5}{z-1} &= 0 \\ z-1+0.5P &= 0 \\ z &= 1-0.5P \end{aligned} \tag{8.19}$$

It has to satisfy the condition

$$\begin{aligned} \|z\| &< 1 \\ \|1-0.5P\| &< 1 \end{aligned} \tag{8.20}$$

And this can only be true if:

$$P \in (0,4) \tag{8.21}$$

In this way, for a controller composed just by a Proportional action, we can guarantee its stability choosing any positive P smaller than four.

These results have the following explanations for our physical model:

- $P < 0$. If the proportional is negative we would assign a velocity contrary to the error. That would make the error to continually increase and the motor would never be stabilized.

- $P > 4$. On the other side, if the proportional is bigger than four, the system would enter into an oscillatory and unstable state. For example, an error of 100 μs and $P = 4$ would return a velocity of 400 $ticks/pulse$, which is the same than 200 $\mu s/pulse$. The consequence would be that the motor overpasses the reference, ending up with the same error but in the opposite direction. That would make the motor to never reach the reference.

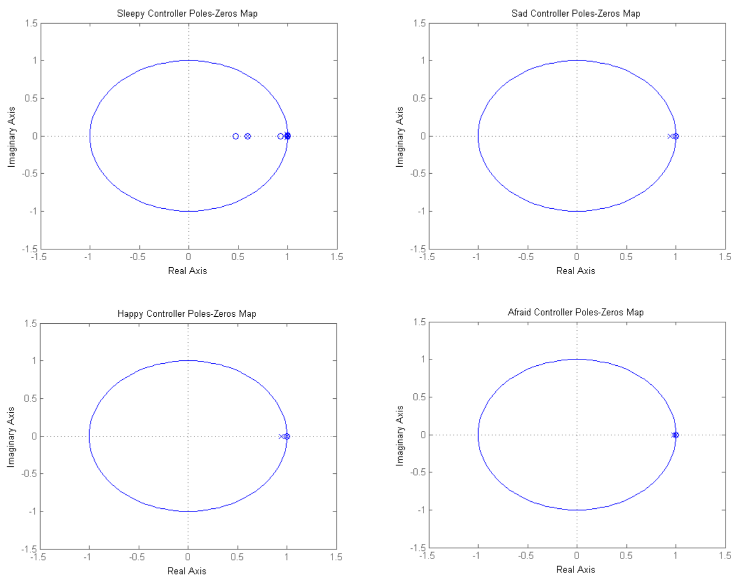


Figure 8.3: Poles and zeros transfer function with the Sleepy, Sad, Happy and Afraid controllers

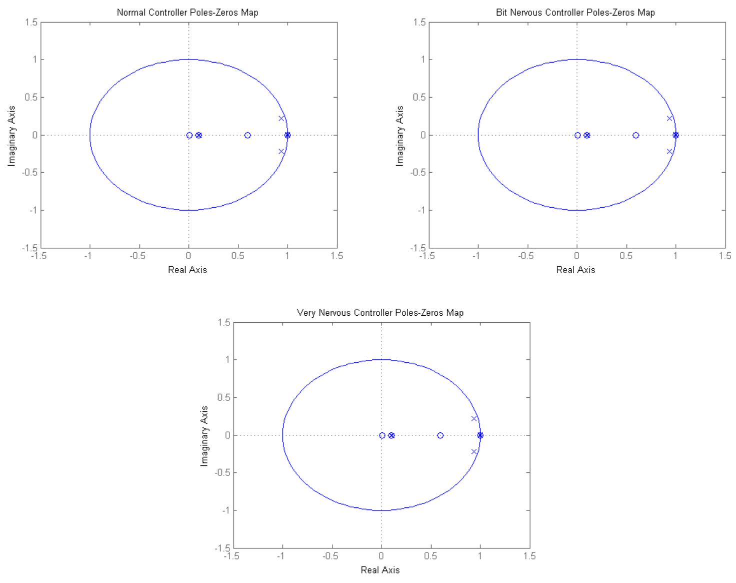


Figure 8.4: Poles and zeros transfer function with the Normal, Bit nervous and Very nervous controllers

