Integration of RC Vehicles in a Robotic Arena

Nuno Gonçalo Pires Martins

Thesis to obtain the Master of Science Degree in

Aerospace Engineering

Supervisor(s): Prof. Carlos Baptista Cardeira
Prof. Paulo Jorge Coelho Ramalho Oliveira

Examination Committee
Chairperson: Professor João Manuel Lage de Miranda Lemos
Supervisor: Prof. Carlos Baptista Cardeira
Member of the Committee: Prof. Agostinho Rui Alves da Fonseca

25 November 2016
“leve leve”
Acknowledgments

I would to start my acknowledgements, by thanking as well as apologizing to my girlfriend for every single time that I wasn’t there for her, due to my academic requirements or other obligations. I am grateful for all her patience, her understanding and her support, and I promise that I will make up for this, big time. Thank-you Raquel.

To all my friends, my very big thank-you. Although I feel that it would not be fair if I did not highlight the importance of four special mates that played a very important role in my college life.

To Jorge, I am grateful, for all the times that you gave me a roof over my head, just so that I was closer to college, to better full fill the academic compromises, but also for our great dinners with friends. I hope that we can continue to enjoy my mother’s Portuguese Stew, and maybe one day you will get lucky to try my grandmother’s Portuguese Stew.

To David (that also likes Portuguese Stew) I thank-you for every single time that you helped me, and for your endless patience with me. I hope to someday return all the good advice that you gave me.

To Passinhas, the guy that I met many years ago at a summer internship, and that I found while doing my master degree to share many good times (and some of Mom’s Stews). Thank-you and I wish you success in your new challenge, the very one I am now finishing.

To Daniel, the most recent of the four, that in his own way shared with me these last crazy times - many thanks. One day, there will be a Portuguese Stew for you too.

To these four people, I am forever grateful by being my friends, and for all the great moments that we shared and spent together.

I am grateful to my family. To my Mother, thanks for all the Stews. To my sister, that by being a Vegetarian, meant there was a lot more Stew left over for us!

Many thanks to Priscilla, for accepting to review my written English, even though you do not know me personally (yet).

Many thanks to Pedro JAG, an old and talented friend that has given me the great pleasure of doing the illustrations for the schemes that integrate the written document of my dissertation.

To Mónica from the mechanical bar, for the countless big coffees, the chocolates in the less good days and all the sympathy that I will never forget. I am grateful for this.

I wish to also acknowledge my colleague Tiago, that worked by my side in the lab and to the Optimal Control students that used a preliminary version of my work and that as such helped me to greatly improve it.

To Mr. Raposeiro and Eng. Camilo, two very important people that manage the lab where I work and all its materials and resources. I express my gratitude for their availability to help me every single time that I asked.

Last, although in no way less important, I want to acknowledgement all the professors who care about your students. All the time and dedication that you give us: teaching us and helping us mould and pave the way so that we can go forward and make something great with your teachings.

A special word to my two supervisors, for the working conditions that you provided me.
Resumo

Este trabalho integra um esforço conjunto para produzir uma arquitetura global capaz de ser utilizada por estudantes para implementar modelos práticos de controlo, de atitude e de posição, em ambientes com um ou com múltiplos agentes.

Ao longo deste trabalho desenvolveram-se soluções integradas para permitir a utilização de veículos RC comercialmente disponíveis, sob um novo conceito de Arena Robótica, em ambientes onde a utilização de GPS é interdita.

A arquitetura foi desenvolvida com foco na versatilidade, replicabilidade e fiabilidade, mas mantendo o custo final acessível. Foram seguidas duas abordagens diferentes, de modo a atingir o mesmo objetivo, mas com diferentes perspetivas de desenvolvimento. Uma das abordagens é baseada em Arduino e outra baseada em Raspberry Pi. Estabelecidos os pré-requisitos dos sistemas, diferentes módulos de hardware foram integrados para responder às necessidades e desenvolver soluções tais como: a comunicação com o veículo, a capacidade de atuação dos motores e a recolha de dados da sua atitude. No final estas e outras soluções independentes foram integradas, originando os dois sistemas alternativos que permitem a utilização de veículos na arquitetura mais global. Um modelo de Simulink capaz de integrar os blocos de controlo a serem produzidos pelos estudantes, foi desenvolvido para comandar remotamente os veículos e interagir com os seus subsistemas.

Deste trabalho, resultam duas arquiteturas integradas alternativas, completamente independentes, prontas a ser replicadas e utilizadas pelos alunos nas aulas práticas de controlo.

Palavras-chave: Arquitetura Integrada, Arena Robótica, Ambientes sem GPS, Comunicação Remota, Veículos RC.
Abstract

This work is integrated in a group effort to produce an architecture capable of being used by students, for a practical implementation of attitude and position control models, whether it be for a single or a multiple agent environment.

Integrated solutions, were developed to allow for the use of commercially available Radio Control (RC) vehicles under a new concept of Robotic Arena, within GPS-denied environments.

This architecture was designed with focus on replicability, versatility and reliability, while aiming for a low-cost solution. Two different approaches were taken, one based in an Arduino, and another based in Raspberry Pi. These different approaches were developed to accomplish the same purposes, but with the potential to provide alternative capabilities. Once the pre-requirements of the system were established, additional hardware was integrated so as to respond to the needs and develop individual solutions. Some of the capabilities that must be ensured are: the communication with the vehicle, the interaction with the vehicle's actuators and the recalling of information about the vehicle’s attitude using suitable sensors. In the end, all these individual components were integrated in a Command System, creating two different approaches to handle with the vehicle and include them in the Global System. A Simulink model capable of integrating the control blocks, to be produced by the students, was created, to allow for interaction with the vehicle and its sub-systems.

This work results in two fully usable independent architectures, ready to be replicated and used by students in practical Control Classes.

Keywords: Integrated Architecture, Robotic Arena, GPS-denied Environments, Remote Communication, RC Vehicles.
## Contents

Acknowledgments ................................................................. v
Resumo ................................................................. vii
Abstract ................................................................. ix
Contents ................................................................. xiii
List of Tables ................................................................. xv
List of Figures ................................................................. xvii
Nomenclature ................................................................. xix
Glossary ................................................................. xxi

1 Introduction ................................................................. 1
  1.1 Motivation ................................................................. 1
  1.2 Scope and Objectives ..................................................... 2
  1.3 Innovation and Contributions ........................................... 3
  1.4 Document Outline ....................................................... 4

2 State of the Art ................................................................. 5
  2.1 Robotics Investigation under Localization Arenas ...................... 5
  2.2 Previous Work ............................................................... 7
    2.2.1 The Taranis X9D Plus RF Transceiver ................................. 7
    2.2.2 System Operation ..................................................... 8

3 Architecture Development .................................................. 11
  3.1 Overall vision ............................................................... 11
  3.2 Hardware Presentation ................................................... 12
    3.2.1 The RC model .......................................................... 12
    3.2.2 Arduino Uno .............................................................. 12
    3.2.3 Raspberry Pi 3 Model B ................................................ 13
  3.3 Original Setup and Actuators ............................................ 14
  3.4 Motors Actuation .......................................................... 17
    3.4.1 Generating Hardware PWM waves in Arduino Uno .................. 17
    3.4.2 Generating Hardware PWM waves in Raspberry Pi ................. 19
  3.5 Communication with the Vehicle ....................................... 19
A User Manual

A.1 Arduino Based Solution
   A.1.1 List of Material
   A.1.2 Hardware Setup
   A.1.3 Hardware Configuration
   A.1.4 Implemented Code
   A.1.5 Mounting the System in the Vehicle
   A.1.6 Operation Verification

A.2 Raspberry Pi Based Solution
   A.2.1 List of Material
   A.2.2 Hardware Setup
   A.2.3 Hardware Configuration
   A.2.4 Mounting the System in the Vehicle
   A.2.5 Implemented Code
   A.2.6 Operation Verification

B Implementation Issues

B.1 PWM Generation
   B.1.1 PWM and Arduino Uno
   B.1.2 PWM and Raspberry Pi
   B.1.3 PWM initial values to Arming the Actuators

B.2 UDP Communication
   B.2.1 Arduino Uno
   B.2.2 Raspberry Pi

B.3 Simulink scaleChange

B.4 IMU Operation
   B.4.1 Arduino Uno
   B.4.2 Raspberry Pi
List of Tables

3.1 PWM signal's parameters, no actuation required. ................................ 15
3.2 PWM signals’ parameters. ................................................................. 17
3.3 Comparison between the original PWM wave, and the Arduino Uno emulated PWM wave. 18
3.4 Comparison between the original PWM wave, and the Raspberry Pi emulated PWM wave. 19
3.5 UDP communication test in Arduino Uno. ............................................ 22
3.6 UDP communication test in Raspberry Pi. ......................................... 23
3.7 Different PWM ranges comparison. ..................................................... 25
4.1 IMU: sensors’ Default Measurement Ranges. ....................................... 38
4.2 IMU: sensors’ Used Measurement Ranges. ........................................... 40
6.1 Comparison of the work frequencies of both solutions. ......................... 55
6.2 Comparison of the running times for each solution. ............................... 56
B.1 Comparison between the original PWM wave, and the Arduino Uno emulated PWM wave
    using analogWrite( ) function. .......................................................... 77
B.2 Comparison between the original PWM wave, and the Arduino Uno emulated PWM wave
    using delay( ) function. ................................................................. 78
B.3 Comparison between the original PWM wave, and the Arduino Uno emulated PWM wave
    using TimerOne library. ................................................................. 79
B.4 Different PWM ranges comparison. ..................................................... 82
B.5 I²C address map used to initiate the Wire Transmission. ....................... 83
# List of Figures

1.1 Robotic Arena: a new concept ......................................................... 2  
2.1 IDMEC's Robotic Arena ................................................................. 6  
2.2 Taranis X9D Plus: operation ......................................................... 7  
2.3 Previous System: Recording Process ................................................ 8  
2.4 Previous System: Replay Process ................................................... 9  

3.1 Overall concept of the developed architecture .................................. 11  
3.2 LaTrax SST from Traxxas .............................................................. 12  
3.3 Arduino Uno. .............................................................................. 13  
3.4 Raspberry Pi 3 Model B. ............................................................... 13  
3.5 Original Vehicle Setup .................................................................. 14  
3.6 Original PWM signal ..................................................................... 15  
3.7 Original PWM signal on the RF receiver’s output. ............................ 16  
3.8 Arduino Uno: hardware PWM ....................................................... 18  
3.9 Hardware PWM Arduino Solution .................................................. 18  
3.10 Raspberry Pi: hardware PWM ...................................................... 19  
3.11 Hardware PWM Raspberry Pi Solution ....................................... 20  
3.12 TinySine Wi-Fi shield for Arduino. ............................................... 21  
3.13 UDP Communication: Arduino Solution ....................................... 22  
3.14 UDP Communication: Raspberry Pi Solution. ............................... 24  
3.15 Simulink environment. ................................................................. 24  
3.16 Keyboard User Interface .............................................................. 26  
3.17 Dashboard User Interface. ............................................................. 26  
3.18 Joystick User Interface. ............................................................... 27  
3.19 User interface of Setup Block form Simulink environment. ............. 27  

4.1 GY-80 IMU .................................................................................. 30  
4.2 GY-80 IMU: Arduino Uno solution. ................................................ 32  
4.3 GY-80 IMU: Raspberry Pi solution. .............................................. 32  
4.4 IMU position .............................................................................. 32  
4.5 Accelerometer’s measurements: **test 1**. ..................................... 34  

xvii
4.6 Magnetometer’s measurements: **test 1**. ................................................. 34
4.7 Magnetometer’s measurements: **test 2**. ................................................. 35
4.8 Perturbation on magnetometer’s measurements due to a metallic object proximity. 36
4.9 Comparison between magnetometer’s measurements from **test 1** and **test 2**. 36
4.10 Gyroscope’s measurements: **test 1**. ......................................................... 37
4.11 Gyroscope’s measurements: **test 2**. ......................................................... 38
4.14 Effect of the motors in accelerometer’s measurements. ................................. 42
4.15 Effect of the motors in gyroscope’s measurements. ....................................... 42
4.16 Effect of the motors in magnetometer’s measurements. ................................. 43

5.1 Removing the RF receiver form the original car setup. ................................. 45
5.2 Electrical Schematic for Arduino Based Solution. ......................................... 46
5.3 Arduino Based Solution implementation diagram. ....................................... 47
5.4 Arduino Based Solution final setup. ............................................................ 48
5.5 Insertion of the Controller in the Loop, using the Simulink environment. .......... 49
5.6 Electrical Schematic for Raspberry Pi Based Solution. ................................ 50
5.7 Raspberry Pi Based Solution implementation diagram. ................................... 51
5.8 Raspberry Pi Based Solution final setup. ..................................................... 52
5.9 Insertion of the Controller in the Loop, without the Simulink environment in the Loop. 53

6.1 Size comparison of the Hardware Blocks of both solutions. .......................... 58

A.1 Wire diagram showing the connections and global setup for Arduino solution. .... 66
A.2 **WiFiBee** Led States. .................................................................................... 69
A.3 Raspberry Pi3’s pin map. ................................................................................ 70
A.4 Wire diagram showing the connections and global setup for Raspberry Pi solution. .... 71

B.1 Comparison between the original PWM wave, and the Arduino Uno emulated PWM wave using **analogWrite** ( ) function. ................................. 77
B.2 Comparison between the original PWM wave, and the Arduino Uno emulated PWM wave using **delay** ( ) function. ................................. 78
B.3 Comparison between the original PWM wave, and the Arduino Uno emulated PWM wave using **TimerOne library**. ......................................... 79
B.4 Line equation used to convert the two input scales for the Raspberry Pi solution. .... 83
Nomenclature

\( g \) gravitational acceleration: \( 1g = 9.8 m/s^2 \)
Glossary

ADC  Analog Digital Converter.

ADIS  Arquitectura Distribuida Para Integração Sensorial.

CS  Command System.

DHCP  Dynamic Host Configuration Protocol.

DOF  Degrees Of Freedom.

ESC  Electronic Speed Controller.

GPIO  General Purpose Input Outputs.

GPS  Global Positioning System.

IDE  Integrated Development Environment.

IDMEC  Instituto de Engenharia Mecânica.

IMU  Inertial Measurement Unit.

IST  Instituto Superior Técnico.

LsB  Less Significant Bit.

MsB  Most Significant Bit.

OS  Operational System.

PPM  Pulse-Position Modulation.

PWM  Pulse-Width Modulation.

RC  Radio Control.

RF  Radio Frequency.
TCP  Transmission Control Protocol.

UART  Universal Asynchronous Receiver/Transmitter.

UDP  User Datagram Protocol.

USB  Universal Serial Bus.
Chapter 1

Introduction

1.1 Motivation

Nowadays with the scientific progress in the most different fields like control, artificial intelligence and robotics, the importance of multi agent environments becomes a reality that is increasingly compelling.

Multi-agent environments proved to be very versatile and have numerous applications. These kind of agents, can work in two different ways: They can cooperate or they can compete. The cooperation itself can be used to reduce the execution time and to improve the execution performance, but also allows to perform a task that is not possible to be performed by one single agent. On the other side, when used in a competitive way, in this kind of environments all the agents are pitted against each other while trying to accomplish the same goal, whether harming (or not) the competition.

Independently of the agent's goal, an important part of the agent's organization in a populated environment, is the knowledge of their location and the knowledge of the other agent's location. For this, the Global Positioning System (GPS) is the most common solution, because it is a cheap technology, and a very powerful tool. However, in many environments the GPS use is denied, due to several reasons, and alternative solutions are required. Different sophisticated systems are available on the market, based on different kinds of technologies, but require quite a significant monetary investment. These solutions, are generally associated to Localization Arenas and advanced optical motion capture systems, which already have started to be used by the best robotics research canters.

This thesis is integrated in a group effort to create an integrated infrastructure that allows for the localization, command and control of one or multiple vehicles, on the ground or in the air, without making use of GPS, once the system should be used in laboratory, where the GPS signal is not available. The infrastructure is the integration of two independent architectures: A localization system, and a Command System (CS) that will allow interact to the vehicles and control them.

