Universitat Politècnica de Catalunya

Escola Tècnica Superior d'Enginyeria Industrial de Barcelona

Industrial Engineering Degree

Final Degree Project

Inertial stabilization system based on a Gough-Stewart parallel platform

Aitor Ramírez Gómez

Descriptive Report

Director:
Dr. Federico Thomas

Codirector:
Dr. Antonio Benito Martinez

January 2017

Inertial stabilization system based on a Gough-Stewart parallel platform

by Aitor Ramírez Gómez

This project has been developed at *Institut de Robòtica i Informàtica Industrial (IRI, CSIC-UPC)* and presented at *Universitat Politècnica de Catalunya* in order to achieve the Industrial Enginyeering Degree title.

Director:
Dr. Federico Thomas

Codirector:
Dr. Antonio Benito Martinez

ETSEIB representative director:
Dr. Manuel Moreno-Eguilaz

Inertial stabilization system based on a Gough-Stewart parallel platform

Aitor Ramírez Gómez

Abstract

The goal of this project is to develop a system able to control the balance of a parallel platform robot. The project includes 1) the Gough-Stewart parallel robot, created and assembled at *IRI*'s lab; 2) 6 *Dynamixel™* rotary actuators, to move the platform; 3) and *Arduino®* *UNO* electronic board, to control the rotary actuators and to process data provided by 4) an inertia measurement unit *MPU-6050™* sensor.

In this project it is described how the communication between the devices and the electronic board, which acts as the master, is established. It is specified what kind, and how, the communication is performed between each device in order to send and receive data. As the controller is implemented on *Arduino®*, the programming language required is *Processing*.

Given the dimensions of the prototype, the project is not focused on the analysis of the involved forces, but in its kinematics. Therefore, this work also details the algebraic manipulations needed to set the different configurations that the platform demands according to the kinematics of the parallel robot.

* * *

Acknowledgements

I would like to thank Federico Thomas for letting me join the *IRI's* team to develop this project, and Antonio Benito Martínez; their guidance and support have been essential to achieve the goals of this project. I would also like to extend my sincere thanks to all of the individuals that have somehow contributed to the development of this project. To Patrick Grosch, to whom I am indebted for all the technical support and the assistance provided whenever was needed. To Daniel Castro for all the many headaches we have shared. To Manuel Moreno-Eguilaz for accepting to be my representative director at my school and encouraging me to write these words in English. To Berta del Hoyo for helping me to better express myself in a foreign language. Last, but not least, to my parents for always having the patience I do not sometimes deserve.

Table of Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Goal . . . 2
- 1.3 Scope and Assumptions . . . 2
  - 1.3.1 The Gough-Stewart parallel robot . . . 2
  - 1.3.2 The singularities . . . 4
  - 1.3.3 The Dynamixel™ rotary actuators . . . 4
  - 1.3.4 The MPU-6050™ drift . . . 4
- 1.4 State-of-the-art . . . 5
- 1.5 Thesis structure . . . 8

2 Tools **9**
- 2.1 Block diagram . . . 9
- 2.2 The hardware . . . 11
  - 2.2.1 The IMU sensor: MPU-6050™ . . . 11
  - 2.2.2 The computer board: Arduino® UNO . . . 15
  - 2.2.3 The rotary actuators: Dynamixel™ AX-12+/AX-12A . . . 16
- 2.3 The software . . . 18
  - 2.3.1 Arduino® IDE . . . 18
  - 2.3.2 Matlab® and Robotics Toolbox . . . 19
  - 2.3.3 Dev-C++ IDE . . . 22
- 2.4 The wiring of the elements . . . 23

3 Communication protocols **25**
- 3.1 Communication IMU - Electronic board . . . 26
  - 3.1.1 Inter-Integrated Circuit (I^2C) . . . 26
  - 3.1.2 Matlab® simulation . . . 30
- 3.2 Communication Electronic board - Rotary actuators . . . 31
  - 3.2.1 Instruction Packets . . . 32
  - 3.2.2 Status Packets . . . 33
  - 3.2.3 The half-duplex asynchronous serial communication interface . . . 35

4 Execution and control **39**
- 4.1 The robot . . . 40
  - 4.1.1 Definition, advantages and structure . . . 40
- 4.2 Kinematics of the parallel robot . . . 43
  - 4.2.1 Inverse kinematics solution . . . 45
  - 4.2.2 Problems writing Arduino® code . . . 53
  - 4.2.3 Alternative 1: The Fourier method . . . 54
  - 4.2.4 Alternative 2: The Jacobian method . . . 59
  - 4.2.5 Matlab® simulation of the final solution . . . 62
- 4.3 The control . . . 63
  - 4.3.1 Strategies of execution . . . 63
  - 4.3.2 Architecture of the closed-loop . . . 66
  - 4.3.3 Controller . . . 67

5 Results **69**

6 Environmental impact **71**

7 Conclusions **73**
- 7.1 Contributions . . . 73
- 7.2 Possible enhancements and future work . . . 73

A Budget **77**

B Images and tables' sources **79**

C Parameters from Fourier series **81**
C.1 Surface plots 82
C.2 Fourier approximations 85
C.3 Fourier parameters 88

D Arduino Code **95**

E Computing files **111**
E.1 Main execution file 112
E.2 Jacobian file 113
E.3 Inverse kinematics file 114
E.4 Solve angles file 116

List of Figures

1 The Gough-Stewart parallel robot . . . 3
2 Application example of a gimbal system . . . 6
3 Block diagram of the proposed solution . . . 10
4 The MPU-6050™ from *InvenSense* . . . 11
5 MPU-6050™ axes and rotations . . . 12
6 MPU-6050™ block diagram . . . 13
7 The Arduino® UNO electronic board . . . 15
8 The Dynamixel™ AX-12+ rotary actuator . . . 16
9 The placing of the actuators and its linking in a daisy chain type connection . . . 17
10 Goal position settings in joint mode . . . 18
11 The Arduino® Integrated Development Environment (IDE) . . . 19
12 Wiring diagram . . . 23
13 I^2C block diagram . . . 27
14 I^2C protocol . . . 28
15 Example of interaction between Arduino® and MPU-6050™ . . . 29
16 Rotation cube simulation according to the Roll and Pitch values . . . 30
17 Instruction packet and Status packet's paths . . . 31
18 Structure of the Instruction packet . . . 32
19 Structure of the Status packet . . . 33
20 Diagram of the communication driver . . . 36
21 Dynamixel™ 's pins . . . 36
22 74F244 driver . . . 37
23 Some Gough-Stewart platforms from IRI's lab . . . 40
24 Structure of the parallel robot's manipulators . . . 42
25 Representation of an arbitrary transformation . . . 44
26 Absolute reference frame . . . 45
27 Platform's reference frame . . . 46
28 Joints' reference frames . . . 48
29 Rotors' reference frames . . . 49
30 The 6-$\underline{R}US$ manipulator . . . 50
31 Front and side view of the serial chain representing the manipulator's legs . . . 51
32 ith joints' reference frame with respect the ith rotor . . . 52
33 Relationship between Roll-ϕ, Pitch-θ and σ, λ angles . . . 55
34 Surface plots of α_i with respect to σ and λ . . . 56
35 Fourier series of each actuator for $\lambda = \frac{\pi}{11}$ . . . 58
36 Pure *pitch* recovery (from left to right) . . . 62
37 Pure *roll* recovery (from left to right) . . . 62
38 Strategy representation for the Fourier method . . . 64
39 Block diagram of the system's closed-loop . . . 66
40 Block diagram of a closed-loop PID controller . . . 67
41 Final experimetnal results . . . 69
42 Surface plot, view 1 . . . 82
43 Surface plot, view 2 . . . 83
44 Surface plot, view 3 . . . 84
45 Representation of the fitted curve at $\lambda=\frac{\pi}{11}$ . . . 85
46 Representation of the fitted curve at $\lambda=\frac{\pi}{22}$ . . . 86
47 Representation of the fitted curve at $\lambda=0$ . . . 87

List of Tables

1 Dynamixel™'s available instructions 32
2 Dynamixel™'s possible execution errors 34
3 Parameters of the *Fourier* series 59
4 Mechanical material for the robot construction 77
5 Electronic material for the robot construction 78
6 Labour costs 78
7 Figures' sources 79
8 Tables' sources 79

Chapter 1

Introduction

1.1 Motivation

Nowadays, robots can be found in the fields of medicine, the military, the industry or even in the household sector. This leads to assume that robotic's field is expanding, involving in many other techno-scientific sectors and reaching, as well, the daily life. Drones, which were only used in the military industry, are proof of this. All this reflects the control we have achieved over robotics to this day.

Therefore, it is a field that expands the accessibility and capacity of the human being in all possible and imaginable directions. Robotics, then, can be studied in all modalities, shaping them in different ways and creating robots of all sizes, appearances, and functionalities; making the analysis and the classification of robots a really diverse task.

This project, which leads the analysis of a 6 degrees of freedom parallel robot as an orientation stabilizer, will be my first little contribution to this huge community.

1.2 Goal

The goal of this project is to build a system able to control the balance of a Gough-Stewart parallel platform robot. The platform must remain in its horizontal configuration, which is parallel to the floor reference, whenever the robot is tilted. To achieve this goal we will work on:

1. An inertia measurement unit (IMU) sensor, able to measure the inertial data needed to determine the platform orientation, or attitude, at any time.
2. An electronic board, to process the data received from the IMU and to send the processed data to the actuators.
3. A platform controlled by a 6-*RUS manipulator* which will set the parallel platform attitude.

1.3 Scope and Assumptions

The scope, as well as the hypotheses that have been considered to develop this project, will be described below.

1.3.1 The Gough-Stewart parallel robot

A Gough-Stewart parallel robot prototype has been chosen for the number of degrees of freedom (DOF) needed to define the kinematic state of the platform. As an isolated solid body in a three-dimensional space, the platform has 6-DOF (3 of translation and 3 of rotation), which means that 6 independent variables are needed to define its *pose*, i.e. orientation and position. For that reason, a Gough-Stewart parallel robot is ideal as it allows both to translate and to rotate the platform due to the 6 degrees of freedom it provides.

This is clearly an advantage. However, restrictions will be found in the domain of the kinematics equations of the robot (section 4.2), that limit the workspace of the platform.

Despite the advantage of providing 6DOF, this project will only work on the reconfiguration of the platform orientation, not on its translation. As a possible extension of the work presented here, it would be interesting to also take into account the translation movement of the platform.

It can be seen that the robot used in this project does not exactly correspond to a *Gough-Stewart* parallel platform, as it actually uses sliding bars instead of rotary actuators. This

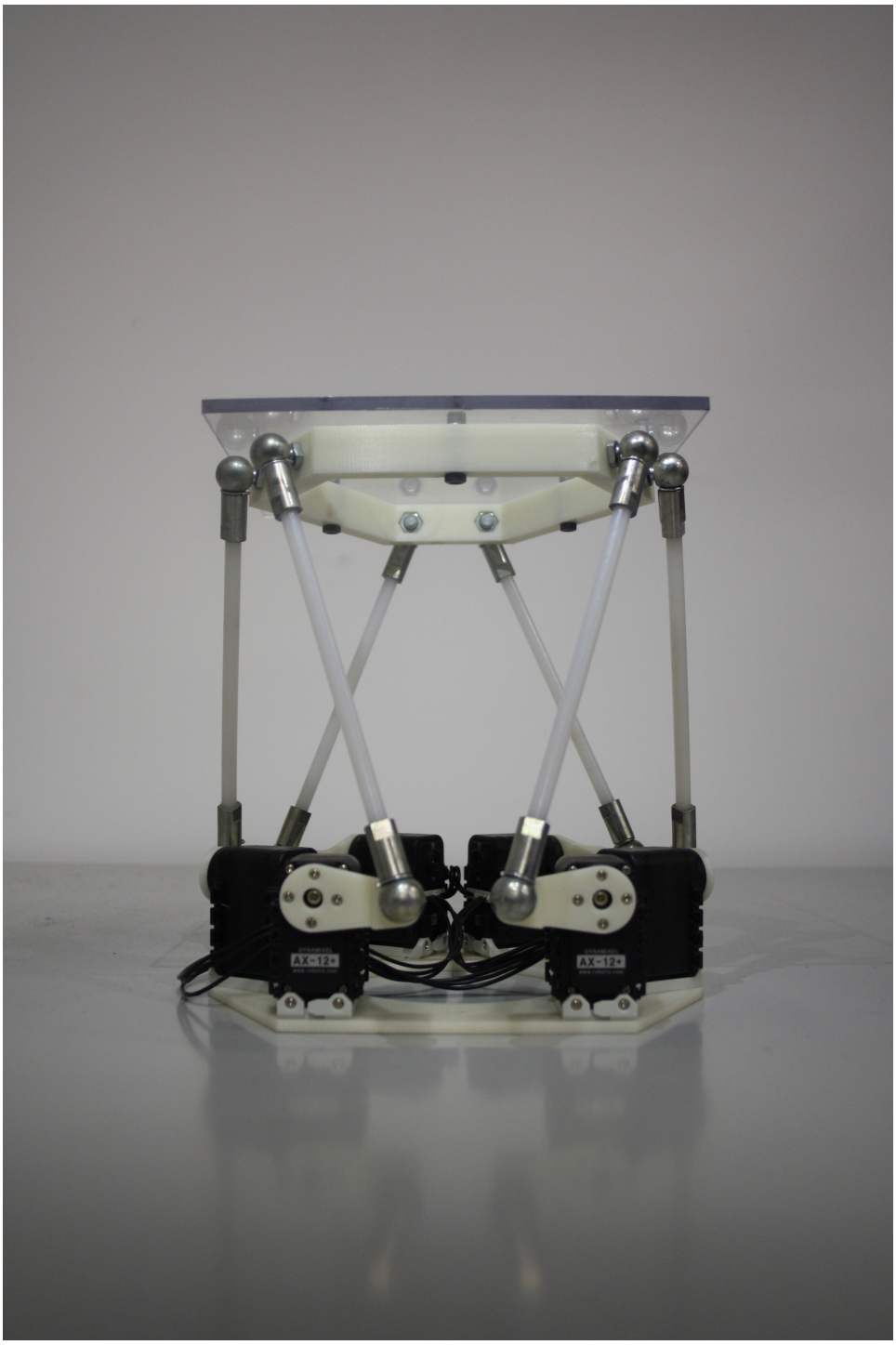

Figure 1: The Gough-Stewart parallel robot

mechanism is called **6-*RUS*** and it makes a slight change in the geometry of the robot that has important consequences in the calculations of its kinematics.

1.3.2 The singularities

Dealing with singularities in a mechanism has always been a problem, as the kinematic performance is drastically reduced and the mechanics can suffer a loss of controllability. Although it is always a desire to get the robot far from those configurations, it is a really complex task to obtain a method able to plan a movement free of singularities.

Two kinds of singularities can be found in a parallel robot; the serial and the parallel singularities.

Regarding the **parallel singularities**, if they exist, they are the most difficult to identify. *J.-P. Merlet* explains in his book *Parallel Robots* [1] how the jacobian of the kinematics of the mechanics in question is used to find them. He also shows the finite particular cases of parallel singularities existing in a Stewart parallel robot. Luckily, the range of configurations of the robot is reduced enough to guarantee that the robot is always far from them.

Regarding the **serial singularities**, they are easier to detect. In our case, they only appear when the arm of a rotary actuator is aligned to the bar attached to the platform. Fortunately, those configurations are easy to avoid.

1.3.3 The Dynamixel™ rotary actuators

The complexity of this project has been highly reduced because of the fact that it does not take into account the forces involved, only its kinematics, i.e., accelerations and higher order derivates are not considered. This is due to the low mass moved by the motors in companion with the rotors themselves after considering the motor reduction. On the one hand, the weight of the different parts of the robot attached to the actuators is so small that its inertial force is much smaller than the force of action of the actuators themselves. This allows to neglect the mass of those parts, and therefore, the forces caused by them. On the other hand, the internal inertia of the *Dynamixel™* rotary actuators is already considered by the internal software of the device.

1.3.4 The MPU-6050™ drift

In this project it is obvious that a device able to somehow track the platform motion is needed. This device is a sensor called Inertia Measurement Unit or IMU, and there are different types of them. As it will be seen, IMUs have independent sensors inside (e.g. accelerometers, gyroscopes or magnetometers) which can work together or separately to get different motion parameters.

Unlike a magnetic compass, the MPU-6050™ , which is the IMU sensor that will be used in this project, does not seek the North pole. When it is used, it slowly drifts away from north. We would need to periodically reorient it using a magnetic compass as an absolute reference. Consequently, one of the parameters the IMU provides, called *yaw*, is not accurate enough to be used it in this project.

1.4 State-of-the-art

The stabilization of an object, such as the platform under study, is becoming a very pursued system due to the improvements it provides to certain device applications. Some of the generic applications of this system can be found in many fields:

- *Audiovisuals*
- *Aerospace*
- *Quadricopters*
- *Industry*

All of these applications have a common denominator in the development of their stabilization system: an *IMU* sensor. An IMU, or Inertial Measurement Unit, is a device which function is to track its own movement through estimations of position, orientation and velocity. Eventually, two different types can be chosen when the mechanical system, in order to recover the object orientation and/or position, needs to be built: the gimbal system and the strapdown system.

Gimbal system

The gimbal systems are composed of an IMU and a series of gimbal actuators. The gimbals are ring structures attached between them. On one end, the gimbal structure is rotated through a rotary actuator and, on the other end, an encoder estimates their relative position.

This system can be useful when accuracy wants to be increased. However, the gimbal structure can suffer from *gimbal lock*[1], which is the loss of a degree of freedom due to the alignment of the outer and the inner gimbals. Besides, it requires a constant recalibration, higher costs and it only provides 3 degrees of freedom, only focused on the reorientation of the object.

[1] In the webside `https://en.wikipedia.org/wiki/Gimbal_lock`, the *gimbal lock* phenomenon is detailed.

This kind of system is extremely used in the aerospace sector, but also in a sector as day-to-day as audiovisuals.

Figure 2: Application example of a gimbal system

Strapdown system

Regarding the strapdown systems, they are the most commonly used for the reduction of complexity of the mechanic structure with respect to the gimbal system, as well as for its versatility since the mechanics are replaced by a programable software.

These systems are based on a microcontroller which is attached to the reference, along an IMU sensor. Besides, this system presents cheaper costs and less power consumption. However, in this case a mechanism that reconfigures the object orientation needs to be built.

This project will be addressed to develop a compensator based on the strapdown system, since the IMU will be placed in the reference of the platform controlled by a robot with 6 degrees of freedom.

The robot

There are several mechanisms capable of orientating and translating an object. They could be classified in two great sections: the **serial robots** and the **parallel robots**.

The difference between them lies in the kinematic chain that relates the different joints of the mechanism. If the kinematic chain is arranged in series, the change in one manipulator orientation influences the rest of the manipulators. Otherwise, if the kinematic chain is arranged in parallel, the behavior of one manipulator does not affect the rest of them.

The reasons that led the project to choose the parallel control are exposed later on, in section 4. However, there also exist a lot of types of parallel robots, classified by the degrees of freedom they have, and the mechanism which provides them. Depending on the application, there are 3-DOF, 4-DOF, 5-DOF or 6-DOF which movements are performed by different kinds of manipulators. The type of these manipulators could be considered as a subsection to classify the final robot.

Then, the robot prototype used in this project is a 6-DOF parallel robot, with 6-$\underline{R}US$ type of manipulators, with a strapdown system implemented.

1.5 Thesis structure

The rest of this report is structured as follows:

- In Chapter 2, it is explained which tools have been used to achieve the proposed goal. There, the connection of all the integral elements to have a general view of the project and the steps that will be followed in the next chapters are showed. It is also explained the utility of the handled hardware devices, and how they work; The software employed to develop the main code is also detailed, as well as simulations and additional computations.
- In Chapter 3, it is explained which kind of communication each device requires and how they interact with the Master, an Arduino® UNO electronic board. The specific connections of each element will be shown, and the different kinds of simulations realized to check the connectivity between them and to familiarize oneself with the use of these devices.
- In Chapter 4, all the necessary mathematical deductions needed in the project are explained. This includes the inverse kinematics of the parallel robot, simulations of the platform movement carried out in Matlab®, simplifications using *Fourier* transformations as an alternative of the implementation of the inverse kinematics, and how the Jacobian transpose of the inverse kinematics is finally used to set the movement of the robot. It is also exposed the adopted control strategy and how it has been adjusted.
- In Chapter 5, some images of the final experimental results are shown and described.
- In Chapter 6, the environmental impact of this project is mentioned.
- In Chapter 7, conclusions are drawn, as well as the contributons of this project and possible improvements for the future.

This report has also four appendixes which are: the budget of the project, the images and tables' sources, the files of the simulations and the calculations used in Matlab® and the implemented code.

Chapter 2

Tools

Once the goal of the project is defined and the state-of-the-art is revised, it is possible to think about which elements will be needed to develop the project. The final used elements will be slightly described below and essentially distinguished in two categories: the hardware and the software.

The work experience of the technicians in *IRI*, their availability of the license permissions and their contact with suppliers made decisive the election of the devices and the programs that were finally used in this project. Needless to say, different alternative elements and methods were tested before taking a choice.

2.1 Block diagram

Figure. 3 presents an outline of the entire system where the role played by each device is and the data exchanged between them is shown.

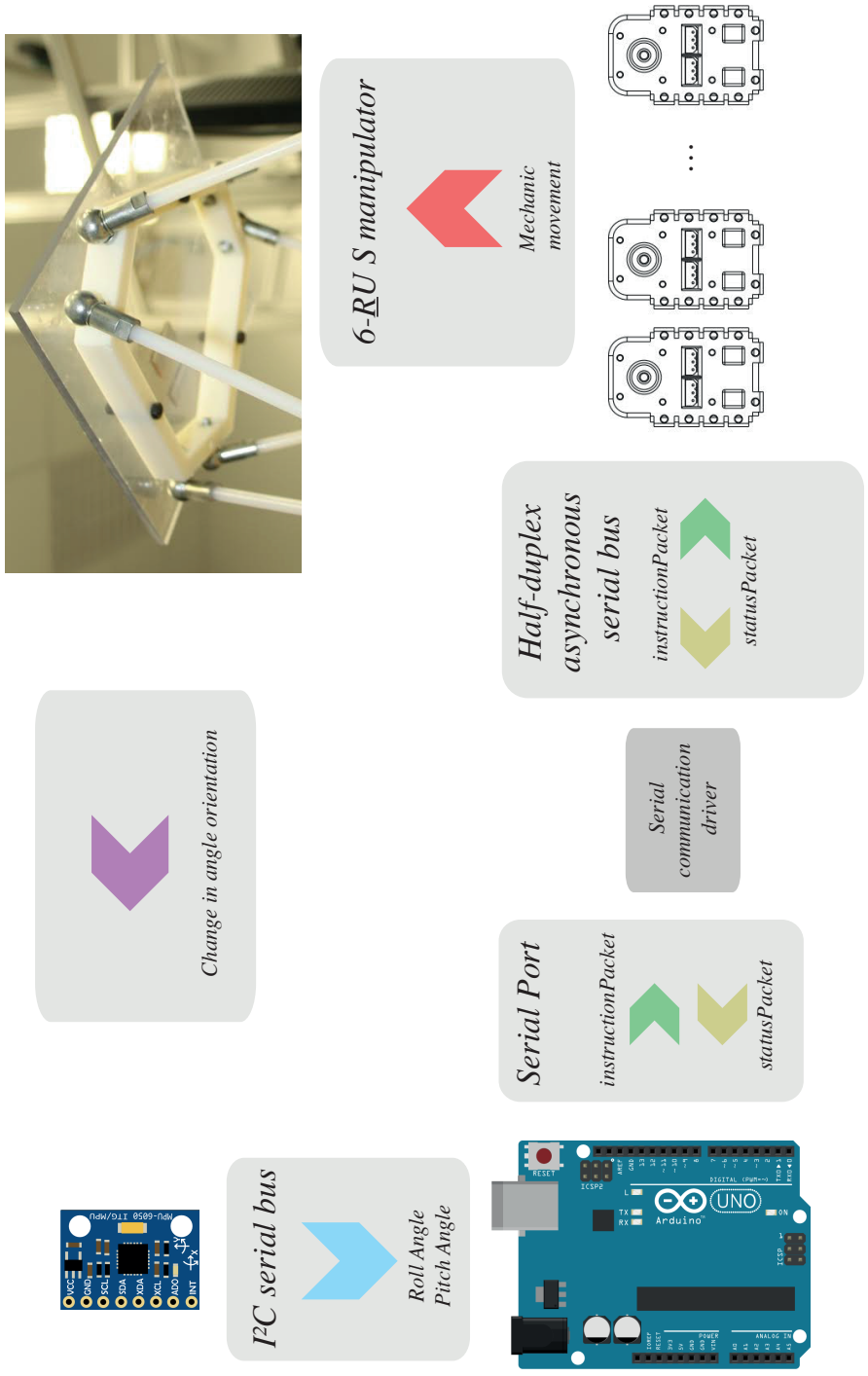

Figure 3: Block diagram of the proposed solution

2.2 The hardware

This section is divided according to the different elements involved in the system. These elements are: an IMU sensor, which takes information from the environment; an electronic board, which processes the data from the IMU; and a set of rotary actuators, which execute the convenient movements of the platform according to the processed data.

2.2.1 The IMU sensor: MPU-6050™

Inertia Measurement Units, or IMUs, are sensors able to track their own movement and/or to detect their own orientation thanks to an accelerometer, a gyroscope, a magnetometer and/or pressure sensors which react to physical motion parameters. This type of sensor is the most commonly used nowadays in all kinds of electronic gadgets and devices, e.g. smartphones, game controllers or even wearables, and it is not strange to also find them in the robotics field, as they have a prolific number of applications.

There are lots of different IMU sensors. However, in this project, the Motion Processing Unit™ MPU-6050™ from *InvenSense* (Fig. 5), which is the most commonly used for its reliability and accuracy, apart from being significantly cheap, is implemented in a GY-512 breakout board.

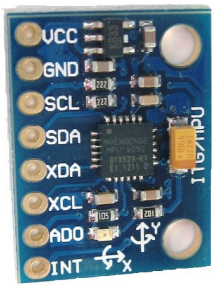

Figure 4: The MPU-6050™ from *InvenSense*

MPU-6050™ is a **6DOF** IMU, which means that it has an **accelerometer** and a **gyroscope** incorporated, both with 3 axes; and combines them with a 9-axis Digital Motion Processor™ unit (DMP™). The function of the DPM™ is to do complex calculations with the sensors' values directly on the chip, which reduces the computational load of, in this case, the Arduino®'s microcontroller. The interest of having a 9-axis DPM™ implemented is because the MPU™ has the possibility to access to an external magnetometer or other sensors through an auxiliary bus explained in section 3.1.

The **gyroscope** measures angular rates ($\dot{\psi}$, $\dot{\theta}$, $\dot{\phi}$) expressed in degrees per second[2] (dps). Then, angle of travel (ψ, θ, ϕ) can result from integrating the gyroscope values with respect to time. This can be used to measure simultaneously **roll, pitch** and **yaw** values in order to track changes in orientation. However, errors in bias estimation or integration must be taken into account, as the tracking of the relative movement is independent from the gravity and it results in an inherent error or *drift*.

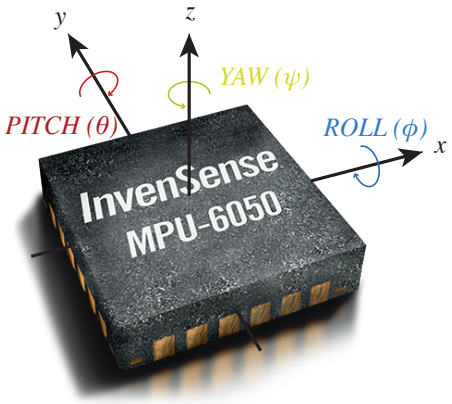

Figure 5: MPU-6050™ axes and rotations

The **accelerometer** measures accelerations, i.e., acceleration components caused by the device motion and acceleration caused by gravity; and these measurements are usually expressed in Gs, which are multiples of the Earth's gravitational force ($1G = 9{,}8\frac{m}{s^2}$). Therefore, the accelerometer is able to determine the static device orientation comparing the gravitational force projected in each axis. Nevertheless, the device orientation during complex motion periods is harder to calculate, as the signal mixes the summation of linear acceleration, centripetal acceleration and gravity.

In addition, the *Motion Process Unit*™ contains a temperature sensor used to measure its die temperature to compensate for error due to temperature changes.

[2]The main characteristics of the MPU-6050™ are described in `https://store.invensense.com/Datasheets/invensense/RM-MPU-6000A.pdf`

The sample rate of these sensors is programmable from 8000 down to 3.9 samples per second, depending on the time required to process the measurements. The latests measurements data coming from those sensors can be directly read in their data registers but, if their analysis and their treatment takes too long, some of the following measurements could be discarded and not analyzed. To minimize this problem, the *MPU-6050*™ contains a 1024-byte FIFO accessible buffer. It works as a memory space where data can be temporary stored. Its main function is to prevent the system from running out of data in asynchronous broadcasts or due to the system process speed. It can be configured to determine which data shall be stored, and which not, but it must be said that an interruption routine should be implemented to warn the system that the FIFO has new data ready to be read. This warning is performed through the *INT* pin of the *Motion Process Unit*™. Finally, FIFO can suffer from overflow which is an important issue to take into account later on.

All these elements interact with each other as shown in Fig. 6.