Chapter 9

CONCLUSION AND FUTURE WORK

On a continuously growing field of robotics the specific area of HRI is acquiring a considerable relevance. Emotions are becoming a part of robots and how to implement them is an important issue. Besides the most common method of implementing the different emotions directly in the code, we propose an alternative approach. The method used in this thesis lays on expressing emotions based on the control of the servomechanisms contained in a robot. The main advantage of our approach is that we do not create an emotion for an specifically robot as the common method does. Basing the control directly on the actuators we can express emotions without the need of writing additional code for every different robot's configuration.

With the help of a simulation tool we have designed different controllers for a set of emotions according our personal judgment. Then we have tested in a physical robot the behavior of each control. In order to obtain more human-like responses we have simulated possible situations of a person's daily-life in which the robot changes the emotion depending on external conditions.

The results obtained from the practical experiments have been positive. Despite the robot follows the same reference for all emotions, we can clearly appreciate each emotion in the way it moves. According to this, we can prove that emotions can be also expressed by the quality of the movements using a control directly on its actuators and not necessarily writing extra code.

On the other side we have guaranteed the overall stability of the switching system performing a total reset of the controller state at every switching time and ensuring the stability of every controller separately. We can see how the transitions between different emotions are clear without any undesired jump or loss of the reference.

The work done in this thesis adds a new point of view for implementing emotions in robotics. However, according to the cited common method we agree that for obtaining a truly human-like behavior it is also necessary to program different movements that we do for expressing some particular emotions. We think that an emotion should be expressed by the movements the robot does and how it perform them. The movements can only be implemented writing a specific code for each robot or using algorithms based on AI, while the way robots do the movements can be easily implemented using EmotionalServo. Hence we do not present our method as a real competing alternative to the common one. What we propose with this thesis is a combination of both methods.

Human emotions and the way in which they express them depend considerably on the person. On the other side, it is well known that people transmit their emotions in a very wide variety of forms, making this issue under consideration very abstract and subjective. It is important to consider that the design of each emotion in this thesis has been developed following the common sense and experience of the authors.

Therefore, one possible future improvement proposal could be to design more objective and global emotions.

The improvement we propose is to develop an experimental method to obtain the parameters of the controllers based on the analysis of the response of people acting under different emotions. In order to reduce the subjectivity of the topic we should do this study for people from different age, social and professional groups. Additionally, since it is impossible to do a general emotion for all people, we must design the same emotion in different forms. For example, a robot destined to deal with elder people should have different ways of expressing emotions than one destined to deal with the children of a nursery. And from here we come back again to one of the initial ideas exposed in this thesis: We have to make robots adapt to humans, not humans to robots.

Bibliography

[1] Nextia Fenix. Servo motor mg995.

[2] Miymakers website. Controlling servo motor with a potentiometer!, 12/05/2017.

[3] Simplypsychology.org Saul McLeod. Maslow's hierarchy of needs. 2017.

[4] Klaus Scherer and Harald G. Wallbott. Expression of emotions, chapter 6, pages 345-422. 1990.

[5] Prof. Dr. Kerstin Dautenhahn. Socially intelligent robots: dimensions of human–robot interaction. phil. trans. r. soc. b. 362 (1480): 679–704. (29 April 2007).

[6] ABB. Abb introduces yumi®, world's first truly collaborative dual-arm robot. 2015.

[7] video source Nanyang Technological University. Sarah Knapton, science editor. Meet nadine, the world's most human-like robot robot. 29 December 2015.

[8] Mail Online Phoebe Weston. Erica, the creepy robot that is so life-like she appears to 'have a soul', will replace a japanese tv news anchor in april. 30 January 2018.