This work uses as localization system the Arquitectura Distribuída Para Integração Sensorial (ADIS) (ongoing MSc Thesis by Tiago Jorge). ADIS is being developed under the Mechatronics Systems Area of Centre of Intelligent Systems in Instituto de Engenharia Mecânica (IDMEC), Instituto Superior Técnico (IST). It will provide 3D localization of each vehicle within the Arena and broadcast this information
via wireless to the Command System. The localization itself works through image processing and is an important part of the final system. The development of the command system is the purpose of this work, and will be explained in this document. Together, these two systems will create a new and custom concept of Robotic Arena. The figure 1.1 helps to understand the overall concept of the global architecture.

Figure 1.1: Global architecture: the interaction between the ADIS system and the Command System.

1.2 Scope and Objectives

The Command System developed is an integrated architecture that allows remotely to command Radio Control (RC) scale vehicles, commercially available on the market. The prerequisites are:

- The interaction between the Control Module and the vehicle should be made through Simulink;
- The vehicles must be integrated on the global architecture without changing their original structure;
- The architecture should be modular and easily replicable;
- The architecture should allow flexibility in the number of vehicles to be commanded;
• The vehicle need to be instrumented with the suitable sensors, to recall information about the vehicle’s dynamic and other proprioceptive inertial sensors.

• The CS must has a working frequency above 30Hz. This corresponds to the ADIS working frequency. The command system should not degrade the overall system performance;

• The final architecture must keep reasonable low costs.

To accomplish these goals, two different approaches were taken throughout this work: The first approach is Arduino based, what leads to a less complex system, easy to implement, but requires processing data in an external machine like a computer. This computer will be used to compute the vehicles’ controller using a Simulink environment.

The second approach, replacing the Arduino by a Raspberry Pi, meaning a bigger calculation capacity. Despite this system being ready to work just like the previous one (using Simulink to implement a controller in a remote computer) this approach makes it possible to increase the vehicle autonomy. There is no need to have a remote computer, because the controller can be implemented and performed on Raspberry Pi (on-board the vehicle). This makes the use of the system a bit more complex, once the controller needs to be implemented in C or Python, which is not as user friendly as the Simulink. However, it paves the way to the use of more complex complementary sensors, in view of improving even more the systems’ autonomy. One possibility to be considered is the use of one on-board camera to detect and avoid obstacles, and replace the ADIS system itself, providing the vehicle self-location and full autonomy.

1.3 Innovation and Contributions

It is expected that the developed architecture is capable of being used by future generations of students in laboratorial control classes, to practice and implement the theoretical knowledge. The system’s multifunctionality allows for the control of a single vehicle to perform a mission like running on a track, drifting on a curve or following the lead vehicle. But the control of formations in a multi-agent environment is also in the scope. The vehicles are able to interact between each other and with the aerial vehicles too. In the edge, the usability is only limited by the user’s imagination.

The fact of using commercially available vehicles provides easier and cheaper replication of the system, which is an ideal feature to a pedagogical instrument. It also allows for the easy replacement of any damaged vehicle’s component.

The ultimate motivation of this work is to produce an innovating, powerful and versatile teaching tool, that permits the accomplishment of serious practical work, and develop new control applications, although at the same time it should be pleasant for students to use.
1.4 Document Outline

This document discusses all the important aspects about the development of the two architectures that are the final purpose of this work.

In Chapter 2, the architecture that inspired the overall concept that underlies this work will be presented. Next, in Chapter 3, a survey of all the aspects that need to be integrated to achieve the two final architectures will be performed. For each specific need, a practical solution will be developed in each platform: Arduino Based and Raspberry Pi Based. All the explored solutions will be analysed in depth in this chapter.

Chapter 4 is addressed to the vehicle instrumentation, where the choice of the used sensors is justified and the sensors’ data is validated. This chapter is also used to refute some incongruences in the sensors’ specifications that are present in the data-sheets provided by the manufacturers. Later, in Chapter 5, the integration of all individual solutions will be explained. Several aspects of the Arduino Based Architecture and the Raspberry Pi Based Architecture are presented here: the required hardware and its setup, the logical implementation, the overall operation, and the users’ interaction. This chapter helps looking at the developed systems explaining each architectures’ node and the way how each node contributes to the global systems’ operation. Chapter 6 provides a comparison between the two different developed architectures and a discussion about the strengths and weaknesses of each approach. Some tests will be performed to validate (or not) all the options taken. It is in this chapter that the accomplished results will be discussed.

The last chapter (Chapter 7) will present the conclusions and some future work will be suggested that will improve the quality of the implemented solutions or even overcome them. Finally, some of the potential applications of the developed systems will be listed, providing some ideas for hypothetical laboratorial works that use the produced architectures.

A user manual is available and some implementation aspects are discussed in Appendices A and B, respectively.
Chapter 2

State of the Art

2.1 Robotics Investigation under Localization Arenas

To allow for the development and the demonstrations of most advanced robotic capabilities, many of the research robotic centres around the world have been architecting Robotic Arenas. This kind of resource has a large application in robotics and the robots are used under these arenas to try to accomplish the most diversified type of capabilities. The following list gives some examples of the diverse applications of land robots and air robots in the most different fields:

- robots to autonomous navigation;
- robots to handle objects;
- swarm of robots;
- humanoid robots;
- robots in sport environments;
- robots in emergency environments;
- robots in rescue environments;
- robot trying to imitate animal behaviour;
- among others.

An arena is an indoor environment that makes it possible to collect a large amount of information about the robots under the infrastructure. The most common approach is based on the use of image processing to detect specific markers on the robots. The images are generally acquired by specialized high-resolution video cameras. The marks installed on the object to track, can be passives, or actives. The passive marks are made by retro-reflective materials, and requires from the system the emission of radiation to be reflected. The active markers, are in their generality light-emitting diodes (LEDs), which require a power consumption. It can, or not, represent a disadvantage, depending on the robot itself.
The optical motion capture systems see the markers strategically placed on the robot, and by triangulation can determine the 3D position of each mark in space, with an elevated accuracy. As so, systems like these are powerful tools to the research centres.

There is already an industry dedicated to the commercialization of these integrated solutions. Few of the commercially available systems are Qualisys\(^1\) and Vicon\(^2\), but there are others. Some of the most renowned research institutions, already use system like these. It is possible to name, for example, the Vijay Kumar Lab\(^3\) (University of Pennsylvania), the Autonomous Robots Arena\(^4\) (University of Nevada), and the Flying Machine Arena\(^5\) (ETH Zurich), among many others. The IST counts with a Localization Arena (figure 2.1), located in the Mechatronics Systems Area and under the supervision of the Centre of Intelligent Systems of IDMEC.

![Figure 2.1: IDMEC's Robotic Arena, using a Qualisys motion capture system.](image)

Despite the quality of these systems and the technological advances they represent these kind of scientific tools have some important limitations. By quoting Shiratori et al. [1]: “Optical systems require indoor setups that typically cost between tens and hundreds of thousands of dollars”. Other limitations can be identified, like the lack of portability of the systems, their intrusiveness and the fact that it requires an indoor use, where the space is usually limited. In this scope, it is an interesting to study ways to overcome these limitations.

\(^1\)http://www.qualisys.com/
\(^2\)https://www.vicon.com/
\(^3\)http://www.kumarrobotics.org/
\(^4\)http://www.kostasalexis.com/autonomous-robots-arena.html
\(^5\)http://flyingmachinearena.org/
2.2 Previous Work

Previous work has been done by Carvalho et. al [2]. A part of the referenced work, was to develop a system that recorded on a computer the command signals of a Radio Frequency (RF) transceiver (that commands a RC car) and replayed it to reproduce the same kind of vehicle behaviour. Once the data was stored on a Matlab structure, it was possible to replay them over and over again. However, the initial conditions of the vehicle (when the replay starts) had direct influence on the vehicle’s motions. This architecture requires, besides the car, a PC running Simulink, a Arduino running a Pulse-Position Modulation (PPM) generator script, and a RF system (par of transceiver and receiver). To understand the whole system, it is important to know how the used transceiver works – the Taranis X9D Plus from the FrSky manufactory [3].

2.2.1 The Taranis X9D Plus RF Transceiver

The used transceiver is composed by two independent modules: the Controller Module and the RF Internal Module. The Controller Module is all that can be actuated by the user, like sticks, buttons, sliders and switches. In the case of this particular transceiver, the Controller Module is equipped with a mini Universal Serial Bus (USB) port that allows the connection of the module to a PC and intercept the module’s output signal. The RF Module is the one responsible to the RF communication between the transceiver and the RF receptor. It was connected do the transceiver antenna. As input, this module receives the Controller’s Module output that is a PPM signal. This transceiver’s RF module have one particularity: a Standard JR-type 3.5 mm jack that is capable of receiving up to 8 channels of input, and sending up to 16 channels of output. Thanks to that, it is possible to communicate with the RF Module without having to interact with the Controller Module. The figure 2.2 provide a visual explanation about the Taranis X9D Plus operation.

![Figure 2.2: Operation of the Taranis X9D Plus.](image-url)
2.2.2 System Operation

The Recording Process

By using the mini USB port, it is possible to get in real time a PPM signal’s value: the Controller’s module output signal, that changes when the analogue sticks are moved. The values obtained in a Simulink environment through a Joystick Input blocks from the Simulink 3D Animation Library, are stored in a MATLAB structure. This structure stores the information about the controller’s stick moves over time. During this process the PPM signal is sent to RF module too, so the RF receiver, on-board the RC car is able to receive the RF signal. As so, the vehicle is on the move. Because the vehicle used is a car, just two actuators are needed to control it (throttle and direction).

![Recording Process]

After performing a record of a sequence of commands, it is possible to replay it.

The Replay Process

In a separated Simulink model and later in the time, the values stored in the Matlab structure are sent to an Arduino UNO, by serial communication. Inside the Arduino runs a PPM generator script, so the Arduino emulates a PPM signals whose data has been recorded from the Controller Module actuators. This PPM signal is given by the Arduino as output through two digital pins. This same signal in given as input to the Taranis RF module through the JR-jack. These signals are recognized by the RF module as stick actuations, which generate a corresponding RF signal. This signal is broadcasted and received by any compatible RF receiver in range. On board the vehicle, one commercial RF receivers catch the RC communication and actuates the car’s motors.

Conceptual Limitations

This kind of approach has several limitations. The RF system is expensive and the architecture requires one RF system for each vehicle to use. It makes the replication economically difficult.

The record/replay process has no utility for control purposes. Once the movement of the vehicle can be replayed, it is possible to modify the recorded control signals, but there is no way to ensure real time
control: the replay process occurs always later in time.

The setup itself is quite intricate, and no information is providing about the car’s attitude.

However, it is fair to say that this system is the one that led to the creation of the overall concept developed in this dissertation.
Chapter 3

Architecture Development

3.1 Overall vision

The purpose of this work is to develop an architecture that allows to remotely control a vehicle. This architecture must be able to be integrated in a more global system, presented in figure 1.1. The general concept to being developed can be found in the figure 3.1.

![Overall concept of the developed architecture.](image)

It is easy to understand that the intended architecture requires an integration of several subsystems: the communication between the car and the computer must be established, hardware must be programmed to interact with the vehicles’ actuators and to deal with the flux of information communicated (to and from the vehicle) and finally, the vehicle needs to be instrumented with sensors to obtain relevant information about its attitude.

In the subsequent sections, all the individual solutions for each node of the architecture will be explained along with the reasons that motivate these choices.

As was referred in the work motivation, two different approaches were taken to achieve different system capabilities. These two separate approaches are based in two distinct kinds of hardware. The first approach is Arduino based and uses a UNO board. The second one is based in a Raspberry Pi and uses the Pi 3 model B.

The two different approaches taken will be differentiated and justified, highlighting the distinctive aspects of each individual solution. Chapter 5 will presents the integration of all the subsystems, establishing the connections between all the nodes of the developed architecture.
3.2 Hardware Presentation

3.2.1 The RC model

The model chosen to integrate the architected system was the LaTrax SST, built by Traxxas [4]. This vehicle is represented on figure 3.2. It is a 1/18 scale model of a 4WD SST race truck. Equipped with independent suspension, and 4-wheel drive, the SST has an electrical motor powered by 6-cell 2/3 A NiMH rechargeable battery. The steering of this model is secured by a servo motor, and the throttle is provided by a brushless motor, commanded by an Electronic Speed Controller (ESC).

Capable of reaching a top speed of 12 meters per second (40 feet per second according to the manufacturer), and to perform aggressive accelerations, this car is a very good model to accomplish some of the more complex sequence of moves, like drifts, spins and j-turns.

The motors can be pre-programmed to select one of the three different performance modes that can limit the motor's full power (forward and reverse) in different ways. The electrical motor integrates a brake system that is a very useful feature.

The high-quality construction materials, and the robustness of the structure makes this model very resistant to potential impacts and crashes. The modularity of the LaTrax SST ensures the possibility of replacing any damaged part, without a loss of the whole vehicle.

The simplicity of this car's design allows the easy instrumentation of the vehicle, adding some hardware, and changing the location of some original components. Despite all these features, the cost is very reasonable when compared with similar models.

For all of this, the LaTrax SST (figure 3.2) is a great model to use in a laboratory and perform the required mission.

![LaTrax SST from Traxxas](https://latrax.com/products/sst)

3.2.2 Arduino Uno

Arduino UNO, presented in figure 3.3, is a widely used prototyping platform, based in the free hardware concept. Equipped with an ATMega328 processor and a set of digital and analogue input/output pins, the Arduino UNO board provides the user with the possibility to assemble different kinds of hardware pieces. The Uno is also equipped with a serial communication interface and an USB port. To program the

---

1[figure from: https://latrax.com/products/sst]
micro-controller, the Arduino provides the Arduino Integrated Development Environment (IDE). It uses a programming language named Processing, which is essentially based on C and C++. The board’s USB interface can be used to load the code compiled in the Arduino IDE, or to power the Arduino. All the pins are identified on the board, which makes it as user friendly as possible.

Arduino Uno board is low-cost equipment, that allows a quick and easy prototyping work, accessible to everyone, in spite of the software or hardware knowledge. The Arduino plataforms are plug and play tools and everything that is required to use them is to install the IDE. The .exe file is provided in the Arduino official website \(^2\).

![Arduino Uno](image)

Figure 3.3: Arduino Uno.

### 3.2.3 Raspberry Pi 3 Model B

The Raspberry Pi is a low-cost computer with a size that fits on the palm of a hand. Developed to allow the teaching of computer science in undeveloped countries. It is capable to perform any task that can be done by a regular computer. The Pi 3 model is featured with a 1.2GHz 64-bits quad-core ARMv8 CPU, 1Gb RAM, wireless LAN, 4 USB ports, 1 full HDMI port, a Micro SD card slot and 40 General Purpose Input Outputs (GPIO) pins. The Raspberry Pi is capable of supporting multiple operational systems, like Raspbian, Debian, UbuntuMATE and Windows 10 IoT Core. A complete list of all official solutions can be founded in the Raspberry Pi official web site \(^3\). In Appendix A, some guidelines are provided in order to help the user to get the Raspberry Pi ready to use. This is a tool with much more processing capability than Arduino, with the performance of a regular computer. It also provides a system of input/output pins, however not so easy to access as in the Arduino platforms.

![Raspberry Pi 3 Model B](image)

Figure 3.4: Raspberry Pi 3 Model B.

---

\(^2\)https://www.arduino.cc  
\(^3\)https://www.raspberrypi.org
3.3 Original Setup and Actuators

To be possible to remotely control the car, it is necessary to know how its actuators works. The first thing that is important to understand is how the original setup is conceived.

The vehicle is sold with a proprietary RF system composed by a remote transceiver and a receiver placed on the car. A servo direction motor is directly connected to the RF receiver. The brushless power motor connects only to its ESC. The ESC itself is powered by the car battery, and connects to the RF receiver too. The RF receiver is powered by the ESC connector and powers the steering servo. To simplify the understanding, a schematic is provided in figure 3.5.

![Figure 3.5: Original Vehicle Setup.](image)

The connections with the RF receiver are made by cables with three wires. These three wires have a colour code that help to understand their function.

- The black wire is the ground;
- The red wire is the voltage (ESC: 5V output; Servo: 5V input);
- The white wire (dashed wire in figure 3.5) is the control signal.