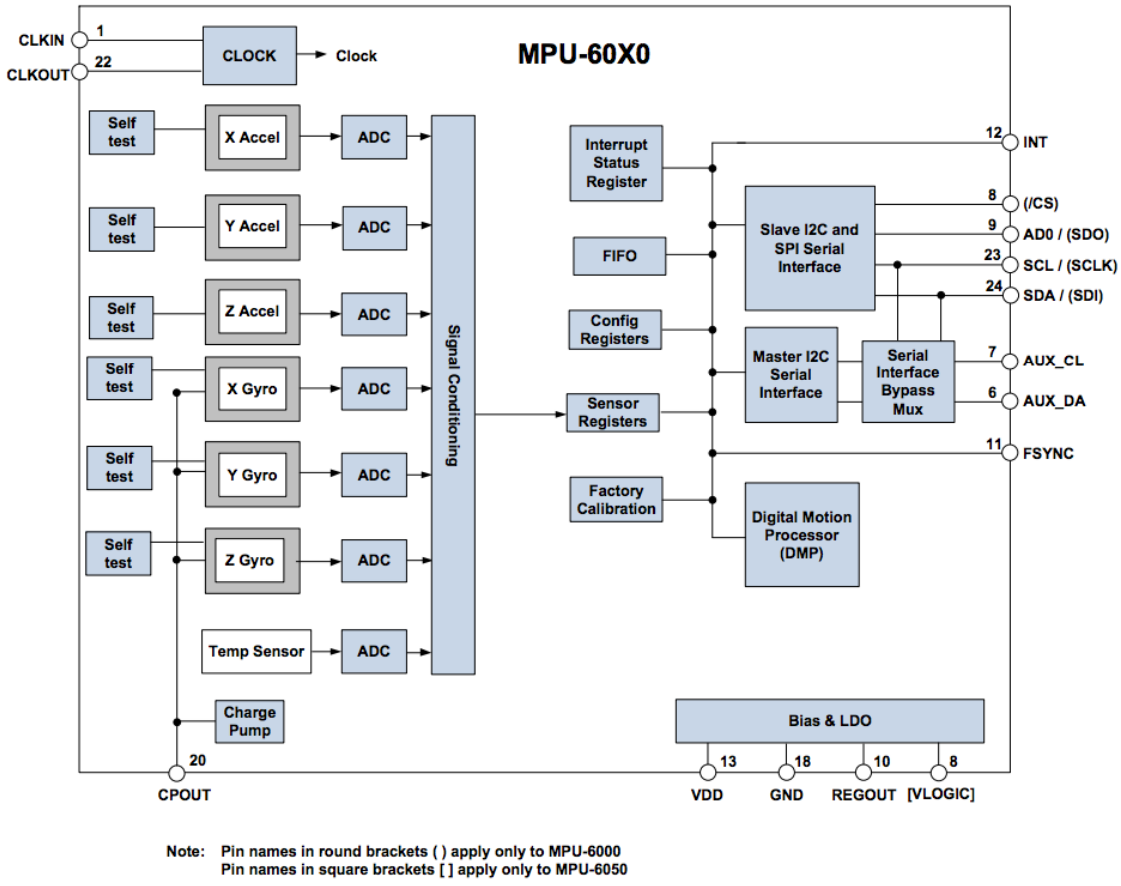

Figure 6: MPU-6050™ block diagram

The problem of tracking the orientation provided by the MPU™ is a complex task already solved in *Motion Sensors Instruction*[3], where *InvenSense* company provides a step-by-step explanation on how this can be achieved through a combination of the different sensors. This method is called **Sensor Fusion**, and its goal is to calculate a quaternion from which the orientation, gravity vector, Euler angles, etc; can be derived.

Regarding some basic characteristics, this IMU usually operates at **3.3V** but some of them, like the one used in this project, can also work at **5V**. The communication protocol used in this IMU is the **Inter-Integrated Circuit ($\mathbf{I^2C}$)**, that will be explained in Chapter 3. However, some other IMU sensors from *InvenSense* can also work with Serial Peripheral Interface bus (SPI) communication protocols.

There are two different ways to read the orientation angles of the IMU sensor through a few previous calculations: the **Euler angles (ZYZ following Robotics toolbox nomenclature)** and the **Yaw-Pitch-Roll angles (XYZ)**. Both can be used to reach the same attitude of a rigid body in a 3-D space, but different angle values need to be applied to each rotation axis. However, both can suffer from a gimbal lock, which refers to a loss of a degree of freedom when some axis align.

Over the time, sensors usually suffer changes in their measurement output due to bias instability and/or gain drifting caused by environmental factors e.g. temperature changes. An **initial calibration** and a **periodic calibration** are required to maintain the sensor performance. The code[4] which provides the offset calibration values taken into account to calibrate the MPU-6050™ is included in the bibliography of this project.

[3]Maths behind the tracking problem are solved in `https://store.invensense.com/datasheets/invensense/Sensor-Introduction.pdf`

[4]Code source: `https://turnsouthates.wordpress.com/2015/07/31/arduino-mpu6050/`

2.2.2 The computer board: Arduino® UNO

Arduino® computer board is an open code platform, low cost, and very extended in the robotics community for medium-level electronic prototypes. The hardware includes an electronic board with a microcontroller and different peripherals such as digital and analog input/output ports, PWM generators, an I^2C communication port, a serial port, etc. However, different kinds of Arduino® boards exist, all of them with their own characteristics.

An Arduino® UNO, as shown in Fig. 7, has been chosen since its technical characteristics are enough to develop this project. However, a more powerful Arduino® board would be needed to improve the results obtained in this project because the size of its RAM could become a restrictive parameter. Indeed, new communicative pins would be needed as they all are already in use. It would also be relevant to consider that Arduino® UNO only has one serial port, which means that it will not be available for checking and printing data to the computer's screen for a significant part of the project as the serial port will be used to communicate with the rotary actuators.

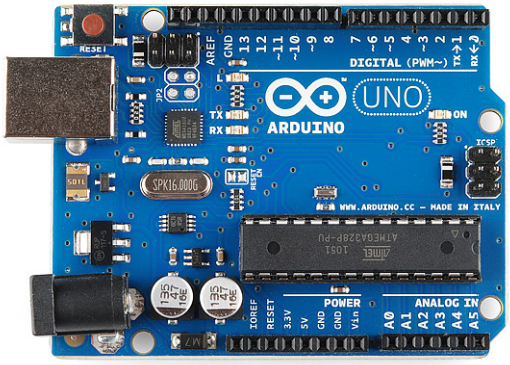

Figure 7: The Arduino® UNO electronic board

The main functions of the Arduino® computer board will be:

1. To communicate with the IMU sensor to get the **pitch** and **roll** of the sensor. These values will be explained with more detail in the IMU - Arduino® communication section (sec. 3.1).

2. To process these two values to get the actuation angles (according to the kinematics of the parallel robot) for the rotary actuators.

3. To communicate with the rotary actuators to send the processed data to accomplish the final function of the parallel robot.

2.2.3 The rotary actuators: Dynamixel™ AX-12+/AX-12A

Leaving the electronics aside, a mechanical system needs to be introduced to complete the final function of the system, which is to make the platform remain in its horizontal orientation. As it has been seen in section 1.3 and in Fig. 1, the parallel robot built at *IRI*'s lab is a variant of the *Stewart* platform as it does not use prismatic actuated joints, but rotary actuators. Then, in order to set the configurations of the platform accurately, a high-performance actuator is needed.

Therefore, a **Dynamixel™ AX-12+** rotary actuator (Fig. 8) from the korean *ROBOTIS* company has been chosen as it is ideal to accomplish the goals of this last stage. The *Dynamixel™ AX-12+* is a high-performance rotary actuator ideal for small robotics applications. However, this model is no longer available in the shop as it has been replaced by its brother the *Dynamixel™ AX-12A*, with exactly the same performance, but with a more advanced external design.

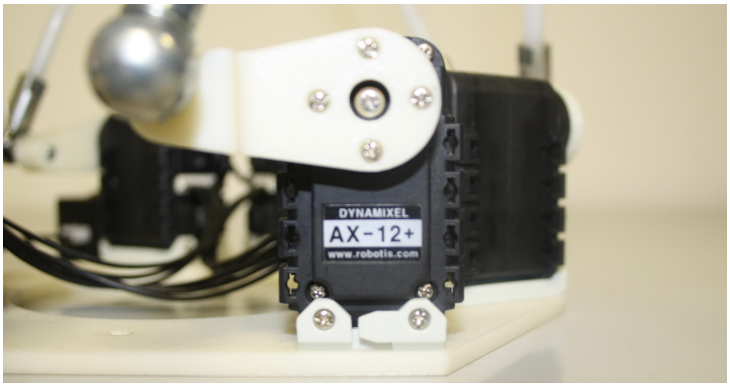

Figure 8: The Dynamixel™ AX-12+ rotary actuator

The *Dynamixel™ AX-12+* has two operation modes: the *joint mode* (thought to set a position of destination between 0^{o} and 300^{o}), and the *wheel mode* (to allow endless turn at a specified speed). Some of its principal characteristics are:

- Resolution: 0.29^{o}
- Potentiometer: 10 bits, 0^{o} - 300^{o} range
- Motor: Cored
- Communication Speed: 7343bps
- Feedback: position, speed, temperature, load, input voltage and current.
- Communication protocol type: half-duplex Asynchronous Serial Communication (8 bits, 1 stop, no parity)

Six *Dynamixels™ AX-12+* will be linked in a *daisy chain* type connection and placed to create the parallel robot prototype shown in Fig. 9. This type of connection has its advantages and its drawbacks, which have to be taken into account when communicating (Chapter 3) with the actuators.

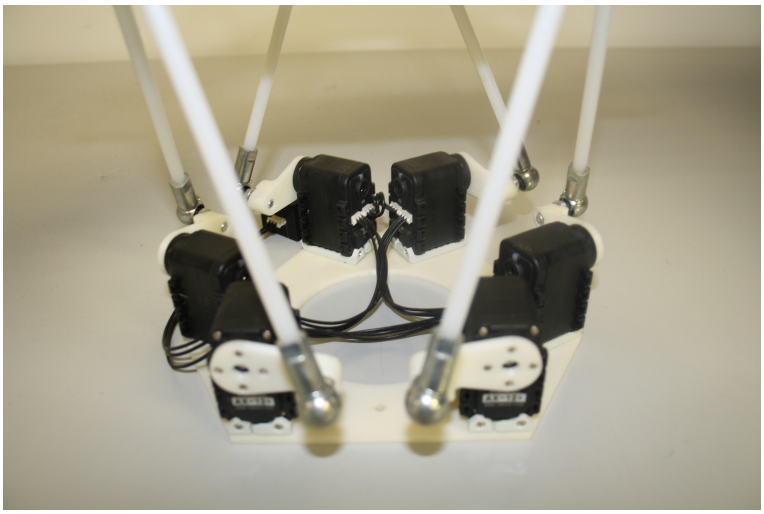

Figure 9: The placing of the actuators and its linking in a daisy chain type connection

The communication with the rotary actuator is established following the *Dynamixel™ communication 1.0.* [5]. It will be done by sending and receiving binary packets called *instruction-Packet* and *Status packet* respectively. All the information regarding the current status and the operation mode of the actuators is available in its control table[6]; and the way to interact with the *Dynamixel™AX-12+/AX-12A* will be reading and/or writing on this table following the protocols that will be explained in Chapter 3.

The motors will work in *joint mode*, which means that the actuators will request the angular positions to move to. The *Goal Position* is a 16 bit data register whose addresses are 0x1E and 0x1F in the Control Table, i.e., the low byte (L) and the high byte (H), respectively. Then, the *Goal Position* value must be decomposed in two bytes (one for each *Goal Position* register), position 511, which is equivalent to 150º (0x1FF in hexadecimal format), should be sent in two bytes: L = 0xFF and H = 0x01. The same example expressed in binary is perhaps more understandable as 00000001 11111111 is easier to separate: the first byte (L) 11111111 which read in hexadecimal format is 0xFF, and the second byte (H) 00000001 which read in hexadecimal format is 0x01.

[5] Dynamixel™ protocol communication is specified in `http://www.trossenrobotics.com/images/productdownloads/AX-12(English).pdf`

[6] Dynamixel™'s control table is shown in its datasheet `http://www.pishrobot.com/files/products/datasheets/dynamixel_ax-12a.pdf`

The possible *Goal Position* values that the *Dynamixels™* can assume in *joint mode* are compressed in a range **from 0 to 1023**. The angular range and the configured angular position settings for each actuator are shown in Fig. 10.

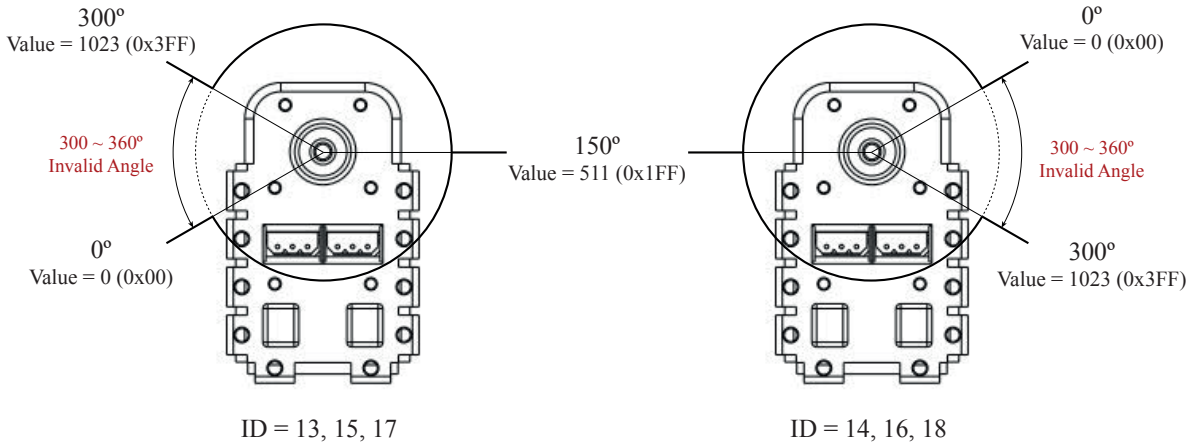

Figure 10: Goal position settings in joint mode

2.3 The software

The usual practice when using an MPU-6050™sensor is to combine it with an Arduino® computer board. This is why an Arduino®'s software has been chosen. Moreover, Matlab® software has been used for simulation for the benefits it provides.

2.3.1 Arduino® IDE

The software of the Arduino® UNO electronic board provides an Integrated Development Environment (IDE) based on the *Processing* language.

Figure 11: The Arduino® Integrated Development Environment (IDE)

As it is seen in Fig. 11, an Arduino® sketch using IDE programmer consists of two main functions: the **setup()** function and the **loop()** function. The *setup* function is used first to initialize variables or to set up the pin modes of the Arduino® board, as the code in this function will only be called once. The *loop* function, unlike the *setup* function, is called repeatedly and it controls the board until it is powered off or reset. Besides these two functions, it is possible to add secondary functions if needed. Therefore, in the *loop* function is where the body of the program is implemented and where all the secondary functions are called.

Finally, the Arduino® IDE allows checking of the written code from possible code syntax errors, and uploading the written code to the Arduino®'s microcontroller.

2.3.2 Matlab® and Robotics Toolbox

Matlab® software is a powerful mathematic tool, with its own programming language, that has been absolutely helpful in lots of different aspects involving this project.

All the mathematical work done, and explained in Chapter 4, has been developed before using this software as the implementation and verification of calculations becomes much faster and easier. Among the basic benefits of the Matlab® software for the development of this project are:

- Matrix manipulation
- Data representation through lineal and superficial graphics using 2D and 3D plots
- Possibility of simulations
- Communication with other hardware devices including *Dynamixels™*

Besides, *toolboxes* can be added, as the **Robotics Toolbox** [2], developed by Peter Corke and used in section 4.2.1. Once the *Robotics toolbox* is downloaded[7] and added to the Matlab®'s path, it is ready to be used. This *toolbox* offers algebraic help for all the matrices transformations needed to set the kinematics in a really simple and understandable way. The common way to express a displacement transformation of an arbitrary 3x1 position vector $\vec{p}$, which involves a translation and a rotation, is:

$$\vec{p_1} = R \cdot \vec{p_0} + \vec{d}, \tag{2.1}$$

where R is a 3x3 rotation matrix and $\vec{d}$ is the translation vector.

However, the *Robotics toolbox* provides a more compact way to express exactly the same operation. The matrices that will substitute the operations of eq. (2.1) are called **homogeneous transformations (T)** and have 4x4 dimension. Through these matrices, the general transformation operations (eq. 2.1) would look like as follows:

$$\vec{p'_1} = T \cdot \vec{p'_0}. \tag{2.2}$$

If T is a 4x4 matrix, then $\vec{p'}$ must be a 4x1 vector. They are represented as follows:

$$T = \left(\begin{array}{ccc|c} & R_{3\times3} & & \vec{d_{3\times1}} \\ \hline 0 & 0 & 0 & 1 \end{array}\right) \tag{2.3}$$

$$\vec{p'_i} = \left(\begin{array}{c} \vec{p_i} \\ \hline 1 \end{array}\right) \tag{2.4}$$

Despite the fact that the matrix representing a rotation about any axis can be found, there exist three basic "pure" rotation matrices, each one regarding to each cartesian axes (x, y and z):

[7] In the webside `http://www.petercorke.com/Robotics_Toolbox`, Corke's Robotics toolbox can be downloaded

$$R_x(\theta) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{pmatrix} \tag{2.5}$$

$$R_y(\theta) = \begin{pmatrix} \cos\theta & 0 & -\sin\theta \\ 0 & 1 & 0 \\ \sin\theta & 0 & \cos\theta \end{pmatrix} \tag{2.6}$$

$$R_z(\theta) = \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{2.7}$$

An arbitrary rotation can be decomposed into sequences of the form $R_x \cdot R_y \cdot R_z$. Hence, this sequence can be used to apply a transformation to a $\mathbb{R}^3$ vector frame from its initial attitude to its final attitude. However, to make these rotation matrices (eq. 2.5, 2.6 and 2.7) to be incorporated in a homogeneous transformation, the commands **trotx(θ), troty(θ)** and **trotz(θ)** of the robotics toolbox will be used. Likewise, the command **transl($\vec{d}$)** is used to incorporate the translation vector $\vec{d}$ in a general homogeneous transformation, only involving the translation term (the last column) and R becoming an identity matrix. Summing up,

$$transl([x,y,z]) = \left(\begin{array}{ccc|c} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ \hline 0 & 0 & 0 & 1 \end{array}\right) \tag{2.8}$$

$$trotx(\theta) = \left(\begin{array}{ccc|c} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ \hline 0 & 0 & 0 & 1 \end{array}\right) \tag{2.9}$$

$$troty(\theta) = \left(\begin{array}{ccc|c} \cos\theta & 0 & -\sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ \sin\theta & 0 & \cos\theta & 0 \\ \hline 0 & 0 & 0 & 1 \end{array}\right) \tag{2.10}$$

$$trotz(\theta) = \left(\begin{array}{ccc|c} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \hline 0 & 0 & 0 & 1 \end{array}\right) \tag{2.11}$$

These four operations (*trotx(), troty(), trotz()* and *transl()*) are combined by multiplying them to get a general transformation matrix T, which involves both a rotation and a translation. To reverse the direction of transformation, we only need to invert the matrix or the sequence of product and the sign of their arguments.

There are different ways to face an orientation problem with the *Robotics toolbox*. One can work through *Euler* angles, *Roll-Pitch-Yaw* angles, quaternions, or just implementing the transformations one by one as it has been showed here. However, in all cases, rotations must be applied in the right order as it is known that **the rotation sequence is non-commutative**. Nevertheless, this issue can be overlooked due to the fact that, as explained in section 4.2.4, a sequence of rotation commutes provided that the angles are small enough. In this case, the rotation sequence can be permuted at will.

2.3.3 Dev-C++ IDE

This is a free integrated development environment for programming in C and C++. It is used as a support for the *blind* parts of the project, where communication with the PC from the electronic board is not available and the implemented code in the Arduino®'s microcontroller is first tested in this environment to verify the correctness of the algorithm.

2.4 The wiring of the elements

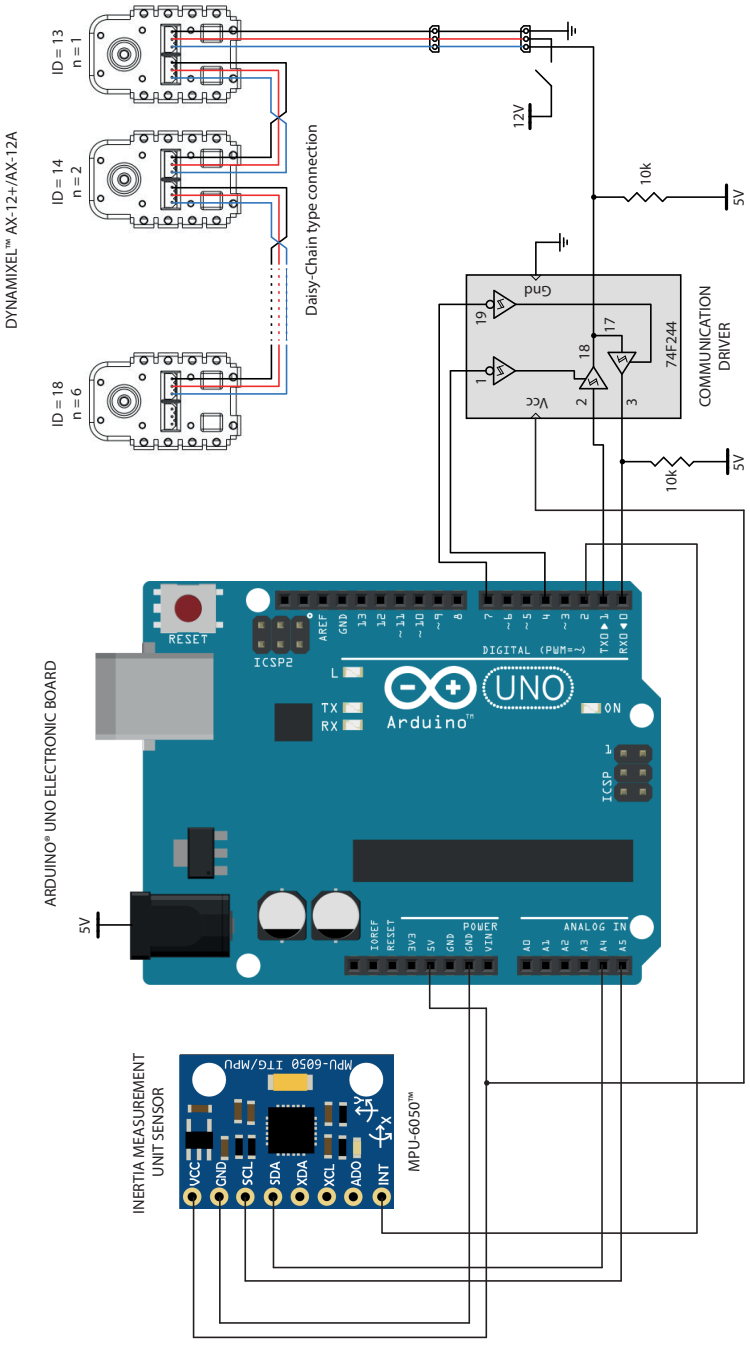

Figure 12: Wiring diagram

Chapter 3

Communication protocols

The way to communicate to a device is always the same, the only thing that changes between the devices is the registers of their "language" and the amount of words in it. In this chapter it is detailed how to understand the "language" of communication established among the devices. It will be explained how those protocols work and which hardware implementations had to be added to achieve the communication according to their protocols.

In the first section, it is examined the interaction between the IMU sensor and the Arduino® electronic board, which is governed by the *InvenSense* communication protocol[8]. In the second section is examined the interaction between the Arduino® microcontroller and the Dynamixel™ AX-12+ rotary actuators, which communication is governed by the *ROBOTIS* communication protocol. Besides, the tests and/or simulations needed to check the communication between the elements is detailed.

All this is explained in the same order that the data follows, which begins in the IMU sensor and concludes in the rotary actuators.

[8] In footnote nº2 (page 12), the MPU-6050™ communication protocol is also specidied.

3.1 Communication IMU - Electronic board

In practice, the code in charge of solving all the problems in section 2.2.1 has already been written by Jeff Rowberg[9], who has done a great job developing a base code able to show readable *yaw-pitch-roll* and *Euler* angles, among others. Moreover, it is also implemented a **filter**, which cleans the raw values, and an **interruption routine**, where other program behavior can be added, such as the code developed in this project. The magnitude of this base code is too large to consider the understanding of it in the scope of this project. Thus, this code will be treat as a *black box*, just looking at how outputs respond to inputs.

Although this code section is not explained, it can be useful to understand how to interact with the IMU sensor from an Arduino® board. The IMU sensor's main task is to convert specific physical parameters into readable binary data which can be processed and analyzed to build up a response. As it has been said in section 2.2.1, the IMU used in this project has two main sensor types: a gyroscope and an accelerometer. Both of them can work separately to get simple data or together (*Sensor Fusion*) to get more complex information.

An Arduino® UNO board provides different kinds of communication protocols: the Serial Peripheral Interface protocol or SPI, the Serial Communication through the serial port and the Inter-Integrated Circuit protocol or I^2C. In Chapter 2 it has already been mentioned the I^2C communication protocol required for the IMU sensor. In the following section it is explained how it works.

3.1.1 Inter-Integrated Circuit (I^2C)

It is time to understand how to access to the internal registers and memory of the *Motion Process Unit™* as all the data of interest is stored somewhere there. The way to access is using I^2C (400 kHz), but it must be reminded that SPI (1 MHz) can work in other *Motion Process Units™* from *InvenSense*.

[9] Rowberg's code source: `https://github.com/jrowberg/i2cdevlib/tree/master/Arduino/MPU6050`

I²C Interface

The I²C interface is a two open-drain and bi-directional connection consisting of the following signals: **serial data (SDA)** and **serial clock (SCL)**. In generalized I²C implementations, attached devices can develop two basic roles, i.e. 1) the master, device which puts an address in the bus in order to interact with 2) the slave device with the matching address, which acknowledges the master.

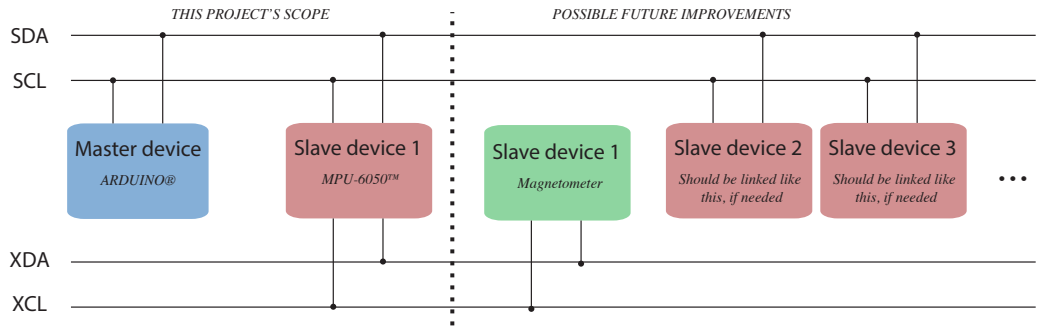

Figure 13: I²C block diagram

As shown in Fig. 14, other slave devices could be attached through the I²C bus if needed to improve the master performance with new data. In addition, MPU-6050™ provides limited capabilities as an I²C master to an optional external magnetometer sensor to improve its own performance. *InvenSense* explains how auxiliary I²C bus works and the operation modes it provides in the MPU-6050™ Datasheet[10]. However, this project does not deal with any of these two cases.

Therefore, MPU-6050™ always acts as a slave device when communicating to the microcontroller, which acts as the master. The SDA and the SCL lines of the Arduino® UNO board are **A4** and **A5 analog pins**, respectively.

The slave address of the MPU-6050™ is 110100X which is 7 bits long. The LSB bit of the 7 bit address is determined by the logic level on pin *AD0*. This allows two *Motion Process Units* to be connected to the same I²C bus. When used in this configuration, the address of the one of the devices should be 1101000 (pin *AD0* is logic LOW) and the address of the other should be 1101001 (pin *AD0* is logic HIGH).

[10]MPU-6050™ datasheet can be found in `http://www.invensense.com/wp-content/uploads/2015/02/MPU-6000-Datasheet1.pdf`

I²C Protocol

What happens on those SDA and SCL lines is governed by the I²C data protocol. If it is looked closely, there are only bits moving through the bus in both directions. Those bits develop their own function depending on which line (SDA or SCL) they are transmitted.

- ***SDA***: This line is where all the exchanged words between master and slave are transferred.
- ***SCL***: This line broadcasts a clock signal which synchronizes the word exchange on the bus. This signal is generated by the master device.

As it has been said before, these two lines are *open-drain*, i.e. pull-up resistors need to be attached so the lines are set to HIGH. That fact is important to understand how data signals are transferred. The following figure shows how a conversation between master and device looks like.