[9] Softbank Robotics website. Pepper, robots, 2014.

[10] Hanson Robotics website. Sophia, robot, 2015.

[11] Las Vegas Zoe Kleinman, Technology reporter. Ces 2018: A clunky chat with sophia the robot. bbc news, technology. 9 January 2018.

[12] Anthony Cuthbertson. New robot doesn't need humans to control it. newsweek. 8 February 2016.

[13] Puppy. In *AIRWiki. Artificial Intelligence and Robotics Laboratory. Department of Electronics, Information and Bioengineering of Politecnico di Milano.*

[14] A. S. Morse. Department of Electrical J. P. Hespanha, University of California Computer Engineering, and Yale University. Department of Electrical Engineering. Switching between stabilizing controllers. automatica 38 (2002) pages 1905-1917.

Appendix A

EmotionalServo library

In this thesis we have created a library that allows to control servomotors expressing a set of designed human-like emotions. Apart from the academic objective, this thesis has the goal of sharing this library in order to improve the Arduino community, adding new behaviors freely to use by any user. The user is also allowed to copy the code and modify it as it desires, specially the emotion controllers because is the most subjective part.

In this way, if other users have to use EmotionalServo, they should know first how it works. In the Chapter 6 we have seen how to use the different functions specific from this library. However, in this appendix we explain how to modify the control parameters of each emotions as well as how to add more emotions. Then in the second section we attach the Arduino code of the new functions that our library adds so the user can understand them better.

A.1 User manual

If we want to create or delete a emotion first we have to go to the .h file of the library and address to the variables declaration part. There we can see a variable type called *emotions*, which has an enumeration structure. This means that any variable of type *emotions* can get the values that are inside the brackets. The first value is *noemotion*, which we should not change it because it is the initial value that the library takes. When we use for first time a function and we establish an emotion, the value of the variable of type *emotions* change to the specified emotion and all the control parameters are reset and updated. For adding a new emotion we just have to write its name inside the brackets of *emotions* as the others are. If we want to delete an emotion we have to remove the name.

Now we have to open the .cpp file and look for the function *control*. There we can see all the control parameters for every emotion inside a switch configuration. If we have added a new emotion we have to create a new case with the name of the emotion and define all its parameters in the same way than the parameters of the other controllers are done. However, if we have deleted an emotion from *emotions* we should also delete the case attached to that emotion.

On the other side, if we desire to change the control parameters of an existing emotion because we think that they are not very realistic we have to open the .cpp file and address again to the function *control*. Now we just have to change the control parameters inside the case attached to the emotion that we want to modify.

Inside the controllers parameters we have the normal PID structure using the Backward Euler method with a proportional (P), integral (I) and derivative (D) terms and the derivative filter coefficient (P). The changes on these parameters are the same than for any other PID of the same type.

Then, there is also a constant (WN) used to reduce or amplify the noise to obtain lower or higher shaking. The noise is generated by a random number between -30 and +30. If we set the constant WN to one, the speed value returned by the controller will be reduced or increased by a maximum value of 30 $ticks/pulse$.

Finally, there is another constant (C) that multiplies directly the sign of the error (-1, 0, 1). In this way, if C is 50 the input velocity sent to the servomotor will be 50 $ticks/pulse$.

The user can freely change the values of all the parameters, but we strongly recommend to check first the parameters in a simulation environment in order to avoid undesired behaviors.