Once the RF receiver's output is directly connected to the actuator's white wires, if these output signals are known, it is possible to emulate them to actuate the motors.

According to the manufacturer, the control pin's output from RF receivers provide a Pulse-Width Modulation (PWM) signal, which contains the information transmitted from the transceiver by Radio Frequency. Each channel outputs a PWM wave to actuate one different motor.

An oscilloscope allows the visualization of the output signal of one of the receiver's channels, when there is no actuation required on radio transceiver (figure 3.6).
Using the oscilloscope tools, it is possible to characterize the PWM wave. Some important parameters were determined with the PicoScope software [5] and registered on table 3.1.

Table 3.1: PWM signal’s parameters, no actuation required.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency</td>
<td>101.6 Hz</td>
</tr>
<tr>
<td>Duty Cycle</td>
<td>15 %</td>
</tr>
<tr>
<td>High Pulse Width</td>
<td>1487 μs</td>
</tr>
</tbody>
</table>

The period of the wave can be directly determined by the inverse of the frequency. For this wave, the period is 9840 μs.

The vertical dashed bars on the figure marks the limits of a maximum and minimum High Pulse Width when 100% actuation is required and when 100% reverse actuation is required, in this specific channel. The response waves for these two situations are available in figure 3.7(a) and figure 3.7(b). In both cases, two PWM waves are visible. The wave presented in figure 3.6 is taken as reference (no actuation required) in both cases and is represented above the wave where the actuation is visible, for comparison purposes.
More measurements were performed with the PicoScope software, to obtain the PWM wave parameters of each situation. The results can be found in table 3.2

After studying the output of the two receiver channels, it is clear that the PWM signal presents the same behaviour in both. The actuation of the vehicle can be done by generating two different PWM signals, one for each actuator. Once the receiver’s outputs are totally known, it is possible to emulate them.

Figure 3.7: Original PWM signal on the RF receiver’s output.
Table 3.2: PWM signals’ parameters.

<table>
<thead>
<tr>
<th>Actuation Required</th>
<th>Minimum value</th>
<th>Mean value</th>
<th>Mean value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Figure 3.7(b)</td>
<td>Figure 3.6</td>
<td>Figure 3.7(a)</td>
</tr>
<tr>
<td>100% reverse</td>
<td>993 $\mu$s</td>
<td>1487 $\mu$s</td>
<td>1979 $\mu$s</td>
</tr>
</tbody>
</table>

3.4 Motors Actuation

As discussed in section 2.2.2, the RF system cannot be integrated in the architecture. As such, it is important to be able to reproduce the actuation signals without using the RF receiver. It is here that the two architecture approaches start to diverge.

3.4.1 Generating Hardware PWM waves in Arduino Uno

It is quite simple to generate a PWM wave in an Arduino Uno. The Arduino IDE provides, in its libraries, a function that allows us to generate PWM waves in digital pins ports 3, 5, 6, 9, 10 and 11. However, because it does not provide any control over frequency, it is not useful to our purposes. Even the available tools that allow for the control of the signal’s frequency and duty cycle cannot be used, because of some hardware limitations. For more information about it, please see appendix B.1.1 about implementation issues.

To better understand the options taken to implement a PWM generator, is important to know that the ATMEGA328 processor integrated in the Uno boards has three different hardware timers. Each one of these timers has two compare registers that allow for the control of the PWM width on the timer’s outputs. For the same timer, the two outputs will normally have the same frequency but different duty cycles are supported. To guarantee that the PWM signals will not suffer any interferences, one timer must be dedicated only to the generation of PWM signals (hardware PWM generation).

The timers’ frequencies are a submultiple of the system clock frequency (16MHz) and can be set by manipulating the chips’ timer registry directly. Beyond the ability to change the frequency, it allows for the choice of some different wave modes and generates interruptions and overflows. This method provides a lot more control over the PWM wave generation. To get more information about that, please see Appendix B.1.1.

TimerOne is a library that provides a collection of routines that allow for a simple configuration of Timer1. Using the pwm() function of this library it is possible to generate a hardware PWM wave with a specific frequency and a specific duty cycle. However, this library is not provided with the original Arduino IDE. To know how to get and install the library, please check Appendix B.1.1.

The Figure 3.8 allows for the comparison of the PWM wave generated with the TimerOne library, and the original wave obtained from the RF receiver. The waves parameters are compared in table 3.3.

The resultant emulated wave is very close to the original one. Despite the difference in the output
Figure 3.8: Using TimerOne library to generate a hardware PWM signal.

Table 3.3: Comparison between the original PWM wave, and the Arduino Uno emulated PWM wave.

<table>
<thead>
<tr>
<th></th>
<th>Arduino signal (red wave)</th>
<th>Original signal (blue wave)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency</td>
<td>99.96 Hz</td>
<td>101.6 Hz</td>
</tr>
<tr>
<td>Cycle Time</td>
<td>10000 μs</td>
<td>9846 μs</td>
</tr>
<tr>
<td>Duty Cycle</td>
<td>15.16%</td>
<td>15.21%</td>
</tr>
<tr>
<td>High Pulse Width</td>
<td>1526 μs</td>
<td>1505 μs</td>
</tr>
</tbody>
</table>

voltage, it is important to recall that the most relevant parameter to allow the motors’ actuation is the wave duty cycle and this is successfully replicated.

By programming directly the Timer1 registers, the TimerOne library provides a way to generate hardware PWM waves that differ from regular PWM waves due to the fact that they are generated by a dedicated hardware and not by software. See appendix B.1 for a comparison of some types of PWM waves.

In the laboratory, experimental tests show that the correct actuation of the vehicle’s motors is possible using PWM waves emulated in this way, so this is the dedicated solution that will be used to actuate the motors in Arduino’s approach (figure 3.9). However, this method has an important limitation: the TimerOne library only works with digital pins 9 and 10, in the Uno board. This means that only two PWM waves can be emulated, so the system is limited to vehicles which only have two actuators.

Figure 3.9: Hardware PWM Arduino Solution.
3.4.2 Generating Hardware PWM waves in Raspberry Pi

Like the Arduino, the Raspberry Pi provides some different methodologies to generate PWM waves on output pins. Thanks to the acquired knowledge about Arduino limitations, the first option taken was to generate hardware PWM waves instead of software PWM waves. For this, the use of the *pigpio* library is required. This is an open source library written in the C programming language that relies on a Python module that allows for the control of the GPIO pins. Therefore, for the sake convenience, the implementation of this solution will be coded in Python. To know how to get this library and install it, please see appendix A.2. The current section only focuses in the result of the use of the *pigpio* library.

Once more, the RF receiver output was analysed on an oscilloscope and its parameters were used to emulate a similar wave. The comparison of the two waves is represented in the figure 3.10. The parameters obtained with the PicoScope tools are summarized on the table 3.4.

![Figure 3.10: Using pigpio library to generate a hardware PWM signal.](image)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Arduino signal (red wave)</th>
<th>Original signal (blue wave)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency</td>
<td>101 Hz</td>
<td>101.5 Hz</td>
</tr>
<tr>
<td>Cycle Time</td>
<td>9900 µs</td>
<td>9852 µs</td>
</tr>
<tr>
<td>Duty Cycle</td>
<td>15.76%</td>
<td>15.65%</td>
</tr>
<tr>
<td>High Pulse Width</td>
<td>1563 µs</td>
<td>1541 µs</td>
</tr>
</tbody>
</table>

The emulated PWM wave is very close to the original one. In practical tests, the emulated signals were given as input to the actuators and it was proven that it is possible to control the vehicle with this implementation. As so, this solution was adopted and represented in figure 3.11.

3.5 Communication with the Vehicle

As an RC car, the LaTrax SST scale model is equipped with a RF system that guarantees the communication between the user and the vehicle.

Despite this is being a very reliable technology, it is not usable on this architecture due to two main reasons: firstly, the use of a computer to control the vehicle is required, and the interaction of a radio transceiver with a PC is expensive and requires a complex setup. As the desired architecture, must be cheap and easy to use, those are very important limitations. On the other hand, once the ADIS
infrastructure is being designed to use a Wi-Fi network to broadcast the information about the vehicle position, it seems a logical move to use and potentiate this available resource.

To perform vehicle control through the wireless communication, it should be guaranteed that the frequency of communication is good enough to allow this. The communication between the vehicle and the remote PC will be used to transmit the command signals to the car. Additionally, it will also be used to transmit the on-board sensors’ data from the vehicle to the PC.

The communication itself is a very important subsystem of the final architecture because it represents the bottleneck of the system's work frequency. In this way, it is extremely important to choose the proper protocol for the transport layer of the IP network. It must be assured that the communication frequency is above of the frequency of the ADIS localization system (30Hz).

Since the communication speed is the most valued feature, the implemented protocol will be the User Datagram Protocol (UDP), against the typically used Transmission Control Protocol (TCP). Despite the TCP’s higher reliability due to its ability to prevent the loss of transmission data and ensuring the correct order of packet reception, it does not allow for broadcast transmissions. TCP is also quite slower than the UDP protocol. However, it is important to understand that the speed performance of UDP is achieved at the expense of not being as reliable as TCP.

### 3.5.1 UDP Communication in Arduino

The Arduino hardware has built-in support for Serial Communication via USB connection. The pins 0 (RX) and 1 (TX) also support serial communication, but they are just an alternative interface to the USB connection. This serial port is allocated to communicate with the computer and the Arduino Serial Monitor. This native serial support is based on a hardware component named Universal Asynchronous Receiver/Transmitter (UART). The UART allows the Atmega processor to receive information through serial communication, even while it is working on different tasks. In this case, the information is stored on a 64-byte serial buffer. No Wi-Fi communication is supported, however, it is possible to get many kinds of add-on hardware to extend the original Arduino functionalities. In this case, to establish UDP communication with an Arduino Uno board, a Wi-Fi shield is required. Between all the different types of Wi-Fi shields in the market, the chosen one was the TinySine WIFI shield. In Appendix A.1 it is possible
to find more information about this hardware component, its setup and how to configure it.

Figure 3.12: TinySine Wi-Fi shield for Arduino.

This shield is a combination of a TinySine Bee shield (blue component) and a WiFiBee module built by TinySine (red component) of figure 3.12. The WiFiBee module provides a way to communicate to a local network by IP address, and the TinySine Bee shield establishes the connection between the WiFiBee module and Arduino Uno. Because the UART module can not be used to establish this connection, there is a need to use the SoftwareSerial library.

The SoftwareSerial library, provided with the Arduino IDE, allows the serial communication between other pairs of Arduino digital pins (other than pins 0 and 1). This library makes this possible by using software to replicate the UART behaviour. The maximum speed supported by the software serial ports is 115200 bps.

It is possible to have multiple software serial ports, although only one at a time can receive data. The full documentation of this library can be found in [6].

To check the viability of this equipment, some tests have to be performed. The communication frequency and the percentage of received wrong messages are the most important parameters to test.

To perform this test, a string was sent with the following format:

```
String = < 89_chars > + < # > + < Number_of_message > + < \n >
```

This format was chosen to allow testing a significant number of characters (at least 92). The messages are numbered in order to understand which are the messages that are not properly communicated. Because they are numbered, the messages have different lengths. As such, there is a need to use the < \n > char as a terminator character. Arduino sends a message with this format at 10 milliseconds intervals. This means that it is possible to send messages at a 100Hz frequency, theoretically. In practice, this doesn’t happen because running the code takes time and so does the communication itself, delaying the procedure.

On the remote computer, a MATLAB script runs during a certain amount of time. In this time, the script receives every message sent from the Arduino. At the end, the frequency is given by the total

---

4Figure from: http://www.tinyosshop.com
number of messages received, divided by the listening time (in seconds). Subtracting the number of the first message and the number of the last one, and comparing the result with the total number of received messages, it is possible to count the number of lost messages. This test was being performed with different time intervals (10 seconds, 1 minute, 5 minutes and 10 minutes) and at two different baud rates (the default 9600 bps and the maximum 115200 bps). The results are presented in the table 3.5.

Table 3.5: UDP communication test in Arduino Uno.

<table>
<thead>
<tr>
<th></th>
<th>9600 Baud rate</th>
<th>115200 Baud rate</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Frequency</td>
<td>Percentage Lost Messages</td>
</tr>
<tr>
<td>10 seconds</td>
<td>9.3 Hz</td>
<td>0%</td>
</tr>
<tr>
<td>1 minute</td>
<td>9.13Hz</td>
<td>0.55%</td>
</tr>
<tr>
<td>5 minute</td>
<td>9.01 Hz</td>
<td>0.74%</td>
</tr>
<tr>
<td>10 minute</td>
<td>9.08 Hz</td>
<td>0.11%</td>
</tr>
</tbody>
</table>

With the increasing of baud rate to the maximum supported value, it is possible to increase the transmission frequency by 5.7 times, relative to the results obtained with the 9600 default baud rate. In every test the percentage of lost messages stays below 1%, meaning that the streaming implemented is reliable.

The representation of this solutions’ setup can be consulted in figure 3.13.

![UDP Communication: Arduino Solution.](image)

3.5.2 UDP Communication in Raspberry Pi

Because the Raspberry Pi3 hardware already has Wi-Fi technology incorporated, no supplementary hardware pieces are needed to implement UDP communication. Once more, and to be coherent with the previous sections, the UDP communication implementation was made in the Python programming language.
Table 3.6: UDP communication test in Raspberry Pi.

<table>
<thead>
<tr>
<th>Delay between message: 10 ms</th>
<th>Delay between message: 20 ms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum theoretical frequency: 100Hz</td>
<td>Maximum theoretical frequency: 50Hz</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Frequency Percentage Lost Messages</th>
<th>Frequency Percentage Lost Messages</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 seconds 97.5 Hz 0.47%</td>
<td>10 seconds 48.8 Hz 1.02%</td>
</tr>
<tr>
<td>1 minute 97.5 Hz 1.66%</td>
<td>1 minute 48.9 Hz 0.99%</td>
</tr>
<tr>
<td>5 minute 95.6 Hz 2.44%</td>
<td>5 minute 48.7 Hz 1.49%</td>
</tr>
<tr>
<td>10 minute 93.9 Hz 4.38%</td>
<td>10 minute 48.2 Hz 2.61%</td>
</tr>
</tbody>
</table>

To be able to implement this kind of protocol, the socket library must be used. This is a regular Python library, installed by default, so no additional installations are needed.

To send and receive messages, two different sockets must be created, and the reception socket must be bind with the Raspberry Pi’s IP and the local receiving port.

The socket library also provides the proper functions to read and write in the socket. To make easy the use of these functions, a `UDP_communication.py` library was written in this work.

The same procedure that was used to test Arduino UDP communication was applied to Raspberry Pi’s UDP communication in similar conditions. In this case, there is no way to change the baud rate, which makes sense as there is no external hardware to communicate with (Wi-Fi hardware integrated on the Pi).

One more time the messages were sent at 10 milliseconds intervals, which means a maximum theoretical frequency of 100Hz. The table 3.6 shows the results. A second test was performed, increasing the delay between messages to 20 milliseconds, which limits the maximum frequency to 50Hz. These results are also represented in the table 3.6.

In this case, the communication works at much more high frequency (always close to the theoretical value). Unfortunately, this increase in performance has a significant cost in terms of percentage of lost messages. This value increases with communication time and, after 10 minutes communicating, this value is above 4%, for the first test. That is the reason that motivates the existence of the second test. When the maximum frequency is limited, the percentage of lost messages decreases.

For now, it is quite difficult to characterize the quality of this solution, because the time delay between communications is being imposed. When the solution is integrated, the other processes running in the Raspberry Pi, like motors’ actuation and sensors’ activity, will naturally delay the communications. It is known that it is possible to force lower frequencies, close to the 30Hz, that is the minimum value admissible to the system. In this case the percent of lost messages will certainly be lower than the values obtained in the tests. for now, this solution is admissible, and its implementation is represented on figure 3.14.

### 3.6 The Simulink

A remote controller is necessary to control the vehicle at a distance. In this section, the replacement of this handheld remote controller by a remote computer will be discussed. This remote computer is
integrated in the same IP network as the vehicles and must use a Simulink environment to communicate the actuation commands to the on-board hardware solution via UDP.

The Simulink implementation itself will be divided in 3 main blocks (figure 3.15):

- the user interface;
- the setup block;
- the communication block.