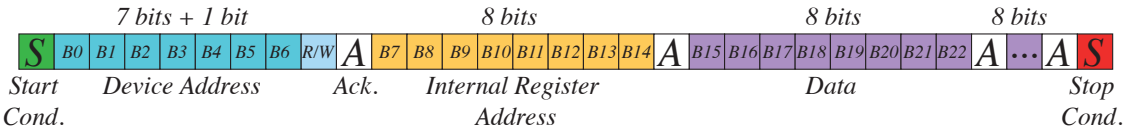

Figure 14: I²C protocol

As seen before, each word in a message is an 8-bit sequence, and each of these sequences refers to a different type of frame. The first sequence (from bit A0 to A7) is the address frame and indicates the slave address which the message is being sent to. The following 8-bit sequences (bit A8 to A15, A16 to A23, and so on) are the data frames, which carry the information. It must be remembered that bit transmissions are carried out most significant bit (MSB) first and the SDA signal should not change when SCL is HIGH to avoid false *Stop Conditions* of transmission.

- ***Start Condition***: To initiate the broadcast, a starting condition needs to befall: the master device must leave the SCL signal HIGH while the sending device pulls SDA signal from HIGH to LOW. When this happens, the master device puts all devices on notice that a transmission is about to start.
- ***Address + R/W***: After the initial warning, the next 7 bits always refer to the device address followed by a R/W bit which indicates if it is a $Read = HIGH$ or a $Write = LOW$ operation.

- ***Acknowledge***: After each 8-bit sequence, it follows a bit called *NACK/ACK*, sent by the receiving device which takes control over the SDA line. If the receiving device either did not receive the data correctly or did not know how to parse it, the *NACK/ACK* bit is set HIGH. Otherwise, if the message was understood by the receiving device it pulls the SDA line LOW, and the control over the SDA line is returned to the sending device to carry on the message.

- ***Internal Address Register***: In most cases, another 8-bit sequence referring to an internal register address of the slave device comes after the first *NACK/ACK* bit. In this case, the internal registers of the MPU-6050™. As it happened before, after this 8-bit sequence it follows a *NACK/ACK* bit.

- ***Data***: After the addressing sequences have been sent, data can begin being broadcasted. As many 8-bit data sequences as they are required are sent until data is transmitted. Between each data sequence the sending device, which at this moment can be either the master or the slave depending on the selected mode in the *R/W* bit, must wait for the *NACK/ACK* bit.

- ***Stop Condition***: To end the broadcast a stopping condition needs to befall: the master device must leave the SCL signal HIGH while the sending device pulls SDA signal from LOW to HIGH.

An example of this kind of communication can be seen more closely in the next figure (Fig. 15), where the Arduino® device asks to read the gyroscope data measurements from the MPU-6050™. Specifically, the master device asks to read the low byte (L) of the gyroscope data measurements from the x axis. Gyroscope's measurements, as well as accelerometer's measurements, are classified in 6 consecutive registers: two for each axis referring to the low byte (L) and the high byte (H) of the data.

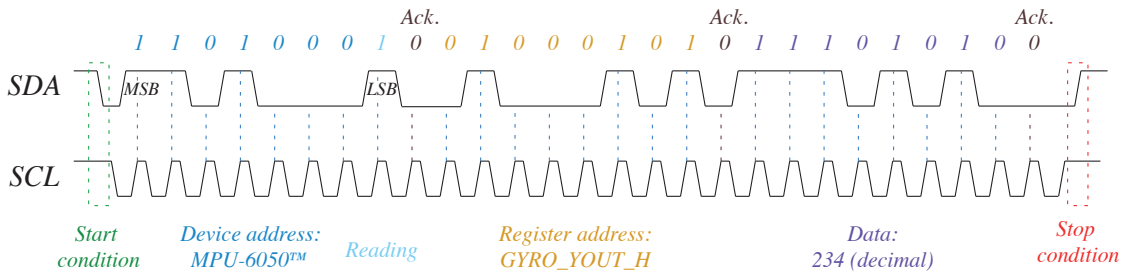

Figure 15: Example of interaction between Arduino® and MPU-6050™

3.1.2 Matlab® simulation

In order to verify that communication between the microcontroller and the sensor, and also to better understand Jeff Rowberg's code, a simple simulation has been developed through the Matlab® software.

The PC must be connected to the Arduino® UNO electronic board through the serial port to be able to read *Yaw-Pitch-Roll* angles, which is the data of interest, obtained using Jeff Rowberg's code. These angle values should be printed in degrees on the *Serial Monitor* of the Arduino® IDE once the code is executed and the communication is correctly performing.

Then, it is time to open the serial port through Matlab® to deviate the data coming from the Arduino® UNO to the Matlab® program. Once it is receiving the information, a simulation can be developed processing and analyzing the information. As the orientation angle values are the data sent by the MPU-6050™, the simulation consists of a red cube reproducing the orientation changes of the *Motion Process Unit™*. The code implemented for this simulation can be found in the appendix.

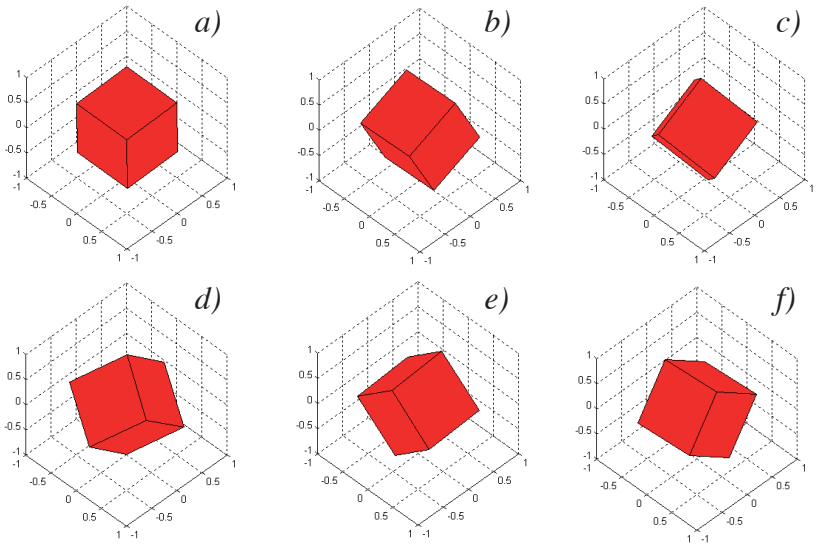

Figure 16: Rotation cube simulation according to the Roll and Pitch values

It must be noted that, at the beginning, **Yaw values were taken into account in the simulation, but then ignored** due to the *drifting* problem (section 1.3.4) could already be witnessed.

Since the microcontroller is now linked to the sensor and to the PC, this simulation could be developed. Later, the serial port will not be available as it will be used for other functions. That will block Arduino® from being able to broadcast data to the PC to print it, which means that the rest of the code will be developed blindly. Here it is where *Dev-C++* software can be useful when writing the code in its IDE first, and looking at how the system responds to similar simulated inputs.

3.2 Communication Electronic board - Rotary actuators

As stated above in the description of this Chapter, the Dynamixel™'s interaction with other devices is governed by *ROBOTIS* communication protocol. The Korean company sets it by sending and receiving encapsulated data packets, called **Instruction packets** and **Status packets**, following the *Dynamixel™ communication 1.0* descriptions.

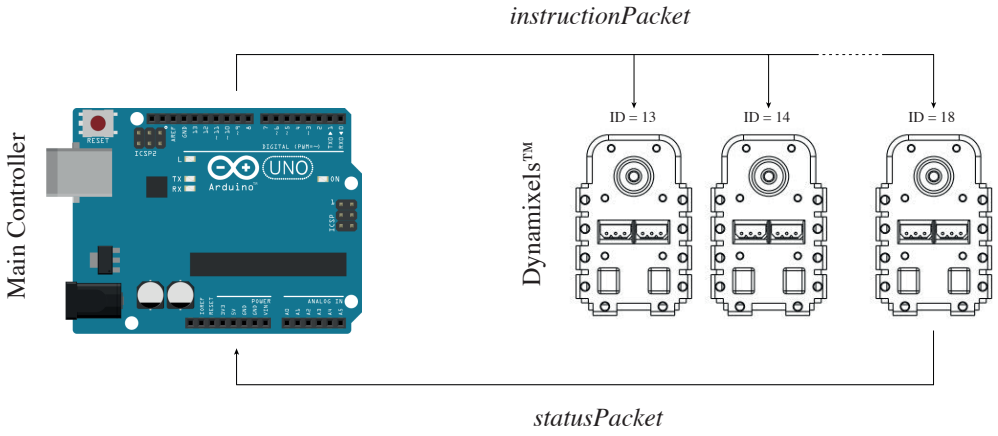

Figure 17: Instruction packet and Status packet's paths

Despite the valuable information that received data packets (*Status packets*) can provide to the system, it was not possible to take advantage of the actuators status data in its maximum performance. The limitations of Arduino® UNO are the only reason why *Status packet* could not be used as it was pretended. However, the two kinds of data packets (sent and received) established in this protocol will be looked closely as the written code offers the chance to work with both of them.

3.2.1 Instruction Packets

Instruction packets are encapsulated data sent by the microcontroller to control the operations of the rotary actuator. It follows the structure shown below.

Figure 18: Structure of the Instruction packet

- ***Heading:*** The value 0xFF indicates the beginning of the instruction. The second value 0xFF is a check value which means that, if both 0xFF are received together, the following values are the content of the data. Remember that 0x refers to the Hexadecimal base.
- ***ID:*** This byte is the rotary actuator identifier value. Each Dynamixel™ has its own ID, i.e. a value between 0 and 253 that identifies them. The *Instruction packet* travels through all the rotary actuators, but it is only executed in the Dynamixel™ which ID matches the ID value of the *Instruction packet*. It should be mentioned that *Broadcast* to all the actuators is possible if ID value is set to 254 in the *Instruction packet*, but no *Status packet* will be received.
- ***LENGTH:*** Refers to the *Instruction packet*'s length, i.e. the number of bytes included in the packet and calculated as "the number of parameters the *INSTRUCTION* needs plus 2".
- ***INSTRUCTION:*** This field determines which operation is going to take place depending on its value. The board can use seven different functions, detailed in table 1, to control the Dynamixel™ behavior.

Values	Parameters	Name	Function
0x01	0	PING	No action. Used to obtain a *Status packet*.
0x02	2	READ DATA	Reading values from the actuator's registers.
0x03	at least 2	WRITE DATA	Writing values to the actuator's registers.
0x04	at least 2	REG WRITE	Writes a value to the actuator without executing.
0x05	0	ACTION	Triggers the action registered by REG WRITE.
0x06	0	RESET	Control Table is reseted to the Factory Default values.
0x83	at least 4	SYNC WRITE	To control many actuators at the same time.

Table 1: Dynamixel™'s available instructions

- ***PARAMETERS:*** As seen before, there can be 0 parameters, 2 parameters, 4 parameters or even more depending on the *INSTRUCTION* that is going to be sent. In case they are needed, the first parameter always refers to the first register address that wants to be read/wrote in the Control Table. Then, the function of the following parameters differs with the *INSTRUCTION* value. If, for example, *WRITE DATA* is going to take place, the second parameter refers to the data that must be written in the starting address sent; the third parameter (if there is one) is the data that must be written in the following address; and so on. If, otherwise, *READ DATA* is going to take place, the second parameter refers to the length of the data to be read from the starting address location sent.

- ***CHECKSUM:*** This value is the result of a method to check if there has been loss of information while sending data. The value is computed as *CHECKSUM* = $\sim$ (ID + LENGTH + INSTRUCTION + PARAMETER 1 + ... PARAMETER N), where "$\sim$" represents the *NOT* logic operation. If the calculated value is larger than 255, the lower byte is defined as the *CHECKSUM* value.

An example of *Instruction packet* which writes data to a Dynamixel™ with ID = 13 should look like this:

0xFF 0xFF 0x0D 0x05 0x03 0x1E 0xFF 0x01

This *Instruction packet* sets the actuator to the position 0x1FF, which angular value is 150º.

3.2.2 Status Packets

Status packets are encapsulated data sent from the Dynamixels™ to the microcontroller once the *Instruction packet* is received. Its structure is organized as follows:

0xFF	0xFF	ID	LENGTH	ERROR	PARAMETER 1	...	PARAMETER N	CHECK SUM

Figure 19: Structure of the Status packet

The *Heading*, the *ID*, the *LENGTH* and the *CHECKSUM* fields are exactly the same as above. Likewise, the *PARAMETERS* field is used if additional information is requested. This information comes from actuators, refers to some of the current status parameters of the actuators, and varies depending on the *Instruction packet* sent. For example, if the internal

temperature of the Dynamixel™ with ID = 13 is to be read, the returned *Status packet* should look as the following.

0xFF 0xFF 0x0D 0x03 0x00 0x20 0xDB

The data of interest is always placed in the *PARAMETERS* field, and the value read is 0x20. Thus, the current internal temperature of Dynamixel™ with ID = 13 is aproximately 32ºC, which is 0x20 in hexadecimal format. However, there is a new field called *ERROR* which differs from the *Instruction packet* ones and also provides important data.

- ***ERROR:*** Some bit of this byte is set to HIGH if an error has been produced during the execution of the *Instruction packet*. Depending on that bit, different errors can be read according to the following table.

Bit	Name	Details
0	Input Voltage Error	Set to 1 if the voltage is out of the operating voltage range defined in the Control Table.
1	Angle Limit Error	Set to 1 if the *Goal Position* is set outside of the range.
2	Overheating Error	Set to 1 if the internal temperature of the unit is above the operating temperature range defined in the Control Table.
3	Range Error	Set to 1 if the instruction sent is out of the defined range.
4	Checksum Error	Set to 1 if the *CHECK SUM* of the instruction packet is incorrect.
5	Overload Error	Set to 1 if the specified maximum torque can't control the applied load.
6	Instruction Error	Set to 1 if an undefined instruction is sent or an ACTION instruction is sent without a REG WRITE instruction.
7	0	–

Table 2: Dynamixel™'s possible execution errors

Implemented code

The following set of functions are programmed and implemented in the code to control the Dynamixels™ from the Arduino® microcontroller. These functions allow to exchange data between the system and the actuators according to the data packet structures just explained above.

- ***void instructionStatus(ID, LENGTH, INSTRUCTION):*** Calls the *instructionPacket* function first and the *statusPacket* function after, giving them the required values if needed. The *PARAMETERS* field of the *Instruction packet* must be filled before calling this function according to the *INSTRUCTION* value.

- ***void instructionPacket(ID, LENGTH, INSTRUCTION):*** Generates a full *Instruction packet* and sends it to the Dynamixels™.
- ***byte statusPacket(void):*** Receives a *Status packet* from the Dynamixels™ as a reply of an *Instruction packet*.

3.2.3 The half-duplex asynchronous serial communication interface

Once the structure of the encapsulated data is understood and a program to build the data packets is developed, it is time to think about how these data bytes are going to be sent from the Arduino® UNO electronic board. As the I^2C bus is required to communicate with MPU-6050™, the Dynamixel™ actuators require a different kind of communication: the **Serial Communication protocol**.

Like its own name indicates, the *Serial Communication* protocol requires an available serial port of the main device which, in this project, is the microcontroller. Arduino® UNO seems to provide 2 serial ports: one through the USB port, another through two special digital pins. Despite both of them are operational, they are actually the same serial port, which means that the two ports cannot be used at the same time. The USB port is implemented then to facilitate the connection with a PC if needed, but the *TX/RX* pins are more commonly used to connect the main device to the rest of the other peripherals.

However, other Arduino® boards such as Arduino® Mega provides more than one serial port available to be used simultaneously. This fact enables the microcontroller to interact with two units through the serial ports allowing, in this case for example, to send and receive data from the Dynamixels™ actuators via one port and showing the status information of the system on the PC's screen via the other port. Adding the PC communication would facilitate the work in a large extent and, in the end, it would become a useful tool to improve the performance and control.

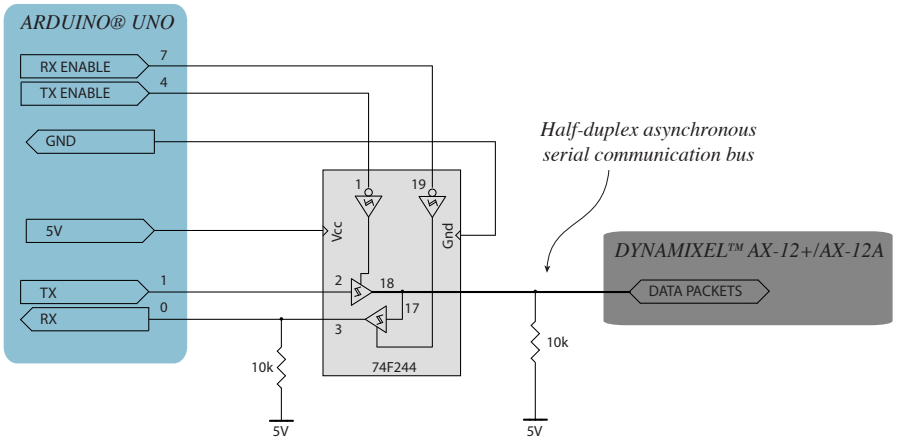

Figure 20: Diagram of the communication driver

In Figure 20, the implementation of a driver to control the actuators from the Arduino® serial port is shown. The function of the driver is to convert the serial port signal which is duplex, i.e. the two communication lines mentioned before (TX/RX), to a half-duplex signal which is a unique communication bus that alternates reception and broadcast function. Therefore, this bus is attached to the first Dynamixel™ data pin, along with the VDD (12V) and GND lines, as shown in Fig. 21. The other 3 pins are linked to the next Dynamixel™ following the *daisy chain* type connection shown in Fig. 12.

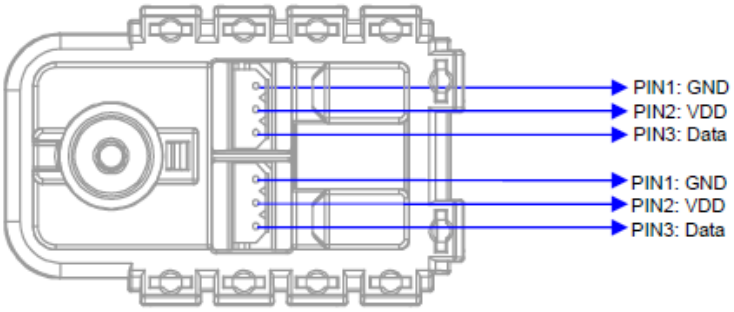

Figure 21: Dynamixel™ 's pins

The communication driver used is the integrated circuit 74F244, an octal buffers/line drivers with 3-state outputs (see Fig. 22). A function has been implemented to control the state of the driver similar to those implemented for *Instruction* and *Status packets*.

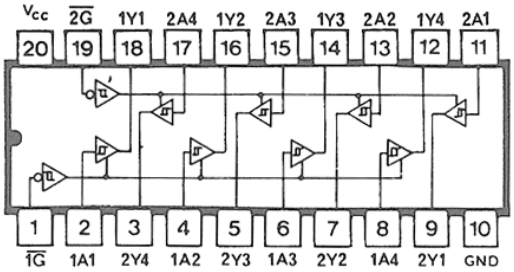

Figure 22: 74F244 driver

Implemented code

- ***communication mode***: This function controls the three states of the driver depending on the parameters entered. The possible parameters are:

  1. *TX MODE*: The function sets the *TX ENABLE* pin to 0 and the *RX ENABLE* pin to 1 if an *Instruction packet* is going to be sent from the Arduino®. This allows the Arduino® board to send the data to the Dynamixels™ through the *TX* pin.
  2. *RX MODE*: The function sets the *RX ENABLE* pin to 0 and the *TX ENABLE* pin to 1 if a *Status packet* is returned from the Dynamixels™. This allows the Arduino® board to receive the data through the *RX* pin.
  3. *HIGH IMPEDANCE MODE*: The *TX ENABLE* and *RX ENABLE* pins are set to 1 if no data is to be broadcasted. This blocks the data traffic through the serial communication bus.

Chapter 4

Execution and control

This chapter addresses the problem of determining the angular value of the Dynamixel™ actuators corresponding to the platform orientation. From now on, the term *revolute joint* will be used to refer to the Dynamixel™ actuators, as well as *end-effector* to refer to the moving platform. The relation giving the actuated joint coordinates for a given *pose* of the platform is referred to as the *inverse kinematics* problem, and this chapter deals with its solution.

Fourier series along with linear interpolations, or the Jacobian transpose method are other alternatives to solve the kinematics problem that will be also examined. This is due to the problems that arise when implementing the corresponding code of the actual kinematics of the robot in the Arduino® microcontroller. Again, the Matlab® software will be of great help to study the different alternatives.

Finally, the closed-loop system of the final solution will be presented, as well as the possible hypothetical implementations to improve the accuracy and performance of the system.

4.1 The robot

The main aim of this project is to control the orientation of a platform, which will be governed by a robot. Due to the nature of this goal, and other aspects that will be introduced in this Chapter, a parallel robot is chosen to develop the task, in particular a *Gough-Stewart* parallel robot. There are a few different types of parallel robots. Nonetheless, the one used works as a 6-*RUS* manipulator [1], already presented in the introduction.

Next, the *Gough-Stewart*, or just *Stewart*, robot with its principal characteristics is introduced.

4.1.1 Definition, advantages and structure

DEFINITION

The *Gough-Stewart* parallel platform is one of the many kinds of parallel robots that have been invented. This particular robot provides 6 degrees of freedom, granted by six independent sliding actuators, which enable the platform to rotate around any rotation axis of its workspace. This previous fact allows the robot to control the compensations of an object that does not need to be necessarily attached on the platform surface. In addition, it also enables to compensate both rotation and translation in any direction which could allow the platform to remain static in the tridimensional space.

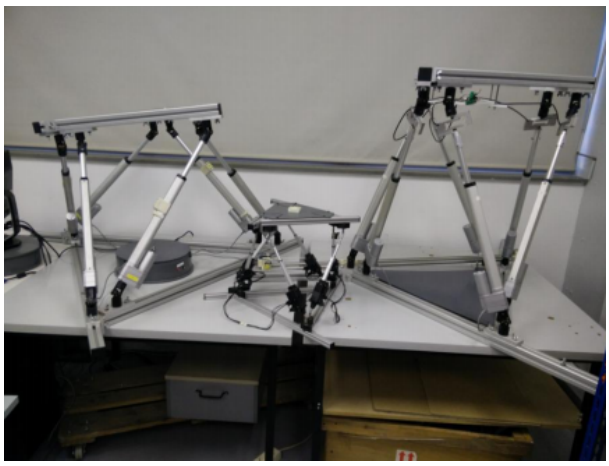

Figure 23: Some Gough-Stewart platforms from IRI's lab

Therefore, our parallel robot is a mechanical system which connects the platform to the fixed base through multiple serial kinematic chain. Then, in a *Gough-Stewart* parallel platform these kinematic chains are equal, each controlled by only one actuator. When solving the inverse kinematics, we will realize the important advantages of a parallel robot against a pure serial robot.

Advantages of parallel robots

When comparing the robustness and accuracy of both types of robots, it can be perceived the important advantages of the parallel ones. Moreover, the load capacity is incremented and, as a consequence, the speed and acceleration capacity is higher in a serial robot of the same dimensions. Some of these advantages are:

1. Simple and closed kinematic chains with few joints.
2. Better distribution of the supported loads.
3. Accuracy, since the positioning errors of each manipulator is damped due to the global average of all the involved serial kinematic chains.
4. Inertial forces are small enough to be neglected in most applications.
5. More stiffness.

As a principal disadvantage, the parallel robots exhibit a quite reduced workspace with respect to their serial counterparts.

Structure of the Gough-Stewart platform

The parallel robot prototype developed at IRI's lab has some relevant characteristics, since it is designed to be a portable gadget. These characteristics have led to assume certain approximations (sec 1.2), which influenced the development of this project. To begin with, the robot requires to have little dimensions to be portable. As previously mentioned in section 1.2.3, and reminded in the *Advantages of parallel robots*, this fact largely reduces the complexity of the problem, as the dynamics of the robot are not considered. In addition, an orientation compensator needs to instantaneously run in real time, which means that the actuators should operate at a significantly high speed. As the consequence, the manipulator should have simple kinematic chains to be easily controlled.

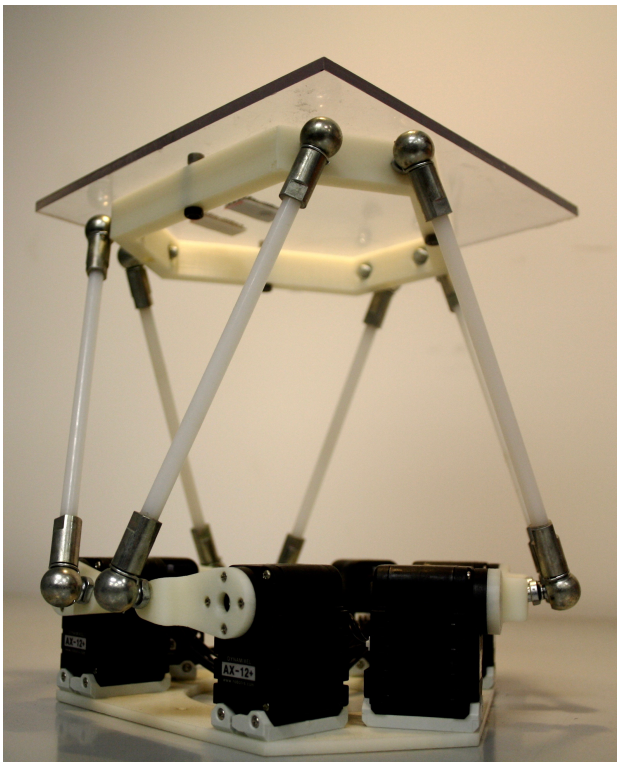

Figure 24: Structure of the parallel robot's manipulators

Note that each serial kinematic chain in the robot is not the typical *crank and connecting rod* mechanism because the bar acting as *connecting rod* can move in more than 2 dimensions of the space. However, the principle of the mechanism is similar.

The kind of parallel manipulator used is called 6-*RUS*, and the serial arms attached to the actuators dispose of two spherical joints, one at each end, which provides an extra degree of freedom. Therefore, the arm can move in a three-dimensional space to guarantee the 6 degrees of freedom of the platform. In addition, the initial setup of the actuators' rotors provides faster movements to the platform in their home position, which means that reaction times are smaller than in other manipulation systems.

As a counterpart, the serial kinematic chain of this kind of manipulator introduces serial singularities. They arise when the actuators' rotor and the arm acting as connecting rod get aligned. In this disposition, the mechanism has two feasible ways to carry on the movement and the system suffers from a loss of performance.

4.2 Kinematics of the parallel robot

In this section we define how the problem of reorienting the platform is mathematically treated according to the mechanics of the robot. At this point, it is assumed that all the communications between the different devices is already established. Therefore, the solution can be found following two possible directions: 1) an abductive procedure, giving an input to the system and looking at how it responds, or 2) an inductive procedure, concluding in a motion planning according to the desired output. These two ways to solve the problem rely on the resolution of the **forward kinematics** and **inverse kinematics** of the robot, respectively.

Inverse kinematics Vs Forward kinematics

The inverse kinematics consists in establishing the value of the joint coordinates, in this case the parameters of the Dynamixel™'s actuators, corresponding to the desired platform configuration. Usually, this relation is simple for parallel robots, and it will be seen that only 2^6 solutions can be found for the same configuration of the platform (each serial kinematic chain has two solutions for the inverse kinematics). Observe that dealing with this problem in serial robots becomes a really difficult task due to the amount of different solutions that can be found.

The forward kinematics faces the problem of determining the *pose* of the platform of the parallel robot from its actuated joint coordinates. In a dual way to the inverse kinematics, the solution for this problem is not unique in parallel robots, i.e., there are several ways of assembling a parallel manipulator for a set of actuated joint coordinates. However, in serial robots this problem is not an issue as it can be easily solved as a concatenation of transformations due to the fact that all joints are actuated. Then, it can be concluded that the solution of the forward kinematics problem for a serial chain is unique.

By the nature of this project, the problem demands to be solved through the **inverse kinematics** of the robot as the platform should be reoriented according to the desired orientation, which is now the input of the system given by the MPU-6050™. For the type of manipulator used (6-$\underline{R}US$, the solutions of the inverse kinematics determine which are the values for the joint variables or, in other words, the angular position of the actuators.

Working with reference frames and Homogeneous transformations

This section deals with all the algebraic work needed to solve the inverse kinematics of the parallel robot used in this project. Matlab® software, along with the *Robotics toolbox* and the functions introduced in section 2.3.2, are of great help to achieve this goal.