A.2 EmotionalServo functions

```
EmotionalServo::EmotionalServo()
{
  if( ServoCount < MAX_SERVOS)
  {
    this->servoIndex = ServoCount++;
    servos[this->servoIndex].ticks = usToTicks(DEFAULT_PULSE_WIDTH);
    this->curSeqPosition = 0;
    this->curSequence = initSeq;
    last_emotion[this->servoIndex] = noemotion;
    last_err[this->servoIndex] = 0;
    timelast[this->servoIndex] = 0;   }
  else
    this->servoIndex = INVALID_SERVO ;
}

int EmotionalServo::moveWithEmotionTo(int pos_ref, emotions emo)
{
  unsigned long timecurr;
  int wait = 1;
  while(wait)
  {
    timecurr = millis();
    if( timecurr - timelast[this->servoIndex] >= 20)
       wait = 0;
  }
  int pos_curr = readMicroseconds();
  this->pos_curr = pos_curr;
  this->pos_ref = pos_ref;
  int pos_err=pos_ref-pos_curr;
  int vel=control(pos_err,emo);

  if(vel<=0) write(0,abs(vel));
  if(vel>=0) write(180,abs(vel));
  if(vel==0) writeMicroseconds(pos_curr);

  timelast[this->servoIndex] = millis();
  return pos_curr;
}
```

Listing A.1: Declaration of the EmotionalServo class and function moveWithEmotionTo

```
int EmotionalServo::control(int err,emotions emo){
  int v, dif_err=err-last_err[this->servoIndex];
  static double P[MAX_SERVOS],I[MAX_SERVOS],D[MAX_SERVOS],N[MAX_SERVOS
    ],WN[MAX_SERVOS],C[MAX_SERVOS];
  static double up[MAX_SERVOS],ui[MAX_SERVOS],ud[MAX_SERVOS],cte[
    MAX_SERVOS],dist[MAX_SERVOS];
    if (last_emotion[this->servoIndex] != emo){
     up[this->servoIndex] = 0;
     ui[this->servoIndex] = 0;
     ud[this->servoIndex] = 0;
     cte[this->servoIndex] = 0;
     dist[this->servoIndex] = 0;
     switch(emo) {
     case sleepy:
        P[this->servoIndex]=0.0162112145797082;
        I[this->servoIndex]=0.00122473304172952;
        D[this->servoIndex]=-0.00882132576341721;
        N[this->servoIndex]=0.674853626856053;
        C[this->servoIndex]=0;
        WN[this->servoIndex]=0;
      break;
      case...
        // Here we define the rest of the emotions with its
    corresponding parameters
      break;    }
  up[this->servoIndex]=P[this->servoIndex]*err;
  ui[this->servoIndex]=I[this->servoIndex]*err*sample_time+ui[this->
    servoIndex];
  ud[this->servoIndex]=(D[this->servoIndex]*dif_err/sample_time+ud[
    this->servoIndex]/sample_time/N[this->servoIndex])/(1+1/sample_time
    /N[this->servoIndex]);
  cte[this->servoIndex]=C[this->servoIndex]*sign(err);
  dist[this->servoIndex] = random(-30,30)*WN[this->servoIndex];
  v=up[this->servoIndex]+ui[this->servoIndex]+ud[this->servoIndex]+cte
    [this->servoIndex]+dist[this->servoIndex];
  if(v==0 && up[this->servoIndex]+ui[this->servoIndex]+ud[this->
    servoIndex]+cte[this->servoIndex]+dist[this->servoIndex] != 0){
     v = sign(up[this->servoIndex]+ui[this->servoIndex]+ud[this->
    servoIndex]+cte[this->servoIndex]+dist[this->servoIndex]);    }
  last_err[this->servoIndex]=err;
  last_emotion[this->servoIndex]=emo;
  return v;
}
```