3.6.1 The Communication Block

To allow UDP communication, Simulink provides the UDP send block, available within the DSP system toolbox. This block is easy to configure. It requires only the target’s IP address and remote port where the data will be delivered as inputs. Note that it is important to guarantee that the Computer and the vehicle are in the same IP network.

Even though the tool box makes UDP communication in Simulink an easy job, there is an important limitation to consider. The UDP block is only capable to send unsigned 8 bit integers (int8 format). With
Table 3.7: Different PWM ranges comparison.

<table>
<thead>
<tr>
<th>PWM Duty Cycle</th>
<th>0% ... 10% ... 15% ... 20% ... 100%</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Arduino Actuation</strong></td>
<td>0 ... 102 ... 153 ... 205 ... 1023</td>
</tr>
<tr>
<td><strong>Raspberry Pi Actuation</strong></td>
<td>0 ... 100 000 ... 150 000 ... 200 000 ... 1 000 000</td>
</tr>
</tbody>
</table>

8 unsigned bits, the range of representable integers is 0 to 255. It is not a big problem but requires some attention. Alphabet characters and others can be easily sent using an ASCII table to convert char format into int8 format. For now, characters are not required because, to send information about the actuation of the motors (to generate hardware PWM waves), numbers are more appropriate. As such, the commands’ scale must be chosen carefully.

It is known that the duty cycle of a PWM wave is in a 0% to 100% range. However, in section 3.3, it was seen that actuation commands supported by the actuators have a reduced range from 10% to 20% duty cycle. Despite of the fact that it is possible to transmit this range via the Simulink UDP block (because it fits on a 0 to 255 range), its discretization is too big. As a duty cycle of 15% corresponds to the no actuation value, the 10% to 20% range would only allow 5 actuation levels in each direction of actuation, which is a limitation. To overcome this limitation, a new set of ranges was outlined to be able to transmit the information of the motors’ actuation from each approach. This is summarized in table 3.7. More information about this issues, can be consulted in appendix B.1.

\[ \text{input argument of PWM generator function} = 1023 \times \frac{\text{duty cycle [\%]}}{100} \]

\[ \text{input argument of PWM generator function}= \text{duty cycle[\%]} \times 1000 \]

Just by inspecting the table, it is possible to see that in the Arduino implementation the problem is easy to solve. The 102 to 205 range represented on the table is already possible to be communicated via mulink’s UDP send block. Additionally, the discretization provided by this scale gives 50 intervals for each direction of actuation, which is an improvement over the initial range. This scale has the benefit of not needing additional processing in the receptor, which is a welcome feature for the Arduino’s limited micro-controller. On the other hand, in the Raspberry Pi implementation, the input arguments are too large to fit in the allowed transmission range. For this reason, the solution consists in dividing the input argument by 1000. So, the transmitted value will be in a 100 to 200 range. In the Raspberry Pi, these values will need to be multiplied by 1000 before being passed to the hardware PWM generate function as an argument. This scale was chosen because it is similar to the Arduino scale. If the user, by mistake, sends a value scaled in a wrong range to one of the architectures, the result will not be perfect, but will be approximately the same, and no unexpected actuations will occur.

Depending on the fact that commands are being sent to either an Arduino or to a Raspberry Pi, the values must be scaled on a different way. This scaling and the decision about what scale to apply must be done in the Setup Block.
3.6.2 User Interface

Because the scales defined above and available to communicate through Simulink’s UDP block are not necessarily user friendly, it is important to think about what scale should be provided as a user interface.

What kind of input must be required from the user, to give him a perception of which is the result of the requested actuation. To achieve this in an intuitive way, a range from -1 to 1 was chosen. This scale now needs to be properly converted before communication is possible. This will be discussed in appendix B.3.

This section will approach the different ways developed in Simulink to interact with the system.

The first and easier concept consists in the use of two Simulink constant blocks to directly receive the value to be actuated. This block's value can be modified while the simulation is running, so it is a direct way to give the commands to the system. This block is pictured in figure 3.16.

![Figure 3.16: Keyboard User Interface.](image)

A new and more interactive interface was designed to be used by students in Optimal Control classes, where a preliminary version of the system was tested. This new concept improves the user’s perception about the vehicle's dynamics and includes visual elements from Simulink Dashboard, such as sliders, buttons, switches, lamps and gauges. The result is present in figure 3.17.

![Figure 3.17: Dashboard User Interface.](image)

Another potentially useful feature, can be the use a joystick. Any kind of remote control, or another instrument capable to be connected to a PC by USB would be useful. This implementation was made by using the Joystick Input block from Simulink 3D Animation Library (figure 3.18). This kind of user
interface is particularly useful to perform tests that require more aggressive moves. Furthermore, it is a funny way to use the system.

![Joystick User Interface](image)

Figure 3.18: Joystick User Interface.

### 3.6.3 The Setup Block

The setup block is the one responsible for converting the -1 to 1 range from the User Interface to a range capable of being communicated to the UDP Simulink block. The conversion itself has been implemented in a specific block named `scaleChange` and its content will be explained in the appendix B.3.

This setup block must be able to choose what scale to apply, and to define the correct target IP in the UDP block. To make the system easy to use, a `START HERE` script was written. Once this script is running, a selection window will request to user to choose the vehicle to be used. It is done, using the graphical window of figure 3.19(a).

Once the user makes a choice, the target IP and communication Port will change automatically. The scale factor will automatically be defined too. After this, the setup will ask the user what kind of interface desires to use (figure 3.19(b)), and will launch the correct Simulink schematic.

![Setup Block Interface](image)

Figure 3.19: User interface of Setup Block form Simulink environment.

The Simulink based transceiver is a very polyvalent tool that can be integrated in other Simulink Projects. The command values can be directly received from different kinds of control loops, scripts or hardware pieces. It is possible to use it even if the user does not know nothing about coding. It is easy to replicate and easy to configure.
Chapter 4

Vehicle Instrumentation

An important part of the developed architecture is the instrumentation of the vehicle. The ADIS system will provide information about the spatial location of the vehicles, but this information is not enough to know about its dynamic. An object can be in a specific location, but presents different spatial orientations. To complement the ADIS system and complete the knowledge about the vehicle dynamic, the vehicles must be instrumented with on-board sensors.

A Inertial Measurement Unit (IMU) is the proper instrument to collect information about the vehicle attitude. A typical IMU integrates accelerometers, gyroscopes and sometimes magnetometers too.

The accelerometers obtain information about the specific force that actuates over the object. The specific force, also known as g-force is important in the body’s attitude determination.

The gyroscope is used to get information about the angular rate of the vehicle.

The magnetometer provides information the magnetic flux density about the around the sensor, as so, provide information about the position of the vehicle relatively to the Earth’s magnetic field.

The data of all these sensors can be combined to provide a more reliable information about the vehicle’s attitude. The present chapter is about the vehicle instrumentation with a GY-80 IMU.

4.1 GY-80 main specifications

The GY-80 IMU, presented in figure 4.1 is a multi-sensor board with 10 Degrees Of Freedom (DOF). This model has been used because it was already available on the lab. The digital 3-axis accelerometer integrated in GY-80 is the ADXL345. This device can perform measurements in the ±2g, ±4g, ±8g and ±16g ranges up to 11 bits’ resolution with low energy consumption. The L3G4200D is the digital gyroscope with 3 DOF and supports measurements in the ±250 dps, ±500 dps, ±2000 dps scales. The magnetometer used is the HMC5883L model, also with 3-Axis and is also digital. In this case, the supported measurements are in some different ranges, between ±0.88 Ga and ±8.1 Ga.

The additional DOF allocated to a fourth sensor, a BMP080 digital pressure sensor that allows the measurement of pressures and temperatures (for now, this sensor will not be used).

1 Figure from: http://diyspacepk.com/
The GY-80 uses Inter-Integrated Circuit (I2C or I2C), a multi-master serial bus, to configure all the sensors and collect their measurements raw data.

To know how to use and configure the GY-80 IMU, please see Appendix A.

![GY-80 IMU](image)

**Figure 4.1: GY-80 IMU.**

### 4.2 Scale Factor: conversion into physical units

After installing the IMU, there is a need to convert the output data sensors in physical units.

One of the purposes of this work is to produce an integrated architecture that will allow the users to develop and test different kinds of control loops. The sensors’ data are an important parameter to do that, so it is important to guarantee that the users understand exactly what this data means. To convert the raw data of the sensors into physical units there is a need to multiply their output by a scale factor. This value can be found in each sensor data-sheet, but due to some incongruences present in the sensor information provided by the manufacturers, this can be an intricate job.

This scale factor is directly related to the measurement precision and the discretization error and it differs with the measurement ranges. It will be discussed for each sensor and their respective used range. Later in section 4.5, the choice of the appropriate range will be discussed.

#### 4.2.1 ADXL345 Accelerometer

The accelerometer is used in full resolution. For the ±4g range, it means that the output has an 11 bits’ resolution, according to the data-sheet [7].

With 11 bits, it is possible to represent \(2^{11} = 2048\) discretization intervals. Representing a range of length 8 g in 2048 intervals, leads to discretization intervals of \(8/2048 = 3.9mg/Lsb\). This value means that every time that the sensor’s output data increases one bit, the specific force increases 0.0039g. Therefore, \(0.0039g/Lsb\) is the scale factor that allows the conversion of the accelerometer's data output to g units.

The inverse of this value \(2048/8 = 1/0.0039 = 256Lsb/g\), is the sensitivity of the sensor.
4.2.2 HMC5883L Magnetometer

The magnetometer scale used is the one defined by default: ±1.3G.

According to the data-sheet [8], the sensor’s output range is from −2048 to 2047, which means the sensor has a 12 bit resolution (and $2^{12} = 4096$ representation intervals). If so, the sensitivity of this sensor must be $4096/2.6 = 1575 LsB/G$. However, in data-sheet [8], the sensitivity (also named Gain by the manufacturer) is stated to be $1090 LsB/G$. So, the theoretical value and the technical value provided in the data-sheet do not match. In this subject, the datasheet provided by the manufacturer is not very clear, leading to some confusion. The Analog Digital Converter (ADC)’s output has 12 bits, resulting in an output range of −2048 to 2047. However, this range is not used for the magnetometer in the totally.

The sensor’s analogue output range, is converted in the ADC to a digital range. For the specific case of the ±1.3G measure range, the analogue output is converted to a digital value between $-1.3 \times 1090$ and $1.3 \times 1090$, which gives a digital range of −1417 to 1417. As so, the sensitivity described in the datasheet is correct, but the range is poorly explained, and can be misleading.

A $1090 LsB/G$ sensitivity can be inverted to give the scale factor $1/1090 = 0.000917 G/Lsb$. This value is coherent with the value of $0.92 mG/Lsb$ from the data-sheet. This value is the scale factor that converts the magnetometer’s output digital units into Gauss, and represents also the value of the sensor’s discretization interval.

4.2.3 L3G4200D Gyroscope

For the gyroscope, the scale used is ±2000. The sensor’s data-sheet gives the sensibility for this range as 70mdps/digit. However, expectable sensitivity units would be digit/mdps. It is assumed here that a digit is a variation of a bit, so it means the same as a Less Significant Bit (LsB). Once more, the manufacturer data-sheet is not totally clear, because the mentioned sensibility is, in fact, the scale factor of the chosen range, and the sensibility is the inverse of the given value: $1/0.07 = 14.29 digit/dps$.

According to the data-sheet, each output value has a 16-bit resolution, which means a digital range between −32768 and 32768 ($2^{16}$) at the ADC’s output. For the selected measurement range, calculations show that the digital range corresponding to the measurement range is ±28571 (−2000×14.29 and 2000×14.29, following the same logic explained before, in subsection 4.2.2).

However, instead of what happens with the magnetometer, for the gyroscope, the sensor itself does not saturate at the stated value. It means that the sensor keeps measuring beyond the ±2000dps and because the ADC’s digital output scale is not completely used, the sensor can output angular rates values up to ±2294dps. These values are obtained multiplying the full ADC’s range by the scale factor ($\pm 32768 \times 0.07$).

With the purpose of future comparisons, it is useful to do the same calculations in the ±250dps range, for which the scale factor is 8.75mdps/digit. In these conditions the sensor can output values up to

$$± 2^{15} \times 8.75/1000 = ±286.9dps.$$  \hspace{2cm} (4.1)

These out of scale values show up for all three of the gyroscope’s available ranges. This fact is not
mentioned by the manufacturer but was verified by these theoretical calculations and by experimental results. No guarantee is given that the measures performed beyond the specified scales are correct and, as such, all the values beyond the stated ranges should be ignored.

4.3 IMU’s Installation on the Vehicle

In order to perform some tests, there is a need to install the IMU on-board the vehicle. In this case, the procedure is similar to both implementations. It is possible to use the \textit{I2C} protocol to configure the IMU, choosing the appropriate measurement range for each sensor and then getting the measurements. Both Arduino Uno and Raspberry Pi, have dedicated pins to this kind of communication protocol, as so, its implementation do not require none additional hardware, or external libraries. The Integration of GY-80 IMU in each solution are represented in figures 4.2 and figure 4.3.

![Figure 4.2: GY-80 IMU: Arduino Uno solution.](image)

![Figure 4.3: GY-80 IMU: Raspberry Pi solution.](image)

Once the IMU is correctly assembled in the system, its relative position to the car must be known. Figure 4.4 illustrates this.

![Figure 4.4: IMU’s relative position to the vehicle.](image)
4.4 Preliminary data validation

At this stage of the work, all the previous subsystems were already implemented and integrated. It was already possible to remotely control the vehicle using Simulink in a computer to transmit UDP commands through a private network and actuating the motors at distance using an Arduino or a Raspberry Pi. This integration will be explained in detail later in chapter 5, but it plays a very important role in the subsequent subsections, so it must be mentioned here.

The architecture allows the transmission of the measurements back to the PC through the implemented UDP subsystem. The measurement’s timestamps must be transmitted too. In the PC, a dedicated script will receive the data, store and process it. This way it will possible to test and validate the quality of the sensors’ measurements.

To achieve this, two main different tests were performed:

- In test 1, the IMU was attached to the car, placed on a horizontal plane, during 60 seconds.
- In test 2, the car, with the IMU on-board, was asked to perform circular movements on the ground for 20 seconds. This test was recorded in a video, to be used in the validation of the results.

For the duration of the experiments, the raw data of the sensors’ output was stored in a MATLAB workspace. This data was converted to physical units and plotted. The results of these preliminary tests will be discussed on the subsequent subsections.

4.4.1 ADXL345 Accelerometer

To test the accelerometer’s measured values, the car remains immobile during the data acquisition. Two of the three axes of the IMU must be parallel to the ground plane, and one of the axes must be perpendicular to the ground. The collected data must be converted into g. The absolute value of the measurement obtained in the perpendicular axis is expected to be $1\, g$. Additionally, the parallel axes should measure around $0\, g$. The plot of figure 4.5 shows the result of this experiment.

As it can be seen, the results are very close to what was expected. The negative values in the z-axis reveal that the IMU was placed upside down and, in fact, it was. The absolute value of each axes’ fluctuation was calculated and shown to be lower than $50\, mg$.

\[
\begin{bmatrix}
X & Y & Z \\
0.0390 & 0.0351 & 0.0468
\end{bmatrix}\, g
\] (4.2)

A last note about the results is that it is not possible to guarantee with 100% certainty that the IMU is horizontal. Even if the sensor was placed directly on the ground, the ground itself may not be level. This fact helps justifying the residual g values on x and y axes, which should be $0g$. As such, the Z value of the specific force which should be $1g$, is very close, but it is a bit less. This test was repeated by lining up each one of the axis with the direction of gravity, and the result was always the same: The axis perpendicular to the ground plane measures always a value close to $1g$. 

33
4.4.2 HMC5883L Magnetometer

The magnetometer measures the magnetic flux density of a surrounding magnetic field. Because the Gauss unit $G$ is not as intuitive as the g-unit, to understand the measured values, a more elaborated test was needed. First, during one minute and in the conditions of test 1, a set of magnetometer measurements was collected. This data was plotted over time and the result is shown in the figure 4.6.