Reference frames will be used to describe a rigid body's ***pose***, i.e., its position and attitude. These reference frames must be defined before starting with the algebraic work. To represent the pose of a body in a $\mathbb{R}^3$ space, homogeneous transformations matrices T (sec. 2.3.2) will be used, adding two reference indexes (i and j), each one referring to different coordinate frames which will be attached to some part of the body. Therefore, ${}^{j}T_i$ matrices will denote **the pose of the frame i with respect to the frame j**.

$$ {}^{j}T_i = \begin{pmatrix} {}^{j}R_i & {}^{j}O_i \\ 0_{1\times 3} & 1 \end{pmatrix} \tag{4.1} $$

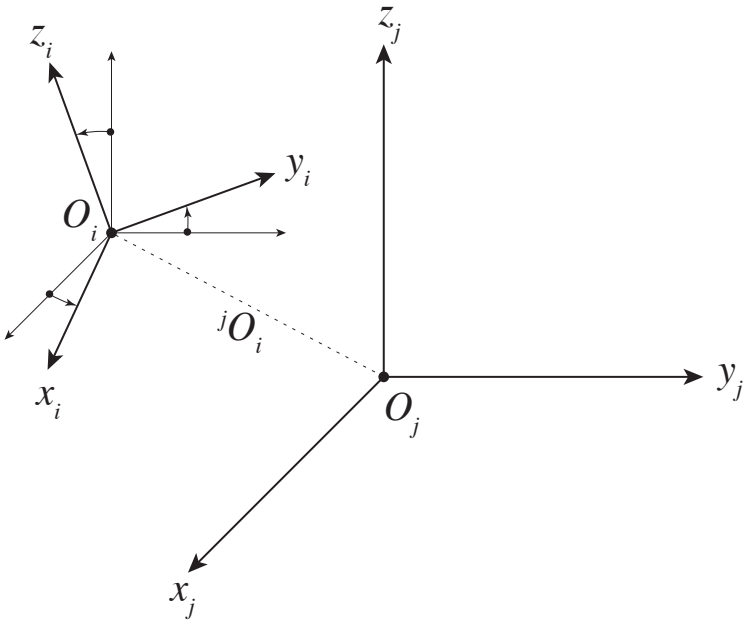

Figure 25: Representation of an arbitrary transformation

Remember that ${}^{j}R_i$ is a 3x3 matrix that describes a rotation, the one needed to orientate the frame i with respect to the frame j. This matrix can be conceived as the result of a pure rotations, as explained in section 2.3.2, or directly as a unitary change basis matrix from i to j as follows:

$$ {}^{j}R_i = \begin{pmatrix} \hat{x}_j & \hat{y}_j & \hat{z}_j \end{pmatrix} \cdot \begin{pmatrix} \hat{x}_i & \hat{y}_i & \hat{z}_i \end{pmatrix}^T \tag{4.2} $$

And ${}^{j}O_i$ is a 3x1 translation vector from the center of the frame j to the center of the frame i.

$$ {}^{j}O_i = \begin{pmatrix} {}^{j}x_i \\ {}^{j}y_i \\ {}^{j}z_i \end{pmatrix} \tag{4.3} $$

4.2.1 Inverse kinematics solution

The calculations to solve the inverse kinematics problem of a 6-$\underline{R}US$ manipulator are set forth in this section. It must be remembered that at this point the communication between the devices is already established, which means that the MPU-6050™ is already sending data. This data are the *Roll* and *Pitch* angle values of the platform, in other words, the known parameters of the inverse kinematics. In the following calculations, the subindex b refers to base, p to platform, j to joint, and r to rotor.

Absolute frame of reference

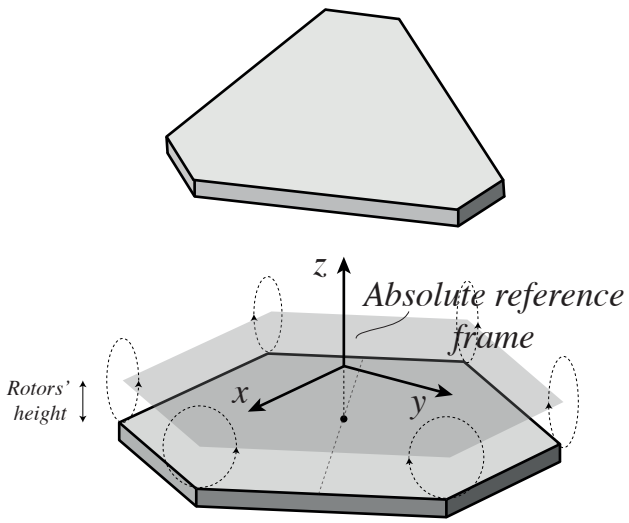

Figure 26: Absolute reference frame

To start with, an **absolute reference frame** must be defined somewhere in order to express the rest of the relative frames. The most intuitive point to place it is in the center of the robot base, at the Dynamixels™ rotor height to make calculations easier.

Platform pose with respect the base: ${}^{b}TR_{p}$

Once the absolute reference frame is defined, other relative frames can be expressed as the ${}^{j}T_{i}$ matrices showed before (eq. 4.1). Using the known ***Roll-***ϕ and ***Pitch-***θ parameters, a relative frame of reference for the platform's actual *pose* can be defined.

$$
{}^{b}TR_{p} = \left(\begin{array}{ccc|c} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & H \\ \hline 0 & 0 & 0 & 1 \end{array}\right) \cdot \left(\begin{array}{ccc|c} 1 & 0 & 0 & 0 \\ 0 & \cos\phi & -\sin\phi & 0 \\ 0 & \sin\phi & \cos\phi & 0 \\ \hline 0 & 0 & 0 & 1 \end{array}\right) \cdot \left(\begin{array}{ccc|c} \cos\theta & 0 & -\sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ \sin\theta & 0 & \cos\theta & 0 \\ \hline 0 & 0 & 0 & 1 \end{array}\right) \tag{4.4}
$$

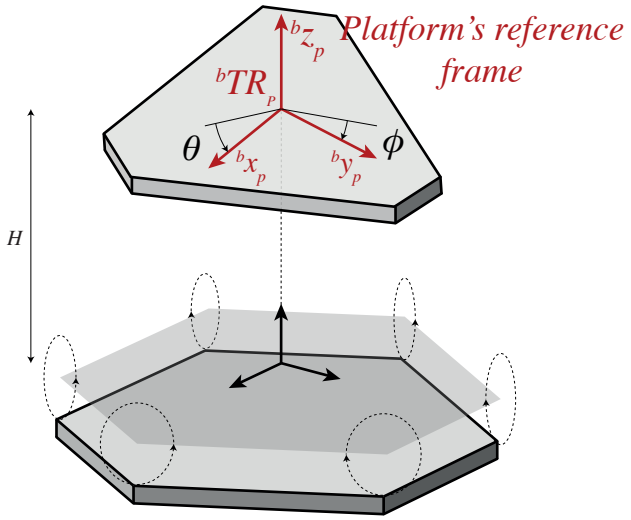

Figure 27: Platform's reference frame

Frames of the spherical joints with respect to the base: ${}^{b}TP_{j}$

Now, the platform reference frame should be translated from its center to each spherical joint. The result will be six ${}^{b}TP_{j}$ matrices, which will describe the *pose* of each joint j with reference to the base b.

$$ {}^{b}TP_{j1} = {}^{b}TR_p \cdot \underbrace{\left(\begin{array}{ccc|c} 1 & 0 & 0 & L2 \\ 0 & 1 & 0 & -LONG \\ 0 & 0 & 1 & 0 \\ \hline 0 & 0 & 0 & 1 \end{array}\right)}_{{}^{p}T_{j1}} \tag{4.5} $$

$$ {}^{b}TP_{j2} = {}^{b}TR_p \cdot \left(\begin{array}{ccc|c} 1 & 0 & 0 & -L2 \\ 0 & 1 & 0 & -LONG \\ 0 & 0 & 1 & 0 \\ \hline 0 & 0 & 0 & 1 \end{array}\right) \tag{4.6} $$

$$ {}^{b}TP_{j3} = {}^{b}TR_p \cdot \left(\begin{array}{ccc|c} \cos\left(-\frac{2\pi}{3}\right) & -\sin\left(-\frac{2\pi}{3}\right) & 0 & 0 \\ \sin\left(-\frac{2\pi}{3}\right) & \cos\left(-\frac{2\pi}{3}\right) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \hline 0 & 0 & 0 & 1 \end{array}\right) \cdot \left(\begin{array}{ccc|c} 1 & 0 & 0 & L2 \\ 0 & 1 & 0 & -LONG \\ 0 & 0 & 1 & 0 \\ \hline 0 & 0 & 0 & 1 \end{array}\right) \tag{4.7} $$

$$ {}^{b}TP_{j4} = {}^{b}TR_p \cdot \left(\begin{array}{ccc|c} \cos\left(-\frac{2\pi}{3}\right) & -\sin\left(-\frac{2\pi}{3}\right) & 0 & 0 \\ \sin\left(-\frac{2\pi}{3}\right) & \cos\left(-\frac{2\pi}{3}\right) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \hline 0 & 0 & 0 & 1 \end{array}\right) \cdot \left(\begin{array}{ccc|c} 1 & 0 & 0 & -L2 \\ 0 & 1 & 0 & -LONG \\ 0 & 0 & 1 & 0 \\ \hline 0 & 0 & 0 & 1 \end{array}\right) \tag{4.8} $$

$$ {}^{b}TP_{j5} = {}^{b}TR_p \cdot \left(\begin{array}{ccc|c} \cos\left(-\frac{4\pi}{3}\right) & -\sin\left(-\frac{4\pi}{3}\right) & 0 & 0 \\ \sin\left(-\frac{4\pi}{3}\right) & \cos\left(-\frac{4\pi}{3}\right) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \hline 0 & 0 & 0 & 1 \end{array}\right) \cdot \left(\begin{array}{ccc|c} 1 & 0 & 0 & L2 \\ 0 & 1 & 0 & -LONG \\ 0 & 0 & 1 & 0 \\ \hline 0 & 0 & 0 & 1 \end{array}\right) \tag{4.9} $$

$$ {}^{b}TP_{j6} = {}^{b}TR_p \cdot \left(\begin{array}{ccc|c} \cos\left(-\frac{4\pi}{3}\right) & -\sin\left(-\frac{4\pi}{3}\right) & 0 & 0 \\ \sin\left(-\frac{4\pi}{3}\right) & \cos\left(-\frac{4\pi}{3}\right) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \hline 0 & 0 & 0 & 1 \end{array}\right) \cdot \left(\begin{array}{ccc|c} 1 & 0 & 0 & -L2 \\ 0 & 1 & 0 & -LONG \\ 0 & 0 & 1 & 0 \\ \hline 0 & 0 & 0 & 1 \end{array}\right) \tag{4.10} $$

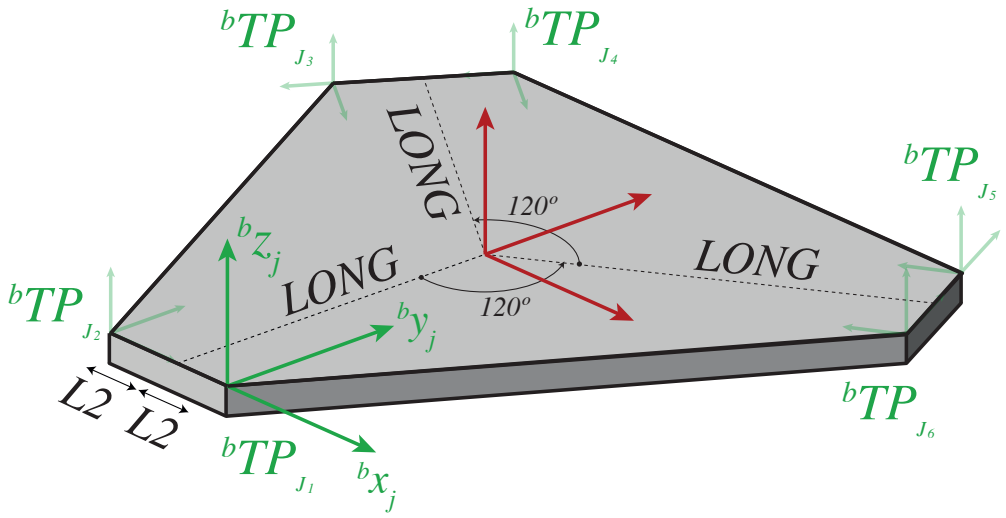

Figure 28: Joints' reference frames

Frames of the actuator's rotors with respect to the base: ${}^{b}TB_r$

The same should be done with the rotor of each actuator. Six ${}^{b}TB_r$ matrices should be found, one for each rotor. Notice that the ${}^{b}TB_r$ matrices will be defined to have one axis on the AA_0 bar of the manipulator in its home position, as it will make the calculations easier.

$$ {}^{b}TB_{r1} = \left(\begin{array}{ccc|c} 1 & 0 & 0 & L1 \\ 0 & 1 & 0 & -RADI \\ 0 & 0 & 1 & 0 \\ \hline 0 & 0 & 0 & 1 \end{array}\right) \cdot \left(\begin{array}{ccc|c} \cos\left(-\frac{2\pi}{3}\right) & -\sin\left(-\frac{2\pi}{3}\right) & 0 & 0 \\ \sin\left(-\frac{2\pi}{3}\right) & \cos\left(-\frac{2\pi}{3}\right) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \hline 0 & 0 & 0 & 1 \end{array}\right) \tag{4.11} $$

$$ {}^{b}TB_{r2} = \left(\begin{array}{ccc|c} 1 & 0 & 0 & -L1 \\ 0 & 1 & 0 & -RADI \\ 0 & 0 & 1 & 0 \\ \hline 0 & 0 & 0 & 1 \end{array}\right) \cdot \left(\begin{array}{ccc|c} \cos\left(\frac{2\pi}{3}\right) & -\sin\left(\frac{2\pi}{3}\right) & 0 & 0 \\ \sin\left(\frac{2\pi}{3}\right) & \cos\left(\frac{2\pi}{3}\right) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \hline 0 & 0 & 0 & 1 \end{array}\right) \tag{4.12} $$

$$ {}^{b}TB_{r3} = \left(\begin{array}{ccc|c} \cos\left(-\frac{2\pi}{3}\right) & -\sin\left(-\frac{2\pi}{3}\right) & 0 & 0 \\ \sin\left(-\frac{2\pi}{3}\right) & \cos\left(-\frac{2\pi}{3}\right) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \hline 0 & 0 & 0 & 1 \end{array}\right) \cdot {}^{b}TB_{r1} \tag{4.13} $$

$$
{}^{b}TB_{r4} = \left(\begin{array}{ccc|c} \cos\left(-\frac{2\pi}{3}\right) & -\sin\left(-\frac{2\pi}{3}\right) & 0 & 0 \\ \sin\left(-\frac{2\pi}{3}\right) & \cos\left(-\frac{2\pi}{3}\right) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \hline 0 & 0 & 0 & 1 \end{array}\right) \cdot {}^{b}TB_{r2} \tag{4.14}
$$

$$
{}^{b}TB_{r5} = \left(\begin{array}{ccc|c} \cos\left(-\frac{4\pi}{3}\right) & -\sin\left(-\frac{4\pi}{3}\right) & 0 & 0 \\ \sin\left(-\frac{4\pi}{3}\right) & \cos\left(-\frac{4\pi}{3}\right) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \hline 0 & 0 & 0 & 1 \end{array}\right) \cdot {}^{b}TB_{r1} \tag{4.15}
$$

$$
{}^{b}TB_{r6} = \left(\begin{array}{ccc|c} \cos\left(-\frac{4\pi}{3}\right) & -\sin\left(-\frac{4\pi}{3}\right) & 0 & 0 \\ \sin\left(-\frac{4\pi}{3}\right) & \cos\left(-\frac{4\pi}{3}\right) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \hline 0 & 0 & 0 & 1 \end{array}\right) \cdot {}^{b}TB_{r2} \tag{4.16}
$$

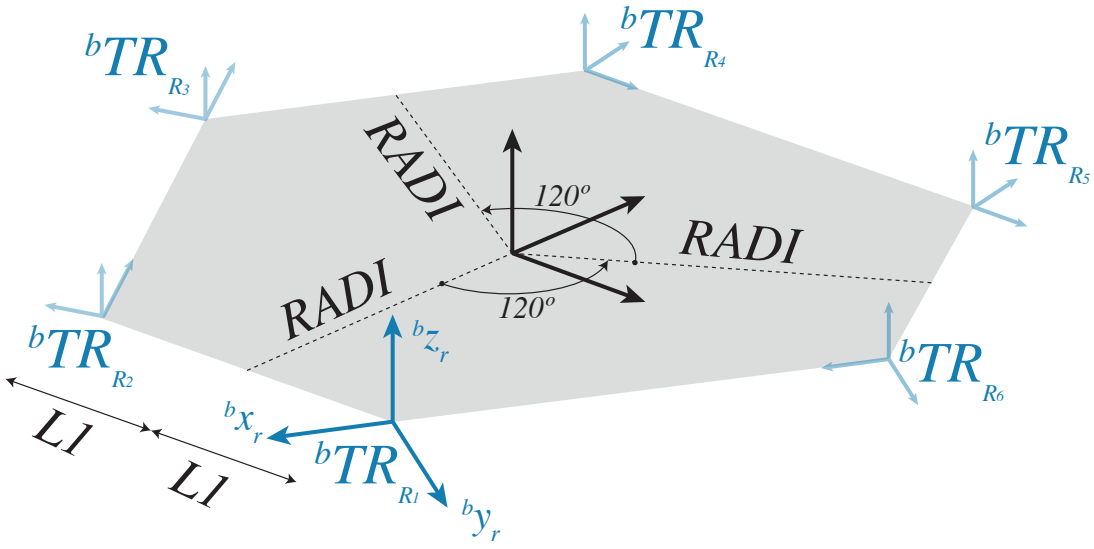

Figure 29: Rotors' reference frames

Resolution of the inverse kinematics

At this point the problem is reduced to find and to solve the equations that relate to the actuators' rotor *pose* to the spherical joints' *pose*. These two elements are physically related through the 6-$\underline{R}US$ manipulator, which means that the transformation matrix that relates them is not that obvious. Then, in order to find the mathematical relationship, the manipulator must be analyzed in more detail.

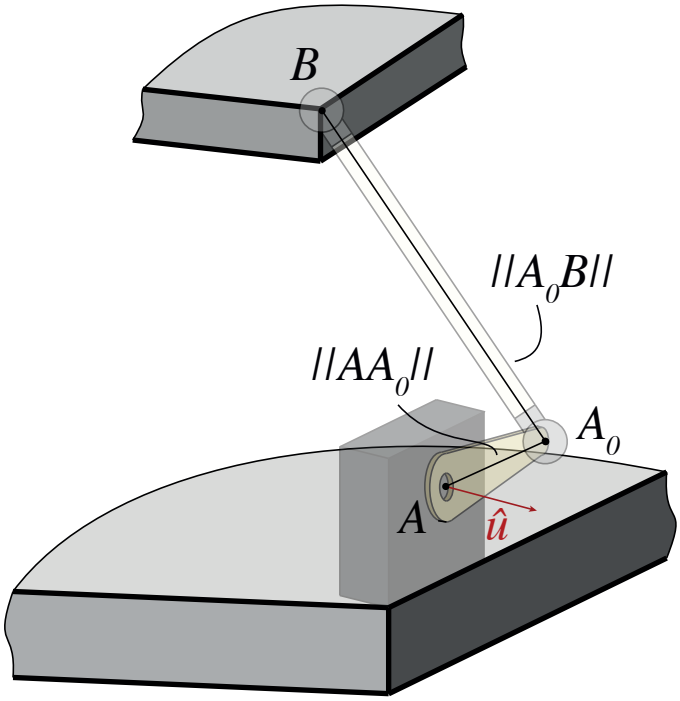

Figure 30: The 6-$\underline{R}US$ manipulator

As it can be seen in Fig. 30, the rotor movement is transmitted through the mechanism AA_0B. The problem of the inverse kinematics is to determine the angle α between the mechanism AA_0 and the vector of the rotor frame placed in the initial position of the mechanism (Fig. 31).

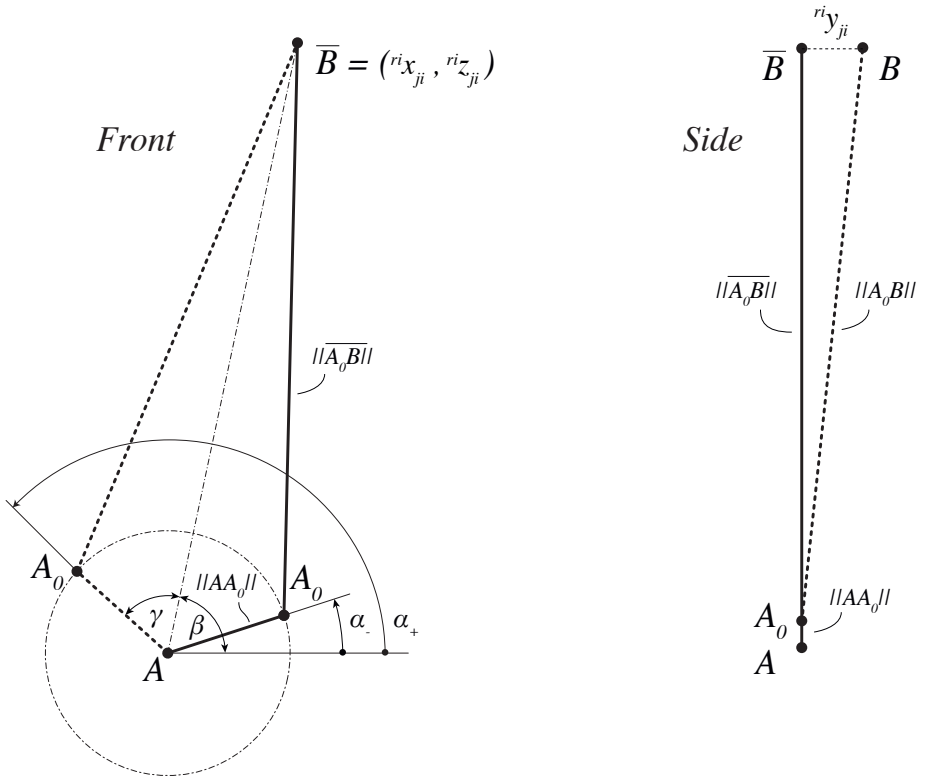

Figure 31: Front and side view of the serial chain representing the manipulator's legs

Figure 31 shows $\overline{A_0B}$ as the projection of A_0B onto the orthogonal plane to the rotor vector $\vec{u}$. The following operation will describe the spherical joint frame with respect the corresponding rotor frame. However, the only interest is to get the relative position of the joint with respect the rotor. (Subindex i refers to each rotor (r) - joint (j) couple as the following equations are analogous to each serial chain).

$$ {}^{\mathrm{ri}}TP_{ji} \cdot \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} = ({}^{\mathrm{b}}TR_{ri})^{-1} \cdot {}^{\mathrm{b}}TP_{ji} \cdot \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} {}^{\mathrm{ri}}x_{ji} \\ {}^{\mathrm{ri}}y_{ji} \\ {}^{\mathrm{ri}}z_{ji} \\ 1 \end{pmatrix} \tag{4.17} $$

Vector $({}^{\mathrm{ri}}x_{ji}, {}^{\mathrm{ri}}y_{ji}, {}^{\mathrm{ri}}z_{ji})$ is the joint B displacement with respect to the rotor reference frame placed in A. This vector can be used to calculate the parameters needed to solve the problem.

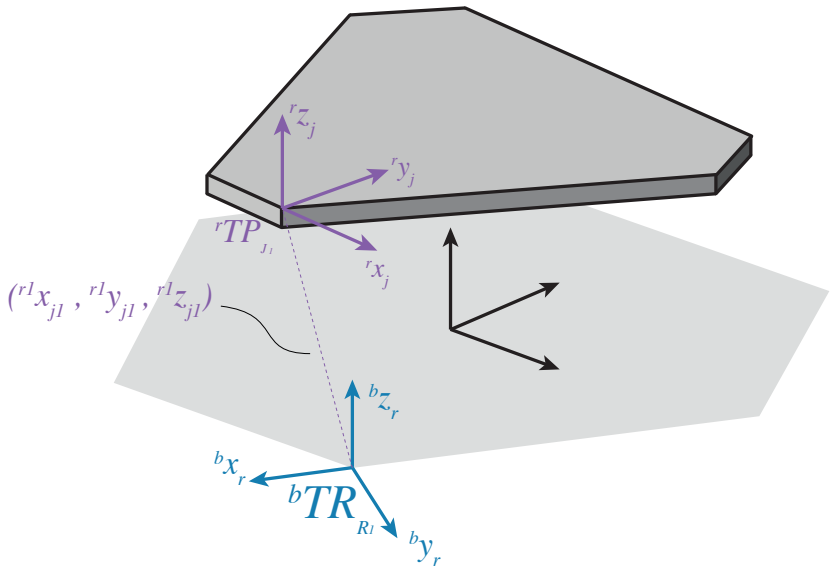

Figure 32: ith joints' reference frame with respect the ith rotor

The length ($||\overline{A_0B}||$) of the $\overline{A_0B}$ projection can be found through the following operation:

$$||\overline{A_0B}|| = \sqrt{||A_0B||^2 - {}^{\mathrm{ri}}y_{ji}^2} \tag{4.18}$$

Examining the angular relationships, it can be found that two values for α lead to the same solution:

$$\alpha = \beta \pm \gamma \tag{4.19}$$

Where β and γ can be found applying the cosine theorem, as follows:

$$\beta = \arctan\left(\frac{{}^{\mathrm{ri}}z_{ji}}{{}^{\mathrm{ri}}x_{ji}}\right) \tag{4.20}$$

$$\gamma = \arccos\left(\frac{(||AA_0||)^2 + ({}^{\mathrm{ri}}x_{ji})^2 + ({}^{\mathrm{ri}}z_{ji})^2 - (||A_0B||)^2}{2 \cdot ||AA_0|| \cdot \sqrt{({}^{\mathrm{ri}}x_{ji})^2 + ({}^{\mathrm{ri}}z_{ji})^2}}\right) \tag{4.21}$$

In order to understand why two solutions can be found, it should be realized that the mechanism BA_0 imposes that point A_0 lies on a **sphere**, due to the spherical joint, centered at B with radius $||A_0B||$. In the same way, the bar AA_0 imposes that point A_0 is constrained to lie on a **circle** centered at A with radius $||AA_0||$ lying in the plane perpendicular to the rotor axis of the actuator. This sphere and circle intersect in two points. Hence, the whole inverse kinematics admits generally $2^6 = 64$ solutions for each platform configuration.

Then, choosing one of the 64 possible solutions depends on the home position. Looking at the circumference drawn by AA_0, the solution will be the angle which sets the mechanism closer to its home configuration. Therefore:

$$\alpha_1 = \beta + \gamma \tag{4.22}$$

$$\alpha_2 = \beta - \gamma \tag{4.23}$$

$$\alpha_3 = \beta + \gamma \tag{4.24}$$

$$\alpha_4 = \beta - \gamma \tag{4.25}$$

$$\alpha_5 = \beta + \gamma \tag{4.26}$$

$$\alpha_6 = \beta - \gamma \tag{4.27}$$

4.2.2 Problems writing Arduino® code

Several problems arise when implementing this section of the code on the Arduino® UNO microcontroller. The most important are:

- ***Basic math operations:*** Arduino®'s software does not provide really powerful mathematic libraries, which means that most of the operations showed above (i.e. matrix operations) would have to be build on our own.
- ***Out of workspace range:*** If the robot is tilted exceeding a certain inclination, the *Motion Process Unit*™ would send ***Roll-***ϕ and ***Pitch-***θ values out of the workspace range of the robot. As a consequence, software limits should be introduced.
- ***Long code:*** Reminding section 2.2.1, the *Motion Process Unit*™ provides a FIFO buffer which stores the motion parameters to avoid running out of values in between. Therefore, if one value from the *MPU*™ takes too long to be analyzed due to the length of the Arduino® code, the FIFO could suffer from underflow problems.

Solving problems

Different ways can be taken to solve these main problems. In order to be able to work with matrices, and solving the first issue, a *C++* library called **Eigen** can be downloaded and installed to make it operative on an Arduino®. There is also a website[11] which explains how basic functions can be used, such us how to generate a n-vector, a nxn-matrix; or how to operate with cross and scalar products. However, it gets hard to work with it as *Debugers* are not available for this library, which means that syntax errors in the code are hard to find.

The problem of working out of the range is easy to solve, just giving upper and lower limit values to the actuators.

It takes longer to solve the third problem because, as it has already been mentioned, the written code provided by Jeff Rowberg to control the *Motion Process Unit*™ was treated as a *black box*. It seemed unfeasible to us, in the short run, to modify Jeff Rowberg's code referring to the length of the FIFO buffer. The only section that could be modified is the way the inverse kinematics is solved.

It must be said that, in order to verify if the buffer was the main problem before changing the way the kinematics is solved, other kinds of inverse kinematics were applied, such as the platform translation, which code is quite shorter. The conclusion remained the same as it worked normally.

Therefore, the following two sections are possible alternatives studied in order to replace the large calculations of the inverse kinematics problem presented above.

4.2.3 Alternative 1: The Fourier method

If one starts thinking about how to relate the actuators angular position depending on the attitude of the platform, to plot the inputs and outputs parameters is the most intuitive operation to begin with. In this case, Matlab® will be essential to represent the relationships between the input parameters and the output parameters.

[11] Eigen's support webside: `https://eigen.tuxfamily.org/dox/GettingStarted.html`

Simplification of the problem using Matlab®

Now, we are going to plot the relationship between the the output variables, i.e., the rotor angular position of each actuator (α_1, α_2, α_3, α_4, α_5, α_6), with respect to the input variables of the inverse kinematics, i.e. the *Roll-*ϕ and *Pitch-*θ angles.

Therefore, using the inverse kinematics already solved and computed in the Matlab® software, six 3D-plots are represented. It must be noticed that the angles represented in the following plots are not directly ϕ and θ, since this study has been done using a different solution for the inverse kinematics that later on were replaced by the solution explained in section 4.2.1. Nevertheless, those other angles (σ and λ) are analogous to ϕ and θ. The inverse kinematics solution used for this alternative is placed in the appendix, and Fig. 33 shows how to find σ and λ angles, being vector $\hat{n}$ orthogonal to the platform.