Listing A.2: Function control

```
void moveToPos (EmotionalServo &servo, int pos_ref, emotions emo){
  int pos_curr;
  int stable_count = 0;
  int stable = 0;
  int time_stable = 100;
  int margin = 3;
  while (stable == 0){
    pos_curr = servo.moveWithEmotionTo(pos_ref, emo);
    if(abs(pos_ref - pos_curr) <= margin){
      stable_count++;
    }
    else {
      stable_count = 0;
    }
    if(stable_count >= time_stable/20){
      stable = 1;
    }
  }
  servo.writeMicroseconds(servo.readMicroseconds());
}
void moveToPin (EmotionalServo &servo, int pin_ref, emotions emo){
  int pos_curr;
  int stable_count = 0;
  int stable = 0;
  int time_stable = 100;
  int margin = 3;
  while (stable == 0){
    int pos_ref = analogRead(pin_ref);
    pos_ref = map(pos_ref, 0, 1023,MIN_PULSE_WIDTH - servo.min * 4,
   MAX_PULSE_WIDTH - servo.max * 4);
    pos_curr = servo.moveWithEmotionTo(pos_ref, emo);
    if(abs(pos_ref - pos_curr) <= margin){
      stable_count++;
    }
    else {
      stable_count = 0;
    }
    if(stable_count >= time_stable/20){
      stable = 1;
    }
  }
  servo.writeMicroseconds(servo.readMicroseconds());
}
```

Listing A.3: Function moveToPos and moveToPin

```
void moveToPosUntil (EmotionalServo &servo, int pos_ref, int pin_stop, inequationsymbol symbol, int value ,emotions emo){
  int pos_curr;
  while ((symbol == lessThan && analogRead(pin_stop) >= value) || (symbol == greaterThan && analogRead(pin_stop) <= value)){
    pos_curr = servo.moveWithEmotionTo(pos_ref, emo);
  }
  servo.writeMicroseconds(servo.readMicroseconds());
}
void moveToPinUntil (EmotionalServo &servo, int pin_ref, int pin_stop, inequationsymbol symbol, int value ,emotions emo){
  int pos_curr;
  while ((symbol == lessThan && analogRead(pin_stop) >= value) || (symbol == greaterThan && analogRead(pin_stop) <= value)){
    int pos_ref = analogRead(pin_ref);
    pos_ref = map(pos_ref, 0, 1023, MIN_PULSE_WIDTH - servo.min * 4, MAX_PULSE_WIDTH - servo.max * 4);
    pos_curr = servo.moveWithEmotionTo(pos_ref, emo);
  }
  servo.writeMicroseconds(servo.readMicroseconds());
}

void moveToPosUntilFunc (EmotionalServo &servo, int pos_ref, int (*f)(),emotions emo){
  int pos_curr;
  while ((*f)() != 1){
    pos_curr = servo.moveWithEmotionTo(pos_ref, emo);
  }
  servo.writeMicroseconds(servo.readMicroseconds());
}

void moveToPinUntilFunc (EmotionalServo &servo, int pin_ref, int (*f)(),emotions emo){
  int pos_curr;
  while ((*f)() != 1){
    int pos_ref = analogRead(pin_ref);
    pos_ref = map(pos_ref, 0, 1023, MIN_PULSE_WIDTH - servo.min * 4, MAX_PULSE_WIDTH - servo.max * 4);
    pos_curr = servo.moveWithEmotionTo(pos_ref, emo);
  }
  servo.writeMicroseconds(servo.readMicroseconds());
}
```

Listing A.4: Function moveToPosUntil moveToPinUntil and moveToPosUntilFunc moveToPinUntilFunc

```cpp
void moveToRefUntilFunc (EmotionalServo &servo,int (*fref)(), int (*
    fstop)(),emotions emo)
{
  int pos_curr;
  while ((*fstop)() != 1){
    int pos_ref = (*fref)();
    pos_curr = servo.moveWithEmotionTo(pos_ref, emo);
  }
  servo.writeMicroseconds(servo.readMicroseconds());
}

void emotionalmove (EmotionalServo &servo,int (*fref)(), int (*fstop)
    (),emotions (*femo)())
{
  int pos_curr;
  while ((*fstop)() != 1){
    int pos_ref = (*fref)();
    emotions emo = (*femo)();
    pos_curr = servo.moveWithEmotionTo(pos_ref, emo);
  }
  servo.writeMicroseconds(servo.readMicroseconds());
}
```

Listing A.5: Function moveToRefUntilFunc and emotionalmove