Because the relative positions of the sensor to the magnetic field do not change (the test was performed in rest), it is expected that all the axes’ measurements remain approximately constant in time. And they did. The absolute value of the fluctuations of the magnetometer’s output values with time are

$$\begin{bmatrix} X & Y & Z \end{bmatrix} = \begin{bmatrix} 0.0074 & 0.0074 & 0.0184 \end{bmatrix} G$$  \hspace{1cm} (4.3)

However, it is not possible to just look at these values and know if they are correct. For this purpose,
a mobile app named *Physics Toolbox Sensor Suite* [9] can provide a precious help. This android application was used to measure the magnetic field in the exactly same place, with the same orientation and in the same units of the sensor. After some time trying to understand the direction of the smartphone’s axes, the obtained results are very close to the plotted one. Still, the possibility to see the measured values in real time, made it clear that this kind of sensor is very sensitive to physical perturbations.

Next, data from test 2 was collected, stored and plotted in figure 4.7.

Because the vehicle has always the same orientation in the z-plane (no inclination) it is expected that the variation of the z-axis’ values will not be significant and the measurements must be approximately constant. If the vehicle moves while on the ground, the most significant measurement variations will occur in the x and y axes.

![Figure 4.7: Magnetometer’s measurements: test 2.](image)

In fact, the z measurements vary much less than the other axes measurements, but still allow for the identification of the circular movement.

On the other hand, the x and y measurements present a sinusoidal behaviour, but not with the same phase. The sensor measures the absolute values of the intensity of the magnetic field in each one of its components. So, if one of these axes are more aligned with the magnetic field than the others, the value measured in this axis will be bigger than the values measured in the other axes. If the vehicle performs circular moves, the measured values will be periodically repeated (one period corresponds to one complete lap).

Sixteen seconds after the test starts, a strange behaviour was registered in the z-axis measurements. In the y-axis and x-axis, a less significant perturbation was registered too. By studying the plot, it is possible to verify that this perturbation happens in the middle of the sixth lap. Analysing a video recording of the experiment, it is possible to see that, at this moment, the vehicle passes very close to a table’s iron leg. To try to understand if the leg can be a cause of the perturbation, an additional test was performed. The mobile application *Physics Toolbox Sensor Suite* was used next to another metal object. The perturbations were registered and are shown in figure 4.8
Test 2 was repeated a few more times farther away from the table and no perturbations were registered. This leads to the conclusion that perturbation seen in figure 4.7 was caused by the presence of the table’s leg. It means that, when the final system will be put to use, the user should be careful about the proximity of the car to this kind of materials. This issue could possibly be solved through the integration of multiple sensors’ information and performing data filtering.

To understand the magnitude of the measurements’ variation, a last graph is shown, where the data of both tests are plotted. Figure 4.9 shows the measurements in y-axis in function of the measurements in x-axis.

Figure 4.9: Comparison between magnetometer’s measurements from test 1 and test 2.
Remember that test 1 has a duration of 60 seconds whilst test 2 has a duration of 20 seconds. This means that five times more data was collected in the first experiment. Even so, in this first set of data, the points are very close to each other, because the relative position of the magnetometer to the magnetic field is approximately constant. On the other hand, in the second set of data it is possible to distinguish the variation of the relative measurements, always with the same pattern.

### 4.4.3 L3G4200D Gyroscope

From the data collected with the test 1, it was to expect that the angular rate would be close to zero as the sensor is in rest position (figure-4.10).

![Gyroscope's measurements: test 1.](image)

In the x-axis measurement, it is possible to identify some outlier measurements, but in general the behaviour of the sensor is very close to the expected one. The outlier's values can be easily filtered. Their origin is unknown, but they appear in various data acquisitions and not always in the same axis. These outliers are especially common in the gyroscope. It is clearly noise in the measurement and the user will need to deal with that, not only with this sensor, but with all four sensors. Apart from that, after filtering the outlier's values, it is possible to calculate the absolute value of the measurement's fluctuations:

\[
\begin{bmatrix}
X \\
Y \\
Z
\end{bmatrix} = \begin{bmatrix}
2.2800 \\
1.8200 \\
3.1500
\end{bmatrix} \text{dps}
\] (4.4)

The experiment performed in test 2 allowed for the collection of gyroscope data too. For circular movements in the ground (z-plane), it is expected that the angular rate measured in the z-axis would be larger than that of the remaining axes. The collected data was plotted over time and is shown in figure 4.11.

To analyse this information, the mean value of the measurements will be helpful. Performing the calculations with the collected data, the mean angular rate of each axis is:
Another method to perform a rough calculation of the mean angular rate about the z-axis is, based on 4.7 (or in the recorded movie of the experience) determine the number of completed laps. Each lap is equal to modifying the angular position by 360 degrees, so the angular position was varied a total $7 \times 360$ degrees in 20 seconds (the vehicle performed 7 laps). So, a good estimate for the angular rate is $(7 \times 360)/20 = 126 \text{ dps}$, which is very close to the mean value presented before. The measures registered in the x-axis can be related with the car’s movement, and the y-axis measurements can be related with the vehicle’s tilt, caused by the centrifugal force of the move.

### 4.5 Select the appropriated Measurement Range

Once the quality of the sensors’ measurements is validated, it is necessary to be sure that the defined measurement ranges of each sensor are the appropriate ones to measure the magnitude of values that the sensor will be submitted to.

To choose the appropriate range for each sensor, a simple test was designed. A preliminary test must be performed using the measurements ranges defined by default. This values are the ones presented in table 4.1.

Table 4.1: IMU: sensors’ Default Measurement Ranges.

<table>
<thead>
<tr>
<th>Sensor</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accelerometer</td>
<td>$\pm 2g$</td>
</tr>
<tr>
<td>Gyroscope</td>
<td>$\pm 250 \text{ dps}$</td>
</tr>
<tr>
<td>Magnetometer</td>
<td>$\pm 1.3G$</td>
</tr>
</tbody>
</table>

The car must be driven using a RC controller, connected by USB to a PC running the Simulink’s joystick solution. In this way, it is possible to manually drive the vehicle, pushing the actuators to the
limits, and forcing the car to perform the most aggressive manoeuvres like drifts, slalom and J-turns. In this way, it is possible to guarantee that the sensors will be submitted to the worst possible scenario.

The collected sensor data can be plotted over time, to see if the sensors measurements saturate and, if so, the sensors ranges must be adjusted. Then, the tests must be repeated, until a good result is found. As result, it is possible to get the ideal measurement range for each sensor, test a preliminary system and have a good time.

Following this procedure, the first set of results (from the default range values and represented in figure 4.12) failed. The magnetometer showed no problem with its $\pm 1.3G$ default range but both accelerometer and gyroscope get saturated.

In the first plot of figure 4.12 it is possible to see that some accelerometer measurements achieve an absolute value of $2g$, which is the maximum value of the default range. As so, this range can not be used.

The second plot shows the gyroscope’s data, which was supposed to saturate at $\pm 250dps$. In fact, when this range is selected, the sensor is capable to output values up to $\pm 285.58dps$ (per comparison with the $\pm 286.9dps$ determined in equation 4.1). As explained in section 4.2.3, no guarantees are provided by the manufacturer for data measured above the $\pm 250dps$ range. So, it is considered that this scale is saturated. The same exercise has been repeated, after a change the accelerometer’s measurement range to $\pm 4g$ and in the gyroscope’s measurement range to $\pm 500dps$. This time, the accelerometer’s measurements were OK and no saturation was verified, however the gyroscope’s range remains insuf-
ficient for the angular rates to be measured. In a third iteration figure 4.13, the gyroscope’s range was modified to $\pm 2000\, \text{dps}$. In this last data set, the z-axis angular rate varies between $-568.68$ and $658.91$, far from saturation, but beyond the $\pm 500\, \text{dps}$ range (the previous available range).

![Figure 4.13: Test results for appropriate sensors’ ranges](image)

So, the ranges to be used in the final architecture are summarized in the table 4.2.

Table 4.2: IMU: sensors’ Used Measurement Ranges.

<table>
<thead>
<tr>
<th>Sensor</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accelerometer</td>
<td>$\pm 4g$</td>
</tr>
<tr>
<td>Gyroscope</td>
<td>$\pm 2000, \text{dps}$</td>
</tr>
<tr>
<td>Magnetometer</td>
<td>$\pm 1.3G$</td>
</tr>
</tbody>
</table>

Note that near the 20 seconds mark, the accelerometer measurements exhibit a spike. It is a consequence of the vehicle having bumped against an obstacle, in an intentional manoeuvre. In situations like this, it is possible that the accelerometer saturates. Even though there are bigger ranges available for this sensor, situations like this are not in the scope of the developed architecture and, as such, a decision was taken to keep the $\pm 4g$ range.
4.6 Measurement Errors: Causes and Sources

There are many sources of errors associated to sensor measurements or with measurement noise:

- The location of the sensors in the system.
- There are errors associated to the Analog Digital Conversion and in the sensor’s output discretization
- The presence of certain material near the sensor can disturb its operation (as previously explained)
- The collision of the vehicle against other objects can generate physical quantities not measurable by the sensors.
- The operation temperature can affect the measurements and causes thermal drift
- The scale factor that converts the digital output to physical units has a mathematical error associated.
- The presence of BIAS errors, associated to a constant polarization of the sensors, independently of the sensors’ input.
- The Random Walk effect, associated to the effect of random noise in the inertial sensors.
- Noise in the Power Supply

The sensors’ error handling is out of the scope of this work so this subject will not be discussed. However, there is an important error source directly related with the system that must be discussed: the interference of the motors in the sensors’ measurements. A new set of data was collected, in test 1 conditions, and compared with a second set of data, collected with the car in same position, but over a platform, leaving the wheels in the air to allow motors to work without making the car move. The results are plotted and discussed here.
For the accelerometer, the fact that the motors are working increases the measurement noise. Despite the fact that the mean values of the measurements over time remain approximately the same, it is possible to see many more noise peaks in the plot related to the Motors ON situation of figure 4.14.

![Motors OFF and Motors ON](image)

**Figure 4.14:** Effect of the motors in accelerometer's measurements.

Once more, with respect to the gyroscope, it is possible to see some measurement noise even before turning the motors on (figure 4.15). With motors ON, there is an increase in noise, but the physical quantity measured varies only slightly. The motor’s operation and the wheel’s free rotation on the air produces a vibration movement in the vehicle. This vibration can be registered by the x and y-axes of the gyroscope, for which the measurement amplitudes are bigger than in the z-axis measurement.

![Motors OFF and Motors ON](image)

**Figure 4.15:** Effect of the motors in gyroscope's measurements.

According to the result of the tests performed to the magnetometer and registered in figure 4.16, the
magnetometer measurements do not seem to suffer any change with the motors operation.

Figure 4.16: Effect of the motors in magnetometer’s measurements.

In spite of the noise induced by the motor operation in some sensors, the measurements performed are still capable of providing important information about the vehicle's dynamic. However, it is important for the user to know that the IMU data must be filtered. To minimize the noise, it is possible to install the IMU away from the motors. But the installation will be always restricted to the vehicle’s structure. This options can jeopardize the system’s modularity.
Chapter 5

Integrating Solutions

In this chapter, all the individual solutions explored in the two previous chapters, will be integrated to accomplish the two final architectures. The interaction between all the subsystems will be explained and some tests will be performed to validate the overall concept.

5.1 Hardware Setup

The present section has the purpose of explain the interaction between this setup and the integrated solutions developed. One of the established objectives in the initial work guidelines, is to integrate the vehicle in the architecture with the minimal modification of the original car’s setup. This setup has been explained and illustrated in the section 3.3. In figure 5.1, this setup will be presented again, this time without a RF receiver.

As can be verified, the car setup has two cable connectors, where the PWM actuation signs must be injected to actuate the motors. A regular user of the system will not have to be concerned about the setup itself, but just about the two input cables. As so, every developed solution must be two connectors
too, to allow an easy assemblage. To the user, it is just plug and play.

5.2 Arduino Solution

The Arduino base solution, this setup involves more hardware pieces than the Raspberry Pi based solution. Here in figure 5.2, will be presented an electrical schematic. A less technical information can be found in the Appendix A.1, which is a user manual of the system.

![Electrical Schematic for Arduino Based Solution](image)

Figure 5.2: Electrical Schematic for Arduino Based Solution.

5.2.1 Logical Implementation

The Arduino solution makes use of TimerOne library to generate hardware PWM signals and this way actuate the car’s motors. This library is not provided with Arduino IDE. To know more how to get and install it, please see section B.1.1.

The SoftwareSerial library is also required, to communicate to an external shield (the WiFi Shield from TinySine manufacture), the one responsible for the UDP communication. In appendix A, there are the instructions about how to configure this shield. To know how to use it, please see section A.1.3. The Software Serial Library is provided by the Arduino IDE.

This solution uses also the wire library (present in Arduino IDE too), to make it possible to have access to each sensor of the GY-80 IMU (please see section B.4.1). An Arduino script developed to this implementation manages all the individual developed subsystems, calling each one at time, to allow the correct global operation.
A diagram in figure 5.3 is a helpful tool to understand the implementation itself.

Figure 5.3: Arduino Based Solution implementation diagram.
5.2.2 Final Command System: Type ino

Integrating all the proper hardware and the suitable logic implementation, it is possible to produce and integrated architecture to accomplish the pre-established requirements. In this section the architecture will be presented in figure 5.8 and will be described in order to help to understand the general operation.

Figure 5.4: Arduino Based Solution final setup.

1. A computer, running a Simulink solution, developed in section 3.6, must be integrated in the same IP network that the Wi-Fi shield.

2. The computer is able to send UDP command to a target IP. The DHCP of the Wi-Fi shield must be deactivated, and they IP must be set. More information about how to do it are present in Appendix A.1.3.

3. The UDP command message contains information about the PWM actuations signs, according the table 3.7. This message is received by the Wifi Shield and passed to the Arduino trough Software Serial Communication.

4. In the Arduino loop, this message will be received and if it is the correct information, the PWM commands will be given as input to the actuators, through the wire connections.

5. After the actuation, the Arduino send I²C commands to each IMU sensor asking for measurements, and listen the sensors' measurements from the I²C port (section B.4.1).

6. The sensors data are converted into a decimal base, and rearranged in a string. This string is passed to the Wi-Fi shield using SoftwareSerial port.
7. The shield send the sensors raw sensors’ measurements to a target IP. This target must be correctly defined by the user during the shield configuration. Note that it is not mandatory that the device who receives the sensors information’s be the same from where the command messages was being send. However, once the idea is perform the control of the vehicle, this seems a more logical option.

This sequence of actions will be repeated in the loop, until the user use the Simulink environment to stop it, or the vehicle be shunted down.

5.2.3 User Interface and the Controller in the Loop

Controlling the vehicle using a computer leads to the developed of some Simulink models which are presented in section 3.6, and listed as 1 in the previous section. The present section appears from the need to explain the achievement of the main goal of all the developed work. The developed architectures’ main purpose is the possibility to use this solution in the control classes, to allow the future generation of students to design and implement control techniques to take the vehicles to perform some a mission. For this, the architecture itself must be ready to integrate control loops, and receive their output as input. In fact, the Simulink user interface, discussed in the section 3.6.2 can include an option for a Controller Block. The system is ready to provide to this block as input the IMU’s raw data containing the cars’ attitude information. The ADIS localization system is being developed, but is already capable to provide the vehicles position information in a 2D referential system. In a near future, 3D localization will be supported too. This way, all the inputs required from a Control Block will be available and usable.

The option of the system outputs the IMU's raw data in physical units was taken, in view to provide to the user a knowledge about the meaning of each value, and allow a bigger range of possible uses. According the individual control implementation of each developer, the raw data to be used as input, must be processed. Figure 5.5 shows the integration of a generic controller in the loop.

![Figure 5.5: Insertion of the Controller in the Loop, using the Simulink environment.](image-url)
5.3 Raspberry Pi Solution

The Raspberry Pi setup is quite more easy than the Arduino’s setup. Here in figure 5.6, will be presented an electrical schematic. A less technical information can be found in the Appendix A.2, which is a user manual of the system.

![Figure 5.6: Electrical Schematic for Raspberry Pi Based Solution.](image)

5.3.1 Logical Implementation

To implement the programmable logic in Raspberry architecture, the python programming language was used. Python in versatile, already provide many libraries and tool boxes. The IMU setup library, included to interact with the IMU itself is an open source code.