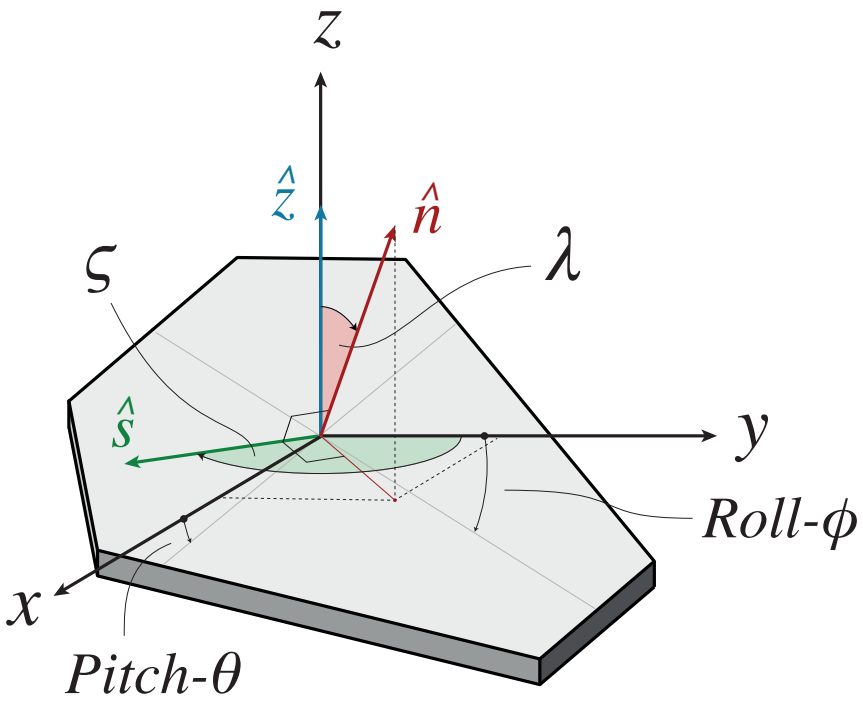

Figure 33: Relationship between Roll-ϕ, Pitch-θ and σ, λ angles

The following figures present each of those plots. They represent α_i with respect to σ and λ.

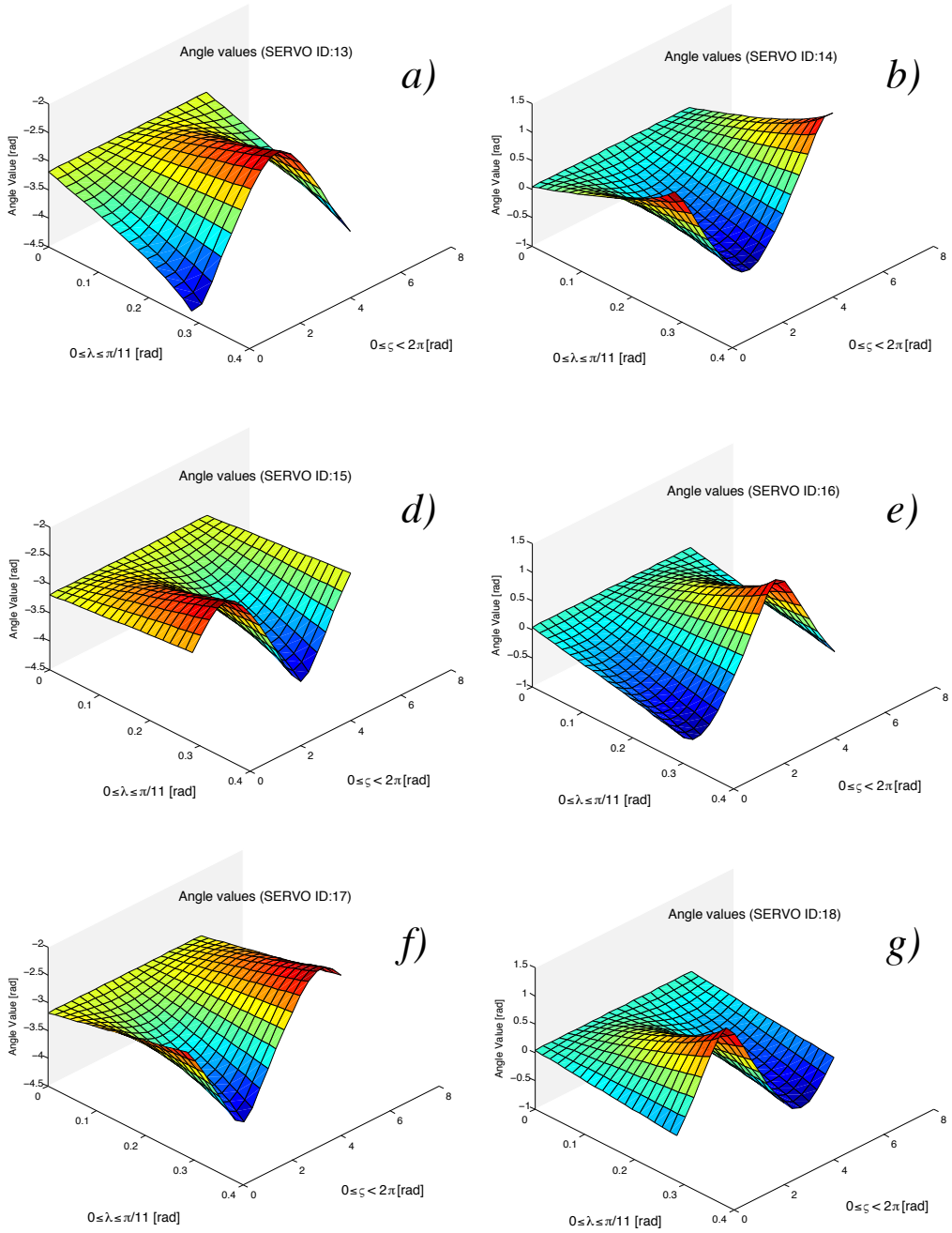

Figure 34: Surface plots of α_i with respect to σ and λ

The reason why three plots are centered at 0, and the other three are centered close to -3 (actually -π = -3,1415..), is due to the home position of each actuator.

Looking at the appendix with respect to these plots, two facts can be verified. On the one hand, it seems that changes in σ make α to follow a **sinusoidal curve** presenting a different phase shift in each actuator. On the other hand, it can be seen that λ is related to the **amplitude** of the oscillating function.

The important fact is that sinusoidal curves with different amplitudes but same phase shift could be found if slices of these plots are made along the λ axis. Moreover, the changing rate of the amplitude is linear. These two facts combined lead to the first alternative for the inverse kinematics problem.

If a few of those sinusoidal curves could be parameterized, linear interpolations would be implemented to interpolate the empty spaces between these curves. It means that all the inverse kinematics would be stored as a few sinusoidal curves for each actuator, which will be fast to compute and would not take up much memory in the microcontroller.

Since to fit a sinusoidal function "by hand" does not seem appropriate for obvious reasons, the best way to achieve it is through the *Fourier* transform.

Parameterizing the sinusoidal curves

The *Fourier* transform is a tool that permits to decompose a periodic signal into an infinite series of sine and cosine functions as follows:

$$f(x) = \frac{a_0}{2} + \sum_{n=0}^{\infty} \left[a_n \cdot \cos\left(n \cdot \omega \cdot x\right) + b_n \cdot \sin\left(n \cdot \omega \cdot x\right)\right] \tag{4.28}$$

The more terms, or *order*, added in the sum, the more accurate is the *Fourier* approximation to the periodic function. Therefore, it is only needed to find a few parameters (a_0, a_1, b_1, ..., a_n, b_n) to approximate the sinusoidal curves found before.

To find these parameters is an easy task using Matlab® because it provides easy-to-use functions to approximate a series of points through *FFT (Fast Fourier Transform)* transform of the convenient order. The following figure shows the 2^{nd}-order *Fourier* series for $\lambda = \frac{\pi}{11}$.

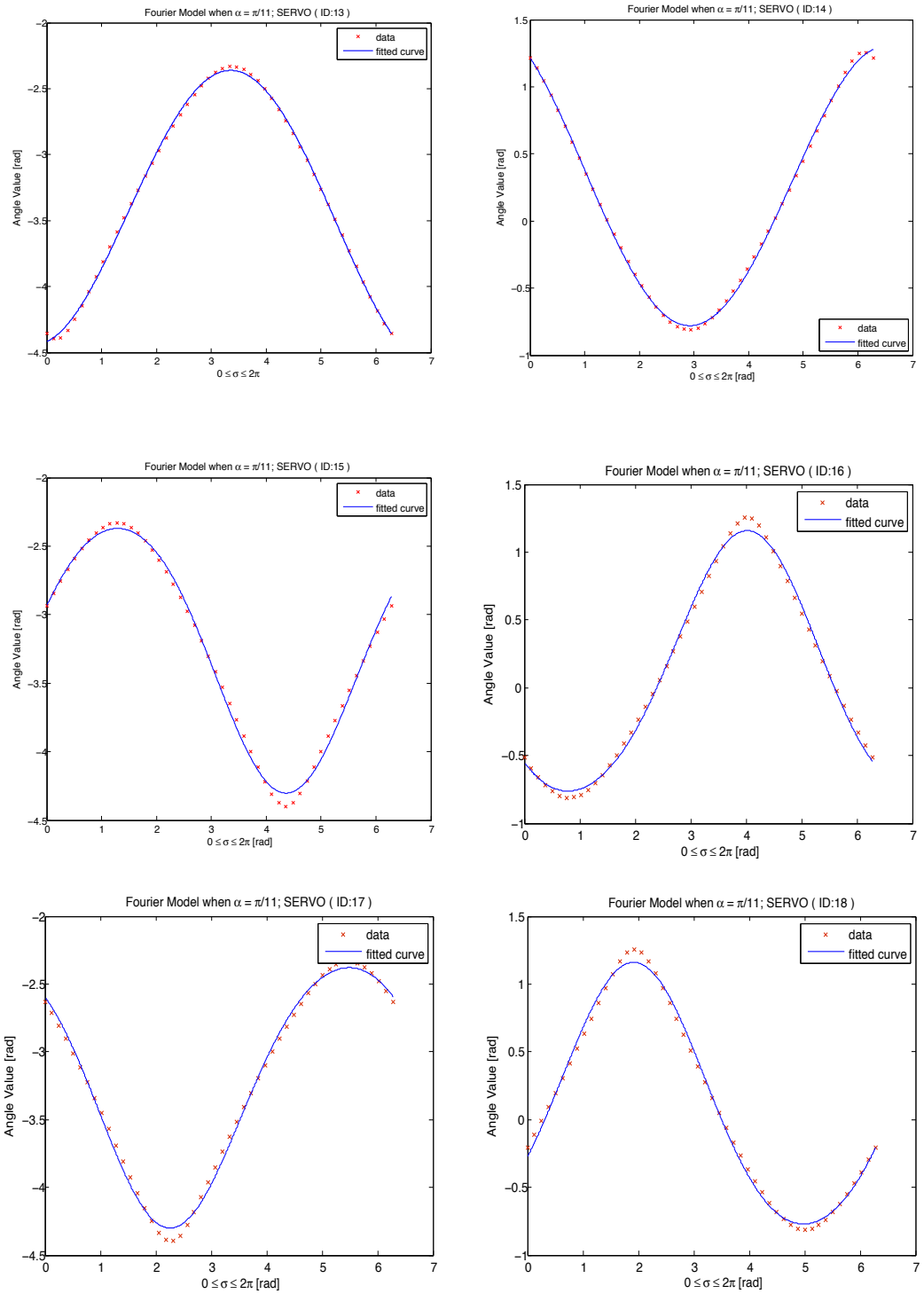

Figure 35: Fourier series of each actuator for $\lambda = \frac{\pi}{11}$

The *Fourier* parameters found for these functions are:

	$\frac{a_0}{2}$	a_1	b_1	a_2	b_2	ω
f_{13}(x)	-2,195	-0,6818	-2,082	-1,543	1,308	0,3596
f_{14}(x)	-0,9467	1,175	1,849	0,9861	-1,766	0,3596
f_{15}(x)	-3,261	0,2638	0,4085	0,007474	-0,009917	1,014
f_{16}(x)	0,1143	-0,6335	-0,7205	-0,02923	0,08123	0,9993
f_{17}(x)	-3,256	0,6301	-0,7235	0,02998	0,08096	0,9993
f_{18}(x)	0,1196	-0,3433	0,9026	-0,05162	-0,05562	1,014

Table 3: Parameters of the *Fourier* series

In the appendix, more graphs and parameters of *Fourier* approximations applied to other slices than those shown in Fig. 34 can be found.

Implemented code

Finally, if this alternative is implemented, a 2^{nd}-order *Fourier* series is enough in practice. As an alternative, a simple linear interpolation can be calculated as explained next.

4.2.4 Alternative 2: The Jacobian method

In this section, the final solution to solve the inverse kinematics problem is described. This alternative reduces the solution to a single matrix, which multiplied by $(\phi, \theta)^{\text{T}}$ gives the angle values of the actuators (α_1, α_2, α_3, α_4, α_5, and α_6).

The method

As stated before, the configuration of the platform, is specified by the Roll-ϕ, Pitch-θ angles. Note that the platform needs to be stabilized at $\phi = 0$, $\theta = 0$, then it can be considered that $\Delta\phi = \phi$, and $\Delta\theta = \theta$.

Since there are six revolute joints, their angular positions have been defined as a vector ${}^{\text{s}}\vec{\alpha}$ = $({}^{\text{s}}\alpha_1, {}^{\text{s}}\alpha_2, {}^{\text{s}}\alpha_3, {}^{\text{s}}\alpha_4, {}^{\text{s}}\alpha_5, {}^{\text{s}}\alpha_6)^{\text{T}}$. Then, the robot will be controlled by giving specific target angles to the joints ${}^{\text{t}}\vec{\alpha} = ({}^{\text{t}}\alpha_1, {}^{\text{t}}\alpha_2, {}^{\text{t}}\alpha_3, {}^{\text{t}}\alpha_4, {}^{\text{t}}\alpha_5, {}^{\text{t}}\alpha_6)^{\text{T}}$, where ${}^{\text{t}}\alpha_i$ is the target position for each actuator.

Therefore, the inverse kinematics permits to express that each *joint angle* as a function of ϕ and θ; this can be written as:

$$^{t}\vec{\alpha} = {}^{s}\vec{\alpha}(\phi, \theta) \tag{4.29}$$

And the desired change for the joints can now be expressed as:

$$\Delta\vec{\alpha} = {}^{t}\vec{\alpha} - {}^{s}\vec{\alpha} \tag{4.30}$$

According to this, an iterative method can be used to converge to the stabilized *pose*. For this, the functions given by (4.29 are linearized. This linealization leads to the *Jacobian* matrix J, which is a function of the ϕ and θ values, and is defined as follows.

$$J(\phi, \theta) = \begin{pmatrix} \frac{\delta\alpha_1}{\delta\phi} & \frac{\delta\alpha_1}{\delta\theta} \\ \frac{\delta\alpha_2}{\delta\phi} & \frac{\delta\alpha_2}{\delta\theta} \\ \vdots & \vdots \\ \frac{\delta\alpha_6}{\delta\phi} & \frac{\delta\alpha_6}{\delta\theta} \end{pmatrix} \tag{4.31}$$

Then, J can be viewed as a 6×2 matrix that describes the changes of the joint angles with respect to the platform orientation, or as a velocity relation between the revolute joints and the end-effector (eq. 4.32).

$$\begin{pmatrix} \dot{\alpha_1} \\ \vdots \\ \dot{\alpha_6} \end{pmatrix} = J(\phi, \theta) \begin{pmatrix} \dot{\phi} \\ \dot{\theta} \end{pmatrix} \tag{4.32}$$

Then,

$$\begin{pmatrix} \frac{\delta\alpha_1}{\delta t} \\ \vdots \\ \frac{\delta\alpha_6}{\delta t} \end{pmatrix} = J(\phi, \theta) \begin{pmatrix} \frac{\delta\phi}{\delta t} \\ \frac{\delta\theta}{\delta t} \end{pmatrix} \Rightarrow \begin{pmatrix} \delta\alpha_1 \\ \vdots \\ \delta\alpha_6 \end{pmatrix} = J(\phi, \theta) \begin{pmatrix} \delta\phi \\ \delta\theta \end{pmatrix} \tag{4.33}$$

Therefore, the difference of the joints angles can be estimated through:

$$\begin{pmatrix} \Delta\alpha_1 \\ \vdots \\ \Delta\alpha_6 \end{pmatrix} \approx J(\phi, \theta) \begin{pmatrix} \Delta\phi \\ \Delta\theta \end{pmatrix} \tag{4.34}$$

How to find the Jacobian matrix

To mathematically find J, the joint equations of the kinematics should be derived with respect to the entry values, which are the ϕ and θ angles, and then it should also be evaluated by them. To find and derive those equations are not goals of this project. However, different ways to reach J can be found. It has been previously mentioned that the Jacobian could be considered as a matrix which **describes the behavior of the joint angles due to the changes produced in the entry values**.

According to this statement, the resolution of the inverse kinematics previously calculated can be used to develop a Matlab® code (placed in the appendix) in charge to build the Jacobian matrix following these easy steps:

First, the code places the platform in its initial configuration, where all the joint values are known. Just after that, a small enough increase is applied to $\phi = \phi + \Delta\phi$, and the solution of the inverse kinematics ($\Delta\vec{\alpha_\phi}$) is placed in the first column of J.

Later on, the same process is applied to θ, and the solution ($\Delta\vec{\alpha_\theta}$) is placed in the second column of J.

$$J_{\phi,\theta} = \begin{pmatrix} \Delta\vec{\alpha_\phi} & \Delta\vec{\alpha_\theta} \end{pmatrix} \tag{4.35}$$

The Matlab® code developed builds a 6×3 Jacobian matrix, in which the interaction caused by *Yaw*-ψ is added. This is done, in case of using a magnetometer, to correct the MPU-6050™ *drifting*.

Implemented code

More approximations need to be done when implementing the corresponding code of this method on an Arduino® microcontroller. As previously stated, a J matrix is a function of ϕ and θ values, which means that different Js should be calculated for each platform orientation in order to perform accurately.

However, it has been seen that to find the Jacobian for each entry value, the inverse kinematics resolution showed in section 4.2.1 requires to be executed, or in other words, requires to be implemented in the Arduino® code. As it can be understood, this cannot be carried out since these methods have been developed under the premise that the resolution for the inverse kinematics cannot be used.

4.2.5 Matlab® simulation of the final solution

Before implementing this method on the microcontroller, a few simulations were carried out on Matlab® to verify the above calculations and its performance.

In section 3.1.2 and 3.2.3, the problem of the simultaneous communication between the PC, the actuators and the microcontroller was mentioned. Due to the introduced limitations, the microcontroller cannot simultaneously interact with Matlab® and the Dynamixels™, which means that the simulation must be performed offline.

To this end, the ***pseudoinverse***, or ***Moore-Penrose*** inverse[12], of J must be introduced since the Jacobian is not squared, and the standard inverse is not defined. The *pseudoinverse* computes the solution which better fits to a linear system of equations that lacks a unique solution in terms of minimum squares. It is used then, to recalculate the position of the platform once the difference of the joints' angles estimated through the $J_{0,0}$ matrix are applied. Just after this, using an iterative method, the new joints' angles will be estimated, and the *pseudoinverse* will be used again to recalculate the platform position. And so on, until the error is small enough.

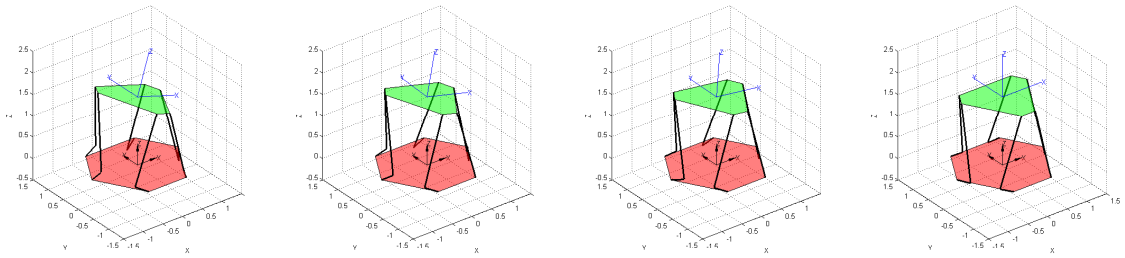

Figure 36: Pure *pitch* recovery (from left to right)

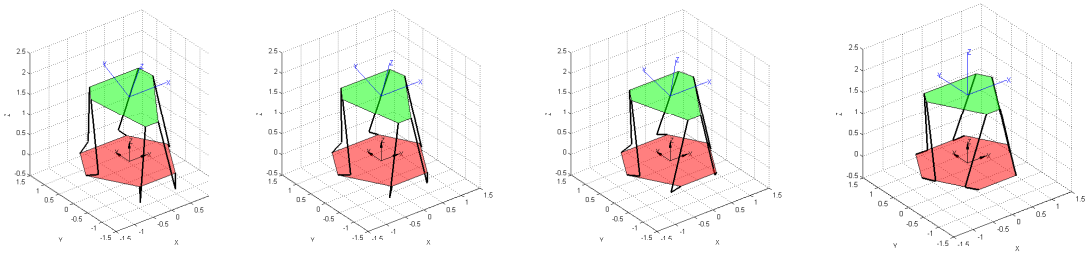

Figure 37: Pure *roll* recovery (from left to right)

[12] MathWork's webside `http://es.mathworks.com/?s_tid=gn_logo` can be visited for Moore-Penrose information and support.

4.3 The control

The control of the system is the most appealing process of any project. After the resolution of the strategy that will be used, incorporating the calculations effectuated in the previous section, the system is supposed to develop its function. This means that the control will determine how the system will operate in order to achieve its goals.

The most common application of a compensation system is to stabilize a photograph or a video camera, in both, accuracy, the well-functioning, the versatility and the reaction speed will define if it is a good stabilization system or if it is not.

In order to control all those parameters, which will make the system valuable, there exist several methods to follow, all of them based on what it is called **closed-loop control**. This method forces to feedback the system, usually through any type of sensor, to work directly with the committed error in every cycle. To treat this committed error as a parameter, something known as **controller** is implemented, which modifies this parameter in order to reach the final state of the system quicker, more accurately and fluidly.

4.3.1 Strategies of execution

It is obvious that each actuator will execute different movements, as well as positioning itself in different angular values for each configuration of the platform. In the same way, it can be deduced that the strategy of the movements is different for each manipulator. However, the control parameters of each manipulator will go through very similar cycles.

Distinct possible alternatives have been previously explained to solve the inverse kinematics. Subsequently, it will be specified the strategies that should be followed for each of the alternatives proposed.

Strategy for the Fourier method

It must be reminded that this method was developed following a resolution for the inverse kinematics different from the one finally used. This resolution is based on the location of the normal vector to the platform (Fig. 33), apart from its projection on the plane x, y.

Then, this resolution aims to redirect the normal vector rotating the platform so that the vector is parallel to the z-axis. The rotation of the platform is performed around a very specific axis. (Fig. 38)

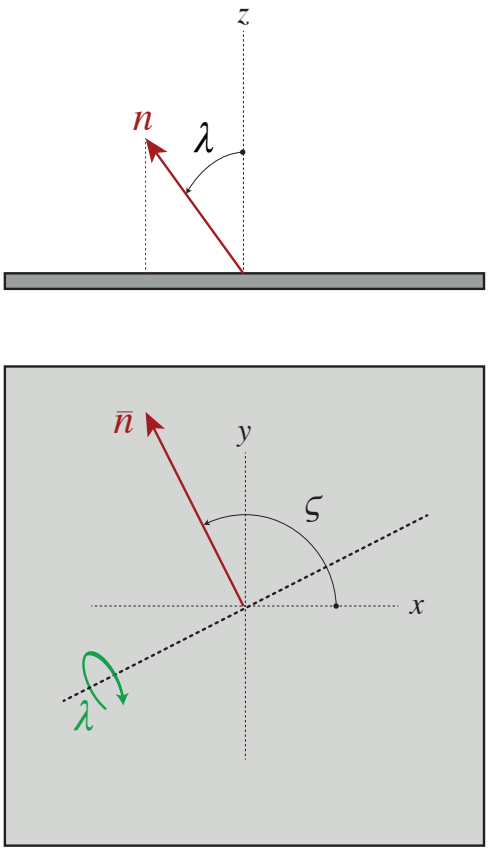

Figure 38: Strategy representation for the Fourier method

The σ and λ values are the ones represented in figure 33, which are analogous to ϕ and θ.

Since this action was unable to be executed through the complete resolution of the kinematics, the relationship between the input parameters σ, λ and the outputs $\vec{\alpha}$ was used. These relations, which were presented in the plots of Fig. 34, are representations of all the possible solutions inside the two-dimensional workspace granted by the input parameters. Eventually, the representations were approximated through 2^{nd}-order *Fourier* series, used to assign values to the actuators depending on the σ parameter entered. Obviously, not all the surface has to be approximated through *Fourier*, since changes in λ are linear, and the solutions between approximations can be estimated through linear interpolation.

The only thing that needs to be carried is a 2^{nd}-order *Fourier* series which will be evaluated in σ's values, and which parameters are chosen depending on λ's.

Strategy for the Jacobian method

If the previously method seeks a position relationship between the end-effector and the actuators' rotors, this method searchs a velocity relationship between both of them. As it has been exposed in section 4.2.4, this relationship is achieved by linearizing the system through the Jacobian matrix. Besides, it has been seen before that J is a function of ϕ and θ, which means that a different Jacobian matrix for each platform position exist.

In order to develop a feasible strategy to be implemented in the limited microcontroller, a few approximations will be assumed, which will be considered negligible near the work zone.

Thus, it will be assumed that only the Jacobian matrix for $\phi = 0$ and $\theta = 0$ ($J_{0,0}$ in its initial state) will be found through Matlab®, since the microcontroller itself can not calculate them. Therefore, only this matrix will have to be implemented in the strategy, which means that the performance of the robot in a region close to these values will be quite accurate. Otherwise, if ϕ or θ deviate from this region, the solution of this method will carry a significant error.

However, in order to reduce the approximation error of the method, the plane-region created by the achievable ϕ and θ values can be divided in 9 equal areas, for example. Then, the kinematics would be governed by 9 different Jacobian matrices, each one calculated in the center of each area. The $J_{\phi,\theta}$ matrix chosen for the kinematics depends on which zone of the region the platform parameters are closer to.

Hysteresis loop

This attempt to reduce the error can lure on vibrations and perturbations in the physical prototype caused by the edge values of the region mentioned before. If ϕ or θ were placed on those edges or quickly permuting between zones, the inverse kinematics would lead to slightly different solutions; successively changing the joint values.

Therefore, an **hysteresis loop** should be defined between every zone, and implemented in the Arduino® code, in order to solve the vibration problems.

Saturation values

Eventually, both are iterative methods which demand this implementation. Unlike the resolution of the inverse kinematics problem, in these methods there is no way to know if an impossible configuration for the platform is being executed. This leads to find the joint values from which the platform movement (about 18^{o}) begins to be blocked by the joint restrictions. In order to keep the manipulators away from dangerous configurations, these positions define the *saturation values* for the actuators, which will be also implemented in the code.

This task can be empirically done since joints' unions are made not of rigid materials, but of plastic; avoiding vibration problems coming from complex configurations, and not breaking up the robot while experimenting.

4.3.2 Architecture of the closed-loop

Each of the six actuators' system will present its own control strategy, independent from the rest of them, represented by the block diagram shown in Fig. 39. In order to set the configuration of the end-effector quickly and fluidly, a closed-loop controller will be placed after the error signal, which will adjust the position signal for the actuators' rotor.

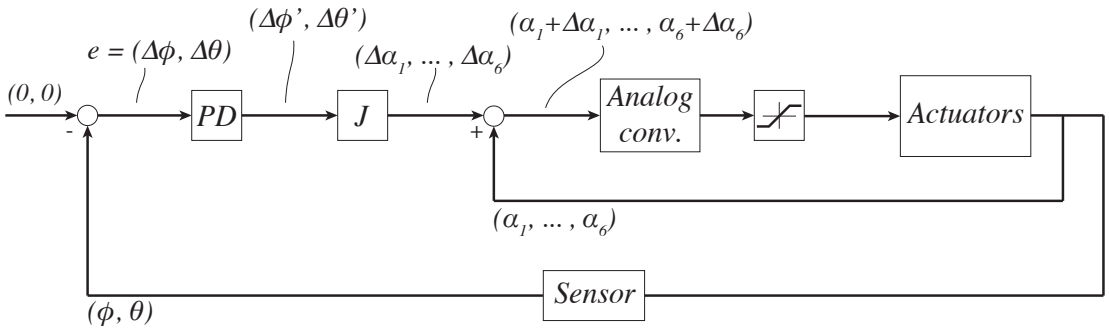

Figure 39: Block diagram of the system's closed-loop

As it is a tiny and light prototype, the resistance forces due to robot's inertia (sec. 1.2.3) can be overlooked, as well as the resistance forces due to gravity.

4.3.3 Controller

Considering the final strategy applied for the element set, the last stage of this project is to develop a control system in to accurately perform and enhance robustness. For this, a *PID controller* would be ideal since their adjustment can be empirically done through several methods and they are significantly easy to perform. It must be noted that 100Hz will be the sampling frequency, as it is imposed by the IMU's FIFO rate.

This controller is a control loop feedback mechanism, which aims to manipulate an internal variable to reduce or erase the error of the system and to influence in the dynamic response between stable states. It is done through the continuous calculation of an error parameter $\boldsymbol{e}$ as a differential value between a desired target point and a measured process variable. These corrections are based on three terms:

- ***Proportional (P)***: This term accounts for present values of the error, and induces a quicker (or slower) response.
- ***Integral (I)***: This term accounts for past values of error, and aims to reduce the error of the final state.
- ***Derivative (D)***: This term accounts for possible future trends of the error based on its current rate of change. Its main goal is to induce a fluid operation of the system, avoiding oscillations or possible instability.