The pigpio toolbox is a freeware toolbox, designed to interact with the GPIO pins, and used in the solution development. To know how to get this code and what is need to use it, please see appendix B.1.2.

Based on the inbuilt functions of socket library, provided with the python original code, an UDP communication library was written.

Several of other inbuilt libraries was used too, to implement a logical solution that allow control the RC car. Once more, a simplified diagram in figure 5.7 helps to understand the code’s structure.
A dark grey block was intentionally not implemented. This block will be explained in section 5.3.3.
5.3.2 Final Command System: Type Pi

The Raspberry Pi based architecture it is able to operate in similar conditions that the Arduino Based Solution. To present the overall architecture, and explain all the single node of the architecture, it is schematized in figure 5.8.

![Raspberry Pi Based Solution final setup](image)

**Figure 5.8: Raspberry Pi Based Solution final setup.**

The architectural’ structure itself is composed by the following sequence of actions:

1. A computer running the same a Simulink solution of section 3.6, must be connected to the same IP network that the Raspberry Pi.

2. The computer is able to send UDP command to a target IP. The user must guarantee that the Raspberry Pi's IP is defined to be static.

3. The UDP command message contains information about the PWM actuations signals, according the table 3.7. The messages are received in the input socket, from where must be read. Once received by the Raspberry Pi, command values must be scaled.

4. The Raspberry Pi actuate each one of the commands just by redefining the duty cycle for each one of the *pigpio* classes. The GPIO pins will output hardware PWM waves.

5. After the actuation, an I2C interfaces is used to communicate with each one of the IMU's sensors and collect the measurements. The I2C feature must be enabled by the user in the Raspberry Pi settings [10]. This functionality is implemented by the *IMU_setup library* that have a lot of resources to operate the specific GY-80 model.

6. The sensors data are converted into a decimal base, and rearranged in a string. this string this string is written in the output socket, to be communicated to a target IP. This target was coded defined by the user.
7. Once more, it is not mandatory, but can be recommended that the IMU’s raw measurements data be communicated to the same remote PC which are sending the control commands.

5.3.3 User Interface and the Controller in the Loop

The system typical use was already being described in the section 5.3.3. The way how the user can interact with the system and use they own controllers to command the vehicle, is equally a valid solution for the Raspberry Pi based solution. However, this alternative approach was thought in order to also accomplish some different purposes. Use a Raspberry Pi, with more calculation capability than Arduino, allow the computation of the control by the Raspberry Pi itself. Looking at the block diagram of the figure 5.7, the darks grey block represents the place in the loop where a coded controller can be implemented. For this, the communication system must receive the localization information (previously communicated from the ADIS to the controller PC). There is no need to communicate the sensors’ raw data, because it would be processed internally, by the coded controller. This alternative operating mode disconnect the car from an external commander (the remote PC) increases they autonomy, but it requires a more proficient user in programming skills.

Figure 5.9: Insertion of the Controller in the Loop, without the Simulink environment in the Loop.
Chapter 6

Discussion and Results

6.1 Performances

Once the two architectures are fully developed and finished, it is important to compare them in several aspects. The first parameter to be evaluated is the system’s performance. At the beginning of this document, a minimum work frequency of 30Hz was established. If the system can work above this goal, it will take maximum advantage of the coordinates position from the ADIS localization system. Each one of the developed systems was put to work at a time. In MATLAB®, the script used to receive and scale the IMU’s raw data, transmitted via UDP from the solutions installed on-board the vehicle, is used to count how many messages are received at a given time interval. This way, it is possible to infer about the system work frequency. The results for the tests’ work frequencies are presented in table 6.1

These frequency values already include the IMU raw data processing in the remote PC. For the Raspberry Pi, the two frequency values marked with an asterisk, are related to the experiences where the VNC remote desktop is running to, sharing the Raspberry Pi screen with a smart-phone. The most obvious result to be commented, is the accomplishment of the work frequency, always above the 32 Hz in all the performed tests. The results from the Raspberry Pi based solution are marginally better than the Arduino Based Solution and the use of a VNC system does not have a significant impact on the work’s frequency. However, a more detailed analysis can be done. For each solution, the table 6.2 shows the range of the running time for each individual solution integrated in the system.

For the Arduino Based Solution, it is possible to understand that the most delayed action is to receive

<table>
<thead>
<tr>
<th>Operation’s Frequency</th>
<th>Arduino</th>
<th>Raspberry Pi</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 seconds</td>
<td>32.20 Hz</td>
<td>33.40 Hz</td>
</tr>
<tr>
<td>30 seconds</td>
<td>32.56 Hz</td>
<td>33.70 Hz</td>
</tr>
<tr>
<td>1 minute</td>
<td>32.95 Hz</td>
<td>33.18 Hz</td>
</tr>
<tr>
<td></td>
<td></td>
<td>33.17 Hz*</td>
</tr>
<tr>
<td>10 minutes</td>
<td>32.33 Hz</td>
<td>33.30 Hz*</td>
</tr>
</tbody>
</table>

Table 6.1: Comparison of the work frequencies of both solutions.
Table 6.2: Comparison of the running times for each solution.

<table>
<thead>
<tr>
<th></th>
<th>Arduino Solutions' Running Time [ms]</th>
<th>Raspberry Pi Solutions' Running Time [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Receive Actuation Commands (UDP)</td>
<td>[2;30]</td>
<td>[0;12]</td>
</tr>
<tr>
<td>Motors Actuation</td>
<td>[0;1]</td>
<td>[0;1]</td>
</tr>
<tr>
<td>Get IMU Measurements</td>
<td>[8;10]</td>
<td>[9;15]</td>
</tr>
<tr>
<td>Communicate IMU Measurements (UDP)</td>
<td>[3;4]</td>
<td>[0;2]</td>
</tr>
<tr>
<td><strong>Total Loop Time</strong></td>
<td>[13;45]</td>
<td>[10;30]</td>
</tr>
</tbody>
</table>

the UDP messages through the UDP shield. In the Raspberry Pi based solution, it takes less than half than Arduino’s Based Solution time. It is an expected result, because in the first solution, there is a limitation associated to the use of an external hardware piece, which means one more node in the architecture and one additional connection to be established to complete the flux of information. The commands message was communicated from the remote PC to the WiFi shield through UDP protocol and then, this message needs to be communicated from the shield to the Arduino through Software Serial communication. It’s this additional connection that is not necessary in the Raspberry Pi approach.

In both cases, the generation of hardware PWM waves is almost instantaneous, and the actuation of the motors is the less delayed process. According the section 2.6.1 in the Arduino solution, the received values are already in a format that allows for a direct actuation, instead of what happens in the Raspberry Pi, where the values need to be scaled before the actuation itself. Despite this, the processes present the same delay in both solutions. Access to the IMU’s sensors to get their measurements is slightly faster in the Arduino Based Solution. In this solution, the IMU access was developed just to read and write directly on the sensors’ registers, in a very low level programming language. On the other hand, the python library used in Raspberry Pi Based Solution, was developed to be much more polyvalent and perform a lot more calculations and data processing. Obviously, all of this takes time and delays the procedure, however, a maximum delay of 15 milliseconds to get the measurements is fully acceptable, considering this time delay relatively to the entire solution. To communicate the IMU’s raw data through the UDP communication, the Raspberry Pi base solution behaves a little better than the Arduino Based solution. Once more, this is because there is a less amount of hardware in the loop.

Analyzing the total delays of both solutions, some considerations need to be done. The maximum delay of the Raspberry Pi’s approach is 7% lower than the maximum delay of the Arduino’s solution. This maximum delay values are directly related with the lowest instantaneous frequencies which the systems can operate. The Raspberry Pi’s Solution maximum delay allows for a minimum instantaneous frequency of 33.3 Hz (1/30 milliseconds), value that agrees with the values in table 6.1. In the Arduino’s solution, this does not happen. The Arduino maximum delay of 45 milliseconds allows for a minimum instantaneous work frequency of 22.2Hz (1/0.045). It is far from the value of 32Hz presented in the table 6.1. 22.2Hz represents the worst-case scenario and table 6.1 refers to a set of messages along a time set. So, it is possible to conclude that this worst delay scenario is not frequent enough to lower the work frequency below the 32 Hz.

As so, both solutions are reliable and the table 6.1 proves that the two systems work above the required 30 Hz. It may be interesting to try increasing the work frequency of the systems, although in
section 3.5.1 it was discussed that for higher work frequencies, the number of lost messages increases, for the UDP protocol. Then, this is an important factor to take into consideration.

6.2 Hardware and Setup Complexity

The used hardware of each developed architecture can be a differentiator in the choice of the system.

The Arduino Based Solution requires an additional hardware to establish Wi-Fi communication with a remote PC. A bigger amount of hardware to assembly leads to a more complex setup, and in this aspect, the Arduino’s Based solution is in disadvantage.

Another important limitation of this approach compared with the available alternative solution, is the hardware PWM waves generation. The Arduino Uno board just supports this feature in 2 of their 13 digital pins, which means that just two actuators can be used. It is ok for a 4 wheels’ vehicle like the LaTrax SST, but can be a limiting factor to the use of more complex vehicles (either land vehicles or not). On the other hand, the Raspberry Pi supports hardware PWM generation in all the 24 GPIO pins. It does not mean that all the GPIO pins can be used at the same time, but certainly more than 2 will be allowed.

Raspberry Pi is basically a small sized computer, as so, it provides more calculation capabilities than the Arduino. This fact can be an advantage or a disadvantage.

It is an advantage because it allows more sensors integration, more on-board data processing and more versatility in the choice of what program language is used to interact with the hardware. But it is a disadvantage, because it requires the use of computer peripherals and/or another methodology to remotely interact the device. Furthermore, the Raspberry Pi requires more current input to power up(2.5A).

The differences are felt in the devices boot too. While the Arduino automatically runs a pre-loaded script in an infinite loop in a few seconds when it's turned on, the Raspberry Pi requires a more delayed start, like every PC.

6.3 Usability and Versatility

Both Arduino and Raspberry Pi are widely used platforms, as so, the internet provides many documents about how to use them and how to solve a big amount of their problems and limitations. The Arduino platform is easier to use, for a less programming skillful user, however it requires a computer to run the Arduino IDE where the user can code, compile and upload the code to be run in the Arduino. The Raspberry Pi is a computer itself, which means that the user can code directly on the device and choose the programming language to use. This fact, associated to the fact that there's a higher calculation capability, makes the Raspberry Pi a more versatile tool. In the case of the developed architectures, the Raspberry Pi based solution provides the user with the possibility of it being independent of the remote PC with a Simulink® environment. The Raspberry Pi itself can be coded to integrate a programmed controller of the system. It gives to this solution a bigger independence, which is without doubt an
advantage over the Arduino solution. In terms of needed configurations, the complexity is similar in both systems. To the Arduino, the Wi-Fi shield requires a manual configuration of several parameters to be used. The instructions to do that can be found on appendix A.1.

The Raspberry Pi requires that the user installs an operative system and configures several services. More information about it is presented in appendix A.2.

Because the system that uses the Raspberry does not need an external hardware shield, it means that the final hardware block of this solution will be lighter and less bulky (see figure 6.1. It is an advantage too, because it makes the hardware easier to couple to the vehicle.

![Figure 6.1: Size comparison of the Hardware Blocks of both solutions. Arduino block (on the left) and Raspberry Pi block (on the right).](image)

### 6.4 Architectures’ Costs

On a costs perspective, both solutions are very close to each other. If, on one hand, the Raspberry Pi is a little more expensive than the Uno Board from Arduino, on the other hand, the Wi-Fi shield, needed for the Arduino Solution costs the same as the Raspberry Pi itself. The Raspberry Pi solution requires an additional cost to an external power supply and a micro SD memory card, that together have the same costs as the Arduino Board.

For this, the cost is not a tiebreaking factor.
Chapter 7

Conclusions and Future Work

Conclusions

This work comes about to develop an architecture that allows for the integration of commercial RC vehicles in an indoor localization system. For environments where the use of GPS is denied, alternatives must be found. In this scope, it is being developed a low-cost solution to provide vehicles' location. The overall concept of the final global system is to have integrated solutions that allows for control on land and air vehicles.

This work explores the required system to be produced in order to provide the land vehicles integration in this environment. Before the development itself, some pre-requirements were established, and some important goals were outlined. The final cost of the final architecture must remain low, but the developed systems must be modular and easy to replicate. It must also be reliable and easy to use. An important aspect to highlight, is the establishment of the minimum work frequency admissible for the system as 30Hz. This requirement is related to the localization system operation, that should not be limited by the developed subsystem, focused on interaction with vehicles.

The first step to be taken, is to choose a vehicle model and to study the way its actuators works. Once that is established, it is possible to start to think in what kind of hardware must be used to achieve the final solution.

It was decided take two different approaches and develop two alternatives systems. Despite these two systems being designed to accomplish the same functionality and work the same way, the decision of these efforts is justified with the different growth prospects of each approach.

An Arduino based approach will aims for a more user friendly solution, perfectly capable of responding to all the requirements. With this solution, it is possible to control the vehicle wirelessly with a remote PC, get the on-board sensors' data and communicate them to the PC where the controlling loop is running. The remote PC must receive as input, the sensors' data sent from the vehicles and the position's information sent from the localization system as well.

A Raspberry Pi based approach, was fully developed to work in the same mode of the Arduino Based approach. However due to a bigger calculation capability of the hardware, this approach provides a
possibility to perform the controller calculations internally. It allows the user to remove the remote PC from the equation and also providing the vehicle with more autonomy. Despite this solution being able to improve the architecture independence, it requires more of the user’s programming skills, whereby the ease of usability can decrease.

Since there are defined which hardware components to use to implement the final architectures, the work passes by creating all the individual solutions to establish all the nodes of the system that are to be implemented. Some of the developed solutions were:

- the generation of hardware PWM waves to be used in the vehicles’ motors actuation;
- the use of the UDP protocol to guarantee the communication between the vehicle and a remote PC
- the implementation of a Simulink model to allow the user to implement their own controller module and use them to remotely command the vehicle

To add value to the global system, the architecture was instrumented with an IMU to recall raw data about the vehicles’ attitude. The validation of the IMU’s sensors data was in the scope of this work too. A last stage of the work consists in integrating all of these individual solutions, to accomplish a final system, capable of responding to all the pre-established requirements. The discussion of the results, provides a comparison of the two produced integrated solutions, in order to evaluate the strengths and weaknesses of each one.

Both approaches are able to work above 32Hz, supplanting the 30Hz established at the beginning.

Even during the development stage, a preliminary version of this work was used by a pair of students of the Optimal Control classes, to design a position controller that commands the vehicle to perform a pre-established route. It was an integral part of their final assessment. As such, this is the ultimate proof of concept and a validation of the systems’ utility. Hereupon, and inspired by these students’ performance in the subsequent section will be presented a list of possibilities of situations where the developed system can be used.

**Suggested Missions**

Once more, it is recalled that the main objective of this work is to produce a tool capable to be used implement control models. This section will suggest a set of missions which the controller can eventually projected to command a car (or a more than one) to perform a mission.

The suggestions were thought taking into account a more general architecture that integrates the developed system with the AIDS system. This way, there is information about position and attitude of the vehicles.

**Missions for a Single Vehicle:**

- perform a circuit with precision;
• perform a circuit as fast as possible;
• perform a circuit with complex moves (such as drifts and j-turns);
• perform a circuit avoiding obstacles;
• follow another vehicle or an object;
• perform a parking manoeuvre (different positions);

**Missions for Multiple Cooperative Vehicles:**
• transport a payload;
• perform coordinated movements;
• compete as a team (relay race);
• work as a swarm;
• auto-management of a parking system;
• emulate a traffic environment;
• emulate air traffic control systems;

**Missions for Multiple Competitive Vehicles:**
• Running alone against the clock (timed lap);
• Running a race and avoiding crashes;
• Work independently but avoiding crashes;
• Follow another vehicle;
• Try to outwit a tracker vehicle;

**Future Work**

Despite the developed work achieving the goals that were proposed at the beginning, some additional efforts should be directed to improve the robustness of the developed systems and therefore extend its scope. For this, the present section will address some considerations about future improvements to this work.

The first and most important target is the inclusion of aerial vehicles in the system. For this, the ADIS localization system must be able to provide the 3D special location of multiple objects. The inclusion of this aerial vehicle will not pass by creating a dedicated system like the ones developed throughout this work, but rather by using the Parrot AR drone 2.0, and the respective API provided by the manufacture
to interact with the vehicle. This interaction must be guaranteed through a Simulink environment like the solutions developed in this work, to keep it usable for the largest possible number of students. In fact, it is already a work in progress and should be finished soon.

Talking about the developed solutions, there is some improvements that will benefit the overall developed systems, a more robust protocol must be architected in order to codify the messages that need to be communicated to and from the system. The used protocols allow discarding wrongly received messages, however, there is a benefit in the use of protocols that use checksums and indicators. This step only make sense after the aerial vehicles are included. Only this way is possible to guarantee that the communication protocol is universal and common to all the systems.

Another possibility can be to increase the set of sensors on-board the vehicle. It will allow the vehicles to gain new capabilities and consequently improve the diversity of allowed missions to these vehicles. An ultimate system upgrade would make a vehicle fully autonomous, and independent of an external system to provide its location is space, but at some time, keep it at price low.

Other possible way to follow, is to decrease the cost of developed architectures. It can be done by producing dedicated hardware. However, one of the scopes of this work is to guarantee the systems’ flexibility, and take this option will compromise this flexibility.

Ultimately, recalling that the overall concept of the developed work was inspired from previous work of another student. These integrated architectures were thought to be used by the students, to practically implement their theoretical control knowledges. When someday, some other student uses the concept developed in this work, new ideas about new applications will appear and maybe someday this work can inspire a new path that leads to new, innovative future work.
Bibliography


Appendix A

User Manual

A.1 Arduino Based Solution

A.1.1 List of Material

- 1 Arduino Uno
- 1 3D-Printed Arduino case
- 1 Wifi Bee (TyniSine) + external antenna
- 1 Wifi Shield (TyniSine)
- 1 IMU GY-80
- 1 External Battery (to power up The Arduino)
- 1 USB type A - USB type B cable
- 2 cable connectors
- 1 LaTraxx SST vehicle
- Velcro cable ties

A.1.2 Hardware Setup

Consulting the FIG on CHAP, it is possible to verify that all the Arduino Uno pins are properly identified in the PCB board.

The connection between the Arduino Uno and the Wi-Fi shield is an easy plug, as the second one fits perfectly into the first. All the shield’s pins must be connected to all the Arduino’s pins. After mounting, no pins should remain free. The two boards just fit one way, so there is no risk of missing this connection or of damaging the materials. This GY-80 requires a 5V power input. The pins SCL and SDA must be connected to pins A4 (SDA), A5 (SCL) of the Uno board.
The TimerOne library just can generate hardware PWM waves in pins 9 and 10 of the Arduino Uno board. Pin 10 was defined as the throttle pin, and Pin 9 was defined as a direction pin. Each one must be connected to a cable connector, together with the Arduino’s ground pins. To powers the direction servo, the output voltage (red wire) of the ESC cable connector must be short-circuited to the power red wire of the servo motor cable. To provide a visual information about how to setup all the hardware components, a wire diagram is provided in figure A.1.

![Figure A.1: Wire diagram showing the connections and global setup for Arduino solution.](image)

To simplify the schematic, all wire connections are represented around the Arduino and WiFi Shield. In the practical implementation, the prototyping area of the WiFi Shield was used to install pins that allows the fast connection and disconnection of all cables. The IMU must remain free, to be able to be installed in any part of the vehicle. After assembling and doing all the connections, the hardware can then be packaged in a 3D printed custom case. This case protects the material from possible damages and also hides the setup complexity from the user, leaving only the connectors wires visible and accessible.

### A.1.3 Hardware Configuration

The Arduino itself is ready to use. However, to be possible to compile and upload the Arduino’s scripts, the Arduino IDE must be installed. It can be found on Arduino Official web site [11]. On the other hand, the Wi-Fi shield requires a complex configuration. All the steps that are needed, are presented here:

**TinySine WIFI shield configuration**

There are two different ways to configure the WIFI shield: through the Arduino serial port, or by means of wireless. Only the first way will be discussed in this document. For more information about the second
process, please read the user manual [12]. For a correct configuration of the shield, please follow these steps:

1. Plug the Arduino board and the Shield together. Set the shield switches: S1 to Normal, S2 to PC and S3 to SoftSerial.

2. Connect the Arduino board to computer via USB port.

3. Open de Arduino Software and upload the example sketch ‘Blink’. Make sure it is uploaded by the correct COM port.

4. Set the S3 switch to Hard position.

5. Open the Serial Monitor of Arduino Software (by pressing ctrl+shift+M). At the bottom bar, select the baud rate set up on the board (9600 bps as the default value. If it has been changed to another value, please choose the proper value). On the same bar, select the ‘No Line Ending’ option.

6. Send the command ‘$$’ (without the single quotes) to enter into command mode. If everything is ok, the reply will be ‘CMD’. Otherwise, please repeat all the previous steps.

7. Change the monitor options again to ‘Carriage Return’. Try the AT command ‘get everything’. A list of every configurable parameter must be shown. Now there is a need to setup some module settings through some AT commands. The reply ‘AOK’ indicates a well succeeded configuration.

8. Join the device to a local network:
   
   ```
   set wlan phrase < wpa password >
   set wlan ssid < network’s ssid >
   set wlan join 1
   save
   ```

9. Fix the device IP address:
   
   ```
   set ip a < desired IP >
   set ip dhcp < flag >
   set ip local < desired local port to listen >
   save
   ```

   Where the command ‘Set ip dhcp <flag>’ enable/disable the Dynamic Host Configuration Protocol (DHCP) mode. DHCP allows a device to obtain a IP address automatically. To fix the IP, DHCP must be disabled (flag=0).

10. To choose the UDP IP protocol (TCP is the default one), please use the flowing command:

   ```
   set ip protocol <flag>
   save
   ```
Flag must be 1 to UDP, 2 to TCP or 3 to both protocols.

11. Finally, configure the transmission baud rate. 115200 bps is the maximum baud rate supported. To see all values supported, please consult the SoftwareSelian library’s documentation [6]

   'set uart baudrate <desired baud rate>'
   'save'
   'reboot'

After rebooting, the Serial Monitor will exit command mode. Repeating points 4 to 7 it’s possible to check if every configuration is ok. It is easiest if the commands

   'get ip'
   'get wlan'
   'get uart'

instead of 'get everything'. The commands must be used without the single quotes. This verification is important, because if we apply a correct command, but with an invalid flag/value, the response will be positive, but the setting won’t be configured.

To check if the configuration process was successful, a Windows Terminal must be opened. The command ping <WiFiBee_ip>, can be used to make sure that WiFiBee is on local network with the desired IP, and it is possible to communicate with him. An example video named 'WiFiBee_config' has been recorded to help to reproduce the process. This video can be found in a multimedia folder, in the main directory.

A.1.4 Implemented Code

The script to upload is the Arduino RC.ino file. This file must be opened with the Arduino IDE. The Arduino must be connected to a PC using a USB cable. Once the PC recognize the Uno board, the user must click in the upload button. This will compile the script and upload it to the Arduino Uno. Once the script is uploaded, the Arduino will run it in an infinite loop. It is not necessary to perform this procedure every time the Arduino is turned on. The script will remain on memory, even when the hardware is turned off.

A.1.5 Mounting the System in the Vehicle

The system has two cables with female connectors. The cars setup has two cables with the male connectors. The car's cable that is connected to the ESC must be connected to the cable which comes from pin 10 of Arduino. The car's cable of the direction servo motor must be connected to digital pin 9 of Arduino. The system case must be secured to the vehicle. For this, Velcro cable ties are available. An external battery and micro-USB cable allows to powers up the Arduino. The battery must equally be fastened to the vehicle. All the cables need to be installed away from the wheels to avoid damaging
the system. The user is free to choose the IMU localization in the vehicle. A 3D printed support for the IMU is available. Once the whole setup is complete, it's time to use the battery's button to power up the system.

### A.1.6 Operation Verification

Turning the system on, several led will blink in the Arduino Uno Board, and in *WiFiBee*. Looking through the opening at the top of the Arduino case, the WiFiBee's led can provide information about the state of the system. The led must be blinking a green colour and in a slow manner. If so, everything is OK, and the system is waiting for UDP commands. It is time to start the Simulink and enjoy it. For this, in a remote PC, the user must run the `START_HERE.m` from LaTraxSST's Simulink Solution. This script provides a graphical way to the system start. Once the system has been correctly started, the led must blink a yellow colour and in a fast manner. It indicates that the car is receiving the commands. If the colour code of the blinking led do not fit this description, something has been done wrong in the shield configuration. The figure A.2, form the *WiFiBee* official document [12] helps to understand the system status:

<table>
<thead>
<tr>
<th>Condition</th>
<th>Red LED-D3</th>
<th>Yellow LED-D2</th>
<th>Green LED-D1</th>
</tr>
</thead>
<tbody>
<tr>
<td>On solid</td>
<td>-</td>
<td>-</td>
<td>Connected over TCP</td>
</tr>
<tr>
<td>Fast blink</td>
<td>Not associated</td>
<td>Rx/Tx data transfer</td>
<td>No IP address</td>
</tr>
<tr>
<td>Slow blink</td>
<td>Associated, no Internet</td>
<td>-</td>
<td>IP address OK</td>
</tr>
<tr>
<td>Off</td>
<td>Associated, Internet OK</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

![Figure A.2: WiFiBee Led States.](image)

**Important note:** if the user want to use the system a IP address that is not the default one, execute the `START_HERE.m` script and before running the Simulink itself, the MATLAB variable `<remotIP>` must be modified to the desired IP address.
A.2 Raspberry Pi Based Solution

Attention: the solution only works with the Pi3, because it is the only Pi (at the date) that supports Wi-Fi.

A.2.1 List of Material

- 1 Raspberry Pi model 3B
- 1 3D-Printed Raspberry Pi case
- Class 10 micro SD memory card
- 1 IMU GY-80
- 1 External Battery (to power up The Arduino)
- 1 microUSB - USB type A cable
- 2 cable connectors
- 1 LaTraxx SST vehicle
- Velcro cable ties

A.2.2 Hardware Setup

The Raspberry PI PCB board gives no any information about the GPIO pin map. Since it’s important for the user to know this information, the Raspberry Pi3 pin map is presented in figure A.3.

![Raspberry Pi3's pin map](image)

Figure A.3: Raspberry Pi3’s pin map.

This GY-80 requires a 5V power input. The pins SCL and SDA must be connected to pin 3 (SDA), 5(SCL) of the Raspberry GPIO pins. The direction pin was defined to be the GPIO pin 33 and the throttle pin was defined to be the GPIO pin 11. The white wire of the connector cables must be connected to
these pins. The black wire must be connected to any ground pin. To powers the direction servo, the output voltage (red wire) of the ESC cable connector must be short-circuited to the power red wire of the servo motor cable. To provide a visual information about how to setup all the hardware components, a wire diagram is provided in figure A.4.

![Wire diagram](image)

Figure A.4: Wire diagram showing the connections and global setup for Raspberry Pi solution.

After assembling and doing all the connections, the hardware can then be packaged in a 3D printed custom case. This case protects the material from possible damages and also hides the setup complexity from the user, leaving only the connectors wires visible and accessible.

### A.2.3 Hardware Configuration

The Raspberry works like a computer, as so, an Operational System (OS) is required. To install it, the user needs a monitor and a keyboard. These computer peripherals are indispensable for the OS installation, but after that, although useful, they are dispensable. The recommended OS is the Raspbian, that is a Debian based OS and can be download from the official raspberry website [13]. After downloading the OS and having the .img file, the Raspberry Pi documentation can be consulted on how to install it. The installation instructions can be found in: official raspberry website [13].

The Raspberry Pi hardware is ready to support the I^2^C communications, however in this case, the service needs to be enabled and configured. All that is needed about how to make it can be founded in adafruit’s web site [10]. It is important to set the Raspberry Pi's IP as static. For this, the first step is to connect the device to a local network through Wi-Fi. The easy way to configure the Raspberry’s IP is to go to the network rooter settings, and manually impose an IP. Every time that the Raspberry connected to this network, it will assume this IP.

The Raspbian operating system already comes with the python’s shell installed, so the user does not need to worry about this, however, there’s still the need to download and install all the necessary external libraries.

The solution to generate usable PWM wave on Raspberry Pi require the use of the pigpio library. This library can be obtained on-line [14]. It is possible to consult all the library documentation in the
same reference. To use this library, there is a need to install the *pigpio* Daemon, also known as *pigpiod*. This utility launches the *pigpio* library as a daemon, but requires sudo privileges. As so, the terminal command 'sudo pigpiod' must be executed every time that the raspberry is turned on. It is possible that the final scripts can run without the execution of this command. If it happens, is because this command was programed to run on the boot. If the command was not executed, the *pigpio library* cannot be launched, and an error occurs.

The GY-80 IMU open source library, is available to downloaded from [15] . To use this library, some modifications was performed. As so, the use of the 'IMU _ setup.py' file is recommended instead of the original file.

### A.2.4 Mounting the System in the Vehicle

The system has two cables with female connectors. The car’s setup has two cables with the male connectors. The car’s cable that is connected to the ESC must be connected to the cable which comes from the GPIO pin 11 of Raspberry. The car’s cable of the direction servo motor must be connected to the digital GPIO pin 33 of Raspberry.

The system case must be secured to the vehicle. For this, Velcro cable ties are available. An external battery and micro-USB cable allows to powers up the Raspberry. The battery must equally be fastened to the vehicle. All the cable need to be installed away from the wheels to avoid damage to the system. The user is free to choose the IMU localization in the vehicle. A 3D printed support for the IMU is available. Once the whole setup is complete, is time to use the battery's button to power up the system. And the run the 'Python - RC.py' script.

### A.2.5 Implemented Code

To run a script in python, the user just needs to open it and press F5. The python's shell will be automatically opened and the script will run. The script to run is the 'Python - RC.py'; however all the included files and external libraries must be in the same domain of the main file. To facilitate this, the Python Solution provided a "lib" folder, containing all the required files. Even so, a list of all the included files is presented here:

- adxl345.py
- bmp085.py
- hmc58831.py
- i2cutils.py
- IMUsetup.py
- l3g4200d.py
- quaternions.py

72
A.2.6 Operation Verification

In order to run the script and check if everything is conform, the user must be able to give keyboard inputs to the raspberry and see the printed messages on the python shell. If the computer’s peripherals are available, these can be used. If not, an alternative approach can be used configuring a remote desktop in the Raspberry Pi to get access from another computer or from a smart-phone. For this the VNC software must be used. To know how to install, configure and use it, please see the set of tutorials presents in [16].

Since the user can access the OS graphic’s environment, the pigpio Daemon must be executed and the ‘Python RC.py’ file must be run. A set of messages will appear on the python shell, this messages helps the user to understand what’s going on. Every time that something goes wrong, the script shows in the shell a red error message and interrupts the code. When no error occurs, the messages is written in blue.

Arming the motors

If by some reason the Raspberry is not able to use the pigpio library to generate hardware PWM waves the error message that will appears is

```
∗ ∗ ∗ Fail in Arm The Motors ∗ ∗ ∗
```

usually the cause of this error is the fact that the pigpio Daemon has not been launched. So please execute the command ‘sudo pigiod’ on the terminal. When no problem occurs, the messages is

```
∗ ∗ ∗ Motors Are Successful Armed ∗ ∗ ∗
```

IMU sanity test

Before receiving UDP commands, perform a first measurements with the IMU and use the result to determine if the vehicle starts from the rest. If not, the script will be interrupted and the error messages will be printed

```
∗ ∗ ∗ Not Starting From The Rest. Please Make Sure Your Vehicle Is Static! ∗ ∗ ∗
```

if the vehicle is at rest, the message will be

```
∗ ∗ ∗ IMU Starting From Rest***
```
Now the script will be listening for the UDP commands. If at this time the user has not started the Simulink yet, the system will output the message

∗∗∗Waiting for commands ∗ ∗ ∗.

Once the Simulink has been started, and the commands have been sent, the system will show its last message, informing the user that it has started:

∗∗∗The System Is Working Now! Enjoy! (; ∗ ∗ ∗.