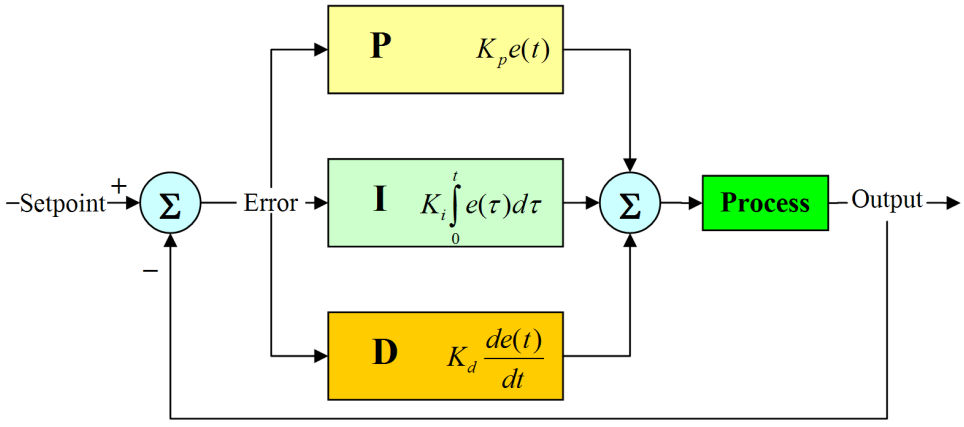

Figure 40: Block diagram of a closed-loop PID controller

The weighted sum of these three terms is used to adjust to the behavior of the system. Therefore, the controller only influences in the control variable e, which means that it does not need to know how the entire system works. This makes this type of controller a really versatile implementation.

These three terms can work separately, or together assembled in any combination, depending on what it is desired to be controlled. However, the omission of any of the terms must be well justified.

Controller setting

As far as this project is concerned, it seems not possible to verify the accuracy of the robot since all the information regarding the status of the system is stored in the Arduino® board and it is not possible to be extracted. The **serial port** cannot be connected to the actuators and to the PC at the same time, which means that the IMU's values cannot be showed and analyzed in order to build a complete PID controller and reduce the error, in addition to a well performance. Therefore, the implementation of the *Integral term* may not have sense in the controller's development as the exact error *(e)* can not be known, nor analyzed.

This leads to develop a simple **PD controller** to basically control the operation speed and the cushioning in order not to destabilize the system. Besides, its adjustment must be arranged manually for the same reason.

Chapter 5

Results

Pictures from the final results are shown below. In the Conclusion chapter, the discussion of results is done quantitatively, as the final values can not be observed and analyzed.

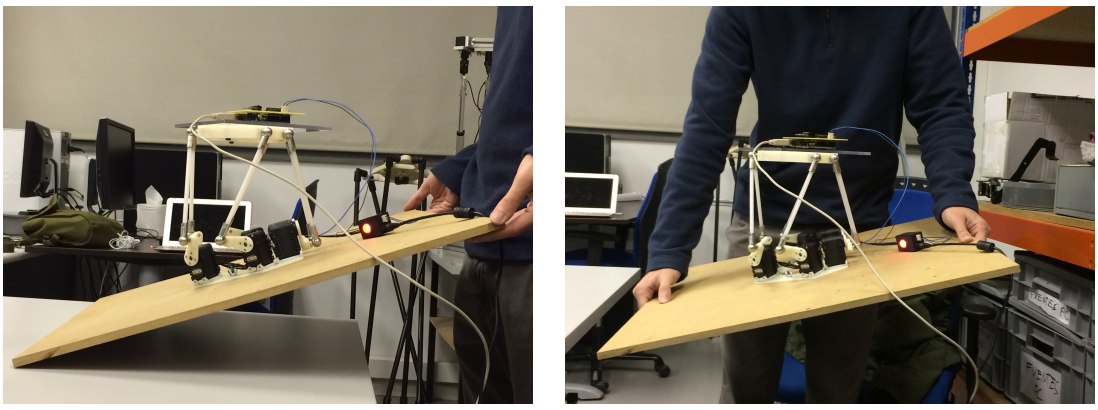

Figure 41: Final experimetnal results

In the previous images, the recovery of the platform's orientation can be seen whenever the base is tilted. The wood base attached to the robot was used to reduce the inertial forces caused by the Dynamixel™'s actuators, preventing the system from sinusoidal entries.

Chapter 6

Environmental impact

The environmental impact caused by this project is minimal, since it started from an already built robot and worked with previously assembled devices (Arduino®, GY-512 board [MPU-6050™], Dynamixels™ AX-12+). The only thing generated in the laboratory was the communication drive to connect the Dynamixels™ AX-12+ to the Arduino® board. Nevertheless, the ISO regulation that specifies the possible environmental impact turned really difficult to find.

Besides that, there has been no need to generate anything other than the paper used for printing the descriptive report to facilitate the checking and the correction. Moreover, some of the device's companies ensure compliance with environmental regulations, such as the Arduino® and InvenSense, which products are *RoHS* and Green compliant.

Chapter 7

Conclusions

7.1 Contributions

In this project a system to stabilize a moving platform to a given orientaton has been developed using a *Gough-Stewart* parallel platform with 6 degrees of freedom. To this end, microcontroller Arduino® has been programmed, to obtain the required values for the actuators, according to the measurements obtained using an Inertial Measurement Unit sensor. This problem essentially reduces to solve the inverse kinematics of the used platform within the workspace of the robot in a feasible way given the limitations of the Arduino®. Despite the lack of a system able to qualitatively detect the robot performance in order to enhance its control, the parallel robot performs correctly in all performed experiments.

7.2 Possible enhancements and future work

Throughout the development of this project many complications have emerged. Most of them connected to the devices which, in more than one occasion, have had to be replaced in a quite advanced point of the project. These last-minute changes have introduced important delays. The limited time and the choice of the devices, still not the most appropriate, have made us take the most pragmatic choice to have a system working at the end of the project. Here it is an enumeration of the possible improvements which should be implemented to have a more robust system if someone else wants to continue with this researching project.

1. Arduino® UNO is clearly the *bottleneck* in this project because of its limited computational capacity. This particular model only provides a single serial port, which has been proved to be insufficient if we want to connect it to a PC and to other devices simultaneously through the serial port. This has prevented the visualization of the different variables during a significant part of the project, as well as the adjustment of a PID controller. The computational capacity of the microcontroller has also introduced limitations in the computation of the inverse kinematics. To improve this, other alternatives of electronic hardware should be studied, such as the Arduino® Mega.

2. The goal of this project has been to compensate the orientation of the platform prototype, which means to work on two generalized coordinates. In some way, this is understood as an untapped potential by the robot since it provides 6 degrees of freedom and only two are used. A possible future study is to develop an stabilization system which compensate both rotations and translations, as well as the possibility of changing the rotation center. This would maintain the platform completely static, although in a reduced workspace.

3. Another improvement to increase the compensation possibilities of the developed robot, a magnetometer could be introduced along with the IMU sensor to minimize the *drifting* problem. Therefore, the *Yaw* angle could be also stabilized.

4. Concerning to Rowberg's code for the IMU sensor, it would be an interesting task to fully understand all possibilities it provides. In this project, this code has been treated as a *black box* as it would take a significant and unavailable amount of time to completely understand it. However, mastering this code would have meant a completely different development of the project. The lack of time to fully understand this code was the main reason why other alternatives were planned, instead of the real resolution, which led to apply approximations and assumptions that slightly hindered the performance of the robot.

5. In case Jeff Rowberg's code is kept as a *black box*, it would be interesting to find a way to avoid the linearization of the equations to reduce the approximation errors.

6. According to the assumptions taken in section 1.2, the dynamics and possible perturbations that could affect somehow the system were not taken into account. The dynamics of the system should be included in the analysis to assess its influence. In the same way, interactions with the different mechanical parts, such us the joints, should be considered. In this scenario, more powerful actuators could be implemented.

Bibliography

[1] J.-P. Merlet, *"Parallel robots,"* Springer, Second edition, pp. 95-102, 2006.

[2] P.I. Corke, *"Robotics, Vision & Control,"* Fundamental algorithms in MATLAB: Second edition, Springer, 2011.

[3] Aginaga García, J, *"Análisis de precisión de manipuladores paralelos,"* (Doctoral Thesis), 2011. Recovered from: `http://www.imac.unavarra.es/jokin/web/pages/curriculum/docs/TesisJokin.pdf`

[4] Samuel R. Buss, *"Introduction to Inverse Kinematics with Jacobian Transpose, Pseudoinverse and Damped Least Squares methods," IEEE Trans.*, 2004. Recovered from: `https://groups.csail.mit.edu/drl/journal_club/papers/033005/buss-2004.pdf`

[5] Bajd, T; Mihelj, M; Lenarčič, J; Stanovnik, A and Munih, M, *"Robotics,"*, Springer, 2010, pp. 9-22.

[6] John J. Craig, *"Robótica,"* PEARSON EDUCACIÓN, Third edition, Mexic, 2006, pp. 62-164.

[7] Rajoy Niubo, A, *Planificación y ejecución de trayectorias libres de singularidades en robots paralelos 3-RRR,* (Master Thesis), 2015.

[8] Bonet Muñoz, A, *"Control de la posició i balanceig d'una bola sobre un pla emprant una plataforma del tipus Gough-Stewart,"* (Degree Thesis), 2015.

[9] J. M. Hilkert, *"Inertially stabilized platform technology: concepts and principles"*, IEEE Control Systems Magazine, vol. 28, 2008.

[10] M. K. Masten, *"Inertially stabilized platforms for optical imaging systems,"*, IEEE Control Systems, vol. 28, 2008.

[11] Jiménez Calvo, J., *"Diseño y desarrollo de una plataforma estabilizadora para una cámara embarcada en un vehiculo,"* (Degree Thesis), Madrid, 2014. Recovered from: `https://www.iit.comillas.edu/pfc/resumenes/53be49cf8f07d.pdf`

[12] "Gimbal Lock." `https://en.wikipedia.org/wiki/Gimbal_lock`

[13] "Gyroscope's drift." `https://en.wikipedia.org/wiki/Gyroscope`

[14] "MPU-6050®'s register maps and descriptions." https://store.invensense.com/Datasheets/invensense/RM-MPU-6000A.pdf

[15] "MPU-6050®, Motion sensors introduction." https://store.invensense.com/datasheets/invensense/Sensor-Introduction.pdf

[16] "Calibration code for the MPU-6050®." https://turnsouthates.wordpress.com/2015/07/31/arduino-mpu6050/

[17] "ROBOTIS e-Manual for Dynamixel™ AX-12A/AX-12+." http://support.robotis.com/en/product/actuator/dynamixel/ax_series/dxl_ax_actuator.htm

[18] "ROBOTIS user's manual for Dynamixel™." http://www.trossenrobotics.com/images/productdownloads/AX-12(English).pdf

[19] "Dynamixel™ AX-12A/AX-12+ Datasheet." http://www.pishrobot.com/files/products/datasheets/dynamixel_ax-12a.pdf

[20] "MPU-6050®'s Datasheet." http://www.invensense.com/wp-content/uploads/2015/02/MPU-6000-Datasheet1.pdf

[21] "74F244 Driver Datasheet." http://www.mouser.com/ds/2/149/fairchild%20semiconductor_74f244-608438.pdf

[22] "Arduino®'s web page." https://www.arduino.cc/

[23] "MathWorks web page." http://es.mathworks.com/?s_tid=gn_logo

[24] "Jeff Rowberg's code." https://github.com/jrowberg/i2cdevlib/tree/master/Arduino/MPU6050

[25] "P. Corke's Robotics toolbox." http://www.petercorke.com/Robotics_Toolbox

[26] "Eigen library" https://eigen.tuxfamily.org/dox/GettingStarted.html

Appendix A

Budget

This appendix is addressed to provide an estimation of the development cost of this project, which includes the construction of the parallel robot, the electronic devices and the used tools. Next, the materials and electronic components, as well as their price and number of units are enumerated. The costs are expressed in Euros (€), at the presentation date of the report, without applied discounts.

Considering an engineering salary (40€/h), labour costs specified in table (6) rise to 14,400€. Therefore, total costs involving also mechanical material (table 4) and electronic material (table 5) are **15,443.748€**

Ref.	Description	Provider	Unitary price	Qty.	Total price
680-230	White nylon rod, 1000mmx6mm	RS Pro	8.22	6	49.32
689-401	Steel M6 Ball and Socket Joint	RS Pro	4.118	12	49.41
423-9555	Velcro black, 20mmx5cm	RS Pro	0.824	1	0.824
281-035	Plain socket screw, M4x12mm	RS Pro	0.279	6	1.674
525-925	Plain washer, M4x0,8mm	RS Pro	0.134	3	0.402
521-850	Wingnut, M4	RS Pro	0.172	3	0.516
837-284	Square nut, M4	RS Pro	0.0862	3	0.2586
	Methacrylate plate + cut		50.00	1	50.00
	Printed fixed base		50.00	1	50.00
	Printed moving plataforma		60.00	1	60.00
	Total mechanical material				262.404

Table 4: Mechanical material for the robot construction

Ref.	Description	Provider	Unitary price	Qty.	Total price
A000066	Arduino® UNO board	Diotronic,S.A.	20.57	1	20.57
RB-Suf-16	MPU-6050™ (GY-512 breakout board)	RobotShop inc.	6.99	1	6.99
DXL0001	Dynamixel™ AX-12A kit	Robótica Global,S.L.	68	6	408
2492	Dynamixel™ cable 10cm	Robótica Global,S.L.	21.95	12	263.4
2126	12V power supply	Robótica Global,S.L.	49.95	1	49.95
210-4542	Breadboard 201x64x18,5mm	RS Pro	23.83	1	23.83
CT2	Perfboard	Diotronic,S.A.	3.23	1	3.23
630-437	IC 74F244	Amidata S.L.U.	0.91	1	0.91
1035025	Electric resistance 10k	Diotronic,S.A.	0.02	2	0.4
758-7494	USB A to USB B cable	RS Pro	2.83	1	2.83
020	DIL20 socket	Diotronic,S.A.	0.15	1	0.15
016	DIL16 socket	Diotronic,S.A.	0.12	1	0.12
797-9105	SPOX™ 5264, connector 3 way	Amidata S.L.U.	0.09	2	0.18
687-7152	SPOX™ 5263, contact	Amidata S.L.U.	0.03	4	0.12
687-8124	SPOX™ 5267, 3 way	Amidata S.L.U.	0.064	1	0.064
896-7620	Molex male pin connector	Amidata S.L.U.	0.04	15	0.6
	Total electronic material				781.344

Table 5: Electronic material for the robot construction

Description	Hours	Total cost
MPU-6050™'s programming and simulations	70	2,800
Dynamixels™'s programming	85	3,400
Communication driver construction	15	600
Inverse kinematics solution and alternatives purposed	125	5,000
Control of the system	65	2,600
Total labour costs		14,400

Table 6: Labour costs

Appendix B

Images and tables' sources

All figures and tables have been taken or created by me except the following ones:

Fig.	Source
2	https://www.ormsdirect.co.za/wenpod-md2-studio-3-axis-gimbal
4	http://playground.arduino.cc/Main/MPU-6050
6	http://www.invensense.com/wp-content/uploads/2015/02/MPU-6000-Datasheet1.pdf
7	https://learn.sparkfun.com/tutorials/what-is-an-arduino
18	http://www.trossenrobotics.com/images/productdownloads/AX-12(English).pdf
19	http://www.trossenrobotics.com/images/productdownloads/AX-12(English).pdf
21	http://www.trossenrobotics.com/images/productdownloads/AX-12(English).pdf
22	http://www.oncomponents.com/uploads/item_gallery/349_item.gif
40	https://en.wikipedia.org/wiki/File:PID-feedback-loop-v1.png

Table 7: Figures' sources

Tab.	Source
1	http://www.pishrobot.com/files/products/datasheets/dynamixel_ax-12a.pdf
2	http://www.pishrobot.com/files/products/datasheets/dynamixel_ax-12a.pdf