Now the user can turn off the Remote Desktop if so desired. The VNC remote desktop does not causes any effect in the system’s performance. If the user chooses to keep it on, it is able to use the ctrl+c to stop the system when wanted, however it can be done just by stopping the Simulink.

**Important note:** if the user wants to use the system a IP address that is not the default one, execute the START HERE.m script and before running the Simulink itself, the MATLAB variable <remotIP> must be modified to the desired IP address.
Appendix B

Implementation Issues

Throughout this chapter will be presented and discussed some implementation issues. The main purpose of this Appendix is to justify some implementations decisions and to facilitate the future replications of the architecture developed along this work. As so, the appendix will focus in the solutions development.

B.1 PWM Generation

B.1.1 PWM and Arduino Uno

analogWrite()

To emulate a PWM signal, an Arduino Uno can be used. The Uno board counts with fourteen digital ports (numbered from 0 to 13), but only ports 3, 5, 6, 9, 10 and 11 supports PWM generation. The easy way to generate a PWM waves to use the analogWrite( ) function, which receives an output pin and a duty cycle as arguments. Please see the documentation [17].

analogWrite (pin, dutyCycle);

Note that the dutyCycle is a variable within the interval from 0 to 255, where dutyCycle=255 means a 100% dutyCycle, and dutyCycle=127 means a 50% dutyCycle and so one [18].

A PWM wave was generated with analogWrite( ) function and compared with the signal that needs to be emulated. Obviously, an oscilloscope is needed to see the results. Despite this simple way to generate a PWM signal, it is not useful. On figure B.1 it is possible to see why. The Arduino PWM wave (red wave) and the same duty cycle wave, generated by the RF receiver (blue wave) are represented. The wave parameters determined on PicScope software are also summarized in the table B.1.
Figure B.1: Comparison between the original PWM wave, and the Arduino Uno emulated PWM wave using `analogWrite()` function.

Table B.1: Comparison between the original PWM wave, and the Arduino Uno emulated PWM wave using `analogWrite()` function.

<table>
<thead>
<tr>
<th></th>
<th>Arduino signal (red wave)</th>
<th>Original signal (blue wave)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency</td>
<td>489.9 Hz</td>
<td>101.6 Hz</td>
</tr>
<tr>
<td>Cycle Time</td>
<td>2041 $\mu$s</td>
<td>9846 $\mu$s</td>
</tr>
<tr>
<td>Duty Cycle</td>
<td>15.21%</td>
<td>15.21%</td>
</tr>
<tr>
<td>High Pulse Width</td>
<td>325.2 $\mu$s</td>
<td>1500 $\mu$s</td>
</tr>
</tbody>
</table>

As can be seen, even with the same duty cycle, the two waves are completely different, as they have different frequencies. The Arduino’s generated wave was 489.9Hz against the already known 101.6Hz of the wave that was pretended to emulate. So this method in not useful for this work proposes.

**Forcing Cycle Time**

As the wave’s duty cycle is the percent of *high time* in the total *cycle time*, an alternative option to generate a PWM signal is to force the frequency of the wave. To do this, a digital pin will be set HIGH and LOW, using the function `digitalWrite()`. Just controlling how long the pin stays in each condition, the duty cycle and the frequency can be set. For this, the `delayMicroseconds()` function must be used.

Figure B.2 shows the comparison between the original wave (blue signal) and the one produced by the Arduino who runs this solution (red signal). The wave parameters determined on PicScope software are summarized in the table B.2.
As can be seen, this result is much better than the previous one. The values of frequency, duty cycle, cycle time and high pulse width between the original wave and the generated wave are very similar. In fact, tests show that this way of generating PWM waves allows the control of the motors and driving the vehicle.

Unfortunately, this method cannot be used, because there is a need to use the Arduino’s SoftwareSerial library, to communicate between Arduino and an external device. To perform this communications, the SoftwareSerial library uses the same timers of the ATmega328 (Arduino’s processor) used by the delayMicroseconds() function. When used together, the SoftwareSerial communication and the delay function interfere with each other. The communications’ quality fails, and the PWM wave gets deteriorated. This deterioration is so much bigger that it causes a loss of the car’s control, which at times stops responding to the commands, or performs unwanted moves.

**TimerOne**

In order to solve the hardware limitation about the timers’ use, a new solution is required. The ATMEGA328 processor counts with three different hardware timers: Timer0, Timer1 and Timer2. Each one of these timers has two compare registers that allows the control of the PWM width on the timer’s outputs. For the same timer, the two outputs will normally have the same frequency but different duty cycles are supported. The timer’s frequencies are a submultiple of the system clock frequency (16MHz) and can be set to manipulate the chip’s timer registry directly. Beyond changing the frequency, it allows
a choice of different wave modes, generates interruptions and overflows. This method provides more control than previous methods. To get more information about that, please see the documentation [17].

A TimerOne library provides a collection of routines that allows to easily configure Timer1. The \texttt{pwm()} function, on TimerOne Library, receives two arguments:

\texttt{Timer1.pwm(A,B)}.

This function generates a PWM signal on pin A with a \textit{dutyCycle} equals to B. Pin A can only be pin 9 or pin 10, and B represents values scaled between 0 (\textit{dutycycle} = 0\%) and 1023 (\textit{dutycycle} = 100\%).

The TimerOne library has been developed just to get a faster and easier way to set a PWM period or frequency, but today many other features have been implemented. To get more information about TimerOne library, please consult the documentation [19]. This library is not provided with the original Arduino software, but can be installed from [20]. To install the library, please download a .zip file, open the Arduino software and go to sketch menu. Choose the option include library and add a ZIP Library.

The results of the implementation of this method are presented in section 3.4.1. However, it will be repeated here for convenience in figure B.3.

![Figure B.3: Comparison between the original PWM wave, and the Arduino Uno emulated PWM wave using TimerOne library.](image)

The wave parameters determined on PicScope software are summarized in the table B.3.

<table>
<thead>
<tr>
<th></th>
<th>Arduino signal (red wave)</th>
<th>Original signal (blue wave)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency</td>
<td>99.96 Hz</td>
<td>101.6 Hz</td>
</tr>
<tr>
<td>Cycle Time</td>
<td>10000 ( \mu \text{s} )</td>
<td>9846 ( \mu \text{s} )</td>
</tr>
<tr>
<td>Duty Cycle</td>
<td>15.16%</td>
<td>15.21%</td>
</tr>
<tr>
<td>High Pulse Width</td>
<td>1526 ( \mu \text{s} )</td>
<td>1505 ( \mu \text{s} )</td>
</tr>
</tbody>
</table>

The result of TimerOne library applications is as good as the previous one, where the \texttt{delay()} function is used to set the cycle time (or frequency) of the PWM signal. Otherwise, the implementation has a reduced complexity as in the \texttt{analogWrite()} function's method. So the TimerOne library provides the best side of the two previous discussed methods, with the advantage that the hardware PWM signs are generated by a dedicated timer, what means they are interference proof. It is a very important feature to
the final work, as this signals will be used to actuate the vehicle’s motors. However, there is an important limitation: once the TimerOne library only works with pins 9 and 10, it means that only 2 PWM waves can be emulated.

**B.1.2 PWM and Raspberry Pi**

**Pigpio library**

The solution to generate usable PWM wave on Raspberry Pi require the use of the pigpio library. To know how to get this library please see APPENDIX A. The pigpio library is written in the C programming language and counts with a Python module that allow the control of the GPIO. Therefore, the implementations of this library were written in Python. To generate a hardware PWM signal, the python’s block of this library provides the `hardware_PWM()` function. This function is documented in [14]. As arguments, this function receives the output pin, the PWM frequency and PWM duty cycle. To generate a PWM wave on pin 13 with 15.65% duty cycle and 101 Hz of frequency, the arguments must be:

```python
pi.hardware_PWM(13, 101, 156500).
```

Special attention is needed to the duty cycle's value, because it is the percentage multiplied by 10000. It is an important issue that will require a special attention. This was being discussed in section 3.6.1 of the this document. The raspberry Pi's pins map can be consulted in appendix A.2.2. Because there is no need to compare the results of different implementations as in the section B.1.1 the results of the pigpio library application will not be repeated here. This results are displayed on section 3.4.2 of the main document.

**B.1.3 PWM initial values to Arming the Actuators**

It is important that, independently of the adopted solution, the initial wave have a duty cycle of 15%. So, the PWM generators must be initialized with a 1487 $\mu$s of High Pulse Width. It corresponds to the situation where no actuation is required. This is a security system of the LaTrax’s ESC, that ensures that the car does not move unexpectedly when it is turned on. In fact, if the initial value was not the required value, the ESC does not activate the power motor, so the car will not move. In this case, the ESC’s led indicator blinks green. Once the initiation value was detected by the ESC, the led remains a solid and red colour. In this case, the vehicle is ready to roll.
B.2 UDP Communication

B.2.1 Arduino Uno

Wifi Shield and SoftwareSerial

To use the TinySine WIFI shield, the Software serial library is required. To use this library, a Software Serial Object must be created with the function

\[
\text{SoftwareSerial ObjectName(PinA, PinB),}
\]

where PinA and PinB are digital pins that emulate the TX Serial Pin and the RX Serial Pin, respectively. In the Arduino setup() function, the object must be initialized:

\[
\text{ObjectName.begin(baud rate).}
\]

At this point, it is important to make sure that the baud rate required is the same as in the settings of the shield board (please see point 11 of the configuration process of section A.1.3). In case of need to use the Arduino Serial Port, or the Arduino Serial Monitor, their baud rate must be the same as that of the software serial object. Now the Serial Object can be easily manipulated just using the functions

\[
\text{ObjectName.print(msg),}
\]

\[
\text{msg=ObjectName.read().}
\]

The use of the same SoftwareSerial object to send and receive UDP messages with the TinySine Wifi Shield, can lead to an unexpected result, because the same SoftSerial port is used to listen all the time, even when the port tries to outputs the IMU’s data. As so, sometimes, some of this data is read from the SoftSerial port before they can be transmitted. It results in a double problem, because the IMU’s transited data was incomplete, and the system receives wrong actuation commands which may degenerate in unpredictable results.

To make sure that no information is lost, one of the limitations of the SoftwareSerial library was taken as an advantage. As it is mentioned in section 3.4.1, this library supports multiple software serial ports, but only one can listen at time. So, the adopted solution pass by creating an auxiliary port. When there is a need to receive data on the WifiBee SoftSerial port (to receive the actuation commands), this port was putted to listen. When there is the intention of communicating the IMU’s data through the WifiBee’s SoftSerial port, the auxiliary SoftSerial port was putted to listen. This way, the first port is free to send the data without any loss or interferences.

B.2.2 Raspberry Pi

Because the Raspberry Pi already supports Wi-Fi communication, no external hardware is needed to guarantee the UDP communication. The socket library inbuilt in python, is a powerful tool that appropriately handles communication issue. The only consideration to do is the need of two sockets, one to
be used in the communications’ input and another for the communications’ output. The bind function relates the socket with a local IP and tells the socket to listen to a user-defined local port.

B.3 Simulink \textit{scaleChange}

Because there is a need to change from the input scale (from the input block) to another scale capable of being communicated and that is useful to actuate the vehicle’s motors, the \textit{scaleCange} block was developed. The scales to consider are presented in the Table B.4.

| Ranges | PWM Duty Cycle | 0% | ... | 10% | ... | 15% | ... | 20% | ... | 100% |
|--------|----------------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| Arduino Actuation* | 0 | ... | 102 | ... | 153 | ... | 205 | ... | 1023 |
| Raspberry Pi Actuation** | 0 | ... | 100 000 | ... | 150 000 | ... | 200 000 | ... | 1 000 000 |

\textit{Admissible Actuation Zone}

* input argument of PWM generator function = $1023 \times \frac{\text{duty cycle}[\%]}{100}$

**input argument of PWM generator function= duty cycle[\%] \times 1000

The scale change is easily obtained through the line equation. For this, consider that the x-axis is the user interface domain $[-1; 1] \in \mathbb{R}$, and that the y-axis is the UDP commands domain. Let’s consider the Raspberry Pi scale, so its domain is $[100; 200] \in \mathbb{N}$. The conversion between the two scales are given by the line equation,

$$y = mx + b.$$ \hspace{1cm} (B.1)

From the two known coordinate points in figure B.4

$$P1 = (-1; 100) \text{ and } P2 = (1; 200),$$ \hspace{1cm} (B.2)

it is possible to deduce the result:

$$y = 50x + 150.$$ \hspace{1cm} (B.3)

The scale change is guaranteed just by applying this equation to the output value of user interface (x-value). With the same procedure, it is possible to obtain a similar result to the Arduino architecture:

$$y = 51.5x + 153.5.$$ \hspace{1cm} (B.4)
Figure B.4: Line equation used to convert the two input scales for the Raspberry Pi solution.

**B.4 IMU Operation**

**B.4.1 Arduino Uno**

To be able to communicate with the IMU via I\(^2\)C. The Wire library is provided by the Arduino IDE to allow this kind of communication. No supplementary installations are required. Each individual sensor inside the IMU has a unique I\(^2\)C address. To send/receive data to a specific sensor, the *Wire Transmission* must be initialized for this specifies address. The table B.5 shows the address map for each sensor.

<table>
<thead>
<tr>
<th>Sensor</th>
<th>Address [Hex]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accelerometer</td>
<td>0x1E</td>
</tr>
<tr>
<td>Gyroscope</td>
<td>0x53</td>
</tr>
<tr>
<td>Magnetometer</td>
<td>0x69</td>
</tr>
</tbody>
</table>

Every single sensor has a set of registers that can be read and/or written. It is by writing in some specific registers that it’s possible to configure each sensor, and is by reading other registers that it’s possible to get the sensors’ measurements. Because all registers of all sensors, their functionality and addresses are already listed in the sensors data-sheet, it will not be done here. However, and despite of some technical errors, the sensors’ data-sheets are important documents to consult when a use of the GY-80 is required.
To write/read a register, a set of functions is needed. The pseudocode shown below helps to understand how to do it:

**Algorithm 1 write in a Register**

1: procedure WRITE REGISTER(SSensorAddress, RegisterAddress, val)
2: beginTransmission (SensorAddress)
3: write (RegisterAddress)
4: write (val)
5: endTransmission ()

**Algorithm 2 Read from a Register**

1: procedure READ REGISTER(SensorAddress, RegisterAddress)
2: int val
3: beginTransmission (SensorAddress)
4: write (RegisterAddress)
5: endTransmission ()
6: requestFrom (SensorAddress, 1)
7: val = read ();
Return: val

**Algorithm 3 Get Sensors Measurements**

1: procedure GET VAL (SSensorAddress, MsB_registerAddress, LsB_registerAddress)
2: byte axis_MsB_sensor = readRegister (SensorAddress, MsB_registerAddress)
3: byte axis_LsB_sensor = readRegister (SensorAddress, LsB_registerAddress)
4: measure = ( (axis_MsB_sensor << 8) | axis_LsB_sensor )
Return: measure

Both the values read and the values written are in a binary base. The registers’ and sensors’ addresses can be used in a binary base or in a hexadecimal base. For each physical quantity, the measured value is output by two registers, one for the LsB and another to the Higher Significant Bits Most Significant Bit (MsB). This last value must be 8 bits shifted left, and the result must suffer a logical disjunction with the LsB value. At the end, it must be converted in an integer. To simplify the understanding, the respective pseudo-code in shown above.

This kind of sequence must be performed so as to get a single value from a single measure, from a single axis, from a single sensor. It must be repeated for all the three sensors and their three DOF, meaning that eighteen registers must be read to get all the sensors’ measurements.

**B.4.2 Raspberry Pi**

For the Raspberry Pi solution, the chosen implementation programming language was python. An advantage of this, is the multiple open source libraries available on the web. Fortunately, there are a library available to use the GY-80 IMU, that makes the implementation easy, but less efficient.

This library is able to get the IMU measurements, convert them into physical units and process them to get additional information. Despite this information the potentially usefulness for this work purposes, some modifications of the original library are needed. As there is a need to keep the system universal to potentiate the most diversified uses, it was defined that the output of the instrumentation subsystem
must be the sensors raw data. It will allow the user to process data in the most convenient way to attend their own purposes. To know how to get the library, please see section A.2.