Table 8: Tables' sources

Appendix C

Parameters from Fourier series

C.1 Surface plots

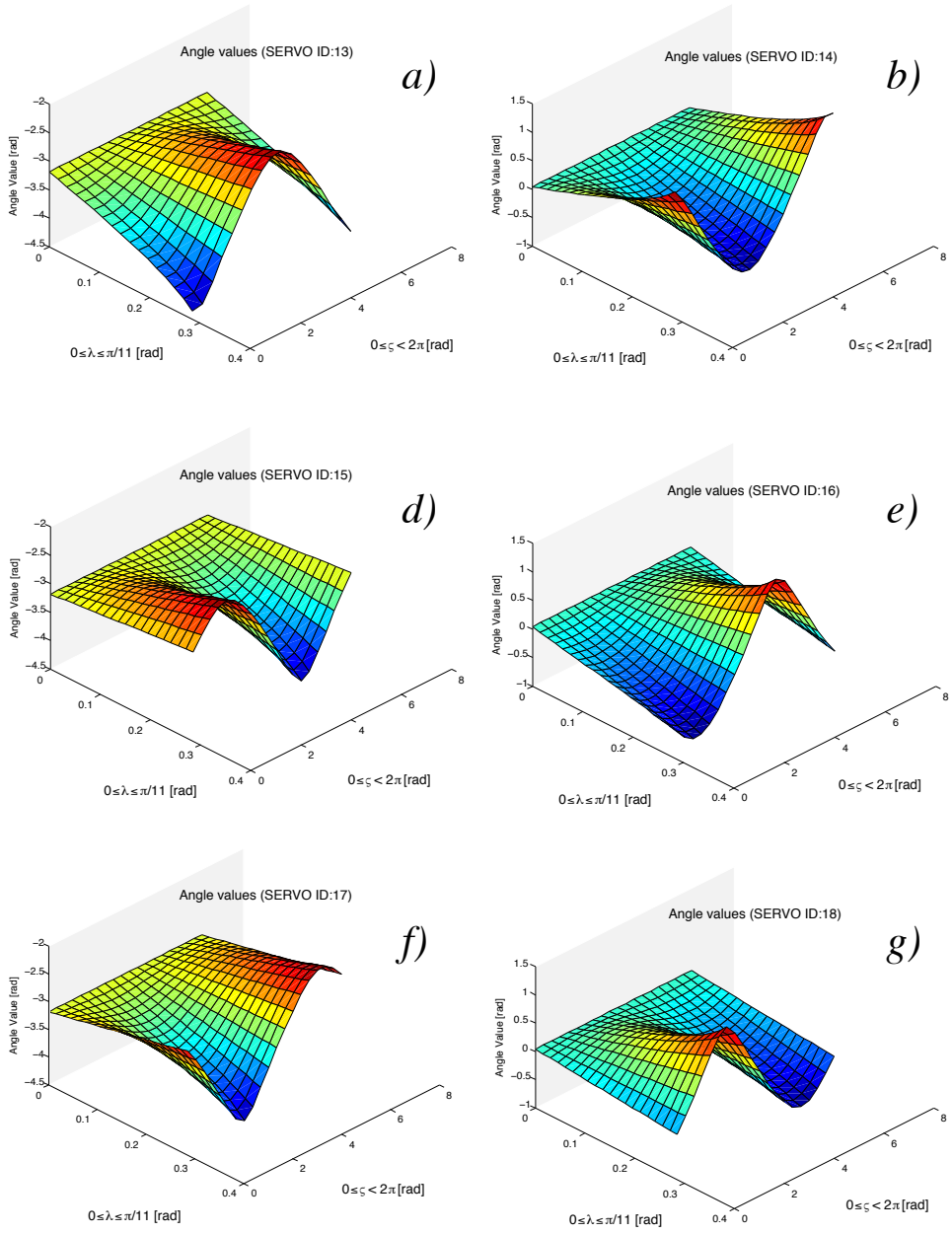

Figure 42: Surface plot, view 1

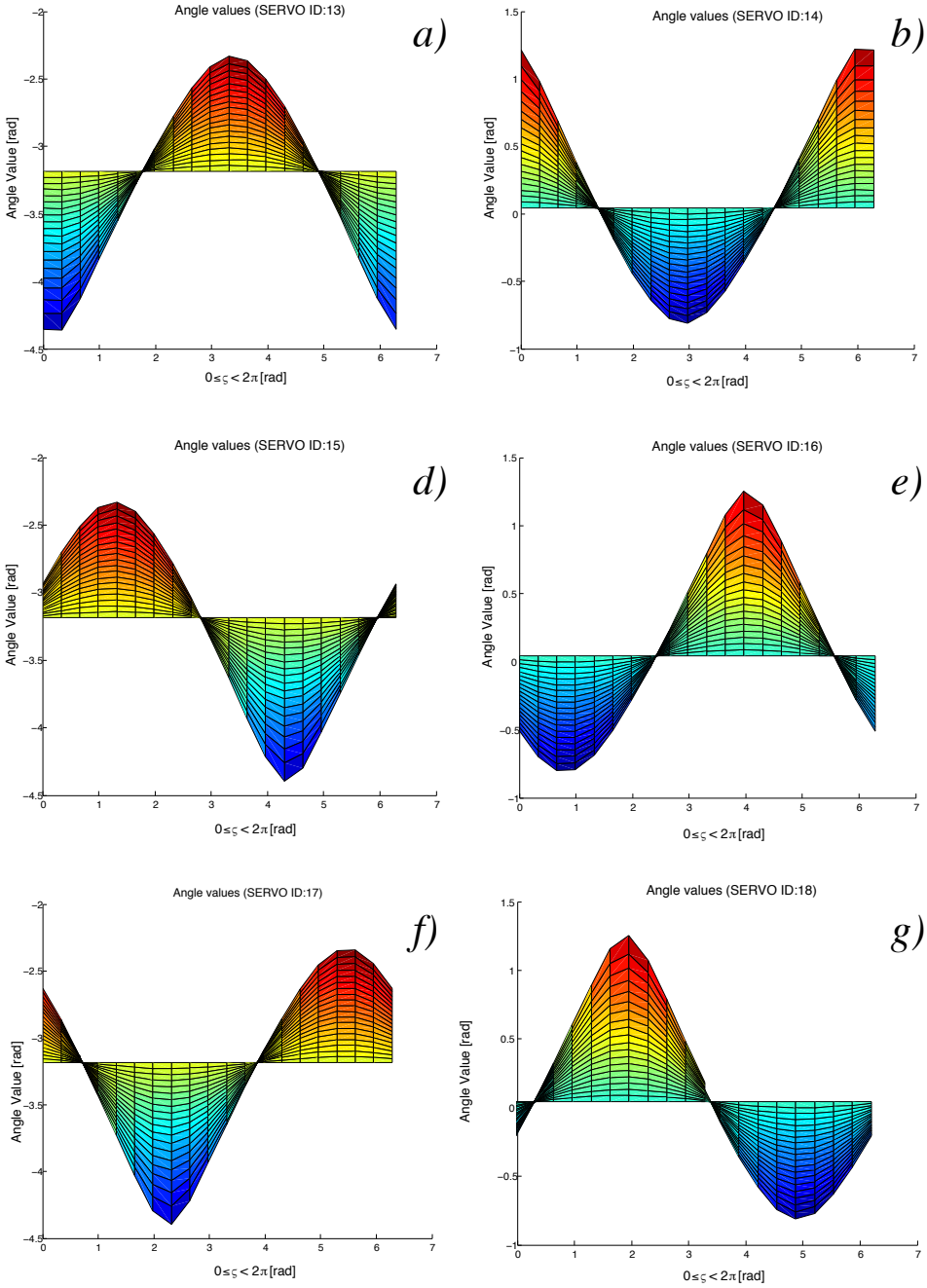

Figure 43: Surface plot, view 2

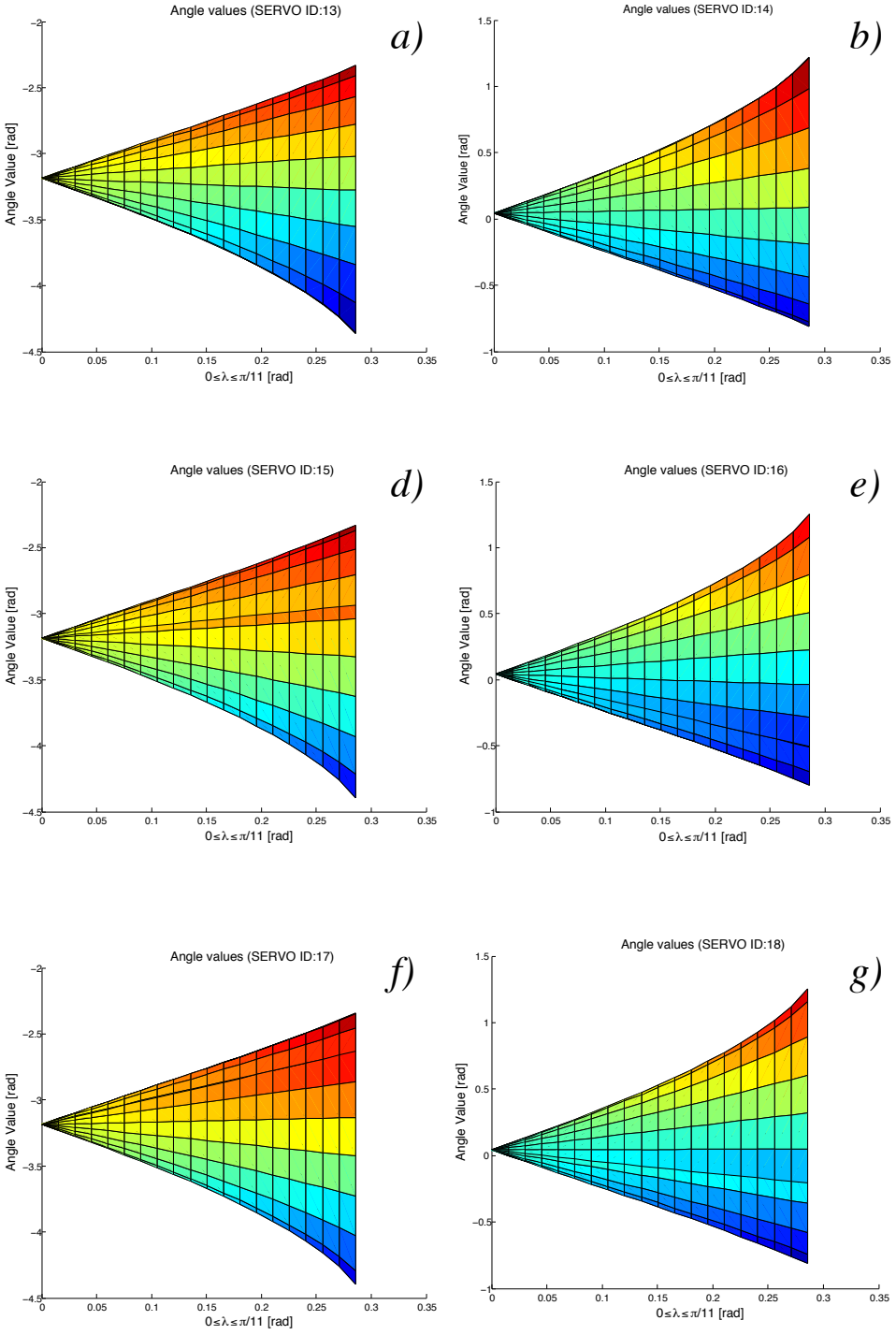

Figure 44: Surface plot, view 3

C.2 Fourier approximations

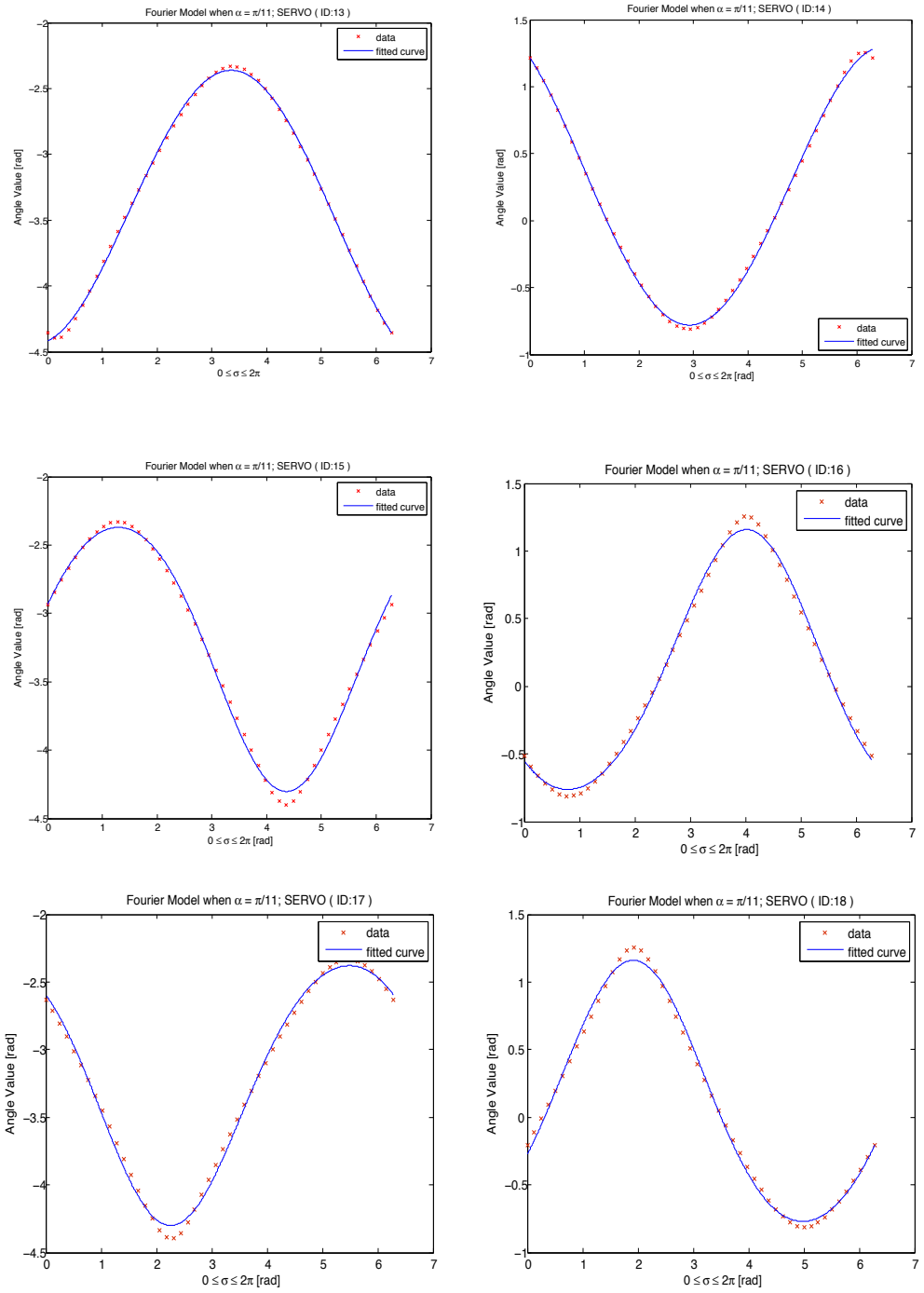

Figure 45: Representation of the fitted curve at $\lambda = \frac{\pi}{11}$

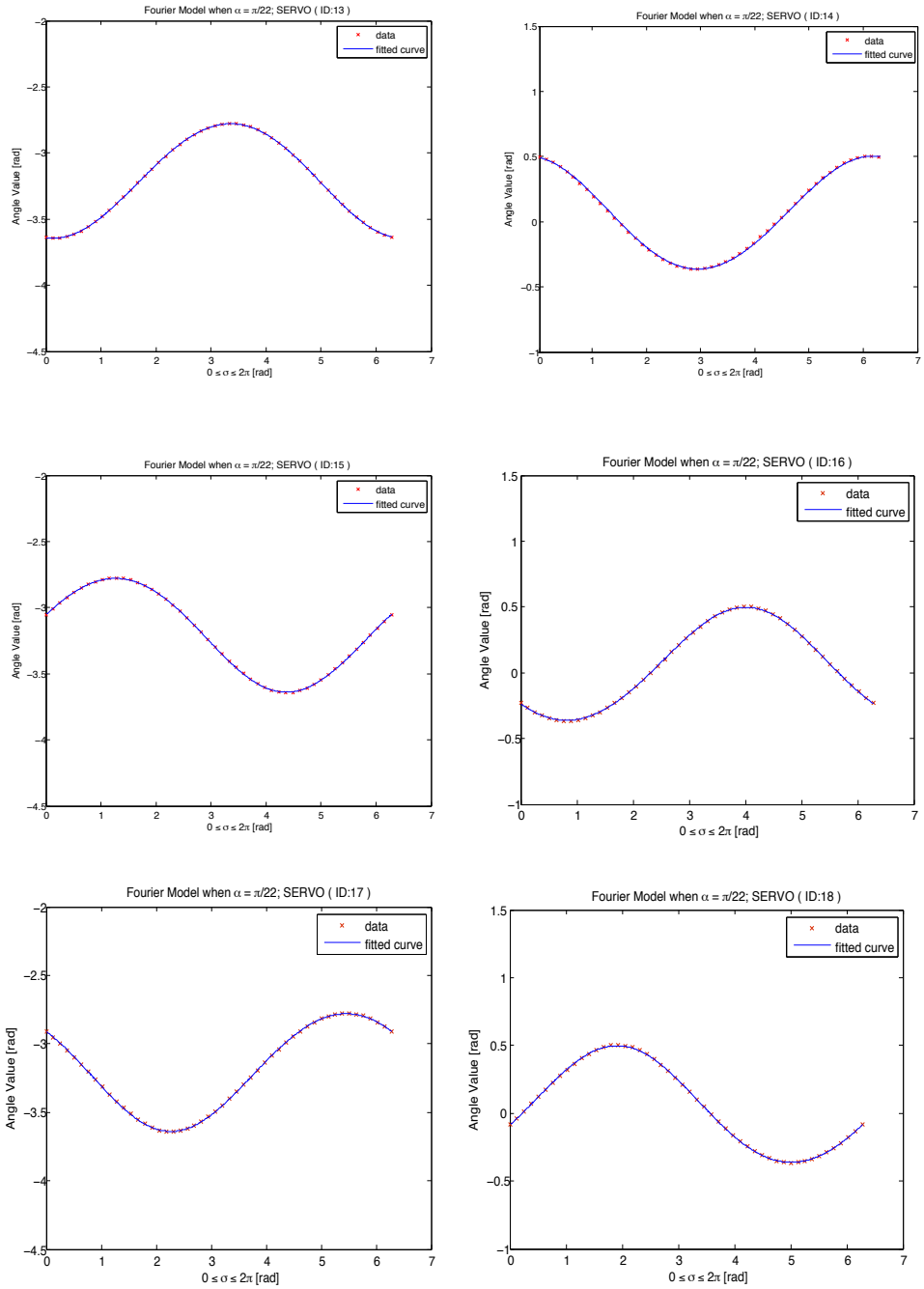

Figure 46: Representation of the fitted curve at $\lambda = \frac{\pi}{22}$

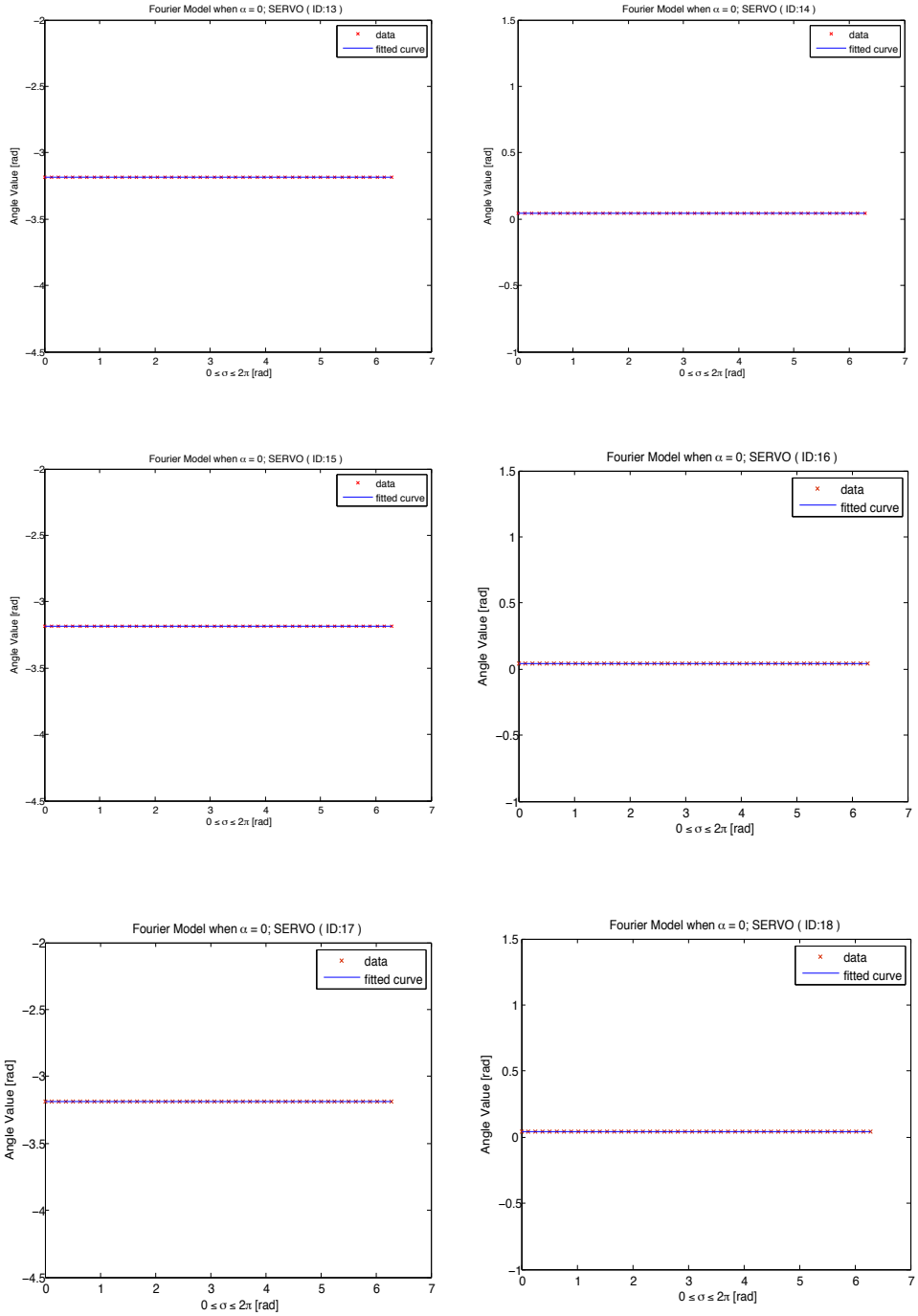

Figure 47: Representation of the fitted curve at λ=0

C.3 Fourier parameters

```
ID:13

%%%%%%%%%%%%%%%%%%ALPHA = PI/11%%%%%%%%%%%%%%%%%%

>> f = fit(sigma',SOL1(:,50),'fourier2')

f =

     General model Fourier2:
     f(x) =  a0 + a1*cos(x*w) + b1*sin(x*w) +
               a2*cos(2*x*w) + b2*sin(2*x*w)
     Coefficients (with 95% confidence bounds):
       a0 =      -2.195  (-7.52, 3.131)
       a1 =     -0.6818  (-4.781, 3.418)
       b1 =      -2.082  (-9.903, 5.739)
       a2 =      -1.543  (-2.745, -0.3407)
       b2 =       1.308  (-2.506, 5.122)
       w =      0.3596  (0.2061, 0.513)


>> plot(f,sigma,SOL1(:,50))


%%%%%%%%%%%%%%%%%%ALPHA = PI/22%%%%%%%%%%%%%%%%%%

>> f = fit(sigma',SOL1(:,50),'fourier2')

f =

     General model Fourier2:
     f(x) =  a0 + a1*cos(x*w) + b1*sin(x*w) +
               a2*cos(2*x*w) + b2*sin(2*x*w)
     Coefficients (with 95% confidence bounds):
       a0 =      -3.204  (-3.206, -3.202)
       a1 =     -0.4283  (-0.4304, -0.4262)
       b1 =    -0.06436  (-0.07049, -0.05823)
       a2 =   -0.009128  (-0.01069, -0.007562)
       b2 =    0.001765  (0.0007271, 0.002803)
       w =      0.9854  (0.981, 0.9897)


%%%%%%%%%%%%%%%%%%ALPHA = 0%%%%%%%%%%%%%%%%%%%%%%%


f =

     General model Fourier2:
     f(x) =  a0 + a1*cos(x*w) + b1*sin(x*w) +
               a2*cos(2*x*w) + b2*sin(2*x*w)
     Coefficients (with 95% confidence bounds):
       a0 =      -3.185  (-3.185, -3.185)
       a1 =  -2.196e-14  (-1.906e-12, 1.862e-12)
       b1 =  -2.892e-14  (-1.595e-12, 1.537e-12)
       a2 =   -2.86e-15  (-1.256e-13, 1.199e-13)
       b2 =    9.58e-15  (-6.777e-13, 6.969e-13)
       w =        0.25  (-4.001, 4.501)
```

```
ID: 14

%%%%%%%%%%%%%%%%%ALPHA=PI/11%%%%%%%%%%%%%%%%%%%%%

f =

     General model Fourier2:
     f(x) =  a0 + a1*cos(x*w) + b1*sin(x*w) +
               a2*cos(2*x*w) + b2*sin(2*x*w)
     Coefficients (with 95% confidence bounds):
       a0 =     -0.9467  (-6.272, 4.379)
       a1 =       1.175  (-4.042, 6.391)
       b1 =       1.849  (-5.154, 8.853)
       a2 =      0.9861  (0.8502, 1.122)
       b2 =      -1.766  (-5.579, 2.048)
       w =      0.3596  (0.2061, 0.513)

%%%%%%%%%%%%%%%%%ALPHA=PI/22%%%%%%%%%%%%%%%%%%%%%

f =

     General model Fourier2:
     f(x) =  a0 + a1*cos(x*w) + b1*sin(x*w) +
               a2*cos(2*x*w) + b2*sin(2*x*w)
     Coefficients (with 95% confidence bounds):
       a0 =      0.0621  (0.06011, 0.06409)
       a1 =      0.4206  (0.4197, 0.4215)
       b1 =     -0.1035  (-0.1091, -0.0978)
       a2 =    0.009297  (0.007849, 0.01075)
       b2 =   6.462e-05  (-0.000836, 0.0009653)
       w =      0.9854  (0.981, 0.9897)

%%%%%%%%%%%%%%%%%ALPHA=PI/0%%%%%%%%%%%%%%%%%%%%%%

f =

     General model Fourier2:
     f(x) =  a0 + a1*cos(x*w) + b1*sin(x*w) +
               a2*cos(2*x*w) + b2*sin(2*x*w)
     Coefficients (with 95% confidence bounds):
       a0 =     0.04348  (0.04348, 0.04348)
       a1 =   -6.22e-17  (-1.372e-16, 1.279e-17)
       b1 =   7.044e-18  (-2.752e-16, 2.893e-16)
       a2 =   2.118e-17  (-4.368e-17, 8.603e-17)
       b2 =  -1.939e-18  (-2.139e-16, 2.101e-16)
       w =           1  (-0.4818, 2.482)
```

```
ID: 15


%%%%%%%%%%%%%%%%%ALPHA=PI/11%%%%%%%%%%%%%%%%%%%%

f =

     General model Fourier2:
     f(x) =  a0 + a1*cos(x*w) + b1*sin(x*w) +
               a2*cos(2*x*w) + b2*sin(2*x*w)
     Coefficients (with 95% confidence bounds):
       a0 =      -3.261  (-3.275, -3.248)
       a1 =      0.2638  (0.2144, 0.3131)
       b1 =       0.929  (0.9025, 0.9554)
       a2 =     0.06046  (0.04182, 0.07909)
       b2 =    -0.04586  (-0.07474, -0.01698)
       w =       1.014  (0.9995, 1.028)


%%%%%%%%%%%%%%%%%ALPHA=PI/22%%%%%%%%%%%%%%%%%%%%

f =

     General model Fourier2:
     f(x) =  a0 + a1*cos(x*w) + b1*sin(x*w) +
               a2*cos(2*x*w) + b2*sin(2*x*w)
     Coefficients (with 95% confidence bounds):
       a0 =      -3.198  (-3.199, -3.197)
       a1 =      0.1344  (0.1313, 0.1374)
       b1 =      0.4085  (0.4068, 0.4103)
       a2 =    0.007474  (0.006274, 0.008674)
       b2 =   -0.009917  (-0.0116, -0.008232)
       w =       1.002  (1, 1.004)


%%%%%%%%%%%%%%%%%ALPHA=PI/0%%%%%%%%%%%%%%%%%%%%%

f =

     General model Fourier2:
     f(x) =  a0 + a1*cos(x*w) + b1*sin(x*w) +
               a2*cos(2*x*w) + b2*sin(2*x*w)
     Coefficients (with 95% confidence bounds):
       a0 =      -3.185  (-3.185, -3.185)
       a1 =  -5.463e-14  (-4.674e-12, 4.565e-12)
       b1 =  -6.628e-14  (-3.682e-12, 3.549e-12)
       a2 =  -4.736e-15  (-4.111e-13, 4.016e-13)
       b2 =   2.364e-14  (-1.6e-12, 1.647e-12)
       w =        0.25  (-3.95, 4.45)
```

```
ID: 16

%%%%%%%%%%%%%%%%%%ALPHA = PI/11%%%%%%%%%%%%%%%%%%%

>> f = fit(sigma',SOL4(:,50),'fourier2')

f =

     General model Fourier2:
     f(x) =  a0 + a1*cos(x*w) + b1*sin(x*w) +
               a2*cos(2*x*w) + b2*sin(2*x*w)
     Coefficients (with 95% confidence bounds):
       a0 =      0.1143  (0.09448, 0.134)
       a1 =     -0.6335  (-0.6961, -0.5709)
       b1 =     -0.7205  (-0.7795, -0.6616)
       a2 =    -0.02923  (-0.05756, -0.0008925)
       b2 =     0.08123  (0.04993, 0.1125)
       w =      0.9993  (0.9765, 1.022)



%%%%%%%%%%%%%%%%%%ALPHA = PI/22%%%%%%%%%%%%%%%%%%%

>> f = fit(sigma',SOL4(:,50),'fourier2')

f =

     General model Fourier2:
     f(x) =  a0 + a1*cos(x*w) + b1*sin(x*w) +
               a2*cos(2*x*w) + b2*sin(2*x*w)
     Coefficients (with 95% confidence bounds):
       a0 =     0.05554  (0.05437, 0.05671)
       a1 =     -0.2847  (-0.2881, -0.2813)
       b1 =     -0.3218  (-0.325, -0.3186)
       a2 =   -0.005704  (-0.007132, -0.004275)
       b2 =     0.01219  (0.01043, 0.01396)
       w =      0.9993  (0.9965, 1.002)


%%%%%%%%%%%%%%%%%%ALPHA = 0%%%%%%%%%%%%%%%%%%%%%%%

f =

     General model Fourier2:
     f(x) =  a0 + a1*cos(x*w) + b1*sin(x*w) +
               a2*cos(2*x*w) + b2*sin(2*x*w)
     Coefficients (with 95% confidence bounds):
       a0 =     0.04348  (0.04348, 0.04348)
       a1 =  -1.078e-17  (-1.037e-15, 1.015e-15)
       b1 =   1.058e-16  (-3.534e-15, 3.745e-15)
       a2 =   5.649e-17  (-1.188e-15, 1.301e-15)
       b2 =   2.303e-18  (-1.154e-15, 1.158e-15)
       w =         0.5  (-2.757, 3.757)
```

```
ID: 17


%%%%%%%%%%%%%%%%%ALPHA = PI/11%%%%%%%%%%%%%%%%%%

>> f = fit(sigma',SOL5(:,50),'fourier2')

f =

     General model Fourier2:
     f(x) =  a0 + a1*cos(x*w) + b1*sin(x*w) +
               a2*cos(2*x*w) + b2*sin(2*x*w)
     Coefficients (with 95% confidence bounds):
       a0 =      -3.256  (-3.276, -3.236)
       a1 =      0.6301  (0.5831, 0.6771)
       b1 =     -0.7235  (-0.7626, -0.6843)
       a2 =     0.02998  (0.01143, 0.04852)
       b2 =     0.08096  (0.05584, 0.1061)
       w =      0.9993  (0.9765, 1.022)

>> plot(f,sigma,SOL5(:,50))


%%%%%%%%%%%%%%%%%ALPHA = PI/22%%%%%%%%%%%%%%%%%%

>> f = fit(sigma',SOL5(:,50),'fourier2')

f =

     General model Fourier2:
     f(x) =  a0 + a1*cos(x*w) + b1*sin(x*w) +
               a2*cos(2*x*w) + b2*sin(2*x*w)
     Coefficients (with 95% confidence bounds):
       a0 =      -3.197  (-3.198, -3.196)
       a1 =      0.2832  (0.2805, 0.2859)
       b1 =     -0.3231  (-0.3254, -0.3208)
       a2 =    0.005816  (0.004563, 0.007069)
       b2 =     0.01214  (0.01051, 0.01377)
       w =      0.9993  (0.9965, 1.002)

%%%%%%%%%%%%%%%%%ALPHA = 0%%%%%%%%%%%%%%%%%%%%%%

f =

     General model Fourier2:
     f(x) =  a0 + a1*cos(x*w) + b1*sin(x*w) +
               a2*cos(2*x*w) + b2*sin(2*x*w)
     Coefficients (with 95% confidence bounds):
       a0 =      -3.185  (-3.185, -3.185)
       a1 =  -6.342e-14  (-3.863e-12, 3.736e-12)
       b1 =   -7.31e-14  (-2.947e-12, 2.801e-12)
       a2 =  -3.954e-15  (-3.882e-13, 3.803e-13)
       b2 =   2.747e-14  (-1.284e-12, 1.339e-12)
       w =        0.25  (-2.721, 3.221)
```

```
ID: 18

%%%%%%%%%%%%%%%%%ALPHA = PI/11%%%%%%%%%%%%%%%%%

>> f = fit(sigma',SOL6(:,50),'fourier2')

f =

     General model Fourier2:
     f(x) =  a0 + a1*cos(x*w) + b1*sin(x*w) +
               a2*cos(2*x*w) + b2*sin(2*x*w)
     Coefficients (with 95% confidence bounds):
       a0 =       0.1196  (0.106, 0.1333)
       a1 =      -0.3433  (-0.3846, -0.3021)
       b1 =       0.9026  (0.8826, 0.9226)
       a2 =     -0.05162  (-0.0694, -0.03384)
       b2 =     -0.05562  (-0.07734, -0.03389)
       w =        1.014  (0.9995, 1.028)

>> plot(f,sigma,SOL6(:,50))


%%%%%%%%%%%%%%%%%ALPHA = PI/22%%%%%%%%%%%%%%%%%

>> f = fit(sigma',SOL6(:,50),'fourier2')

f =

     General model Fourier2:
     f(x) =  a0 + a1*cos(x*w) + b1*sin(x*w) +
               a2*cos(2*x*w) + b2*sin(2*x*w)
     Coefficients (with 95% confidence bounds):
       a0 =      0.05605  (0.05517, 0.05693)
       a1 =      -0.1398  (-0.1425, -0.1371)
       b1 =       0.4067  (0.4054, 0.408)
       a2 =    -0.007208  (-0.008371, -0.006046)
       b2 =     -0.01011  (-0.01166, -0.008559)
       w =        1.002  (1, 1.004)

%%%%%%%%%%%%%%%%%ALPHA = 0%%%%%%%%%%%%%%%%%%%%%


f =

     General model Fourier2:
     f(x) =  a0 + a1*cos(x*w) + b1*sin(x*w) +
               a2*cos(2*x*w) + b2*sin(2*x*w)
     Coefficients (with 95% confidence bounds):
       a0 =      0.04348  (0.04348, 0.04348)
       a1 =   -2.238e-17  (-1.027e-16, 5.794e-17)
       b1 =    5.408e-17  (-1.715e-17, 1.253e-16)
       a2 =    3.909e-17  (-6.261e-17, 1.408e-16)
       b2 =    4.876e-17  (-4.268e-17, 1.402e-16)
       w =           10  (9.746, 10.25)
```

Appendix D

Arduino Code

```
#include <SPI.h>

// I2Cdev and MPU6050 must be installed as libraries, or else the .cpp/.h files
// for both classes must be in the include path of your project
#include "I2Cdev.h"

//#include "MPU6050.h"  // not necessary if using MotionApps include file
#include "MPU6050_6Axis_MotionApps20_prueba.h"


// Arduino Wire library is required if I2Cdev I2CDEV_ARDUINO_WIRE implementation
// is used in I2Cdev.h
#if I2CDEV_IMPLEMENTATION == I2CDEV_ARDUINO_WIRE
    #include "Wire.h"
#endif


MPU6050 mpu;


//###############################################################################
//####  Preprocessor options  ###################################################
//###############################################################################



//###############################################################################
//####  MPU6050  ################################################################
//###############################################################################

#define OUTPUT_READABLE_YAWPITCHROLL  // yaw/pitch/roll angles (in degrees)
#define INTERRUPT_PIN 2               // use pin 2 on Arduino Uno & most boards


//### MPU control/status vars  ##################################################


bool dmpReady = false;  // set true if DMP init was successful
uint8_t mpuIntStatus;   // holds actual interrupt status byte from MPU
uint8_t devStatus;      // return status after each device operation (0 = success, !0 = error)
uint16_t packetSize;    // expected DMP packet size (default is 42 bytes)
uint16_t fifoCount;     // count of all bytes currently in FIFO
uint8_t fifoBuffer[64]; // FIFO storage buffer


//### MPU motion vars ###########################################################

Quaternion1 q;           // [w, x, y, z]         quaternion container
VectorFloat gravity;    // [x, y, z]            gravity vector
float ypr[3];           // [yaw, pitch, roll]  container and gravity vector
double rp[2];           // [roll, pitch, yaw]  container and gravity vector
double lastrp[2]={0,0};
double yaw,pitch,roll;

//###############################################################################
//####   Kinematic Vars and Constants   #########################################
//###############################################################################

double servo[6];
double servoInstruction[6];
double servoAux[6];
int flag;
const double LON=0.8556;
const double L2=0.1723;
const double RADI=0.9899;
const double L1=0.475;
const double LEG=1.68;
const double BRAC=0.3;
const double H=1.6;
const double R=0.2;
```

```
//#################################################################################
//####   Dynamixel AX-12A   ######################################################
//#################################################################################

//####  Control table (AX-12A)  ###################################################

  // EEPROM AREA  ----------------------------------------------------------
#define MX_MODEL_NUMBER_L             0
#define MX_MODEL_NUMBER_H             1
#define MX_VERSION_OF_FIRMWARE        2
#define MX_ID                         3
#define MX_BAUD_RATE                  4
#define MX_RETURN_DELAY_TIME          5
#define MX_CW_ANGLE_LIMIT_L           6
#define MX_CW_ANGLE_LIMIT_H           7
#define MX_CCW_ANGLE_LIMIT_L          8
#define MX_CCW_ANGLE_LIMIT_H          9
#define MX_HIGHEST_LIMIT_TEMPERATURE  11
#define MX_LOWEST_LIMIT_VOLTAGE       12
#define MX_HIGHEST_LIMIT_VOLTAGE      13
#define MX_MAX_TORQUE_L               14
#define MX_MAX_TORQUE_H               15
#define MX_STATUS_RETURN_LEVEL        16
#define MX_ALARM_LED                  17
#define MX_ALARM_SHUTDOWN             18

  // RAM AREA  -------------------------------------------------------------
#define MX_TORQUE_ENABLE          24
#define MX_LED                    25
#define MX_D_GAIN                 26
#define MX_I_GAIN                 27
#define MX_P_GAIN                 28
#define MX_GOAL_POSITION_L        30
#define MX_GOAL_POSITION_H        31
#define MX_MOVING_SPEED_L         32
#define MX_MOVING_SPEED_H         33
#define MX_TORQUE_LIMIT_L         34
#define MX_TORQUE_LIMIT_H         35
#define MX_PRESENT_POSITION_L     36
#define MX_PRESENT_POSITION_H     37
#define MX_PRESENT_SPEED_L        38
#define MX_PRESENT_SPEED_H        39
#define MX_PRESENT_LOAD_L         40
#define MX_PRESENT_LOAD_H         41
#define MX_PRESENT_VOLTAGE        42
#define MX_PRESENT_TEMPERATURE    43
#define MX_REGISTERED             44
#define MX_MOVING                 46
#define MX_LOCK                   47
#define MX_PUNCH_L                48
#define MX_PUNCH_H                49

//####  Instruction Set  ##########################################################

#define MX_PING             1
#define MX_READ_DATA        2
#define MX_WRITE_DATA       3
#define MX_REG_WRITE        4
#define MX_ACTION           5
#define MX_RESET            6
#define MX_SYNC_WRITE       131

//####  Others  ###################################################################

#define BROADCAST_ID          254
```

```
//#############################################################################
//####  Arduino  ##############################################################
//#############################################################################


//####  Dynamixel bus communication modes by driver 74F244  ###############

#define TX_MODE                   1
#define RX_MODE                   2
#define HIGH_IMPEDANCE_MODE       3


//####  Pin assignments  ##################################################

const byte RX_PIN_ENABLE = 7;
const byte TX_PIN_ENABLE = 4;

#define LED         13


//#############################################################################
//####  6S Robot  #############################################################
//#############################################################################


//####  Data arrays  ##########################################################

byte InstructionPacket[24];   // Array containing the Instruction Packet frame
byte StatusPacket[24];        // Array containing the Status Packet frame

//int SetPoints[7];             // Set points of a sample


//####  Variables  ############################################################

unsigned int lengthTable;


byte state = 0;
unsigned long timeCounterMicros;
unsigned long timeCount;

#define STATUS_PACKET_TIMEOUT     10    // Time out to receive the Status Packet (in
milliseconds)

//#############################################################################
//####  Control driver 74F244  ################################################
//#############################################################################

void communication_mode(unsigned char comm_mode){

  switch (comm_mode){
    case 1:       // case transmission mode
      digitalWrite(RX_PIN_ENABLE, HIGH);
      digitalWrite(TX_PIN_ENABLE, LOW);
      break;
    case 2:       // case reception mode
      digitalWrite(TX_PIN_ENABLE, HIGH);
      digitalWrite(RX_PIN_ENABLE, LOW);
      break;
    case 3:       // case high impedance mode
      digitalWrite(RX_PIN_ENABLE, HIGH);
      digitalWrite(TX_PIN_ENABLE, HIGH);
      break;
  }
}
```

```
//#############################################################################
//####  Servomotor Dynamixel AX-12A functions  ################################
//#############################################################################

byte mxPing(byte id){
  return instructionStatus(id, 2, MX_PING);
}
//_____________________________________________________________________________

byte instructionStatus(byte id, byte length, byte instruction){
  /*  Transmit a Instruction Packet frame and receive the corresponding Status Packet frame from
the servo.

    Return :  0 > The transmission-reception was ok.
              1 > Communication error between Arduino-Servo.
  */

  byte answer = 128;    // Initialized as 128 because it's not a possible answer of the
statusPacket function
  byte counter = 0;

  do{
    instructionPacket(id, length, instruction);   // Transmit the Instruction Packet frame
    answer = statusPacket();            // Receive the Status Packet frame
    counter++;
  }while (answer != 0  &&  counter <= 2);      // Makes three attempts before considering a
communication error

  if (counter == 3){ return 1; }    //Third attempt: COMMUNICATION ERROR

  return 0;
}
//_____________________________________________________________________________
```

```
void instructionPacket(byte id, byte length, byte instruction){
  /*  Transmit a Instruction Packet frame from the InstructionPacket array to the specified
servo.

    The structure of the InstructionPacket array is:
      InstructionPacket[] = {0xFF, 0xFF, id, length, instruction, parameter 1, ..., parameter N,
checkSum}
        0xFF, 0xFF : Header (defined in the setup function).
        id : Identification number of the servo. [0..253]. Broadcasting ID: 254.
        length : Number of parameters + 2.
        instruction : 1 : PING
                      2 : READ_DATA
                      3 : WRITE_DATA
                      4 : REG WRITE
                      5 : ACTION
                      6 : RESET
                    131 : SYNC WRITE
                (http://support.robotis.com/en/product/dynamixel/communication/dxl_packet.htm)
        parameters : parameters needed for each instruction.
        checkSum : byte to check the consistency of the sent data.

    NOTE: The parameters must be written in the InstructionPacket array  before calling this
function.
        InstructionPacket[5] must hold the first parameter.
  */

  byte numParameters = length - 2;  // Length = number of parameters(bytes) + 2
  byte lengthPacket = length + 4;   // Bytes of the Instruction Packet
  unsigned int sumParameters = 0;   // Sum of the parameters, needed to compute the checkSum
byte

  communication_mode(TX_MODE);

  for(byte i=0; i<numParameters; i++){      // Sum of parameters to compute the checksum byte.
    sumParameters += InstructionPacket[5+i];  //   InstructionPacket[5] contains the first
parameter.
  }

  InstructionPacket[2] = id;
  InstructionPacket[3] = length;
  InstructionPacket[4] = instruction;
  InstructionPacket[lengthPacket-1] = (~(id + length + instruction + sumParameters) & 0xFF);  //
CheckSum

  while (Serial.available() > 0){          // Empty the reception register
    Serial.read();
  }

  noInterrupts();
  Serial.write(InstructionPacket,lengthPacket);    // Transmit the Instruction Packet
  Serial.flush();                                  // Wait for the transmission finish
  interrupts();

  communication_mode(HIGH_IMPEDANCE_MODE);
}
```

```
//_____________________________________________________________________________


byte statusPacket(void){
  /*  Receive the Status Packet frame from the servo, and save it in the StatusPacket array with
the structure:
      StatusPacket[] = {head1, head2, id, length, error, parameter 1, ..., parameter N,
checkSumReceived}
        id : Identification number of the servo which transfers the status packet.
        length : Number of parameters + 2.
        error : error status occurred during the servo operation.
          0 : No error.
          bit 0 = 1 : Input voltage out of range.
          bit 1 = 1 : Goal position out of range.
          bit 2 = 1 : Internal temperature out of range.
          bit 3 = 1 : Command out of the range.
          bit 4 = 1 : Incorrect Checksum in the transmitted Instruction Packet frame.
          bit 5 = 1 : Current load cannot be controlled by the set Torque.
          bit 6 = 1 : Undefined instruction or action command without reg_write command.
          (http://support.robotis.com/en/product/dynamixel/communication/dxl_packet.htm)
        parameters : Returned parameters. First parameter at StatusPacket[5].
        checkSum : Byte to check the consistency of the received data.

    The function return:
      0  Correct reception of the Status Packet
      1  Error byte activated
      2  Time out when waiting for the first five bytes of the Status Packet
      3  Time out when waiting for the parameters or checkSum of the Status Packet
      4  CheckSum error in Status Packet received
      5  Header not found in Status Packet received
  */

  byte head1, head2, id, length, error, checkSumReceived;
  byte checkSum, numParameters;
  unsigned int sumParameters = 0;
  unsigned long timeCounter;

  communication_mode(RX_MODE);

  for(byte i=0; i < sizeof(StatusPacket); i++){     // Erase the StatusPacket array
    StatusPacket[i] = 0;
  }
```

```
  for(byte i=0; i < sizeof(StatusPacket); i++){     // Erase the StatusPacket array
    StatusPacket[i] = 0;
  }

  timeCounter = millis() + STATUS_PACKET_TIMEOUT;     //millis() - time since the program
started
                                                      //Give to the counter 10ms max for the
communication

  // Wait for the first 5 bytes of the Status Packet {head1, head2, id, length, error}
  while(Serial.available() < 5){
    if (millis() >= timeCounter){                     //if the time exceeds the 10ms given to
read the 5 firsts bytes
      communication_mode(HIGH_IMPEDANCE_MODE);
      for(byte i=0; i<Serial.available(); i++){
        StatusPacket[i] = Serial.read();
      }

      return 2;
    }
  }

  head1 = Serial.read();              // Head 1
  head2 = Serial.read();              // Head 2
  id = Serial.read();                 // Id
  length = Serial.read();             // Length
  error = Serial.read();              // Error

  StatusPacket[0] = head1;
  StatusPacket[1] = head2;
  StatusPacket[2] = id;
  StatusPacket[3] = length;
  StatusPacket[4] = error;

  if((head1 != 0xFF) || (head2 != 0xFF)){
    communication_mode(HIGH_IMPEDANCE_MODE);

    return 5;
  }
```

```
  numParameters = length - 2;

  // Wait for the reception of the Status Packet parameters and checkSum
  while(Serial.available() < (numParameters+1)){
    if (millis() >= timeCounter){                          //if the time exceeds the 10ms given
to complete the communication
      communication_mode(HIGH_IMPEDANCE_MODE);
      for(byte i=5; i<(5+Serial.available()); i++){
        StatusPacket[i] = Serial.read();
      }

      return 3;
    }
  }

  // Read parameters and sum it to compute the checkSum byte
  for(byte i=5; i<(numParameters+5); i++){
    StatusPacket[i] = Serial.read();
    sumParameters += StatusPacket[i];
  }

  checkSumReceived = Serial.read();              // CheckSum received
  checkSum = (~(id + length + error + sumParameters) & 0xFF); // CheckSum computed

  StatusPacket[5+numParameters] = checkSumReceived;

  if (checkSum != checkSumReceived){
    communication_mode(HIGH_IMPEDANCE_MODE);

    return 4;
  }

  communication_mode(HIGH_IMPEDANCE_MODE);

  if (error != 0) { return 1; }

  return 0;
}



//#############################################################################
//####  6S Robot functions  ###################################################
//#############################################################################
byte initialize6S(void){
  int led_counter = 0;

  for(byte i=13; i<=18; i++){
    if (mxPing(i) == 0){
      //INITIALIZE CONTROL TABLE?   #######
      led_counter=led_counter+1;
    }
  }
  if (led_counter > 6){ digitalWrite(LED, HIGH); }
  return 1;
}
//_____________________________________________________________________________

byte stabilize(void){
  InstructionPacket[5] = 0x1E;
  InstructionPacket[6] = 0x00;
  InstructionPacket[7] = 0x02;

  for (byte id=13;id<19;id++){
    instructionStatus(id, 5, MX_WRITE_DATA);
  }
  return 0;
}
//_____________________________________________________________________________
```

```
void try_movement(double angle1, double angle2, double angle3, double angle4, double angle5,
double angle6){
  int a,b,c,d,e,f;
  a=int(angle1);
  b=int(angle2);
  c=int(angle3);
  d=int(angle4);
  e=int(angle5);
  f=int(angle6);

  InstructionPacket[5] = 0x1E;
  InstructionPacket[6] = a%256;
  InstructionPacket[7] = a/256;
  instructionStatus(13, 5, MX_WRITE_DATA);

  InstructionPacket[5] = 0x1E;
  InstructionPacket[6] = b%256;
  InstructionPacket[7] = b/256;
  instructionStatus(14, 5, MX_WRITE_DATA);

  InstructionPacket[5] = 0x1E;
  InstructionPacket[6] = c%256;
  InstructionPacket[7] = c/256;
  instructionStatus(15, 5, MX_WRITE_DATA);

  InstructionPacket[5] = 0x1E;
  InstructionPacket[6] = d%256;
  InstructionPacket[7] = d/256;
  instructionStatus(16, 5, MX_WRITE_DATA);

  InstructionPacket[5] = 0x1E;
  InstructionPacket[6] = e%256;
  InstructionPacket[7] = e/256;
  instructionStatus(17, 5, MX_WRITE_DATA);

  InstructionPacket[5] = 0x1E;
  InstructionPacket[6] = f%256;
  InstructionPacket[7] = f/256;
  instructionStatus(18, 5, MX_WRITE_DATA);

}

//_____________________________________________________________________________

void stopServo(double *servo,double *servoAux){
  servo[0] = servoAux[0];
  servo[1] = servoAux[1];
  servo[2] = servoAux[2];
  servo[3] = servoAux[3];
  servo[4] = servoAux[4];
  servo[5] = servoAux[5];

}


//_____________________________________________________________________________


int mapping(double *servoInstruction,double *servo){
  servoInstruction[0]=servo[0]+M_PI;
  servoInstruction[2]=servo[2]+M_PI;
  servoInstruction[4]=servo[4]+M_PI;

  servoInstruction[0] = (1024*(servoInstruction[0]+M_PI))/(2*M_PI);
  servoInstruction[1] = (1024*(servo[1]+M_PI))/(2*M_PI);
  servoInstruction[2] = (1024*(servoInstruction[2]+M_PI))/(2*M_PI);
  servoInstruction[3] = (1024*(servo[3]+M_PI))/(2*M_PI);
  servoInstruction[4] = (1024*(servoInstruction[4]+M_PI))/(2*M_PI);
  servoInstruction[5] = (1024*(servo[5]+M_PI))/(2*M_PI);
}
```

```
int domainServo(double *servoInstruction,double *servo,double *servoAux){
  for (int i=0; i<6; i++){
    if (servoInstruction[i]>780 or servoInstruction[i]<100){
      return 1;
    }
  }

  servoAux[0] = servo[0];
  servoAux[1] = servo[1];
  servoAux[2] = servo[2];
  servoAux[3] = servo[3];
  servoAux[4] = servo[4];
  servoAux[5] = servo[5];

  return 0;
}

//_____________________________________________________________________________

void kinematic(double *rp,double *lastrp,double *servo){
  //Jacobian matrix close to the (0,0,0) position
  double J0[6][2]={{-2.88187,-0.58007},{2.88187,-0.58015},{0.93771,2.78395},{-1.94247,-2.20386},
{1.94247,-2.20555},{-0.93771,2.78570}};
  double J1[6][2]={{-4.3043,-0.8652},{4.3043,-0.8647},{0.8435,2.5050},{-1.9502,-2.2130},
{1.9502,-2.2126},{-0.8435,2.5051}};
  double J2[6][2]={{-2.8990,-0.5839},{2.8990,-0.5843},{1.0397,3.0859},{-2.2080,-2.5047},
{2.2080,-2.5096},{-1.0397,3.0898}};
  double J3[6][2]={{-2.9963,-0.5555},{2.8084,-0.5877},{0.8173,2.8666},{-2.0156,-2.0625},
{2.3795,-2.8753},{-1.5652,3.9056}};
  double J4[6][2]={{-2.8084,-0.5877},{2.9963,-0.5557},{1.5652,3.8921},{-2.3795,-2.8692},
{2.0156,-2.0622},{-0.8173,2.8635}};
  double J5[6][2]={{-5.3259,-1.0030},{3.7463,-0.7738},{0.7320,2.9539},{-2.5192,-2.7109},
{2.0159,-2.3275},{-1.3220,2.8881}};
  double J6[6][2]={{-2.8118,-0.5176},{3.0678,-0.6453},{0.9046,2.8174},{-1.9871,-1.9344},
{5.0896,-6.5033},{-2.1452,6.2180}};
  double J7[6][2]={{-3.0678,-0.6450},{2.8118,-0.5179},{2.1452,6.1555},{-5.0896,-6.4022},
{1.9871,-1.9360},{-0.9046,2.8165}};
  double J8[6][2]={{-3.7463,-0.7742},{5.3259,-1.0021},{1.3220,2.8837},{-2.0159,-2.3260},
{2.5192,-2.7045},{-0.7320,2.9478}};
  double incservo[6]={0};
  double P = 0.08;              // <------- mejor a 0.05 (no hay tanta oscilacion)
  double D = 0.00001;
  double sumaux;


  if (rp[0]<=0.15 and rp[0]>=-0.15 and rp[1]<=0.15 and rp[1]>=-0.15){
    for (int i=0;i<6;i++){
      sumaux=0;
      for (int j=0;j<2;j++){
        sumaux = sumaux + P*J0[i][j]*rp[j] + D*(J0[i][j]*(rp[j]-lastrp[j]))/0.01;
      }
      incservo[i] = sumaux;
    }
          //if close to rp = [0,0]
          //uses this Jacobian
  }
  if (rp[0]>0.15 and rp[1]<=0.15 and rp[1]>=-0.15){
    for (int i=0;i<6;i++){
      sumaux=0;
      for (int j=0;j<2;j++){
        sumaux = sumaux + P*J1[i][j]*rp[j] + D*(J1[i][j]*(rp[j]-lastrp[j]))/0.01;
      }
      incservo[i] = sumaux;
    }
          //if close to rp = [0.2,0]
          //uses this Jacobian
  }
```

```
if (rp[0]<-0.15 and rp[1]<=0.15 and rp[1]>=-0.15){
  for (int i=0;i<6;i++){
    sumaux=0;
    for (int j=0;j<2;j++){
      sumaux = sumaux + P*J2[i][j]*rp[j] + D*(J2[i][j]*(rp[j]-lastrp[j]))/0.01;
    }
    incservo[i] = sumaux;
  }
        //if close to rp = [-0.2,0]
        //uses this Jacobian
}
if (rp[1]>0.15 and rp[0]<=0.15 and rp[0]>=-0.15){
  for (int i=0;i<6;i++){
    sumaux=0;
    for (int j=0;j<2;j++){
      sumaux = sumaux + P*J3[i][j]*rp[j] + D*(J3[i][j]*(rp[j]-lastrp[j]))/0.01;
    }
    incservo[i] = sumaux;
  }
        //if close to rp = [0,0.2]
        //uses this Jacobian
}
if (rp[1]<-0.15 and rp[0]<=0.15 and rp[0]>=-0.15){
  for (int i=0;i<6;i++){
    sumaux=0;
    for (int j=0;j<2;j++){
      sumaux = sumaux + P*J4[i][j]*rp[j] + D*(J4[i][j]*(rp[j]-lastrp[j]))/0.01;
    }
    incservo[i] = sumaux;
  }

        //if close to rp = [0,-0.2]
        //uses this Jacobian
}
if (rp[0]>0.15 and rp[1]>0.15){
  for (int i=0;i<6;i++){
    sumaux=0;
    for (int j=0;j<2;j++){
      sumaux = sumaux + P*J5[i][j]*rp[j] + D*(J5[i][j]*(rp[j]-lastrp[j]))/0.01;
    }
    incservo[i] = sumaux;
  }
        //if close to rp = [0.2,0.2]
        //uses this Jacobian
}
```

```
  if (rp[0]<-0.15 and rp[1]>0.15){
    for (int i=0;i<6;i++){
      sumaux=0;
      for (int j=0;j<2;j++){
        sumaux = sumaux + P*J6[i][j]*rp[j] + D*(J6[i][j]*(rp[j]-lastrp[j]))/0.01;
      }
      incservo[i] = sumaux;
    }
          //if close to rp = [-0.2,0.2]
          //uses this Jacobian
  }
  if (rp[0]<-0.15 and rp[1]<-0.15){
    for (int i=0;i<6;i++){
      sumaux=0;
      for (int j=0;j<2;j++){
        sumaux = sumaux + P*J7[i][j]*rp[j] + D*(J7[i][j]*(rp[j]-lastrp[j]))/0.01;
      }
      incservo[i] = sumaux;
    }
          //if close to rp = [-0.2,-0.2]
          //uses this Jacobian
  }
  if (rp[0]>0.15 and rp[1]<-0.15){
    for (int i=0;i<6;i++){
      sumaux=0;
      for (int j=0;j<2;j++){
        sumaux = sumaux + P*J8[i][j]*rp[j] + D*(J8[i][j]*(rp[j]-lastrp[j]))/0.01;
      }
      incservo[i] = sumaux;
    }
          //if close to rp = [0.2,-0.2]
          //uses this Jacobian
  }


  //ADD THE INCREMENT TO THE SERVOS ANGLES
  for (int k=0;k<6;k++){
    servo[k] = servo[k] - incservo[k];
  }

}


//_____________________________________________________________________________
```

```
// ================================================================
// ===               INTERRUPT DETECTION ROUTINE                ===
// ================================================================

volatile bool mpuInterrupt = false;     // indicates whether MPU interrupt pin has gone high
void dmpDataReady() {
    mpuInterrupt = true;
}


//##########     SETUP     ########################################

void setup() {

//#############################################################################
//####  Arduino Setup Function  ###############################################
//#############################################################################

  pinMode(RX_PIN_ENABLE, OUTPUT);
  pinMode(TX_PIN_ENABLE, OUTPUT);

  pinMode(LED, OUTPUT);
  digitalWrite(LED,LOW);

  communication_mode(HIGH_IMPEDANCE_MODE);

  // Serial pins 1(Tx), 0(Rx) to communicate with the servos
  Serial.begin(1000000);

  // Initialize the SPI bus
  SPI.setClockDivider(SPI_CLOCK_DIV2);    // Set up the SPI speed
  SPI.setBitOrder(MSBFIRST);              // Sets the order of the bits into the SPI bus, most-
significant bit first
  SPI.setDataMode(1);                     // Set up the data mode. CPOL = 0, CPHA = 0 => SPI
mode 1
  SPI.begin();                            // Initialize the SPI bus.


  InstructionPacket[0] = 0xFF;  // Header 1
  InstructionPacket[1] = 0xFF;  // Header 2

  servo[0] = -M_PI;
  servo[1] = 0;
  servo[2] = -M_PI;
  servo[3] = 0;
  servo[4] = -M_PI;
  servo[5] = 0;
```

```
// ================================================================
// ===                      I2C MPU6050 SETUP                   ===
// ================================================================

  // join I2C bus (I2Cdev library doesn't do this automatically)
  #if I2CDEV_IMPLEMENTATION == I2CDEV_ARDUINO_WIRE
      Wire.begin();
      Wire.setClock(400000); // 400kHz I2C clock. Comment this line if having compilation
difficulties
  #elif I2CDEV_IMPLEMENTATION == I2CDEV_BUILTIN_FASTWIRE
      Fastwire::setup(400, true);
  #endif

  // initialize device
  mpu.initialize();
  pinMode(INTERRUPT_PIN, INPUT);

  // load and configure the DMP
  devStatus = mpu.dmpInitialize();

  // supply your own gyro offsets here, scaled for min sensitivity
  mpu.setXGyroOffset(18);
  mpu.setYGyroOffset(7);
  mpu.setZGyroOffset(39);
  mpu.setZAccelOffset(2529); // 1688 factory default for my test chip


  // make sure it worked (returns 0 if so)
  if (devStatus == 0) {
    // turn on the DMP, now that it's ready
    mpu.setDMPEnabled(true);

    // enable Arduino interrupt detection
    attachInterrupt(digitalPinToInterrupt(INTERRUPT_PIN), dmpDataReady, RISING);
    mpuIntStatus = mpu.getIntStatus();

    // set our DMP Ready flag so the main loop() function knows it's okay to use it
    dmpReady = true;

    // get expected DMP packet size for later comparison
    packetSize = mpu.dmpGetFIFOPacketSize();
  }
}


//##############################################################################
//####  Arduino Loop Function  #################################################
//##############################################################################


void loop() {
  if (state == 0){
    //initialize6S();
    stabilize();
    delay(2000);
    state = 10;
  }

  if (state == 10){
    if (!dmpReady) return;
```

```
//#############################################################################

    // wait for MPU interrupt or extra packet(s) available
    while (!mpuInterrupt && fifoCount < packetSize) {

      kinematic(rp,lastrp,servo);
      //SAVE THE ACTUAL ERROR FOR THE DERIVATIVE CONTROL
      lastrp[0] = rp[0];
      lastrp[1] = rp[1];
      mapping(servoInstruction,servo);
      flag = domainServo(servoInstruction,servo,servoAux);
      if (flag==1){
        stopServo(servo,servoAux);
        mapping(servoInstruction,servo);
      }

try_movement(servoInstruction[0],servoInstruction[1],servoInstruction[2],servoInstruction[3],ser
voInstruction[4],servoInstruction[5]);

    }

//#############################################################################

    // reset interrupt flag and get INT_STATUS byte
    mpuInterrupt = false;
    mpuIntStatus = mpu.getIntStatus();

    // get current FIFO count
    fifoCount = mpu.getFIFOCount();

    // check for overflow (this should never happen unless our code is too inefficient)
    if ((mpuIntStatus & 0x10) || fifoCount == 1024) {
        // reset so we can continue cleanly
        mpu.resetFIFO();
        //Serial.println(F("FIFO overflow!"));

    // otherwise, check for DMP data ready interrupt (this should happen frequently)
    } else if (mpuIntStatus & 0x02) {
        // wait for correct available data length, should be a VERY short wait
        while (fifoCount < packetSize) fifoCount = mpu.getFIFOCount();

        // read a packet from FIFO
        mpu.getFIFOBytes(fifoBuffer, packetSize);

        // track FIFO count here in case there is > 1 packet available
        // (this lets us immediately read more without waiting for an interrupt)
        fifoCount -= packetSize;

        #ifdef OUTPUT_READABLE_YAWPITCHROLL
            // display Euler angles in degrees
            mpu.dmpGetQuaternion(&q, fifoBuffer);
            mpu.dmpGetGravity(&gravity, &q);
            mpu.dmpGetYawPitchRoll(ypr, &q, &gravity);

            rp[0]=ypr[2];
            rp[1]=ypr[1];
            //rpy[2]=-ypr[0];

        #endif
      }
  }

}
```

Appendix E

Computing files

E.1 Main execution file

```
%function Stabilize(rpy)
clear all;
close all;

rpy=[-0.2, 0.2, 0.2];
proportional = 1.2;
J = JEBola([0 0 0]);

IKEBola(rpy, 1, -1);

IJ =pinv(JEBola(rpy));

angmotor = IKEBola(rpy, -1, -1)

x(1)=rpy(1);
y(1)=rpy(2);
z(1)=rpy(3);


samples = 30;
for i=1:samples

    incmotor = proportional*J*rpy';

    angmotor = angmotor - incmotor';

    incrpy = IJ*incmotor;

    rpy = rpy - incrpy';

    x(i+1)=rpy(1);
    y(i+1)=rpy(2);
    z(i+1)=rpy(3);


    IJ =pinv(JEBola(rpy));

    IKEBola(rpy, 1, -1);
end

plot(x,'r')
hold on
plot(y,'b')
plot(z,'g')
hold off
```

E.2 Jacobian file

```
function J = JEBola(rpy)

epsilon = 0.001;

MotAng = IKEBola(rpy, -1, -1);

%%% derivadas respecto a roll

MotAngroll = IKEBola(rpy + [epsilon 0 0], -1, -1);

Drpy1 = (MotAngroll-MotAng)/epsilon;

%%% derivadas respecto a pitch

MotAngpitch = IKEBola(rpy + [0 epsilon 0], -1, -1);

Drpy2 = (MotAngpitch-MotAng)/epsilon;

%%% derivadas respecto a yaw

MotAngyaw = IKEBola(rpy + [0 0 epsilon], -1, -1);

Drpy3 = (MotAngyaw-MotAng)/epsilon;

J = [Drpy1; Drpy2; Drpy3]';

end
```

E.3 Inverse kinematics file

```
function Sol = IKEbola(rpy, flag1, flag2)

% DEFINICIÓN DE CONSTANTES
LONG = 0.8556;
L2 = 0.1723;
RADI = 0.9899;
L1 = 0.4750;
LEG = 1.68;
BRAC = 0.30;
H = 1.6; % Radio de la pelota
R = 0.2;

% CALCULO DE LA POSE DE LA PLATAFORMA
TR = transl(0,0,H)*trotx(rpy(1))*troty(rpy(2))*trotz(rpy(3));

% SISTEMAS DE REFRENCIA ASOCIADOS A LOS EXTREMOS DE LAS PATAS EN
% LA PLATAFORMA MOVIL
TP(:,:,1) = TR*transl(L2, -LONG, 0);
TP(:,:,2) = TR*transl(-L2, -LONG, 0);
TP(:,:,3) = TR*trotz(-2*pi/3)*transl(L2, -LONG, 0);
TP(:,:,4) = TR*trotz(-2*pi/3)*transl(-L2, -LONG, 0);
TP(:,:,5) = TR*trotz(-4*pi/3)*transl(L2, -LONG, 0);
TP(:,:,6) = TR*trotz(-4*pi/3)*transl(-L2, -LONG, 0);

% SISTEMAS DE REFRENCIA ASOCIADOS A LOS EXTREMOS DE LAS PATAS EN
% LA BASE FIJA

TB(:,:,1) = transl(L1, -RADI, 0)*trotz(-2*pi/3);
TB(:,:,2) = transl(-L1, -RADI, 0)*trotz(2*pi/3);
TB(:,:,3) = trotz(-2*pi/3)*transl(L1, -RADI, 0)*trotz(-2*pi/3);
TB(:,:,4) = trotz(-2*pi/3)*transl(-L1, -RADI, 0)*trotz(2*pi/3);
TB(:,:,5) = trotz(-4*pi/3)*transl(L1, -RADI, 0)*trotz(-2*pi/3);
TB(:,:,6) = trotz(-4*pi/3)*transl(-L1, -RADI, 0)*trotz(2*pi/3);

% Coordenades del centro de las articulaciones en la plataforma movil en el
% sistema de referencia de la articulacion de la base.

for i=1:6
    Centers(:,i) = inv(TB(:,:,i))*TP(:,:,i)*[0; 0; 0; 1];
end

% Calculo de las soluciones de la cinemática inversa

for i=1:6
    [phi1, phi2] = SolveAngles(Centers(1,i), Centers(3,i),...
                               BRAC, sqrt(LEG^2 - Centers(2,i)^2));
    if mod(i,2)
        Sol(i) = -phi1;
    else
        Sol(i) = -phi2;
    end
end
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% DIBUJO DE LA PLATAFORMA

if flag1==1
    figure(1);
    trplot(eye(3), 'arrow', 'length', 0.5, 'color', 'k');
    hold on;
    axis([-1.5 1.5 -1.5 1.5 -0.5 2.5]);


    for i=1:6
        if flag2==1
            trplot(TP(:,:,i), 'arrow', 'length', 0.5, 'color', 'r', 'frame', num2str(i));
            trplot(TB(:,:,i), 'arrow', 'length', 0.5, 'color', 'b', 'frame', num2str(i));
        end
        CentersP(:,i) = TP(:,:,i)*[0; 0; 0; 1];
        CentersB(:,i) = TB(:,:,i)*[0; 0; 0; 1];
    end


    fill3(CentersP(1,:),CentersP(2,:),CentersP(3,:), 'g', 'facealpha',.5);
    fill3(CentersB(1,:),CentersB(2,:),CentersB(3,:), 'r', 'facealpha',.5);

    if flag2==1
        samples=30;
        for j=1:6
            for i = 0:1:samples;
                MAT(i+1,:) = TB(:,:,j)*troty(2*pi*i/samples)*transl(BRAC,0,0)*[0; 0; 0; 1];
            end
            plot3(MAT(:,1),MAT(:,2),MAT(:,3),'k','LineWidth',1);
        end
    end

    for i=1:6
        PointArm(:,i) = TB(:,:,i)*troty(Sol(i))*transl(BRAC,0,0)*[0; 0; 0; 1];
        plot3([CentersB(1,i), PointArm(1,i)], ...
            [CentersB(2,i), PointArm(2,i)], ...
            [CentersB(3,i), PointArm(3,i)], 'color' , 'k', 'LineWidth', 2);
        plot3([CentersP(1,i), PointArm(1,i)], ...
            [CentersP(2,i), PointArm(2,i)], ...
            [CentersP(3,i), PointArm(3,i)], 'color' , 'k', 'LineWidth', 2);
    end

    trplot(TR);
    hold off;
end
end
```

E.4 Solve angles file

```
function [phi1, phi2] = SolveAngles(cx,cy,r1,r2)
%UNTITLED6 Summary of this function goes here
%   Detailed explanation goes here

alpha = atan2(cy, cx);
gamma = acos((r1^2 + cx^2 + cy^2 - r2^2)/(2*r1*sqrt(cx^2+cy^2)));
phi1 = alpha + gamma;
phi2 = alpha - gamma;

end
```